TERASOLUNA Server Framework for Java (5.x) Development Guideline Documentation

リリース *5.4.2.RELEASE*

NTT DATA Corporation

2019 年 **06** 月 **07** 日

目次

第 1 章　はじめに　3

1.1　利用規約　3

1.2　参考　4

1.2.1　Macchinetta 利用規約　4

1.3　このドキュメントが示すこと　5

1.4　このドキュメントの対象読者　5

1.5　このドキュメントの構成　5

1.6　このドキュメントの読み方　6

1.7　このドキュメントの動作検証環境　7

1.8　ガイドラインの観点別マッピング　8

1.8.1　セキュリティ対策に関するマッピング　8

OWASP(Open Web Application Security Project) による観点　8

CVE(Common Vulnerabilities and Exposures) による観点　9

1.9　更新履歴　11

第 2 章　アーキテクチャ概要　49

2.1　TERASOLUNA Server Framework for Java (5.x) のスタック　49

2.1.1　TERASOLUNA Server Framework for Java (5.x) の Software Framework 概要　49

2.1.2　Software Framework の主な構成要素　49

DI コンテナ　49

MVC フレームワーク　49

O/R Mapper　50

View　50

セキュリティ　51

バリデーション　51

ロギング　51

共通ライブラリ　51

2.1.3　利用する OSS のバージョン　51

2.1.4　共通ライブラリの構成要素　54

terasoluna-gfw-common　59

terasoluna-gfw-string　59

terasoluna-gfw-codepoints　59

terasoluna-gfw-validator　59

terasoluna-gfw-jodatime　60

terasoluna-gfw-web　60

terasoluna-gfw-web-jsp . . . 61
terasoluna-gfw-security-web . . . 62
2.2 Spring MVC アーキテクチャ概要 . . . 63
2.2.1 Overview of Spring MVC Processing Sequence . . . 63
2.2.2 Implementations of each component . . . 64
Implementation of HandlerMapping . . . 64
Implementation of HandlerAdapter . . . 65
Implementation of ViewResolver . . . 65
Implementation of View . . . 66
2.3 はじめての Spring MVC アプリケーション . . . 68
2.3.1 検証環境 . . . 68
2.3.2 新規プロジェクト作成 . . . 68
2.3.3 サーバーを起動する . . . 76
2.3.4 エコーアプリケーションの作成 . . . 77
フォームオブジェクトの作成 . . . 78
Controller の作成 . . . 79
JSP の作成 . . . 81
入力チェックの実装 . . . 83
まとめ . . . 87
2.4 アプリケーションのレイヤ化 . . . 88
2.4.1 レイヤの定義 . . . 89
アプリケーション層 . . . 89
Controller . . . 89
View . . . 89
Form . . . 90
Helper . . . 91
ドメイン層 . . . 91
Domain Object . . . 91
Repository . . . 92
Service . . . 92
インフラストラクチャ層 . . . 92
RepositoryImpl . . . 93
O/R Mapper . . . 93
Integration System Connector . . . 93
2.4.2 レイヤ間の依存関係 . . . 94
Repository を使用する時の処理の流れ . . . 95
Repository を使用しない時の処理の流れ . . . 97
2.4.3 プロジェクト構成 . . . 98
[projectName]-domain . . . 99
[projectName]-web . . . 100
[projectName]-env . . . 102

第 3 章 アプリケーション開発 105
3.1 Web アプリケーション向け開発プロジェクトの作成 . . . 105

3.1.1 ブランクプロジェクトの種類 . . . 105
3.1.2 開発プロジェクトの作成 . . . 106
3.1.3 開発プロジェクトのカスタマイズ . . . 109
POM ファイルのプロジェクト情報 . . . 109
x.xx.fw.9999 形式のメッセージ ID . . . 111
メッセージ文言 . . . 112
エラー画面 . . . 113
画面フッターの著作権 . . . 114
インメモリデータベース (H2 Database) . . . 115
データソース設定 . . . 116
3.1.4 開発プロジェクトの構成 . . . 119
マルチプロジェクトの構成 . . . 121
web モジュールの構成 . . . 124
domain モジュールの構成 . . . 130
env モジュールの構成 . . . 133
initdb モジュールの構成 . . . 135
selenium モジュールの構成 . . . 136
3.1.5 Appendix . . . 137
プロジェクトの階層構造 . . . 137
アプリケーションコンテキストの構成と Bean 定義ファイルの関係 . . . 140
オフライン環境におけるアプリケーション開発 . . . 142
3.2 ドメイン層の実装 . . . 145
3.2.1 ドメイン層の役割 . . . 145
3.2.2 ドメイン層の開発の流れ . . . 145
3.2.3 Entity の実装 . . . 147
Entity クラスの作成方針 . . . 147
Entity クラスの作成例 . . . 149
テーブル構成 . . . 149
Entity 構成 . . . 152
3.2.4 Repository の実装 . . . 156
Repository の役割 . . . 156
Repository の構成 . . . 157
Repository の作成方針 . . . 159
Repository の作成例 . . . 160
Repository 構成 . . . 160
Repository インタフェースの定義 . . . 160
Repository インタフェースの作成 . . . 160
Repository インタフェースのメソッド定義 . . . 163
RepositoryImpl の作成 . . . 167
3.2.5 Service の実装 . . . 167
Service の役割 . . . 167
Service のクラス構成 . . . 168
Service クラスと SharedService クラスを分ける理由について . . . 171
Service クラスから、別の Service クラスの呼び出しを禁止する理由について 171

メソッドのシグネチャを限定するようなインタフェースや基底クラスについて . . . 172
Service の作成単位 . . . 176
Entity 毎に Service を作成する際の開発イメージ . . . 178
ユースケース毎に作成する際の開発イメージ . . . 179
イベント毎に作成する際の開発イメージ . . . 182
Service クラスの作成 . . . 186
Service クラスの作成方法 . . . 186
Service クラスのメソッドの作成方法 . . . 188
Service クラスのメソッド引数と返り値について . . . 190
SharedService クラスの実装 . . . 191
SharedService クラスの作成方法 . . . 191
SharedService クラスのメソッドの作成方法 . . . 191
SharedService クラスのメソッド引数と返り値について . . . 192
処理の実装 . . . 192
業務データを操作する . . . 192
メッセージを返却する . . . 192
警告メッセージを返却する . . . 193
業務エラーを通知する . . . 194
システムエラーを通知する . . . 195
3.2.6 トランザクション管理について . . . 197
トランザクション管理の方法 . . . 197
宣言型トランザクション管理 . . . 197
「宣言型トランザクション管理」で必要となる情報 . . . 198
トランザクションの伝播 . . . 204
トランザクション管理対象となるメソッドの呼び出し方 . . . 207
トランザクション管理を使うための設定について . . . 208
PlatformTransactionManager の設定 . . . 208
@Transactional を有効化するための設定 . . . 209
<tx:annotation-driven>要素の属性について . . . 210
3.2.7 Tips . . . 211
ビジネスルールの違反をフィールドエラーとして扱う方法 . . . 211
3.3 インフラストラクチャ層の実装 . . . 212
3.3.1 RepositoryImpl の実装 . . . 212
MyBatis3 を使って Repository を実装 . . . 212
JPA を使って Repository を実装 . . . 214
3.4 アプリケーション層の実装 . . . 216
3.4.1 Controller の実装 . . . 216
Controller クラスの作成方法 . . . 218
リクエストとハンドラメソッドのマッピング方法 . . . 218
リクエストパスでマッピング . . . 221
HTTP メソッドでマッピング . . . 222
リクエストパラメータでマッピング . . . 222
リクエストヘッダでマッピング . . . 223

Content-Type ヘッダでマッピング . . . 223
Accept ヘッダでマッピング . . . 223
リクエストとハンドラメソッドのマッピング方針 . . . 223
サンプルアプリケーションの概要 . . . 224
リクエスト URL . . . 225
リクエストとハンドラメソッドのマッピング . . . 227
フォーム表示の実装 . . . 230
入力内容確認表示の実装 . . . 232
フォーム再表示の実装 . . . 235
新規作成の実装 . . . 238
新規作成完了表示の実装 . . . 239
HTML form 上に複数のボタンを配置する場合の実装 . . . 241
サンプルアプリケーションの Controller のソースコード . . . 242
ハンドラメソッドの引数について . . . 243
画面 (View) にデータを渡す . . . 245
URL のパスから値を取得する . . . 247
リクエストパラメータを個別に取得する . . . 249
リクエストパラメータをまとめて取得する . . . 251
入力チェックを行う . . . 253
リダイレクト先にデータを渡す . . . 254
リダイレクト先へリクエストパラメータを渡す . . . 257
リダイレクト先 URL のパスに値を埋め込む . . . 258
Cookie から値を取得する . . . 259
Cookie に値を書き込む . . . 259
ページネーション情報を取得する . . . 260
アップロードファイルを取得する . . . 261
画面に結果メッセージを表示する . . . 261
ハンドラメソッドの返り値について . . . 261
HTML を応答する . . . 262
ダウンロードデータを応答する . . . 263
処理の実装 . . . 265
入力値の相関チェック . . . 266
業務処理の呼び出し . . . 267
ドメインオブジェクトへの値反映 . . . 267
フォームオブジェクトへの値反映 . . . 269
3.4.2 フォームオブジェクトの実装 . . . 271
フォームオブジェクトの作成方法 . . . 272
フィールド単位の数値型変換 . . . 273
フィールド単位の日時型変換 . . . 274
Controller 単位の型変換 . . . 275
入力チェック用のアノテーションの指定 . . . 275
フォームオブジェクトの初期化方法 . . . 276
HTML form へのバインディング方法 . . . 277
リクエストパラメータのバインディング方法 . . . 278

バインディング結果の判定 . . . 279
3.4.3 View の実装 . . . 280
JSP の実装 . . . 280
インクルード用の共通 JSP の作成 . . . 282
モデルに格納されている値を表示する . . . 283
モデルに格納されている数値を表示する . . . 284
モデルに格納されている日時を表示する . . . 285
リクエスト URL を生成する . . . 286
HTML form へフォームオブジェクトをバインドする . . . 289
入力チェックエラーを表示する . . . 290
処理結果のメッセージを表示する . . . 290
コードリストを表示する . . . 291
固定文言を表示する . . . 292
条件によって表示を切り替える . . . 292
コレクションの要素に対して表示処理を繰り返す . . . 294
ページネーション用のリンクを表示する . . . 294
権限によって表示を切り替える . . . 295
JavaScript の実装 . . . 295
スタイルシートの実装 . . . 295
3.4.4 共通処理の実装 . . . 296
Controller の呼び出し前後で行う共通処理の実装 . . . 296
Servlet Filter の実装 . . . 296
HandlerInterceptor の実装 . . . 298
Controller の共通処理の実装 . . . 300
HandlerMethodArgumentResolver の実装 . . . 300
`@ControllerAdvice` の実装 . . . 302
3.4.5 二重送信防止について . . . 309
3.4.6 セッションの使用について . . . 309
3.5 開発プロジェクトのビルド . . . 311
3.5.1 開発プロジェクトのビルド . . . 311
env モジュールの jar ファイルを war ファイルに含めないビルド方法 . . . 313
war ファイルの作成 . . . 313
env モジュールの jar ファイルの作成 . . . 314
env モジュールの jar ファイルを war ファイルに含めるビルド方法 . . . 314
war ファイルの作成 . . . 314
デプロイ . . . 315
Tomcat へのデプロイ . . . 315
Tomcat 以外のアプリケーションサーバへのデプロイ . . . 317
継続的なデプロイ . . . 319

第 4 章 Web アプリ開発機能 325
4.1 入力チェック . . . 325
4.1.1 Overview . . . 325
入力チェックの分類 . . . 325

4.1.2 How to use . . . 326
依存ライブラリの追加 . . . 326
単項目チェック . . . 327
基本的な単項目チェック . . . 327
日時フォーマットのチェック . . . 339
ネストした Bean の単項目チェック . . . 340
バリデーションのグループ化 . . . 352
相関項目チェック . . . 364
Spring Validator による相関項目チェック実装 . . . 364
Bean Validation による相関項目チェック実装 . . . 373
エラーメッセージの定義 . . . 373
ValidationMessages.properties に定義するメッセージ . . . 375
application-messages.properties に定義するメッセージ . . . 379
4.1.3 How to extend . . . 380
既存ルールを組み合わせた Bean Validation アノテーションの作成 . . . 381
コレクション内の値を Bean Validation のアノテーションを使用してチェックする方法 385
Java SE 8 と Hibernate Validator 5.2+ による実装 . . . 386
Java Bean を使った String のラッパークラスによる実装 . . . 389
新規ルールを実装した Bean Validation アノテーションの作成 . . . 394
既存のルールの組み合わせでは表現できないルール . . . 395
相関項目チェックルール . . . 397
業務ロジックチェック . . . 403
Method Validation . . . 405
アプリケーションの設定 . . . 405
Method Validation 対象のメソッドにするための定義方法 . . . 407
制約違反時の例外ハンドリング . . . 412
4.1.4 Appendix . . . 413
Hibernate Validator が用意する入力チェックルール . . . 413
Bean Validation のチェックルール . . . 413
Hibernate Validator のチェックルール . . . 417
Hibernate Validator が用意するデフォルトメッセージ . . . 419
共通ライブラリが用意する入力チェックルール . . . 420
terasoluna-gfw-common のチェックルール . . . 420
terasoluna-gfw-codepoints のチェックルール . . . 421
terasoluna-gfw-validator のチェックルール . . . 421
共通ライブラリのチェックルールの適用方法 . . . 428
共通ライブラリのチェックルールの拡張方法 . . . 429
型のミスマッチ . . . 431
文字列フィールドが未入力の場合に null をバインドする . . . 433
Native to Ascii を行わないメッセージの読み込み . . . 434
OS コマンドインジェクション対策 . . . 435
OS コマンドインジェクションとは . . . 436
対策方法 . . . 436
4.2 例外ハンドリング . . . 438

4.2.1 Overview . . . 438
例外の分類 . . . 438
例外のハンドリング方法 . . . 439
4.2.2 Detail . . . 444
例外の種類 . . . 444
ビジネス例外 . . . 445
正常稼働時に発生するライブラリ例外 . . . 446
システム例外 . . . 447
予期しないシステム例外 . . . 447
致命的なエラー . . . 448
リクエスト不正時に発生するフレームワーク例外 . . . 448
例外ハンドリングのパターン . . . 449
ユースケースの一部やり直し (途中からのやり直し) を促す場合 . . . 451
ユースケースのやり直し (先頭からのやり直し) を促す場合 . . . 451
システム、またはアプリケーションが、正常な状態でない事を通知する場合 452
リクエスト内容が、不正であることを通知する場合 . . . 452
致命的なエラーが発生したことを検知する場合 . . . 453
プレゼンテーション層 (JSP など) で、例外が発生したことを通知する場合 . 455
例外ハンドリングの基本フロー . . . 455
リクエスト単位で Controller クラスがハンドリングする場合の基本フロー . 456
ユースケース単位で Controller クラスがハンドリングする場合の基本フロー 457
サーブレット単位でフレームワークがハンドリングする場合の基本フロー . 458
Web アプリケーション単位でサーブレットコンテナがハンドリングする場合の基本フロー . . . 459
4.2.3 How to use . . . 460
アプリケーションの設定 . . . 460
共通設定 . . . 460
ドメイン層の設定 . . . 466
アプリケーション層の設定 . . . 466
サーブレットコンテナの設定 . . . 472
コーディングポイント（Service 編） . . . 474
ビジネス例外を発生させる . . . 474
システム例外を発生させる . . . 478
例外をキャッチして、処理を継続させる . . . 480
コーディングポイント（Controller 編） . . . 482
リクエスト単位で例外をハンドリングする方法 . . . 482
ユースケース単位で例外をハンドリングする方法 . . . 483
コーディングポイント（JSP 編） . . . 485
MessagesPanelTag を使用して、メッセージを画面表示する方法 . . . 485
システム例外の例外コードを、画面表示する方法 . . . 486
4.2.4 How to use (Ajax) . . . 487
4.2.5 Appendix . . . 487
共通ライブラリから提供している例外ハンドリング用のクラスについて . . . 487
SystemExceptionResolver の設定項目について . . . 493

結果メッセージの属性名 . . . 495
例外コード (メッセージ ID) の属性名 . . . 496
例外コード (メッセージ ID) のヘッダ名 . . . 496
例外オブジェクトの属性名 . . . 497
HTTP レスポンスのキャッシュ制御有無 . . . 498
HandlerExceptionResolverLoggingInterceptor の設定項目について . . . 499
ログ出力対象から除外する例外クラスの一覧 . . . 499
DefaultHandlerExceptionResolver で設定される HTTP レスポンスコードについて . . . 500
4.3 セッション管理 . . . 503
4.3.1 Overview . . . 503
セッションのライフサイクル . . . 506
セッションの生成 . . . 506
セッションへの属性格納 . . . 508
セッションからの属性削除 . . . 510
セッションの破棄 . . . 511
セッションタイムアウト後のリクエスト検知 . . . 513
セッションの利用について . . . 515
セッション利用時のメリットとデメリット . . . 516
セッションを利用しない時のメリットとデメリット . . . 517
セッションに格納するデータについて . . . 517
シリアライズ可能なオブジェクト . . . 518
セッションに格納するデータの容量 . . . 518
AP サーバ多重化時の考慮点について . . . 519
セッションの保存先について . . . 520
4.3.2 How to use . . . 520
`@SessionAttributes` アノテーションの使用 . . . 520
セッションに格納するオブジェクトの指定 . . . 521
セッションにオブジェクトを追加 . . . 523
セッションに格納されているオブジェクトの取得 . . . 525
セッションに格納したオブジェクトの削除 . . . 529
@SessionAttributes を使った処理の実装例 . . . 533
Spring Framework の session スコープの Bean の使用 . . . 533
session スコープの Bean 定義 . . . 533
session スコープの Bean の利用 . . . 536
セッションに格納したオブジェクトの削除 . . . 538
session スコープの Bean を使った処理の実装例 . . . 539
セッション操作のデバッグログ出力 . . . 539
JSP の暗黙オブジェクト `sessionScope` を使用する . . . 539
4.3.3 How to extend . . . 540
同一セッション内のリクエストの同期化 . . . 540
4.3.4 Appendix . . . 541
`@SessionAttributes` アノテーションを使ったウィザード形式の画面遷移の実装例 541
session スコープの Bean を使った複数の Controller を跨いだ画面遷移の実装例 . . . 557
4.4 ページネーション . . . 568

4.4.1 Overview . . . 568
ページ分割時の一覧画面の表示について . . . 568
ページ検索について . . . 570
Spring Data 提供のページ検索機能について . . . 570
ページネーションの表示について . . . 572
取得データの表示について . . . 572
ページネーションリンクの表示について . . . 573
ページネーション情報の表示について . . . 590
ページネーション機能使用時の処理フロー . . . 591
4.4.2 How to use . . . 594
アプリケーションの設定 . . . 594
Spring Data のページネーション機能を有効化するための設定 . . . 594
ページ検索の実装 . . . 595
アプリケーション層の実装 . . . 595
ドメイン層の実装 (MyBatis3 編) . . . 600
ドメイン層の実装 (JPA 編) . . . 600
JSP の実装 (基本編) . . . 601
取得データの表示 . . . 601
ページネーションリンクの表示 . . . 603
ページネーション情報の表示 . . . 608
ページリンクで検索条件を引き継ぐ . . . 611
ページリンクでソート条件を引き継ぐ . . . 613
JSP の実装 (レイアウト変更編) . . . 613
先頭ページと最終ページに移動するリンクの削除 . . . 614
前ページと次ページに移動するリンクの削除 . . . 614
disabled 状態のリンクの削除 . . . 615
指定ページへ移動するリンクの最大表示数の変更 . . . 616
指定ページへ移動するリンクの削除 . . . 617
JSP の実装 (動作編) . . . 618
ソート条件の指定 . . . 618
JavaScript を使用したページリンクの無効化 . . . 619
4.4.3 Appendix . . . 620
PageableHandlerMethodArgumentResolver のプロパティ値について . . . 620
SortHandlerMethodArgumentResolver のプロパティ値について . . . 625
4.5 二重送信防止 . . . 626
4.5.1 Overview . . . 626
Problems . . . 626
更新系ボタンの二重クリック . . . 626
更新処理完了後の画面の再読み込み . . . 628
ブラウザの戻るボタンを使用した不正な画面遷移 . . . 629
Solutions . . . 633
JavaScript によるボタンの 2 度押し防止について . . . 634
PRG(Post-Redirect-Get) パターンについて . . . 635
トランザクショントークンチェックについて . . . 637

トランザクショントークンのネームスペースについて 645
ネームスペースがない場合の問題点について 645
ネームスペース指定時の動作について 647
4.5.2 How to use 648
JavaScript によるボタンの 2 度押し防止の適用 648
PRG(Post-Redirect-Get) パターンの適用 648
トランザクショントークンチェックの適用 653
共通ライブラリから提供しているトランザクショントークンチェックについて 653
`@TransactionTokenCheck` アノテーションの属性について 653
トランザクショントークンの形式について 655
トランザクショントークンのライフサイクルについて 658
トランザクショントークンチェックを使用するための設定 662
トランザクショントークンエラーをハンドリングするための設定 663
トランザクショントークンチェックの Controller での利用方法 664
トランザクショントークンチェックの View(JSP) での利用方法 665
1 つの Controller 内で複数のユースケースを実施する場合 668
ユースケース内にファイルダウンロード処理等の画面を更新しない処理が含まれる場合 671
トランザクショントークンチェックの代表的な適用例 675
セッション使用時の並行処理の排他制御について 677
4.5.3 How to extend 677
トランザクショントークンの上限数の変更方法について 678
4.5.4 Appendix 678
ブラウザキャッシュ無効時のトランザクショントークンチェック 678
グローバルトークン 679
NameSpace ごとに保持できるトランザクショントークンの上限数の変更 . . 679
Controller の実装 679
Quick Reference 684
4.6 メッセージ管理 689
4.6.1 Overview 689
メッセージタイプ 689
パターン別メッセージタイプの分類 690
メッセージ ID 体系 692
タイトル 692
ラベル 693
結果メッセージ 695
入力チェックエラーメッセージ 699
4.6.2 How to use 699
プロパティファイルに設定したメッセージの表示 699
プロパティを使用する際の設定 699
プロパティに設定したメッセージの表示 700
結果メッセージの表示 702
基本的な結果メッセージの使用方法 702

結果メッセージの属性名指定 . . . 709
業務例外メッセージの表示 . . . 711
4.6.3 How to extend . . . 712
独自メッセージタイプを作成する . . . 712
4.6.4 Appendix . . . 714
<t:messagesPanel>タグの属性変更 . . . 714
ResultMessages を使用しない結果メッセージの表示 . . . 717
メッセージキー定数クラスの自動生成ツール . . . 720
4.7 国際化 . . . 724
4.7.1 Overview . . . 724
4.7.2 How to use . . . 725
メッセージ定義の設定 . . . 725
Locale をブラウザの設定により切り替える . . . 727
AcceptHeaderLocaleResolver の設定 . . . 727
メッセージの設定 . . . 728
JSP の実装 . . . 729
Locale を画面操作等で動的に変更する . . . 730
LocaleChangeInterceptor の設定 . . . 732
SessionLocaleResolver の設定 . . . 733
CookieLocaleResolver の設定 . . . 734
メッセージの設定 . . . 735
JSP の実装 . . . 736
国際化が適用されない場合の対処方法 . . . 736
4.8 コードリスト . . . 739
4.8.1 Overview . . . 739
4.8.2 How to use . . . 741
SimpleMapCodeList の使用方法 . . . 741
コードリスト設定例 . . . 741
JSP でのコードリスト使用 . . . 743
Java クラスでのコードリスト使用 . . . 745
NumberRangeCodeList の使用方法 . . . 746
コードリスト設定例 . . . 747
JSP でのコードリスト使用 . . . 748
Java クラスでのコードリスト使用 . . . 749
JdbcCodeList の使用方法 . . . 749
コードリスト設定例 . . . 750
JSP でのコードリスト使用 . . . 754
Java クラスでのコードリスト使用 . . . 755
EnumCodeList の使用方法 . . . 755
コードリスト設定例 . . . 755
JSP でのコードリスト使用 . . . 759
Java クラスでのコードリスト使用 . . . 759
I18nCodeList の使用方法 . . . 759
コードリスト設定例 . . . 760

JSP でのコードリスト使用 . . . 765
Java クラスでのコードリスト使用 . . . 766
特定のコード値からコード名を表示する . . . 767
コードリストを用いたコード値の入力チェック . . . 768
@ExistInCodeList の設定例 . . . 769
4.8.3 How to extend . . . 770
コードリストをリロードする場合 . . . 770
Task Scheduler で実現する方法 . . . 771
Controller(Service) クラスで refresh メソッドを呼び出す方法 . . . 772
コードリストを独自カスタマイズする方法 . . . 775
4.8.4 Appendix . . . 777
SimpleI18nCodeList のコードリスト設定方法 . . . 777
行単位で Locale 毎の java.util.Map(key=コード値, value=ラベル) を設定する . . . 777
列単位でコード値毎の java.util.Map(key=Locale, value=ラベル) を設定する . . . 778
NumberRangeCodeList のバリエーション . . . 780
降順の NumberRangeCodeList の作成 . . . 780
NumberRangeCodeList のインターバルの変更 . . . 781
JSP から直接コードリスト Bean を参照する . . . 783
SimpleI18nCodeList を JSP から直接参照する方法 . . . 784
4.9 ファイルアップロード . . . 787
4.9.1 Overview . . . 787
アップロード処理の基本フロー . . . 788
Spring Web から提供されているクラスについて . . . 791
4.9.2 How to use . . . 793
アプリケーションの設定 . . . 793
Servlet 3.0 のアップロード機能を有効化するための設定 . . . 793
Servlet Filter の設定 . . . 796
例外ハンドリングの設定 . . . 798
単一ファイルのアップロード . . . 801
フォームの実装 . . . 802
JSP の実装 . . . 802
Controller の実装 . . . 803
ファイルアップロードの Bean Validation . . . 807
ファイルが選択されていることを検証するためのバリデーションの実装 . . . 807
ファイルが空でないことを検証するためのバリデーションの実装 . . . 808
ファイルのサイズが許容サイズ内であることを検証するためのバリデーションの実装 . . . 810
フォームの実装 . . . 811
Controller の実装 . . . 811
複数ファイルのアップロード . . . 812
フォームの実装 . . . 812
JSP の実装 . . . 814

Controller の実装 . . . 814
HTML5 の multiple 属性を使った複数ファイルのアップロード . . . 815
フォームの実装 . . . 815
Validator の実装 . . . 816
JSP の実装 . . . 818
Controller の実装 . . . 818
仮アップロード . . . 819
Controller の実装 . . . 822
4.9.3 How to extend . . . 824
仮アップロード時の不要ファイルの Housekeeping . . . 824
不要ファイルを削除するコンポーネントクラスの実装 . . . 824
不要ファイルを削除する処理のスケジューリング設定 . . . 826
4.9.4 Appendix . . . 828
ファイルアップロードに関するセキュリティ問題への考慮 . . . 828
アップロード機能に対する DoS 攻撃 . . . 828
アップロードしたファイルを Web サーバ上で実行する攻撃 . . . 829
ディレクトリトラバーサル攻撃 . . . 829
Commons FileUpload を使用したファイルのアップロード . . . 830
4.10 ファイルダウンロード . . . 834
4.10.1 Overview . . . 834
4.10.2 How to use . . . 835
PDF ファイルのダウンロード . . . 835
カスタム View の実装 . . . 835
ViewResolver の定義 . . . 837
コントローラでの View の指定 . . . 838
Excel ファイルのダウンロード . . . 839
カスタム View の実装 . . . 839
ViewResolver の定義 . . . 841
コントローラでの View の指定 . . . 841
任意のファイルのダウンロード . . . 841
カスタム View の実装 . . . 842
ViewResolver の定義 . . . 843
コントローラでの View の指定 . . . 843
4.11 Tiles による画面レイアウト . . . 845
4.11.1 Overview . . . 845
4.11.2 How to use . . . 846
pom.xml の設定 . . . 846
Spring MVC と Tiles の連携 . . . 847
4.11.3 How to extend . . . 856
複数レイアウトを設定する場合 . . . 856
4.12 共通ライブラリが提供する JSP Tag Library と EL Functions . . . 864
4.12.1 Overview . . . 864
JSP Tag Library . . . 864
EL Functions . . . 864

4.12.2 How to use . . . 866
<t:pagination> . . . 866
<t:messagesPanel> . . . 866
<t:transaction> . . . 867
f:h() . . . 867
f:h() 関数仕様 . . . 867
f:h() 使用方法 . . . 868
f:js() . . . 868
f:js() 関数仕様 . . . 868
f:js() 使用方法 . . . 868
f:hjs() . . . 869
f:hjs() 関数仕様 . . . 869
f:hjs() 使用方法 . . . 869
f:query() . . . 869
f:query() 関数仕様 . . . 870
f:query() 使用方法 . . . 872
f:u() . . . 872
f:u() 関数仕様 . . . 872
f:u() 使用方法 . . . 873
f:link() . . . 873
f:link() 関数仕様 . . . 873
f:link() 使用方法 . . . 874
f:br() . . . 875
f:br() 関数仕様 . . . 875
f:br() 使用方法 . . . 875
f:cut() . . . 876
f:cut() 関数仕様 . . . 876
f:cut() 使用方法 . . . 876
4.13 Ajax . . . 878
4.13.1 Overview . . . 878
4.13.2 How to use . . . 878
アプリケーションの設定 . . . 878
Spring MVC の Ajax 関連の機能を有効化するための設定 . . . 878
Controller の実装 . . . 880
データを取得する . . . 881
フォームデータを POST する . . . 888
フォームデータを JSON として POST する . . . 896
入力エラーのハンドリング . . . 899
BindException のハンドリング . . . 899
MethodArgumentNotValidException のハンドリング . . . 903
HttpMessageNotReadableException のハンドリング . . . 903
BindingResult を使用したハンドリング . . . 904
業務エラーのハンドリング . . . 906
例外ハンドリング用のメソッドで業務例外をハンドリング . . . 907

ハンドラメソッド内で業務例外をハンドリング 907
4.14 ヘルスチェック 909
4.14.1 Overview 909
ロードバランサの負荷分散と縮退運転 909
ヘルスチェックの種類 910
本ガイドラインで示すヘルスチェックの構成 912
4.14.2 How to use 914
Repository インタフェース 914
Service クラス 915
Controller クラス 916
JSP ファイル 917
アクセス権の設定 917
4.14.3 Appendix 918
レスポンスのデータ量を最低限にする設定 918
Apache Tiles の設定を受けないようにする 918
ヘッダファイルやフッタファイルを読み込まないようにする 918
レスポンスから余計な改行を削除する 919

第 5 章 Web Service 921
5.1 RESTful Web Service 921
5.1.1 Overview 921
RESTful Web Service とは 921
RESTful Web Service の開発について 923
RESTful Web Service のモジュールの構成 927
REST API の実装サンプル 929
5.1.2 Architecture 934
Web 上のリソースとして公開 936
URI によるリソースの識別 937
HTTP メソッドによるリソースの操作 938
適切なフォーマットの使用 942
適切な HTTP ステータスコードの使用 943
ステートレスなクライアント/サーバ間の通信 944
関連のあるリソースへのリンク 945
5.1.3 How to design 951
リソースの抽出 951
URI の割り当て 952
REST API であることを示すための URI の割り当て 952
API バージョンを識別するための URI の割り当て 952
リソースを識別するためのパスの割り当て 953
HTTP メソッドの割り当て 954
リソースコレクションの URI に対する HTTP メソッドの割り当て 955
特定リソースの URI に対する HTTP メソッドの割り当て 956
リソースのフォーマット 956
JSON のフィールド名 956

NULL とブランク文字 957
日時のフォーマット 957
ハイパーメディアリンクの形式 958
エラー応答時のフォーマット 958
HTTP ステータスコード 959
リクエストが成功した場合の HTTP ステータスコード 960
リクエストが失敗した原因がクライアント側にある場合の HTTP ステータスコード 961
リクエストが失敗した原因がサーバ側にある場合の HTTP ステータスコード 963
認証・認可 964
リソースの条件付き更新の制御 964
リソースの条件付き取得の制御 964
リソースのキャッシュ制御 965
5.1.4 How to use 965
Web アプリケーションの構成 965
pom.xml の設定 968
アプリケーションの設定 969
RESTful Web Service で必要となる Spring MVC のコンポーネントを有効化するための設定 969
RESTful Web Service 用のサーブレットの設定 974
REST API の実装 976
REST API 用パッケージの作成 981
Resource クラスの作成 982
Controller クラスの作成 988
リソースのコレクションを取得する REST API の実装 990
リソースをコレクションに追加する API REST の実装 999
指定されたリソースを取得する REST API の実装 1001
指定されたリソースを更新する REST API の実装 1004
指定されたリソースを削除する REST API の実装 1007
例外のハンドリングの実装 1009
レスポンス Body にエラー情報を出力するための実装 1011
入力エラー例外のハンドリング実装 1017
リソース未検出エラー例外のハンドリング実装 1024
業務エラー例外のハンドリング実装 1027
排他エラー例外のハンドリング実装 1029
システムエラー例外のハンドリング実装 1030
ExceptionCodeResolver を使ったエラーコードとメッセージの解決 1032
サーブレットコンテナに通知されたエラーのハンドリング実装 1036
エラー応答を行うための Controller の実装 1038
致命的なエラーが発生した際に応答する静的な JSON ファイルの作成 . . . 1040
サーブレットコンテナに通知されたエラーをハンドリングするための設定 . 1040
セキュリティ対策 1042
認証・認可 1042
CSRF 対策 1042

リソースの条件付き操作 1042
リソースのキャッシュ制御 1043
5.1.5 How to extend 1043
@JsonView を使用したレスポンスの出力制御 1043
5.1.6 Appendix 1051
JSR-310 Date and Time API / Joda Time を使う場合の設定 1052
RESTful Web Service とクライアントアプリケーションを同じ Web アプリケーションとして動かす際の設定 1052
RESTful Web Service 用の DispatcherServlet を設ける方法 1052
ハイパーメディアリンクの実装 1055
共通部品の実装 1055
リソース毎の実装 1057
HTTP の仕様に準拠した RESTful Web Service の作成 1060
POST 時の Location ヘッダの設定 1060
リソース毎の実装 1061
CSRF 対策の無効化 1063
XXE 対策の有効化 1064
アプリケーション層のソースコード 1065
MemberRestController.java 1065
ApiErrorCreator.java 1068
ApiGlobalExceptionHandler.java 1069
REST API 実装時に作成したドメイン層のクラスのソースコード 1072
Member.java 1074
MemberCredentia.java 1077
Gender.java 1079
MemberRepository.java 1079
MemberService.java 1080
MemberServiceImpl.java 1081
DomainMessageCodes.java 1084
GenderTypeHandler.java 1084
member-mapping.xml 1086
mybatis-config.xml 1087
MemberRepository.xml 1087
5.2 REST クライアント（HTTP クライアント） 1092
5.2.1 Overview 1092
RestTemplate とは 1092
HttpMessageConverter 1094
ClientHttpRequestFactory 1100
ResponseErrorHandler 1102
ClientHttpRequestInterceptor 1102
5.2.2 How to use 1103
RestTemplate のセットアップ 1103
依存ライブラリ設定 1103
RestTemplate の bean 定義 1104

RestTemplate の利用 1105
GET リクエストの送信 1105
getForObject メソッドを使用した実装 1106
getForEntity メソッドを使用した実装 1106
exchange メソッドを使用した実装 1107
POST リクエストの送信 1108
postForObject メソッドを使用した実装 1109
postForEntity メソッドを使用した実装 1109
exchange メソッドを使用した実装 1110
コレクション形式のデータ取得 1111
リクエストヘッダの設定 1112
Content-Type ヘッダの設定 1112
Accept ヘッダの設定 1113
任意のリクエストヘッダの設定 1113
エラーハンドリング 1114
例外ハンドリング (デフォルトの動作) 1114
ResponseEntity の返却（エラーハンドラの拡張） 1115
通信タイムアウトの設定 1117
SSL 自己署名証明書の使用 1118
Basic 認証 1121
ファイルアップロード (マルチパートリクエスト) 1122
ファイルダウンロード 1123
RESTful な URL(URI テンプレート) を扱う方法と実装例 1126
5.2.3 How to extend 1127
任意の HttpMessageConverter を登録する方法 1127
共通処理の適用（ClientHttpRequestInterceptor） 1128
ロギング処理 1128
Basic 認証用のリクエストヘッダ設定処理 1129
ClientHttpRequestInterceptor の適用 1130
非同期リクエスト 1131
AsyncRestTemplate の bean 定義 1131
非同期リクエストの実装 1132
非同期リクエストの共通処理の実装 1133
5.2.4 Appendix 1136
HTTP Proxy サーバの設定方法 1136
SimpleClientHttpRequestFactory を使用した HTTP Proxy サーバの設定方法 1136
HttpComponentsClientHttpRequestFactory を使用した HTTP Proxy サーバの設定方法 1138
JSON で JSR-310 Date and Time API を使う場合の設定 1143
5.3 SOAP Web Service（サーバ/クライアント） 1143
5.3.1 Overview 1143
SOAP とは 1144
JAX-WS とは 1145
JAX-WS を利用した Web サービスの開発について 1145

Spring Framework の JAX-WS 連携機能について 1146
JAX-WS を利用した Web サービスのモジュールの構成 1149
Web サービスとして公開される URL 1151
5.3.2 How to use 1152
SOAP サーバの作成 1153
プロジェクトの構成 1153
アプリケーションの設定 1154
Web サービスの実装 1156
入力チェックの実装 1162
セキュリティ対策 1164
例外ハンドリングの実装 1167
MTOM を利用した大容量のバイナリデータを扱う方法 1177
クライアントの作成 1180
プロジェクトの構成 1180
Web サービス クライアントの実装 1182
セキュリティ対策 1187
例外ハンドリングの実装 1188
タイムアウトの設定 1189
5.3.3 Appendix 1191
SOAP サーバ用にプロジェクトの設定を変更する 1192
既存プロジェクトの変更 1192
model プロジェクトの作成 1193
webservice プロジェクトの作成 1194
SOAP サーバのパッケージ構成 1195
[server projectName]-domain 1196
[server projectName]-web 1197
[server projectName]-env 1199
[server projectName]-model 1199
[server projectName]-webservice 1200
クライアントのパッケージ構成 1200
[client projectName]-domain 1201
[client projectName]-web 1202
[client projectName]-env 1202
wsimport について 1203
wsimport の使い道 1203
wsimport の使い方 1204
Tomcat 上での Web サービス開発 1206
CXFServlet を使用する場合の設定 1206
POJO 方式で必要な設定 1208
SpringBeanAutowiringSupport を継承する方式で必要な設定 1210

第 6 章 データアクセス 1211
6.1 データベースアクセス（共通編） 1211
6.1.1 Overview 1211

JDBC DataSource について 1211
アプリケーションサーバ提供の JDBC データソース 1212
OSS/Third-Party ライブラリ提供の JDBC データソース 1213
Spring Framework 提供の JDBC データソース 1214
トランザクションの管理方法について 1214
トランザクション境界/属性の宣言について 1214
データの排他制御について 1214
例外ハンドリングについて 1214
複数データソースについて 1216
共通ライブラリから提供しているクラスについて 1216
6.1.2 How to use 1217
データソースの設定 1217
アプリケーションサーバで定義した DataSource を使用する場合の設定 . . . 1217
Bean 定義した DataSource を使用する場合の設定 1218
トランザクション管理を有効化するための設定 1220
JDBC の Debug 用ログの設定 1221
log4jdbc 提供のデータソースの設定 1221
log4jdbc 用ロガーの設定 1222
log4jdbc のオプションの設定 1224
6.1.3 How to extend 1224
動的にデータソースを切り替えるための設定 1224
AbstractRoutingDataSource の実装 1224
データソースの定義 1226
6.1.4 how to solve the problem 1226
N+1 問題の対策方法 1226
JOIN(Join Fetch) を使用して解決する 1228
関連レコードを一括で取得する事で解決する 1229
6.1.5 Appendix 1230
LIKE 検索時のエスケープについて 1230
共通ライブラリのエスケープ仕様について 1231
共通ライブラリから提供しているエスケープ用のメソッドについて 1233
共通ライブラリの使用方法 1235
Sequencer について 1236
共通ライブラリから提供しているクラスについて 1236
共通ライブラリの利用方法 1237
Spring Framework から提供されているデータアクセス例外へ変換するクラス 1239
Spring Framework から提供されている JDBC データソースクラス 1240
6.2 データベースアクセス（MyBatis3 編） 1243
6.2.1 Overview 1243
MyBatis3 について 1243
MyBatis3 のコンポーネント構成について 1244
MyBatis3 と Spring の連携について 1247
MyBatis-Spring のコンポーネント構成について 1248
6.2.2 How to use 1252

pom.xml の設定 1253
MyBatis3 と Spring を連携するための設定 1254
データソースの設定 1254
トランザクション管理の設定 1254
MyBatis-Spring の設定 1256
MyBatis3 の設定 1258
`fetchSize` の設定 1259
SQL 実行モードの設定 1260
TypeAlias の設定 1261
NULL 値と JDBC 型のマッピング設定 1263
TypeHandler の設定 1265
データベースアクセス処理の実装 1269
Repository インタフェースの作成 1269
マッピングファイルの作成 1270
CRUD 処理の実装 1271
検索結果と JavaBean のマッピング方法 1272
検索結果の自動マッピング 1273
検索結果の手動マッピング 1277
Entity の検索処理 1279
単一キーの Entity の取得 1280
複合キーの Entity の取得 1283
Entity の検索 1285
Entity の件数の取得 1289
Entity のページネーション検索 (MyBatis3 標準方式) 1290
Entity のページネーション検索 (SQL 絞り込み方式) 1295
Entity のページネーション検索 (検索結果のソート) 1299
Entity の登録処理 1301
Entity の 1 件登録 1301
キーの生成 1304
Entity の一括登録 1307
Entity の更新処理 1310
Entity の 1 件更新 1310
Entity の一括更新 1314
Entity の削除処理 1317
Entity の 1 件削除 1317
Entity の一括削除 1320
動的 SQL の実装 1321
if 要素の実装 1322
choose 要素の実装 1323
where 要素の実装 1325
set 要素の実装例 1327
foreach 要素の実装例 1328
bind 要素の実装例 1330
LIKE 検索時のエスケープ 1332

SQL Injection 対策 . . . 1333
バインド変数を使って埋め込む方法 . . . 1334
置換変数を使って埋め込む方法 . . . 1335
6.2.3 How to extend . . . 1337
SQL 文の共有 . . . 1337
TypeHandler の実装 . . . 1338
Joda-Time 用の TypeHandler の実装 . . . 1339
ResultHandler の実装 . . . 1340
SQL 実行モードの利用 . . . 1345
PreparedStatement 再利用モードの利用 . . . 1345
バッチモードの利用 . . . 1346
バッチモードの Repository 利用時の注意点 . . . 1352
ストアドプロシージャの実装 . . . 1358
6.2.4 Appendix . . . 1359
Mapper インタフェースの仕組みについて . . . 1359
TypeAlias の設定 . . . 1363
TypeAlias をクラス単位に設定 . . . 1364
デフォルトで付与されるエイリアス名の上書き . . . 1364
データベースによる SQL 切り替えについて . . . 1365
関連 Entity を 1 回の SQL で取得する方法について . . . 1369
テーブルレイアウトとデータ . . . 1371
Entity のクラス図 . . . 1374
Repository インタフェースの実装 . . . 1378
SQL の実装 . . . 1378
マッピングの実装 . . . 1383
関連 Entity をネストした SQL を使用して取得する方法について . . . 1395
関連 Entity をネストした SQL を使用して取得する実装例 . . . 1396
関連 Entity を Lazy Load するための設定 . . . 1397
6.3 データベースアクセス（JPA 編） . . . 1401
6.3.1 Overview . . . 1401
JPA について . . . 1402
JPA の O/R Mapping . . . 1402
JPA の基本用語 . . . 1403
Entity のライフサイクル管理 . . . 1406
Spring Data JPA について . . . 1408
6.3.2 How to use . . . 1411
pom.xml の設定 . . . 1412
アプリケーションの設定 . . . 1412
データソースの設定 . . . 1412
EntityManager の設定 . . . 1412
PlatformTransactionManager の設定 . . . 1417
persistence.xml の設定 . . . 1417
Spring Data JPA を有効化するための設定 . . . 1418
JPA のアノテーションを使用するための設定 . . . 1421

JPA の例外を DataAccessException に変換するための設定 1421
OpenEntityManagerInViewInterceptor の設定 1421
Repository インタフェースの作成 1425
Spring Data 提供のインタフェースを継承する 1426
必要なメソッドのみ定義したインタフェースを継承する 1435
インタフェースの継承は行わない 1436
Query メソッドの追加 1438
Query メソッドを定義する 1438
実行する Query を指定する 1438
Entity のロックを取得する 1440
永続層の Entity を直接操作する 1441
Query ヒントを設定する 1443
Query メソッドの Query 指定 1443
@Query アノテーションで指定する 1444
命名規約ベースのメソッド名で指定する 1449
プロパティファイルに Named query として指定する 1452
Entity の検索処理の実装 1454
条件に一致する Entity を全件検索 1454
条件に一致する Entity のページ検索 1455
Entity の動的条件による検索処理の実装 1457
動的条件に一致する Entity を全件検索 1457
動的条件に一致する Entity をページ検索 1465
Entity の取得処理の実装 1470
ID を指定して Entity を 1 件取得 1470
ID 以外の条件を指定して Entity を 1 件取得 1472
Entity の追加処理の実装 1474
Entity の追加 1474
Entity と関連 Entity の追加 1478
関連 Entity の追加 1481
関連 Entity の直接追加 1482
Entity の更新処理の実装 1484
Entity の更新 1484
関連 Entity の更新 1485
関連 Entity の直接更新 1485
Query メソッドを使用して更新 1486
Entity の削除処理の実装 1486
Entity と関連 Entity の削除 1486
関連 Entity の削除 1487
関連 Entity の直接削除 1490
Query メソッドを使用して削除 1491
LIKE 検索時のエスケープについて 1492
一致方法を Query 側で指定する場合の使用方法 1492
一致方法をロジック側で指定する場合の使用方法 1493
JOIN FETCH について 1495

6.3.3 How to extend . . . 1496
カスタムメソッドの追加方法 . . . 1496
Entity 毎の Repository インタフェースに個別に追加する . . . 1497
すべての Repository インタフェースに一括で追加する . . . 1499
Entity 以外のオブジェクトに Query の取得結果を格納する方法 . . . 1503
Audit 用プロパティの設定方法 . . . 1505
永続層から Entity を取得する JPQL に共通条件を加える方法 . . . 1511
Entity を取得する JPQL に共通条件を加える . . . 1511
関連 Entity を取得する JPQL に共通条件を加える . . . 1514
複数の PersistenceUnit を使用する方法 . . . 1515
Native クエリの使用方法 . . . 1516
6.4 排他制御 . . . 1517
6.4.1 Overview . . . 1517
排他制御の必要性 . . . 1517
Problem1 . . . 1517
Problem2 . . . 1518
Problem3 . . . 1519
トランザクションの分離レベルによる排他制御 . . . 1521
データベースのロック機能による排他制御 . . . 1523
データベースの行ロック機能による排他制御 . . . 1525
楽観ロックによる排他制御 . . . 1528
悲観ロックによる排他制御 . . . 1531
デッドロックの予防 . . . 1535
テーブル内でのデッドロック . . . 1535
テーブル間でのデッドロック . . . 1537
6.4.2 How to use . . . 1538
MyBatis3 使用時の実装方法 . . . 1538
RDBMS の行ロック機能 . . . 1538
楽観ロック . . . 1541
悲観ロック . . . 1546
JPA(Spring Data JPA) 使用時の実装方法 . . . 1546
RDBMS の行ロック機能 . . . 1546
楽観ロック . . . 1548
悲観ロック . . . 1552
排他エラーのハンドリング方法 . . . 1555
楽観ロックの失敗時のエラーハンドリング . . . 1555
悲観ロックの失敗時のエラーハンドリング . . . 1557

第 7 章 アプリケーション形態に依存しない汎用機能 1561
7.1 ロギング . . . 1561
7.1.1 Overview . . . 1561
ログの種類 . . . 1562
ログの出力内容 . . . 1564
ログの出力ポイント . . . 1565

7.1.2 How to use . . . 1568
Logback の設定 . . . 1568
SLF4J の API 呼び出しによる基本的なログ出力 . . . 1574
ログ出力の記述の注意点 . . . 1577
7.1.3 How to extend . . . 1578
ログメッセージの一元管理 . . . 1578
ログメッセージの出力フォーマットの統一 . . . 1585
フレームワークが例外を検知して出力するログのフォーマットに統一 . . . 1585
独自のフォーマットに統一 . . . 1586
7.1.4 Appendix . . . 1588
MDC の使用 . . . 1589
基本的な使用方法 . . . 1589
Filter で MDC に値を Put する . . . 1590
共通ライブラリが提供するログ出力関連機能 . . . 1594
HttpSessionEventLoggingListener . . . 1594
TraceLoggingInterceptor . . . 1595
ExceptionLogger . . . 1596
7.2 プロパティ管理 . . . 1597
7.2.1 Overview . . . 1597
7.2.2 How to use . . . 1598
プロパティファイル定義方法について . . . 1598
bean 定義ファイル内でプロパティを使用する . . . 1600
Java クラス内でプロパティを使用する . . . 1602
7.2.3 How to extend . . . 1604
暗号化したプロパティ値を復号して使用する . . . 1605
7.3 日付操作 (JSR-310 Date and Time API) . . . 1611
7.3.1 Overview . . . 1611
7.3.2 How to use . . . 1611
日時取得 . . . 1613
現在日時で取得 . . . 1613
年月日時分秒を指定して取得 . . . 1614
タイムゾーンを指定する場合の日時取得 . . . 1614
期間 . . . 1616
期間の取得 . . . 1616
型変換 . . . 1616
Date and Time API の各クラスの相互運用性 . . . 1616
java.util.Date との相互運用性 . . . 1618
java.sql パッケージとの相互運用性 . . . 1619
org.terasoluna.gfw.common.date パッケージの利用方法 . . . 1620
文字列へのフォーマット . . . 1620
文字列からのパース . . . 1622
日付操作 . . . 1623
日時の計算 . . . 1623
日時の比較 . . . 1624

日時の判定 . . . 1624
年月日時分秒の取得 . . . 1625
和暦（JapaneseDate） . . . 1626
和暦の取得 . . . 1626
文字列へのフォーマット . . . 1626
文字列からのパース . . . 1627
西暦・和暦の変換 . . . 1627
7.4 日付操作 (Joda Time) . . . 1627
7.4.1 Overview . . . 1627
7.4.2 How to use . . . 1628
日付取得 . . . 1628
現在時刻を取得 . . . 1628
タイムゾーンを指定して現在時刻を取得 . . . 1629
タイムゾーンを指定せず現在時刻を取得 . . . 1629
年月日時分秒を指定して取得 . . . 1630
年月日等の個別取得 . . . 1630
型変換 . . . 1632
java.util.Date との相互運用性 . . . 1632
文字列へのフォーマット . . . 1632
文字列からのパース . . . 1633
日付操作 . . . 1633
日付の計算 . . . 1633
月末月初の取得 . . . 1634
週末週初の取得 . . . 1635
日時の比較 . . . 1636
期間の取得 . . . 1636
Interval . . . 1636
Period . . . 1638
JSP Tag Library . . . 1639
設定方法 . . . 1639
joda:format タグ . . . 1639
応用例 (カレンダーの表示) . . . 1641
7.4.3 Appendix . . . 1645
Java8 未満の和暦操作 . . . 1645
7.5 システム時刻 . . . 1646
7.5.1 Overview . . . 1646
共通ライブラリから提供しているコンポーネントについて . . . 1646
terasoluna-gfw-common . . . 1646
terasoluna-gfw-jodatime . . . 1647
7.5.2 How to use . . . 1649
pom.xml の設定 . . . 1649
サーバーのシステム時刻を返却する . . . 1650
DB から取得した固定の時刻を返却する . . . 1651
サーバーのシステム時刻に DB に登録した差分値を加算した時刻を返却する . . . 1654

差分のキャッシュとリロード方法 1657
7.5.3 Testing 1658
Unit Test 1658
日付によって処理が変わる場合の例 1661
Integration Test 1663
System Test 1664
Production 1665
7.6 文字列処理 1667
7.6.1 Overview 1667
7.6.2 How to use 1667
トリム 1667
パディング・サプレス 1668
サロゲートペアを考慮した文字列処理 1668
文字列長の取得 1668
指定範囲の文字列取得 1669
文字列分割 1670
全角・半角文字列変換 1671
共通ライブラリの適用方法 1671
全角文字列に変換 1671
半角文字列に変換 1672
独自の全角文字と半角文字のペア定義を登録した FullHalfConverter クラスの作成 1673
コードポイント集合チェック (文字種チェック) 1676
共通ライブラリの適用方法 1676
コードポイント集合の作成 1676
コードポイント集合同士の集合演算 1679
コードポイント集合を使った文字列チェック 1681
Bean Validation と連携した文字列チェック 1682
コードポイント集合クラスの新規作成 1683
共通ライブラリから提供しているコードポイント集合クラス 1684
7.7 Bean マッピング (Dozer) 1688
7.7.1 Overview 1688
7.7.2 How to use 1690
Dozer を使用するための Bean 定義 1690
Bean 間のフィールド名、型が同じ場合のマッピング 1690
Bean 間のフィールド名は同じ、型が異なる場合のマッピング 1693
Bean 間のフィールド名が異なる場合のマッピング 1693
単方向・双方向マッピング 1696
Nest したフィールドのマッピング 1698
Collection マッピング 1700
7.7.3 How to extend 1709
カスタムコンバーターの作成 1710
7.7.4 Appendix 1713
フィールド除外設定 (field-exclude) 1713

マッピングの特定化 (map-id) 1715
コピー元の null・空フィールドを除外する設定 (map-null, map-empty) 1717
文字列から日付・時刻オブジェクトへのマッピング 1718
マッピングのエラー 1720

第 8 章 メッセージ連携 1721

8.1 E-mail 送信 (SMTP) 1721
8.1.1 Overview 1721
JavaMail について 1721
Spring Framework の Mail 連携用コンポーネントについて 1722
8.1.2 How to use 1724
依存ライブラリについて 1724
JavaMailSender の設定方法 1725
アプリケーションサーバ提供のメールセッションを使用する場合 1725
アプリケーションサーバ提供のメールセッションを使用しない場合（認証なし） 1727
アプリケーションサーバ提供のメールセッションを使用しない場合（認証あり） 1727
SimpleMailMessage によるメール送信方法 1729
MimeMessage によるメール送信方法 1732
テキストメールの送信 1732
HTML メールの送信 1733
添付ファイル付きメールの送信 1735
インラインリソース付きメールの送信 1737
メール送信時の例外について 1740
8.1.3 How to extend 1740
テンプレートを使用したメール本文の作成方法 1741
FreeMarker を使用したメール本文の作成 1741
8.1.4 Appendix 1744
ISO-2022-JP のエンコードについての考慮 1744
JavaMail でマルチバイト文字を使用する際の注意点 1747
メールヘッダ・インジェクション対策 1747
処理方式 1748
データベースまたはメッセージキューに保持されたメール情報をもとにメール送信を行う 1748
GreenMail を利用したテスト 1749
8.2 JMS(Java Message Service) 1751
8.2.1 Overview 1751
JMS とは 1751
JMS の利用 1754
Spring Framework のコンポーネントを使用した JMS の利用 1755
メッセージを同期送信する場合 1757
メッセージを非同期受信する場合 1758
メッセージを同期受信する場合 1759

プロジェクト構成について 1760
8.2.2 How to use 1761
メッセージの送受信に共通する設定 1762
依存ライブラリの設定 1762
ConnectionFactory の設定 1762
DestinationResolver の設定 1763
メッセージを同期送信する方法 1764
基本的な同期送信 1764
メッセージヘッダを編集して同期送信する場合 1773
トランザクション管理 1774
メッセージを非同期受信する方法 1779
基本的な非同期受信 1779
メッセージのヘッダ情報を取得する 1785
非同期受信後の処理結果をメッセージ送信 1786
非同期受信するメッセージを限定する場合 1788
非同期受信したメッセージの入力チェック 1789
トランザクション管理 1791
非同期受信時の例外ハンドリング 1799
メッセージを同期受信する方法 1803
8.2.3 Appendix 1807
JMS プロバイダに依存する設定 1808
Apache ActiveMQ を利用する場合 1808
同一メッセージの大量送信 1811
サイズの大きなデータの送受信 1812
Apache ActiveMQ を利用する場合 1812

第 9 章 セキュリティ対策 1817
9.1 Spring Security 概要 1817
9.1.1 Spring Security の機能 1817
セキュリティ対策の基本機能 1817
セキュリティ対策の強化機能 1818
9.1.2 Spring Security のアーキテクチャ 1818
Spring Security のモジュール 1818
フレームワークスタックモジュール群 1818
要件に合わせて使用するモジュール群 1819
テスト用のモジュール 1820
要件に合わせて利用する関連モジュール群 1820
フレームワーク処理 1821
FilterChainProxy 1822
HttpFirewall 1822
SecurityFilterChain 1822
Security Filter 1823
9.1.3 Spring Security のセットアップ 1823
依存ライブラリの適用 1824

bean 定義ファイルの作成 . . . 1825
サーブレットフィルタの設定 . . . 1827
セキュリティ対策を適用しないため設定 . . . 1827
9.2 認証 . . . 1829
9.2.1 Overview . . . 1829
認証処理のアーキテクチャ . . . 1829
Authentication Filter . . . 1830
AuthenticationManager . . . 1831
AuthenticationProvider . . . 1832
9.2.2 How to use . . . 1832
フォーム認証 . . . 1832
フォーム認証の適用 . . . 1834
デフォルトの動作 . . . 1835
ログインフォームの作成 . . . 1835
認証成功時のレスポンス . . . 1838
デフォルトの動作 . . . 1838
認証失敗時のレスポンス . . . 1838
デフォルトの動作 . . . 1839
DB 認証 . . . 1839
UserDetails の作成 . . . 1841
UserDetailsService の作成 . . . 1846
DB 認証の適用 . . . 1850
パスワードのハッシュ化 . . . 1851
BCryptPasswordEncoder . . . 1853
認証イベントのハンドリング . . . 1856
認証成功イベント . . . 1858
認証失敗イベント . . . 1858
イベントリスナの作成 . . . 1859
ログアウト . . . 1860
ログアウト処理の適用 . . . 1862
デフォルトの動作 . . . 1863
ログアウト成功時のレスポンス . . . 1864
デフォルトの動作 . . . 1864
認証情報へのアクセス . . . 1864
Java からのアクセス . . . 1864
JSP からのアクセス . . . 1866
認証処理と Spring MVC の連携 . . . 1867
認証情報へのアクセス . . . 1867
9.2.3 How to extend . . . 1869
フォーム認証のカスタマイズ . . . 1869
認証パスの変更 . . . 1869
資格情報を送るリクエストパラメータ名の変更 . . . 1870
認証成功時のレスポンスのカスタマイズ . . . 1870
デフォルト遷移先の変更 . . . 1870

遷移先の固定化 . . . 1871
AuthenticationSuccessHandler の適用 . . . 1871
認証失敗時のレスポンスのカスタマイズ . . . 1872
遷移先の変更 . . . 1872
AuthenticationFailureHandler の適用 . . . 1873
ログアウト処理のカスタマイズ . . . 1876
ログアウトパスの変更 . . . 1876
ログアウト成功時のレスポンスのカスタマイズ . . . 1878
遷移先の変更 . . . 1878
LogoutSuccessHandler の適用 . . . 1878
エラーメッセージのカスタマイズ . . . 1879
システムエラー時のメッセージ . . . 1879
認証時の入力チェック . . . 1880
Bean Validation による入力チェック . . . 1880
認証処理の拡張 . . . 1882
Authentication インターフェースの実装クラスの作成 . . . 1883
AuthenticationProvider インターフェースの実装クラスの作成 . . . 1885
Authentication Filter の作成 . . . 1886
ログインフォームの修正 . . . 1888
拡張した認証処理の適用 . . . 1889
非推奨アルゴリズムの PasswordEncoder の利用 . . . 1894
MessageDigestPasswordEncoder の利用 . . . 1894
9.2.4 Appendix . . . 1896
Spring MVC でリクエストを受けてログインフォームを表示する . . . 1896
Remember Me 認証の利用 . . . 1897
9.3 認可 . . . 1900
9.3.1 Overview . . . 1900
認可処理のアーキテクチャ . . . 1900
ExceptionTranslationFilter . . . 1902
FilterSecurityInterceptor . . . 1902
AccessDecisionManager . . . 1902
AccessDecisionVoter . . . 1903
9.3.2 How to use . . . 1904
アクセスポリシーの記述方法 . . . 1904
Built-In の Common Expressions . . . 1905
Built-In の Web Expressions . . . 1907
演算子の使用 . . . 1907
Web リソースへの認可 . . . 1908
認可処理の適用 . . . 1908
アクセスポリシーの定義 . . . 1909
メソッドへの認可 . . . 1915
AOP の有効化 . . . 1915
認可処理の適用 . . . 1916
アクセスポリシーの定義 . . . 1916

JSP の画面項目への認可 . . . 1917
アクセスポリシーの定義 . . . 1918
Web リソースに指定したアクセスポリシーとの連動 . . . 1918
認可処理の判定結果を変数に格納 . . . 1919
認可エラー時のレスポンス . . . 1920
AccessDeniedHandler . . . 1922
AuthenticationEntryPoint . . . 1922
認可エラー時の遷移先 . . . 1924
9.3.3 How to extend . . . 1924
認可エラー時のレスポンス (認証済みユーザー編) . . . 1925
AccessDeniedHandler の適用 . . . 1925
認可エラー時のレスポンス (未認証ユーザー編) . . . 1926
リクエスト毎に AuthenticationEntryPoint を適用 . . . 1926
ロールの階層化 . . . 1928
階層関係の設定 . . . 1928
Web リソースの認可処理への適用 . . . 1929
メソッドの認可処理への適用 . . . 1930
9.4 セッション管理 . . . 1932
9.4.1 Overview . . . 1932
セッション利用時のセキュリティ対策 . . . 1932
セッションハイジャック攻撃への対策 . . . 1932
セッション固定攻撃への対策 . . . 1934
Spring Security が提供するセッション管理機能 . . . 1934
9.4.2 How to use . . . 1935
セッションハイジャック攻撃への対策 . . . 1935
Spring Security による URL Rewriting 機能の無効化 . . . 1935
サーブレットコンテナによる URL Rewriting 機能の無効化 . . . 1935
セッション管理機能の適用 . . . 1936
セッション固定攻撃への対策 . . . 1937
セッションのライフサイクル制御 . . . 1938
セッションの作成 . . . 1938
セッションの破棄 . . . 1939
セッションタイムアウトの制御 . . . 1940
セッションタイムアウトの指定 . . . 1940
無効なセッションを使ったリクエストの検知 . . . 1940
除外パスの指定 . . . 1941
多重ログインの制御 . . . 1942
セッションのライフサイクル検知の有効化 . . . 1942
多重ログインの禁止 (先勝ち) . . . 1943
多重ログインの禁止 (後勝ち) . . . 1943
9.5 CSRF 対策 . . . 1945
9.5.1 Overview . . . 1945
Spring Security の CSRF 対策 . . . 1946
トークンチェックの対象リクエスト . . . 1947

9.5.2 How to use . . . 1948
CSRF 対策機能の適用 . . . 1948
CSRF トークン値の連携 . . . 1949
Spring MVC を使用した連携 . . . 1949
HTML フォーム使用時の連携 . . . 1951
Ajax 使用時の連携 . . . 1952
トークンチェックエラー時の遷移先の制御 . . . 1953
9.6 ブラウザのセキュリティ対策機能との連携 . . . 1956
9.6.1 Overview . . . 1956
デフォルトでサポートしているセキュリティヘッダ . . . 1956
Cache-Control . . . 1957
X-Frame-Options . . . 1958
X-Content-Type-Options . . . 1958
X-XSS-Protection . . . 1958
Strict-Transport-Security . . . 1959
Content-Security-Policy . . . 1959
Public-Key-Pins . . . 1960
9.6.2 How to use . . . 1961
セキュリティヘッダ出力機能の適用 . . . 1961
セキュリティヘッダの選択 . . . 1961
セキュリティヘッダのオプション指定 . . . 1963
カスタムヘッダの出力 . . . 1964
リクエストパターン毎のセキュリティヘッダの出力 . . . 1964
9.7 XSS 対策 . . . 1967
9.7.1 Overview . . . 1967
Stored, Reflected XSS Attacks . . . 1967
How to use . . . 1967
Output Escaping . . . 1967
出力値をエスケープしない脆弱性のある例 . . . 1968
出力値を f:h() 関数でエスケープする例 . . . 1969
JavaScript Escaping . . . 1971
出力値をエスケープしない脆弱性のある例 . . . 1972
出力値を f:js() 関数でエスケープする例 . . . 1973
Event handler Escaping . . . 1974
出力値をエスケープしない脆弱性のある例 . . . 1974
出力値を f:hjs() 関数でエスケープする例 . . . 1975
9.8 暗号化 . . . 1976
9.8.1 Overview . . . 1976
暗号化方式 . . . 1976
共通鍵暗号化方式 . . . 1976
公開鍵暗号化方式 . . . 1977
ハイブリッド暗号化方式 . . . 1977
暗号化アルゴリズム . . . 1978
DES / 3DES . . . 1978

AES . . . 1978
RSA . . . 1978
DSA / ECDSA . . . 1979
疑似乱数 (生成器) . . . 1979
javax.crypto.Cipher クラス . . . 1979
Spring Security における暗号化機能 . . . 1979
暗号化・復号用のコンポーネント . . . 1979
乱数生成用のコンポーネント . . . 1980
9.8.2 How to use . . . 1981
共通鍵暗号化方式 . . . 1981
文字列の暗号化 . . . 1981
文字列の復号 . . . 1984
バイト配列の暗号化 . . . 1985
バイト配列の復号 . . . 1986
公開鍵暗号化方式 . . . 1987
事前準備（JCA によるキーペアの生成） . . . 1987
暗号化 . . . 1989
復号 . . . 1990
OpenSSL . . . 1990
ハイブリッド暗号化方式 . . . 1994
暗号化 . . . 1994
復号 . . . 1997
乱数生成 . . . 1998
文字列型の疑似乱数生成 . . . 1998
バイト配列型の疑似乱数生成 . . . 1999
9.9 OAuth . . . 2001
9.9.1 Overview . . . 2001
OAuth 2.0 とは . . . 2001
OAuth 2.0 のアーキテクチャ . . . 2003
ロール . . . 2003
スコープ . . . 2005
プロトコルフロー . . . 2005
認可グラント . . . 2007
アクセストークンのライフサイクル . . . 2014
Spring Security OAuth のアーキテクチャ . . . 2018
認可サーバ . . . 2021
リソースサーバ . . . 2027
クライアント . . . 2030
9.9.2 How to use . . . 2032
How to Use の構成 . . . 2032
Spring Security OAuth のセットアップ . . . 2032
認可コードグラントの実装 . . . 2033
認可サーバの実装 . . . 2033
リソースサーバの実装 . . . 2064

クライアントの実装 . . . 2072
インプリシットグラントの実装 . . . 2087
認可サーバの実装 . . . 2087
リソースサーバの実装 . . . 2088
クライアントの実装 . . . 2088
リソースオーナパスワードクレデンシャルグラントの実装 . . . 2105
認可サーバの実装 . . . 2105
リソースサーバの実装 . . . 2108
クライアントの実装 . . . 2108
クライアントクレデンシャルグラントの実装 . . . 2113
認可サーバの実装 . . . 2113
リソースサーバの実装 . . . 2116
クライアントの実装 . . . 2116
9.9.3 How to extend . . . 2118
認可サーバで複数のグラントタイプをサポートする場合 . . . 2118
HTTP アクセスを介した認可サーバとリソースサーバの連携 . . . 2119
認可サーバの設定 . . . 2119
リソースサーバの設定 . . . 2120
リソースサーバへの独自項目連携方法 . . . 2122
認可サーバにおけるパスのカスタマイズ . . . 2128
カスタマイズ可能なパス . . . 2128
パスのカスタマイズ . . . 2130
9.9.4 Appendix . . . 2132
クライアントから認可サーバ、リソースサーバアクセス時に発生するエラー . . . 2132
認可エンドポイントで発生するエラー . . . 2132
トークンエンドポイントで発生するエラー . . . 2135
リソースサーバで発生するエラー . . . 2137
Spring Security OAuth の拡張について . . . 2139
アクセストークンリクエストとそのレスポンスの拡張 . . . 2140
9.10 代表的なセキュリティ要件の実装例 . . . 2147
9.10.1 はじめに . . . 2147
この章で説明すること . . . 2148
対象読者 . . . 2148
9.10.2 アプリケーションの説明 . . . 2149
セキュリティ要件 . . . 2149
機能 . . . 2153
認証・認可に関する仕様 . . . 2153
認証 . . . 2153
認可 . . . 2153
パスワード再発行時の認証 . . . 2153
設計情報 . . . 2154
画面遷移 . . . 2154
URL 一覧 . . . 2156
ER 図 . . . 2158

9.10.3 実装方法とコード解説 2161
パスワード変更の強制・促進 2161
実装する要件一覧 2161
動作イメージ 2161
実装方法 2161
コード解説 2163
パスワードの品質チェック 2175
実装する要件一覧 2175
動作イメージ 2176
実装方法 2176
コード解説 2177
アカウントのロックアウト 2191
実装する要件一覧 2191
動作イメージ 2191
実装方法 2192
コード解説 2195
最終ログイン日時の表示 2209
実装する要件一覧 2209
動作イメージ 2209
実装方法 2209
コード解説 2210
パスワード再発行のための認証情報の生成 2218
実装する要件一覧 2218
動作イメージ 2219
実装方法 2219
コード解説 2220
パスワード再発行のための認証情報の配布 2228
実装する要件一覧 2228
動作イメージ 2229
実装方法 2229
コード解説 2229
パスワード再発行実行時の検査 2234
実装する要件一覧 2234
動作イメージ 2234
実装方法 2235
コード解説 2236
パスワード再発行の失敗上限回数の設定 2245
実装する要件一覧 2245
動作イメージ 2245
実装方法 2245
コード解説 2246
セキュリティ観点での入力値チェック 2252
実装する要件一覧 2252
動作イメージ 2253

実装方法 . . . 2253
コード解説 . . . 2254
監査ログ出力 . . . 2273
実装する要件一覧 . . . 2273
動作イメージ . . . 2274
実装方法 . . . 2276
コード解説 . . . 2276
9.10.4 おわりに . . . 2281
9.10.5 Appendix . . . 2281
Passay . . . 2281
パスワード入力チェック . . . 2282
パスワード生成 . . . 2287

第 10 章 単体テスト 2291
10.1 単体テスト概要 . . . 2291
10.1.1 はじめに . . . 2291
10.1.2 単体テストガイドラインが示すこと . . . 2291
単体テストで利用する OSS ライブラリ構成 . . . 2291
テスティングフレームワーク . . . 2291
アサーション . . . 2291
モック化 . . . 2291
DI コンテナ . . . 2291
MVC フレームワーク . . . 2291
トランザクション管理 . . . 2292
データアクセス . . . 2292
単体テストで利用する OSS ライブラリのバージョン . . . 2292
単体テストの実装 . . . 2293
10.1.3 対象読者 . . . 2293
10.1.4 単体テストの動作検証環境 . . . 2293
10.2 単体テストの実装 . . . 2294
10.2.1 テストの事前準備 . . . 2294
OSS ライブラリの設定 . . . 2294
データベースのセットアップ . . . 2295
スキーマとテストデータのセットアップ（Spring Test 標準機能のみを利用したテストの場合） . . . 2295
テストデータのセットアップ（Spring Test DBUnit を利用したテスト場合） 2298
テスト実装例で使用する設定ファイル . . . 2300
10.2.2 レイヤごとのテスト実装 . . . 2301
インフラストラクチャ層の単体テスト . . . 2302
Repository の単体テスト . . . 2303
ドメイン層の単体テスト . . . 2313
Service の単体テスト . . . 2313
アプリケーション層の単体テスト . . . 2319
アプリケーション層の単体テスト対象 . . . 2319

Controller の単体テスト . . . 2320
Helper の単体テスト . . . 2335
10.2.3 機能ごとのテスト実装 . . . 2335
入力チェックの単体テスト . . . 2335
Bean Validation で実装した Validator の単体テスト . . . 2336
Spring Validator で実装した Validator の単体テスト . . . 2343
10.2.4 単体テストで利用する OSS ライブラリの使い方 . . . 2346
Spring Test . . . 2346
Spring Test とは . . . 2346
MockMvc . . . 2352
MockMvc とは . . . 2352
MockMvc のセットアップ . . . 2355
MockMvc によるテストの実装 . . . 2358
Mockito . . . 2363
Mockito とは . . . 2363
Mockito の利用 . . . 2364
Mockito の機能 . . . 2365
10.3 単体テストの実行 . . . 2371
10.3.1 テストの実行方法 . . . 2371
10.3.2 テストの実行 . . . 2371
IDE 上でテストを実行 . . . 2371
テストクラスの実行 . . . 2371
プロジェクト、メソッド単位で実行 . . . 2374
Maven でテストを実行 . . . 2376
テストフェーズの実行 . . . 2376
コマンドオプションによる任意クラス、メソッドの指定 . . . 2378

第 11 章 チュートリアル 2379
11.1 チュートリアル (Todo アプリケーション) . . . 2379
11.1.1 はじめに . . . 2379
このチュートリアルで学ぶこと . . . 2379
対象読者 . . . 2379
検証環境 . . . 2379
11.1.2 作成するアプリケーションの説明 . . . 2380
アプリケーションの概要 . . . 2380
アプリケーションの業務要件 . . . 2380
アプリケーションの処理仕様 . . . 2380
Show all TODO . . . 2381
Create TODO . . . 2381
Finish TODO . . . 2381
Delete TODO . . . 2381
エラーメッセージ一覧 . . . 2382
11.1.3 環境構築 . . . 2382
プロジェクトの作成 . . . 2382

O/R Mapper に依存しないブランクプロジェクトの作成 2383
MyBatis3 用のブランクプロジェクトの作成 2383
JPA 用のブランクプロジェクトの作成 2383
プロジェクトのインポート 2384
プロジェクトの構成 2387
設定ファイルの確認 2389
プロジェクトの動作確認 2390
11.1.4 Todo アプリケーションの作成 2395
ドメイン層の作成 2396
Domain Object の作成 2396
Repository の作成 2399
RepositoryImpl の作成 (インフラストラクチャ層) 2401
Service の作成 2403
アプリケーション層の作成 2410
Controller の作成 2410
Show all TODO の実装 2411
Create TODO の実装 2418
Finish TODO の実装 2426
Delete TODO の実装 2433
CSS ファイルの使用 2439
11.1.5 データベースアクセスを伴うインフラストラクチャ層の作成 2442
O/R Mapper に依存したブランクプロジェクトの作成 2443
データベースのセットアップ 2443
todo-infra.properties の修正 2443
MyBatis3 を使用したインフラストラクチャ層の作成 2444
TodoRepository の作成 2444
TodoRepositoryImpl の作成 2444
Mapper ファイルの作成 2444
Spring Data JPA を使用したインフラストラクチャ層の作成 2450
Entity の修正 2450
TodoRepository の作成 2452
TodoRepositoryImpl の作成 2453
11.1.6 おわりに 2455
11.1.7 Appendix 2455
設定ファイルの解説 2455
web.xml 2455
インクルード JSP 2461
Bean 定義ファイル 2462
logback.xml 2482
11.2 チュートリアル (Todo アプリケーション REST 編) 2485
11.2.1 はじめに 2485
このチュートリアルで学ぶこと 2485
対象読者 2485
検証環境 2485

11.2.2 環境構築 . . . 2485
DHC のインストール . . . 2485
プロジェクト作成 . . . 2487
11.2.3 REST API の作成 . . . 2488
API 仕様 . . . 2489
GET Todos . . . 2490
POST Todos . . . 2490
GET Todo . . . 2491
PUT Todo . . . 2491
DELETE Todo . . . 2492
エラー応答 . . . 2492
REST API 用の DispatcherServlet を用意 . . . 2493
web.xml の修正 . . . 2493
spring-mvc-rest.xml の作成 . . . 2497
REST API 用の Spring Security の定義追加 . . . 2500
REST API 用パッケージの作成 . . . 2502
Resource クラスの作成 . . . 2503
Controller クラスの作成 . . . 2505
GET Todos の実装 . . . 2506
POST Todos の実装 . . . 2508
GET Todo の実装 . . . 2513
PUT Todo の実装 . . . 2519
DELETE Todo の実装 . . . 2522
例外ハンドリングの実装 . . . 2526
ドメイン層の実装を変更 . . . 2526
エラーメッセージの定義 . . . 2530
エラーハンドリング用のクラスを格納するパッケージの作成 . . . 2532
REST API のエラーハンドリングを行うクラスの作成 . . . 2532
REST API のエラー情報を保持する JavaBean の作成 . . . 2533
HTTP レスポンス BODY にエラー情報を出力するための実装 . . . 2534
入力エラーのエラーハンドリングの実装 . . . 2536
業務例外のエラーハンドリングの実装 . . . 2539
リソース未検出例外のエラーハンドリングの実装 . . . 2541
システム例外のエラーハンドリングの実装 . . . 2544
11.2.4 おわりに . . . 2549
11.3 セッションチュートリアル . . . 2550
11.3.1 始めに . . . 2550
学習の流れ . . . 2550
このチュートリアルで学ぶこと . . . 2550
対象読者 . . . 2550
検証環境 . . . 2550
11.3.2 アプリケーションの概要と要件 . . . 2550
概要 . . . 2550
要件 . . . 2551

機能要件 . . . 2551
非機能要件 . . . 2552
基盤構成 . . . 2553
11.3.3 アプリケーションの設計 . . . 2554
画面定義 . . . 2554
URL の抽出 . . . 2555
入出力データの設計 . . . 2558
データの抽出 . . . 2558
ライフサイクルの定義 . . . 2558
セッション利用有無の判断 . . . 2560
セッション中のデータを利用するための実装方法 . . . 2564
セッションを利用する際の考慮事項 . . . 2565
セッションの同期化 . . . 2565
セッションタイムアウト . . . 2565
アプリケーション設計の全体 . . . 2566
11.3.4 プロジェクトの構成 . . . 2566
プロジェクトの作成 . . . 2566
プロジェクトの構成 . . . 2567
動作確認 . . . 2570
11.3.5 簡易 EC サイトアプリケーションの作成 . . . 2571
アカウント情報変更機能を作成する . . . 2571
フォームオブジェクトの作成 . . . 2572
Controller の作成 . . . 2576
JSP の作成 . . . 2579
動作確認 . . . 2584
カートアイテム登録機能を作成する . . . 2585
セッションスコープ Bean を定義 . . . 2585
フォームオブジェクトの作成 . . . 2586
Controller の作成 . . . 2587
JSP の作成 . . . 2589
動作確認 . . . 2593
商品検索情報を保持する仕組みを作成する . . . 2593
セッションスコープ Bean を作成 . . . 2593
Controller の修正 . . . 2595
動作確認 . . . 2598
カートアイテム削除機能を作成する . . . 2598
フォームオブジェクトの作成 . . . 2599
Controller の作成 . . . 2599
JSP の作成 . . . 2601
動作確認 . . . 2602
商品注文機能を作成する . . . 2602
Controller の作成 . . . 2603
JSP の作成 . . . 2605
動作確認 . . . 2609

セッションの同期化とタイムアウトの設定 . . . 2609
11.3.6 終わりに . . . 2610
11.4 Spring Security チュートリアル . . . 2611
11.4.1 はじめに . . . 2611
このチュートリアルで学ぶこと . . . 2611
対象読者 . . . 2611
検証環境 . . . 2611
11.4.2 作成するアプリケーションの概要 . . . 2611
11.4.3 環境構築 . . . 2612
プロジェクトの作成 . . . 2612
11.4.4 アプリケーションの作成 . . . 2613
ドメイン層の実装 . . . 2613
Domain Object の作成 . . . 2614
AccountRepository の作成 . . . 2615
AccountSharedService の作成 . . . 2616
認証サービスの作成 . . . 2618
データベースの初期化スクリプトの設定 . . . 2621
ドメイン層の作成後のパッケージエクスプローラー . . . 2623
アプリケーション層の実装 . . . 2624
Spring Security の設定 . . . 2624
ログインページを返す Controller の作成 . . . 2628
ログインページの作成 . . . 2628
JSP からログインユーザーのアカウント情報へアクセス . . . 2631
ログアウトボタンの追加 . . . 2632
Controller からログインユーザーのアカウント情報へアクセス . . . 2634
アプリケーション層の作成後のパッケージエクスプローラー . . . 2636
11.4.5 おわりに . . . 2636
11.4.6 Appendix . . . 2637
設定ファイルの解説 . . . 2637
spring-security.xml . . . 2638
spring-mvc.xml . . . 2640

第 12 章 Appendix(Know How) 2645
12.1 NEXUS による Maven リポジトリの管理 . . . 2645
12.1.1 Why NEXUS ? . . . 2645
12.1.2 Install and Start up . . . 2645
12.1.3 settings.xml . . . 2646
12.1.4 mvn deploy how to . . . 2647
12.1.5 pom.xml . . . 2648
12.1.6 Upload 3rd party artifact (ex. ojdbc6.jar) . . . 2649
use artifact . . . 2651
12.2 ボイラープレートコードの排除 (Lombok) . . . 2652
12.2.1 Lombok とは . . . 2652
12.2.2 Lombok の効果 . . . 2652

12.2.3 Lombok のセットアップ 2654
依存ライブラリの追加 2654
IDE 連携 2655
Lombok のダウンロード 2655
Lombok のインストール 2656
12.2.4 Lombok の使用方法 2657
Lombok が提供しているアノテーション 2657
JavaBean の作成 2658
toString の対象から特定のフィールドを除外する方法 2659
equals と hashCode の対象から特定のフィールドを除外する方法 2660
フィールド初期化用のコンストラクタを生成する方法 2662
ロガーインスタンスの作成 2664
12.3 参考書籍 2666
12.4 Spring Framework 理解度チェックテスト 2667

注釈: 内容の誤りやコメントは Github の Issues にご登録お願いします。

第 1 章

はじめに

1.1 利用規約

本ドキュメントを使用するにあたり、以下の規約に同意していただく必要があります。同意いただけない場合は、本ドキュメント及びその複製物の全てを直ちに消去又は破棄してください。

1. 本ドキュメントの著作権及びその他一切の権利は、NTT データあるいは NTT データに権利を許諾する第三者に帰属します。
2. 本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、および NTT データの著作権表示を削除することはできません。
3. 本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「参考文献：TERASOLUNA Server Framework for Java (5.x) Development Guideline」あるいは同等の表現を、作成したドキュメント及びその複製物に記載するものとします。
4. 前 2 項によって作成したドキュメント及びその複製物を、無償の場合に限り、第三者へ提供することができます。
5. NTT データの書面による承諾を得ることなく、本規約に定められる条件を超えて、本ドキュメント及びその複製物を使用したり、本規約上の権利の全部又は一部を第三者に譲渡したりすることはできません。
6. NTT データは、本ドキュメントの内容の正確性、使用目的への適合性の保証、使用結果についての的確性や信頼性の保証、及び瑕疵担保義務も含め、直接、間接に被ったいかなる損害に対しても一切の責任を負いません。
7. NTT データは、本ドキュメントが第三者の著作権、その他如何なる権利も侵害しないことを保証しません。また、著作権、その他の権利侵害を直接又は間接の原因としてなされる如何なる請求（第三者との間の紛争を理由になされる請求を含む。）に関しても、NTT データは一切の責任を負いません。

本ドキュメントは Macchinetta を参考に作成しています「参考文献：Macchinetta Server Framework Development Guideline」 。

本ドキュメントで使用されている各社の会社名及びサービス名、商品名に関する登録商標および商標は、以下の通りです。

- TERASOLUNA は、株式会社 NTT データの登録商標です。
- Macchinetta は、NTT の登録商標です。
- その他の会社名、製品名は、各社の登録商標または商標です。

1.2 参考

1.2.1 Macchinetta 利用規約

本ドキュメントを使用するにあたり、以下の規約に同意していただく必要があります。同意いただけない場合は、本ドキュメント及びその複製物の全てを直ちに消去又は破棄してください。

1. 本ドキュメントの著作権及びその他一切の権利は、日本電信電話株式会社（以下「NTT」とする）あるいは NTT に権利を許諾する第三者に帰属します。
2. 本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、および NTT の著作権表示を削除することはできません。
3. 本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「参考文献：Macchinetta Server Framework Development Guideline」あるいは同等の表現を、作成したドキュメント及びその複製物に記載するものとします。
4. 前 2 項によって作成したドキュメント及びその複製物を、無償の場合に限り、第三者へ提供することができます。
5. NTT の書面による承諾を得ることなく、本規約に定められる条件を超えて、本ドキュメント及びその複製物を使用したり、本規約上の権利の全部又は一部を第三者に譲渡したりすることはできません。
6. NTT は、本ドキュメントの内容の正確性、使用目的への適合性の保証、使用結果についての的確性や信頼性の保証、及び瑕疵担保義務も含め、直接、間接に被ったいかなる損害に対しても一切の責任を負いません。
7. NTT は、本ドキュメントが第三者の著作権、その他如何なる権利も侵害しないことを保証しません。また、著作権、その他の権利侵害を直接又は間接の原因としてなされる如何なる請求（第三者との間の紛争を理由になされる請求を含む。）に関しても、NTT は一切の責任を負いません。

本ドキュメントで使用されている各社の会社名及びサービス名、商品名に関する登録商標および商標は、以下の通りです。

- Macchinetta は、NTT の登録商標です。
- TERASOLUNA は、株式会社 NTT データの登録商標です。
- その他の会社名、製品名は、各社の登録商標または商標です。

1.3 このドキュメントが示すこと

本ガイドラインでは Spring、Spring MVC や JPA、MyBatis を中心としたフルスタックフレームワークを利用して、保守性の高い Web アプリケーション開発をするためのベストプラクティスを提供する。

本ガイドラインを読むことで、ソフトウェア開発 (主にコーディング) が円滑に進むことを期待する。

1.4 このドキュメントの対象読者

本ガイドラインはソフトウェア開発経験のあるアーキテクトやプログラマ向けに書かれており、以下の知識があることを前提としている。

- Spring Framework の DI や AOP に関する基礎的な知識がある
- Servlet/JSP を使用して Web アプリケーションを開発したことがある
- SQL に関する知識がある
- Maven を使用して Web アプリケーションをビルドしたことがある

これから Java を勉強し始めるという人向けではない。

Spring Framework に関して、本ドキュメントを読むための基礎知識があるかどうかを測るために *Spring Framework 理解度チェックテスト*を参照されたい。この理解度テストが 4 割回答できない場合は、別途以下のような入門書籍で学習することを推奨する。

- Spring 徹底入門 (翔泳社) [日本語]
- Spring3 入門—Java フレームワーク・より良い設計とアーキテクチャ (技術評論社) [日本語]
- Pro Spring 4th Edition (Apress)

1.5 このドキュメントの構成

- アーキテクチャ概要　Spring MVC の概要や、TERASOLUNA Server Framework for Java (5.x) の基本的な考え方を説明する。
- アプリケーション開発　TERASOLUNA Server Framework for Java (5.x) を利用してアプリケーション開発する上で必ず押さえておかなくてはならない知識や作法について説明する。
- *Web アプリ開発機能*　Web アプリケーション開発で必要となる機能をどう実装するか、何に気を付けるべきかを説明する。
- *Web Service*　Web サービス開発で必要となる機能をどう実装するか、何に気を付けるべきかを説明する。
- データアクセス　データアクセス機能をどう実装するか、何に気を付けるべきかを説明する。
- アプリケーション形態に依存しない汎用機能　アプリケーション形態に依存しない汎用機能をどう実装するか、何に気を付けるべきかを説明する。
- メッセージ連携　メッセージ連携機能をどう実装するか、何に気を付けるべきかを説明する。

- セキュリティ対策　Spring Security を中心としたセキュリティ対策について説明する。
- 単体テスト　Spring Test を利用した単体テストについて説明する。
- チュートリアル　簡単なアプリケーション開発を通して、TERASOLUNA Server Framework for Java (5.x) によるアプリケーション開発を体験する。
- *Appendix(Know How)*　TERASOLUNA Server Framework for Java (5.x) を利用する場合の付加情報を説明する。

1.6 このドキュメントの読み方

まずは」アーキテクチャ概要」から読み進めていただきたい。特に Spring MVC の経験がない場合は」はじめての *Spring MVC* アプリケーション」を実施すること。「アプリケーションのレイヤ化」は本ガイドラインで共通する用語と概念の説明を行っているため、必ず一読されたい。

次に」チュートリアル」に進む。このチュートリアルでは」習うより慣れろ」を目的として、詳細な説明の前にまず手を動かして、TERASOLUNA Server Framework for Java (5.x) によるアプリケーション開発を体感していただきたい。

チュートリアルを実践したのちに、」アプリケーション開発」でアプリケーション開発の詳細を学ぶ。特に」アプリケーション層の実装」で Spring MVC による開発のノウハウを凝集して説明しているため、何度も読み返すことを推奨する。本章を読み終えた後にもう一度」チュートリアル」を振り返るとより理解が深まる。

ここまでは TERASOLUNA Server Framework for Java (5.x) を使用するすべての開発者が読むことを強く推奨する。

「*Web* アプリ開発機能」、」*Web Service*」、」データアクセス」、」アプリケーション形態に依存しない汎用機能」、」メッセージ連携」、」セキュリティ対策」については目的に応じて必要なタイミングで参照すればよい。ただし、」入力チェック」はアプリケーション開発で通常は必要となるため、基本的には読んでおくこと。

テクニカルリーダーはこれらをすべて読み内容を把握した上でプロジェクトにおいて、どのような方針を定めるか検討していただきたい。

注釈: 時間がない場合、まずは

1. はじめての *Spring MVC* アプリケーション
2. アプリケーションのレイヤ化
3. チュートリアル *(Todo* アプリケーション*)*
4. アプリケーション開発
5. チュートリアル *(Todo* アプリケーション*)*
6. 入力チェック

を読むとよい。

1.7 このドキュメントの動作検証環境

本ガイドラインで説明している内容の動作検証環境については、「テスト済み環境」を参照されたい。

1.8 ガイドラインの観点別マッピング

ガイドラインの 4 章以降は機能別に説明されているが、機能面とは別の観点で、どの節にどの内容が含まれているかのマッピングを示す。

1.8.1 セキュリティ対策に関するマッピング

OWASP(Open Web Application Security Project) による観点

OWASP Top 10 for 2017 を軸として、セキュリティに関連するガイドライン内の説明へのリンクを記載する。

項番	項目名	ガイドライン内の関連箇所
A1:2017	Injection SQL Injection	• *SQL Injection* 対策 • *Named Parameters* について
A1:2017	Injection OS Command Injection	• *OS* コマンドインジェクション対策
A1:2017	Injection Email Header Injection	• メールヘッダ・インジェクション対策
A1:2017	Injection	• セキュリティ観点での入力値チェック
A2:2017	Broken Authentication	• セッションハイジャック攻撃への対策 • セッション固定攻撃への対策 • パスワードのハッシュ化
A3:2017	Sensitive Data Exposure	• 暗号化したプロパティ値を復号して使用する • 暗号化 • パスワードのハッシュ化
A4:2017	XML External Entities (XXE)	• *iText 2.1.7* に存在する *XXE (XML External Entity)* 脆弱性について • *XXE(XML External Entity)* 対策について • *XXE* 対策の有効化
A5:2017	Broken Access Control	• ディレクトリトラバーサル攻撃

Continued on next page

表 1.1 – continued from previous page

項番	項目名	ガイドライン内の関連箇所
A6:2017	Security Misconfiguration	• ログの出力内容 • システム例外の例外コードを、画面表示する方法 • 認可エラー時の遷移先
A7:2017	Cross-Site Scripting (XSS)	• *XSS* 対策 • *X-XSS-Protection*
A8:2017	Insecure Deserialization	• デシリアライズ時の注意点 • フォームデータを *POST* する • *Resource* クラスの作成
A9:2017	Using Components with Known Vulnerabilities	• 特に言及なし
A10:2017	Insufficient Logging & Monitoring	• イベントリスナの作成 • 監査ログ出力

CVE(Common Vulnerabilities and Exposures) による観点

ガイドラインで言及している CVE ごとにその説明とガイドラインへのリンクを記載する。ガイドラインで言及していない CVE については、Pivotal Product Vulnerability Reports を参照されたい。

CVE	概要	ガイドラインでの言及箇所
CVE-2014-0050 CVE-2016-3092	Apache Commons FileUpload を使用するとファイルをアップロードする処理で細工されたリクエストによる DoS 攻撃を受ける可能性がある	• *Overview* • *Commons FileUpload* を使用したファイルのアップロード
CVE-2014-1904	`<form:form>`タグの `action` 属性を省略すると XSS 攻撃を受ける可能性がある	• *JSP* の実装
CVE-2015-3192	DTD を使用した DoS 攻撃が可能となる	• *How to use* • アプリケーションの設定
CVE-2016-5007	Spring Security と Spring MVC のパス比較方法の差異を利用して認可のすり抜けが可能となる	• アクセスポリシーを適用する *Web* リソースの指定
CVE-2019-3778	認可コードグラントを利用した認可サーバにおけるオープンリダイレクト脆弱性	• *Spring Security OAuth* のセットアップ

1.9 更新履歴

更新日付	更新箇所	更新内容
2019-05-31	-	5.4.2 RELEASE 版公開
	全般	ガイドラインの誤記 (タイプミスや単純な記述ミスなど) の修正 記載内容の改善 記載内容の修正・追加 • ViewResolver の定義について、Spring 4.0 以前からの<bean>要素を使用した定義方法を削除し、Spring 4.1 以降の<mvc:view-resolvers>要素を使用した定義方法のみ解説するよう変更 (管理 ID#3197) • 利用するミドルウェアのバージョンを更新（管理 ID#3839） Spring Framework 4.3.23 より XML スキーマ処理が改善されたため、ブランクプロジェクトにおける Bean 定義ファイルの XML スキーマファイル (.xsd) 参照を http から https に変更 (管理 ID#3920)
	ガイドラインの観点別マッピング	OWASP Top 10 を 2013 版から 2017 版へ変更 • OWASP(Open Web Application Security Project) による観点の更新 (管理 ID#3585)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	TERASOLUNA Server Framework for Java (5.x) のスタック	利用する OSS のバージョンを更新 • Spring IO Platform のバージョンを Brussels-SR17 に更新 (管理 ID#3908) • Spring Framework のバージョンを 4.3.23 に更新 • Spring Security のバージョンを 4.2.12 に更新 • Spring Data のバージョンを 1.13.20 に更新 • Spring Security OAuth を 2.0.17 に更新 • Hibernate Validator のバージョンを 5.3.6.Final に更新 • Jackson のバージョンを 2.8.11 に更新 • MyBatis のバージョンを 3.4.6 に更新 (管理 ID#3908) • MyBatis-spring のバージョンを 1.3.2 に更新 (管理 ID#3908) 利用する OSS のバージョンの更新による主な変更 • Spring Security 4.2.5 で変更となったセキュリティヘッダの付与タイミングによる、リクエストパスのマッチングにおける注意事項の追加 (管理 ID#3601) • Spring Security OAuth 2.0.17 よりリダイレクト URI のホワイトリストチェックの仕様が変更されたことへの対応 (管理 ID#3590) • Lombok 1.16.22 より`@Data`と`@NoArgsConstructor`を付与する順序によってコンパイルエラーが発生するようになったことへの対応 (管理 ID#3613)

	TERASOLUNA Server Framework for Java (5.x) のスタック	共通ライブラリの新機能追加
		`terasoluna-gfw-validator`
		• バイト長チェック用 Bean Validation 制約アノテーション `@ByteSize` (管理 ID#3853)
		共通ライブラリの機能改善
		`terasoluna-gfw-common`
		• `SimpleReloadableI18nCodeList` の追加 (管理 ID#3862)
		• `@ExistInCodeList` で `Number` 型をサポートするよう改善 (管理 ID#2998)
		• `ReloadableCodeList` のイミュータブル対応に伴う `CodeListInterceptor` の仕様変更 (管理 ID#3861)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	Web アプリケーション向け開発プロジェクトの作成	記載内容の追加 • 大量にコードリストを定義する場合の Bean 定義方法に関する記載を追加 (管理 ID#689)
	アプリケーション層の実装	記載内容の追加 • Spring Framework 4.3 より追加された `@RequestMapping` の合成アノテーションの説明を追加 (管理 ID#3802)
	入力チェック	記載内容の追加 • Hibernate Validator で提供される `TimeProvider` を実装することで、基準日付の変更が可能である旨の説明を追加 (管理 ID#3767)
	例外ハンドリング	記載内容の修正 • `DefaultHandlerExceptionResolver` がハンドリングする例外一覧に Spring Framework 4.2 より追加された `org.springframework.web.bind.MissingPathVariableException` を追加 (管理 ID#3737) • `SystemExceptionResolver#preventResponseCaching` と Spring Security の Cache-Control ヘッダの併用についての注意を追加 (管理 ID#3601)
	ページネーション	構成見直し • Overview を取得データの表示、ページネーションリンクの表示、ページネーション情報の表示の 3 点について説明するように変更 (管理 ID#3218)
	メッセージ管理	記載内容の修正 • `SPRING_SECURITY_LAST_EXCEPTION` が格納されるスコープの誤記を修正 (管理 ID#2994)
	国際化	記載内容の追加 • `AcceptHeaderLocaleResolver` と `LocaleChangeInterceptor` の指定可能な設定についての説明を追加 (管理 ID#3658)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	コードリスト	記載内容の修正 • 独自カスタマイズしたコードリストの Bean 定義方法を、コンポーネントスキャンから Bean 定義ファイルによる定義に変更 (管理 ID#3175) 記載内容の追加 • コードリスト Bean を JSP から直接参照する方法を追加 (管理 ID#3246)
	ファイルダウンロード	記載内容の追加 • CVE-2017-9096 に関する説明及び注意点を追加 (管理 ID#3583)
	Ajax	OWASP Top 10 2017 対応に伴う修正 • A8:2017 に関連する、デシリアライズ時の Warning を追加 (管理 ID#3585) • TERASOLUNA Server Framework for Java (5.x) では XXE 対策済みの Spring MVC を使用しているため、XXE 対策についての Warning を Note へ変更し、spring-oxm による対策方法の記述を削除 (管理 ID#3585) 記述内容の修正 • 存在しないプロパティの記述（5.0.0 リリース時の修正漏れ）を削除 (管理 ID#3315)
	RESTful Web Service	OWASP Top 10 2017 対応に伴う修正 • TERASOLUNA Server Framework for Java (5.x) では XXE 対策済みの Spring MVC を使用しているため、XXE 対策についての Warning を Note へ変更し、spring-oxm による対策方法の記述を削除 (管理 ID#3585) 記載内容の追加 • Spring Framework 4.3 より追加された `@RequestMapping` の合成アノテーションの説明を追加 (管理 ID#3802) 記述内容の修正 • 存在しないプロパティの記述（5.0.0 リリース時の修正漏れ）を削除 (管理 ID#3315) • Dozer のカスタムコンバーターに関する記述を *Bean マッピング (Dozer)* に統合 (管理 ID#3804)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	データベースアクセス（*MyBatis3* 編）	記載内容の追加 • `Pageable` を利用した検索結果のソートについての説明を追加 (管理 ID#3681) • JSR-310 Date and Time API を使う場合の設定の記事を削除し、依存ライブラリとして別途 `mybatis-typehandlers-jsr310` を追加する必要はなくなった旨の Note を追加 (管理 ID#3376)
	データベースアクセス（*JPA* 編）	記載内容の追加 • EntityManager の生存期間外に Entity を引き継がないよう注意喚起を WARNING に追加 (管理 ID#3557) • Hibernate 5.2.10 以前のバージョンに存在する、Query(JPQL) 解釈のバグについての WARNING を追加 (管理 ID#3791)
	ロギング	記載内容の修正 • 共通ライブラリが提供する `TraceLoggingInterceptor` の WARN ログ出力に関する閾値の設定例を修正 (管理 ID#3728)
	Bean マッピング (Dozer)	記載内容の追加 • Dozer 6.2.0 より前のバージョンに存在する、単方向マッピングを定義したにも関わらず逆方向にマッピングされる場合があるバグについての Warning を追加 (管理 ID#3656) 記載内容の削除 • 現バージョン（Dozer5.5.0 以降）では Collection<T>を使用した Bean 間のマッピングも可能であるため、マッピングが失敗する旨を記述した Todo を削除 (管理 ID#3618)
	JMS(Java Message Service)	OWASP Top 10 2017 対応に伴う修正 • A8:2017 に関連する、デシリアライズ時の Warning を追加 (管理 ID#3585) 記載内容の修正・追加 • JMS を利用する際の Bean 定義の記載場所を再整理 (管理 ID#3383) • JNDI を使用しない場合の `DynamicDestinationResolver` の Bean 定義方法に関する記載を追加 (管理 ID#3383)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	認証	OWASP Top 10 2017 対応に伴う修正 • A10:2017 に関連する、ログイン認証時のログについての Tip を追加 (管理 ID#3585) 記載内容の修正 • `PasswordEncoder` インターフェースの実装クラス一覧を推奨するクラスのみ記載するように変更 (管理 ID#3564) • 非推奨の `PasswordEncoder` について、`ShaPasswordEncoder` から `MessageDigestPasswordEncoder` を使用する方法に記載を変更 (管理 ID#3564) 記載内容の追加 • `SPRING_SECURITY_LAST_EXCEPTION` が格納されるスコープの誤記を修正 (管理 ID#2994)
	CSRF 対策	OWASP Top 10 2017 対応に伴う修正 • OWASP Top 10 2013 版へのリンクを OWASP Cheat Sheet へのリンクへ変更 (管理 ID#3585)
	ブラウザのセキュリティ対策機能との連携	記載内容の追加 • Spring Security が提供する `HeaderWriterFilter` の仕様変更と `DelegatingRequestMatcherHeaderWriter` でのリクエストパスのマッチングにおけるバグについての注意事項を追加 (管理 ID#3601) • Spring Security がサポートするセキュリティヘッダの一覧に Referrer-Policy ヘッダを追加 (管理 ID#3675)
	OAuth	記載内容の修正・追加 • CVE-2019-3778(オープンリダイレクト脆弱性) に関する注意喚起を追加 (管理 ID#3931) • Spring Security OAuth のバージョン更新に伴いリダイレクト URI 情報を保持するテーブルへの説明に Warning を追加 (管理 ID#3590) • `alias` 属性を用いた `authentication-manager` の定義に関する実装例、説明の修正 (管理 ID#3386)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	チュートリアル *(Todo アプリケーション)*	記載内容の修正・追加 • 一覧表示機能作成時に、登録機能の一部を作成していた部分を変更し、一覧表示機能の動作確認できるように、コード例を追加 (管理 ID#3233) • ガイドライン修正に伴う、サンプルコードの最新化
	チュートリアル *(Todo アプリケーション REST 編)*	記載内容の修正 • spring-mvc-rest.xml を作成する方法の説明を変更 (管理 ID#3283) • ガイドライン修正に伴う、サンプルコードの最新化
	セッションチュートリアル	記載内容の修正 • JSP のコードを Tiles 形式に修正 (管理 ID#3270) • ガイドライン修正に伴う、サンプルコードの最新化
	Spring Security チュートリアル	記載内容の修正 • `SPRING_SECURITY_LAST_EXCEPTION` が格納されるスコープの誤記を修正 (管理 ID#2994) • ガイドライン修正に伴う、サンプルコードの最新化
	ボイラープレートコードの排除 *(Lombok)*	記載内容の追加 • Lombok 1.16.22 における `@Data` と `@NoArgsConstructor` を付与する順序についての Warning を追加 (管理 ID#3613)
2018-03-16	-	5.4.1 RELEASE 版公開
	全般	ガイドラインの誤記 (タイプミスや単純な記述ミスなど) の修正 記載内容の改善
	TERASOLUNA Server Framework for Java (5.x) のスタック	利用する OSS のバージョンを更新 (管理 ID#3061) • Spring IO Platform のバージョンを Brussels-SR5 に更新 • MyBatis のバージョンを 3.4.5 に更新 Spring IO Platform のバージョン更新に伴い利用する OSS のバージョンを更新 CVE-2018-1199 への対応のため、利用する OSS のバージョンを更新 (管理 ID#3300) • Spring Framework のバージョンを 4.3.14 に更新 • Spring Security のバージョンを 4.2.4 に更新

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	ドメイン層の実装	記載内容の追加 • `@Transactional` アノテーションの `timeout` 属性に関する記載を追加 (管理 ID#1776)
	入力チェック	記載内容の追加 • `@Compare` アノテーションの `operator` 属性に新たに追加された `NOT_EQUAL` の説明を追加 (管理 ID#2842) • `@Email` アノテーションを使用する際の注意事項を追加 (管理 ID#2930) ガイドラインのバグ修正 • 共通ライブラリのチェックルールの拡張方法の実装例を修正 (管理 ID#2822)
	例外ハンドリング	記載内容の修正 • 共通ライブラリ (`ExceptionLoggingFilter`) の変更に伴う修正、及び既存の誤記の修正 (管理 ID#2794)
	Tiles による画面レイアウト	記載内容の修正 • `<definition>` タグ (Tiles 定義ファイル) の `name` 属性のマッチングに関する説明、及び関連する箇所の誤解を招く表現を修正 (管理 ID#2717)
	REST クライアント（*HTTP* クライアント）	Spring Framework 4.3 対応に伴う修正 • Basic 認証用のリクエストヘッダの設定に関する記載を変更 (管理 ID#2742)
	SOAP Web Service（サーバ/クライアント）	記載内容の修正 • SOAP Web Service の実装に伴うインジェクションで使用するアノテーションを`@Inject` から`@Autowired` に変更 (管理 ID#2763) • Spring Framework の JAX-WS 連携機能についての誤記修正と、SOAP サーバが Java EE サーバの JAW-WS 実装上で動作することに伴なう注意事項の追記 (管理 ID#2770)
	JMS(Java Message Service)	記載内容の修正 • 非同期送信のトランザクション管理は ChainedTransactionManager ではなく DefaultMessageListenerContainer で行うよう記述を修正 (管理 ID#2814)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	認証	記載内容の修正 • パスワードハッシュ化のためのクラス（Pbkdf2PasswordEncoder）の説明を追記し、それに伴い BCryptPasswordEncoder を推奨する記述を削除 (管理 ID#3011)
	認可	Spring Framework 4.3 対応に伴う修正 • ブランクプロジェクトから mvc:path-matching の定義を削除し Spring MVC のデフォルト設定を使用するよう変更したことに伴う記載内容の修正 (管理 ID#2941) 記載内容の修正 • Spring Security でパス変数を使用するアクセスポリシーの定義に関する記載内容を修正 (管理 ID#3090)
	XSS 対策	記載内容の修正・追加 • JavaScript Escaping のサンプルソースを修正 (管理 ID#2531) • document.write() を使用する際の注意事項を追加 (管理 ID#2531)
	OAuth	構成見直し • How to use をグラントタイプ毎に説明する章構成に変更 (管理 ID#2818) 記載内容の追加 • Spring Security OAuth で発生する例外の一覧とハンドリング方法の追加 (管理 ID#2819) • Spring Security OAuth の拡張ポイントについての説明を追加 (管理 ID#2820) • リソースサーバに対する Basic 認証設定方法の追加 (管理 ID#2891) • インプリシットにおける後処理（アクセストークンクリア）の追加 (管理 ID#2891) 記載内容の改善 • サンプルコードの修正 (管理 ID#2891) • フロー図およびその説明の改善 (管理 ID#2891) • 認可サーバのチェックトークンエンドポイントの URL 設定が反映されない不具合への Warning を削除 (管理 ID#3263)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	単体テスト	新規追加 • 単体テストを追加 (管理 ID#1817)
2017-11-10	-	5.3.1 RELEASE 版公開
	全般	ガイドラインの誤記 (タイプミスや単純な記述ミスなど) の修正
2017-03-17	-	5.3.0 RELEASE 版公開
	全般	ガイドラインの誤記 (タイプミスや単純な記述ミスなど) の修正 記載内容の改善 ブランクプロジェクト生成用の Maven アーキタイプのデプロイ先変更 (Maven Central に変更) に伴う起動オプションの修正 (管理 ID#2444) • はじめての *Spring MVC* アプリケーション • *Web* アプリケーション向け開発プロジェクトの作成 • チュートリアル *(Todo* アプリケーション*)* • *Spring Security* チュートリアル
	ガイドラインの観点別マッピング	記載内容の追加 • セキュリティ対策に関するマッピングに CVE による観点をまとめた表を追加 (管理 ID#2439)
	利用規約	記載内容の修正 • 利用規約の修正 (管理 ID#2625)
	TERASOLUNA Server Framework for Java (5.x) のスタック	利用する OSS のバージョンを更新 (管理 ID#2441) • Spring IO Platform のバージョンを Athens-SR2 に更新 • MyBatis のバージョンを 3.4.2 に更新 • MyBatis-Spring のバージョンを 1.3.1 に更新 • mybatis-typehandlers-jsr310 を 1.0.2 に更新 Spring IO Platform のバージョン更新に伴い利用する OSS のバージョンを更新
	ドメイン層の実装	記載内容の改善 • シグネチャを制限するインタフェースおよび基底クラスの実装サンプルを修正 (管理 ID#2219)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	アプリケーション層の実装	記載内容の追加 • 単純な view controller を作成したい場合、<mvc:view-controller> を使用する様に追記 (管理 ID#2371) • Cookie の名前や値に使用できない文字が存在することの注意事項を追加 (管理 ID#2518) Spring Framework 4.3 対応に伴う修正 • JSR-310 Date and Time API のクラスに対して、@DateTimeFormat を使用する際の注意点を削除 (管理 ID#2505)
	入力チェック	記載内容の追加 • コレクション内の値に対する入力チェック方法を追加 (管理 ID#407) 記載内容の改善 • メッセージに入力チェック対象を含める方法の説明を追加 (管理 ID#2002) • @URL による入力チェックのチェック内容に関する記述を修正 (管理 ID#2260)
	例外ハンドリング	Spring Framework 4.3 対応に伴う修正 • 致命的なエラーのハンドリング方法について追記 (管理 ID#2368)
	セッション管理	記載内容の追加 • セッションスコープに格納しているオブジェクトを受け取る際にリクエストパラメータのバインドを防止する方法について追記 (管理 ID#1293)
	国際化	記載内容の追加 • 国際化が適用されない場合の例とその対策方法を追加 (管理 ID#2427)
	ファイルアップロード	記載内容の追加 • JBoss EAP 7.0 使用時の文字化け回避方法に関する説明を追加 (管理 ID#2403)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	RESTful Web Service	Spring Framework 4.3 対応に伴う修正 • HEAD と OPTIONS メソッドが暗黙的に用意される説明を追加 (管理 ID#1704) 記載内容の追加 • HTTP ステータスコードの説明句（`reason-phrase`）の出力仕様に関する説明を追加 (管理 ID#2518)
	REST クライアント（*HTTP* クライアント）	Spring Framework 4.3 対応に伴う修正 • 非同期リクエストの共通処理の実装に関する説明を追加 (管理 ID#2369)
	データベースアクセス（*MyBatis3* 編）	記載内容の変更、追加 • JSR-310 Date and Time API を使う場合の設定方法に関する記載を変更 (管理 ID#2365) 記載内容の追加 • コミット時にエラーが発生した場合にロールバック処理を呼び出すための設定に関する記載を追加 (管理 ID#2375) 記載内容の修正 • BLOB と CLOB を使用する場合の実装例を修正 (管理 ID#1775) •「Lazy Load」を実行するのタイミングを制御するオプションの説明を修正 (管理 ID#2364)
	データベースアクセス（*JPA* 編）	記載内容の追加 • PostgreSQL 使用時に」nowait」句が付加されない不具合に対する注意事項を追加 (管理 ID#2372)
	排他制御	記載内容の追加 • PostgreSQL 使用時に」nowait」句が付加されない不具合に対する注意事項を追加 (管理 ID#2372)
	E-mail 送信 *(SMTP)*	記載内容の追加 • JavaMail で発生する問題とその回避方法を追加 (管理 ID#2190)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	認証	記載内容の追加 • Remember Me 認証に使用するチェックボックスの value 属性値について追記 (管理 ID#785) • `<mvc:view-controller>` を使用する際の注意点を追記 (管理 ID#2371) 記載内容の修正 • SecureRandom の使用についての記載を修正 (管理 ID#2177)
	認可	Spring Framework 4.3 対応に伴う修正 • `AntPathMatcher` の `trimTokens` プロパティのデフォルト値が変更されたことに伴い、CVE-2016-5007 の対応方法に関する説明及び注意点を修正 (管理 ID#2565) 記載内容の追加 • 特定 URL に対するアクセス制限に関する Warning を追記 (管理 ID#2399) • パス変数の使用方法の説明と使用時の注意点を追記 (管理 ID#2406) • `AntPathRequestMatcher` のパスマッチングの仕様変更に対する注意点を追記 (管理 ID#2428)
	ブラウザのセキュリティ対策機能との連携	Spring Security 4.1.4 対応に伴う修正 • Content Security Policy (CSP)」に関する記載を追加 (管理 ID#2400) • HTTP Public Key Pinning (HPKP) に関する記載を追加 (管理 ID#2401)
	OAuth	新規追加 • OAuth を追加 (管理 ID#2145)
	チュートリアル *(Todo* アプリケーション*)*	記載内容の修正 • JPA を利用する場合の Entity のコード例の修正 (管理 ID#2476)
	NEXUS による *Maven* リポジトリの管理	Maven Central への移行に伴う修正 • TERASOLUNA Server Framework for Java (5.x) のリポジトリに関する記述を削除 (管理 ID#2496)
2016-08-31	-	5.2.0 RELEASE 版公開

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	全般	ガイドラインの誤記 (タイプミスや単純な記述ミスなど) の修正 記載内容の改善 章立てを全面見直し (管理 ID#1683) 共通ライブラリのバージョンを 5.2.0.RELEASE に更新 (管理 ID#2076) 記載内容の改善 • 共通ライブラリの pom の依存性について追加 (管理 ID#1982)
	TERASOLUNA Server Framework for Java (5.x) のスタック	記載内容の追加 • ブランクプロジェクトの共通ライブラリ標準の組込状況を追加 (管理 ID#1700) • mybatis-typehandlers-jsr310 、jackson-datatype-jsr310 を OSS スタックに追加 (管理 ID#1966) • spring-jms およびその依存ライブラリを OSS スタックに追加 (管理 ID#1992) 利用する OSS のバージョン (Spring IO Platform のバージョン) を更新 • Spring IO Platform のバージョンを 2.0.6.RELEASE に更新 • Spring Framework のバージョンを 4.2.7.RELEASE に更新 • Spring Security のバージョンを 4.0.4.RELEASE に更新 Spring IO Platform のバージョン更新に伴い利用する OSS のバージョンを更新
	ドメイン層の実装	記載内容の追加 • MyBatis 3.3 + MyBatis-Spring 1.2 において、@Transactinal の timeout 属性に指定した値は使用されない旨を追加 (管理 ID#1777)
	アプリケーション層の実装	記載内容の追加 • HttpSession をハンドラメソッドの引数として使用すべきでない旨を追加 (管理 ID#1313) • JSR-310 Date and Time API を使用する際の注意点を記載 (管理 ID#1991)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	入力チェック	記載内容の改善 • メッセージプロパティファイルを Native to Ascii せずに直接扱う方法を追加 (管理 ID#994) • cross-field validation について追加 (管理 ID#1561) • @DateTimeFormat の説明を追加 (管理 ID#1873) • ValidationMessages.properties についての説明を修正 (管理 ID#1948) • Method Validation を利用した入力チェックの注意事項を追加 (管理 ID#1998) 記載内容の追加 • OS コマンドインジェクションに関する記載を追加 (管理 ID#1957)
	例外ハンドリング	Spring Framework 4.2.7 対応に伴う修正 • HTTP レスポンスヘッダー出力に関する説明内容を修正 (管理 ID#1965)
	二重送信防止	記載内容の追加 • `@TransactionTokenCheck` アノテーションの type 属性に新たに追加された `TransactionTokenType.CHECK` の仕様、利用方法に関する記載の追加 (管理 ID#2071) 「How To Extend プログラマティックにトランザクショントークンのライフサイクルを管理する方法について」を削除。 • `TransactionTokenContext` が提供していたアプリケーション向け API を使用した場合、`TransactionToken` を正しい状態に維持できなくなるなど、フレームワーク内部の挙動に影響を及ぼすような作り込みができてしまうことから、当該 API の非推奨化がなされた。非推奨化にあわせて該当機能の利用方法の記述を削除した。
	国際化	記載内容の改善 • リクエストパラメータ (デフォルトのパラメータ名) の説明の位置を修正 (管理 ID#1354)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	ファイルアップロード	記載内容の追加 • CVE-2016-3092(File Upload の脆弱性) に関する注意喚起を追加 (管理 ID#1973) • ディレクトリトラバーサル攻撃に関する記載を追加 (管理 ID#2010)
	ヘルスチェック	新規追加 • ヘルスチェックを追加 (管理 ID#1698)
	RESTful Web Service	記載内容の変更、追加 • JSR-310 Date and Time API / Joda Time を使う場合の設定の記述を変更 (管理 ID#1966) • Jackson を Java SE 7 環境で使用する場合の注意点を記載 (管理 ID#1966) • JSON で JSR-310 Date and Time API を使う場合の設定を記載 (管理 ID#1966)
	REST クライアント（*HTTP* クライアント）	記載内容の改善 • RestClient における HTTP Proxy サーバの設定を追加 (管理 ID#1856)
	SOAP Web Service（サーバ/クライアント）	記載内容の追加 • SOAP クライアント起動時に SOAP サーバに接続しないオプションを追加 (管理 ID#1871) • SOAP クライアントの env プロジェクトに関する説明の修正 (管理 ID#1901) • SOAP Web サービス例外発生時のステータスコード取得方法について追加 (管理 ID#2007)
	データベースアクセス（*MyBatis3* 編）	記載内容の追加 • 暫定的な WARN ログ出力回避方法を削除 (管理 ID#1292) • JSR-310 Date and Time API を Mybatis3.3 で使用するための設定方法を記載 (管理 ID#1966) • MyBatis を Java SE 7 環境で使用する場合の注意点を記載 (管理 ID#1966)
	排他制御	記載内容の追加 • ExclusionControl に warning メッセージを追加 (管理 ID#1694)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	ロギング	記載内容の追加 • ID 付きログメッセージを出力するための拡張方法を記載 (管理 ID#1928)
	文字列処理	記載内容の追加 • terasoluna-gfw-string を dependency に追加する例を追加 (管理 ID#1699) • @Size アノテーションの説明にサロゲートペアについての注意を追加 (管理 ID#1874) • JIS 漢字 U+2014(EM DASH) の UCS(Unicode) 文字対応について記載を追加 (管理 ID#1914)
	Bean マッピング (Dozer)	記載内容の追加 • JSR-310 Date and Time API を使用する際の注意点を記載 (管理 ID#1966)
	JMS(Java Message Service)	新規追加 • JMS を追加 (管理 ID#1407)
	認証	Spring Security 4.0.4 対応に伴う修正 • Spring Security 4.0.4 にて authentication-failure-url の仕様が改善されたことによるコード例の修正と Note の削除 (管理 ID#1963)
	認可	記載内容の追加 • CVE-2016-5007 Spring Security / MVC Path Matching Inconsistency の対応方法を追加 (管理 ID#1976)
	代表的なセキュリティ要件の実装例	記載内容の追加 •「セキュリティ観点での入力値チェック」を追加 •「監査ログ出力」を追加
	参考書籍	記載内容の追加 •「Spring 徹底入門」を参考書籍として追加 (管理 ID#2043)
2016-02-24	-	5.1.0 RELEASE 版公開
	全般	ガイドラインの誤記 (タイプミスや単純な記述ミスなど) の修正 記載内容の改善

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	はじめに	記載内容の追加 • ガイドラインに記載している内容の動作検証環境に関する記載を追加

	TERASOLUNA Server Framework for Java (5.x) のスタック	利用する OSS のバージョン (Spring IO Platform のバージョン) を更新
		• Spring IO Platform のバージョンを 2.0.1.RELEASE に更新
		• Spring Framework のバージョンを 4.2.4.RELEASE に更新
		• Spring Security のバージョンを 4.0.3.RELEASE に更新
		Spring IO Platform のバージョン更新に伴い利用する OSS のバージョンを更新
		• 使用する OSS のバージョンを更新。更新内容は、version 5.1.0 の移行ガイドを参照されたい。
		新規プロジェクト追加
		• `terasoluna-gfw-string`、`terasoluna-gfw-codepoints`、`terasoluna-gfw-validator`、`terasoluna-gfw-web-jsp` プロジェクトの説明を追加。
		共通ライブラリの新機能追加
		`terasoluna-gfw-string`
		• 半角全角変換
		`terasoluna-gfw-codepoints`
		• コードポイントチェック
		• コードポイントチェック用 Bean Validation 制約アノテーション
		`terasoluna-gfw-validator`
		• バイト長チェック用 Bean Validation 制約アノテーション
		• フィールド値比較相関チェック用 Bean Validation 制約アノテーション

	はじめての *Spring MVC* アプリケーション	記述内容の改善 • Spring Security 4 対応に伴うサンプルソースの修正 (管理 ID#1519) • `AuthenticationPrincipalArgumentResolver` のパッケージ変更

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	チュートリアル *(Todo アプリケーション)*	Spring Security 4 対応に伴う修正 • Spring Security 4 対応に伴うソースの修正 (管理 ID#1519) • `AuthenticationPrincipalArgumentResolver` のパッケージ変更 • デフォルトで true になる仕様のため、サンプルソースから`<use-expressions="true">`を削除
	Web アプリケーション向け開発プロジェクトの作成	記述内容の改善 • オフライン環境上で mvn コマンドを利用する方法を追加 (管理 ID#1197)
	アプリケーション層の実装	記述内容の改善 • EL 関数を用いたリクエスト URL 作成方法について追加 (管理 ID#632)
	データベースアクセス（共通編）	記載内容の追加 • `Log4jdbcProxyDataSource` のオーバヘッドに対する注意点を追加 (管理 ID#1471)
	データベースアクセス（*MyBatis3* 編）	MyBatis 3.3 対応に伴う記載内容の追加 • `defaultFetchSize` の設定方法を追加 (管理 ID#965) • 遅延読み込み時のデフォルトが `JAVASSIST` に変更されている点を追加 (管理 ID#1384) • `ResultHandler` に Generics を付与したサンプルコードに修正 (管理 ID#1384) • 新規追加された`@Flush` アノテーションを利用したソース例、及び注意点を追加 (管理 ID#915)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	データベースアクセス（*JPA* 編）	ガイドラインのバグ修正 • Like 条件を使用するユーティリティを適切に修正 (管理 ID#1464) • JPQL における真偽値の不適切な実装を修正 (管理 ID#1525) • ページネーションの不適切な実装を修正 (管理 ID#1463) • `DateTimeProvider` を実装したサンプルコードの不適切な実装を修正 (管理 ID#1327) • 共通 Repository インタフェースの実装クラスのインスタンスを生成するための Factory クラスにおいて不適切な実装を修正 (管理 ID#1327) 記載内容の改善 • `hibernate.hbm2ddl.auto` のデフォルト値を修正 (管理 ID#1282)
	入力チェック	記述内容の改善 • MethodValidation に対する記述を追加 (管理 ID#708)
	ロギング	記述内容の改善 • Logback の設定に `ServiceLoader` の仕組みを利用した記述の追加 (管理 ID#1275) • Spring Security 4 対応に伴うサンプルソースの修正 (管理 ID#1519) • デフォルトで true になる仕様のため、サンプルソースから`<use-expressions="true">`を削除
	セッション管理	記述内容の改善 • SpEL 式を用いたセッションスコープ参照の記述を追加 (管理 ID#1306)
	国際化	記述内容の改善 • JSP に適切にロケールを反映させるための記述を追加 (管理 ID#1439) • `SessionLocalResolver` の `defaultLocale` の説明を修正 (管理 ID#686)
	コードリスト	記載内容の追加 • JdbcCodeList に `JdbcTemplate` を指定するパターンを推奨とする記述を追加 (管理 ID#501)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	RESTful Web Service	記述内容の改善 • `Jackson2ObjectMapperFactoryBean` を利用した ObjectMapper 作成を追加 (管理 ID#1022) • REST API アプリケーションのドメイン層の実装に MyBatis3 を前提とした形に修正 (管理 ID#1323)
	REST クライアント（HTTP クライアント）	新規追加 • REST クライアント（HTTP クライアント）を追加 (管理 ID#1307)
	SOAP Web Service（サーバ/クライアント）	新規追加 • SOAP Web Service（サーバ/クライアント）を追加 (管理 ID#1340)
	ファイルアップロード	記述内容の改善 • アップロード処理の基本フロー、及びその説明を Spring の `MultipartFilter` を用いた記述に修正 (管理 ID#193) • セキュリティ上の問題や、AP サーバによって動作が異なる等の課題があるため、「クエリパラメータで CSRF トークンを送る方法」を削除。ファイルアップロードの許容サイズを超過した場合、一部 AP サーバで CSRF トークンチェックが正しく行われない注意点を追加 (管理 ID#1602)
	ファイルダウンロード	Spring Framework 4.2 対応に伴う記載内容の追加 • xlsx 形式を操作する `AbstractXlsxView` の追加 (管理 ID#996) 記述内容の改善 • iText の仕様変更のため、`com.lowagie:itext:4.2.1` を利用したソース例を `com.lowagie:itext:2.1.7` を利用する形に修正
	E-mail 送信 (SMTP)	新規追加 • E-mail 送信 (SMTP) を追加 (管理 ID#1165)
	日付操作 (JSR-310 Date and Time API)	新規追加 • 日付操作 (JSR-310 Date and Time API) を追加 (管理 ID#1450)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	日付操作 *(Joda Time)*	記載内容の改善・追加 • タイムゾーンを利用しない年月日を扱うサンプルコードのオブジェクトを `LocalDate` に修正 (管理 ID#1283) • Java8 未満のバージョンで和暦を扱う方法を追加 (管理 ID#1450)
	文字列処理	新規追加 • 文字列処理を追加 (管理 ID#1451)
	セキュリティ対策	構成見直し • パスワードハッシュ化は、認証 に移動 • 認証 から、セッション管理の項目を セッション管理 として独立
	Spring Security 概要	Spring Security 4 対応に伴う修正 • 全記述の再編 • `spring-security-test` の紹介 • デフォルトで true になる仕様のため、サンプルソースから`<use-expressions="true">`を削除 • `RedirectAuthenticationHandler` 非推奨化に伴う説明の削除
	Spring Security チュートリアル	Spring Security 4 対応に伴う修正 • チュートリアルのソースを Spring Security 4 に対応した形に修正 (管理 ID#1519)
	認証	Spring Security 4 対応に伴う修正 (管理 ID#1519) • 全記述の再編 • `auto-config="true"`の削除 • 認証イベントリスナを`@org.springframework.context.event.EventListener` に修正 • `AuthenticationPrincipal` のパッケージを修正 • デフォルトでプレフィックスが付与されるため、サンプルソースから `ROLE_`プレフィックスの削除

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	認可	Spring Security 4 対応に伴う修正 (管理 ID#1519) • 全記述の再編 • デフォルトでプレフィックスが付与されるため、サンプルソースから ROLE_プレフィックスの削除 • デフォルトで true になる仕様のため、サンプルソースから<use-expressions="true">を削除 • @PreAuthorize の定義例追加
	CSRF 対策	Spring Security 4 対応に伴う修正 • 全記述の再編 • CSRF 無効化の設定を修正<sec:csrf disabled="true"/> • 記述内容の改善 • マルチパートリクエストに関する項目を ファイルアップロード に移動 (管理 ID#1602)
	暗号化	新規追加 • 暗号化ガイドラインの追加 (管理 ID#1106)
	代表的なセキュリティ要件の実装例	新規追加 • 代表的なセキュリティ要件の実装例を追加 (管理 ID#1604)
	セッションチュートリアル	新規追加 • セッションチュートリアルを追加 (管理 ID#1599)
	チュートリアル *(Todo* アプリケーション *REST* 編*)*	Spring Security 4 対応に伴う修正 • Spring Security 4 対応に伴うソースの修正 (管理 ID#1519) • CSRF 無効化の設定を修正<sec:csrf disabled="true"/> • デフォルトで true になる仕様のため、サンプルソースから<use-expressions="true">を削除
2015-08-05	-	5.0.1 RELEASE 版公開
	全般	ガイドラインの誤記 (タイプミスや単純な記述ミスなど) の修正 記載内容の改善 アプリケーションサーバに関する記載内容の修正 • Resin に関する記載を削除 • リファレンスページへのリンクを最新化

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	はじめに	記載内容の追加 • ガイドラインに記載している内容の動作検証環境に関する記載を追加
	TERASOLUNA Server Framework for Java (5.x) のスタック	セキュリティ脆弱性対応に伴い利用する OSS のバージョン (Spring IO Platform のバージョン) を更新 • Spring IO Platform のバージョンを 1.1.3.RELEASE に更新 • Spring Framework のバージョンを 4.1.7.RELEASE に更新 (CVE-2015-3192) • JSTL のバージョンを 1.2.5 に更新 (CVE-2015-0254) Spring IO Platform のバージョン更新に伴い利用する OSS のバージョンを更新 • 使用する OSS のバージョンを更新。更新内容は、version 5.0.1 の移行ガイドを参照されたい。 記載内容の改善 (管理 ID#1148) • `terasoluna-gfw-recommended-dependencies`、`terasoluna-gfw-recommended-web-dependencies`、`terasoluna-gfw-parent` プロジェクトの説明を追加。 • プロジェクトの説明を修正。 • プロジェクト間の依存関係を示す図を追加。
	Web アプリケーション向け開発プロジェクトの作成	記載内容の追加 • war ファイルのビルド方法を追加 (管理 ID#1146)
	データベースアクセス（共通編）	記載内容の追加 • データソース切り替え機能の説明を追加 (管理 ID#1071)
	データベースアクセス（*MyBatis3* 編）	ガイドラインのバグ修正 • バッチ実行のタイミングに関する説明を修正 (管理 ID#903)
	ロギング	記載内容の改善 • `<logger>`タグの `additivity` 属性に関する説明を追加 (管理 ID#977)
	セッション管理	記載内容の改善 • セッションスコープの Bean の定義方法に関する説明を修正 (管理 ID#1082)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	二重送信防止	記載内容の追加 • レスポンスをキャッシュしないように設定している時のトランザクショントークンチェックの動作を補足 (管理 ID#1260)
	コードリスト	記載内容の追加 • コード名の表示方法を追加 (管理 ID#1109)
	Ajax *RESTful Web Service*	CVE-2015-3192(XML の脆弱性) に関する注意喚起を追加 • StAX(Streaming API for XML) を使用する際の注意事項を追加 (管理 ID#1211)
	ページネーション 共通ライブラリが提供する *JSP Tag Library と EL Functions*	共通ライブラリのバグ改修に伴う修正 • 共通ライブラリのバグ改修 (terasoluna-gfw#297) に伴い、`f:query` の仕様に関する説明を修正 (管理 ID#1244)
	認証	記載内容の改善 • `ExceptionMappingAuthenticationFailureHandler` の親クラスのプロパティの扱いに関する注意点を追加 (管理 ID#812) • `AbstractAuthenticationProcessingFilter` の `requiresAuthenticationRequestMatcher` プロパティの設定例を修正 (管理 ID#1110)
	認可	ガイドラインのバグ修正 • `<sec:authorize>` タグ (JSP タグライブラリ) の `access` 属性の設定例を修正 (管理 ID#1003)
	環境依存性の排除	記載内容の追加 • Tomcat8 使用時の外部クラスパス (Tomcat7 の `VirtualWebappLoader` の代替機能) の適用方法を追加 (管理 ID#1081)
2015-06-12	全般	5.0.0 RELEASE 英語版公開

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
2015-03-06	*RESTful Web Service*	ガイドラインのバグ修正 • 例外ハンドリング用のサンプルコード (`NullPointerException` が発生するコードが含まれている問題) を修正 (管理 ID#918)
	チュートリアル *(Todo アプリケーション REST 編)*	ガイドラインのバグ修正 • 例外ハンドリングの処理で `NullPointerException` が発生する問題を修正 (管理 ID#918)
2015-02-23	-	5.0.0 RELEASE 版公開
	全般	ガイドラインの誤記 (タイプミスや単純な記述ミスなど) の修正 記載内容の改善 新規追加 • *Web* アプリケーション向け開発プロジェクトの作成 • データベースアクセス（*MyBatis3* 編） • 共通ライブラリが提供する *JSP Tag Library* と *EL Functions* • ボイラープレートコードの排除 *(Lombok)* version 5.0.0 対応に伴う更新 • MyBatis2 の説明を削除
	TERASOLUNA Server Framework for Java (5.x) のスタック	Spring IO Platform 対応 • 一部のライブラリを除き、推奨ライブラリの管理を Spring IO Platform に委譲する構成に変更した旨を追加。 OSS バージョンの更新 • 使用する OSS のバージョンを更新。更新内容は、version 5.0.0 の移行ガイドを参照されたい。
	はじめての *Spring MVC* アプリケーション	version 5.0.0 対応に伴う更新 • Spring Framework 4.1 の適用。 • ドキュメント上の構成の見直し。
	アプリケーションのレイヤ化	英語翻訳のバグ修正 • ドメイン層と他の層との関係に関する翻訳ミスを修正 (管理 ID#364)
	チュートリアル *(Todo アプリケーション)*	version 5.0.0 対応に伴う更新 • Spring Framework 4.1 の適用。 • インフラストラクチャ層として MyBatis3 をサポート。 • ドキュメント上の構成の見直し。

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	Web アプリケーション向け開発プロジェクトの作成	新規追加 • マルチプロジェクト構成のプロジェクト作成方法を追加。
	ドメイン層の実装	Spring Framework 4.1 対応に伴う修正 • JTA 1.2 の`@Transactional` の扱いに関する記載を追加 (管理 ID#562) • JPA(Hibernate 実装) 使用時の`@Transactional(readOnly = true)` の扱い関する説明を修正。SPR-8959(Spring Framework 4.1 以降) の対応により、JDBC ドライバに対して「読み取り専用のトランザクション」として扱うように指示できるように改善された。 記載内容の追加 •「読み取り専用のトランザクション」が有効にならないケースに関する注意点を追加。追加内容の詳細は、(管理 ID#861)
	インフラストラクチャ層の実装	MyBatis3 対応に伴う修正 • RepositoryImpl の実装として MyBatis3 の仕組みを利用する方法を追加。
	アプリケーション層の実装	Spring Framework 4.1 対応に伴う修正 • `@ControllerAdvice` に追加された属性 (適用対象を Controller で絞り込むための属性) に関する説明を追加 (管理 ID#549) • `<mvc:view-resolvers>`に関する説明を追加 (管理 ID#609)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	データベースアクセス（共通編）	共通ライブラリのバグ改修に伴う修正 • 共通ライブラリのバグ改修 (terasoluna-gfw#78) に伴い、全角文字のワイルドカード文字 (「％」，「＿」) の扱いに関する説明を追加 (管理 ID#712) Spring Framework 4.1 対応に伴う修正 • JPA(Hibernate 実装) の悲観ロックエラーが Spring Framework の `PessimisticLockingFailureException` に変換されない問題に関する記載を削除。この問題は、SPR-10815(Spring Framework 4.0 以降) で解決済みである。 Apache Commons DBCP 2.0 対応に伴う修正 • Apache Commons DBCP 2.0 用のコンポーネントを使用するようにサンプルコード及び説明を変更。
	データベースアクセス（*MyBatis3* 編）	新規追加 • O/R Mapper として MyBatis3 を使用してインフラストラクチャ層を実装する方法を追加。
	排他制御	ガイドラインのバグ修正 • ロングトランザクションの楽観ロックのサンプルコード (レコードが取得できない時の処理) の修正 (管理 ID#450) Spring Framework 4.1 対応に伴う修正 • JPA(Hibernate 実装) の悲観ロックエラーが Spring Framework の `PessimisticLockingFailureException` に変換されない問題に関する記載を削除。この問題は、SPR-10815(Spring Framework 4.0 以降) で解決済みである。 MyBatis3 対応に伴う修正 • MyBatis3 使用時の排他制御の実装方法を追加。

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	入力チェック	ガイドラインのバグ修正 • `@GroupSequence` の説明を修正 (管理 ID#296) 共通ライブラリのバグ改修に伴う修正 • 共通ライブラリのバグ改修 (terasoluna-gfw#256) に伴い、`ValidationMessages.properties` に関する注意点を追加 (管理 ID#766) 記載内容の追加 • Spring Validator を使用した相関項目チェック時に、Bean Validation の Group Validation の仕組みと連携する方法を追加。追加内容の詳細は、(管理 ID#320) Bean Validation 1.1(Hibernate Validator 5.1) 対応に伴う修正 • `@DecimalMin` と`@DecimalMax` の `inclusive` 属性の説明を追加。 • Expression Language に関する記載を追加。 • Bean Validation 1.1 から非推奨になった API について記載。 • Hibernate Validator 5.1.x の `ValidationMessages.properties` に関するバグ (HV-881) に関する記載と回避方法を追加。
	例外ハンドリング	記載内容の追加 • 513 バイトより小さいサイズのエラーレスポンスを返すと Internet Explorer で簡易エラーページが表示される可能性がある旨の説明を追加。追加内容の詳細は、(管理 ID#189) Spring Framework 4.1 対応に伴う修正 • JPA(Hibernate 実装) の悲観ロックエラーが Spring Framework の `PessimisticLockingFailureException` に変換されない問題に関する記載を削除。この問題は、SPR-10815(Spring Framework 4.0 以降) で解決済みである。
	セッション管理	Spring Security 3.2 対応に伴う修正 • POST リクエスト時にセッションタイムアウトではなく CSRF トークンエラーが発生する問題 (SEC-2422) に関する記載を削除。Spring Security 3.2 の正式版ではセッションタイムアウトを検知できる仕組みが組み込まれており、問題が解消されている。

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	メッセージ管理	共通ライブラリの変更内容の反映 • 共通ライブラリの改善 (terasoluna-gfw#24) に伴い、新たに追加したメッセージタイプ (warning) と非推奨にしたメッセージタイプ (warn) に関する説明を追加 (管理 ID#74)
	ページネーション	共通ライブラリの変更内容の反映 • 共通ライブラリの改善 (terasoluna-gfw#13) に伴い、active 状態のページリンクの説明を変更 (管理 ID#699) • 共通ライブラリの改善 (terasoluna-gfw#14) に伴い、disabled 状態のページリンクの説明を変更 (管理 ID#700) Spring Data Common 1.9 対応に伴う修正 • バージョンアップに伴い、API 仕様が変更されているクラス (`Page` インタフェースなど) に対する注意点を追加。
	コードリスト	共通ライブラリのバグ改修に伴う修正 • 共通ライブラリのバグ改修 (terasoluna-gfw#16) に伴い、`ExistInCodeList` のメッセージキーを変更とバージョンアップ時の注意点を追加 (管理 ID#638) • 共通ライブラリのバグ改修 (terasoluna-gfw#256) に伴い、`@ExistInCodeList` のメッセージ定義に関する注意点を追加 (管理 ID#766) 共通ライブラリの変更内容の反映 • 共通ライブラリの機能追加 (terasoluna-gfw#25) に伴い、`EnumCodeList` クラスの使用方法を追加。
	Ajax	Spring Security 3.2 対応に伴う修正 • CSRF 対策のサンプルコード (CSRF 対策用の`<meta>`タグの生成方法) を変更。 Jackson 2.4 対応に伴う修正 • Jackson 2.4 用のコンポーネントを使用するようにサンプルコード及び説明を変更。

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	RESTful Web Service	記載内容の改善 • Location ヘッダやハイパーメディアリンクに設定する URL を組み立てる方法を改善。改善内容の詳細は、(管理 ID#374) Spring Framework 4.1 対応に伴う修正 • `@RestController` に関する説明を追加 (管理 ID#560) • ビルダースタイルの API を使用して `ResponseEntity` を生成するようにサンプルコードを変更。 Jackson 2.4 対応に伴う修正 • Jackson 2.4 用のコンポーネントを使用するようにサンプルコード及び説明を変更。 Spring Data Common 1.9 対応に伴う修正 • バージョンアップに伴い、API 仕様が変更されているクラス (`Page` インタフェースなど) に対する注意点を追加。
	ファイルアップロード	ガイドラインのバグ修正 • CVE-2014-0050(File Upload の脆弱性) が解決された Apache Commons FileUpload のバージョンを修正 (管理 ID#846) 記載内容の追加 • 一部のアプリケーションサーバで Servlet 3 のファイルアップロード機能が文字化けする問題があるため、この事象の回避策として Apache Commons FileUpload を使用する方法を追加。追加内容の詳細は、(管理 ID#778)
	システム時刻	共通ライブラリの変更内容の反映 • 共通ライブラリの改善 (terasoluna-gfw#224) に伴い、ドキュメント内の構成とパッケージ名及びクラス名を変更 (管理 ID#701)
	Tiles による画面レイアウト	Tiles 3.0 対応に伴う修正 • Tiles 3.0 用のコンポーネントを使用するように設定例及び説明を変更。 Spring Framework 4.1 対応に伴う修正 • `<mvc:view-resolvers>`、`<mvc:tiles>`、`<mvc:definitions>`に関する説明を追加 (管理 ID#609)

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	日付操作 *(Joda Time)*	記載内容の追加 • `LocalDateTime` の使い方を追加。追加内容の詳細は、(管理 ID#584) Joda Time 2.5 対応に伴う修正 • バージョンアップに伴い `DateMidnight` クラスが非推奨になったため、指定日の開始時刻 (0:00:00.000) の取得方法を変更。
	Spring Security 概要	Spring Security 3.2 対応に伴う修正 • Appendix に「セキュアな HTTP ヘッダー付与の設定」を追加。
	Spring Security チュートリアル	version 5.0.0 対応に伴う更新 • インフラストラクチャ層として MyBatis3 を使用するように変更。 • Spring Framework 4.1 対応の適用。 • Spring Security 3.2 対応の適用。 • ドキュメント上の構成の見直し。

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	認証	ガイドラインのバグ修正 • <form-login>、<logout>、<session-management>タグの説明不備や説明不足の修正 (管理 ID#754) • AuthenticationFilter の拡張方法を示すサンプルコードの修正 (セッション・フィクセーション攻撃対策や CSRF 対策を有効にするための設定を追加) (管理 ID#765) Spring Security 3.2 対応に伴う修正 • CSRF 対策を有効にしている際のログアウト方法に関する注意点を追加。 • Controller から UserDetails(認証ユーザ情報クラス) にアクセスする方法として、@AuthenticationPrincipal の説明を追加。 • <sec:session-management>の session-fixation-protection 属性のパラメータとして、changeSessionId の説明を追加。 • セッションタイムアウトの検出方法と注意点を追加。 • 同一ユーザの同時セッション数制御 (Concurrent Session Control) を有効にするための設定方法を変更 (<sec:concurrency-control>を使用するように変更)。 • 同一ユーザの同時セッション数制御関連のクラスが非推奨になり別のクラスが提供されている旨を追加。
	CSRF 対策	Spring Security 3.2 対応に伴う修正 • version 1.0.x の共通ライブラリに同封していた Spring Security 3.2.0(正式リリース前の暫定バージョン) の CSRF 対策用コンポーネントに関する記載を削除。 • CSRF 対策を有効にするための設定方法を Spring Security 3.2 の正式な作法 (<sec:csrf>を使用する方法) に変更。 • CSRF 対策用の JSP タグライブラリ (<sec:csrfInput>と<sec:csrfMetaTags>) に関する記載を追加。 • CSRF 対策を有効にしている時のセッションタイムアウトの検出方法と注意点を追加。 Spring Framework 4.1 対応に伴う修正 • <form:form>を使用した際に、CSRF トークンが hidden として出力される条件に関する記載を変更。

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	チュートリアル *(Todo アプリケーション REST 編)*	記載内容の改善 • チュートリアル *(Todo アプリケーション)* で作成したプロジェクトに REST API を追加する手順にすることで、特定のインフラストラクチャ層 (O/R Mapper) に依存しない内容に変更 (管理 ID#325) version 5.0.0 対応に伴う更新 • Spring Framework 4.1 対応の適用。 • Spring Security 3.2 対応の適用。 • Jackson 2.4 対応の適用。
	ブランクプロジェクトから新規プロジェクトの作成	記載内容の改善 • マルチプロジェクト構成のプロジェクト作成方法をサポート。 • シングルプロジェクト構成のプロジェクト作成方法を最新化。
	共通ライブラリが提供する *JSP Tag Library* と *EL Functions*	新規追加 • 共通ライブラリから提供している JSP タグライブラリと EL 関数の説明を追加。
	ボイラープレートコードの排除 *(Lombok)*	新規追加 • Lombok を使用したボイラープレートコードの排除方法の説明を追加。

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	英語版	以下の英語版を追加 • *Web* アプリケーション向け開発プロジェクトの作成 • データベースアクセス（共通編） • データベースアクセス（*JPA* 編） • データベースアクセス（*MyBatis3* 編） • 排他制御 • ロギング • プロパティ管理 • ページネーション • 二重送信防止 • 国際化 • コードリスト • *Ajax* • *RESTful Web Service* • ファイルアップロード • ファイルダウンロード • *Tiles* による画面レイアウト • システム時刻 • *Bean* マッピング *(Dozer)* • *Spring Security* 概要 • 認証 • 認可 • *CSRF* 対策 • ブランクプロジェクトから新規のプロジェクトの作成 • *NEXUS* による *Maven* リポジトリの管理 • 環境依存性の排除 • Project Structure Standard • ボイラープレートコードの排除 *(Lombok)* • *Spring Framework* 理解度チェックテスト
2014-08-27	-	1.0.1 RELEASE 版公開
	全般	ガイドラインのバグ (タイプミスや記述ミスなど) を修正
	日本語版	以下の日本語版を追加 • ガイドラインの観点別マッピング • *RESTful Web Service* • チュートリアル *(Todo* アプリケーション *REST* 編*)*

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	英語版	以下の英語版を追加 • はじめに • アーキテクチャ概要 • チュートリアル *(Todo アプリケーション)* • アプリケーション開発 • 入力チェック • 例外ハンドリング • メッセージ管理 • 日付操作 *(Joda Time)* • *XSS* 対策 • 参考書籍
	TERASOLUNA Server Framework for Java (5.x) のスタック	バグ改修に伴い利用する OSS のバージョンを更新 • GroupId「`org.springframework`」のバージョンを 3.2.4.RELEASE から 3.2.10.RELEASE に更新 • GroupId「`org.springframework.data`」ArtifactId「`spring-data-commons`」のバージョンを 1.6.1.RELEASE から 1.6.4.RELEASE に更新 • GroupId「`org.springframework.data`」ArtifactId「`spring-data-jpa`」のバージョンを 1.4.1.RELEASE から 1.4.3.RELEASE に更新 • GroupId「`org.aspectj`」のバージョンを 1.7.3 から 1.7.4 に更新 • GroupId「`javax.transaction`」ArtifactId「`jta`」を削除
	アプリケーション層の実装	CVE-2014-1904(`<form:form>`タグの action 属性の XSS 脆弱性) に関する注意喚起を追加
	日本語版 メッセージ管理	バグ改修に関する記載を追加 • 共通ライブラリから提供している`<t:messagesPanel>`タグのバグ改修 (terasoluna-gfw#10)
	日本語版 ページネーション	バグ改修に関する記載を更新 • 共通ライブラリから提供している`<t:pagination>`タグのバグ改修 (terasoluna-gfw#12) • Spring Data Commons のバグ改修 (terasoluna-gfw#22)
	日本語版 *Ajax*	XXE Injection 対策に関する記載を更新

Continued on next page

表 1.2 – continued from previous page

更新日付	更新箇所	更新内容
	日本語版 ファイルアップロード	CVE-2014-0050(File Upload の脆弱性) に関する注意喚起を追加 ガイドラインのバグを修正 • `MultipartFilter` を設定した場合、`SystemExceptionResolver` を使用して `MultipartException` をハンドリングする事が出来ないため、サーブレットコンテナの error-page 機能を使用してハンドリングする方法を追加 (管理 ID#59)
	日本語版	以下のプロジェクト作成方法を `mvn archetype:generate` から行うように変更 • はじめての *Spring MVC* アプリケーション • チュートリアル *(Todo* アプリケーション*)* • チュートリアル *(Todo* アプリケーション*)*
	日本語版	以下の Maven アーキタイプ作成方法を微修正 • *Spring Security* チュートリアル • ブランクプロジェクトから新規プロジェクトの作成
2013-12-17	日本語版	1.0.0 Public Review 版公開

第 2 章

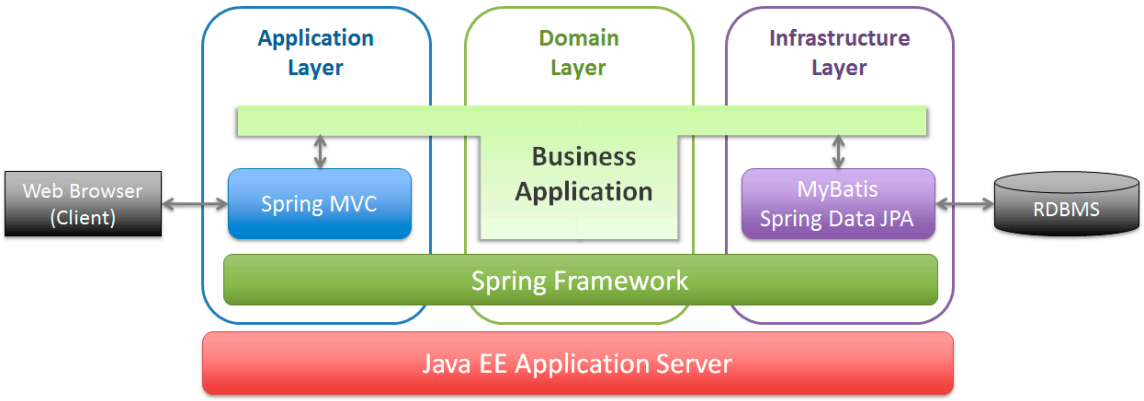
アーキテクチャ概要

本ガイドラインで想定しているアーキテクチャについて説明する。

2.1 TERASOLUNA Server Framework for Java (5.x) のスタック

2.1.1 TERASOLUNA Server Framework for Java (5.x) の Software Framework 概要

TERASOLUNA Server Framework for Java (5.x) で使用する Software Framework は独自のフレームワークではなく、Spring Framework を中心とした OSS の組み合わせである。



2.1.2 Software Framework の主な構成要素

TERASOLUNA Server Framework for Java (5.x) を構成するライブラリを以下に示す。

DI コンテナ

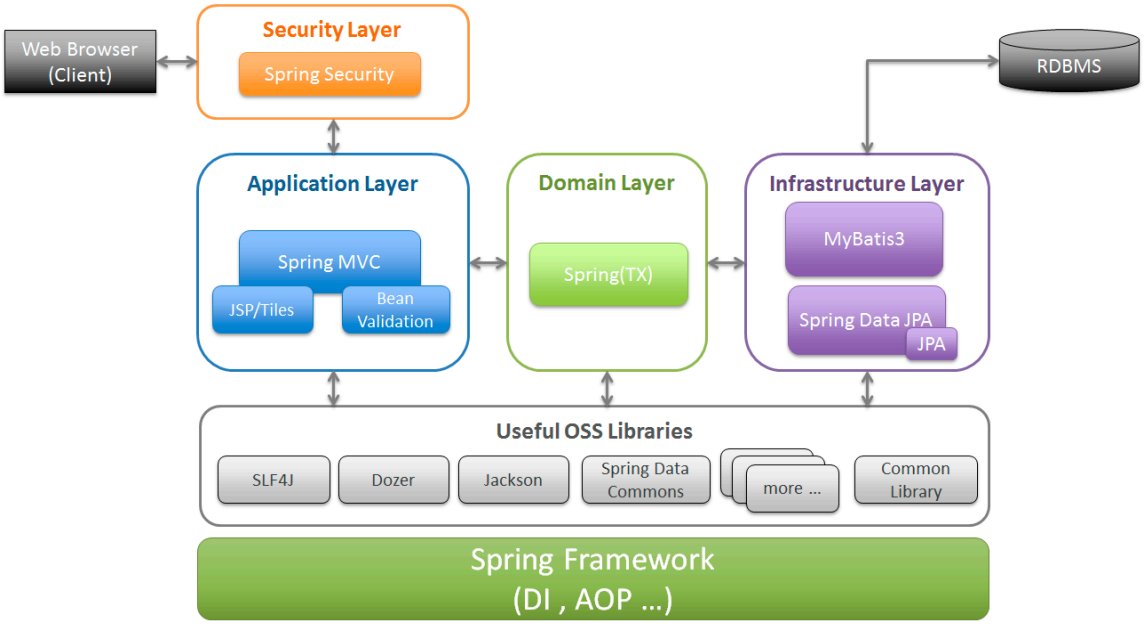
DI コンテナとして Spring Framework を利用する。

- Spring Framework 4.3

MVC フレームワーク

Web MVC フレームワークとして Spring MVC を利用する。

- Spring MVC 4.3



O/R Mapper

本ガイドラインでは、以下のいずれかを想定している。

- MyBatis 3.4
  - Spring Framework との連携ライブラリとして、MyBatis-Spring を使用する。
- JPA2.1
  - プロバイダは、Hibernate 5.0 を使用する。

注釈: MyBatis は正確には「SQL Mapper」であるが、本ガイドラインでは「O/R Mapper」に分類する。

> **警告:** どんなプロジェクトでも JPA を採用できるわけではない。」テーブルがほとんど正規化されていない」、」テーブルのカラム数が多すぎる」というテーブル設計がされている場合には、JPA の利用は難しい。
>
> また、本ガイドラインでは JPA の基本的な説明は行っておらず、JPA 利用経験者がチーム内にいることが前提である。

View

View には JSP を利用する。

View のレイアウトを共通化する場合は、

- Apache Tiles 3.0

を利用する。

セキュリティ
認証・認可のフレームワークとして Spring Security を利用する。

- Spring Security 4.2

ちなみに： Spring Security 3.2 から、認証・認可の仕組みの提供に加えて、悪意のある攻撃者から Web アプリケーションを守るための仕組みが強化されている。

悪意のある攻撃者から Web アプリケーションを守るための仕組みについては、

- *CSRF* 対策
- ブラウザのセキュリティ対策機能との連携

を参照されたい。

バリデーション
- 単項目チェックには BeanValidation 1.1 を利用する。
  - 実装は、Hibernate Validator 5.3 を利用する。
- 相関チェックには Bean Validation、もしくは Spring Validation を利用する。
  - 使い分けについては入力チェックを参照されたい。

ロギング
- ロガーの API は SLF4J を使用する。
  - ロガーの実装は、Logback を利用する。

共通ライブラリ
- https://github.com/terasolunaorg/terasoluna-gfw
- 詳細は共通ライブラリの構成要素を参照されたい。

2.1.3 利用する OSS のバージョン

version 5.4.2.RELEASE で利用する OSS の一覧を以下に示す。

ちなみに： version 5.0.0.RELEASE より、Spring IO platform の`<dependencyManagement>`をインポートする構成を採用している。

Spring IO platform の`<dependencyManagement>`をインポートすることで、

- Spring Framework が提供しているライブラリ
- Spring Framework が依存している OSS ライブラリ
- Spring Framework と相性のよい OSS ライブラリ

への依存関係を解決しており、TERASOLUNA Server Framework for Java (5.x) で使用する OSS のバージョンは、原則として、Spring IO platform の定義に準じている。

なお、version 5.4.2.RELEASE で指定している Spring IO platform のバージョンは、Brussels-SR17 である。

Type	GroupId	ArtifactId	Version	Spring IO platform	Remarks
Spring	org.springframework	spring-aop	4.3.23.RELEASE	*	
Spring	org.springframework	spring-aspects	4.3.23.RELEASE	*	
Spring	org.springframework	spring-beans	4.3.23.RELEASE	*	
Spring	org.springframework	spring-context	4.3.23.RELEASE	*	
Spring	org.springframework	spring-context-support	4.3.23.RELEASE	*	
Spring	org.springframework	spring-core	4.3.23.RELEASE	*	
Spring	org.springframework	spring-expression	4.3.23.RELEASE	*	
Spring	org.springframework	spring-jdbc	4.3.23.RELEASE	*	
Spring	org.springframework	spring-orm	4.3.23.RELEASE	*	
Spring	org.springframework	spring-tx	4.3.23.RELEASE	*	
Spring	org.springframework	spring-web	4.3.23.RELEASE	*	
Spring	org.springframework	spring-webmvc	4.3.23.RELEASE	*	
Spring	org.springframework	spring-jms	4.3.23.RELEASE	*	
Spring	org.springframework	spring-messaging	4.3.23.RELEASE	*	
Spring	org.springframework.data	spring-data-commons	1.13.20.RELEASE	*	
Spring	org.springframework.security	spring-security-acl	4.2.12.RELEASE	*	
Spring	org.springframework.security	spring-security-config	4.2.12.RELEASE	*	
Spring	org.springframework.security	spring-security-core	4.2.12.RELEASE	*	
Spring	org.springframework.security	spring-security-taglibs	4.2.12.RELEASE	*	
Spring	org.springframework.security	spring-security-web	4.2.12.RELEASE	*	
Spring	org.springframework.security.oauth	spring-security-oauth2	2.0.17.RELEASE	*	
MyBatis3	org.mybatis	mybatis	3.4.6		*1
MyBatis3	org.mybatis	mybatis-spring	1.3.2		*1
JPA(Hibernate)	antlr	antlr	2.7.7	*	*2
JPA(Hibernate)	dom4j	dom4j	1.6.1	*	*2
JPA(Hibernate)	org.hibernate	hibernate-core	5.0.12.Final	*	*2
JPA(Hibernate)	org.hibernate	hibernate-entitymanager	5.0.12.Final	*	*2
JPA(Hibernate)	org.hibernate.common	hibernate-commons-annotations	5.0.1.Final	*	*2 *4
JPA(Hibernate)	org.hibernate.javax.persistence	hibernate-jpa-2.1-api	1.0.0.Final	*	*2 *4
JPA(Hibernate)	org.javassist	javassist	3.21.0-GA	*	*2
JPA(Hibernate)	org.jboss	jandex	2.0.0.Final	*	*2 *4

Continued on next page

表 2.1 – continued from previous page

Type	GroupId	ArtifactId	Version	Spring IO platform	Remarks
JPA(Hibernate)	org.springframework.data	spring-data-jpa	1.11.20.RELEASE	*	*2
JPA(Hibernate)	org.apache.geronimo.specs	geronimo-jta_1.1_spec	1.1.1	*	*2
DI	javax.inject	javax.inject	1	*	
AOP	aopalliance	aopalliance	1.0	*	
AOP	org.aspectj	aspectjrt	1.8.14	*	
AOP	org.aspectj	aspectjweaver	1.8.14	*	
ログ出力	ch.qos.logback	logback-classic	1.1.11	*	
ログ出力	ch.qos.logback	logback-core	1.1.11	*	*4
ログ出力	org.lazyluke	log4jdbc-remix	0.2.7		
ログ出力	org.slf4j	jcl-over-slf4j	1.7.26	*	
ログ出力	org.slf4j	slf4j-api	1.7.26	*	
JSON	com.fasterxml.jackson.core	jackson-annotations	2.8.0	*	
JSON	com.fasterxml.jackson.core	jackson-core	2.8.11	*	
JSON	com.fasterxml.jackson.core	jackson-databind	2.8.11.3	*	
JSON	com.fasterxml.jackson.datatype	jackson-datatype-joda	2.8.11	*	
JSON	com.fasterxml.jackson.datatype	jackson-datatype-jsr310	2.8.11	*	*6
入力チェック	javax.validation	validation-api	1.1.0.Final	*	
入力チェック	org.hibernate	hibernate-validator	5.3.6.Final	*	
入力チェック	org.jboss.logging	jboss-logging	3.3.2.Final	*	*4
入力チェック	com.fasterxml	classmate	1.3.4	*	*4
Bean 変換	commons-beanutils	commons-beanutils	1.9.3	*	*3
Bean 変換	net.sf.dozer	dozer	5.5.1		*3
Bean 変換	net.sf.dozer	dozer-spring	5.5.1		*3
Bean 変換	org.apache.commons	commons-lang3	3.5	*	*3
日付操作	joda-time	joda-time	2.9.9	*	
日付操作	joda-time	joda-time-jsptags	1.1.1		*3
日付操作	org.jadira.usertype	usertype.core	5.0.0.GA		*2
日付操作	org.jadira.usertype	usertype.spi	5.0.0.GA		*2
コネクションプール	org.apache.commons	commons-dbcp2	2.1.1	*	*3
コネクションプール	org.apache.commons	commons-pool2	2.4.3	*	*3
Tiles	commons-digester	commons-digester	2.0	*	*3
Tiles	org.apache.tiles	tiles-api	3.0.8	*	*3
Tiles	org.apache.tiles	tiles-core	3.0.8	*	*3
Tiles	org.apache.tiles	tiles-jsp	3.0.8	*	*3
Tiles	org.apache.tiles	tiles-servlet	3.0.8	*	*3

Continued on next page

表 2.1 – continued from previous page

Type	GroupId	ArtifactId	Version	Spring IO platform	Remarks
Tiles	org.apache.tiles	tiles-template	3.0.8	*	*3 *4
Tiles	org.apache.tiles	tiles-autotag-core-runtime	1.2	*	*3 *4
Tiles	org.apache.tiles	tiles-request-servlet	1.0.7	*	*3 *4
Tiles	org.apache.tiles	tiles-request-api	1.0.7	*	*3
Tiles	org.apache.tiles	tiles-request-jsp	1.0.7	*	*3 *4
ユーティリティ	com.google.guava	guava	20.0	*	
ユーティリティ	commons-collections	commons-collections	3.2.2	*	*3
ユーティリティ	commons-io	commons-io	2.5	*	*3
サーブレット	org.apache.taglibs	taglibs-standard-jstlel	1.2.5	*	
サーブレット	org.apache.taglibs	taglibs-standard-spec	1.2.5	*	*4
サーブレット	org.apache.taglibs	taglibs-standard-impl	1.2.5	*	*4

1. データアクセスに、MyBatis3 を使用する場合に依存するライブラリ
2. データアクセスに、JPA を使用する場合に依存するライブラリ
3. 共通ライブラリに依存しないが、TERASOLUNA Server Framework for Java (5.x) でアプリケーションを開発する場合に、利用することを推奨しているライブラリ
4. Spring IO platform でサポートしているライブラリが個別に依存しているライブラリ
   (Spring IO platform としては依存関係の管理は行っていないライブラリ)
5. Spring IO platform で適用されるバージョンが、Beta や RC(Release Candidate) であるライブラリ
   (TERASOLUNA Server Framework for Java (5.x) 側で GA のバージョンを明示的に指定しているライブラリ)
6. Java SE 8 以降の API を前提としたライブラリ
   Java SE 7 以下の場合は本ライブラリへの依存関係を明示的に除去する必要がある

2.1.4 共通ライブラリの構成要素

共通ライブラリは、TERASOLUNA Server Framework for Java (5.x) が含む Spring Ecosystem や、その他依存ライブラリでは足りない $+\alpha$ な機能を提供するライブラリである。基本的には、このライブラリがなくても TERASOLUNA Server Framework for Java (5.x) によるアプリケーション開発は可能であるが、」あると便利」な存在である。また、提供している 2 種類の マルチプロジェクト構成のブランクプロジェクトおよび シングルプロジェクト構成のブランクプロジェクトの共通ライブラリの標準の組込状況は以下の通りである。

項番	プロジェクト名	概要	Java ソースコード有無	マルチプロジェクト構成のブランクプロジェクト組込	シングルプロジェクト構成のブランクプロジェクト組込
(1)	terasoluna-gfw-parent	依存ライブラリの管理とビルド用プラグインの推奨設定を提供する。	無	有*1	有*1
(2)	terasoluna-gfw-common-libraries	共通ライブラリのうち、Java ソースコードを含むプロジェクトの構成を定義する。依存関係として pom.xml に追加する必要はない。(5.2.0 から追加)	無	無	無
(3)	terasoluna-gfw-dependencies	共通ライブラリのうち、依存関係定義のみを提供するプロジェクト (terasoluna-gfw-parent 以外) の構成を定義する。依存関係として pom.xml に追加する必要はない。(5.2.0 から追加)	無	無	無
(4)	terasoluna-gfw-common	Web に依存しない汎用的に使用できる機能を提供する。本ライブラリを利用する場合は、依存関係として terasoluna-gfw-common-dependencies を pom.xml に追加する。	有	有*2	有*2
(5)	terasoluna-gfw-common-dependencies	terasoluna-gfw-common プロジェクトが提供する機能を使用する場合の依存関係定義を提供する。(5.2.0 から追加)	無	有	有
(6)	terasoluna-gfw-jodatime	Joda Time に依存する機能を提供する。本ライブラリを利用する場合は、依存関係として terasoluna-gfw-jodatime-dependencies を pom.xml に追加する。(5.0.0 から追加)	有	有*2	有*2
(7)	terasoluna-gfw-jodatime-dependencies	terasoluna-gfw-jodatime プロジェクトが提供する機能を使用する場合の依存関係定義を提供する。(5.2.0 から追加)	無	有	有
(8)	terasoluna-gfw-web	Web アプリケーションを作成する場合に使用する機能を提供する。View に依存しない機能を集約している。本ライブラリを利用する場合は、依存関係として terasoluna-gfw-web-dependencies を pom.xml に追加する。	有	有*2	有*2
(9)	terasoluna-gfw-web-dependencies	terasoluna-gfw-web プロジェクトが提供する機能を使用する場合の依存関係定義を提供する。(5.2.0 から追加)	無	有	有

Continued on next page

表 2.2 – continued from previous page

項番	プロジェクト名	概要	Java ソースコード有無	マルチプロジェクト構成のブランクプロジェクト組込	シングルプロジェクト構成のブランクプロジェクト組込
(10)	terasoluna-gfw-web-jsp	View に JSP を採用する Web アプリケーションを作成する場合に使用する機能を提供する。本ライブラリを利用する場合は、依存関係として terasoluna-gfw-web-jsp-dependencies を pom.xml に追加する。	有	有*2	有*2
(11)	terasoluna-gfw-web-jsp-dependencies	terasoluna-gfw-web-jsp プロジェクトが提供する機能を使用する場合の依存関係定義を提供する。(5.2.0 から追加)	無	有	有
(12)	terasoluna-gfw-security-web	Spring Security の拡張部品を提供する。本ライブラリを利用する場合は、依存関係として terasoluna-gfw-security-web-dependencies を pom.xml に追加する。	有	有*2	有*2
(13)	terasoluna-gfw-security-web-dependencies	Spring Security を使用する場合の依存関係定義 (Web 関連) と、terasoluna-gfw-security-web プロジェクトが提供する機能を使用する場合の依存関係定義を提供する。(5.2.0 から追加)	無	有	有
(14)	terasoluna-gfw-string	文字列処理に関連する機能を提供する。(5.1.0 から追加)	有	無	無
(15)	terasoluna-gfw-codepoints	対象の文字列を構成するコードポイントがコードポイント集合に含まれることをチェックする機能を提供する。(5.1.0 から追加)	有	無*3	無*3
(16)	terasoluna-gfw-validator	汎用的な Bean Validation の制約アノテーションを追加して提供する。(5.1.0 から追加)	有	無	無
(17)	terasoluna-gfw-security-core-dependencies	Spring Security を使用する場合の依存関係定義 (Web 以外) を提供する。(5.2.0 から追加)	無	有	有
(18)	terasoluna-gfw-mybatis3-dependencies	MyBatis3 を使用する場合の依存関係定義を提供する。(5.2.0 から追加)	無	有*4	有*4

Continued on next page

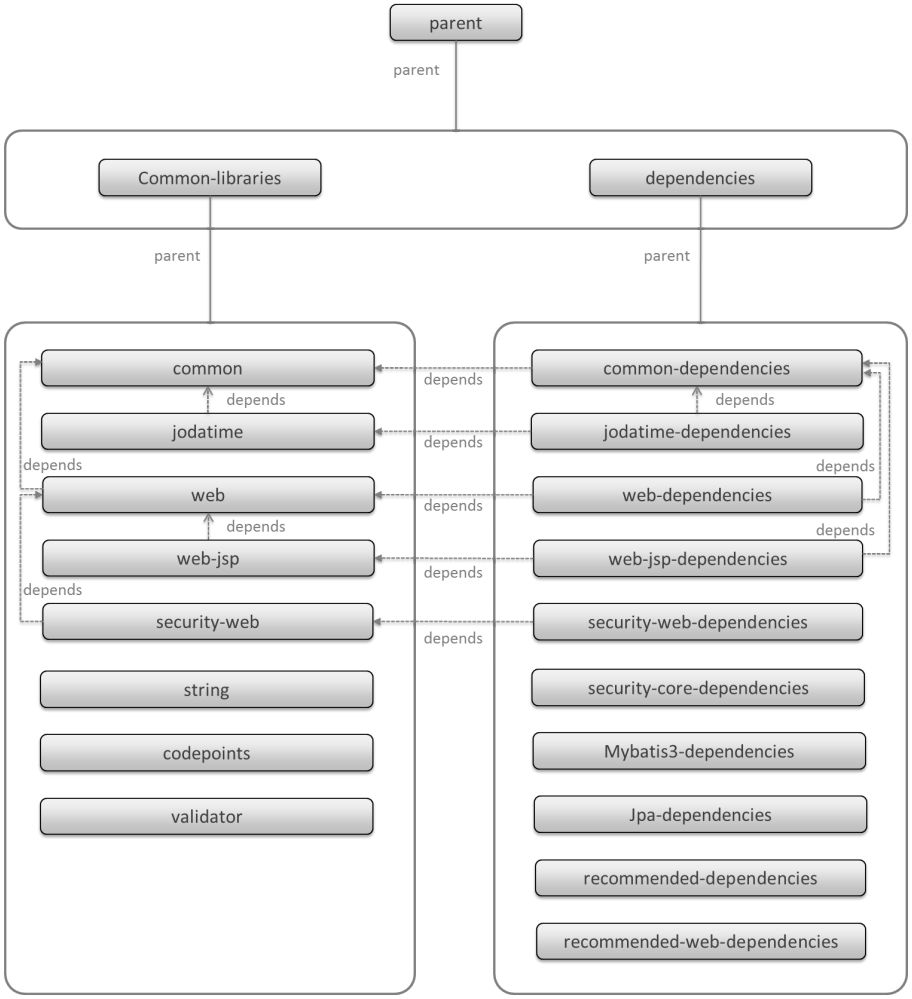
表 2.2 – continued from previous page

項番	プロジェクト名	概要	Java ソースコード有無	マルチプロジェクト構成のブランクプロジェクト組込	シングルプロジェクト構成のブランクプロジェクト組込
(19)	terasoluna-gfw-jpa-dependencies	JPA を使用する場合の依存関係定義を提供する。(5.2.0 から追加)	無	有*5	有*5
(20)	terasoluna-gfw-recommended-dependencies	Web に依存しない推奨ライブラリへの依存関係定義を提供する。	無	有	有
(21)	terasoluna-gfw-recommended-web-dependencies	Web に依存する推奨ライブラリへの依存関係定義を提供する。	無	有	有

1. <dependency>要素ではないが、各プロジェクトの<parent>要素として組み込まれる。
2. <dependency>要素ではないが、<dependency>要素からの推移的依存関係として組み込まれる。
3. 使用するコードポイント集合に応じて複数のアーティファクトを提供している。詳細は 共通ライブラリから提供しているコードポイント集合クラス を参照されたい。
4. データアクセスに、MyBatis3 を使用する場合に標準で組み込まれる共通ライブラリ
5. データアクセスに、JPA を使用する場合に標準で組み込まれる共通ライブラリ

Java ソースコードを含まないものは、ライブラリの依存関係のみ定義しているプロジェクトである。

なお、プロジェクトの依存関係は以下の通りである。



注釈: 一部を除き、共通ライブラリにはプロジェクト名末尾に」dependencies」が付与されたプロジェクトが存在する。(例えば、terasoluna-gfw-common に対応する terasoluna-gfw-common-dependencies などである)

このようなプロジェクトでは、共通ライブラリへの依存関係定義の他に、利用を推奨する OSS ライブラリへの依存関係定義を提供している為、共通ライブラリを利用する際は」dependencies」が付与されたプロジェクトの方を、依存関係として pom.xml に追加することを推奨する。

terasoluna-gfw-common

terasoluna-gfw-common は以下の部品を提供している。

分類	部品名	説明
例外ハンドリング	例外クラス	汎用的に使用できる例外クラスを提供する。
	例外ロガー	アプリケーション内で発生した例外をログに出力するためのロガークラスを提供する。
	例外コード	例外クラスに対応する例外コード (メッセージ ID) を解決するための仕組み (クラス) を提供する。
	例外ログ出力インターセプタ	ドメイン層で発生した例外をログ出力するためのインターセプタクラス (AOP) を提供する。
システム時刻	システム時刻ファクトリ	システム時刻を取得するためのクラスを提供する。
コードリスト	コードリスト	コードリストを生成するためのクラスを提供する。
データベースアクセス（共通編）	クエリエスケープ	SQL 及び JPQL にバインドする値のエスケープ処理を行うクラスを提供する。
	シーケンサ	シーケンス値を取得するためのクラスを提供する。

terasoluna-gfw-string

terasoluna-gfw-string は以下の部品を提供している。

分類	部品名	説明
文字列処理	半角全角変換	半角文字列と全角文字列のマッピングテーブルに基づき、入力文字列の半角文字を全角に変換する処理と全角文字を半角に変換する処理を行うクラスを提供する。

terasoluna-gfw-codepoints

terasoluna-gfw-codepoints は以下の部品を提供している。

分類	部品名	説明
文字列処理	コードポイントチェック	対象の文字列を構成するコードポイントが、定義されたコードポイント集合に含まれることをチェックするクラスを提供する。
入力チェック	コードポイントチェック用 Bean Validation 制約アノテーション	コードポイントチェックを Bean Validation で行うための制約アノテーションを提供する。

terasoluna-gfw-validator

terasoluna-gfw-validator は以下の部品を提供している。

分類	部品名	説明
入力チェック	バイト長チェック用 Bean Validation 制約アノテーション	入力文字列の文字コードにおけるバイト長が、指定した最大値以下であること、最小値以上であることのチェックを Bean Validation で行うための制約アノテーションを提供する。
	プロパティ値比較チェック用 Bean Validation 制約アノテーション	2 つのプロパティ値の比較チェックを Bean Validation で行うための制約アノテーションを提供する。

terasoluna-gfw-jodatime

terasoluna-gfw-jodatime は以下の部品を提供している。

分類	部品名	説明
システム時刻	Joda Time 用システム時刻ファクトリ	Joda Time の API を利用してシステム時刻を取得するためのクラスを提供する。

terasoluna-gfw-web

terasoluna-gfw-web は以下の部品を提供している。

分類	部品名	説明
二重送信防止	トランザクショントークンチェック	リクエストの二重送信から Web アプリケーションを守るための仕組み (クラス) を提供する。
例外ハンドリング	例外ハンドラ	共通ライブラリが提供する例外ハンドリングの部品と連携するための例外ハンドラクラス (Spring MVC 提供のクラスのサブクラス) を提供する。
	例外ログ出力インターセプタ	Spring MVC の例外ハンドラがハンドリングした例外をログ出力するためのインターセプタクラス (AOP) を提供する。
コードリスト	コードリスト埋込インターセプタ	View からコードリストを取得できるようにするために、コードリストの情報をリクエストスコープに格納するためのインターセプタクラス (Spring MVC Interceptor) を提供する。
ファイルダウンロード	汎用ダウンロード View	ストリームから取得したデータを、ダウンロード用のストリームに出力するための抽象クラスを提供する。
ロギング	トラッキング ID 格納用サーブレットフィルタ	トレーサビリティを向上させるために、クライアントから指定されたトラッキング ID を、ロガーの MDC(Mapped Diagnostic Context)、リクエストスコープ、レスポンスヘッダに設定するためのサーブレットフィルタクラスを提供する。(クライアントからトラッキング ID の指定がない場合は、本クラスでトラッキング ID を生成する)
	汎用 MDC 格納用サーブレットフィルタ	ロガーの MDC に任意の値を設定するための抽象クラスを提供する。
	MDC クリア用サーブレットフィルタ	ロガーの MDC に格納されている情報をクリアするためのサーブレットフィルタクラスを提供する。

terasoluna-gfw-web-jsp

terasoluna-gfw-web-jsp は以下の部品を提供している。

分類	部品名	説明
二重送信防止	トランザクショントークン出力用の JSP タグ	トランザクショントークンを hidden 項目として出力するための JSP タグライブラリを提供する。
ページネーション	ページネーションリンク表示用の JSP タグ	Spring Data Commons 提供のクラスと連携してページネーションリンクを表示するための JSP タグライブラリを提供する。
メッセージ管理	結果メッセージ表示用の JSP タグ	処理結果を表示するための JSP タグライブラリを提供する。
EL Functions	XSS 対策用 EL 関数	XSS 対策用の EL 関数を提供する。
	URL 用 EL 関数	URL エンコーディングなどの URL 用の EL 関数を提供する。
	DOM 変換用 EL 関数	DOM 文字列に変換するための EL 関数を提供する。
	ユーティリティ EL 関数	汎用的なユーティリティ処理を行うための EL 関数を提供する。

terasoluna-gfw-security-web

terasoluna-gfw-security-web は以下の部品を提供している。

分類	部品名	説明
ロギング	認証ユーザ名格納用サーブレットフィルタ	トレーサビリティを向上させるために、認証ユーザ名をロガーの MDC に設定するためのサーブレットフィルタクラスを提供する。

2.2 Spring MVC アーキテクチャ概要

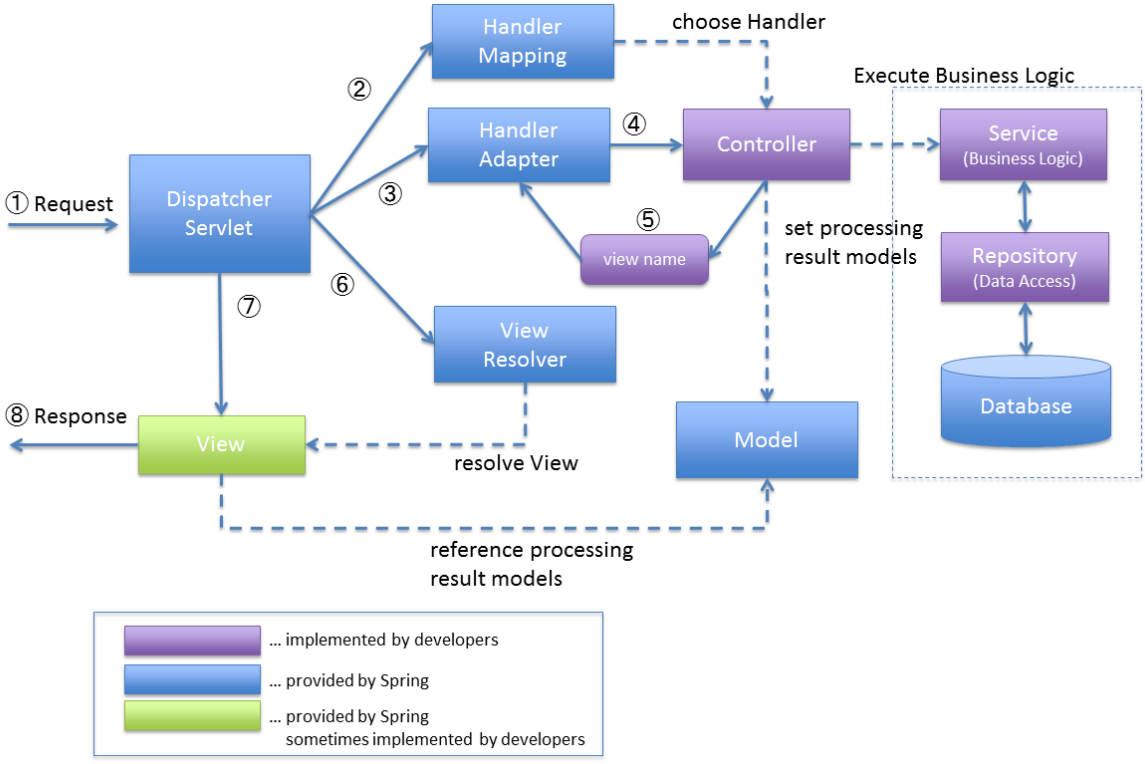
Spring MVC は、公式で以下のように説明されている。

Spring Reference Document.

> Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central Servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications. Spring's DispatcherServlet however, does more than just that. It is completely integrated with the Spring IoC container and as such allows you to use every other feature that Spring has.

2.2.1 Overview of Spring MVC Processing Sequence

リクエストを受けてから、レスポンスを返すまでの Spring MVC の処理フローを、以下の図に示す。



1. `DispatcherServlet` が、リクエストを受け取る。
2. `DispatcherServlet` は、リクエスト処理を行う `Controller` の選択を `HandlerMapping` に委譲する。`HandlerMapping` は、リクエスト URL にマッピングされている `Controller` を選定し（`Choose Handler`）、`Controller` を `DispatcherServlet` へ返却する。
3. `DispatcherServlet` は、`Controller` のビジネスロジック処理の実行を `HandlerAdapter` に委譲する。
4. `HandlerAdapter` は、`Controller` のビジネスロジック処理を呼び出す。

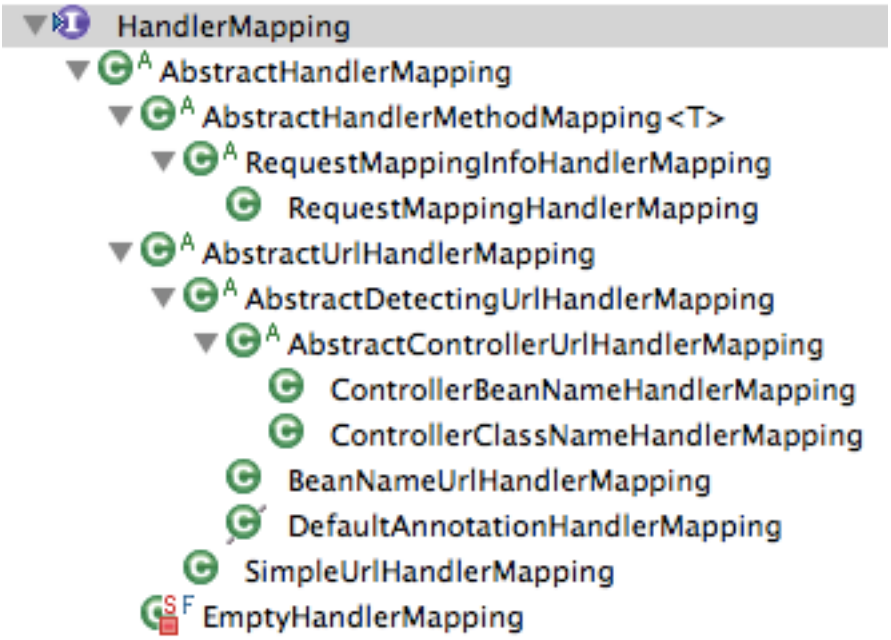
5. Controller は、ビジネスロジックを実行し、処理結果を Model に設定し、ビューの論理名を HandlerAdapter に返却する。
6. DispatcherServlet は、ビュー名に対応する View の解決を、ViewResolver に委譲する。ViewResolver は、ビュー名にマッピングされている View を返却する。
7. DispatcherServlet は、返却された View にレンダリング処理を委譲する。
8. View は、Model の持つ情報をレンダリングしてレスポンスを返却する。

2.2.2 Implementations of each component

これまで説明したコンポーネントのうち、拡張可能なコンポーネントを紹介する。

Implementation of HandlerMapping

Spring から提供されている HandlerMapping のクラス階層を、以下に示す。



通常使用するのは、org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping で
ある。

このクラスは、Bean 定義されている Contorller から@RequestMapping アノテーションを読み取り、URL と合致する Controller のメソッドを Handler クラスとして扱うクラスである。

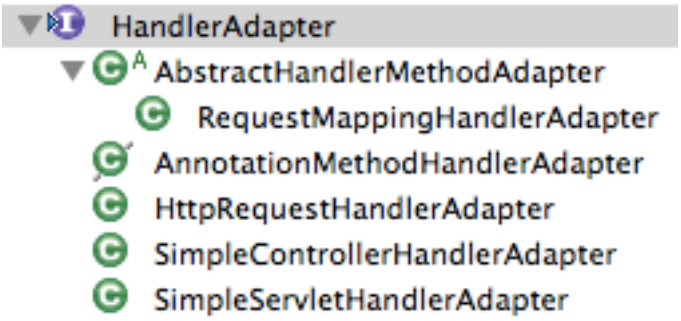
Spring Framework 3.1 からは、RequestMappingHandlerMapping は、DispatcherServlet が読み込む Bean 定義ファイルに、

<mvc:annotation-driven>の設定がある場合、デフォルトで設定される。

(<mvc:annotation-driven>アノテーションで有効になる設定は、Web MVC framework を参照されたい。)

Implementation of HandlerAdapter

Spring から提供されている HandlerAdapter のクラス階層を、以下に示す。



通常使用するのは、org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter で
ある。

このクラスは、HandlerMapping によって選択された Handler クラス (Controller) のメソッドを呼び出すクラスである。

このクラスも Spring Framework 3.1 からは、<mvc:annotation-driven>の設定がある場合、デフォルトで設定される。

Implementation of ViewResolver

Spring および依存ライブラリから提供されている ViewResolver のクラスを、以下に示す。

通常 (JSP を使う場合) は、

- org.springframework.web.servlet.view.InternalResourceViewResolver を使用するが、

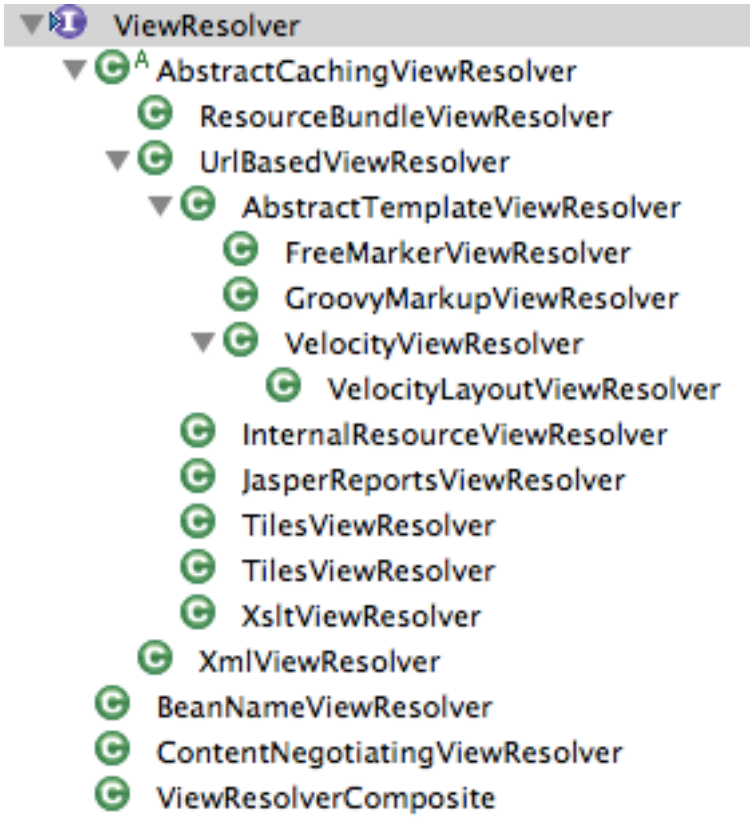
テンプレートエンジン Tiles を使う場合は、

- org.springframework.web.servlet.view.tiles3.TilesViewResolver

ファイルダウンロード用にストリームを返す場合は

- org.springframework.web.servlet.view.BeanNameViewResolver

のように、返す View によって使い分ける必要がある。



複数の種類の `View` を扱う場合、`ViewResolver` の定義が複数必要となるケースがある。

複数の `ViewResolver` を使う代表的な例として、ファイルのダウンロード処理が存在する画面アプリケーションが挙げられる。

画面 (JSP) は、`InternalResourceViewResolver` で `View` を解決し、

ファイルダウンロードは、`BeanNameViewResolver` などを使って `View` を解決する。

詳細はファイルダウンロードを参照されたい。

Implementation of View

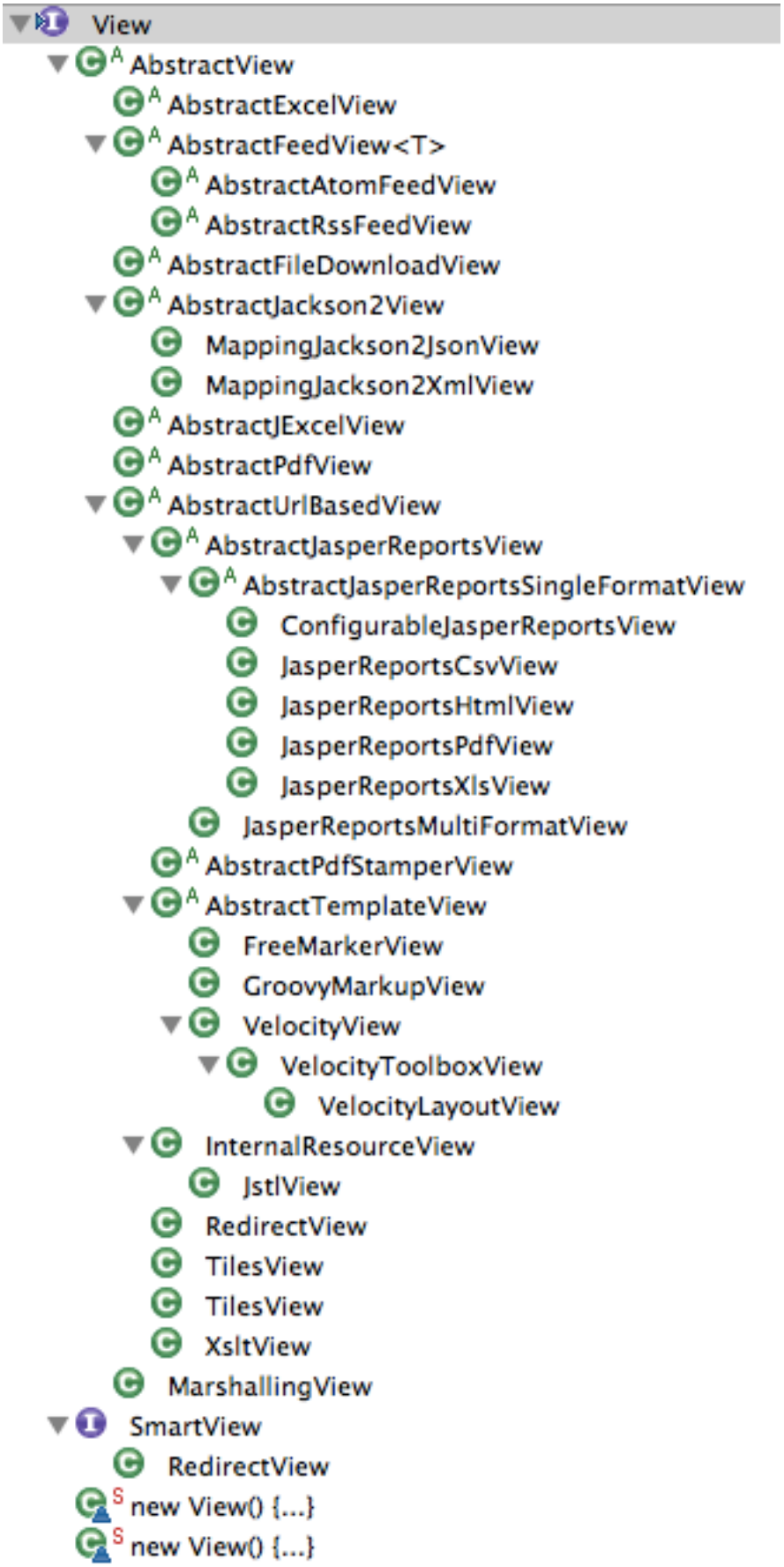
Spring および依存ライブラリから提供されている `View` のクラスを、以下に示す。

`View` は、返したいレスポンスの種類によって変わる。

JSP を返す場合、`org.springframework.web.servlet.view.JstlView` が使用される。

Spring および依存ライブラリから提供されていない `View` を扱いたい場合、`View` インタフェースを実装したクラスを拡張する必要がある。

詳細はファイルダウンロードを参照されたい。


View
AbstractView
AbstractExcelView
AbstractFeedView<T>
AbstractAtomFeedView
AbstractRssFeedView
AbstractFileDownloadView
AbstractJackson2View
MappingJackson2JsonView
MappingJackson2XmlView
AbstractJExcelView
AbstractPdfView
AbstractUrlBasedView
AbstractJasperReportsView
AbstractJasperReportsSingleFormatView
ConfigurableJasperReportsView
JasperReportsCsvView
JasperReportsHtmlView
JasperReportsPdfView
JasperReportsXlsView
JasperReportsMultiFormatView
AbstractPdfStamperView
AbstractTemplateView
FreeMarkerView
GroovyMarkupView
VelocityView
VelocityToolboxView
VelocityLayoutView
InternalResourceView
JstlView
RedirectView
TilesView
TilesView
XsltView
MarshallingView
SmartView
RedirectView
new View() {...}
new View() {...}

2.3 はじめての Spring MVC アプリケーション

Spring MVC の、詳細な使い方の解説に入る前に、実際に Spring MVC に触れることで、Spring MVC を用いた Web アプリケーションの開発に対するイメージをつかむ。

2.3.1 検証環境

本節の説明では、次の環境で動作検証している。(他の環境で実施する際は、本書をベースに適宜読み替えて設定していくこと。)

種別	プロダクト
OS	Windows 7
JVM	Java 1.8
IDE	Spring Tool Suite 3.6.4.RELEASE (以降「STS」と呼ぶ)
Build Tool	Apache Maven 3.3.9 (以降「Maven」と呼ぶ)
Application Server	Pivotal tc Server Developer Edition v3.1 (STS に同封)
Web Browser	Google Chrome 46.0.2490.80 m

注釈: インターネット接続するために、プロキシサーバーを介する必要がある場合、以下の作業を行うため、STS の Proxy 設定と、 Maven の Proxy 設定が必要である。

2.3.2 新規プロジェクト作成

インターネットから *mvn archetype:generate* を利用して、プロジェクトを作成する。

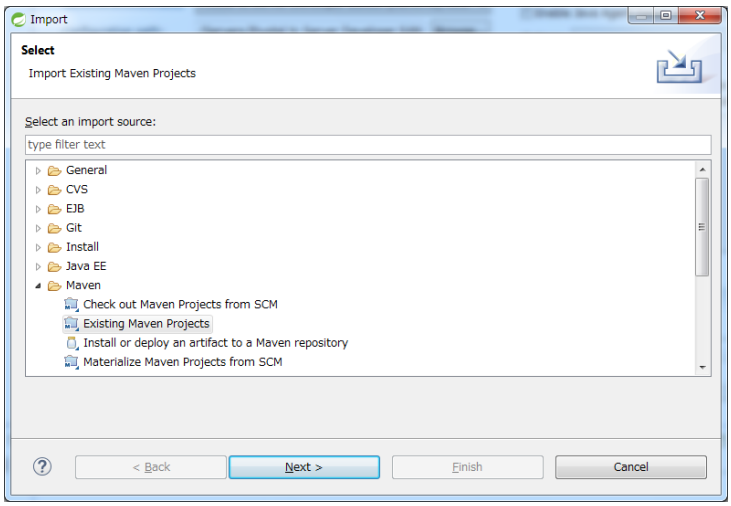
```
mvn archetype:generate -B^
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype^
 -DarchetypeVersion=5.4.2.RELEASE^
 -DgroupId=com.example.helloworld^
 -DartifactId=helloworld^
 -Dversion=1.0.0-SNAPSHOT
```

ここでは Windows 上にプロジェクトの元を作成する。

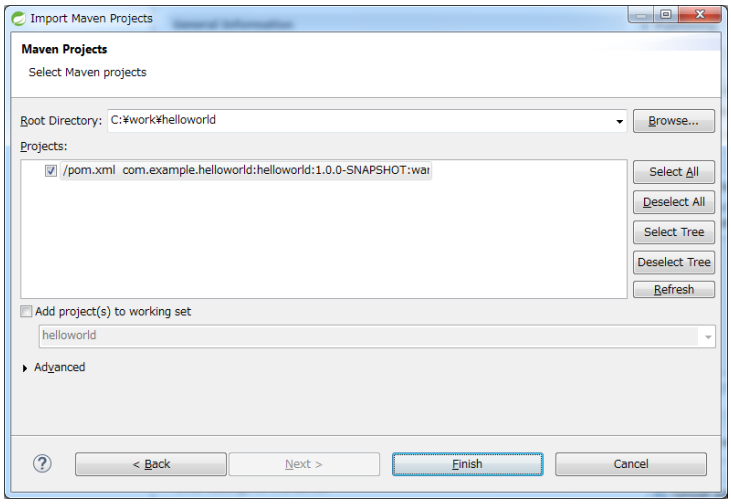
```
C:\work>mvn archetype:generate -B^
More?  -DarchetypeGroupId=org.terasoluna.gfw.blank^
More?  -DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype^
More?  -DarchetypeVersion=5.4.2.RELEASE^
More?  -DgroupId=com.example.helloworld^
More?  -DartifactId=helloworld^
More?  -Dversion=1.0.0-SNAPSHOT
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
```

```
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.4:generate (default-cli) > generate-sources @␣
↪standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.4:generate (default-cli) < generate-sources @␣
↪standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.4:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] Archetype repository not defined. Using the one from [org.terasoluna.gfw.
↪blank:terasoluna-gfw-web-blank-archetype:5.4.2.RELEASE] found in catalog remote
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: terasoluna-
↪gfw-web-blank-archetype:5.4.2.RELEASE
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.example.helloworld
[INFO] Parameter: artifactId, Value: helloworld
[INFO] Parameter: version, Value: 1.0.0-SNAPSHOT
[INFO] Parameter: package, Value: com.example.helloworld
[INFO] Parameter: packageInPathFormat, Value: com/example/helloworld
[INFO] Parameter: package, Value: com.example.helloworld
[INFO] Parameter: version, Value: 1.0.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.example.helloworld
[INFO] Parameter: artifactId, Value: helloworld
[INFO] project created from Archetype in dir: C:\work\helloworld
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 9.266 s
[INFO] Finished at: 2017-02-10T11:46:01+09:00
[INFO] Final Memory: 13M/188M
[INFO] ------------------------------------------------------------------------
C:\work>
```

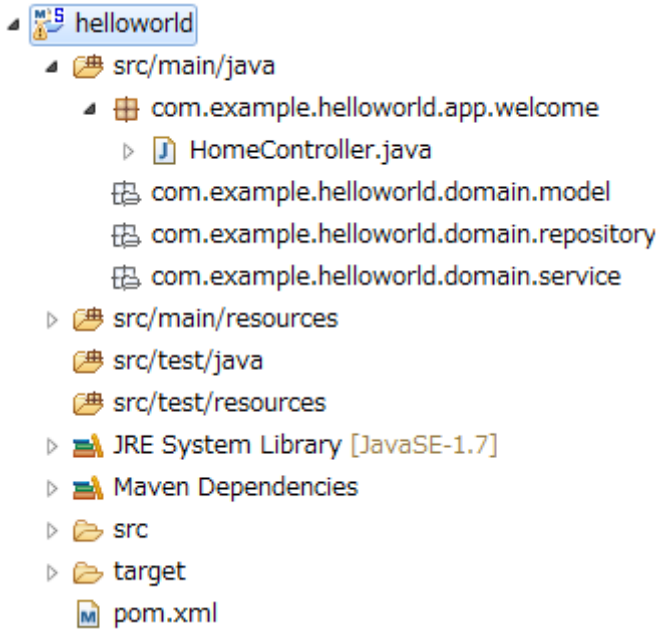
STS のメニューから、[File] -> [Import] -> [Maven] -> [Existing Maven Projects] -> [Next] を選択し、archetype で作成したプロジェクトを選択する。



Root Directory に `C:\work\helloworld` を設定し、Projects に helloworld の pom.xml が選択された状態で、 [Finish] を押下する。



Package Explorer に、次のようなプロジェクトが生成される。



Spring MVC の設定方法を理解するために、生成された Spring MVC の設定ファイル (src/main/resources/META-INF/spring/spring-mvc.xml) について、簡単に説明する。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc https://www.
↪springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
```

```
        http://www.springframework.org/schema/util https://www.springframework.org/
↪schema/util/spring-util.xsd
        http://www.springframework.org/schema/context https://www.springframework.
↪org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop https://www.springframework.org/
↪schema/aop/spring-aop.xsd
    ">

    <context:property-placeholder
        location="classpath*:/META-INF/spring/*.properties" />

    <!-- (1) Enables the Spring MVC @Controller programming model -->
    <mvc:annotation-driven>
        <mvc:argument-resolvers>
            <bean
                class="org.springframework.data.web.
↪PageableHandlerMethodArgumentResolver" />
            <bean
                class="org.springframework.security.web.method.annotation.
↪AuthenticationPrincipalArgumentResolver" />
        </mvc:argument-resolvers>
    </mvc:annotation-driven>

    <mvc:default-servlet-handler />

    <!-- (2) -->
    <context:component-scan base-package="com.example.helloworld.app" />

    <mvc:resources mapping="/resources/**"
        location="/resources/,classpath:META-INF/resources/"
        cache-period="#{60 * 60}" />

    <mvc:interceptors>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
                class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor" />
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
                class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenInterceptor" />
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean class="org.terasoluna.gfw.web.codelist.CodeListInterceptor">
                <property name="codeListIdPattern" value="CL_.+" />
            </bean>
        </mvc:interceptor>
```

```
        <!--  REMOVE THIS LINE IF YOU USE JPA
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
                class="org.springframework.orm.jpa.support.
↪OpenEntityManagerInViewInterceptor" />
        </mvc:interceptor>
            REMOVE THIS LINE IF YOU USE JPA  -->
    </mvc:interceptors>

    <!-- (3) Resolves views selected for rendering by @Controllers to .jsp␣
↪resources in the /WEB-INF/views directory -->
    <!-- Settings View Resolver. -->
    <mvc:view-resolvers>
        <mvc:jsp prefix="/WEB-INF/views/" />
    </mvc:view-resolvers>

    <bean id="requestDataValueProcessor"
        class="org.terasoluna.gfw.web.mvc.support.
↪CompositeRequestDataValueProcessor">
        <constructor-arg>
            <util:list>
                <bean
                    class="org.springframework.security.web.servlet.support.csrf.
↪CsrfRequestDataValueProcessor" />
                <bean
                    class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenRequestDataValueProcessor" />
            </util:list>
        </constructor-arg>
    </bean>

    <!-- Setting Exception Handling. -->
    <!-- Exception Resolver. -->
    <bean id="systemExceptionResolver"
        class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">
        <property name="exceptionCodeResolver" ref="exceptionCodeResolver" />
        <!-- Setting and Customization by project. -->
        <property name="order" value="3" />
        <property name="exceptionMappings">
            <map>
                <entry key="ResourceNotFoundException" value="common/error/
↪resourceNotFoundError" />
                <entry key="BusinessException" value="common/error/businessError" /
↪>
                <entry key="InvalidTransactionTokenException" value="common/error/
↪transactionTokenError" />
                <entry key=".DataAccessException" value="common/error/
↪dataAccessError" />
            </map>
        </property>
        <property name="statusCodes">
```

```
            <map>
                <entry key="common/error/resourceNotFoundError" value="404" />
                <entry key="common/error/businessError" value="409" />
                <entry key="common/error/transactionTokenError" value="409" />
                <entry key="common/error/dataAccessError" value="500" />
            </map>
        </property>
        <property name="excludedExceptions">
            <array>
                <value>org.springframework.web.util.NestedServletException</value>
            </array>
        </property>
        <property name="defaultErrorView" value="common/error/systemError" />
        <property name="defaultStatusCode" value="500" />
    </bean>
    <!-- Setting AOP. -->
    <bean id="handlerExceptionResolverLoggingInterceptor"
        class="org.terasoluna.gfw.web.exception.
↪HandlerExceptionResolverLoggingInterceptor">
        <property name="exceptionLogger" ref="exceptionLogger" />
    </bean>
    <aop:config>
        <aop:advisor advice-ref="handlerExceptionResolverLoggingInterceptor"
            pointcut="execution(* org.springframework.web.servlet.
↪HandlerExceptionResolver.resolveException(..))" />
    </aop:config>

</beans>
```

項番	説明
(1)	<mvc:annotation-driven>要素を定義することにより、Spring MVC のデフォルト設定が行われる。デフォルトの設定については、 Spring の公式ページである Enabling the MVC Java Config or the MVC XML Namespace を参照されたい。
(2)	Spring MVC で使用するコンポーネントを探すパッケージを定義する。
(3)	JSP 用の ViewResolver を指定し、JSP ファイルの配置場所を定義する。

次に、Welcome ページを表示するための Controller (com.example.helloworld.app.welcome.HelloController) について、簡単に説明する。

```
package com.example.helloworld.app.welcome;

import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 * Handles requests for the application home page.
 */
@Controller // (4)
public class HelloController {

    private static final Logger logger = LoggerFactory
            .getLogger(HelloController.class);

    /**
     * Simply selects the home view to render by returning its name.
     */
    @RequestMapping(value = "/", method = {RequestMethod.GET, RequestMethod.POST}
↪) // (5)
    public String home(Locale locale, Model model) {
        logger.info("Welcome home! The client locale is {}.", locale);

        Date date = new Date();
        DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG,
                DateFormat.LONG, locale);

        String formattedDate = dateFormat.format(date);

        model.addAttribute("serverTime", formattedDate); // (6)

        return "welcome/home"; // (7)
    }

}
```

項番	説明
(4)	`@Controller` アノテーションを付けることで、DI コンテナにより、コントローラクラスが自動で読み込まれる。前述「Spring MVC の設定ファイルの説明 (2)」の設定により、component-scan の対象となっている。
(5)	HTTP メソッドが GET または POST で、Resource（もしくは Request URL）が」/」で、アクセスする際に実行される。
(6)	View に渡したいオブジェクトを `Model` に設定する。
(7)	View 名を返却する。前述「Spring MVC の設定ファイルの説明 (3)」の設定により、」WEB-INF/views/welcome/home.jsp」がレンダリングされる。

最後に、Welcome ページを表示するための JSP (`src/main/webapp/WEB-INF/views/welcome/home.jsp`) について、簡単に説明する。

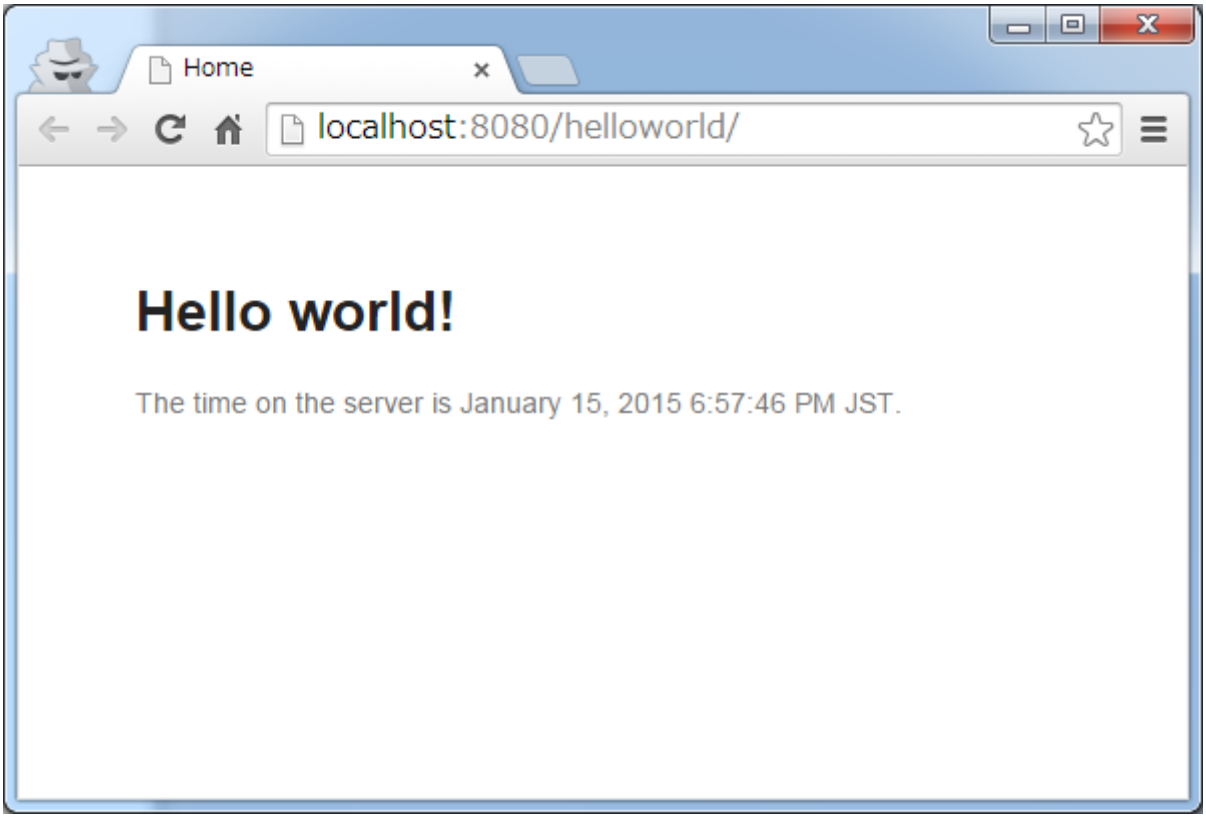
```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Home</title>
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/css/
↪styles.css">
</head>
<body>
    <div id="wrapper">
        <h1>Hello world!</h1>
        <p>The time on the server is ${serverTime}.</p> <%-- (8) --%>
    </div>
</body>
</html>
```

項番	説明
(8)	前述の「Controller の説明 (6)」で Model に設定したオブジェクト (serverTime) は、HttpServletRequest に格納される。そのため、JSP で`${serverTime}`と記述することで、Controller で設定した値を画面に出力することができる。 ただし、**${XXX}**の記述は、**XSS** 対象になる可能性があるので、文字列を出力する場合は **HTML** エスケープする必要がある。

2.3.3 サーバーを起動する

STS で、」helloworld」プロジェクトを右クリックして、」Run As」 -> 「Run On Server」 -> 「localhost」 -> 「Pivotal tc Server Developer Edition v3.0」 -> 「Finish」を実行し、helloworld プロジェクトを起動する。

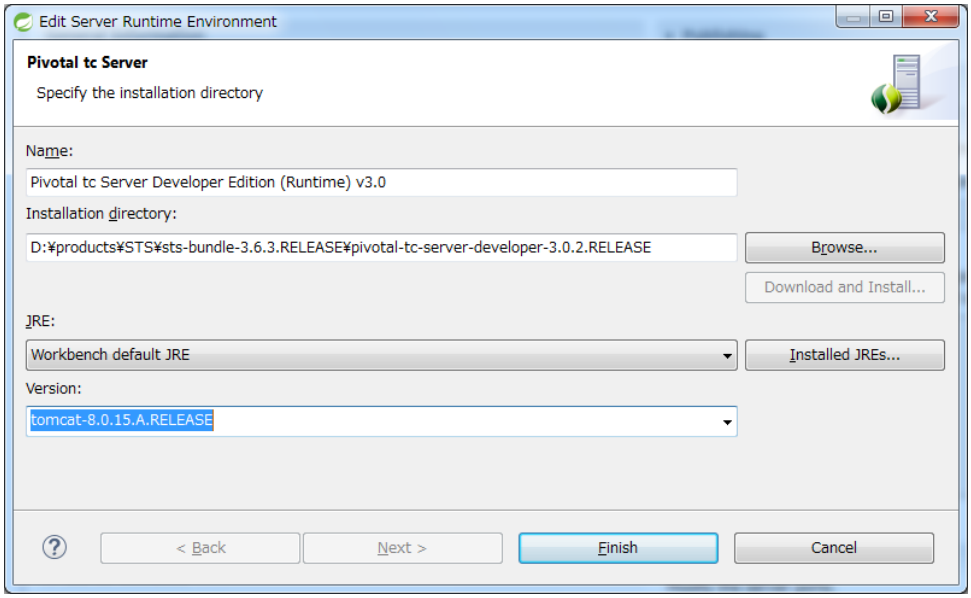
ブラウザに 「http://localhost:8080/helloworld/」 を入力し、実行すると下記の画面が表示される。



注釈: tc Server は内部で Tomcat を利用しており、動作検証で使用した STS では以下の 2 つのバージョンを選択する事ができる。

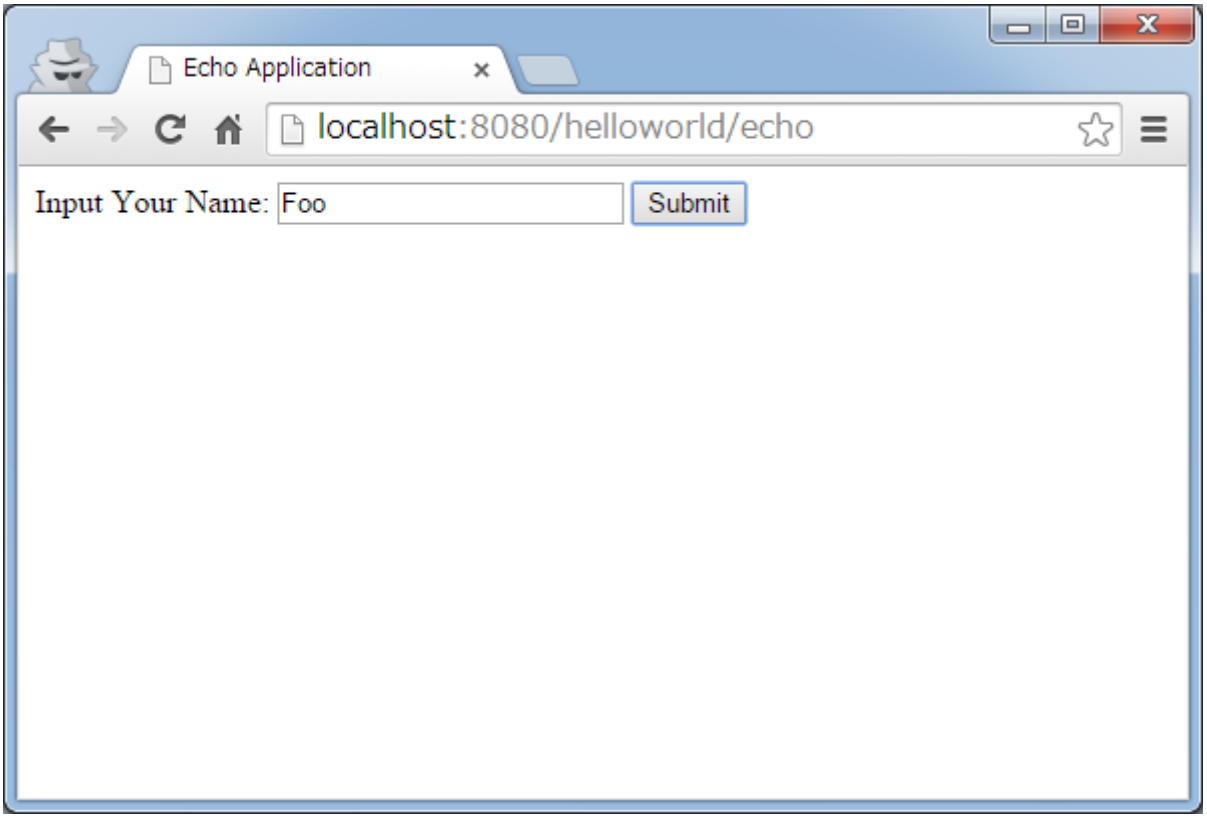
- tomcat-8.0.15.A.RELEASE (デフォルトで利用されるバージョン)
- tomcat-7-0.57.A.RELEASE

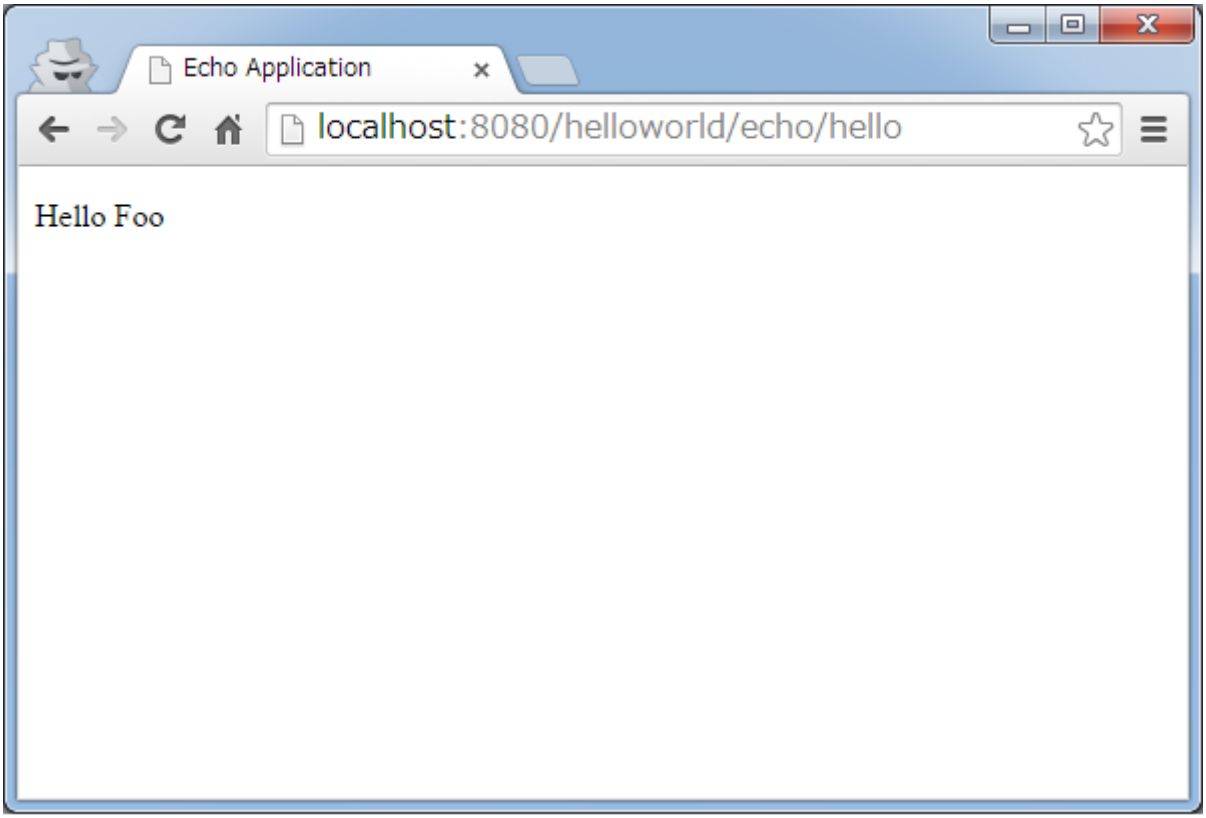
利用する Tomcat を切り替えたい場合は、ts Server の「Edit Server Runtime Environment」ダイアログを開き「Version」フィールドを変更すればよい。Java(JRE) のバージョンもこのダイアログから変更する事ができる。



2.3.4 エコーアプリケーションの作成

続いて、簡単なアプリケーションを作成する。作成するのは、次の図のようなテキストフィールドに、名前を入力するとメッセージを表示する、いわゆるエコーアプリケーションである。





フォームオブジェクトの作成

まずは、テキストフィールドの値を受け取るための、フォームオブジェクトを作成する。

`com.example.helloworld.app.echo` パッケージに `EchoForm` クラスを作成する。プロパティを 1 つだけ持つ、単純な JavaBean である。

```
package com.example.helloworld.app.echo;

import java.io.Serializable;

public class EchoForm implements Serializable {
    private static final long serialVersionUID = 2557725707095364445L;

    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Controller の作成

次に、Controller を作成する。

同じく `com.example.helloworld.app.echo` パッケージに、`EchoController` クラスを作成する。

```
package com.example.helloworld.app.echo;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("echo")
public class EchoController {

    @ModelAttribute // (1)
    public EchoForm setUpEchoForm() {
        EchoForm form = new EchoForm();
        return form;
    }

    @RequestMapping // (2)
    public String index(Model model) {
        return "echo/index"; // (3)
    }

    @RequestMapping(value = "hello", method = RequestMethod.POST) // (4)
    public String hello(EchoForm form, Model model) {// (5)
        model.addAttribute("name", form.getName()); // (6)
        return "echo/hello";
    }
}
```

項番	説明
(1)	`@ModelAttribute` というアノテーションを、メソッドに付加する。このアノテーションがついたメソッドの返り値は、自動で Model に追加される。 Model の属性名を、 `@ModelAttribute` で指定することもできるが、デフォルトでは、クラス名の先頭を小文字にした値が、属性名になる。この場合は、”echoForm”である。フォームの属性名は、次に説明する `form:form` **タグ** の `modelAttribute` 属性の値に一致している必要がある。
(2)	メソッドに付加した `@RequestMapping` アノテーションの `value` 属性に、何も指定しない場合、クラスに付加した `@RequestMapping` のルートに、マッピングされる。この場合、」<contextPath>/echo」にアクセスすると、 `index` メソッドが呼ばれる。 `method` 属性に何もしない場合は、任意の HTTP メソッドでマッピングされる。
(3)	View 名で」echo/index」を返すので、ViewResolver により、 「WEB-INF/views/echo/index.jsp」がレンダリングされる。
(4)	メソッドに付加した `@RequestMapping` アノテーションの `value` 属性に」hello」を、`method` 属性に `RequestMethod.POST` を指定しているので、この場合、」<contextPath>/echo/hello」に POST メソッドを使用してアクセスすると `hello` メソッドが呼ばれる。
(5)	引数に、EchoForm には (1) により Model に追加された EchoForm オブジェクトが渡される。
(6)	フォームで入力された `name` を、View にそのまま渡す。

注釈: `@RequestMapping` アノテーションの `method` 属性に指定する値は、クライアントから送信されたデータの扱い方によって変えるのが一般的である。

- データをサーバに保存する場合 (更新系の処理の場合) は、POST メソッド。
- データをサーバに保存しない場合 (参照系の処理の場合) は、GET メソッド又は未指定 (任意のメソッド)。

エコーアプリケーションでは、

- index メソッドはデータをサーバに保存しない処理なので未指定 (任意のメソッド)
- hello メソッドはデータを Model オブジェクトに保存する処理なので POST メソッド

を指定している。

JSP の作成

最後に、入力画面と、出力画面の JSP を作成する。それぞれのファイルパスは、View 名に合わせて、次のようになる。

入力画面 (src/main/webapp/WEB-INF/views/echo/index.jsp) を作成する。

```
<!DOCTYPE html>
<html>
<head>
<title>Echo Application</title>
</head>
<body>
  <%-- (1) --%>
  <form:form modelAttribute="echoForm" action="${pageContext.request.contextPath}/
↪echo/hello">
    <form:label path="name">Input Your Name:</form:label>
    <form:input path="name" />
    <input type="submit" />
  </form:form>
</body>
</html>
```

項番	説明
(1)	タグライブラリを利用し、HTML フォームを構築している。 modelAttribute 属性に、Controller で用意したフォームオブジェクトの名前を指定する。 タグライブラリは こちらを参照されたい。

注釈: <form:form>タグの method 属性を省略した場合は、POST メソッドが使用される。

出力される HTML は、

```
<!DOCTYPE html>
<html>
<head>
<title>Echo Application</title>
</head>
<body>
  <form id="echoForm" action="/helloworld/echo/hello" method="post">
    <label for="name">Input Your Name:</label>
    <input id="name" name="name" type="text" value=""/>
    <input type="submit" />
  <input type="hidden" name="_csrf" value="43595f38-3edd-4c08-843b-3c31a00d2b15" />
</form>
</body>
</html>
```

となる。

出力画面 (src/main/webapp/WEB-INF/views/echo/hello.jsp) を作成する。

```
<!DOCTYPE html>
<html>
<head>
<title>Echo Application</title>
</head>
<body>
  <p>
    Hello <c:out value="${name}" /> <%-- (2) --%>
  </p>
</body>
</html>
```

項番	説明
(2)	Controller から渡された」name」を出力する。 `c:out` タグにより、XSS 対策を行っている。

注釈: ここでは XSS 対策を標準タグの `c:out` で実現したが、より容易に使用できる `f:h()` 関数を共通ライブラリで用意している。詳細は、 *XSS 対策* を参照されたい。

これでエコーアプリケーションの実装は完了である。

サーバーを起動し、「http://localhost:8080/helloworld/echo」にアクセスするとフォームが表示される。

入力チェックの実装

ここまでのアプリケーションでは、入力チェックを行っていない。Spring MVC では、 Bean Validation をサポートしており、アノテーションベースな入力チェックを、簡単に実装することができる。例として、エコーアプリケーションで名前の入力チェックを行う。

EchoForm の name フィールドに、入力チェックルールを指定するアノテーションを付与する。

```
package com.example.helloworld.app.echo;

import java.io.Serializable;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class EchoForm implements Serializable {
    private static final long serialVersionUID = 2557725707095364445L;

    @NotNull // (1)
    @Size(min = 1, max = 5) // (2)
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

項番	説明
(1)	@NotNull アノテーションをつけることで、HTTP リクエスト中に name パラメータがあることを確認する。
(2)	@Size(min = 1, max = 5) をつけることで、name のサイズが、1 以上 5 以下であることを確認する。

入力チェックが実行されるように修正し、入力チェックでエラーが発生した場合の処理を実装する。

```
package com.example.helloworld.app.echo;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("echo")
public class EchoController {

    @ModelAttribute
    public EchoForm setUpEchoForm() {
        EchoForm form = new EchoForm();
        return form;
    }

    @RequestMapping
    public String index(Model model) {
        return "echo/index";
    }

    @RequestMapping(value = "hello", method = RequestMethod.POST)
    public String hello(@Validated EchoForm form, BindingResult result, Model␣
↪model) { // (1)
        if (result.hasErrors()) { // (2)
            return "echo/index";
        }
        model.addAttribute("name", form.getName());
        return "echo/hello";
    }
}
```

項番	説明
(1)	コントローラー側には、Validation 対象の引数に `@Validated` アノテーションを付加し、`BindingResult` オブジェクトを引数に追加する。 Bean Validation による入力チェックは、自動で行われる。結果は、`BindingResult` オブジェクトに渡される。
(2)	`hasErrors` メソッドを実行して、エラーがあるかどうかを確認する。入力エラーがある場合は、入力画面を表示するための View 名を返却する。

入力画面 (src/main/webapp/WEB-INF/views/echo/index.jsp) に、入力エラーのメッセージを表示するための実装を追加する。

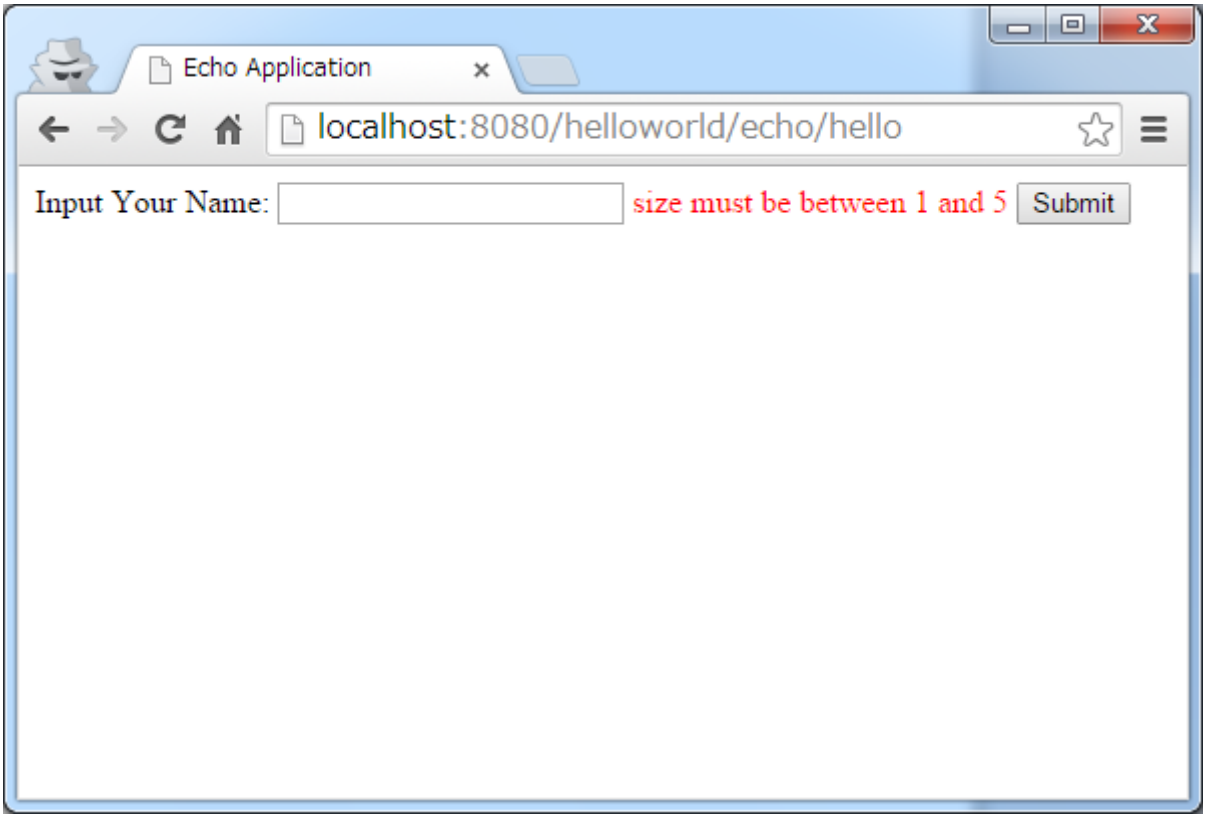
```
<!DOCTYPE html>
<html>
<head>
<title>Echo Application</title>
</head>
<body>
  <form:form modelAttribute="echoForm" action="${pageContext.request.contextPath}/
↪echo/hello">
    <form:label path="name">Input Your Name:</form:label>
    <form:input path="name" />
    <form:errors path="name" cssStyle="color:red" /><%-- (1) --%>
    <input type="submit" />
  </form:form>
</body>
</html>
```

項番	説明
(1)	入力画面には、エラーがあった場合に、エラーメッセージを表示するため、`form:errors` タグを追加する。

以上で、入力チェックの実装は完了である。

実際に、次のような場合、エラーメッセージが表示される。

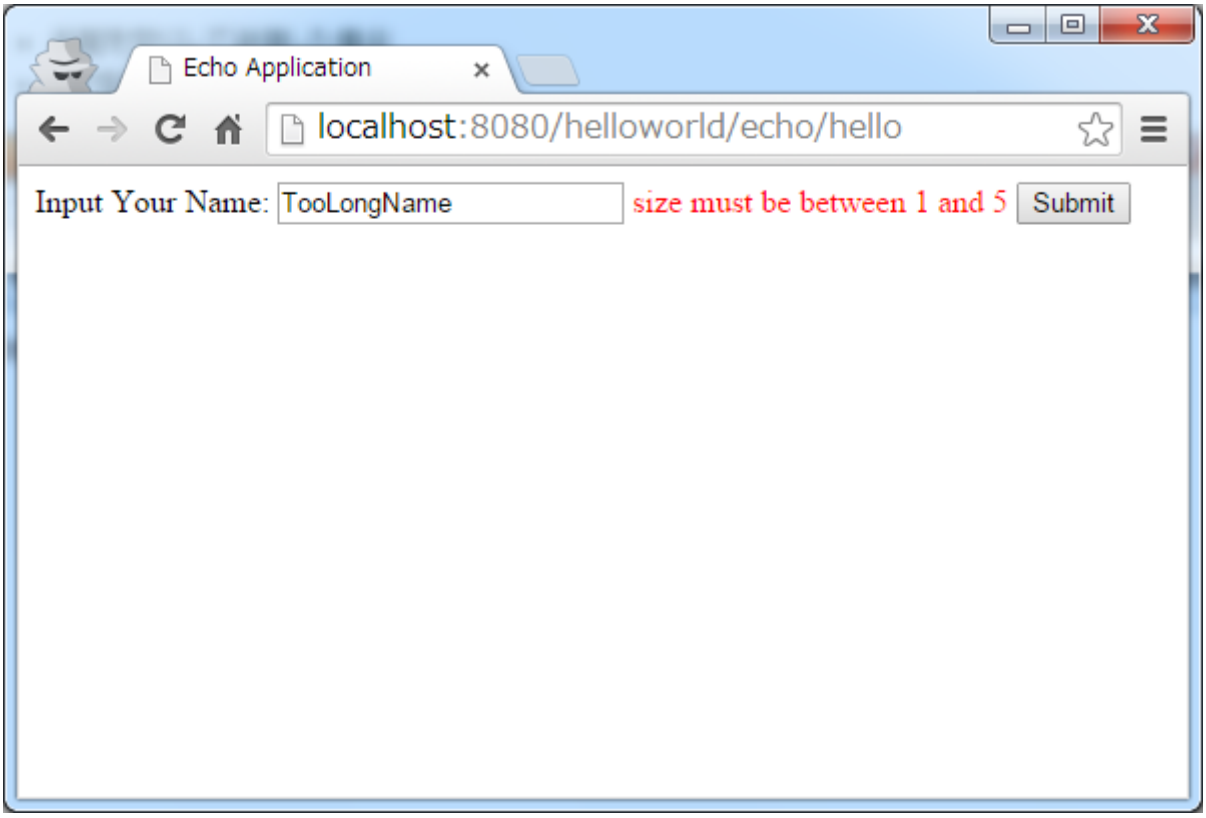
- 名前を空にして送信した場合
- 5 文字より大きいサイズで送信した場合



出力される HTML は、

```
<!DOCTYPE html>
<html>
<head>
<title>Echo Application</title>
</head>
<body>
  <form id="echoForm" action="/helloworld/echo/hello" method="post">
    <label for="name">Input Your Name:</label>
    <input id="name" name="name" type="text" value=""/>
    <span id="name.errors" style="color:red">size must be between 1 and 5</span>
    <input type="submit" />
  <input type="hidden" name="_csrf" value="6e94a78d-4a2c-4a41-a514-0a60f0dbedaf" />
</form>
</body>
</html>
```

となる。



まとめ

この章では、

1. `mvn archetype:generate` を利用したブランクプロジェクトの作成方法
2. Spring MVC の基本的な設定方法
3. 最も簡易な、画面遷移方法
4. 画面間での値の引き渡し方法
5. シンプルな入力チェック方法

を学んだ。

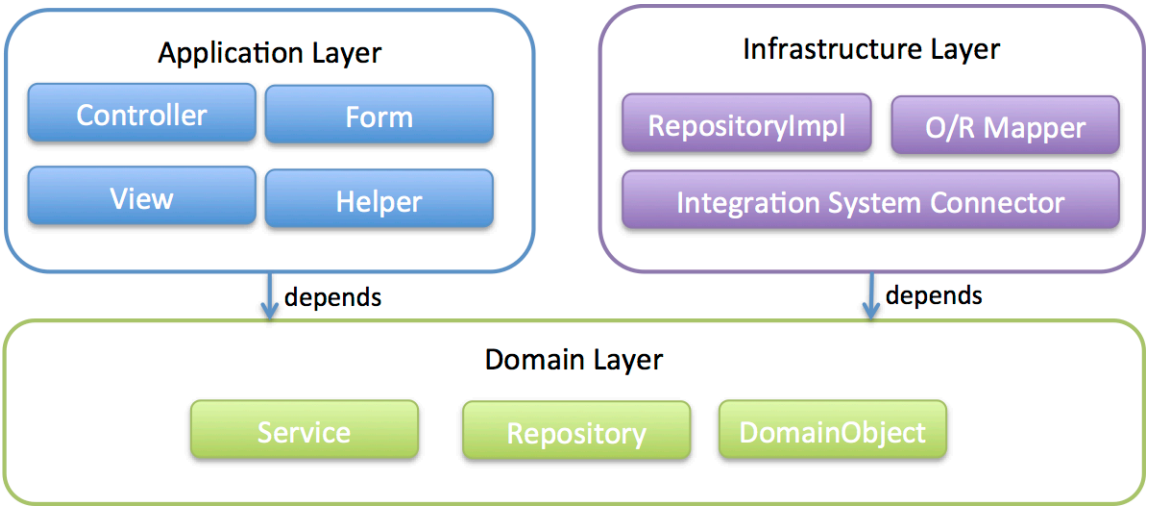
上記の内容が理解できていない場合は、もう一度、本節を読み、環境構築から始めて、進めていくことで理解が深まる。

2.4 アプリケーションのレイヤ化

本ガイドラインでは、アプリケーションを、次の 3 レイヤに分割する。

- アプリケーション層
- ドメイン層
- インフラストラクチャ層

各層には、以下のコンポーネントが含まれる。



アプリケーション層とインフラストラクチャ層は、ドメイン層に依存するが、**ドメイン層が、他の層に依存してはいけない。**
ドメイン層の変更によって、アプリケーション層に変更が生じるのは良いが、
アプリケーション層の変更によって、ドメイン層の変更が生じるべきではない。

各層について、説明する。

注釈: アプリケーション層、ドメイン層、インフラストラクチャー層は Eric Evans の」Domain-Driven Design (2004, Addison-Wesley)」で説明されている用語である。ただし、用語は使用しているが以後」Domain-Driven Design」の考えにのっとっているわけではない。

2.4.1 レイヤの定義

入力から出力までのデータの流れは、アプリケーション層 → ドメイン層 → インフラストラクチャ層であるため、この順に説明する。

アプリケーション層

アプリケーション層は、クライアントとのデータの入出力を制御する層である。

この層では、

- データの入出力を行う UI(User Interface) の提供
- クライアントからのリクエストハンドリング
- 入力データの妥当性チェック
- リクエスト内容に対応するドメイン層のコンポーネントの呼び出し

などの実装を行う。

この層で行う実装は、できるだけ薄く保たれるべきであり、ビジネスルールを含んではいけない。

Controller

Controller は、主に以下の役割を担う。

- 画面遷移の制御 (リクエストマッピングと処理結果に対応する View を返却する)
- ドメイン層の Service の呼び出し (リクエストに対応する主処理を実行する)

Spring MVC では、`@Controller` アノテーションが付与されている POJO クラスが該当する。

注釈: クライアントとの入出力データをセッションに格納する場合は、セッションに格納するデータのライフサイクルを制御する役割も担う。

View

View は、クライアントへの出力 (UI の提供を含む) を担う。HTML/PDF/Excel/JSON など、様々な形式で出力結果を返す。

Spring MVC では、`View` クラスが該当する。

ちなみに: REST API や Ajax 向けのリクエストで JSON や XML 形式の出力を行う場合は、`HttpMessageConverter` クラスが `View` の役割を担う。

詳細は、「*RESTful Web Service*」を参照されたい。

Form

Form は、主に以下の役割を担う。

- HTML のフォームを表現（フォームのデータを Controller に渡したり、処理結果をフォームに出力する）
- 入力チェックルールの宣言 (Bean Validation のアノテーションを付与する)

Spring MVC では、Form オブジェクトは、リクエストパラメータを保持する POJO クラスが該当する。form backing bean と呼ばれる。

注釈: ドメイン層がアプリケーション層に依存しないようにするために、以下の変換処理をアプリケーション層で行う。

- Form から Domain Object(Entity 等) への変換処理
- Domain Object から Form への変換処理

これらの変換処理を Controller 内で行うと、ソースコードが長くなり、本来の Controller の処理 (画面遷移など) の見通しが、悪くなりがちである。

変換処理のコードが多くなる場合は、以下のいずれか又は両方の対策を行い、Controller 内のソースコードをシンプルな状態に保つこと推奨する。

- Helper クラスを作成して変換処理を委譲する
- *Dozer* を使用する

ちなみに: REST API や Ajax 向けのリクエストで JSON や XML 形式の入力を受ける場合は、`Resource` クラスが `Form` の役割を担う。また、JSON や XML 形式の入力データを `Resource` クラスに変換する役割は、`HttpMessageConverter` クラスが担う。

詳細は、「*RESTful Web Service*」を参照されたい。

Helper

Helper は、Controller を補助する役割を担う。

Helper の作成はオプションである。必要に応じて、POJO クラスとして作成すること。

注釈: Controller の役割はルーティング (URL マッピングと遷移先の返却) であり、それ以外の処理 (JavaBean の変換等) が必要になったら Helper に切り出して、そちらに処理を委譲することを推奨する。

Helper は Controller の見通しを良くするためのものであるため、Helper は Controller の一部として扱ってよい。(Controller 内の private メソッドみたいなものである)

ドメイン層

ドメイン層は、アプリケーションのコアとなる層であり、ビジネスルールを実行 (業務処理を提供) する。

この層では、

- Domain Object
- Domain Object に対するビジネスルールのチェック (口座へ入金する場合に、残高が十分であるかどうかのチェックなど)
- Domain Object に対するビジネスルールの実行 (ビジネスルールに則った値の反映)
- Domain Object に対する CRUD 操作

などの実装を行う。

ドメイン層は、他の層からは疎であり、再利用できる。

Domain Object

Domain Object はビジネスを行う上で必要な資源や、ビジネスを行っていく過程で発生するものを表現するモデルである。

Domain Object は、大きく分けて、以下 3 つに分類される。

- Employee や Customer, Product などのリソース系モデル (一般的には、名詞で表現される)
- Order, Payment などイベント系モデル (一般的には動詞で表現される)
- YearlySales, MonthlySales などのサマリ系モデル

データベースのテーブルの 1 レコードを表現するクラスである Entity は、Domain Object である。

注釈: 本ガイドラインでは主に、状態のみもつモデルを扱う。

Martin Fowler の」Patterns of Enterprise Application Architecture (2002, Addison-Wesley)」では、Domain Model は、状態と振る舞いをもつものと定義されているが、厳密には触れない。

Eric Evans の提唱するような Rich なドメインモデルも、本ガイドラインでは扱わないが、分類上はここに含まれる。

Repository

Domain Object のコレクションのような位置づけであり、Domain Object の問い合わせや、作成、更新、削除のような CRUD 処理を担う。

この層では、インタフェースのみ定義する。

実体はインフラストラクチャ層の RepositoryImpl で実装するため、どのようなデータアクセスが行われているかについての情報は持たない。

Service

業務処理を提供する。

本ガイドラインでは、Service のメソッドをトランザクション境界にすることを推奨している。

注釈: Service では、Form や HttpRequest など、Web に関わる情報を扱うべきではない。

これらの情報は、Service のメソッドを呼び出す前に、アプリケーション層でドメイン層のオブジェクトに変換すべきである。

インフラストラクチャ層

インフラストラクチャ層は、ドメイン層 (Repository インタフェース) の実装を提供する層である。

データストア (RDBMS や、NoSQL などのデータを格納する場所) への永続化や、メッセージの送信などを担う。

RepositoryImpl

RepositoryImpl は、Repository インタフェースの実装として、Domain Object のライフサイクル管理を行う処理を提供する。

RepositoryImpl の実装は Repository インタフェースによって隠蔽されるため、ドメイン層のコンポーネント (Service など) では、どのようにデータアクセスされているか意識しなくて済む。

要件によっては、この処理もトランザクション境界となりうる。

ちなみに: MyBatis3 や Spring Data JPA を使用する場合は、RepositoryImpl の実体を (一部) 自動で作成する仕組みが提供されている。

O/R Mapper

O/R Mapper は、データベースと Entity の相互マッピングを担う。

MyBatis / JPA / Spring JDBC が、本機能を提供する。

具体的には、

- MyBatis3 を用いる場合は、Mapper インタフェースや `SqlSession`
- JPA を用いる場合は、`EntityManager`
- Spring JDBC を用いる場合は、`JdbcTemplate`

が、O/R Mapper に該当する。

O/R Mapper は、Repository インタフェースの実装に用いられる。

注釈: MyBatis, Spring JDBC は「O/R Mapper」というより、「SQL Mapper」と呼んだ方が正確であるが、本ガイドラインでは「O/R Mapper」に分類する。

Integration System Connector

Integration System Connector は、データベース以外のデータストア（メッセージングシステム、Key-Value-Store、Web サービス、既存システム、外部システムなど）との連携を担う。

Integration System Connector は、Repository インタフェースの実装に用いられる。

2.4.2 レイヤ間の依存関係

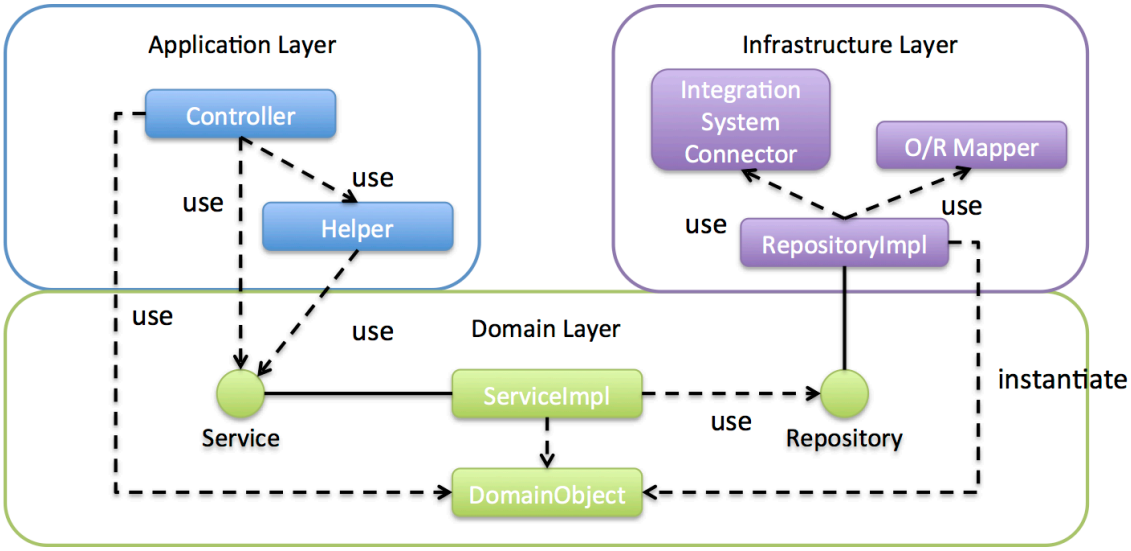
冒頭で説明したとおり、ドメイン層がコアとなり、アプリケーション層、インフラストラクチャ層がそれに依存する形となる。

本ガイドラインでは、実装技術として、

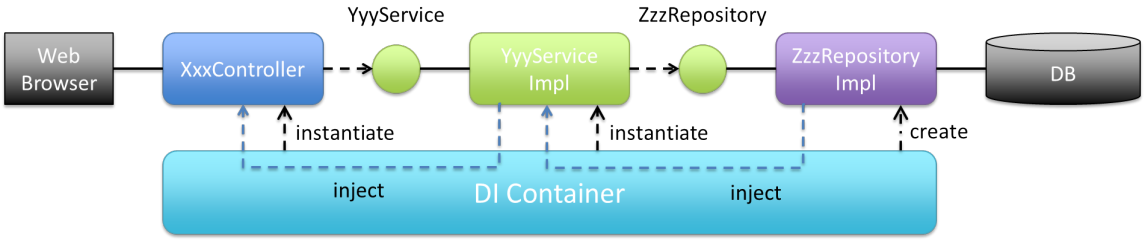
- アプリケーション層に Spring MVC
- インフラストラクチャ層に MyBatis, Spring Data JPA

を使用することを想定しているが、本質的には、実装技術が変わっても、それぞれの層で違いが吸収され、ドメイン層には影響を与えない。レイヤ間の結合部は、インタフェースとして公開することで、各層が使用している実装技術に依存しない形式とすることができる。

レイヤ化を意識して、疎結合な設計を行うことを推奨する。

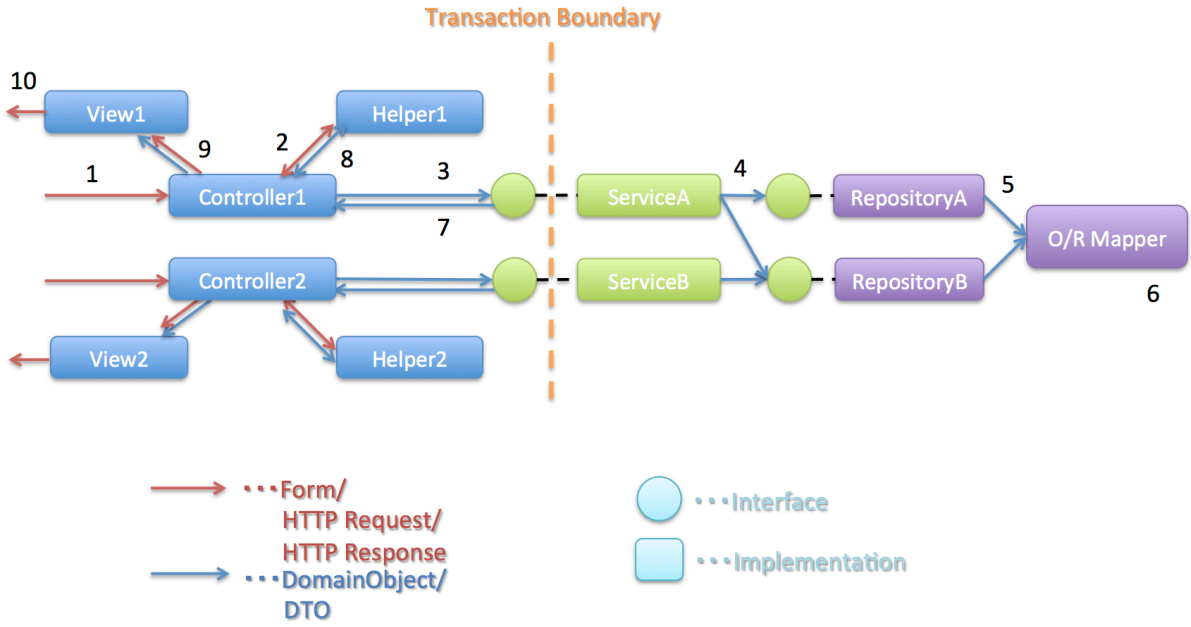


各レイヤのオブジェクトの依存関係は、DI コンテナによって解決される。



Repository を使用する時の処理の流れ

入力から出力までの流れで表現すると、次の図のようになる。



更新系の処理を例に、シーケンスを説明する。

項番	説明
1.	Controller が、Request を受け付ける
2.	(Optional) Controller は、Helper を呼び出し、Form の情報を、Domain Object または DTO に変換する
3.	Controller は、Domain Object または DTO を用いて、Service を呼び出す
4.	Service は、Repository を呼び出して、業務処理を行う
5.	Repository は、O/R Mapper を呼び出し、Domain Object または DTO を永続化する
6.	(実装依存) O/R Mapper は、DB に Domain Object または DTO の情報を保存する
7.	Service は、業務処理結果の Domain Object または DTO を、Controller に返却する
8.	(Optional) Controller は、Helper を呼び出し、Domain Object または DTO を、Form に変換する
9.	Controller は、遷移先の View 名を返却する
10.	View は、Response を出力する。

各コンポーネント間の呼び出し可否を、以下にまとめる。

表 2.3　コンポーネント間の呼び出し可否

Caller/Callee	Controller	Service	Repository	O/R Mapper
Controller	✗	✓	✗	✗
Service	✗	⚠	✓	✗
Repository	✗	✗	✗	✓

注意するべきことは、**基本的に Service から Service の呼び出しは、禁止している点である。**

もし他のサービスからも利用可能なサービスが必要な場合は、呼び出し可否を明確にするために、SharedService

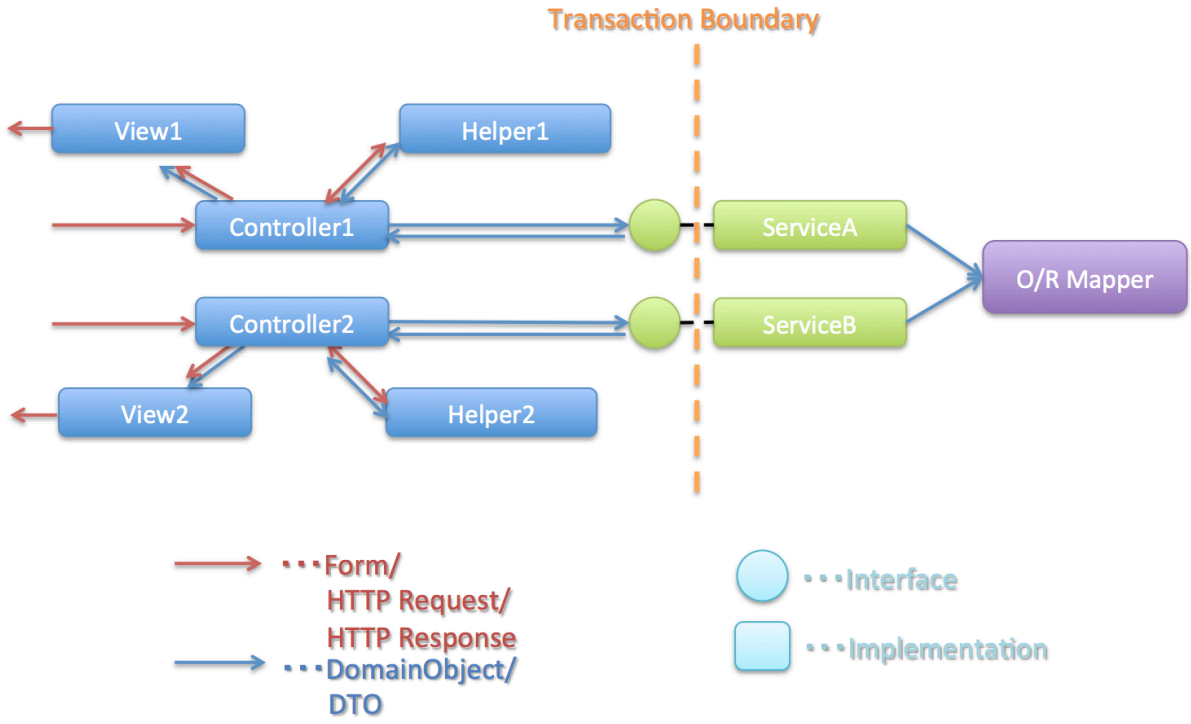
を作成すること。詳細については、ドメイン層の実装を参照されたい。

注釈: この呼び出し可否ルールを守ることは、アプリケーション開発の初期段階では、煩わしく感じられるかもしれない。確かに、一つの処理だけみると、たとえば Controller から直接 Repository を呼び出したほうが、速くアプリケーションを作成できる。しかし、ルールを守らない場合、開発規模が大きくなった際に、修正の影響範囲が分かりにくくなったり、横断的な共通処理を追加しにくくなるなど、保守性に大きな問題が生じることが多い。後で問題にならないように、初めから依存関係に気を付けて開発することを強く推奨する。

Repository を使用しない時の処理の流れ

Repository を作成することにより、永続化技術を隠蔽できたり、データアクセス処理を共通化できるなどのメリットがある。

しかし、プロジェクトのチーム体制によっては、データアクセスの共通化が難しい場合がある（複数の会社が、別々に業務処理を実装し、共通化のコントロールが難しい場合など）。その場合、データアクセスの抽象化が必要ないのであれば、Repository は作成せず、以下の図のように、Service から直接 O/R Mapper を呼び出すようにすればよい。



各コンポーネント間の呼び出し可否を、以下にまとめる。

表 2.4　コンポーネント間の呼び出し可否 (without Repository)

Caller/Callee	Controller	Service	O/R Mapper
Controller			
Service			

2.4.3 プロジェクト構成

上記のように、アプリケーションのレイヤ化を行った場合に推奨する構成について、説明する。

ここでは、Maven の標準ディレクトリ構造を前提とする。

基本的には、以下の構成でマルチプロジェクトを作成することを推奨する。

プロジェクト名	説明
[projectName]-domain	ドメイン層に関するクラス・設定ファイルを格納するプロジェクト
[projectName]-web	アプリケーション層に関するクラス・設定ファイルを格納するプロジェクト
[projectName]-env	環境に依存するファイル等を格納するプロジェクト

([projectName] には、対象のプロジェクト名を入れること)

注釈: RepositoryImpl などインフラストラクチャ層のクラスも、project-domain に含める。

本来は、[projectName]-infra プロジェクトを別途作成すべきであるが、通常 infra プロジェクトを隠蔽化する必要がなく、domain プロジェクトに格納されている方が開発しやすいためである。必要であれば、[projectName]-infra プロジェクトを作成してよい。

ちなみに: マルチプロジェクト構成の例として、サンプルアプリケーションや共通ライブラリのテストアプリケーションを参照されたい。

[projectName]-domain

[projectName]-domain のプロジェクト推奨構成を、以下に示す。

```
[projectName]-domain
  └src
      └main
          ├java
          |  └com
          |      └example
          |          └domain ...(1)
          |              ├model ...(2)
          |              |  ├Xxx.java
          |              |  ├Yyy.java
          |              |  └Zzz.java
          |              ├repository ...(3)
          |              |  ├xxx
          |              |  |  └XxxRepository.java
          |              |  ├yyy
          |              |  |  └YyyRepository.java
          |              |  └zzz
          |              |     ├ZzzRepository.java
          |              |     └ZzzRepositoryImpl.java
          |              └service ...(4)
          |                  ├aaa
          |                  |  ├AaaService.java
          |                  |  └AaaServiceImpl.java
          |                  └bbb
          |                     ├BbbService.java
          |                     └BbbServiceImpl.java
          └resources
              └META-INF
                  └spring
                      ├[projectName]-codelist.xml ...(5)
                      ├[projectName]-domain.xml ...(6)
                      └[projectName]-infra.xml ...(7)
```

項番	説明
(1)	ドメイン層の構成要素を格納するパッケージ。
(2)	Domain Object を格納するパッケージ。
(3)	リポジトリを格納するパッケージ。 エンティティごとにパッケージを作成する。関連するエンティティがあれば、主となるエンティティのパッケージに、従となるエンティティ (Order と OrderLine の関係であれば OrderLine) の Repository も配置する。また、検索条件などを保持する DTO などが必要な場合は、このパッケージに配置する。 RepositoryImpl は、インフラストラクチャ層に属するが、通常、このプロジェクトに含めても問題ない。異なるデータストアを使うなど、複数の永続化先があり、実装を隠蔽したい場合は、別プロジェクト (またはパッケージ) に、RepositoryImpl を実装するようにする。
(4)	サービスを格納するパッケージ。 業務 (またはエンティティ) ごとに、パッケージインタフェースと実装を、同じ階層に配置する。入出力クラスが必要な場合は、このパッケージに配置する。
(5)	コードリストの Bean 定義を行う。
(6)	ドメイン層に関する Bean 定義を行う。
(7)	インフラストラクチャ層に関する Bean 定義を行う。

[projectName]-web

[projectName]-web のプロジェクト推奨構成を、以下に示す。

```
[projectName]-web
  └src
      └main
          ├java
          |  └com
          |      └example
```

```
        |           └app ...(1)
        |               ├abc
        |               |   ├AbcController.java
        |               |   ├AbcForm.java
        |               |   └AbcHelper.java
        |               └def
        |                   ├DefController.java
        |                   ├DefForm.java
        |                   └DefOutput.java
        ├resources
        |   ├META-INF
        |   |   └spring
        |   |       ├applicationContext.xml ...(2)
        |   |       ├application.properties ...(3)
        |   |       ├spring-mvc.xml ...(4)
        |   |       └spring-security.xml ...(5)
        |   └i18n
        |       └application-messages.properties ...(6)
        └webapp
            ├resources ...(7)
            └WEB-INF
                ├views ...(8)
                |   ├abc
                |   |   ├list.jsp
                |   |   └createForm.jsp
                |   └def
                |       ├list.jsp
                |       └createForm.jsp
                └web.xml ...(9)
```

項番	説明
(1)	アプリケーション層の構成要素を格納するパッケージ。
(2)	アプリケーション全体に関する Bean 定義を行う。
(3)	アプリケーションで使用するプロパティを定義する。
(4)	Spring MVC の設定を行う Bean 定義を行う。
(5)	SpringSecurity の設定を行う Bean 定義を行う。
(6)	画面表示用のメッセージ (国際化対応) 定義を行う。
(7)	静的リソース (css、js、画像など) を格納する。
(8)	View(jsp) を格納する。
(9)	Servlet のデプロイメント定義を行う。

[projectName]-env

[projectName]-env のプロジェクト推奨構成を、以下に示す。

```
[projectName]-env
  ├configs ...(1)
  |    └[envName] ...(2)
```

```
|       └resources ...(3)
└src
    └main
        └resources ...(4)
           ├META-INF
           |   └spring
           |       ├[projectName]-env.xml ...(5)
           |       └[projectName]-infra.properties ...(6)
           ├dozer.properties
           ├log4jdbc.properties
           └logback.xml ...(7)
```

項番	説明
(1)	全環境の環境依存ファイルを管理するためのディレクトリ。
(2)	環境毎の環境依存ファイルを管理するためのディレクトリ。 ディレクトリ名は、環境を識別する名前を指定する。
(3)	環境毎の設定ファイルを管理するためのディレクトリ。 サブディレクトリの構成や管理する設定ファイルは、(4) と同様。
(4)	ローカル開発環境用の設定ファイルを管理するためのディレクトリ。
(5)	ローカル開発環境用の Bean 定義 (DataSource 等) を行う。
(6)	ローカル開発環境用のプロパティを定義する。
(7)	ローカル開発環境用のログ出力定義を行う。

注釈: [projectName]-domain と [projectName]-web を別プロジェクトに分ける理由は、依存関係の逆転を防ぐためである。

[projectName]-web が [projectName]-domain を使用するのは当然であるが、[projectName]-domain が [projectName]-web を参照してはいけない。

1 つのプロジェクトに [projectName]-web と [projectName]-domain の構成要素をまとめてしまうと、誤って不正な参照をしてしまうことがある。プロジェクトを分けて参照順序をつけることで [projectName]-domain が [projectName]-web を参照できないようにすることを強く推奨する。

注釈: [projectName]-env を作成する理由は環境に依存する情報を外出し、環境毎に切り替えられるようにするためである。

たとえばデフォルトではローカル開発環境用の設定をして、アプリケーションビルド時には [projectName]-env を除いて war を作成する。結合テスト用の環境やシステムテスト用の環境を別々の jar として作成すると、そこだけ差し替えてデプロイするということが可能である。

また使用する RDBMS が変わるようなプロジェクトの場合にも影響を最小限に抑えることができる。

この点を考慮しない場合は、環境ごとに設定ファイルの内容を行いビルドしなおすという作業が入る。

第 3 章

アプリケーション開発

TERASOLUNA Server Framework for Java (5.x) を使用する上での各種ルールや推奨実装方法を記述する。

本ガイドラインでは以下のような開発の流れを想定している。

3.1 Web アプリケーション向け開発プロジェクトの作成

本節では、Web アプリケーション向けの開発プロジェクトを作成する方法について説明する。

本ガイドラインでは、マルチプロジェクト構成を採用することを推奨している。推奨するマルチプロジェクト構成の説明については、「プロジェクト構成」を参照されたい。

3.1.1 ブランクプロジェクトの種類

ブランクプロジェクトは、使用用途に応じて以下の 2 種類を提供している。

種別	使用用途
マルチプロジェクト構成のブランクプロジェクト	商用環境にリリースするような本格的なアプリケーションを開発する際に使用する。 プロジェクトの雛形は、Maven の Archetype として、以下の 2 種類を用意している。 • MyBatis3 用の設定が盛り込まれた雛形 • JPA(Spring Data JPA) 用の設定が盛り込まれた雛形 **本ガイドラインでは、マルチプロジェクト構成のプロジェクトを使用する事を推奨している。**
シングルプロジェクト構成のブランクプロジェクト	POC(Proof Of Concept)、プロトタイプ、サンプルなどの簡易的なアプリケーションを作成する際に使用する。 プロジェクトの雛形は、Maven の Archetype として、以下の 3 種類を用意している。 (Eclipse の WTP 用のプロジェクトも用意しているが、本節では説明は割愛する) • MyBatis3 用の設定が盛り込まれた雛形 • JPA(Spring Data JPA) 用の設定が盛り込まれた雛形 • O/R Mapper に依存しない雛形 本ガイドラインでは、各種チュートリアルについてシングルプロジェクトを使用して行う手順となっている。

3.1.2 開発プロジェクトの作成

マルチプロジェクト構成の開発プロジェクトを、Maven Archetype Plugin の archetype:generate を使用して作成する。

注釈: 前提条件

以降の説明では、

- Maven (mvn コマンド) が使用可能であること
- インターネットに繋がっていること
- インターネットにプロキシ経由で繋ぐ場合は、Maven のプロキシ設定 が行われていること

を前提としている。

前提条件が整っていない場合は、まずこれらのセットアップを行ってほしい。

マルチプロジェクトを作成するための Archetype として、以下の 2 種類を用意している。

項番	Archetype(ArtifactId)	説明
1.	terasoluna-gfw-multi-web-blank-mybatis3-archetype	O/R Mapper として MyBatis3 を使用するためのプロジェクトを生成するための Archetype。
2.	terasoluna-gfw-multi-web-blank-jpa-archetype	O/R Mapper として JPA(with Spring Data JPA and Hibernate) を使用するためのプロジェクトを生成するための Archetype。

プロジェクトを作成するフォルダに移動する。

```
cd C:\work
```

Maven Archetype Plugin の archetype:generate を使用して、プロジェクトを作成する。

```
mvn archetype:generate -B^
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-multi-web-blank-mybatis3-archetype^
 -DarchetypeVersion=5.4.2.RELEASE^
 -DgroupId=com.example.todo^
 -DartifactId=todo^
 -Dversion=1.0.0-SNAPSHOT
```

パラメータ	説明
-B	batch mode (対話を省略)
-DarchetypeGroupId	ブランクプロジェクトの groupId を指定する。(固定)
-DarchetypeArtifactId	ブランクプロジェクトの archetypeId(雛形を特定するための ID) を指定する。**(カスタマイズが必要)** 以下の何れかの archetypeId を指定する。 • `terasoluna-gfw-multi-web-blank-mybatis3-archetype` • `terasoluna-gfw-multi-web-blank-jpa-archetype` 上記例では、`terasoluna-gfw-multi-web-blank-mybatis3-archetype` を指定している。
-DarchetypeVersion	ブランクプロジェクトのバージョンを指定する。(固定)
-DgroupId	作成するプロジェクトの groupId を指定する。**(カスタマイズが必要)** 上記例では、`com.example.todo` を指定している。
-DartifactId	作成するプロジェクトの artifactId を指定する。**(カスタマイズが必要)** 上記例では、`todo` を指定している。
-Dversion	作成するプロジェクトのバージョンを指定する。**(カスタマイズが必要)** 上記例では、`1.0.0-SNAPSHOT` を指定している。

プロジェクトの作成が成功した場合、以下のようなログが出力される。(以下は、MyBatis3 用の Archetype を使用して作成した場合の出力例)

```
(... omit)
[INFO] ------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: terasoluna-
↪gfw-multi-web-blank-mybatis3-archetype:5.4..RELEASE
[INFO] ------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.example.todo
[INFO] Parameter: artifactId, Value: todo
[INFO] Parameter: version, Value: 1.0.0-SNAPSHOT
[INFO] Parameter: package, Value: com.example.todo
[INFO] Parameter: packageInPathFormat, Value: com/example/todo
[INFO] Parameter: package, Value: com.example.todo
[INFO] Parameter: version, Value: 1.0.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.example.todo
[INFO] Parameter: artifactId, Value: todo
[INFO] Parent element not overwritten in C:\work\todo\todo-env\pom.xml
[INFO] Parent element not overwritten in C:\work\todo\todo-domain\pom.xml
[INFO] Parent element not overwritten in C:\work\todo\todo-web\pom.xml
[INFO] Parent element not overwritten in C:\work\todo\todo-initdb\pom.xml
[INFO] Parent element not overwritten in C:\work\todo\todo-selenium\pom.xml
[INFO] project created from Archetype in dir: C:\work\todo
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 9.929 s
[INFO] Finished at: 2015-07-31T12:03:21+00:00
[INFO] Final Memory: 10M/26M
[INFO] ------------------------------------------------------------------------
```

プロジェクトの作成が成功した場合、Maven のマルチプロジェクトが作成される。Maven Archetype で作成したプロジェクトの詳細な説明については、「開発プロジェクトの構成」を参照されたい。

```
todo
├── pom.xml
├── todo-domain
├── todo-env
├── todo-initdb
├── todo-selenium
└── todo-web
```

3.1.3 開発プロジェクトのカスタマイズ

Maven Archetype で作成したプロジェクトには、アプリケーション毎にカスタマイズが必要な箇所がいくつか存在する。

カスタマイズが必要な箇所を以下に示す。

- *POM* ファイルのプロジェクト情報
- *x.xx.fw.9999* 形式のメッセージ *ID*
- メッセージ文言
- エラー画面
- 画面フッターの著作権
- インメモリデータベース *(H2 Database)*
- データソース設定

注釈: 上記以外のカスタマイズポイントとしては、

- 認証・認可 の設定
- ファイルアップロード を有効化するための設定
- 国際化 を有効化するための設定
- ロギング の定義
- 例外ハンドリング の定義
- *RESTful Web Service* 向けの設定の適用

などがある。

これらのカスタマイズについては、各節の How to use を参照し、必要に応じてカスタマイズしてほしい。

注釈: 以降の説明で `artifactId` と表現している部分は、プロジェクト作成時に指定した `artifactId` に置き換えて読み進めてほしい。

POM ファイルのプロジェクト情報

Maven Archetype で作成したプロジェクトの POM ファイルでは、

- プロジェクト名 (`name` 要素)

- プロジェクト説明 (description 要素)
- プロジェクト URL(url 要素)
- プロジェクト創設年 (inceptionYear 要素)
- プロジェクトライセンス (licenses 要素)
- プロジェクト組織 (organization 要素)

といったプロジェクト情報が、Archetype 自身のプロジェクト情報が設定されている状態となっている。実際の設定内容を以下に示す。

```
<!-- ... -->

<name>TERASOLUNA Server Framework for Java (5.x) Web Blank Multi Project</name>
<description>Web Blank Multi Project using TERASOLUNA Server Framework for Java (5.
↪x)</description>
<url>http://terasoluna.org</url>
<inceptionYear>2014</inceptionYear>
<licenses>
    <license>
        <name>Apache License, Version 2.0</name>
        <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
        <distribution>manual</distribution>
    </license>
</licenses>
<organization>
    <name>TERASOLUNA Framework Team</name>
    <url>http://terasoluna.org</url>
</organization>

<!-- ... -->
```

注釈: プロジェクト情報には、適切な値を設定すること。

カスタマイズ対象のファイルとカスタマイズ方法を以下に示す。

項番	対象ファイル	カスタマイズ方法
1.	マルチプロジェクト全体の構成を定義する POM(Project Object Model) ファイル artifactId/pom.xml	プロジェクト情報に適切な値を指定する。

x.xx.fw.9999 形式のメッセージ ID

Maven Archetype で作成したプロジェクトでは、x.xx.fw.9999 形式のメッセージ ID を、

- エラー画面に表示するメッセージ
- 例外発生時に出力するエラーログ

を生成する際に使用している。実際の使用箇所 (サンプリング) を以下に示す。

[application-messages.properties]

```
e.xx.fw.5001 = Resource not found.
```

[JSP]

```
<div class="error">
    <c:if test="${!empty exceptionCode}">[${f:h(exceptionCode)}]</c:if>
    <spring:message code="e.xx.fw.5001" />
</div>
```

[applicationContext.xml]

```
<bean id="exceptionCodeResolver"
    class="org.terasoluna.gfw.common.exception.SimpleMappingExceptionCodeResolver">
    <!-- ... -->
            <entry key="ResourceNotFoundException" value="e.xx.fw.5001" />
    <!-- ... -->
</bean>
```

x.xx.fw.9999 形式のメッセージ ID は、本ガイドラインの「メッセージ管理」で紹介しているメッセージ ID 体系であるが、プロジェクト区分の値が暫定値「xx」の状態になっている。

注釈:

- 本ガイドラインで紹介しているメッセージ ID 体系を利用する場合は、プロジェクト区分に適切な値を指定すること。 本ガイドラインで紹介しているメッセージ ID 体系については、「結果メッセージ」を参照されたい。
- 本ガイドラインで紹介しているメッセージ ID 体系を利用しない場合は、以下に示す修正対象ファイル内で使用しているメッセージ ID を全て置き換える必要がある。

カスタマイズ対象のファイルとカスタマイズ方法を以下に示す。

項番	対象ファイル	カスタマイズ方法
1.	メッセージ定義ファイル `artifactId/artifactId-web/src/main/resources/i18n/application-messages.properties`	プロパティキーに指定しているメッセージ ID のプロジェクト区分の暫定値「xx」を、適切な値に修正する。
2.	エラー画面用の JSP `artifactId/artifactId-web/src/main/webapp/WEB-INF/views/common/error/*.jsp`	`<spring:message>`要素の `code` 属性に指定しているメッセージ ID のプロジェクト区分の暫定値「xx」を、適切な値に修正する。
3.	Web アプリケーション用のアプリケーションコンテキストを作成するための Bean 定義ファイル `artifactId/artifactId-web/src/main/resources/META-INF/spring/applicationContext.xml`	BeanID が `exceptionCodeResolver` の Bean 定義内で指定している例外コード (メッセージ ID) のプロジェクト区分の暫定値「xx」を、適切な値に修正する。

メッセージ文言

Maven Archetype で作成したプロジェクトでは、いくつかのメッセージ定義を提供しているが、メッセージ文言は簡易的なメッセージになっている。実際のメッセージ (サンプリング) を以下に示す。

[application-messages.properties]

```
e.xx.fw.5001 = Resource not found.

# ...

# typemismatch
typeMismatch="{0}" is invalid.

# ...
```

注釈: メッセージ文言については、アプリケーション要件 (メッセージ規約など) に合わせて修正すること。

カスタマイズ対象のファイルとカスタマイズ方法を以下に示す。

項番	対象ファイル	カスタマイズ方法
1.	メッセージ定義ファイル `artifactId/artifactId-web/src/main/resources/i18n/application-messages.properties`	アプリケーション要件に応じたメッセージに修正する。 入力チェックでエラーとなった際に表示するメッセージ (Bean Validation のメッセージ) についても、アプリケーション要件に応じて修正 (デフォルトメッセージの上書き) が必要になる。デフォルトメッセージの上書き方法については、「エラーメッセージの定義」を参照されたい。

エラー画面

Maven Archetype で作成したプロジェクトでは、エラーの種類毎にエラー画面を表示するための JSP 及び HTML を提供しているが、

- 画面レイアウト
- 画面タイトル
- メッセージの文言

などが簡易的な実装になっている。実際の JSP の実装 (サンプリング) を以下に示す。

[JSP]

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Resource Not Found Error!</title>
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/css/
↪styles.css">
</head>
<body>
    <div id="wrapper">
        <h1>Resource Not Found Error!</h1>
        <div class="error">
            <c:if test="${!empty exceptionCode}">[${f:h(exceptionCode)}]</c:if>
            <spring:message code="e.xx.fw.5001" />
        </div>
        <t:messagesPanel />
        <br>
        <!-- ... -->
        <br>
```

```
    </div>
</body>
</html>
```

注釈: エラー画面を表示するための JSP と HTML については、アプリケーション要件 (UI 規約など) に合わせて修正すること。

カスタマイズ対象のファイルとカスタマイズ方法を以下に示す。

項番	対象ファイル	カスタマイズ方法
1.	エラー画面用の JSP `artifactId/artifactId-web/src/main/webapp/WEB-INF/views/common/error/*.jsp`	アプリケーション要件 (UI 規約など) に合わせて修正する。 エラー画面を表示する JSP をカスタマイズする際は、「例外ハンドリング の コーディングポイント（*JSP* 編）」を参照されたい。
2.	エラー画面用の HTML `artifactId/artifactId-web/src/main/webapp/WEB-INF/views/common/error/unhandledSystemError.html`	アプリケーション要件 (UI 規約など) に合わせて修正する。

画面フッターの著作権

Maven Archetype で作成したプロジェクトでは、Tiles を使用して画面レイアウトを構成しているが、画面フッター部の著作権が暫定値「`Copyright © 20XX CompanyName`」の状態になっている。実際の JSP の実装 (サンプリング) を以下に示す。

[template.jsp]

```
<div class="container">
    <tiles:insertAttribute name="header" />
    <tiles:insertAttribute name="body" />
    <hr>
    <p style="text-align: center; background: #e5eCf9;">Copyright &copy; 20XX␣
↪CompanyName</p>
</div>
```

注釈: **Tiles** を使用して画面レイアウトを構成する場合は、著作権に適切な値を指定すること。

カスタマイズ対象のファイルとカスタマイズ方法を以下に示す。

項番	対象ファイル	カスタマイズ方法
1.	Tiles 用のテンプレート JSP `artifactId/artifactId-web/src/main/webapp/WEB-INF/views/layout/template.jsp`	著作権の暫定値「`Copyright © 20XX CompanyName`」を適切な値に修正する。

インメモリデータベース (H2 Database)

Maven Archetype で作成したプロジェクトには、インメモリデータベース (H2 Database) をセットアップするための設定が行われているが、これはちょっとした動作検証 (プロトタイプ作成や POC(Proof Of Concept)) を行うための設定である。そのため、本格的なアプリケーション開発を行う場合は、不要な設定になる。

[artifactId-env.xml]

```
<jdbc:initialize-database data-source="dataSource"
    ignore-failures="ALL">
    <jdbc:script location="classpath:/database/${database}-schema.sql" encoding=
↪"UTF-8" />
    <jdbc:script location="classpath:/database/${database}-dataload.sql" encoding=
↪"UTF-8" />
</jdbc:initialize-database>
```

```
└── src
    └── main
        └── resources
            ├── META-INF
          (...)
            ├── database
            │   ├── H2-dataload.sql
            │   └── H2-schema.sql
```

注釈: 本格的なアプリケーション開発を行う場合は、インメモリデータベース (H2 Database) をセットアッ

プするための定義と SQL を管理するためのディレクトリを削除すること。

カスタマイズ対象のファイルとカスタマイズ方法を以下に示す。

項番	対象ファイル	カスタマイズ方法
1.	環境依存するコンポーネントを定義する Bean 定義ファイル `artifactId-env/src/main/resources/META-INF/spring/artifactId-env.xml`	`<jdbc:initialize-database>`要素を削除する。
2.	インメモリデータベース (H2 Database) をセットアップするための SQL を格納するディレクトリ `artifactId/artifactId-env/src/main/resources/database/`	ディレクトリを削除する。

データソース設定

Maven Archetype で作成したプロジェクトでは、インメモリデータベース (H2 Database) にアクセスするためのデータソース設定が行われているが、これはちょっとした動作検証 (プロトタイプ作成や POC(Proof Of Concept)) を行うための設定である。そのため、本格的なアプリケーション開発を行う場合は、アプリケーション稼働時に利用するデータベースにアクセスするためのデータソース設定に変更する必要がある。

[artifactId/artifactId-domain/pom.xml]

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

[artifactId-infra.properties]

```
database=H2
database.url=jdbc:h2:mem:todo;DB_CLOSE_DELAY=-1
database.username=sa
database.password=
database.driverClassName=org.h2.Driver
# connection pool
cp.maxActive=96
cp.maxIdle=16
cp.minIdle=0
cp.maxWait=60000
```

[artifactId-env.xml]

```
<bean id="realDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="${database.driverClassName}" />
    <property name="url" value="${database.url}" />
    <property name="username" value="${database.username}" />
    <property name="password" value="${database.password}" />
    <property name="defaultAutoCommit" value="false" />
    <property name="maxTotal" value="${cp.maxActive}" />
    <property name="maxIdle" value="${cp.maxIdle}" />
    <property name="minIdle" value="${cp.minIdle}" />
    <property name="maxWaitMillis" value="${cp.maxWait}" />
</bean>
```

注釈: 本格的なアプリケーション開発を行う場合は、アプリケーション稼働時に利用するデータベースにアクセスするためのデータソース設定に変更すること。

Maven Archetype で作成したプロジェクトでは、Apache Commons DBCP を使用する設定となっているが、アプリケーションサーバから提供されているデータソースを使用して、JNDI(Java Naming and Directory Interface) 経由でデータソースにアクセスする方法を採用するケースも多い。

開発環境では Apache Commons DBCP のデータソースを使用して、テスト環境及び商用環境ではアプリケーションサーバから提供されているデータソースを使用するといった使い分けを行うケースもある。

データソースの設定方法については、「データベースアクセス（共通編） の データソースの設定」を参照されたい。

カスタマイズ対象のファイルとカスタマイズ方法を以下に示す。

項番	対象ファイル	カスタマイズ方法
1.	POM ファイル • `artifactId/pom.xml` • `artifactId/artifactId-web/pom.xml`	インメモリデータベース (H2 Database) の JDBC ドライバを依存ライブラリから削除する。 アプリケーション稼働時に利用するデータベースにアクセスするための JDBC ドライバを依存ライブラリに追加する。
2.	環境依存する設定値を定義するプロパティファイル `artifactId/artifactId-env/src/main/resources/META-INF/spring/artifactId-infra.properties`	データソースとして Apache Commons DBCP を使用する場合は、以下のプロパティにアプリケーション稼働時に利用するデータベースにアクセスするための接続情報を指定する。 • `database` • `database.url` • `database.username` • `database.password` • `database.driverClassName` アプリケーションサーバから提供されているデータソースを使用する場合は、以下のプロパティ以外は不要なプロパティになるので削除する。 • `database`
3.	環境依存するコンポーネントを定義する Bean 定義ファイル `artifactId/artifactId-env/src/main/resources/META-INF/spring/artifactId-env.xml`	アプリケーションサーバから提供されているデータソースを使用する場合は、JNDI 経由で取得したデータソースを使用するように設定を変更する。 データソースの設定方法については、「データベースアクセス（共通編） の データソースの設定」を参照されたい。

注釈: 環境依存する設定値を定義するプロパティファイルの database プロパティについて

O/R Mapper として MyBatis を使用する場合は、`database` プロパティは不要なプロパティである。削除してもよいが、使用しているデータベースを明示するために設定を残しておいてもよい。

ちなみに: JDBC ドライバの追加方法について

使用するデータベースが PostgreSQL と Oracle の場合は、POM ファイル内のコメントアウトを外せばよい。JDBC ドライバのバージョンについては、使用するデータベースのバージョンに対応するバージョンに修正すること。

ただし Oracle を使用する場合は、コメントを外す前に、Maven のローカルリポジトリに Oracle の JDBC ドライバをインストールしておく必要がある。

以下は、PostgreSQL を使用する場合の設定例である。

- artifactId/pom.xml

```
                <dependency>
                    <groupId>org.postgresql</groupId>
                    <artifactId>postgresql</artifactId>
                    <version>${postgresql.version}</version>
                </dependency>
<!--            <dependency> -->
<!--                <groupId>com.oracle.jdbc</groupId> -->
<!--                <artifactId>ojdbc7</artifactId> -->
<!--                <version>${ojdbc.version}</version> -->
<!--            </dependency> -->

    <!-- ... -->

    <postgresql.version>42.2.5.jre7</postgresql.version>
    <ojdbc.version>12.1.0.2</ojdbc.version>
```

- artifactId/artifactId-web/pom.xml

```
            <dependency>
                <groupId>org.postgresql</groupId>
                <artifactId>postgresql</artifactId>
                <scope>runtime</scope><!-- (1) -->
            </dependency>
<!--        <dependency> -->
<!--            <groupId>com.oracle.jdbc</groupId> -->
<!--            <artifactId>ojdbc7</artifactId> -->
<!--            <scope>runtime</scope> -->
<!--        </dependency> -->
```

項番	説明
(1)	JDBC ドライバはコンパイルには使用せず、アプリケーション実行時のみ使用するため、runtime スコープを指定している。 単体テストで使用する場合などは、適切なスコープに変更して使用されたい。

3.1.4 開発プロジェクトの構成

Maven Archetype で作成したプロジェクトの構成について説明する。

Maven Archetype で作成したプロジェクトは、以下の構成になっている。

- 本ガイドラインで推奨しているレイヤ毎のプロジェクト構成

- 本ガイドラインで紹介している環境依存性の排除を考慮したプロジェクト構成
- CI(Continuous Integration) を意識したプロジェクト構成

また、本ガイドラインで推奨している各種設定が盛り込まれた、

- Web アプリケーションの構成定義ファイル (web.xml)
- Spring Framework の Bean 定義ファイル
- Spring MVC 用の Bean 定義ファイル
- Spring Security 用の Bean 定義ファイル
- O/R Mapper の設定ファイル
- Tiles 用の設定ファイル
- プロパティファイル (メッセージ定義ファイルなど)

と、アプリケーション要件との依存度が低い (=どんなアプリケーションでも作成する必要がある) コンポーネントの簡易実装として、

- Welcome ページを表示するための Controller と JSP
- エラー画面を表示するための JSP(HTML)
- Tiles 用のテンプレート JSP
- JSP タグライブラリの読み込み設定などが定義されているインクルード用 JSP
- アプリケーション全体の画面スタイルを定義する CSS ファイル

などが提供されている。

警告: 簡易実装として提供しているコンポーネントの扱いについて

簡易実装として提供しているコンポーネントは、以下のいずれかの対応を行うこと。

- アプリケーション要件にあわせて修正
- 不要なコンポーネントは削除

注釈: REST API 用のプロジェクトを作成する場合の手順について

Maven Archetype で作成したプロジェクトは、伝統的な Web アプリケーション (リクエストパラメータを受け取って HTML を応答するアプリケーション) を構築する際に必要となる推奨設定が行われている。

そのため、JSON や XML を扱う REST API を構築する際には不要な設定が存在する。REST API を構築するためのプロジェクトを作成する場合は、「*RESTful Web Service* の アプリケーションの設定」を参照し、REST API 向けの設定を適用してほしい。

注釈: 以降の説明で `artifactId` と表現している部分は、プロジェクト作成時に指定した `artifactId` に置き換えて読み進めてほしい。

マルチプロジェクトの構成

まず、マルチプロジェクト全体の構成について説明する。

```
artifactId
    ├─── pom.xml  ... (1)
    ├─── artifactId-web  ... (2)
    ├─── artifactId-domain  ... (3)
    ├─── artifactId-env  ... (4)
    ├─── artifactId-initdb  ... (5)
    └─── artifactId-selenium  ... (6)
```

項番	説明
(1)	マルチプロジェクト全体の構成を定義する POM(Project Object Model) ファイル。 このファイルでは、主に以下の定義を行う。 • 依存ライブラリのバージョン • ビルド用のプラグインの設定 (ビルド方法の設定) マルチプロジェクトの階層関係については、「プロジェクトの階層構造」を参照されたい。
(2)	アプリケーション層 (Web 層) のコンポーネントを管理するモジュール。 このモジュールでは、主に以下のコンポーネントやファイルを管理する。 • Controller クラス • 相関チェック用の Validator クラス • Form クラス (REST API の場合は Resource クラス) • View(JSP) • CSS ファイル • JavaScript ファイル • アプリケーション層のコンポーネント用の JUnit • アプリケーション層のコンポーネントを定義するための Bean 定義ファイル • Web アプリケーションの構成定義ファイル (web.xml) • メッセージ定義ファイル

Continued on next page

表 3.1 – continued from previous page

項番	説明
(3)	ドメイン層のコンポーネントを管理するモジュール。 このモジュールでは、主に以下のコンポーネントやファイルを管理する。 • Entity などのドメインオブジェクト • Repository • Service • DTO • ドメイン層のコンポーネント用の JUnit • ドメイン層のコンポーネントを定義するための Bean 定義ファイル
(4)	環境依存性をもつ設定ファイルを管理するモジュール。 このモジュールでは、主に以下のファイルを管理する。 • 環境依存するコンポーネントを定義するための Bean 定義ファイル • 環境依存するプロパティ値を定義するプロパティファイル
(5)	データベースを初期化するための SQL ファイルを管理するモジュール このモジュールでは、主に以下のファイルを管理する。 • テーブルなどのデータベースオブジェクトを作成するための SQL ファイル • マスタデータなどの初期データを投入するための SQL ファイル • E2E(End To End) テストで使用するテストデータを投入するための SQL ファイル
(6)	Selenium を使用した E2E テスト用のコンポーネントを管理するモジュール。 このモジュールでは、主に以下のファイルを管理する。 • Selenium を操作してテストを行う JUnit • Assert 時に使用する期待値ファイル (必要に応じて)

注釈: 本ガイドラインにおける「マルチプロジェクト」の用語定義について

Maven Archetype で作成したプロジェクトは、正確にはマルチモジュール構成のプロジェクトとなる。

本ガイドラインでは、マルチモジュールとマルチプロジェクトを同じ意味で使用していることを補足しておく。

注釈: ２つの Web アプリケーションと１つの共通ライブラリが必要となる開発プロジェクトについて

- bar-parent
- bar-initdb
- bar-common
- bar-common-web
- bar-domain-a
- bar-domain-b
- bar-web-a
- bar-web-b
- bar-env
- bar-web-a-selenium
- bar-web-b-selenium

それぞれのプロジェクトの内容は下記のようになる。

- bar-parent

  parent-pom（親 POM）と呼ばれるプロジェクト。pom.xml ファイルだけを持ち、その他のソースコードや設定ファイルは一切持たない、シンプルなプロジェクト。他のプロジェクトの pom 上で、この bar-parent プロジェクトを<parent>タグに指定することによって、親 POM に指定された共通設定情報を自身に反映させることができる。

- bar-initdb

  RDBMS のテーブル定義 (DDL) と初期データを INSERT するための SQL 文を格納する。これも maven プロジェクトとして管理する。pom.xml に sql-maven-plugin の設定を定義することにより、ビルドライフサイクルの過程で任意の RDBMS に対する DDL 文や初期データ INSERT 文の実行を自動化することができる。

- bar-common

  プロジェクト共通ライブラリを格納する。ここは web 非依存にし、web に関わるクラスは bar-common-web に配置する。

- bar-common-web

  プロジェクト共通 web ライブラリを格納する

- bar-domain-a

  a ドメインに関わるドメイン層の java クラス、単体テストケース等を格納するプロジェクト。最終的に*.jar ファイル化する。

- bar-domain-b

  b ドメインに関わるドメイン層のクラス。

- bar-web-a

  アプリケーション層の java クラス、jsp、設定ファイル、単体テストケース等を格納するプロジェクト。最終的に Web アプリケーションとして*.war ファイル化する。bar-web-a は、bar-common と bar-env への依存性を持つ。

- bar-web-b

  もう一つのサブシステムとしての Web アプリケーション。構造は bar-web-a と同じ。

- bar-env

  環境依存性のある設定ファイルだけを集めるプロジェクト。

- bar-web-a-selenium

  web-a プロジェクトのための、Selenium WebDriver によるテストケースを格納するプロジェクト。

- bar-web-b-selenium

  web-b プロジェクトのための、Selenium WebDriver によるテストケースを格納するプロジェクト。

web モジュールの構成

アプリケーション層 (Web 層) のコンポーネントを管理するモジュールの構成について説明する。

```
artifactId-web
    ├─── pom.xml  ... (1)
```

項番	説明
(1)	web モジュールの構成を定義する POM(Project Object Model) ファイル。このファイルでは、以下の定義を行う。 • 依存ライブラリとビルド用プラグインの定義 • war ファイルを作成するための定義

注釈: **REST API** 用のプロジェクトを作成する際の **web** モジュールのモジュール名について

REST API を構築する場合は、モジュール名を `artifactId-api` といった感じの名前にしておくと、アプリケーションの種類が識別しやすくなる。

```
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── project
    │   │               └── app  ... (2)
    │   │                   └── welcome
    │   │                       └── HelloController.java  ... (3)
    │   ├── resources
    │   │   ├── META-INF
    │   │   │   ├── dozer  ... (4)
    │   │   │   └── spring  ... (5)
    │   │   │       ├── application.properties  ... (6)
    │   │   │       ├── applicationContext.xml  ... (7)
    │   │   │       ├── spring-mvc.xml  ... (8)
    │   │   │       └── spring-security.xml  ... (9)
    │   │   └── i18n  ... (10)
    │   │       └── application-messages.properties  ... (11)
```

項番	説明
(2)	アプリケーション層のクラスを格納するためのパッケージ。 REST API を構築する場合は、パッケージ名を `api` といった感じの名前にしておくと、コンポーネントの種類が識別しやすくなる。
(3)	Welcome ページを表示するためのリクエストを受け取るための Controller クラス。
(4)	Dozer(Bean Mapper) のマッピング定義ファイルを格納するディレクトリ。Dozer については、「*Bean マッピング (Dozer)*」を参照されたい。 作成時点では空のディレクトリである。マッピングファイルが必要になった場合 (高度なマッピングが必要になった場合) は、このディレクトリ配下に格納すると、自動的にマッピングファイルが読み込まれる。 --- **注釈:** このディレクトリには、以下のファイルを格納する。 • アプリケーション層の JavaBean とドメイン層の JavaBean をマッピングするための定義ファイル • アプリケーション層の JavaBean 同士をマッピングするための定義ファイル ドメイン層の JavaBean 同士のマッピングはドメイン層のディレクトリに格納することを推奨している。 ---
(5)	Spring Framework の Bean 定義ファイルとプロパティファイルを格納するディレクトリ。
(6)	アプリケーション層で使用する設定値を定義するプロパティファイル。 作成時点では、空のファイルである。
(7)	Web アプリケーション用のアプリケーションコンテキストを作成するための Bean 定義ファイル。 このファイルには、以下の Bean を定義する。 • Web アプリケーション全体で使用するコンポーネント • ドメイン層のコンポーネント (ドメイン層のコンポーネントが定義されている Bean 定義ファイルを import する)
(8)	`DispatcherServlet` 用のアプリケーションコンテキストを作成するための Bean 定義ファイル。 このファイルには、以下の Bean を定義する。 • Spring MVC のコンポーネント • アプリケーション層のコンポーネント REST API を構築する場合は、ファイル名を `spring-mvc-api.xml` といった感じの名前にしておくと、 アプリケーションの種類が識別しやすくなる。

Continued on next page

表 3.2 – continued from previous page

項番	説明
(9)	Spring Security のコンポーネントを定義するための Bean 定義ファイル。 このファイルは、Web アプリケーション用のアプリケーションコンテキストを作成する際に読み込む。
(10)	アプリケーション層で使用するメッセージ定義ファイルを格納するディレクトリ。
(11)	アプリケーション層で使用するメッセージを定義するプロパティファイル。 作成時点では、いくつかの汎用的なメッセージが定義されている。 --- **注釈: メッセージについては、アプリケーションの要件 (メッセージ規約など) にあわせて必ず修正すること。**メッセージ定義については、「メッセージ管理」を参照されたい。 ---

注釈: アプリケーションコンテキストと Bean 定義ファイルの関連については、「アプリケーションコンテキストの構成と *Bean* 定義ファイルの関係」を参照されたい。

```
|    └── webapp
|        ├── WEB-INF
|        |    ├── tiles  ... (12)
|        |    |    └── tiles-definitions.xml
|        |    ├── views  ... (13)
|        |    |    ├── common
|        |    |    |    ├── error  ... (14)
|        |    |    |    |    ├── accessDeniedError.jsp
|        |    |    |    |    ├── businessError.jsp
|        |    |    |    |    ├── dataAccessError.jsp
|        |    |    |    |    ├── invalidCsrfTokenError.jsp
|        |    |    |    |    ├── missingCsrfTokenError.jsp
|        |    |    |    |    ├── resourceNotFoundError.jsp
|        |    |    |    |    ├── systemError.jsp
|        |    |    |    |    ├── transactionTokenError.jsp
|        |    |    |    |    └── unhandledSystemError.html
|        |    |    |    └── include.jsp  ... (15)
|        |    |    ├── layout  ... (16)
|        |    |    |    ├── header.jsp
|        |    |    |    └── template.jsp
|        |    |    └── welcome
|        |    |        └── home.jsp  ... (17)
|        |    └── web.xml  ... (18)
|        └── resources  ... (19)
|            └── app
|                └── css
|                    └── styles.css  ... (20)
└── test
    ├── java
    └── resources
```

項番	説明
(12)	Tiles の設定ファイルを格納するディレクトリ。Tiles の設定ファイルについては、「*Tiles* による画面レイアウト」を参照されたい。

Continued on next page

表 3.3 – continued from previous page

項番	説明
(13)	View を構築するテンプレートファイル (JSP など) を格納するディレクトリ。
(14)	エラー画面を表示するための JSP 及び HTML を格納するディレクトリ。 作成時点では、アプリケーション実行時に発生する可能性があるエラーに対応する JSP(HTML) が格納されている。 **注釈: エラー画面用の JSP 及び HTML については、アプリケーションの要件 (UI 規約など) にあわせて必ず修正すること。**
(15)	インクルード用の共通 JSP ファイル。 このファイルは、全ての JSP ファイルの先頭にインクルードされる。インクルード用の共通 JSP ファイルについては、「インクルード用の共通 *JSP* の作成」を参照されたい。
(16)	Tiles のレイアウト用の JSP ファイルを格納するディレクトリ。Tiles のレイアウト用の JSP ファイルについては、「*Tiles* による画面レイアウト」を参照されたい。
(17)	Welcome ページを表示する JSP ファイル。
(18)	Web アプリケーションの構成定義ファイル。
(19)	静的なリソースファイルを格納するディレクトリ。 このディレクトリは、リクエストの内容によって応答する内容がかわらないファイルを格納する。 具体的には以下のファイルを格納する。 • JavaScript ファイル • CSS ファイル • 画像ファイル • HTML ファイル Spring MVC が提供する静的リソースの管理メカニズムを適用しやすくするために、専用のディレクトリを設ける構成を採用している。
(20)	アプリケーション全体に適用する画面スタイルを定義する CSS ファイル。

domain モジュールの構成

ドメイン層のコンポーネントを管理するモジュールの構成について説明する。

```
artifactId-domain
    ├─── pom.xml  ... (1)
```

項番	説明
(1)	domain モジュールの構成を定義する POM(Project Object Model) ファイル。このファイルでは、以下の定義を行う。 • 依存ライブラリとビルド用プラグインの定義 • jar ファイルを作成するための定義

```
└─── src
    ├─── main
    │    ├─── java
    │    │    └─── com
    │    │        └─── example
    │    │            └─── project
    │    │                └─── domain  ... (2)
    │    │                    ├─── model
    │    │                    ├─── repository
    │    │                    └─── service
    │    └─── resources
    │        └─── META-INF
    │            ├─── dozer  ... (3)
    │            └─── spring  ... (4)
    │                ├─── artifactId-codelist.xml  ... (5)
    │                ├─── artifactId-domain.xml  ... (6)
    │                └─── artifactId-infra.xml  ... (7)
```

項番	説明
(2)	ドメイン層のクラスを格納するためのパッケージ。
(3)	Dozer(Bean Mapper) のマッピング定義ファイルを格納するディレクトリ。Dozer については、「*Bean マッピング (Dozer)*」を参照されたい。 作成時点では空のディレクトリである。マッピングファイルが必要になった場合 (高度なマッピングが必要になった場合) は、このディレクトリ配下に格納すると、自動的にマッピングファイルが読み込まれる。 **注釈:** このディレクトリには、以下のファイルを格納する。 • ドメイン層の JavaBean 同士をマッピングするための定義ファイル
(4)	Spring Framework の Bean 定義ファイルとプロパティファイルを格納するディレクトリ。
(5)	コードリストを定義するための Bean 定義ファイル。 **注釈:** 大量にコードリストを定義する場合は、Bean 定義ファイルを複数用意し、コードリストが使用される業務ごとやコードリストが使用されるレイヤごとなどの観点で分類してもよい。 たとえば、アプリケーション層（画面）のみで使用するコードリストを web モジュールの artifactId-web-codelist.xml に定義し、ドメイン層でも使用するコードリストを domain モジュールの artifactId-domain-codelist.xml に定義するといった方法が考えられる。
(6)	ドメイン層のコンポーネントを定義するための Bean 定義ファイル。 このファイルには、以下の Bean を定義する。 • ドメイン層のコンポーネント (Service, Repository など) • インフラストラクチャ層のコンポーネント (インフラストラクチャ層のコンポーネントが定義されている Bean 定義ファイルを import する) • Spring Framework から提供されているトランザクション管理用のコンポーネント
(7)	インフラストラクチャ層のコンポーネントを定義するための Bean 定義ファイル。 このファイルには、O/R Mapper などの Bean 定義を行う。

```
└── test
    ├── java
    │   └── com
    │       └── example
    │           └── project
    │               └── domain
    │                   ├── repository
    │                   └── service
    └── resources
        └── test-context.xml  ... (8)
```

項番	説明
(8)	ドメイン層のユニットテスト用のコンポーネントを定義するための Bean 定義ファイル。

MyBatis3 用のプロジェクトを作成した場合

```
└── src
    ├── main
    │   ├── java
  (...)
    │   └── resources
    │       ├── META-INF
    │       │   ├── dozer
    │       │   ├── mybatis  ... (9)
    │       │   │   └── mybatis-config.xml  ... (10)
    │       │   └── spring
  (...)
    │       └── com
    │           └── example
    │               └── project
    │                   └── domain
    │                       └── repository  ... (11)
    │                           └── sample
    │                               └── SampleRepository.xml  ... (12)
```

項番	説明
(9)	MyBatis3 の設定ファイルを格納するディレクトリ。
(10)	MyBatis3 の設定ファイル。 作成時点では、いくつかの推奨設定が定義されている。
(11)	MyBatis3 の Mapper ファイルを格納するディレクトリ。
(12)	MyBatis3 の Mapper ファイルのサンプルファイル。 作成時点では、サンプル実装がコメントアウトされた状態になっている。このファイルは最終的には不要なファイルである。

env モジュールの構成

環境依存性をもつ設定ファイルを管理するモジュールの構成について説明する。

```
artifactId-env
    ├── configs  ... (1)
    │    ├── production-server  ... (2)
    │    │    └── resources
    │    └── test-server
    │         └── resources
    ├── pom.xml  ... (3)
```

項番	説明
(1)	環境依存する設定ファイルを管理するためのディレクトリ。 環境毎にサブディレクトリを作成し、環境依存する設定ファイルを管理する。
(2)	環境毎の設定ファイルを管理するためのディレクトリ。 作成時点では、最もシンプルな構成として、以下のディレクトリ (雛形のディレクトリ) が用意されている。 • production-server (商用環境向けの設定ファイルを格納するディレクトリ) • test-server (テスト環境向けの設定ファイルを格納するディレクトリ)
(3)	env モジュールの構成を定義する POM(Project Object Model) ファイル。このファイルでは、以下の定義を行う。 • 依存ライブラリとビルド用プラグインの定義 • 環境毎の jar ファイルを作成するための Profile の定義

```
└── src
    └── main
        └── resources  ... (4)
            ├── META-INF
            │   └── spring
            │       ├── artifactId-env.xml  ... (5)
            │       └── artifactId-infra.properties  ... (6)
            ├── database  ... (7)
            │   ├── H2-dataload.sql
            │   └── H2-schema.sql
            ├── dozer.properties  ... (8)
            ├── log4jdbc.properties  ... (9)
            └── logback.xml  ... (10)
```

項番	説明
(4)	開発用の設定ファイルを管理するためのディレクトリ。
(5)	環境依存するコンポーネントを定義する Bean 定義ファイル。 このファイルには、以下の Bean を定義する。 • データソース • 共通ライブラリから提供している `JodaTimeDateFactory`(環境によって異なる実装を使用する場合) • Spring Framework から提供されているトランザクション管理用のコンポーネント (環境によって異なる実装を使用する場合)
(6)	環境依存する設定値を定義するプロパティファイル。 作成時点では、データソースの設定値 (接続情報とコネクションプールの設定値) が定義されている。
(7)	インメモリデータベース (H2 Database) をセットアップするための SQL を格納するディレクトリ。 このディレクトリは、ちょっとした動作検証を行う時のために用意しているディレクトリである。 **実際のアプリケーション開発で使用することは想定していないので、基本的にはこのディレクトリは削除すること。**
(8)	Dozer(Bean Mapper) のグローバル設定を行うためのプロパティファイル。Dozer については、「*Bean マッピング (Dozer)*」を参照されたい。 作成時点では、空のファイルである。(ファイルがないと起動時に警告ログが出力されるため、これを防ぐために空のファイルを用意している)
(9)	Log4jdbc-remix(JDBC 関連のログ出力を行うライブラリ) のグローバル設定を行うためのプロパティファイル。Log4jdbc-remix については、「*JDBC の Debug 用ログの設定*」を参照されたい。 作成時点では、ログに出力する SQL の改行に関する設定のみ指定されている。
(10)	Logback(ログ出力) の設定ファイル。ログ出力については、「ロギング」を参照されたい。

initdb モジュールの構成

データベースを初期化するための SQL ファイルを管理するモジュールの構成について説明する。

```
artifactId-initdb
    ├── pom.xml  ... (1)
    └── src
        └── main
            └── sqls  ... (2)
```

項番	説明
(1)	initdb モジュールの構成を定義する POM(Project Object Model) ファイル。このファイルでは、以下の定義を行う。 • ビルド用プラグイン (SQL Maven Plugin) の定義 作成時点では、PostgreSQL 用の雛形設定が定義されている。
(2)	データベースを初期化するための SQL ファイルを格納するためのディレクトリ。 作成時点では、空のディレクトリである。作成例については、サンプルアプリケーションの initdb プロジェクト を参照されたい。

注釈: SQL Maven Plugin の sql:execute を使用して、SQL を実行できる。

```
mvn sql:execute
```

---

selenium モジュールの構成

Selenium を使用した E2E(End To End) テスト用のコンポーネントを管理するモジュールの構成について説明する。

```
artifactId-selenium
    ├── pom.xml  ... (1)
    └── src
        └── test  ... (2)
            ├── java
            │   └── com
            │       └── example
            │           └── project
            │               └── selenium
            │                   └── welcome
            │                       └── HelloTest.java  ... (3)
            └── resources
                └── META-INF
                    └── spring
```

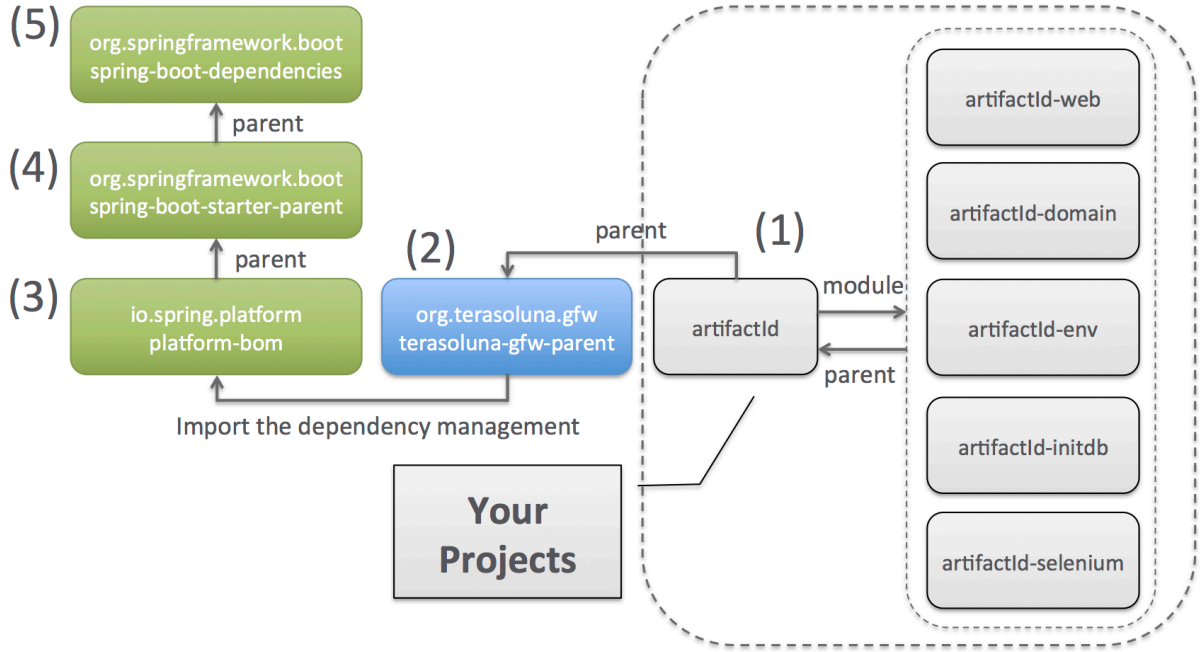
```
                    ├── selenium.properties  ... (4)
                    └── seleniumContext.xml  ... (5)
```

項番	説明
(1)	selenium モジュールの構成を定義する POM(Project Object Model) ファイル。 このファイルでは、以下の定義を行う。 • 依存ライブラリとビルド用プラグインの定義 • jar ファイルを作成するための定義
(2)	テスト用のコンポーネントと設定ファイルを格納するディレクトリ。 作成例については、サンプルアプリケーションの selenium プロジェクト を参照されたい。
(3)	Selenium WebDriver を使用したサンプルテストクラス。 作成時点では、Welcome ページのタイトルを検証するテストケースが実装されている。
(4)	テストで使用する設定値を定義するプロパティファイル。 作成時点では、アプリケーションサーバの URL は `http://localhost:8080/`である。
(5)	テスト用のコンポーネントを定義するための Bean 定義ファイル。 作成時点では、サンプルのテストを実行するために必要な設定がされている。

3.1.5 Appendix

プロジェクトの階層構造

Maven Archetype で作成したプロジェクトのプロジェクト階層の構造を以下に示す。



項番	説明
(1)	Maven Archetype で作成したプロジェクト。 Maven Archetype で作成したプロジェクトはマルチモジュール構成となっており、親プロジェクトと各サブモジュールは相互参照の関係になっている。 version 5.4.2.RELEASE 用の Maven Archetype で作成したプロジェクトでは、親プロジェクトとして「org.terasoluna.gfw:terasoluna-gfw-parent:5.4.2.RELEASE」を指定している。
(2)	TERASOLUNA Server Framework for Java (5.x) Parent プロジェクト。 TERASOLUNA Server Framework for Java (5.x) Parent プロジェクトでは、 • ビルド用のプラグインの設定 • Spring IO Platform 経由で管理されているライブラリのカスタマイズ (バージョンの調整) • Spring IO Platform で管理されていない推奨ライブラリのバージョン管理 を行っている。 なお、Spring IO Platform 経由で依存ライブラリのバージョンを管理するために、本プロジェクトの`<dependencyManagement>`に「Spring IO Platform」をインポートしている。 利用している Spring IO Platform のバージョンは利用する *OSS* のバージョン参照のこと。
(3)	Spring IO Platform プロジェクト。 親プロジェクトとして「org.springframework.boot:spring-boot-starter-parent:1.2.5.RELEASE」が指定されているため、spring-boot-starter-parent の pom ファイルに定義されている`<dependencyManagement>`の定義も、terasoluna-gfw-parent の pom ファイルにインポートされる。

Continued on next page

表 3.4 – continued from previous page

項番	説明
(4)	Spring Boot Starter Parent プロジェクト。 親プロジェクトとして「org.springframework.boot:spring-boot-dependencies:1.2.5.RELEASE」が指定されているため、spring-boot-dependencies の pom ファイルに定義されている<dependencyManagement>の定義も、terasoluna-gfw-parent の pom ファイルにインポートされる。
(5)	Spring Boot Dependencies プロジェクト。

ちなみに： version 5.0.0.RELEASE より、Spring IO Platform の<dependencyManagement>をインポートする構成に変更しており、推奨ライブラリのバージョン管理を Spring IO Platform に委譲するスタイルを採用している。

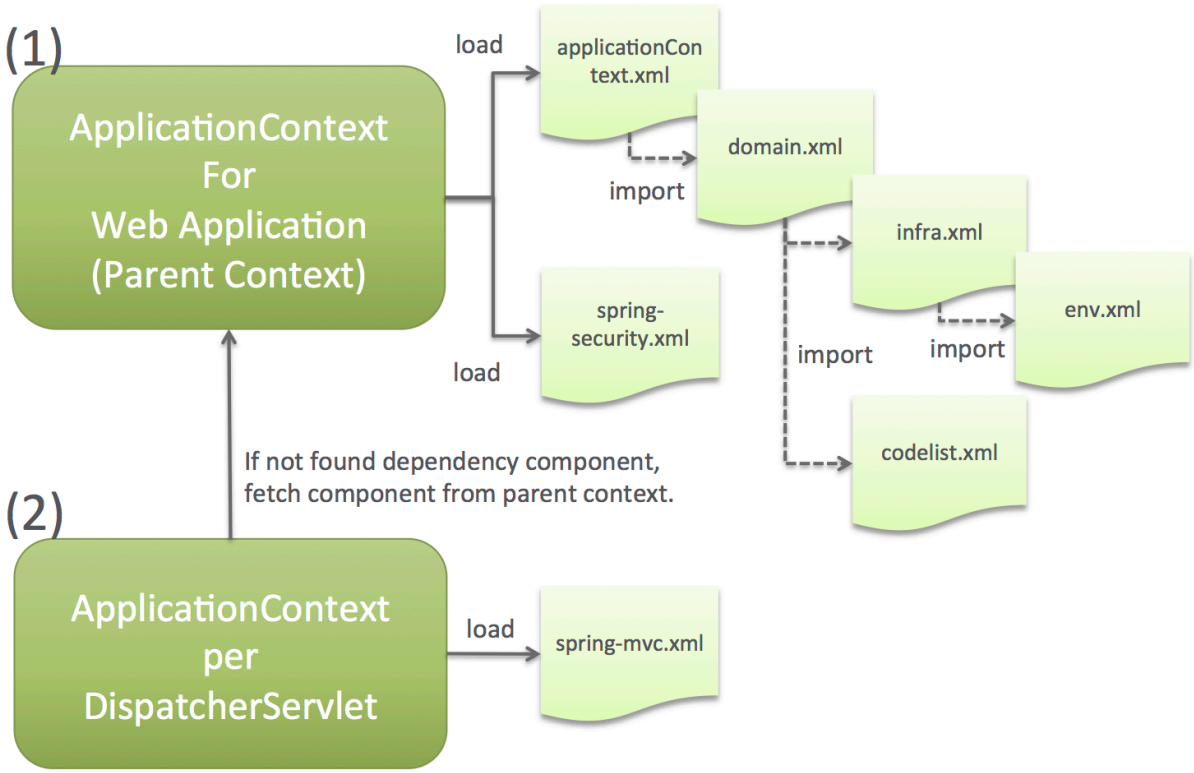
警告： version 5.0.0.RELEASE より、Spring IO Platform の<dependencyManagement>をインポートする構成に変更したため、子プロジェクトからライブラリのバージョンを管理するためのプロパティにアクセスする事が出来なくなっている。

そのため、子プロジェクト側でプロパティ値を参照又は上書きしている場合は、version 1.0.x からバージョンアップする際に pom ファイルの修正が必要になる。

なお、Spring IO Platform で管理していない推奨ライブラリ (TERASOLUNA Server Framework for Java (5.x) 独自の推奨ライブラリ) については、従来通りバージョンを管理するためのプロパティにアクセスする事ができる。

アプリケーションコンテキストの構成と Bean 定義ファイルの関係

Spring Framework のアプリケーションコンテキスト (DI コンテナ) の構成と Bean 定義ファイルの関係を以下に示す。



項番	説明
(1)	Web アプリケーション用のアプリケーションコンテキスト。 上記図で示す通り、 • artifactId-web/src/main/resource/META-INF/spring/applicationContext.xml • artifactId-domain/src/main/resource/META-INF/spring/artifactId-domain.xml • artifactId-domain/src/main/resource/META-INF/spring/artifactId-infra.xml • artifactId-env/src/main/resource/META-INF/spring/artifactId-env.xml • artifactId-domain/src/main/resource/META-INF/spring/artifactId-codelist.xml • artifactId-web/src/main/resource/META-INF/spring/spring-security.xml で定義したコンポーネントが Web アプリケーション用のアプリケーションコンテキスト (DI コンテナ) に登録される。 Web アプリケーション用のアプリケーションコンテキストに登録されているコンポーネントは、各 `DispatcherServlet` 用のアプリケーションコンテキストから参照する事ができる仕組みとなっている。
(2)	`DispatcherServlet` 用のアプリケーションコンテキスト。 上記図で示す通り、 • artifactId-web/src/main/resource/META-INF/spring/spring-mvc.xml で定義したコンポーネントが `DispatcherServlet` 用のアプリケーションコンテキスト (DI コンテナ) に登録される。 `DispatcherServlet` 用のアプリケーションコンテキストに存在しないコンポーネントは、Web アプリケーション用のアプリケーションコンテキスト (親コンテキスト) を参照して取得する仕組みになっているため、ドメイン層のコンポーネントをアプリケーション層のコンポーネントに対してインジェクションする事ができる。

注釈: 同じコンポーネントを両方のアプリケーションコンテキストに登録した時の動作について

Web アプリケーション用のアプリケーションコンテキストと `DispatcherServlet` 用のアプリケーションコンテキストの両方に同じコンポーネントが登録されている場合は、同じアプリケーションコンテキスト (`DispatcherServlet` 用のアプリケーションコンテキスト) 内に登録されているコンポーネントがインジェクションされる点を補足しておく。

特に、ドメイン層のコンポーネント (Service や Repository など) を `DispatcherServlet` 用のアプリケーションコンテキストに登録しないように注意する必要である。

ドメイン層のコンポーネントを `DispatcherServlet` 用のアプリケーションコンテキストに登録してしまうと、トランザクション制御を行うコンポーネント (AOP) が有効にならないため、データベースへの操作がコミットされない不具合が発生してしまう。

なお、Maven Archetype で作成したプロジェクトでは、上記のような現象は発生しないように設定が行われている。設定の追加又は変更を行う場合は、注意してほしい。

オフライン環境におけるアプリケーション開発

「開発プロジェクトの作成」では、マルチプロジェクト構成の開発プロジェクトを、Maven Archetype Plugin の archetype:generate を使用して作成する方法について述べた。Maven はオンライン環境での動作が前提であるが、以下にオフライン環境でも使用できるようにする方法について述べる。

オフライン環境でプロジェクト開発を続けるためには、開発に必要となるライブラリやプラグイン等のファイルを事前にコピーする必要がある。以下の作業は **オンライン環境** で行うこと。

開発プロジェクトのルートディレクトリへ移動する。ここでは「開発プロジェクトの作成」で作成したプロジェクトを例に説明をする。

```
cd C:\work\todo
```

プロジェクト開発に必要であるライブラリやプラグイン等のファイルをコピーする。Maven Archetype Plugin の dependency:go-offline を実行することでコピーする。

```
mvn dependency:go-offline -Dmaven.repo.local=repository
```

パラメータ	説明
–Dmaven.repo.local	コピー先を指定する。コピー先が存在しない場合は新たに作成される。今回はコピー先を repository と指定している。

成果物を配布しやすくするために、war ファイルまたは jar ファイルを作成する。この時、ビルドに必要となるライブラリやプラグイン等のファイルがコピーされる。

```
mvn package -Dmaven.repo.local=repository
```

ビルドが成功した場合、以下のようなログが出力される。

```
(... omit)
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] TERASOLUNA Server Framework for Java (5.x) Web Blank Multi Project (MyBa
tis3) SUCCESS [  0.006 s]
[INFO] todo-env ........................................... SUCCESS [ 46.565 s]
[INFO] todo-domain ........................................ SUCCESS [  0.684 s]
[INFO] todo-web ........................................... SUCCESS [ 12.832 s]
[INFO] todo-initdb ........................................ SUCCESS [  0.067 s]
[INFO] todo-selenium ...................................... SUCCESS [01:13 min]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 02:14 min
[INFO] Finished at: 2015-10-01T10:32:34+09:00
[INFO] Final Memory: 36M/206M
[INFO] ------------------------------------------------------------------------
```

以上で、プロジェクト開発に必要なライブラリやプラグイン等のファイルをコピーした。この repository をオフライン環境マシンの${HOME}/.m2 へコピーすることで、作業は完了となる。オンライン環境で一度も実行していない処理をオフライン環境で実行すると、必要なライブラリやプラグイン等のファイルを取得できず処理に失敗するが、コピーを行ったことにより、オフライン環境へ移行した場合においても継続して開発を進めることが可能となる。

警告：オフライン環境での開発における注意点

オフライン環境では新規に依存関係をインターネットから取得することが不可能となるため、POM（Project Object Model）ファイルを編集しないこと。POM ファイルに編集を加える場合は、再度オンライン環境へ戻る必要がある。

3.2 ドメイン層の実装

3.2.1 ドメイン層の役割

ドメイン層は、 アプリケーション層に提供する**業務ロジックを実装する**ためのレイヤとなる。

ドメイン層の実装は、以下 3 つに分かれる。

項番	分類	説明
1.	*Entity の実装*	業務データを保持するためのクラス (Entity クラス) を作成する。
2.	*Repository の実装*	業務データを操作するためのメソッドを実装し、Service クラスに提供する。 業務データを操作するためのメソッドとは、具体的には、Entity オブジェクトに対する CRUD 操作となる。
3.	*Service の実装*	業務ロジックを実行するためのメソッドを実装し、アプリケーション層に提供する。 業務ロジック内で必要となる業務データは、Repository を介して、Entity オブジェクトとして取得する。

本ガイドラインでは、以下 2 点を目的として、Entity クラスおよび Repository を作成する構成を推奨している。

1. 業務ロジック (Service) と業務データへアクセスするためのロジックを分離することで、**業務ロジックの実装範囲をビジネスルールに関する実装に専念させる。**
2. 業務データに対する操作を Repository に集約することで、**業務データへのアクセスの共通化を行う。**

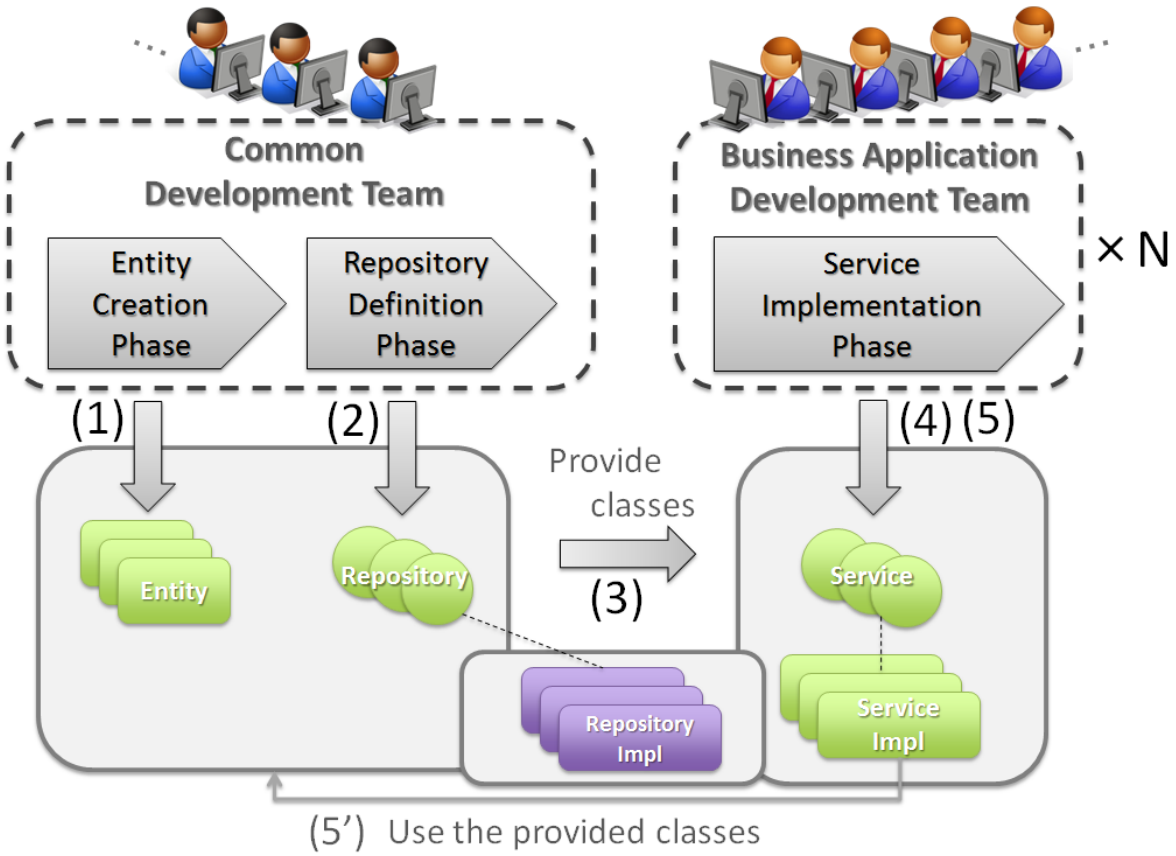
注釈: 本ガイドラインでは、Entity クラスおよび Repository を作成する構成を推奨しているが、この構成で開発することを強制するものではない。

作成するアプリケーションの特性、プロジェクトの特性 (開発体制や開発プロセスなど) を加味して、採用する構成を決めて頂きたい。

3.2.2 ドメイン層の開発の流れ

ドメイン層の開発の流れと、役割分担について説明する。

下記の説明では、複数の開発チームが存在する状態でアプリケーションを構築するケースを想定しているが、1 チームで開発する場合でも、開発フロー自体は変わらない。



項番	担当チーム	説明
(1)	共通開発チーム	共通開発チームは、Entity クラスの設計および Entity クラスの作成を行う。
(2)	共通開発チーム	共通開発チームは、(1) で抽出した Entity クラスに対するアクセスパターンを整理し、Repository インタフェースのメソッド設計を行う。 複数の開発チームで共有するメソッドに対する実装については、共通開発チームで実装することが望ましい。
(3)	共通開発チーム	共通開発チームは、(1) と (2) で作成した Entity クラスと、Repository を業務アプリケーション開発チームに提供する。 このタイミングで、各業務アプリケーション開発チームに対して、Repository インタフェースの実装を依頼する。
(4)	業務アプリケーション開発チーム	業務アプリケーション開発チームは、自チーム担当分の Repository インタフェースの実装を行う。
(5)	業務アプリケーション開発チーム	業務アプリケーション開発チームは、共通開発チームから提供された Entity クラスおよび Repository と自チームで作成した Repository を利用して、Service インタフェースおよび Service クラスの実装を行う。

警告: 開発規模が大きいシステムでは、アプリケーションを複数のチームに分担して開発を行う場合がある。その場合は、Entity クラスおよび Repository を設計するための共通チームを設けることを強く推奨する。

共通チームを設ける体制が組めない場合は、Entity クラスおよび Repository を作成せずに、Service から O/R Mapper(MyBatis など) を直接呼び出して、業務データにアクセスする方法を採用することを検討すること。

3.2.3 Entity の実装

Entity クラスの作成方針

Entity は原則以下の方針で作成する。

具体的な作成方法については、*Entity クラスの作成例*で示す。

項番	方針	補足
1.	Entity クラスは、テーブル毎に作成する。	ただし、テーブル間の関連を保持するためのマッピングテーブルについては、Entity クラスは不要である。 また、テーブルが正規化されていない場合は、必ずしもテーブル毎にはならない。テーブルが正規化されていない時のアプローチは、表外の警告欄と備考欄を参照されたい。
2.	テーブルに FK(Foreign Key) がある場合は、FK 先のテーブルの Entity クラスをプロパティとして定義する。	FK 先のテーブルとの関係が、1:N になる場合は、`java.util.List<E>`または`java.util.Set<E>`のどちらかを使用する。 FK 先のテーブルに対応する Entity のことを、本ガイドライン上では、関連 Entity と呼ぶ。
3.	コード系テーブルは、Entity として扱うのではなく、`java.lang.String` などの基本型で扱う。	コード系テーブルとは、コード値と、コード名のペアを管理するためのテーブルのことである。 コード値によって処理分岐する必要がある場合は、コード値に対応する enum クラスを作成し、作成した enum をプロパティとして定義することを推奨する。

警告: テーブルが正規化されていない場合は、以下の点を考慮して **Entity クラスおよび Repository を作成する方式を採用すべきか検討した方がよい。**特に正規化されていないテーブルと JPA との相性はあまりよくないので、テーブルが正規化されていない場合は、JPA を使用して Entity オブジェクトを操作する方式は採用しない方が無難である。

- Entity を作成する難易度が高くなるため、適切な Entity クラスの作成が出来ない可能性がある。

  加えて、Entity クラスを作成するために、必要な工数が多くなる可能性も高い。

  前者は、「適切に正規化できるエンジニアをアサインできるか？」という観点、後者は、「工数をかけて正規化された Entity クラスを作成する価値があるか？」という観点で、検討することになる。

- 業務データにアクセスする際の処理として、Entity クラスとテーブルの構成の差分を埋める

ための処理が、必要となる。

これは、「工数をかけて、Entity とテーブルの差分を埋めるための処理を実装する価値があるか？」という観点で検討することになる。

Entity クラスと Repository を作成する方式を採用することを推奨するが、作成するアプリケーションの特性、プロジェクトの特性 (開発体制や開発プロセスなど) を加味して、採用する構成を決めて頂きたい。

注釈: テーブルは正規化されていないが、アプリケーションとして、正規化された Entity として業務データを扱いたい場合は、インフラストラクチャ層の RepositoryImpl の実装として、MyBatis を採用することを推奨する。

MyBatis は、データベースで管理されているレコードとオブジェクトをマッピングするという考え方ではなく、SQL とオブジェクトをマッピングという考え方で開発された O/R Mapper であるため、SQL の実装次第で、テーブル構成に依存しないオブジェクトへのマッピングができる。

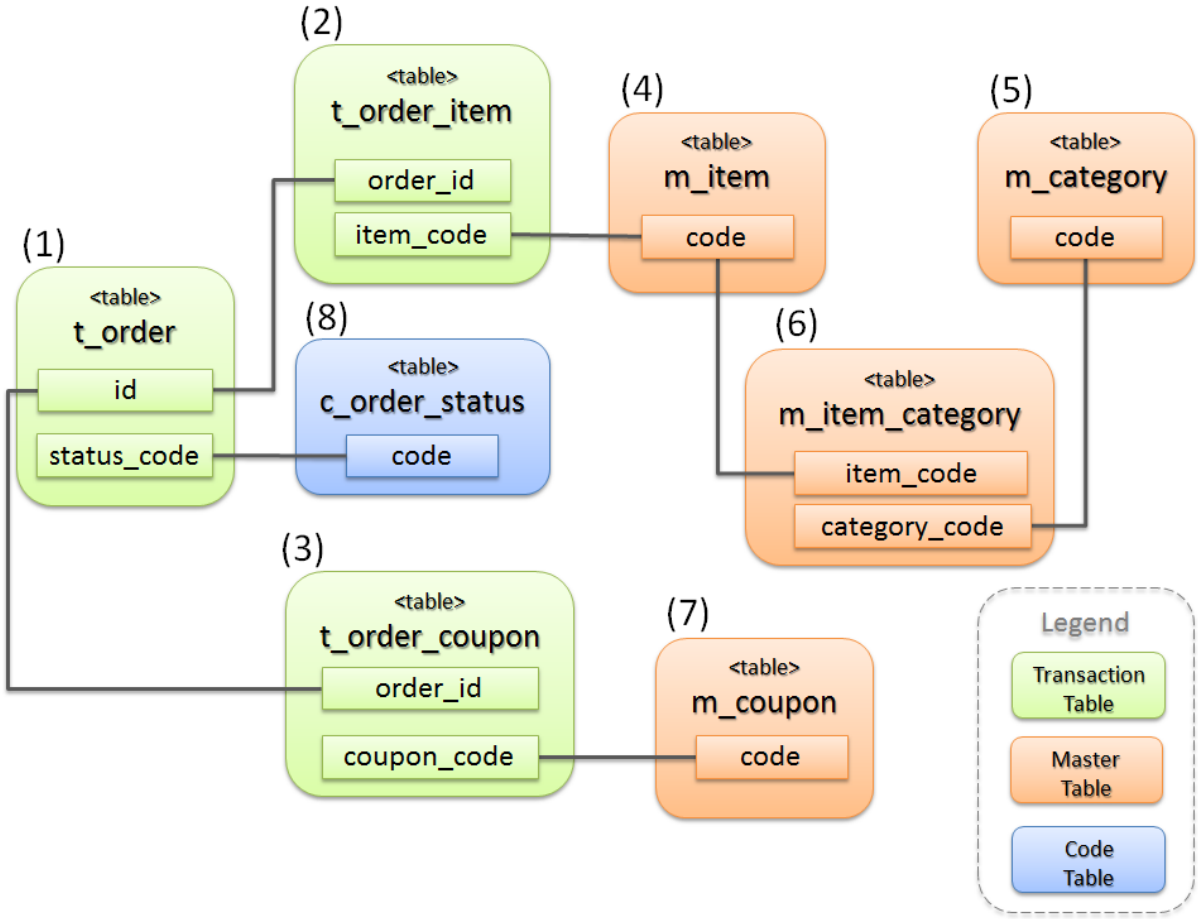
Entity クラスの作成例

Entity クラスの作成方法を、具体例を用いて説明する。

以下は、ショッピングサイトで商品を購入する際に必要となる業務データを、Entity クラスとして作成する例となっている。

テーブル構成

商品を購入する際に必要となる業務データを保持するテーブルは、以下の構成となっている。



項番	分類	テーブル名	説明
(1)	トランザクション系	t_order	注文を保持するテーブル。 1 つの注文に対して 1 レコードが格納される。
(2)		t_order_item	1 つの注文で購入された商品を保持するテーブル。 1 つの注文で複数の商品が購入された場合は商品数分レコードが格納される。
(3)		t_order_coupon	1 つの注文で使用されたクーポンを保持するテーブル。 1 つの注文で複数のクーポンが使用された場合はクーポン数分レコードが格納される。クーポンを使用しなかった場合、レコードは格納されない。

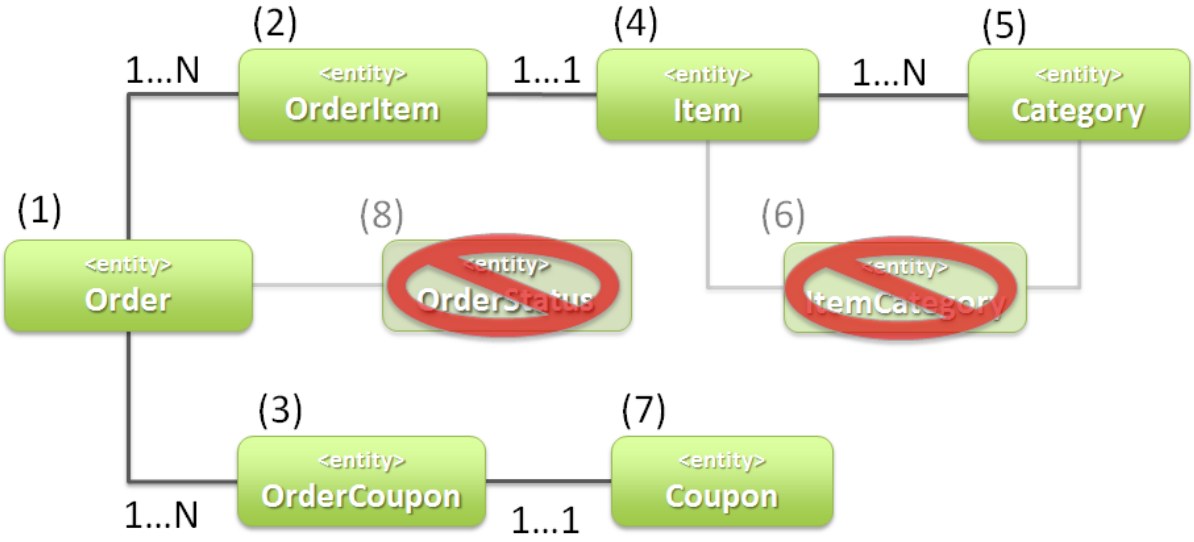
Continued on next page

表 3.6 – continued from previous page

項番	分類	テーブル名	説明
(4)	マスタ系	m_item	商品を定義するマスタテーブル。
(5)		m_category	商品のカテゴリを定義するマスタテーブル。
(6)		m_item_category	商品が所属するカテゴリを定義するマスタテーブル。商品とカテゴリのマッピングを保持している。1 つの商品は複数のカテゴリに属すことができるモデルとなっている。
(7)		m_coupon	クーポンを定義するマスタテーブル。
(8)	コード系	c_order_status	注文ステータスを定義するコードテーブル。

Entity 構成

上記テーブルから作成方針に則って Entity クラスを作成すると、以下のような構成となる。



項番	クラス名	説明
(1)	Order	t_order テーブルの 1 レコードを表現する Entity クラス。 関連 Entity として、`OrderItem` および `OrderCoupon` を複数保持する。
(2)	OrderItem	t_order_item テーブルの 1 レコードを表現する Entity クラス。 関連 Entity として、 `Item` を保持する。
(3)	OrderCoupon	t_order_coupon テーブルの 1 コードを表現する Entity クラス。 関連 Entity として、`Coupon` を保持する。
(4)	Item	m_item テーブルの 1 コードを表現する Entity クラス。 関連 Entity として、所属している `Category` を複数保持する。 `Item` と `Category` の紐づけは、m_item_category テーブルによって行われる。
(5)	Category	m_category テーブルの 1 レコードを表現する Entity クラス。

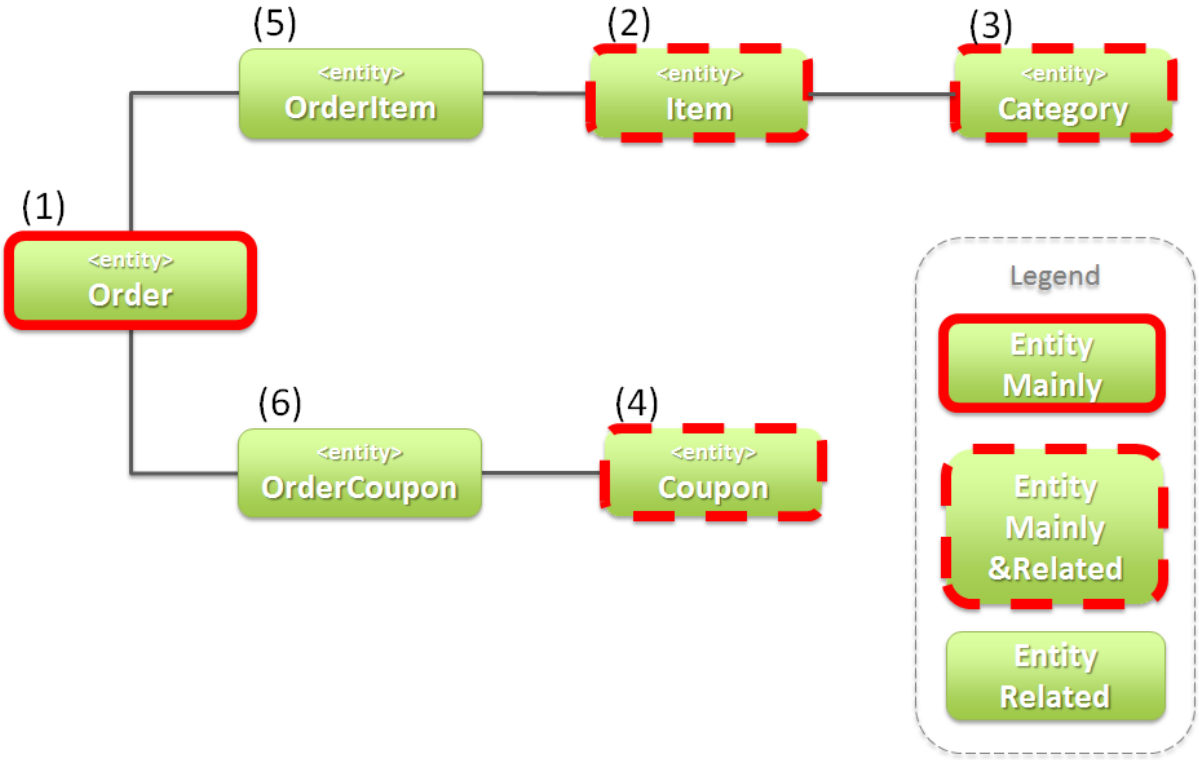
Continued on next page

表 3.7 – continued from previous page

項番	クラス名	説明
(6)	ItemCategory	m_item_category テーブルは、m_item テーブルと m_category テーブルとの関連を保持するためのマッピングテーブルなので、Entity クラスは作成しない。
(7)	Coupon	m_coupon テーブルの 1 レコードを表現する Entity クラス。
(8)	OrderStatus	c_order_status テーブルはコード系テーブルなので、Entity クラスは作成しない。

上記のエンティティ図をみると、ショッピングサイトのアプリケーションとして主体の Entity クラスとして扱われるのは、Order クラスのみと思ってしまうかもしれないが、主体となる得る Entity クラスは Order クラス以外にも存在する。

以下に、主体の Entity としてなり得る Entity と、主体の Entity にならない Entity を分類する。



ショッピングサイトのアプリケーションを作成する上で、主体の Entity としてなり得るのは、以下 4 つである。

項番	Entity クラス	主体の Entity となる得る理由
(1)	Order クラス	ショッピングサイトにおいて、最も重要な主体となる Entity クラスのひとつである。 Order クラスは、注文そのものを表現する Entity であり、Order クラスなくしてショッピングサイトを作成することはできない。
(2)	Item クラス	ショッピングサイトにおいて、最も重要な主体となる Entity クラスのひとつである。 Item クラスは、ショッピングサイトで扱っている商品そのものを表現する Entity であり、Item クラスなくしてショッピングサイトを作成することはできない。
(3)	Category クラス	一般的なショッピングサイトでは、トップページや共通的メニューとして、サイトで扱っている商品のカテゴリを表示している。 このようなショッピングサイトのアプリケーションでは、Category クラスを主体の Entity として扱うことになる。カテゴリの一覧検索などの処理が想定される。
(4)	Coupon クラス	ショッピングサイトにおいて、商品の販売促進を行う手段としてクーポンによる値引きを行うことがある。 このようなショッピングサイトのアプリケーションでは、Coupon クラスを主体の Entity として扱うことなる。クーポンの一覧検索などの処理が想定される。

ショッピングサイトのアプリケーションを作成する上で、主体の Entity とならないのは、以下 2 つである。

項番	Entity クラス	主体の Entity にならない理由
(5)	OrderItem クラス	このクラスは、1 つの注文で購入された商品 1 つを表現するクラスであり、Order クラスの関連 Entity としてのみ存在するクラスとなる。 そのため、OrderItem クラスが、主体の Entity として扱われることは原則ない。
(6)	OrderCoupon	このクラスは、1 つの注文で使用されたクーポン 1 つを表現するクラスであり、Order クラスの関連 Entity としてのみ存在するクラスとなる。 そのため、OrderCoupon クラスが主体の Entity として扱われることは原則ない。

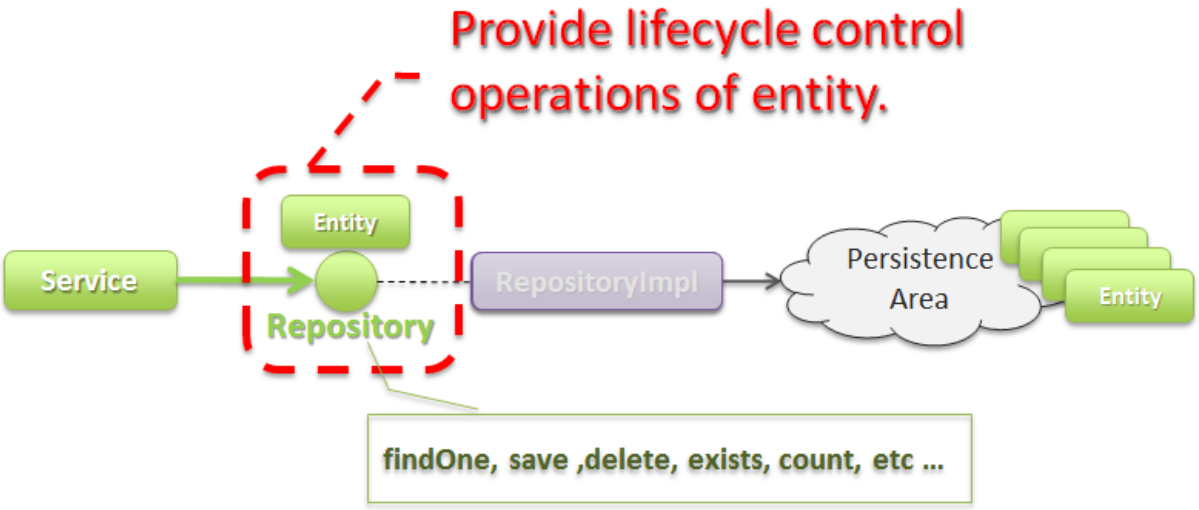
3.2.4 Repository の実装

Repository の役割

Repository は、以下 2 つの役割を担う。

1. **Service** に対して、**Entity** のライフサイクルを制御するための操作（**Repository** インタフェース）を提供する。

   Entity のライフサイクルを制御するための操作は、Entity オブジェクトへの CRUD 操作となる。



2. **Entity** を永続化する処理 (**Repository** インタフェースの実装クラス) を提供する。

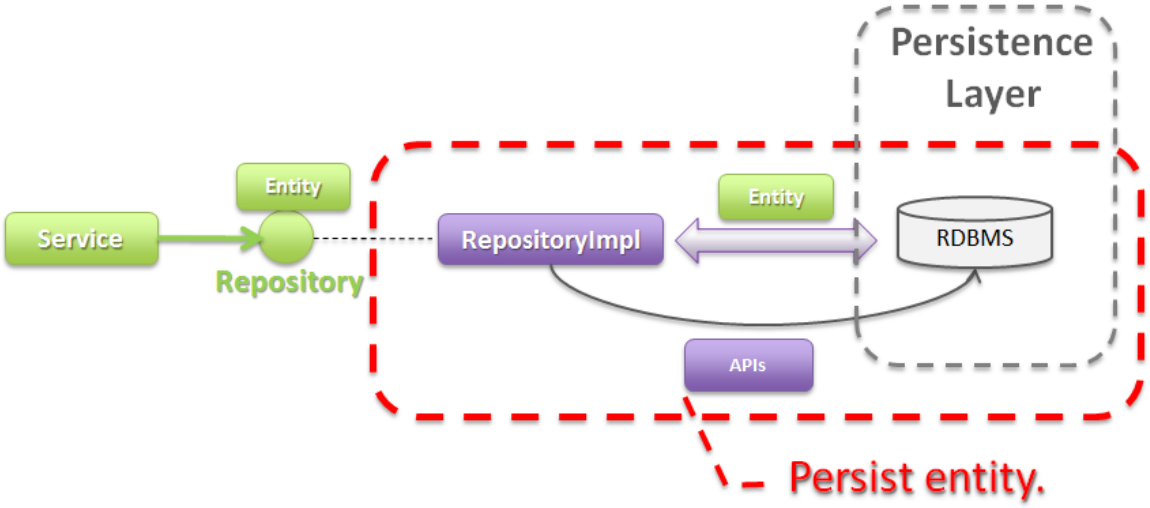
   Entity オブジェクトは、アプリケーションのライフサイクル (サーバの起動や、停止など) に依存しない

レイヤに、永続化しておく必要がある。

Entity の永続先は、リレーショナルデータベースになることが多いが、NoSQL データベース、キャッシュサーバ、外部システム、ファイル（共有ディスク）などになることもある。

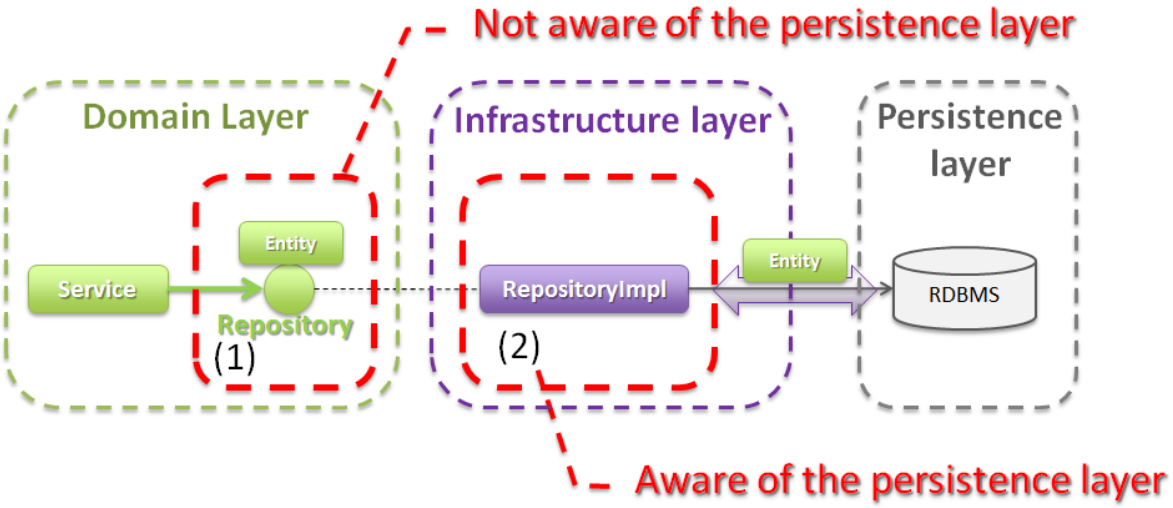
実際の永続化処理は、O/R Mapper などから提供されている API を使って行う。

この役割は、インフラストラクチャ層の RepositoryImpl で実装することになる。詳細については、インフラストラクチャ層の実装を参照されたい。



Repository の構成

Repository は、Repository インタフェースと RepositoryImpl で構成され、それぞれ以下の役割を担う。



項番	クラス (インタフェース)	役割	説明
(1)	Repository インタフェース	業務ロジック (Service) を実装する上で必要となる Entity のライフサイクルを制御するメソッドを定義する。	永続先に依存しない Entity の、CRUD 操作用のメソッドを定義する。 Repository インタフェースは、業務ロジック (Service) を実装する上で必要となる Entity の操作を定義する役割を担うので、ドメイン層に属することになる。
(2)	RepositoryImpl	Repository インタフェースで定義されたメソッドの実装を行う。	永続先に依存した Entity の CRUD 操作の実装を行う。実際の CRUD 処理は、Spring Framework、O/R Mapper、ミドルウェアなどから提供されている永続処理用の API を利用して行う。 RepositoryImpl は、Repository インタフェースで定義された操作の実装を行う役割を担うので、インフラストラクチャ層に属することになる。 RepositoryImpl の実装については、インフラストラクチャ層の実装を参照されたい。

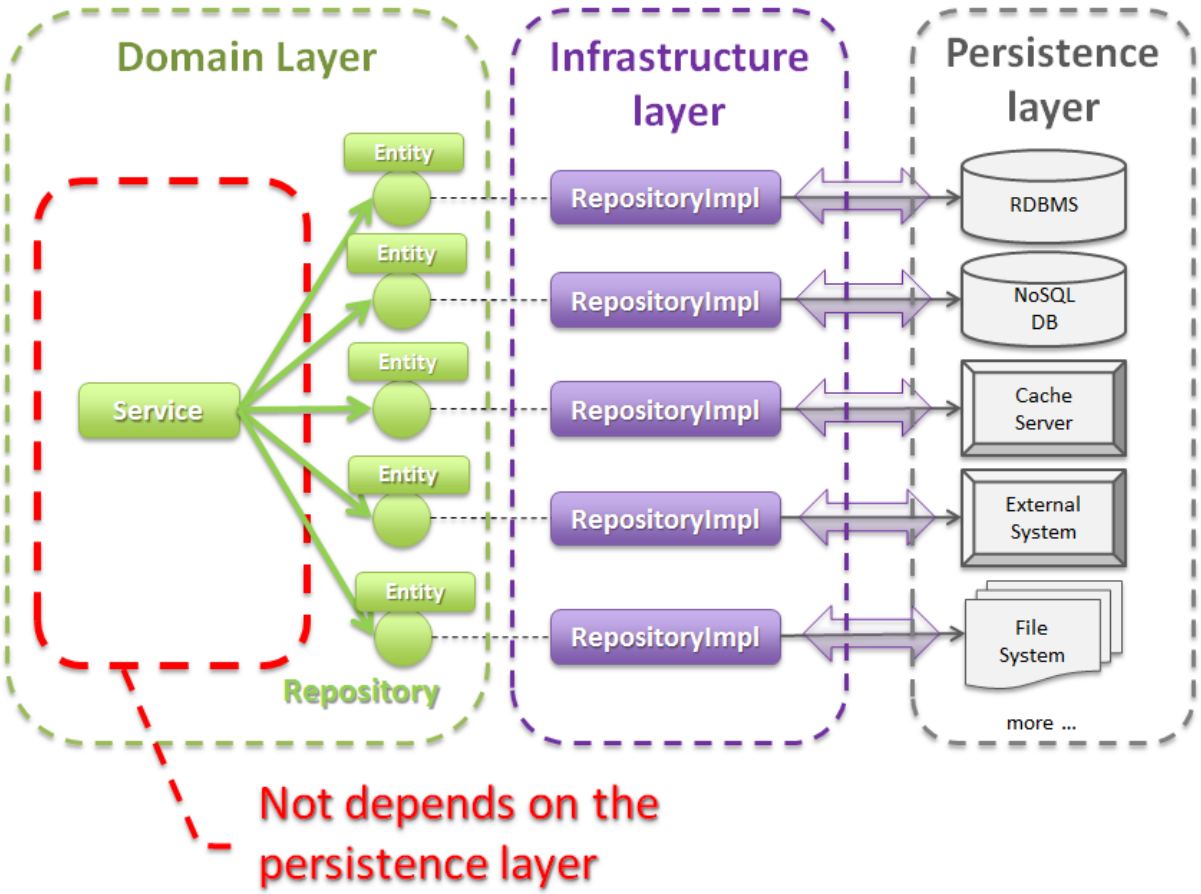
永続先が複数になる場合、以下のような構成となる。

以下のような構成を取ることで、Entity の永続先に依存したロジックを、業務ロジック (Service) から排除することができる。

注釈: 永続先に依存したロジックを、Service から 100 %排除できるのか？

永続先の制約や、使用するライブラリの制約などにより、排除できないケースもある。可能な限り、永続先に依存するロジックは、Service ではなく、RepositoryImpl で実装することを推奨するが、永続先に依存するロジックを排除するのが難しい場合や、排除することで得られるメリットが少ない場合は、無理に排除せず、業務ロジック (Service) の処理として、永続先に依存するロジックを実装してもよい。

排除できない具体例として、Spring Data JPA から提供されている `org.springframework.data.jpa.repository.JpaRepository` インタフェースの save メソッドの呼び出し時に、一意制約エラーをハンドリングしたい場合である。JPA では Entity への操作はキャッシュされ、トランザクショ



ンコミット時に SQL を発行する仕組みになっている。そのため、JpaRepository の save メソッドを呼び出しても、SQL は発行されないので、一意制約違反をロジックでハンドリングすることができない。JPA では、明示的に SQL を発行する手段として、キャッシュされている操作を反映するためのメソッド（flush メソッド）があり、JpaRepository では saveAndFlush、flush というメソッドが同じ目的で提供されている。そのため、Spring Data JPA の JpaRepository を使って、一意制約違反エラーをハンドリングする必要がある場合は、JPA 依存のメソッド（saveAndFlush や、flush）を呼び出す必要がある。

警告: Repository を設ける最も重要な目的は、永続先に依存するロジックを、業務ロジックから排除することではないという点である。最も重要な目的は、業務データへアクセスするための操作を Repository へ分離することで、業務ロジック (Service) の実装範囲をビジネスルールに関する実装に専念させるという点である。結果として、永続先に依存するロジックは業務ロジック (Service) ではなく、Repository 側に実装される事になる。

Repository の作成方針

Repository は原則以下の方針で作成する。

項番	方針	補足
1.	Repository は、主体となる Entity に対して作成する。	これは、関連 Entity を操作するためだけの Repository が不要であることを意味する。 ただし、アプリケーションの特性 (高い性能要件があるアプリケーションなど) では、関連 Entity を操作するための Repository を設けた方が、よい場合もある。
2.	Repository インタフェースと、RepositoryImpl は、基本的にドメイン層の同じパッケージに配置する。	Repository は、Repository インタフェースがドメイン層、RepositoryImpl がインフラストラクチャ層に属することとなるが、 Java のパッケージとしては、基本的には、ドメイン層の Repository インタフェースと同じパッケージでよい。
3.	Repository で使用する DTO は、Repository インタフェースと同じパッケージに配置する。	例えば、検索条件を保持する DTO や、Entity の一部の項目のみを定義したサマリ用の DTO などがあげられる。

Repository の作成例

Repository の作成例を説明する。

以下は、*Entity クラスの作成例*の説明で使用した、Entity クラスの Repository を作成する例となっている。

Repository 構成

*Entity クラスの作成例*の説明で使用した、Entity クラスの Repository を作成すると、以下のような構成となる。

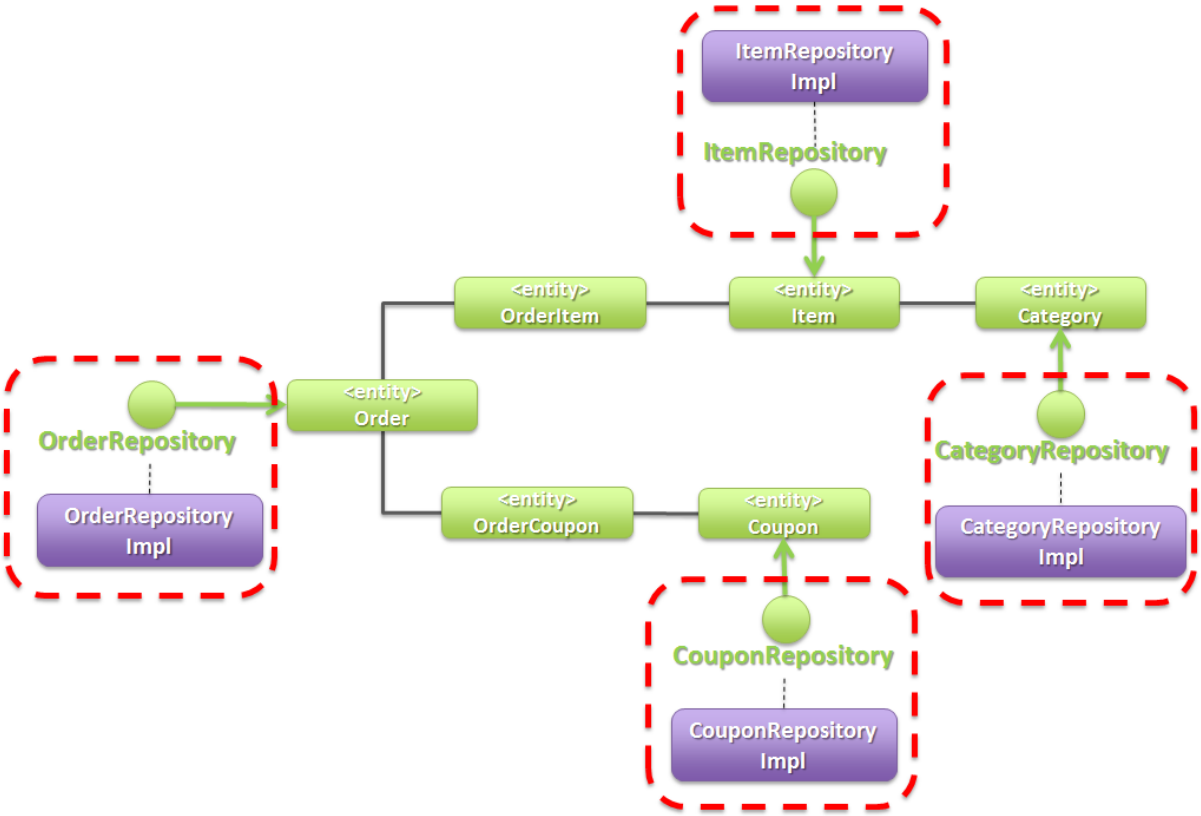
主体となる Entity クラスに対して、Repository を作成している。

パッケージの推奨構成については、プロジェクト構成を参照されたい。

Repository インタフェースの定義

Repository インタフェースの作成

以下に Repository インタフェースの作成例を紹介する。



- `SimpleCrudRepository.java`

  このインタフェースは、シンプルな CRUD 操作のみを提供している。

  メソッドのシグネチャは、Spring Data から提供されている `CrudRepository` インタフェースや、`PagingAndSortingRepository` インタフェースを参考に作成している。

```
public interface SimpleCrudRepository<T, ID extends Serializable> {
    // (1)
    T findOne(ID id);
    // (2)
    boolean exists(ID id);
    // (3)
    List<T> findAll();
    // (4)
    Page<T> findAll(Pageable pageable);
    // (5)
    long count();
    // (6)
    T save(T entity);
    // (7)
    void delete(T entity);
}
```

項番	説明
(1)	指定した ID に対応する Entity を、取得するためのメソッド。
(2)	指定した ID に対応する Entity が、存在するか判定するためのメソッド。
(3)	全ての Entity を取得するためのメソッド。 Spring Data では、`java.util.Iterable` であったが、サンプルとしては、`java.util.List` にしている。
(4)	指定したページネーション情報（取得開始位置、取得件数、ソート情報）に該当する Entity のコレクションを取得するためのメソッド。 `Pageable` インタフェースおよび `Page` インタフェースは Spring Data より提供されているクラス（インターフェース）である。
(5)	Entity の総件数を取得するためのメソッド。
(6)	指定された Entity のコレクションを保存（作成、更新）するためのメソッド。
(7)	指定した Entity を、削除するためのメソッド。

- `TodoRepository.java`

  下記は、チュートリアルで作成した Todo エンティティの Repository を、上で作成した `SimpleCrudRepository` インタフェースベースに作成した場合の例である。

```
// (1)
public interface TodoRepository extends SimpleCrudRepository<Todo, String> {
    // (2)
    long countByFinished(boolean finished);
}
```

項番	説明
(1)	エンティティの型を示すジェネリック型「T」に Todo エンティティ、エンティティの ID 型を示すジェネリック型「ID」に String クラスを指定することで、 Todo エンティティ用の Repository インタフェースが生成される。
(2)	`SimpleCrudRepository` インタフェースから提供されていないメソッドを追加している。 ここでは、「指定したタスクの終了状態に一致する Todo エンティティの件数を取得するメソッド」を追加している。

Repository インタフェースのメソッド定義

汎用的な CRUD 操作を行うメソッドについては、Spring Data から提供されている `CrudRepository` や、`PagingAndSortingRepository` と同じシグネチャにすることを推奨する。

ただし、コレクションを返却する場合は、`java.lang.Iterable` ではなく、ロジックで扱いやすいインタフェース（`java.util.Collection` や、`java.util.List`）でもよい。

実際のアプリケーション開発では、汎用的な CRUD 操作のみで開発できることは稀で、かならずメソッドの追加が必要になる。

追加するメソッドは、以下のルールに則り追加することを推奨する。

項番	メソッドの種類	ルール
1.	1 件検索系のメソッド	1. メソッド名は、条件に一致する Entity を、1 件取得するためのメソッドであることを明示するために、**findOneBy** で始める。 2. メソッド名の findOneBy 以降は、検索条件となるフィールドの物理名、または、論理的な条件名などを指定し、どのような状態の Entity が取得されるのか、推測できる名前とする。 3. 引数は、条件となるフィールド毎に用意する。ただし、条件が多い場合は、条件をまとめた DTO を用意してもよい。 4. 返り値は、Entity クラスを指定する。

Continued on next page

表 3.8 – continued from previous page

項番	メソッドの種類	ルール
2.	複数件検索系のメソッド	1. メソッド名は、条件に一致する Entity を、すべて取得するためのメソッドであることを明示するために、 **findAllBy** で始める。 2. メソッド名の findAllBy 以降は、検索条件となるフィールドの物理名または論理的な条件名を指定し、どのような状態の Entity が取得されるのか推測できる名前とする。 3. 引数は、条件となるフィールド毎に用意する。ただし、条件が多い場合は、条件をまとめた DTO を用意してもよい。 4. 返り値は、Entity クラスのコレクションを指定する。
3.	複数件ページ検索系のメソッド	1. メソッド名は、条件に一致する Entity の該当ページ部分を取得するためのメソッドである事を明示するために、 **findPageBy** で始める。 2. メソッド名の findPageBy 以降は、検索条件となるフィールドの物理名または論理的な条件名を指定し、どのような状態の Entity が取得されるのか推測できる名前とする。 3. 引数は、条件となるフィールド毎に用意する。ただし、条件が多い場合は、条件をまとめた DTO を用意してもよい。ページネーション情報（取得開始位置、取得件数、ソート情報）は、Spring Data より提供されている `Pageable` インタフェースとすることを推奨する。 4. 返り値は、Spring Data より提供されている `Page` インタフェースとすることを推奨する。
4.	件数のカウント系のメソッド	1. メソッド名は、条件に一致する Entity の件数をカウントするためのメソッドである事を明示するために、 **countBy** で始める。 2. 返り値は、long 型にする。 3. メソッド名の countBy 以降は、検索条件となるフィールドの物理名または論理的な条件名を指定し、どのような状態の Entity の件数が取得されるのか推測できる名前とする。 4. 引数は、条件となるフィールド毎に用意する。ただし、条件が多い場合は、条件をまとめた DTO を用意してもよい。

Continued on next page

表 3.8 – continued from previous page

項番	メソッドの種類	ルール
5.	存在判定系のメソッド	1. メソッド名は、条件に一致する Entity が存在するかチェックするためのメソッドである事を明示するために、 **existsBy** で始める。 2. メソッド名の existsBy 以降は、検索条件となるフィールドの物理名または論理的な条件名を指定し、どのような状態の Entity の存在チェックを行うのか推測できる名前とする。 3. 引数は、条件となるフィールド毎に用意する。ただし、条件が多い場合は、条件をまとめた DTO を用意してもよい。 4. 返り値は、boolean 型にする。

注釈: 更新系のメソッドも、同様のルールに則り、追加することを推奨する。find の部分が、update または delete となる。

- `Todo.java` (Entity)

```
public class Todo implements Serializable {
    private String todoId;
    private String todoTitle;
    private boolean finished;
    private Date createdAt;
    // ...
 }
```

- `TodoRepository.java`

```
public interface TodoRepository extends SimpleCrudRepository<Todo, String> {
    // (1)
    Todo findOneByTodoTitle(String todoTitle);
    // (2)
    List<Todo> findAllByUnfinished();
    // (3)
    Page<Todo> findPageByUnfinished();
    // (4)
    long countByExpired(int validDays);
    // (5)
    boolean existsByCreateAt(Date date);
}
```

項番	説明
(1)	タイトルが一致する TODO(todoTitle=引数で指定した値の TODO) を取得するメソッドの定義例。 findOneBy 以降に、条件となるフィールドの物理名 (todoTitle) を指定している。
(2)	未完了の TODO(finished=false の TODO) を全件取得するメソッドの定義例。 findAllBy 以降に、論理的な条件名を指定している。
(3)	未完了の TODO(finished=false の TODO) の該当ページ部分を取得するメソッドの定義例。 findPageBy 以降に、論理的な条件名を指定している。
(4)	完了期限を過ぎた TODO(createdAt < sysdate - 引数で指定した有効日数 && finished=false の TODO) の件数を取得するメソッドの定義例。 countBy 以降に、論理的な条件名を指定している。
(5)	指定日に作成されている、TODO(createdAt=指定日) が存在するか判定するメソッドの定義例。 existsBy 以降に、条件となるフィールドの物理名 (createdAt) を指定している。

RepositoryImpl の作成

RepositoryImpl の実装については、インフラストラクチャ層の実装を参照されたい。

3.2.5 Service の実装

Service の役割

Service は、以下 2 つの役割を担う。

1. **Controller に対して業務ロジックを提供する。**

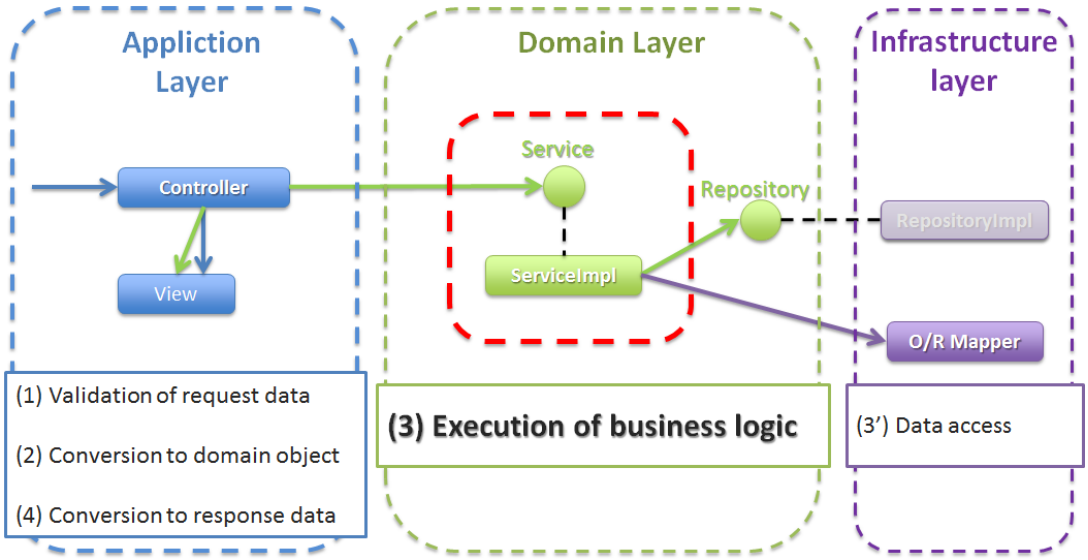
   業務ロジックは、アプリケーションで使用する業務データの参照、更新、整合性チェックおよびビジネスルールに関わる各種処理で構成される。

   業務データの参照および更新処理を Repository(または O/R Mapper) に委譲し、**Service ではビジネスルールに関わる処理の実装に専念することを推奨する。**

注釈: **Controller** と **Service** で実装するロジックの責任分界点について

本ガイドラインでは、Controller と Service で実装するロジックは、以下のルールに則って実装することを推奨する。

1. クライアントからリクエストされたデータに対する単項目チェック、相関項目チェックは Controller 側 (Bean Validation または Spring Validator) で行う。
2. Service に渡すデータへの変換処理 (Bean 変換、型変換、形式変換など) は、Service ではなく Controller 側で行う。
3. ビジネスルールに関わる処理は **Service** で行う。業務データへのアクセスは、Repository または O/R Mapper に委譲する。
4. Service から Controller に返却するデータ（クライアントへレスポンスするデータ）に対する値の変換処理 (型変換、形式変換など) は、Service ではなく、Controller 側（View クラスなど）で行う。



2. トランザクション境界を宣言する。

   データの一貫性を保障する必要がある処理（主にデータの更新処理）を行う業務ロジックの場合、トランザクション境界を宣言する。

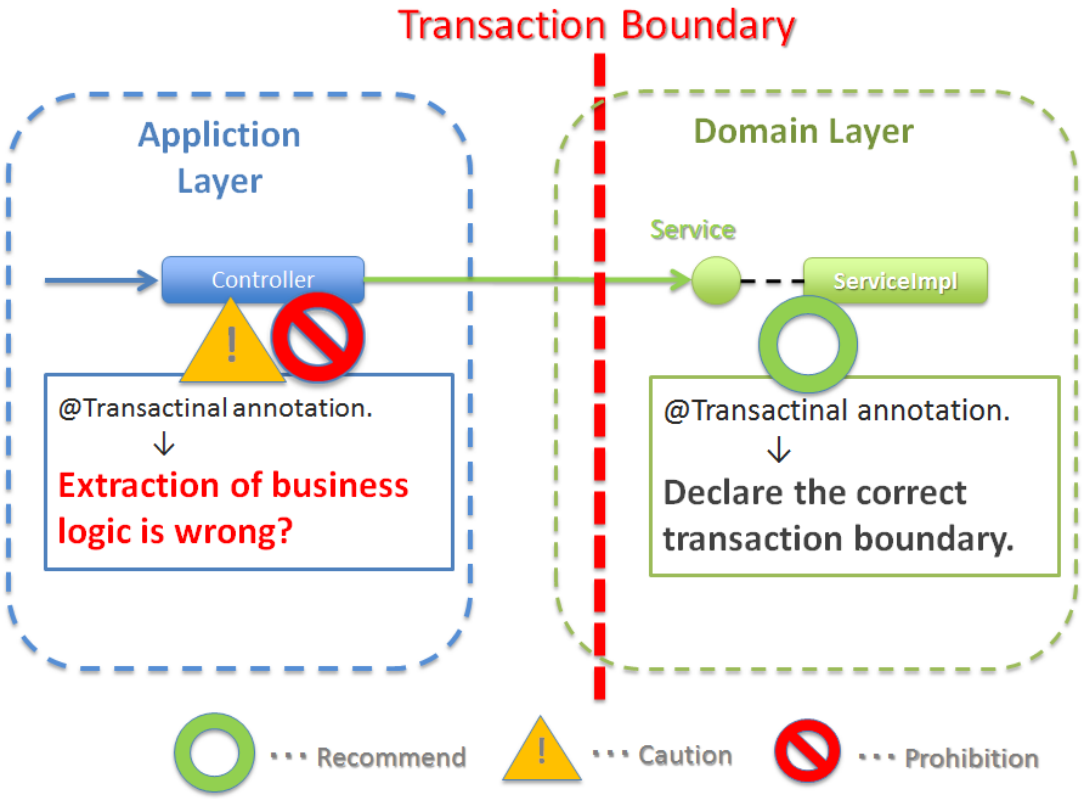
   データの参照処理の場合でも業務要件によっては、トランザクション管理が必要になる場合もあるので、その場合は、トランザクション境界を宣言する。

   トランザクション境界は、原則 Service に設ける。アプリケーション層 (Web 層) にトランザクション境界が設けられている場合、業務ロジックの抽出が正しく行われていない可能性があるので、見直しを行うこと。

   詳細は、トランザクション管理についてを参照されたい。

Service のクラス構成

Service は、Service クラスと SharedService クラスで構成され、それぞれ以下の役割を担う。

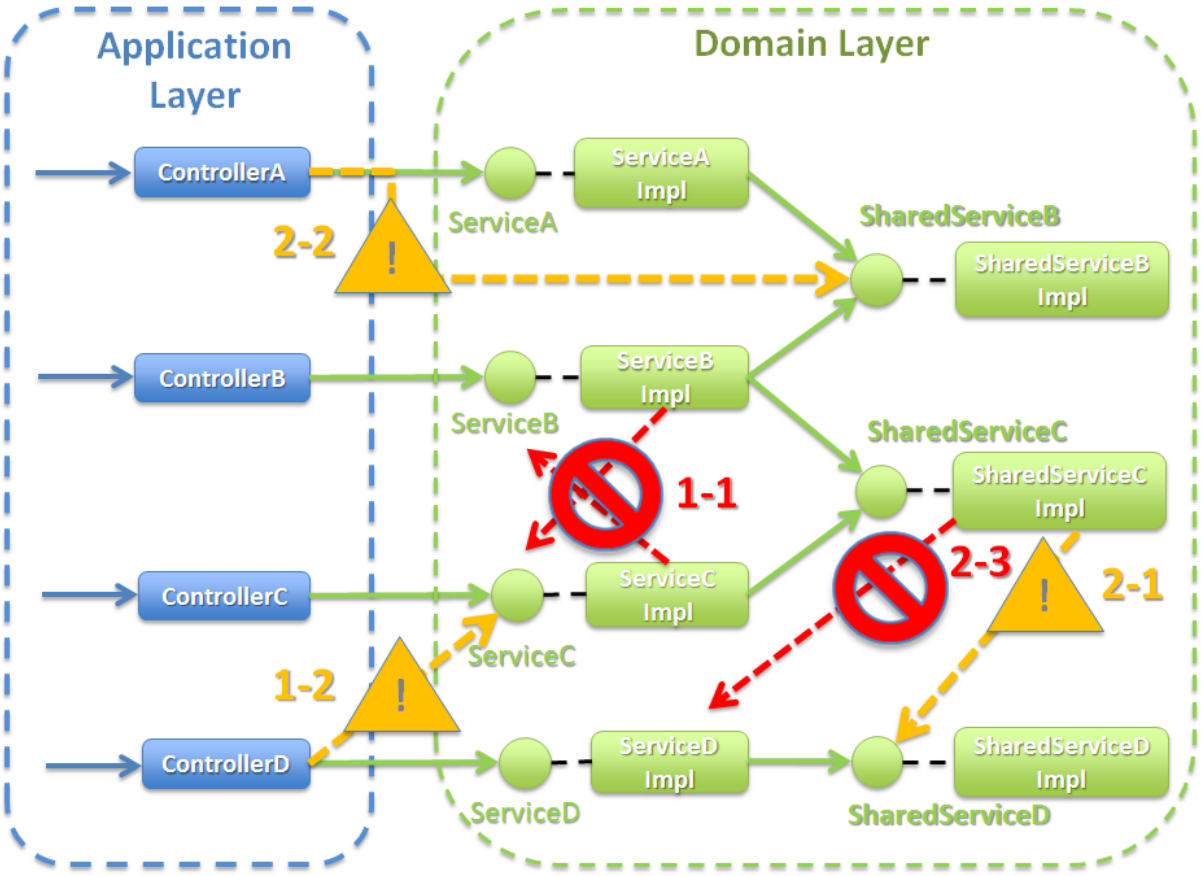


本ガイドラインでは、`@Service` アノテーションが付与された POJO(Plain Old Java Object) のことを、Service クラスおよび SharedService クラスと定義しているが、メソッドのシグネチャを限定するようなインタフェースや、基底クラスを作成することを、禁止しているわけではない。

項番	クラス	役割	依存関係に関する注意点
1.	Service クラス	特定の **Controller** に対して業務ロジックを提供する。 Service クラスのメソッドは、再利用されることを考慮したロジックは実装しない。	1. 他の **Service** クラスのメソッドを呼び出すことは、原則禁止とする（※図中 **1-1**）。他の Service と処理を共有したい場合は、SharedService クラスのメソッドを作成し、呼び出すようにすることを推奨する。 2. Service クラスのメソッドは、複数の Controller から呼び出してもよい（※図中 1-2）。ただし、呼び出し元の **Controller** によって、処理分岐が必要になる場合は、**Controller** 毎に、**Service** クラスのメソッドを作成することを推奨する。その上で共通的な処理は、SharedService クラスのメソッドを作成し呼び出すようにする。
2	SharedService クラス	複数の Controller や Service クラスで、共有 (再利用) されるロジックを提供する。	1. 他の SharedService クラスのメソッドを呼び出してもよいが（※図中 2-1）、 呼び出し階層が複雑にならないように考慮すること。 呼び出し階層が複雑になると保守性が低下する危険性が高まるので注意が必要。 2. Controller から SharedService クラスのメソッドを呼び出してもよい（※図中 2-2）が、トランザクション管理の観点で問題がない場合に限る。直接呼び出した場合に、トランザクション管理の観点で問題がある場合は、Service クラスにメソッドを用意し、適切なトランザクション管理が行われるようにすること。 3. SharedService クラスから **Service** クラスのメソッドを呼び出すことは禁止する（※図中 **2-3**）。

Service クラスと、SharedService クラスの依存関係を、以下に示す。

図中の番号は、上の表の「依存関係に関する注意点」欄の記載と連動しているため、あわせて確認すること。



Service クラスと SharedService クラスを分ける理由について

業務ロジックを構成する処理の中には、再利用できない (すべきでない) ロジックと再利用できる（すべき）ロジックが存在する。

この二つのロジックを、同じクラスのメソッドとして実装してしまうと、再利用してよいメソッドか否かの判断が、難しくなる。

この問題を回避する目的として、本ガイドラインでは、**再利用されることを想定しているメソッドについては、SharedService クラスに実装することを強く推奨している。**

Service クラスから、別の Service クラスの呼び出しを禁止する理由について

本ガイドラインでは、Service クラスのメソッドから、別の Service クラスのメソッドを呼び出すことを、原則禁止としている。

これは、Service クラスは、特定の Controller に対して業務ロジックを提供するクラスであり、別の Service から利用される前提で作成しないためである。

仮に、別の Service クラスから直接呼び出してしまうと、以下のような状況が発生しやすくなり、**保守性などを低下させる危険性が、高まる。**

項番	発生しうる状況
1.	本来は、呼び出し元の Service クラスで実装すべきロジックが、処理を一ヶ所にまとめたいという理由などにより、呼び出し先の Service クラスで実装されてしまう。 その際に、呼び出し元を意識するための引数（フラグ）などが、安易に追加され、間違った共通化が行われてしまう。結果として、見通しの悪いモジュール構成になってしまう。
2.	呼び出し経路やパターンが多くなることで、仕様変更や、バグ改修の際のソース修正に対する影響範囲の把握が難しくなる。

メソッドのシグネチャを限定するようなインタフェースや基底クラスについて

業務ロジックの作りを統一したい場合に、シグネチャを限定するようなインタフェースや、基底クラスを作成することがある。
シグネチャを限定するインタフェースや基底クラスを設けることで、開発者ごとに、作りの違いが発生しないようにする目的もある。

注釈: 大規模開発において、サービスイン後の保守性等を考慮して業務ロジックの作りを合わせておきたい場合や、開発者のスキルがあまり高くない場合などの状況下では、シグネチャを限定するようなインタフェースを設けることも、選択肢の一つとして考えてもよい。

本ガイドラインでは、シグネチャを限定するようなインタフェースを作成することは、特に推奨していないが、プロジェクトの特性を加味して、どのようなアーキテクチャにするか決めて頂きたい。

注釈: シグネチャを制限するインタフェースおよび基底クラスの実装サンプル- シグネチャを限定するようなインタフェース

```
// (1)
public interface BLogic<I, O> {
  O execute(I input);
}
```

項番	説明
(1)	業務ロジックの実装メソッドのシグニチャを制限するためのインタフェース。 上記例では、入力情報 (I) と出力情報 (O) の総称型として定義されており、 業務ロジックを実行するためのメソッド (execute) を一つもつ。 本ガイドラインでは、上記のようなインタフェースを、BLogic インタフェースと呼ぶ。

定型的な共通処理を Service に盛り込む場合、ビジネスロジックの処理フローを統一したい場合に、メソッドのシグネチャを限定するような基底クラスを作成することがある。

- シグネチャを限定するような基底クラス

```
// (2)
@Service
@Transactional
public abstract class AbstractBLogic<I, O> implements BLogic<I, O> {

    public O execute(I input){
      try{

          // omitted

          // (3)
          preExecute(input);

          // (4)
          O output = doExecute(input);

          // omitted

          return output;
      } finally {
          // omitted
      }

    }

    protected abstract void preExecute(I input);

    protected abstract O doExecute(I input);

}
```

項番	説明
(2)	基底クラスを作成する場合、*@Transactional* の仕様上、AOP の対象となるのは外部から実行されるメソッドもしくはメソッドを実装しているクラスであるため、トランザクション制御が必要な場合はこの基底クラスに付与する。 *@Servicve* も同様に、*ResultMessagesLoggingInterceptor* のように AOP によって Service を対象とするような場合はこの基底クラスに付与する必要がある。
(3)	基底クラスより、業務ロジックを実行する前の、事前処理を行うメソッドを呼び出す。 上記のような事前処理を行うメソッドでは、ビジネスルールのチェックなどを実装することになる。
(4)	基底クラスより、業務ロジックを実行するメソッドを呼び出す。

以下に、シグネチャを限定するような、基底クラスを継承する場合の、サンプルを示す。

- BLogic クラス (Service)

```
// (5)
public interface XxxBLogic extends BLogic<XxxInput, XxxOutput> {

}
```

項番	説明
(5)	タイプセーフなインジェクションを可能にするために、BLogic インタフェースを継承したインタフェースを作成する。 親インタフェースのメソッド経由での呼び出しを行うために、BLogic を継承したサブインタフェースを実装する。

```
@Service
public class XxxBLogicImpl extends AbstractBLogic<XxxInput, XxxOutput>␣
↪implements XxxBLogic {

    // (6)
    @Override
    protected void preExecute(XxxInput input) {
```

```
        // omitted
        Tour tour = tourRepository.findOne(input.getTourId());
        Date reservationLimitDate = tour.reservationLimitDate();
        if(input.getReservationDate().after(reservationLimitDate)){
            throw new BusinessException(ResultMessages.error().add("e.xx.
↪xx.0001"));
        }

    }

    // (7)
    @Override
    protected XxxOutput doExecute(XxxInput input) {
        TourReservation tourReservation = new TourReservation();

        // omitted

        tourReservationRepository.save(tourReservation);
        XxxOutput output = new XxxOutput();
        output.setTourReservation(tourReservation);

        // omitted
        return output;
    }

}
```

項番	説明
(6)	業務ロジックを実行する前の事前処理を実装する。 ビジネスルールのチェックなどを実装する事になる。
(7)	業務ロジックを実装する。 ビジネスルールを充たすために、ロジックを実装する事になる。

- Controller

```
// (8)
@Inject
XxxBLogic xxxBLogic;

public String reserve(XxxForm form, RedirectAttributes redirectAttributes)
↪{

    XxxInput input = new XxxInput();
    // omitted
```

```
    // (9)
    XxxOutput output = xxxBlogic.execute(input);

    // omitted

    redirectAttributes.addFlashAttribute(output.getTourReservation());
    return "redirect:/xxx?complete";
}
```

項番	説明
(8)	Controller は、呼び出す BLogic インタフェースを Inject する。
(9)	Controller は、BLogic インタフェースの execute メソッドを呼び出し、業務ロジックを実行する。

Service の作成単位

Service の作成単位は主に以下の 3 パターンとなる。

項番	単位	作成方法	特徴
1.	Entity 毎	主体となる Entity と対で Service を作成する。	主体となる Entity とは、業務データの事であり、**業務データを中心にしてアプリケーションを設計・実装する場合は、この単位で Service を作成することを推奨する。** この単位で Service を作成すると、業務データ毎に業務ロジックが集約されるため、業務処理の共通化が図られやすい。 ただし、このパターンで Service を作成した場合、同時に大量の開発者を投入して作成するアプリケーションとの相性は、あまりよくない。どちらかと言うと、小規模・中規模のアプリケーションを開発する場合に向いているパターンと言える。

Continued on next page

表 3.9 – continued from previous page

項番	単位	作成方法	特徴
2.	ユースケース毎	ユースケースと対で Service を作成する。	**画面からのイベントを中心にしてアプリケーションを設計・実装する場合は、この単位で Service を作成することになる。** この単位で Service を作成する場合は、ユースケース毎に担当者を割り当てることが出来るため、同時に大量の開発者を投入して開発するアプリケーションとの相性はよい。 一方で、このパターンで Service を作成すると、ユースケース内での業務ロジックの共通化は行うことができるが、ユースケースを跨いだ業務ロジックの共通化は行われない可能性が高くなる。 ユースケースを跨いで業務ロジックの共通化を行う必要がある場合は、共通化を行うための共通チームを設けるなどの工夫が必要となる。
3	イベント毎	画面から発生するイベントと対で Service を作成する。	**画面からのイベントを中心にしてアプリケーションを設計・実装する場合で且つ「TERASOLUNA ViSC」を使用して BLogic クラスを生成する場合は、この単位で Service を作成することになる。** 本ガイドラインでは、このような単位で作成される Service クラスの事を、BLogic と呼ぶ。 この単位で Service を作成する場合の特徴としては、基本的にはユースケース毎に作成する際と同じである。 ただし、イベント毎に Service クラスを設計・実装する事になるため、ユースケース毎に作成する場合に比べて、より共通化が行われない可能性が高くなる。 本ガイドラインとしては、イベント毎に作成するパターンは特に推奨しない。ただし、大規模開発において、保守性等を考慮して業務ロジックの作りを合わせておきたいといった理由がある場合は、イベント毎に作成する事を選択肢の一つとして考えてもよい。

警告：Service の作成単位については、開発するアプリケーションの特性や開発体制などを加味して決めて頂きたい。

また、提示した 3 つの作成パターンの **どれか一つのパターンに絞る必要はない。**無秩序にいろいろな単位の Service を作成する事は避けるべきだが、**アーキテクトによって方針が示されている状況下においては、併用しても特に問題はない。**例えば、以下のような組み合わせが考えられる。

【組み合わせて使用する場合の例】

- アプリケーションとして重要な業務ロジックについては、Entity 毎の SharedService クラスとして作成する。
- 画面からのイベントを処理するための業務ロジックについては、Controller 毎の Service クラスとして作成する。
- Controller 毎の Service クラスでは、必要に応じて SharedService クラスのメソッドを呼び出す事で業務ロジックを実装する。

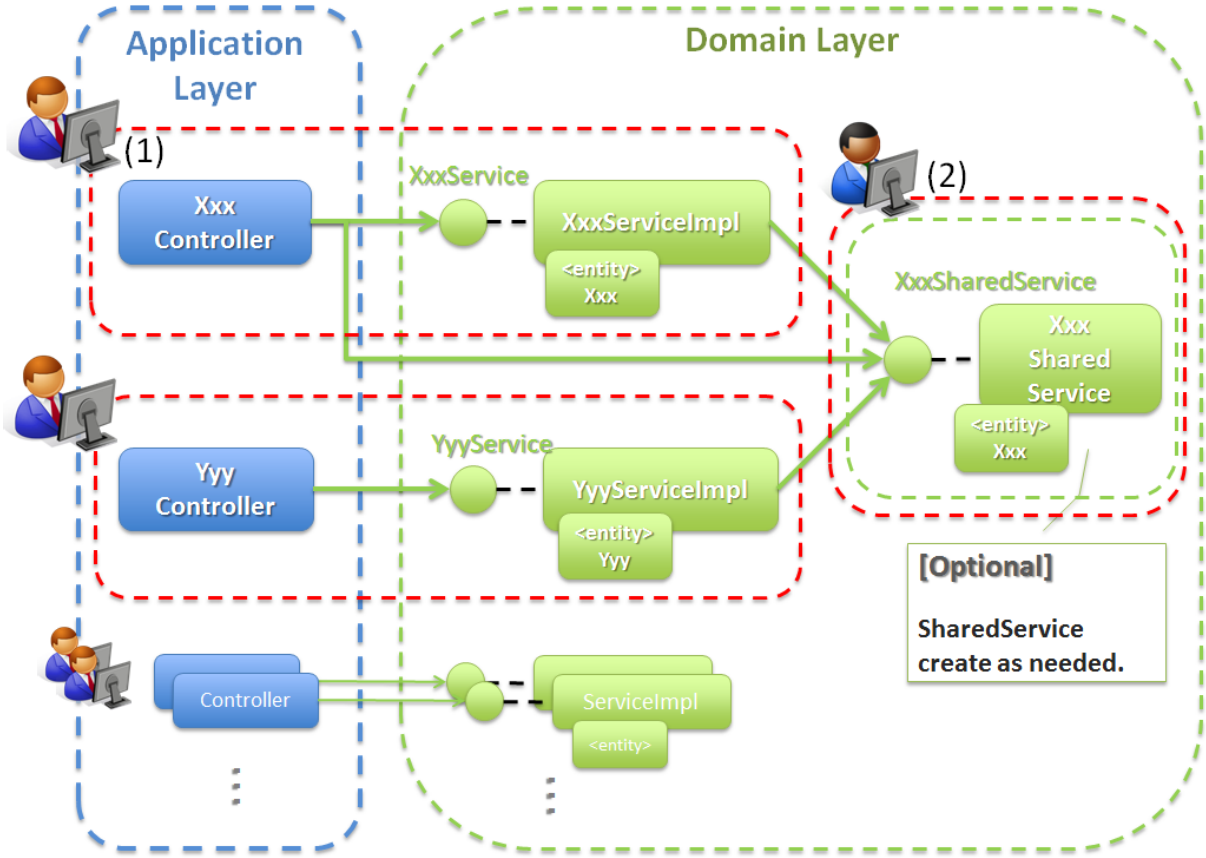
ちなみに：「TERASOLUNA ViSC」を使用する場合は、BLogic は設計書から出力される。

Entity 毎に Service を作成する際の開発イメージ

Entity 毎に Service を作成する場合は、以下のような開発イメージとなる。

注釈： Entity 毎に Service を作成する代表的なアプリケーションの例としては、REST アプリケーションがあげられる。REST アプリケーションは、HTTP 上に公開するリソースに対して CRUD 操作 (HTTP の POST, GET, PUT, DELETE) を提供する事になる。HTTP 上に公開するリソースは、業務データ (Entity) または業務データ (Entity) の一部となる事が多いため、Entity 毎に Service を作成する方法との相性がよい。

REST アプリケーションの場合は、ユースケースが Entity 毎に抽出されることが多い。そのため、ユースケース毎に作成する際の構成イメージと似た構成となる。



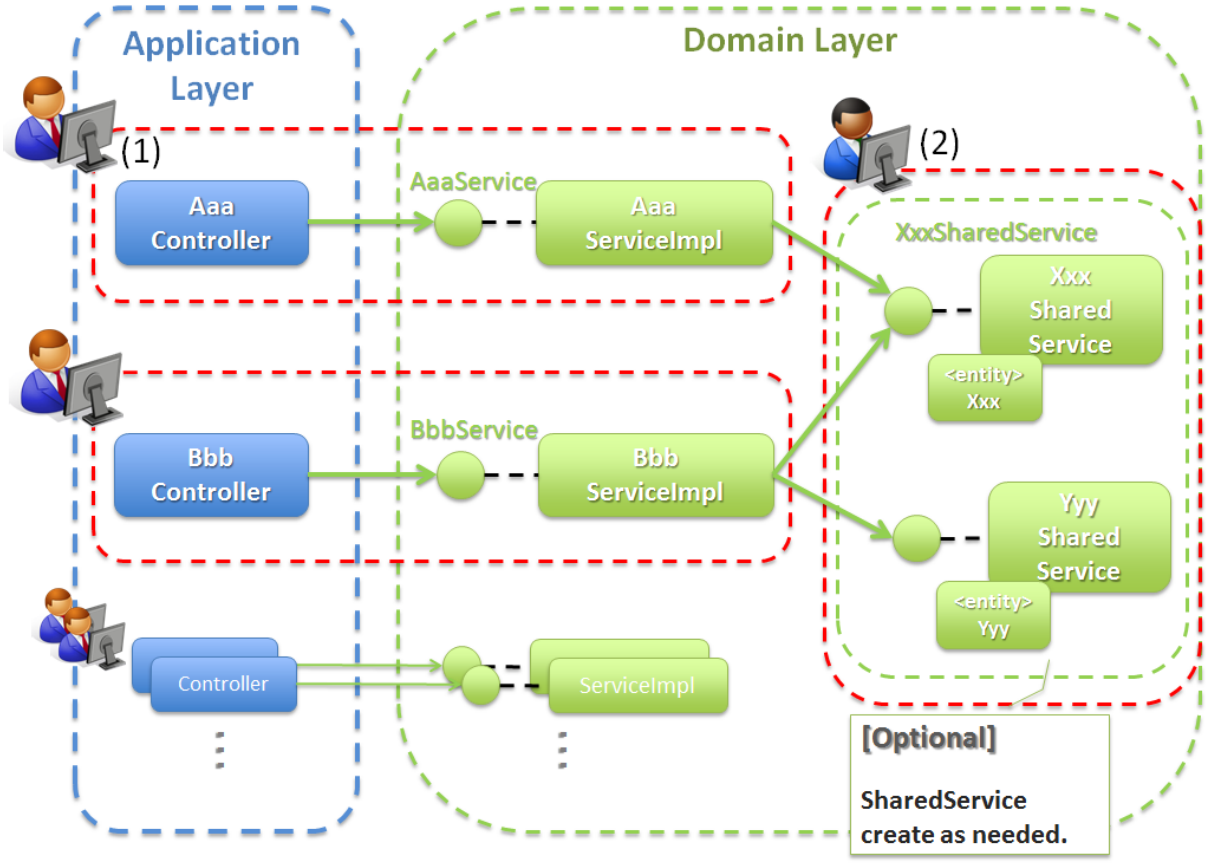
項番	説明
(1)	Entity 毎に開発者を割り当てて、Service を実装する。 特に理由がない場合は、Controller も Entity 毎に作成し、Service と同じ開発者を担当者にすることが望ましい。
(2)	複数の業務ロジックで共有したいロジックがある場合は、SharedService に実装する。 上の図では、別の開発者 (共通チームの担当者) を割り当てているが、プロジェクトの体制によっては (1) と同じ開発者でもよい。

ユースケース毎に作成する際の開発イメージ

ユースケース毎に Service を作成する場合は、以下のような開発イメージとなる。

Entity の CRUD 操作を行う様なユースケースの場合は、Entity 毎に Service を作成する際の構成イメージと同

じ構成となる。

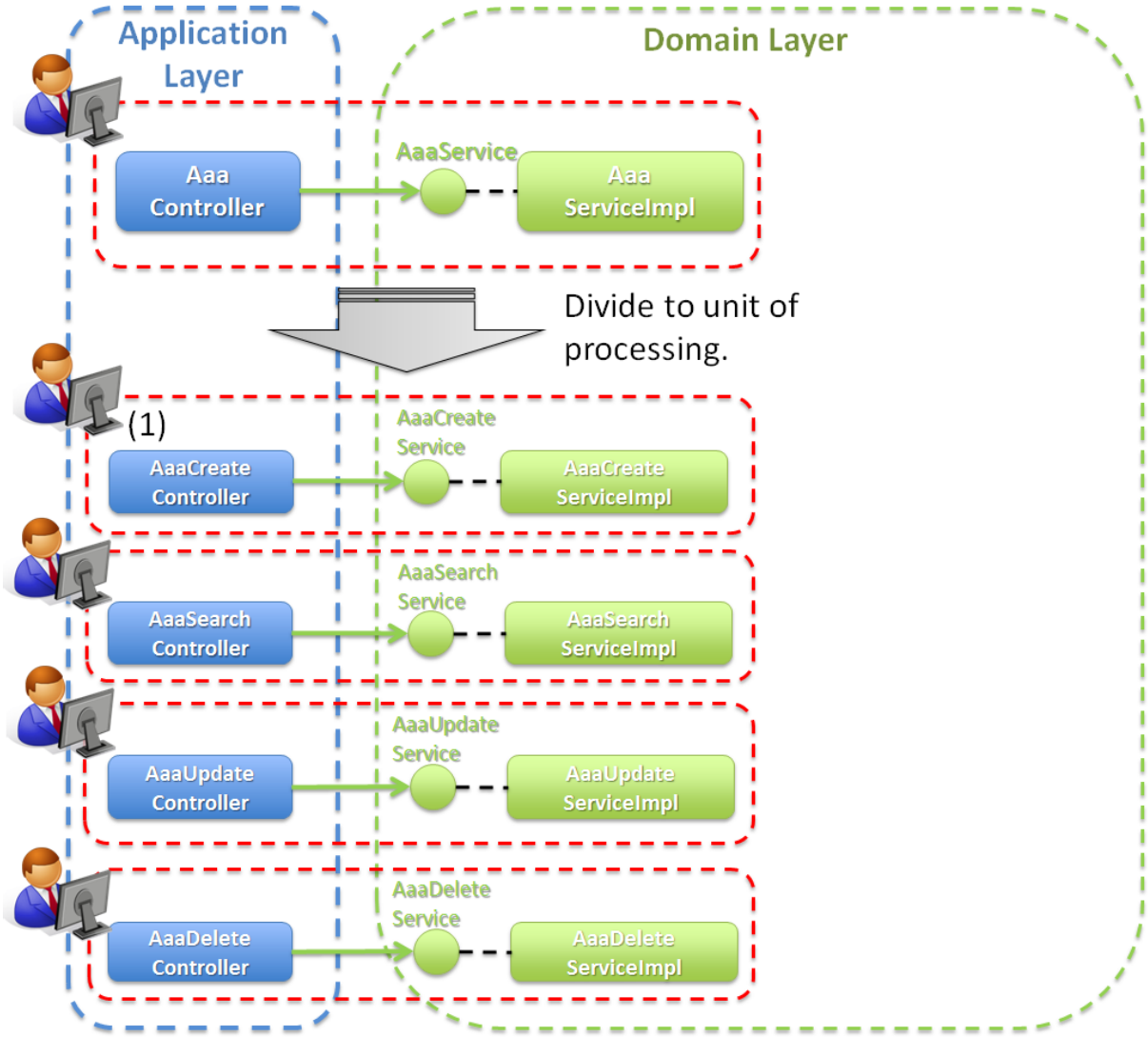


項番	説明
(1)	ユースケース毎に開発者を割り当てて、Service を実装する。 特に理由がない場合は、Controller もユースケース毎に作成し、Service と同じ開発者を担当者にすることが望ましい。
(2)	複数の業務ロジックで共有したいロジックがある場合は、SharedService に実装する。 上の図では、別の開発者 (共通チームの担当者) を割り当てているが、プロジェクトの体制によっては (1) と同じ開発者でもよい。

注釈: ユースケースの規模が大きくなると、一人が担当する開発範囲が大きくなるため、作業分担しづらくなる。同時に大量の開発者を投入して開発するアプリケーションの場合は、ユースケースを更に分割して、担当者を割り当てる事を検討すること。

ユースケースを更に分割した場合は、以下のような開発イメージとなる。

ユースケースの分割を行うことで、SharedService に影響はないため、説明は割愛している。

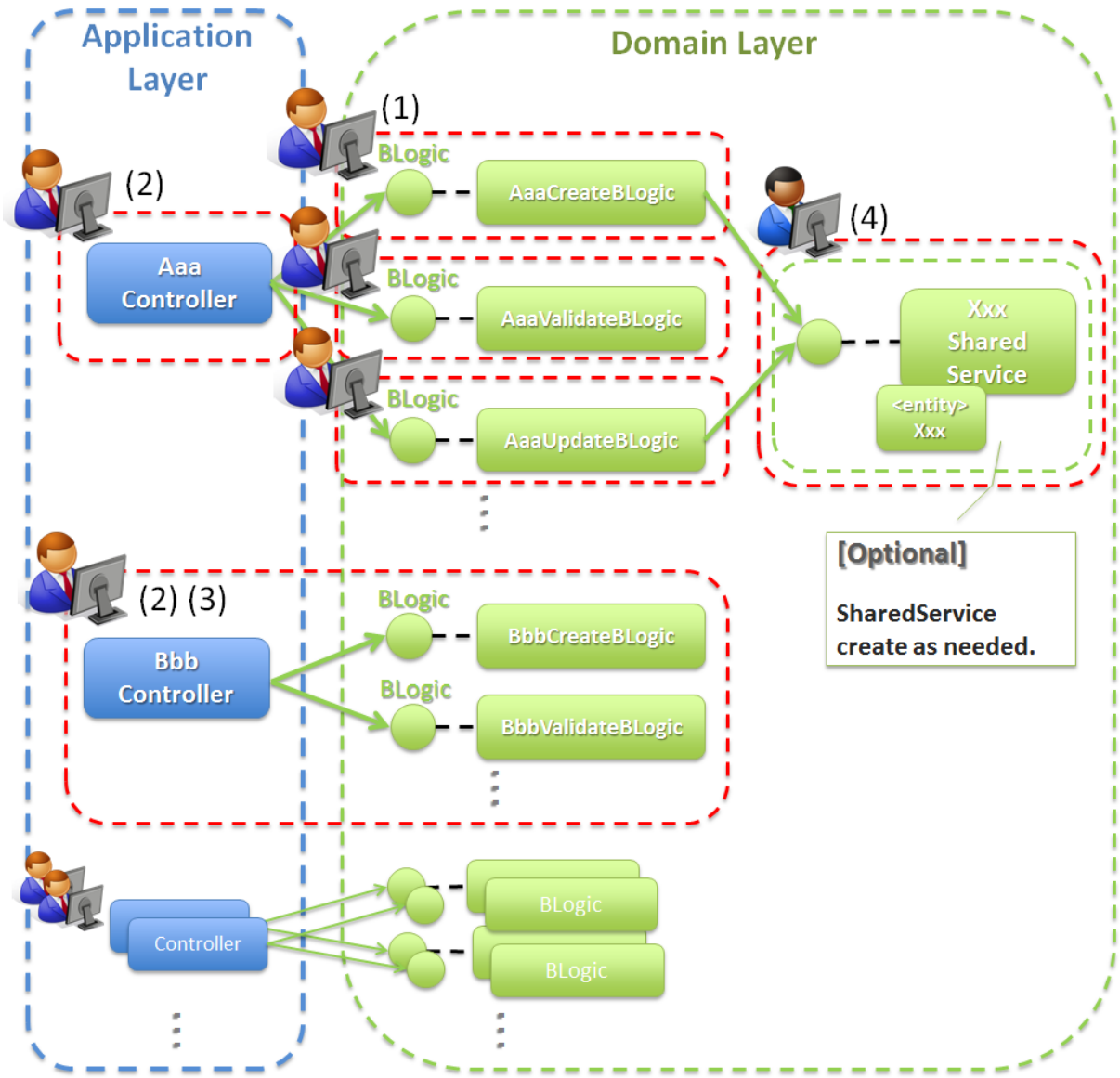


項番	説明
(1)	ユースケースを構成する処理単位に分割し、処理毎に開発者を割り当てて、Service を実装する。 ここで言う処理とは、検索処理、登録処理、更新処理、削除処理といった単位であり、画面から発生するイベント毎の処理ではない点に注意すること。 例えば「更新処理」であれば、「更新対象データの取得」や「更新内容の妥当性チェック」といった単位の処理が複数含まれる。 特に理由がない場合は、Controller も処理毎に作成し、Service と同じ開発者を担当者にすることが望ましい。

ちなみに： 本ガイドライン上で使っている「ユースケース」と「処理」の事を、「ユースケースグループ」と「ユースケース」と呼ぶプロジェクトもある。

イベント毎に作成する際の開発イメージ

イベント毎に Service(BLogic) を作成する場合は、以下のような開発イメージとなる。



項番	説明
(1)	イベント毎に開発者を割り当てて、Service(BLogic) を実装する。 上記例ではそれぞれ別の担当者を割り当てる図になっているが、これは極端な例である。 実際は、ユースケース毎に担当者を割り当てる事になる。
(2)	特に理由がない場合は、Controller はユースケース毎に作成することが望ましい。
(3)	イベント毎に Service(BLogic) を実装する場合でも、担当者はユースケース毎に割り当てることを推奨する。

Continued on next page

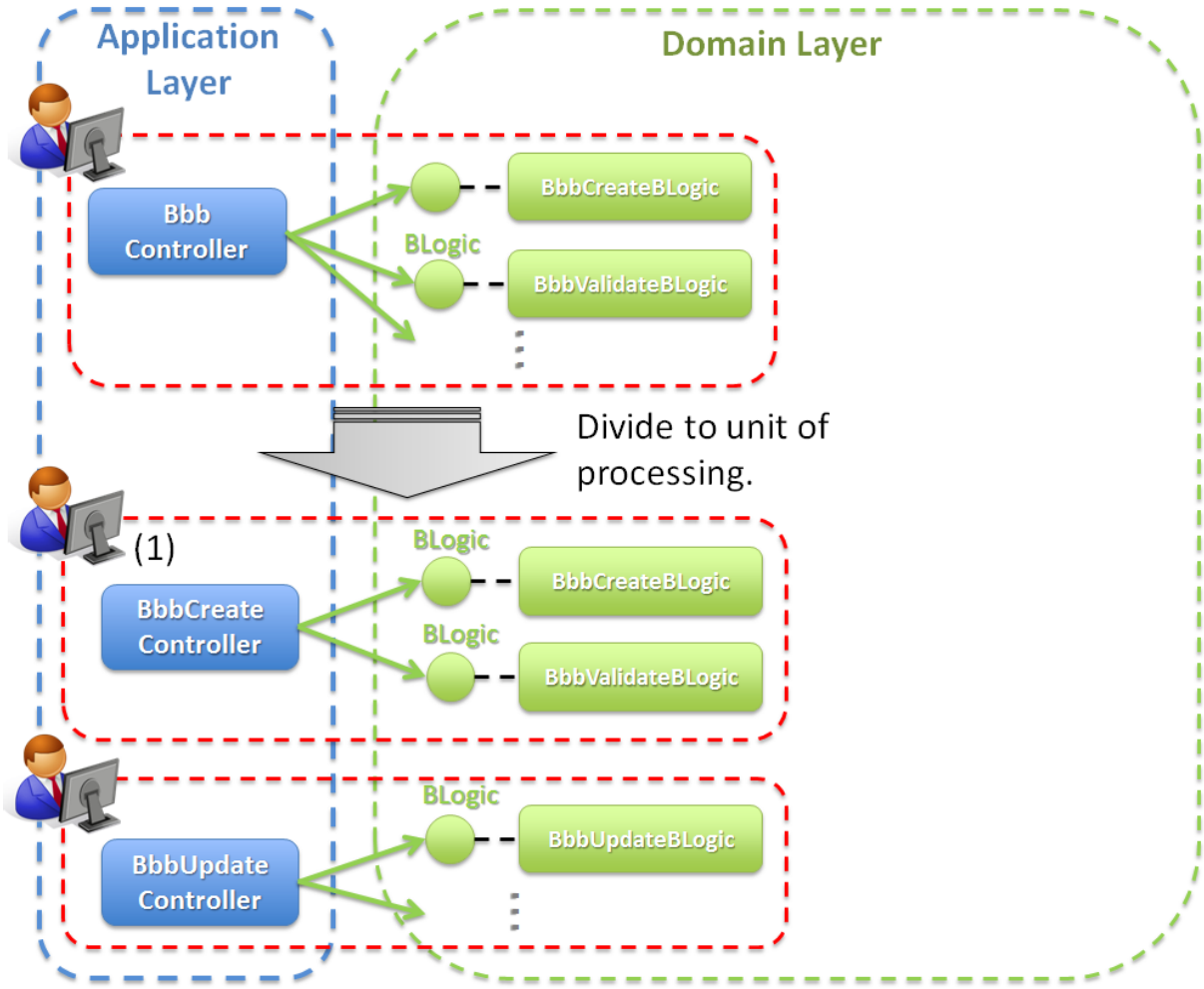
表 3.10 – continued from previous page

項番	説明
(4)	複数の業務ロジックで共有したいロジックがある場合は、SharedService に実装する。 上の図では、別の開発者 (共通チームの担当者) を割り当てているが、プロジェクトの体制によっては (1) と同じ開発者でもよい。

注釈: ユースケースの規模が大きくなると、一人が担当する開発範囲が大きくなるため、作業分担しづらくなる。同時に大量の開発者を投入して開発するアプリケーションの場合は、ユースケースを更に分割して、担当者を割り当てる事を検討すること。

ユースケースを更に分割した場合は、以下のような開発イメージとなる。

ユースケースの分割を行うことで、SharedService に影響はないため、説明は割愛している。



項番	説明
(1)	ユースケースを構成する処理単位に分割し、処理毎に開発者を割り当てて、Service(BLogic)を実装する。 ここで言う処理とは、検索処理、登録処理、更新処理、削除処理といった単位であり、画面から発生するイベント毎の処理ではない点に注意すること。 例えば「更新処理」であれば、「更新対象データの取得」や「更新内容の妥当性チェック」といった単位の処理が複数含まれる。 特に理由がない場合は、Controller も処理毎に作成し、Service と同じ開発者を担当者にすることが望ましい。

Service クラスの作成

Service クラスの作成方法

Service クラスを作成する際の注意点を、以下に示す。

- Service インタフェースの作成

```
public interface CartService { // (1)
    // omitted
}
```

項番	説明
(1)	**Service インタフェースを作成することを推奨する。** インタフェースを設けることで、Service として公開するメソッドを明確にすることが出来る。

注釈: アーキテクチャ観点でのメリット例

1. AOP を使う場合に、JDK 標準の Dynamic proxies 機能が使われる。インタフェースがない場合は Spring Framework に内包されている CGLIB が使われるが、final メソッドに対して Advice できないなどの制約がある。詳細は、Spring Reference Document -Aspect Oriented Programming with Spring(Proxying mechanisms)-を参照されたい。
2. 業務ロジックをスタブ化しやすくなる。アプリケーション層とドメイン層を別々の体制で並行して開発する場合は、アプリケーション層を開発するために、Service のスタブが必要になるケースがある。スタブを作成する必要がある場合は、インタフェースを設けておくことを推奨する。

- Service クラスの作成

```
@Service // (1)
@Transactional // (2)
public class CartServiceImpl implements CartService { // (3) (4)
    // omitted
}
```

```
<context:component-scan base-package="xxx.yyy.zzz.domain" /> <!-- (1) -->
```

項番	説明
(1)	**クラスに @Service アノテーションを付加する。** アノテーションを付与することで、component が scan 対象となり、設定ファイルへの bean 定義が、不要となる。 <context:component-scan>要素の base-package 属性に、component を scan する対象のパッケージを指定する。 上記設定の場合、「xxx.yyy.zzz.domain」パッケージ配下に格納されているクラスが、コンテナに登録される。
(2)	**クラスに @Transactional アノテーションを付加する。** アノテーションを付与することで、すべての業務ロジックに対してトランザクション境界が設定される。 属性値については、要件に応じた値を指定すること。 詳細は、「宣言型トランザクション管理」で必要となる情報を参照されたい。 また、`@Transactional` アノテーションを使用する際の注意点を理解するために、「トランザクション管理を使うための設定について」を合わせて確認するとよい。
(3)	**インターフェース名は XxxService、クラス名は XxxServiceImpl とする。** 上記以外の命名規約でもよいが、Service クラスと SharedService クラスは、区別できる命名規約を設けることを推奨する。
(4)	**Service クラスでは状態は保持せず、singleton スコープの bean としてコンテナに登録する 。** フィールド変数には、スレッド毎に状態が変わるオブジェクト (Entity/DTO/VO などの POJO) や、値 (プリミティブ型、プリミティブラッパークラスなど) を保持してはいけない。 また、`@Scope` アノテーションを使って singleton 以外のスコープ (prototype, request, session) にしてはいけない。

注釈: クラスに @Transactional アノテーションを付加する理由

トランザクション境界の設定が必須なのは更新処理を含む業務ロジックのみだが、設定漏れによるバグを防ぐ事を目的として、クラスレベルにアノテーションを付与することを推奨している。もちろん必要な箇所（更新処理を行うメソッド）のみに、`@Transactional` アノテーションを定義する方法を採用してもよい。

注釈: singleton 以外のスコープを禁止する理由

1. prototype, request, session は、状態を保持する bean を登録するためのスコープであるため、Service クラスに対して使用すべきでない。
2. スコープを request や prototype にした場合、DI コンテナによる bean の生成頻度が高くなるため、性能に影響を与えることがある。
3. スコープを request や session にした場合、Web アプリケーション以外のアプリケーション (例えば、Batch アプリケーションなど) で使用できなくなる。

Service クラスのメソッドの作成方法

Service クラスのメソッドを作成する際の注意点を、以下に示す。

- Service インタフェースのメソッド作成

```
public interface CartService {
    Cart createCart(); // (1) (2)
    Cart findCart(String cartId); // (1) (2)
}
```

- Service クラスのメソッドの作成

```
@Service
@Transactional
public class CartServiceImpl implements CartService {

    @Inject
    CartRepository cartRepository;

    public Cart createCart() { // (1) (2)
        Cart cart = new Cart();
        // ...
        cartRepository.save(cart);
        return cart;
    }

    @Transactional(readOnly = true) // (3)
    public Cart findCart(String cartId) { // (1) (2)
        Cart cart = cartRepository.findByCartId(cartId);
```

```
        // ...
        return cart;
    }

}
```

項番	説明
(1)	**Service** クラスのメソッドは、業務ロジック毎に作成する。
(2)	業務ロジックは、**Service** インタフェースでメソッドの定義を行い、**Service** クラスのメソッドで実装を行う。
(3)	業務ロジックのトランザクション定義をデフォルト（クラスアノテーションで指定した定義）から変更する場合は、**@Transactional** アノテーションを付加する。 属性値については、要件に応じた値を指定すること。 詳細は、「宣言型トランザクション管理」で必要となる情報 を参照されたい。 また、`@Transactional` アノテーションを使用する際の注意点を理解するために、「トランザクション管理を使うための設定について」を合わせて確認するとよい。

ちなみに：参照系の業務ロジックのトランザクション定義について

参照系の業務ロジックを実装する場合は、`@Transactional(readOnly = true)` を指定することで、JDBC ドライバに対して「読み取り専用のトランザクション」のもとで SQL を実行するように指示することができる。

読み取り専用のトランザクションの扱い方は、JDBC ドライバの実装に依存するため、使用する JDBC ドライバの仕様を確認されたい。

注釈：「読み取り専用のトランザクション」を使用する際の注意点

コネクションプールからコネクションを取得する際にヘルスチェックを行う設定にしている場合、「読み取り専用のトランザクション」が有効にならないケースがある。本事象の詳細及び回避方法については、「読み取り専用のトランザクション」が有効にならないケースについて を参照されたい。

注釈: 新しいトランザクションを開始する必要がある場合のトランザクション定義について

呼び出し元のメソッドが参加しているトランザクションには参加せず、新しいトランザクションを開始する必要がある場合は、`@Transactional(propagation = Propagation.REQUIRES_NEW)` を設定する。

Service クラスのメソッド引数と返り値について

Service クラスのメソッド引数と返り値は、以下の点を考慮すること。

Service クラスの引数と返り値は、Serialize 可能なクラス (`java.io.Serializable` を実装しているクラス) とする。
Service クラスは、分散アプリケーションとしてデプロイされる可能性もあるので、引数と返り値は、Serialize 可能なクラスのみ、許可することを推奨する。

メソッド引数/返り値となる代表的な型を以下に示す。

- プリミティブ型 (`int`, `long` など)
- プリミティブラッパークラス (`java.lang.Integer`, `java.lang.Long` など)
- java 標準クラス (`java.lang.String`, `java.util.Date` など)
- ドメインオブジェクト (Entity、DTO など)
- 入出力オブジェクト (DTO)
- 上記型のコレクション (`java.util.Collection` の実装クラス)
- void
- etc …

注釈: 入出力オブジェクトとは

1. 入力オブジェクトとは、Service のメソッドを実行するために必要な入力値をまとめたオブジェクトのことをさす。
2. 出力オブジェクトとは、Service のメソッドの実行結果（出力値）をまとめたオブジェクトのことをさす。

   「TERASOLUNA ViSC」を使用して、業務ロジック (BLogic クラス) を生成する場合、BLogic の引数と返り値には、入出力オブジェクトを使用することになる。

メソッド引数/返り値として禁止するものを以下に示す。

- アプリケーション層の実装アーキテクチャ (Servlet API や Spring の web 層の API など) に依存するオブジェクト (`javax.servlet.http.HttpServletRequest` 、 `javax.servlet.http.HttpServletResponse` 、 `javax.servlet.http.HttpSession` 、 `org.springframework.http.server.ServletServerHttpRequest` など)
- アプリケーション層のモデル (Form,DTO など)
- `java.util.Map` の実装クラス

注釈: 禁止する理由

1. アプリケーション層の実装アーキテクチャに依存するオブジェクトを許可してしまうと、アプリケーション層とドメイン層が密結合になってしまう。
2. `java.util.Map` は、インタフェースとして汎用性が高すぎるため、メソッドの引数や返り値に使うと、どのようなオブジェクトが格納されているかわかりづらい。 また、値の管理がキー名で行われるため、以下の問題が発生しやすくなる。
   - 値を設定する処理と値を取得する処理で異なるキー名を指定してしまい、値が取得できない。
   - キー名の変更した場合の影響範囲の把握が困難になる。

アプリケーション層とドメイン層で同じ **DTO** を共有する場合の方針を、以下に示す。

- ドメイン層のパッケージに属する DTO として作成し、アプリケーション層で利用する。

> **警告:** アプリケーション層の Form や DTO を、ドメイン層で利用してはいけない。

SharedService クラスの実装

SharedService クラスの作成方法

SharedService クラスを作成する際の注意点を、以下に示す。
ここでは Service クラスと異なる箇所にフォーカスを当てて説明する。

1. 必要に応じて、クラスに **@Transactional** アノテーションを付加する。
   データアクセスを伴わない場合は、`@Transactional` アノテーションは不要である。
2. インターフェース名は **XxxSharedService**、クラス名は **XxxSharedServiceImpl** とする。
   上記以外の命名規約でもよいが、Service クラスと SharedService クラスは、区別できる命名規約を設けることを推奨する。

SharedService クラスのメソッドの作成方法

SharedService クラスのメソッドを作成する際の注意点を、以下に示す。
ここでは、Service クラスと異なる箇所にフォーカスを当てて説明する。

1. **SharedService** クラスのメソッドは、複数の業務ロジックで共有されるロジック毎に作成する。
2. 必要に応じて、クラスに **@Transactional** アノテーションを付加する。
   データアクセスを伴わない場合は、アノテーションは不要である。

SharedService クラスのメソッド引数と返り値について

Service クラスのメソッド引数と返り値についてと同様の点を考慮すること。

処理の実装

Service および SharedService のメソッドで実装する処理について説明する。

Service および SharedService では、アプリケーションで使用する業務データの取得、更新、整合性チェックおよびビジネスルールに関わる各種ロジックの実装を行う。

以下に、代表的な処理の実装例について説明する。

業務データを操作する

業務データ (Entity) の取得、更新の実装例については、

- MyBatis3 を使う場合は、データベースアクセス（*MyBatis3* 編）
- JPA を使う場合は、データベースアクセス（*JPA* 編）

を参照されたい。

メッセージを返却する

Service で解決すべきメッセージは、警告メッセージ、業務エラーメッセージの 2 つとなる (下図赤破線部参照)。
それ以外のメッセージは、アプリケーション層で解決される。
メッセージの種類とメッセージのパターンについては、メッセージ管理を参照されたい。

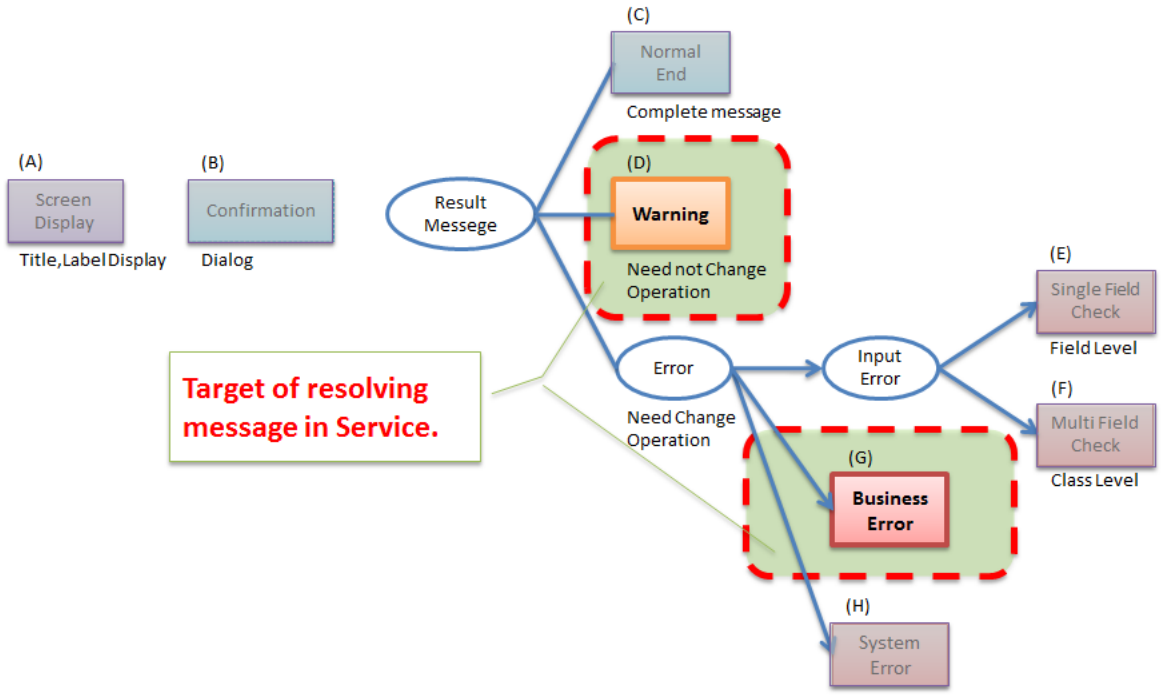
注釈: メッセージの解決について

Service で解決するのは、メッセージ文言ではなく、**メッセージ文言を組み立てるために必要な情報（メッセージコード、メッセージ埋め込み値）の解決**であるという点を補足しておく。

詳細な実装方法は、

- 警告メッセージを返却する
- 業務エラーを通知する

を参照されたい。



警告メッセージを返却する

警告メッセージの返却は、戻り値としてメッセージオブジェクトを返却する。

Entity などのドメイン層のオブジェクトと一緒に返却する必要がある場合は、出力オブジェクト (DTO) にメッセージオブジェクトとドメインオブジェクトを詰めて返却する。

共通ライブラリとしてメッセージオブジェクト (`org.terasoluna.gfw.common.message.ResultMessages`) を用意している。

共通ライブラリで用意しているクラスだと要件を満たせない場合は、プロジェクト毎にメッセージオブジェクトを作成すること。

- DTO の作成

```
public class OrderResult implements Serializable {
    private ResultMessages warnMessages;
    private Order order;

    // omitted

}
```

- Service クラスのメソッドの実装

下記の例では、注文した商品の中に取り寄せ商品が含まれているため、分割配達となる可能性がある旨を警告メッセージとして表示する場合の実装例である。

```
public OrderResult submitOrder(Order order) {

    // omitted

    boolean hasOrderProduct = orderRepository.existsByOrderProduct(order); //(1)

    // omitted

    Order order = orderRepository.save(order);

    // omitted

    ResultMessages warnMessages = null;
    // (2)
    if(hasOrderProduct) {
        warnMessages = ResultMessages.warn().add("w.xx.xx.0001");
    }
    // (3)
    OrderResult orderResult = new OrderResult();
    orderResult.setOrder(order);
    orderResult.setWarnMessages(warnMessages);
    return orderResult;
}
```

項番	説明
(1)	取り寄せ商品が含まれる場合は、`hasOrderProduct` に `true` が設定される。
(2)	上記例では、取り寄せ商品が含まれる場合に、警告メッセージを生成している。
(3)	上記例では、登録した `Order` オブジェクトと警告メッセージを一緒に返却するために、`OrderResult` という DTO にオブジェクトを格納して返却している。

業務エラーを通知する

業務ロジック実行中に、ビジネスルールの違反が発生した場合はビジネス例外をスローする。
例えば次のような場合である。

- 旅行を予約する際に予約日が期限を過ぎている場合

- 商品を注文する際に在庫切れの場合
- etc …

共通ライブラリとしてビジネス例外
(`org.terasoluna.gfw.common.exception.BusinessException`) を用意している。
共通ライブラリで用意しているビジネス例外クラスだと要件を満たせない場合は、プロジェクト毎にビジネス例外クラスを作成すること。
ビジネス例外クラスは、java.lang.RuntimeException のサブクラスとして作成することを推奨する 。

> **注釈: ビジネス例外を非検査例外にする理由**
>
> ビジネス例外は、Controller でハンドリングが必要になるため、本来は検査例外にした方がよい。しかし、本ガイドラインでは、設定漏れによるバグを防ぐ事を目的として、デフォルトでロールバックされる java.lang.RuntimeException のサブクラスとすることを推奨する。もちろん検査例外のサブクラスとしてビジネス例外を作成し、ビジネス例外クラスをロールバック対象として定義する方法を採用してもよい。

ビジネス例外のスロー例を以下に示す。
下記の例では、予約期限日が過ぎていることを業務エラーとして通知する際の実装例である。

```
// omitted

if(currentDate.after(reservationLimitDate)) { // (1)
    throw new BusinessException(ResultMessages.error().add("e.xx.xx.0001"));
}

// omitted
```

項番	説明
(1)	旅行を予約する際に、予約日が期限を過ぎているので、ビジネス例外をスローしている。

例外ハンドリング全体の詳細は、例外ハンドリングを参照されたい。

システムエラーを通知する

業務ロジック実行中に、システムとして異常な状態が発生した場合は、システム例外をスローする。
例えば、次のような場合である。

- 事前に存在しているはずのマスタデータ、ディレクトリ、ファイルなどが存在しない場合
- 利用しているライブラリのメソッドから発生する検査例外のうち、システム異常に分類される例外を補足した場合
- etc …

共通ライブラリとしてシステム例外
(`org.terasoluna.gfw.common.exception.SystemException`) を用意している。
共通ライブラリで用意しているシステム例外クラスだと要件を満たせない場合は、プロジェクト毎にシステム例外クラスを作成すること。

システム例外クラスは、java.lang.RuntimeException のサブクラスとして作成することを推奨する 。

理由は、システム例外は、アプリケーションのコード上でハンドリングする必要がないという点と、
`@Transactinal` アノテーションのデフォルトのロールバック対象が、
`java.lang.RuntimeException` のためである。

システム例外のスロー例を以下に示す。
下記の例では、指定された商品が、商品マスタに存在しないことを、システムエラーとして通知する際の実装例である。

```
ItemMaster itemMaster = itemMasterRepository.findOne(itemCode);
if(itemMaster == null) { // (1)
    throw new SystemException("e.xx.fw.0001",
        "Item master data is not found. item code is " + itemCode + ".");
}
```

項番	説明
(1)	事前に存在しているはずのマスタデータがないので、システム例外をスローしている。（ロジックで、システム異常を検知した場合の実装例）

下記の例では、ファイルコピー時の IO エラーをシステムエラーとして通知する際の実装例である。

```
// ...

try {
    FileUtils.copy(srcFile, destFile);
} catch(IOException e) { // (1)
    throw new SystemException("e.xx.fw.0002",
        "Failed file copy. src file '" + srcFile + "' dest file '" + destFile
↪+ "'.", e);
}
```

項番	説明
(1)	利用しているライブラリのメソッドから、システム異常に分類される例外が発生したシステム例外をスローしている。 **利用しているライブラリから発生した例外は、原因例外としてシステム例外クラスに必ず渡すこと。** 原因例外が失われると、スタックトレースよりエラー発生箇所および本質的なエラー原因が追えなくなってしまう。

注釈: データアクセスエラーの扱いについて

業務ロジック実行中に、Repository や O/R Mapper でデータアクセスエラーが発生した場合、`org.springframework.dao.DataAccessException` のサブクラスに変換されてスローされる。基本的には、業務ロジックではキャッチせず、アプリケーション層でエラーハンドリングすればよいが、一意制約違反などの一部のエラーについては、業務要件によっては、業務ロジックでハンドリングする必要がある。詳細は、データベースアクセス（共通編）を参照されたい。

3.2.6 トランザクション管理について

データの一貫性を保証する必要がある処理ではトランザクションの管理が必要となる。

トランザクション管理の方法

トランザクションの管理方法はいろいろあるが、本ガイドラインでは、**Spring Framework から提供されている「宣言型トランザクション管理」を利用することを推奨する。**

宣言型トランザクション管理

「宣言型トランザクション管理」では、トランザクション管理に必要な情報を以下に 2 つの方法で宣言することができる。

- XML(bean 定義ファイル) で宣言する。
- **アノテーション（@Transactional）で宣言する。（推奨）**

Spring Framework から提供されている「宣言型トランザクション管理」の詳細については、Spring Reference Document -Transaction Management(Declarative transaction management)-を参照されたい。

注釈: 「アノテーションで指定する」方法を推奨する理由

1. ソースコードを見ただけで、どのようなトランザクション管理が行われるかについて、把握することができる。
2. XML にトランザクション管理するための AOP の設定が不要であり、XML がシンプルになる。

「宣言型トランザクション管理」で必要となる情報

トランザクション管理対象とするクラスまたはクラスメソッドに対して`@Transactional` アノテーションを指定する。
トランザクション制御に必要となる情報は、`@Transactional` アノテーションの属性で指定する。

注釈: 本ガイドラインでは、Spring Framework から提供されている `@org.springframework.transaction.annotation.Transactional` アノテーションを使用する前提である。

ちなみに: Spring 4 からは、JTA 1.2 から追加された `@javax.transaction.Transactional` アノテーションを使用する事ができる。

ただし、本ガイドラインでは、「宣言型トランザクション管理」で必要となる情報をより細かく指定できる Spring Framework のアノテーションを使用することを推奨する。

Spring Framework のアノテーションを使用すると、

- トランザクションの伝播方法 (`propagation` 属性) の属性値として `NESTED`(JDBC のセーブポイント)
- トランザクションの独立レベル (`isolation` 属性)
- トランザクションのタイムアウト時間 (`timeout` 属性)
- トランザクションの読み取り専用フラグ (`readOnly` 属性)

の指定が可能となる。

項番	属性名	説明
1	propagation	トランザクションの伝播方法を指定する。 **[REQUIRED]** トランザクションが開始されていなければ開始する。 (省略時のデフォルト) **[REQUIRES_NEW]** 常に、新しいトランザクションを開始する。 **[SUPPORTS]** トランザクションが開始されていれば、それを利用する。開始されていなければ、利用しない。 **[NOT_SUPPORTED]** トランザクションを利用しない。 **[MANDATORY]** トランザクションが開始されている必要がある。開始されていなければ、例外が発生する。 **[NEVER]** トランザクションを利用しない（開始されていてはいけない）。開始していれば、例外が発生する。 **[NESTED]** セーブポイントが設定される。JDBC のみ有効である。

Continued on next page

表 3.11 – continued from previous page

項番	属性名	説明
2	isolation	トランザクションの独立レベルを指定する。 この設定は、DB の仕様に依存するため、使用する DB の仕様を確認し、設定値を決めること。 **[DEFAULT]** DB が提供するデフォルトの独立性レベル。(省略時のデフォルト) **[READ_UNCOMMITTED]** 他のトランザクションで変更中（未コミット）のデータが読める。 **[READ_COMMITTED]** 他のトランザクションで変更中（未コミット）のデータは読めない。 **[REPEATABLE_READ]** 他のトランザクションが読み出したデータは更新できない。 **[SERIALIZABLE]** トランザクションを完全に独立させる。 トランザクションの独立レベルは、排他制御に関連するパラメータとなる。 排他制御については、排他制御を参照されたい。
3	timeout	トランザクションのタイムアウト時間 (秒) を指定する。 デフォルトは-1(使用する DB の仕様や設定に依存)
4	readOnly	トランザクションの読み取り専用フラグを指定する。 デフォルトは false(読み取り専用でない)
5	rollbackFor	トランザクションのロールバック対象とする例外クラスのリストを指定する。 デフォルトは空（指定なし）
6	rollbackForClassName	トランザクションのロールバック対象とする例外クラス名のリストを指定する。 デフォルトは空（指定なし）

Continued on next page

表 3.11 – continued from previous page

項番	属性名	説明
7	noRollbackFor	トランザクションのコミット対象とする例外クラスのリストを指定する。 デフォルトは空（指定なし）
8	noRollbackForClassName	トランザクションのコミット対象とする例外クラス名のリストを指定する。 デフォルトは空（指定なし）

注釈: @Transactional アノテーションを指定する場所

クラスまたはクラスのメソッドに指定することを推奨する。インタフェースまたはインタフェースのメソッドでない点が、ポイント。理由は、Spring Reference Document -Transaction Management(Using @Transactional)-の 2 個目の Tips を参照されたい。

警告: 例外発生時の rollback と commit のデフォルト動作

rollbackFor および noRollbackFor を指定しない場合、Spring Framework は、以下の動作となる。

- 非検査例外クラス（java.lang.RuntimeException および java.lang.Error）またはそのサブクラスの例外が発生した場合は、rollback する。
- 検査例外クラス（java.lang.Exception）またはそのサブクラスの例外が発生した場合は、commit する。**(注意が必要)**

注釈: @Transactional アノテーションの value 属性について

`@Transactional` アノテーションには value 属性があるが、これは複数の Transaction Manager を宣言した際に、どの Transaction Manager を使うのかを指定する属性である。Transaction Manager が一つの場合、指定は不要である。複数の Transaction Manager を使う必要がある場合は、Spring Reference Document -Transaction Management(Multiple Transaction Managers with @Transactional)-を参照されたい。

注釈: 主要 DB の isolation のデフォルトについて

主要 DB のデフォルトの独立性レベルは、以下の通りである。

- Oracle : READ_COMMITTED
- DB2 : READ_COMMITTED
- PostgreSQL : READ_COMMITTED
- SQL Server : READ_COMMITTED
- MySQL : REPEATABLE_READ

注釈: @Transactional アノテーションの timeout 属性について

クエリ発行時（Repository のメソッド実行時）に `timeout` 属性に指定した時間に従って、トランザクションタイムアウトのチェックが行なわれるが、このときの挙動について以下の点に注意されたい。

- タイムアウトチェック時に既にタイムアウトしていないかを確認するため、timeout 属性に指定した時間が経過したタイミングで例外が発生するわけではない。
- タイムアウトチェック後に、関係ない業務処理にいくら時間がかかってもタイムアウトにはならない。

また、トランザクションタイムアウトに関して以下の事象にも注意されたい。

- クエリを発行した後のタイムアウトの挙動は JDBC ドライバの実装に依存する。
- 使用する Transaction Manager によっては、コミット時にもトランザクションタイムアウトのチェックが行われる。

注釈: 「読み取り専用のトランザクション」が有効にならないケースについて

readOnly = true を指定することで「読み取り専用のトランザクション」のもとで SQL を実行する仕組みが提供されているが、以下の条件にすべて一致する場合、「読み取り専用のトランザクション」が有効にならない JDBC ドライバが存在する。

[本事象の発生条件]

- コネクションプールからコネクションを取得する際に、ヘルスチェックを行う。
- コネクションプールから取得したコネクションの自動コミットを無効にする。
- PlatformTransactionManager として、DataSourceTransactionManager 又は JpaTransactionManager を使用する。(JtaTransactionManager を使用する場合、本事象は発生しない)

[本事象の発生が確認されている JDBC ドライバ]

- org.postgresql:postgresql:9.3-1102-jdbc41 (PostgreSQL 9.3 向け JDBC4.1 互換の JDBC ドライバ)

[本事象の回避方法]

「読み取り専用のトランザクション」が有効にならないケースに一致する場合は、readOnly = true を指定すると無駄な処理が行われる事になるため、参照系の処理についても「更新可能なトランザクション」のもとで実行することを推奨する。

他の回避方法として、

- コネクションプールからコネクションを取得する際に、ヘルスチェックを行わない。
- コネクションプールから取得したコネクションの自動コミットを有効にする。(トランザクション管理が必要な時のみ自動コミットを無効にする)

という方法もあるが、本事象を回避するために、ヘルスチェックや自動コミットに対する設計を変更する事は避けるべきである。

[備考]

- 本事象の再現確認は、PostgreSQL 9.3 及び Oracle 12c で行っており、他のデータベース及びバージョンでは行っていない。
- PostgreSQL 9.3 では、`java.sql.Connection#setReadOnly(boolean)` メソッドを呼び出した際に `SQLException` が発生する。
- *log4jdbc* を使用して SQL や JDBC の API の呼び出しをロギングしている場合、JDBC ドライバから発生した `SQLException` は ERROR レベルでログに出力される。
- **JDBC ドライバから発生する SQLException は Spring Framework が行う例外処理によって無視されるため、アプリケーションの動作としてはエラーにはならないが、「読み取り専用のトランザクション」は有効にならない。**
- Oracle 12c では、本事象の発生は確認されていない。

[参考]

log4jdbc を使用して以下のようなログが出力された場合は、本事象に該当するケースとなる。

```
date:2015-02-20 16:11:56        thread:main     user:    X-Track:
↪level:ERROR     logger:jdbc.audit
↪message:3. Connection.setReadOnly(true)
org.postgresql.util.PSQLException: Cannot change transaction read-only␣
↪property in the middle of a transaction.
    at org.postgresql.jdbc2.AbstractJdbc2Connection.
↪setReadOnly(AbstractJdbc2Connection.java:741) ~[postgresql-9.3-1102-
↪jdbc41.jar:na]
    ...
```

トランザクションの伝播

トランザクションの伝播方法は、ほとんどの場合は「REQUIRED」でよい。

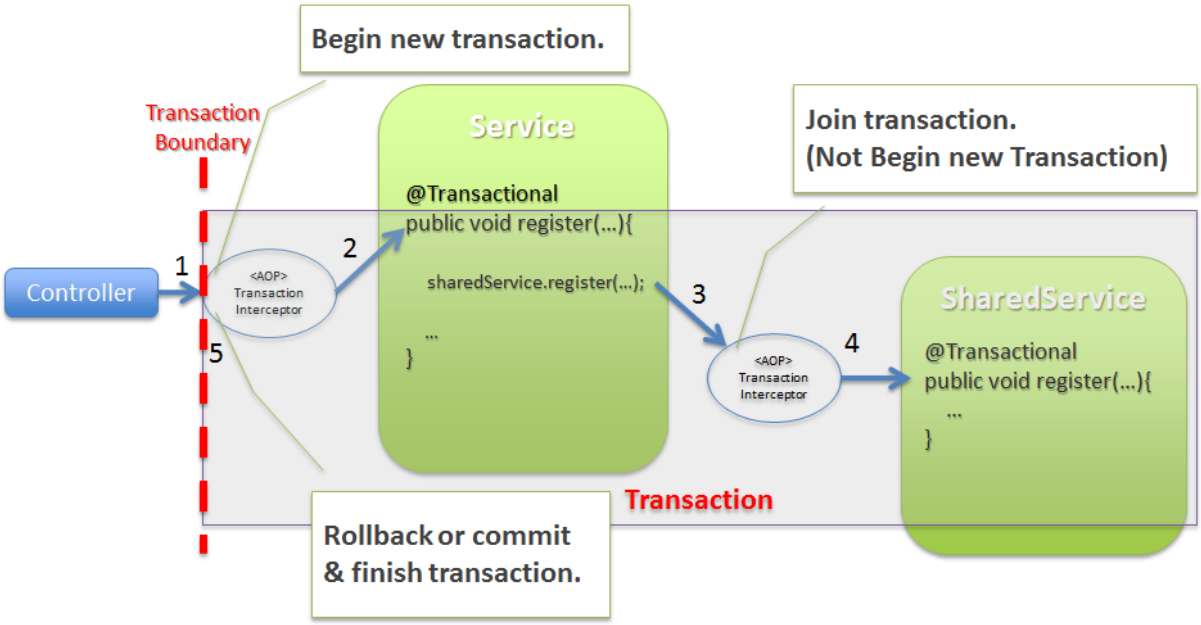
ただし、 **アプリケーションの要件によっては「REQUIRES_NEW」を使うこともある** ので、「REQUIRED」と「REQUIRES_NEW」を指定した場合のトランザクション制御フローを、以下に示す。

他の伝播方法の使用頻度は低いと思われるので、本ガイドラインでの説明は省略する。

トランザクションの伝播方法を「REQUIRED」にした場合のトランザクション管理フロー

トランザクションの伝播方法を「REQUIRED」にした場合、Controller から呼び出された一連の処理が、すべて同じトランザクション内で処理される。

1. Controller からトランザクション管理対象の Service のメソッドを呼び出す。この時点で開始されているトランザクションは存在しないため、`TransactionInterceptor` によってトランザクションが開始される。



2. `TransactionInterceptor` は、トランザクション開始した後に、トランザクション管理対象のメソッドを呼び出す。
3. Service からトランザクション管理対象の `SharedService` のメソッドを呼び出す。この時点で開始済みのトランザクションが存在しているため、`TransactionInterceptor` は、新たにトランザクションは開始せず、開始済みのトランザクションに参加する。
4. `TransactionInterceptor` は、開始済みのトランザクションに参加した後に、トランザクション管理対象のメソッドを呼び出す。
5. `TransactionInterceptor` は、処理結果に応じてコミットまたはロールバックを行い、トランザクションを終了する。

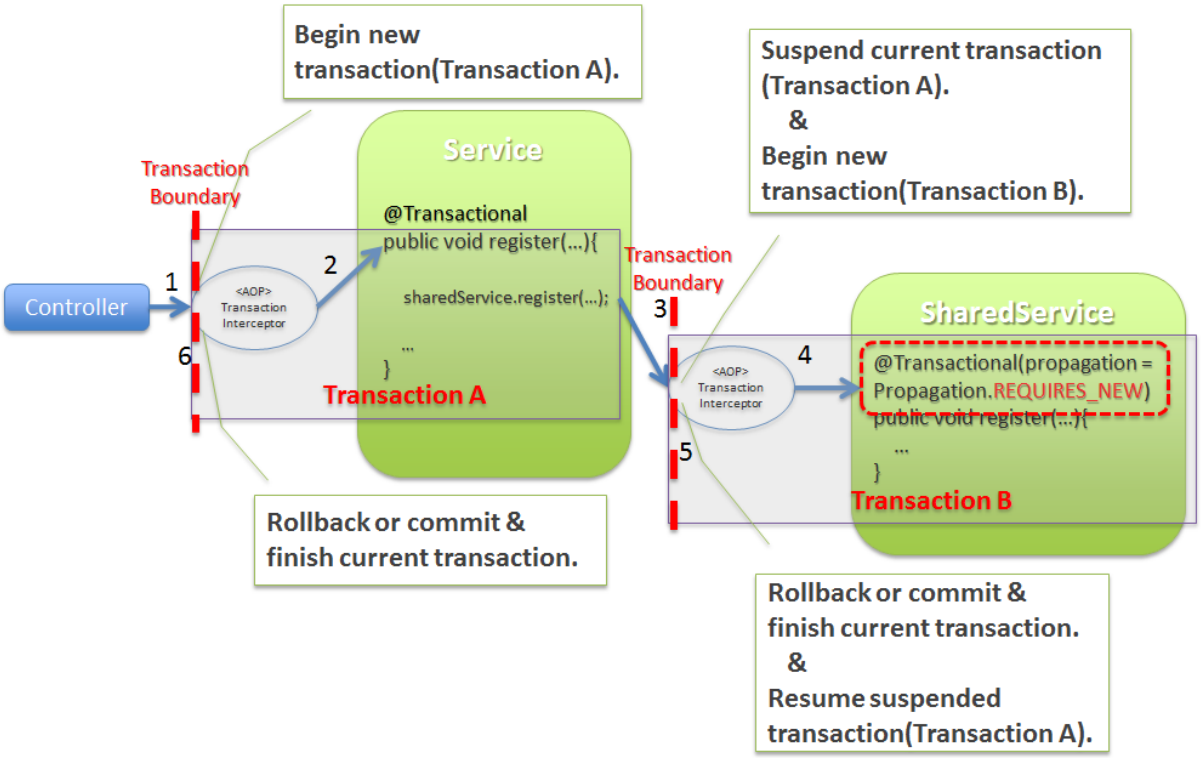
注釈: org.springframework.transaction.UnexpectedRollbackException が発生する理由

トランザクションの伝播方法を「REQUIRED」にした場合、物理的なトランザクションは一つだが、Spring Framework では内部的なトランザクション制御境界が設けられている。上記例だと、SharedService が呼び出された際に実行される `TransactionInterceptor` が、内部的なトランザクション制御を行っている。そのため、`SharedService` でロールバック対象の例外が発生した場合、`TransactionInterceptor` によって、トランザクションはロールバック状態（rollback-only）に設定され、トランザクションをコミットすることはできなくなる。この状態でトランザクションのコミットを行おうとすると、Spring Framework は、`UnexpectedRollbackException` を発生させ、トランザクション制御に矛盾が発生している事を通知してくれる。`UnexpectedRollbackException` が発生した場合、rollbackFor および noRollbackFor の定義に、矛盾がないか、確認すること。

トランザクションの伝播方法を「REQUIRES_NEW」にした場合のトランザクション管理フロー

トランザクションの伝播方法を「REQUIRES_NEW」にした場合、Controller から呼び出された時に行われる

一連の処理の一部（SharedService で行っている処理）が別のトランザクションで処理される。



1. Controller からトランザクション管理対象の Service のメソッドを呼び出す。この時点で開始されているトランザクションは存在しないため、`TransactionInterceptor` によってトランザクションが開始される (ここで開始したトランザクションを以降「Transaction A」と呼ぶ)。
2. `TransactionInterceptor` は、トランザクション（Transaction A）を開始した後に、トランザクション管理対象のメソッドを呼び出す。
3. Service からトランザクション管理対象の `SharedService` のメソッドを呼び出す。この時点で開始済みのトランザクション（Transaction A）が存在しているが、トランザクションの伝播方法が「REQUIRES_NEW」なので `TransactionInterceptor` によって新しいトランザクションが開始される (ここで開始したトランザクションを以降「Transaction B」と呼ぶ)。この時点で「Transaction A」のトランザクションは、中断され再開待ちの状態となる。
4. `TransactionInterceptor` は、トランザクション（Transaction B）を開始した後に、トランザクション管理対象のメソッドを呼び出す。
5. `TransactionInterceptor` は、処理結果に応じてコミットまたはロールバックを行い、トランザクション（Transaction B）を終了する。この時点で、「Transaction A」のトランザクションが再開され、アクティブな状態になる。
6. `TransactionInterceptor` は、処理結果に応じてコミットまたはロールバックを行い、トランザクション（Transaction A）を終了する。

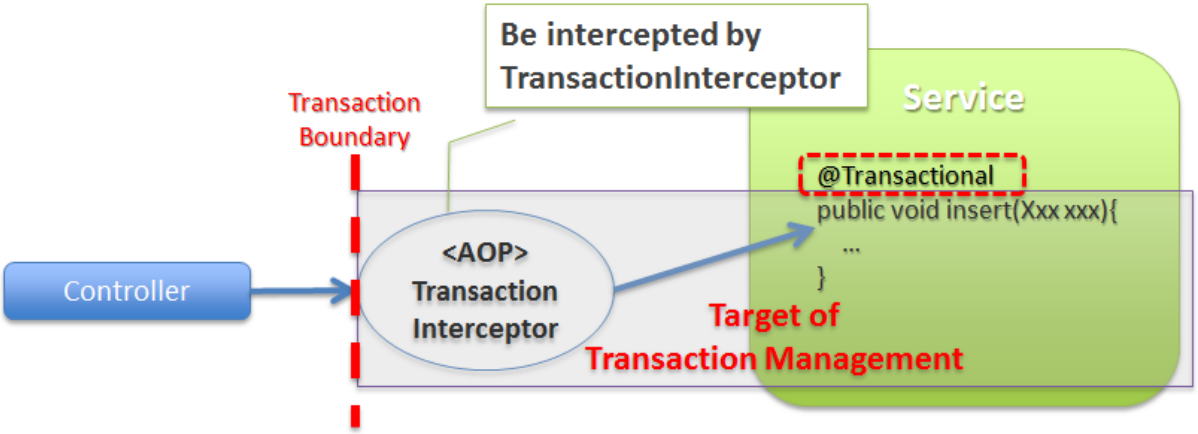
トランザクション管理対象となるメソッドの呼び出し方

Spring Framework から提供されている「宣言型トランザクション管理」は AOP で実現されているため、AOP が有効となるメソッド呼び出しに対してのみ、トランザクション管理が適用される。

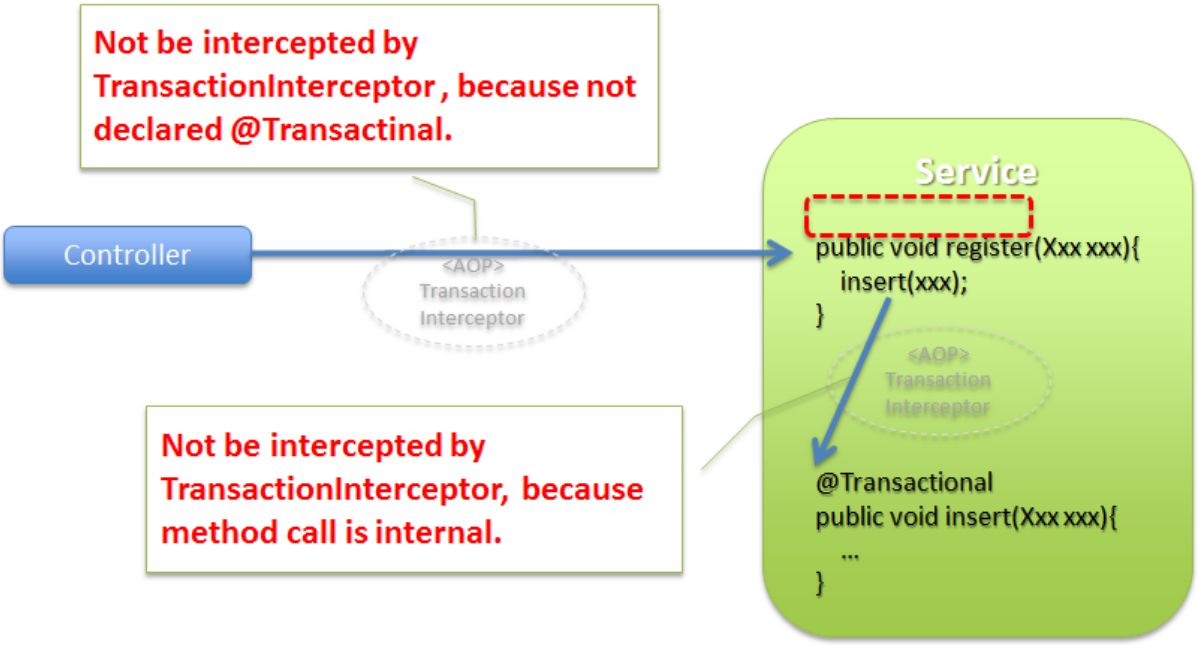
デフォルトの AOP モードが、**proxy** モードなので、別のクラスから **public** メソッドが呼び出された場合のみトランザクション管理対象となる。

public メソッドであっても、内部呼び出しの場合は、トランザクション管理対象にならないので注意が必要となる。

- トランザクション管理対象となるメソッドの呼び出し方



- トランザクション管理対象にならないメソッドの呼び出し方



注釈: 内部呼び出しをトランザクション管理対象にしたい場合

AOP モードを aspectj にすることで、内部呼び出しをトランザクション管理対象にすることができる。ただし、内部呼び出しもトランザクション管理対象にしてしまうと、トランザクション管理の経路が複雑になる可能性があるので、基本的には AOP モードはデフォルトの proxy を使用することを推奨する。

トランザクション管理を使うための設定について

トランザクション管理を使うために必要な設定について説明する。

PlatformTransactionManager の設定

トランザクション管理を行う場合、PlatformTransactionManager の bean を設定する必要がある。

Spring Framework より用途毎のクラスが提供されているので、使用するクラスを指定すればよい。

- xxx-env.xml

  以下に、DataSource から取得される JDBC コネクションの機能を使って、トランザクションを管理する場合の設定例を示す。

```
<!-- (1) -->
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager
↪">
    <property name="dataSource" ref="dataSource" />
    <property name="rollbackOnCommitFailure" value="true" />
</bean>
```

項番	説明
(1)	用途にあった PlatformTransactionManager の実装クラスを指定する。 id は「transactionManager」としておくことを推奨する。

注釈: 複数 DB（複数リソース）に対するトランザクション管理（グローバルトランザクションの管理）が必要な場合

- org.springframework.transaction.jta.JtaTransactionManager を利用し、アプリケーションサーバから提供されている JTA の機能を使って、トランザクション管理を行う必要がある。
- WebSphere、Oracle WebLogic Server で JTA を使う場合、<tx:jta-transaction-manager/> を指定することで、アプリケーションサーバ用に拡張された JtaTransactionManager が、自動的で設定される。

表 3.12 Spring Framework から提供されている PlatformTransactionManager の実装クラス

項番	クラス名	説明
1.	org.springframework.jdbc.datasource.DataSourceTransactionManager	JDBC(`java.sql.Connection`) の API を呼び出して、トランザクションを管理するための実装クラス。 MyBatis や、`JdbcTemplate` を使う場合は、本クラスを使用する。
2.	org.springframework.orm.jpa.JpaTransactionManager	JPA(`javax.persistence.EntityTransaction`) の API を呼び出して、トランザクションを管理するための実装クラス。 JPA を使う場合は、本クラスを使用する。
3.	org.springframework.transaction.jta.JtaTransactionManager	JTA(`javax.transaction.UserTransaction`) の API を呼び出してトランザクションを管理するための実装クラス。 アプリケーションサーバから提供されている JTS(Java Transaction Service) を利用して、リソース (データベース/メッセージングサービス/汎用 EIS(Enterprise Information System) など) とのトランザクションを管理する場合は、本クラスを使用する。 複数のリソースに対する操作を同一トランザクションで行う必要がある場合は、JTA を利用して、リソースとのトランザクションを管理する必要がある。

@Transactional を有効化するための設定

本ガイドラインでは、`@Transactional` アノテーションを使った「宣言型トランザクション管理」を使って、トランザクション管理することを推奨している。

ここでは、`@Transactional` アノテーションを使うために、必要な設定について説明する。

- `xxx-domain.xml`

```
<tx:annotation-driven /> <!-- (1) -->
```

項番	説明
(1)	<tx:annotation-driven>要素を XML（bean 定義ファイル）に追加することで、`@Transactional` アノテーションを使ったトランザクション境界の指定が有効となる。

注釈: トランザクション管理の落とし穴について

IBM DeveloperWorks に「トランザクションの落とし穴を理解する」という記事がある。この記事ではトランザクション管理で注意しなくてはいけないことや、Spring Framework の@Transactional を使う場合の注意点がまとめられているので、ぜひ一読してほしい。詳細は、IBM DeveloperWorks の記事を参照されたい。

※ IBM DeveloperWorks の記事は 2009 年の記事のため (古いため)、一部の内容が Spring Framework 4.1 使用時の動作と異なる部分がある。

具体的には、「Listing 7. Using read-only with REQUIRED propagation mode ― JPA」の内容である。

Spring Framework 4.1 より、JPA のプロバイダとして Hibernate ORM 4.2 以上を使用している場合は、JDBC ドライバに対して「読み取り専用のトランザクション」のもとで SQL を実行するように指示することが出来るように改善 (SPR-8959) されている。

読み取り専用のトランザクションの扱い方は、JDBC ドライバの実装に依存するため、使用する JDBC ドライバの仕様を確認されたい。

注釈: プログラマティックにトランザクションを管理する方法

本ガイドラインでは、「宣言型トランザクション管理」を推奨しているが、プログラマティックにトランザクションを管理することもできる。詳細については、Spring Reference Document -Transaction Management(Programmatic transaction management)-を参照されたい。

<tx:annotation-driven>要素の属性について

<tx:annotation-driven>にはいくつかの属性が指定でき、デフォルトの振る舞いを拡張することができる。

- `xxx-domain.xml`

```
<tx:annotation-driven
    transaction-manager="txManager"
    mode="aspectj"
    proxy-target-class="true"
    order="0" />
```

項番	属性	説明
1	transaction-manager	`PlatformTransactionManager` の bean を指定する。省略した場合「transactionManager」という bean 名で登録されている bean が使用される。
2	mode	AOP のモードを指定する。省略した場合、`proxy` となる。`aspectj` を指定できるが、原則デフォルトの `proxy` を使う。
3	proxy-target-class	proxy のターゲットをクラスに限定するかを指定するフラグ（mode=」proxy」の場合のみ、有効な設定)。省略した場合「false」となる。 • false の場合、対象がインタフェースを実装している場合は、JDK 標準の Dynamic proxies 機能によって proxy され、インタフェースを実装していない場合は Spring Framework に内包されている GCLIB の機能によって proxy される。 • true の場合、インタフェースの実装有無に関係なく、GCLIB の機能によって proxy される。
4	order	AOP で Advice される順番（優先度）を指定する。省略した場合「最後（もっとも低い優先度)」となる。

3.2.7 Tips

ビジネスルールの違反をフィールドエラーとして扱う方法

ビジネスルールのエラーをフィールド毎に出力する必要がある場合、Controller 側 (Bean Validation または Spring Validator) の仕組みを利用する必要がある。

このケースの場合、チェックロジック自体は Service として実装し、Bean Validation または Spring Validator から Service のメソッドを呼び出す方式で実現することを推奨する。

詳細は、入力チェックの業務ロジックチェックを参照されたい。

3.3 インフラストラクチャ層の実装

インフラストラクチャ層では、*RepositoryImpl の実装*を行う。

RepositoryImpl は、Repository インタフェースで定義したメソッドの実装を行う。

3.3.1 RepositoryImpl の実装

以下に、MyBatis3 と JPA を使って、リレーショナルデータベース用の Repository を作成する方法を紹介する。

- *MyBatis3 を使って Repository を実装*
- *JPA を使って Repository を実装*

MyBatis3 を使って Repository を実装

リレーショナルデータベースとの永続 API として MyBatis3 を使う場合、MyBatis3 から提供されている「*Mapper インタフェースの仕組みについて*」を利用して Repository インタフェースを作成すると、基本的には RepositoryImpl を実装する必要はない。

これは、MyBatis3 が、Mapper インタフェースのメソッドと呼び出すステートメント (SQL) のマッピングを自動で行う仕組みになっているためである。

MyBatis3 を使用する場合、アプリケーション開発者は、

- Repository インタフェース (メソッドの定義)
- マッピングファイル (SQL と O/R マッピングの定義)

の作成を行う。

以下に、Repository インタフェースとマッピングファイルの作成例を示す。
MyBatis3 の使用方法の詳細は、データベースアクセス（*MyBatis3* 編）を参照されたい。

- Repository インタフェース (Mapper インタフェース) の作成例

```
package com.example.domain.repository.todo;

import com.example.domain.model.Todo;

// (1)
public interface TodoRepository {
    // (2)
    Todo findOne(String todoId);
}
```

項番	説明
(1)	POJO のインタフェースとして作成する。 MyBatis3 のインタフェースやアノテーションなどを指定する必要はない。
(2)	Repository のメソッドを定義する。 基本的には、MyBatis3 のアノテーションを付与する必要はないが、一部のケースでアノテーションを指定する事もある。

- マッピングファイルの作成例

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org/DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!-- (3) -->
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <!-- (4) -->
    <select id="findOne" parameterType="string" resultMap="todoResultMap">
      SELECT
          todo_id,
          title,
          finished
      FROM
          t_todo
      WHERE
          todo_id = #{todoId}
    </select>

    <!-- (5) -->
    <resultMap id="todoResultMap" type="Todo">
        <result column="todo_id" property="todoId" />
        <result column="title" property="title" />
        <result column="finished" property="finished" />
    </resultMap>

</mapper>
```

項番	説明
(3)	Repository インタフェース毎にマッピングファイルを作成する。 マッピングファイルのネームスペース (`mapper` 要素の `namespace` 属性) には、Repository インタフェースの FQCN(Fully Qualified Class Name) を指定する。
(4)	Repository インタフェースに定義したメソッド毎に実行するステートメント (SQL) の定義を行う。 ステートメント ID(各ステートメント要素 (`select`/`insert`/`update`/`delete` 要素の `id` 属性) には、Repository インタフェースのメソッド名を指定する。
(5)	クエリを発行する場合は、必要に応じて O/R マッピングの定義を行う。 シンプルな O/R マッピングであれば自動マッピングを利用する事ができるが、複雑な O/R マッピングを行う場合は、個別にマッピングの定義が必要となる。 上記例のマッピング定義は、シンプルな O/R マッピングなので自動マッピングを利用する事もできる。

JPA を使って Repository を実装

リレーショナルデータベースとの永続 API として、JPA を使う場合、Spring Data JPA の `org.springframework.data.jpa.repository.JpaRepository` を使用すると、非常に簡単に Repository を作成することが出来る。

Spring Data JPA の使用方法の詳細は、データベースアクセス（*JPA* 編）を参照されたい。

Spring Data JPA を使った場合、基本的な CRUD 操作は、JpaRepository を継承したインタフェースを作成するだけでよい。つまり、基本的には、RepositoryImpl は不要である。

ただし、動的なクエリ (JPQL) を発行する必要がある場合は、RepositoryImpl が必要となる。

Spring Data JPA 使用時の RepositoryImpl の実装については、データベースアクセス（*JPA* 編）を参照されたい。

- TodoRepository.java

```
public interface TodoRepository extends JpaRepository<Todo, String> { // (1)
    // ...
}
```

項番	説明
(1)	JpaRepository を継承したインタフェースを定義するだけで、Todo エンティティに対する基本的な CRUD 操作を実装なしで実現できる。

JpaRepository から提供されていない操作を追加する場合について説明する。

Spring Data JPA を使った場合、静的なクエリであればインタフェースにメソッドを追加し、そのメソッドが呼び出された時に実行するクエリ（JPQL）をアノテーションで指定すればよい。

- TodoRepository.java

```
public interface TodoRepository extends JpaRepository<Todo, String> {
    @Query("SELECT COUNT(t) FROM Todo t WHERE finished = :finished") // (1)
    long countByFinished(@Param("finished") boolean finished);
    // ...
}
```

項番	説明
(1)	@Query アノテーションで、クエリ（JPQL）を指定する。

3.4 アプリケーション層の実装

本節では、HTML form を使った画面遷移型のアプリケーションにおけるアプリケーション層の実装について説明する。

注釈: Ajax の開発や REST API の開発で必要となる実装についての説明は以下のページを参照されたい。

- *Ajax*

アプリケーション層の実装は、以下の 3 つにわかれる。

1. *Controller の実装*

   Controller は、リクエストの受付、業務処理の呼び出し、モデルの更新、View の決定といった処理を行い、リクエストを受けてからの一連の処理フローを制御する。

   アプリケーション層の実装において、もっとも重要な実装となる。

2. フォームオブジェクトの実装

   フォームオブジェクトは、HTML form とアプリケーションの間での値の受け渡しを行う。

3. *View の実装*

   View(JSP) は、モデル（フォームオブジェクトやドメインオブジェクトなど）からデータを取得し、画面 (HTML) を生成する。

3.4.1 Controller の実装

まず、Controller の実装から説明する。

Controller は、以下 5 つの役割を担う。

1. リクエストを受け取るためのメソッドを提供する。

   `@RequestMapping` アノテーションが付与されたメソッドを実装することで、リクエストを受け取ることができる。

2. リクエストパラメータの入力チェックを行う。

   入力チェックが必要なリクエストを受け取るメソッドでは、`@Validated` アノテーションをフォームオブジェクトの引数に指定することで、リクエストパラメータの入力チェックを行うことができる。

   単項目チェックは Bean Validation、相関チェックは Spring Validator 又は Bean Validation でチェックを行う。

3. 業務処理の呼び出しを行う。

   Controller では業務処理の実装は行わず、Service のメソッドに処理を委譲する。

4. 業務処理の処理結果を **Model** に反映する。

   Service のメソッドから返却されたドメインオブジェクトを `Model` に反映することで、View から処理結果を参照できるようにする。

5. 処理結果に対応する **View** 名を返却する。

   Controller では処理結果に対する描画処理を実装せず、描画処理は JSP 等の View で実装する。

   Controller では描画処理が実装されている View の View 名の返却のみ行う。

   View 名に対応する View の解決は、Spring Framework より提供されている `ViewResolver` によって行われ、処理結果に対応する View(JSP など) が呼び出される仕組みになっている。

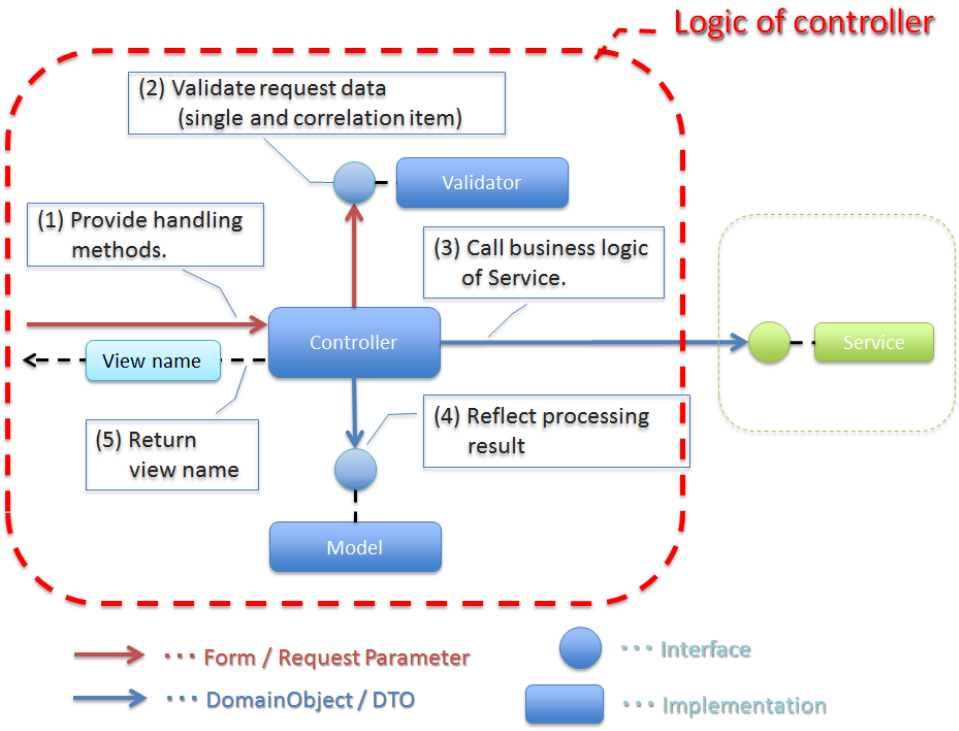


図 3.1 **Picture - Logic of controller**

注釈: Controller では、業務処理の呼び出し、処理結果の `Model` への反映、遷移先 (View 名) の決定などのルーティング処理の実装に徹することを推奨する。

Controller の実装について、以下 4 つの点に着目して説明する。

- *Controller* クラスの作成方法
- リクエストとハンドラメソッドのマッピング方法

- ハンドラメソッドの引数について
- ハンドラメソッドの返り値について

Controller クラスの作成方法

Controller は、POJO クラスに @Controller アノテーションを付加したクラス (Annotation-based Controller) として作成する。

Spring MVC の Controller としては、`org.springframework.web.servlet.mvc.Controller` インタフェースを実装する方法 (Interface-based Controller) もあるが、Spring3 以降は Deprecated になっているため、原則使用しない。

```
@Controller
public class SampleController {
    // ...
}
```

リクエストとハンドラメソッドのマッピング方法

リクエストを受け取るメソッドは、`@RequestMapping` アノテーションを付与する。

本ガイドラインでは、`@RequestMapping` が付加されたメソッドのことを「ハンドラメソッド」と呼ぶ。

```
@RequestMapping(value = "hello")
public String hello() {
    // ...
}
```

リクエストとハンドラメソッドをマッピングするためのルールは、`@RequestMapping` アノテーションの属性に指定する。

項番	属性名	説明
1.	value	マッピング対象にするリクエストパスを指定する (複数可)。
2.	method	マッピング対象にする HTTP メソッド (`RequestMethod` 型) を指定する (複数可)。 GET/POST については HTML form 向けのリクエストをマッピングする際にも使用するが、それ以外の HTTP メソッド (PUT/DELETE など) は REST API 向けのリクエストをマッピングする際に使用する。
3.	params	マッピング対象にするリクエストパラメータを指定する (複数可)。 主に HTML form 向けのリクエストをマッピングする際に使用する。このマッピング方法を使用すると、HTML form 上に複数のボタンが存在する場合のマッピングを簡単に実現する事ができる。
4.	headers	マッピング対象とするリクエストヘッダを指定する (複数可)。 主に REST API や Ajax 向けのリクエストをマッピングする際に使用する。
5.	consumes	リクエストの Content-Type ヘッダを使ってマッピングすることが出来る。マッピング対象とするメディアタイプを指定する (複数可)。 主に REST API や Ajax 向けのリクエストをマッピングする際に使用する。
6.	produces	リクエストの Accept ヘッダを使ってマッピングすることが出来る。マッピング対象とするメディアタイプを指定する (複数可)。 主に REST API や Ajax 向けのリクエストをマッピングする際に使用する。

注釈: マッピングの組み合わせについて

複数の属性を組み合わせることで複雑なマッピングを行うことも可能だが、保守性を考慮し、可能な限りシンプルな定義になるようにマッピングの設計を行うこと。2 つの属性の組み合わせ（value 属性と別の属性 1 つ）を目安にすることを推奨する。

注釈: **HTTP メソッドごとの@RequestMapping アノテーション**

Spring Framework 4.3 から、HTTP メソッドごとの `@RequestMapping` 合成アノテーションが追加された。よりシンプルにマッピングを定義することができ、意図しない HTTP メソッドのマッピング防止とソースコードの可読性向上が期待できる。

- `@GetMapping`
- `@PostMapping`

以下の定義は、`@RequestMapping(value = "hello", method = RequestMethod.GET)` と定義しているのと同様である。

```
@GetMapping(value = "hello")
public String hello() {
    // ...
}
```

詳細は、Spring Framework Reference Documentation - Composed @RequestMapping Variants を参照されたい。

以下、マッピングの具体例を 6 つ示す。

- リクエストパスでマッピング
- *HTTP* メソッドでマッピング
- リクエストパラメータでマッピング
- リクエストヘッダでマッピング
- *Content-Type* ヘッダでマッピング
- *Accept* ヘッダでマッピング

以降の説明では、以下の Controller クラスにハンドラメソッドを定義する前提となっている。

```
@Controller // (1)
@RequestMapping("sample") // (2)
public class SampleController {
    // ...
}
```

項番	説明
(1)	`@Controller` アノテーションを付加することで Annotation-based なコントローラークラスとして認識され、component scan の対象となる。
(2)	クラスレベルで`@RequestMapping("sample")` アノテーションを付けることでこのクラス内のハンドラメソッドが sample 配下の URL にマッピングされる。

リクエストパスでマッピング

下記の定義の場合、`sample/hello` という URL にアクセスすると、hello メソッドが実行される。

```
@RequestMapping(value = "hello")
public String hello() {
```

複数指定した場合は、OR 条件で扱われる。

下記の定義の場合、 `sample/hello` 又は `sample/bonjour` という URL にアクセスすると、hello メソッドが実行される。

```
@RequestMapping(value = {"hello", "bonjour"})
public String hello() {
```

指定するリクエストパスは、具体的な値ではなくパターンを指定することも可能である。パターン指定の詳細は、Spring Framework の Reference Document を参照。

- URI Template Patterns
- URI Template Patterns with Regular Expressions
- Path Patterns
- Patterns with Placeholders

HTTP メソッドでマッピング

下記の定義の場合、 sample/hello という URL に POST メソッドでアクセスすると、hello メソッドが実行される。サポートしている HTTP メソッドの一覧は RequestMethod の Javadoc を参照されたい。指定しない場合、サポートしている全ての HTTP メソッドがマッピング対象となる。

```
@RequestMapping(value = "hello", method = RequestMethod.POST)
public String hello() {
```

複数指定した場合は、OR 条件で扱われる。

下記の定義の場合、 sample/hello という URL に GET 又は HEAD メソッドでアクセスすると、hello メソッドが実行される。

```
@RequestMapping(value = "hello", method = {RequestMethod.GET, RequestMethod.
↪HEAD})
public String hello() {
```

リクエストパラメータでマッピング

下記の定義の場合、 sample/hello?form という URL にアクセスすると、hello メソッドが実行される。

POST でリクエストする場合は、リクエストパラメータは URL になくてもリクエスト BODY に存在していればよい。

```
@RequestMapping(value = "hello", params = "form")
public String hello() {
```

複数指定した場合は、AND 条件で扱われる。

下記の定義の場合、 sample/hello?form&formType=foo という URL にアクセスすると、hello メソッドが実行される。

```
@RequestMapping(value = "hello", params = {"form", "formType=foo"})
public String hello(@RequestParam("formType") String formType) {
```

サポートされている指定形式は以下の通り。

項番	形式	説明
1.	paramName	指定した parameName のリクエストパラメータが存在する場合にマッピングされる。
2.	!paramName	指定した parameName のリクエストパラメータが存在しない場合にマッピングされる。
3.	paramName=paramValue	指定した parameName の値が paramValue の場合にマッピングされる。
4.	paramName!=paramValue	指定した parameName の値が paramValue でない場合にマッピングされる。

リクエストヘッダでマッピング

主に REST API や Ajax 向けのリクエストをマッピングする際に使用するため、詳細は以下のページを参照されたい。

- *Ajax*

Content-Type ヘッダでマッピング

主に REST API や Ajax 向けのリクエストをマッピングする際に使用するため、詳細は以下のページを参照されたい。

- *Ajax*

Accept ヘッダでマッピング

主に REST API や Ajax 向けのリクエストをマッピングする際に使用するため、詳細は以下のページを参照されたい。

- *Ajax*

リクエストとハンドラメソッドのマッピング方針

以下の方針でマッピングを行うことを推奨する。

- 業務や機能といった意味のある単位で、リクエストの **URL** をグループ化する。

URL のグループ化とは、 `@RequestMapping(value = "xxx")` をクラスレベルのアノテーションとして定義することを意味する。

- **処理内の画面フローで使用するリクエストの URL は、同じ URL にする。**

  同じ URL とは `@RequestMapping(value = "xxx")` の value 属性の値を同じ値にすることを意味する。

  処理内の画面フローで使用するハンドラメソッドの切り替えは、HTTP メソッドと HTTP パラメータによって行う。

以下にベーシックな画面フローを行うサンプルアプリケーションを例にして、リクエストとハンドラメソッドの具体的なマッピング例を示す。

- サンプルアプリケーションの概要
- リクエスト *URL*
- リクエストとハンドラメソッドのマッピング
- フォーム表示の実装
- 入力内容確認表示の実装
- フォーム再表示の実装
- 新規作成の実装

サンプルアプリケーションの概要

サンプルアプリケーションの機能概要は以下の通り。

- Entity の CRUD 処理を行う機能を提供する。
- 以下の 5 つの処理を提供する。

項番	処理名	処理概要
1.	Entity 一覧取得	作成済みの Entity を全て取得し、一覧画面に表示する。
2.	Entity 新規作成	指定した内容で新たに Entity を作成する。処理内には、画面フロー（フォーム画面、確認画面、完了画面）が存在する。
3.	Entity 参照	指定された ID の Entity を取得し、詳細画面に表示する。
4.	Entity 更新	指定された ID の Entity を更新する。処理内には、画面フロー（フォーム画面、確認画面、完了画面）が存在する。
5.	Entity 削除	指定された ID の Entity を削除する。

- 機能全体の画面フローは以下の通り。

  画面フロー図には記載していないが、入力チェックエラーが発生した場合はフォーム画面を再描画するものとする。

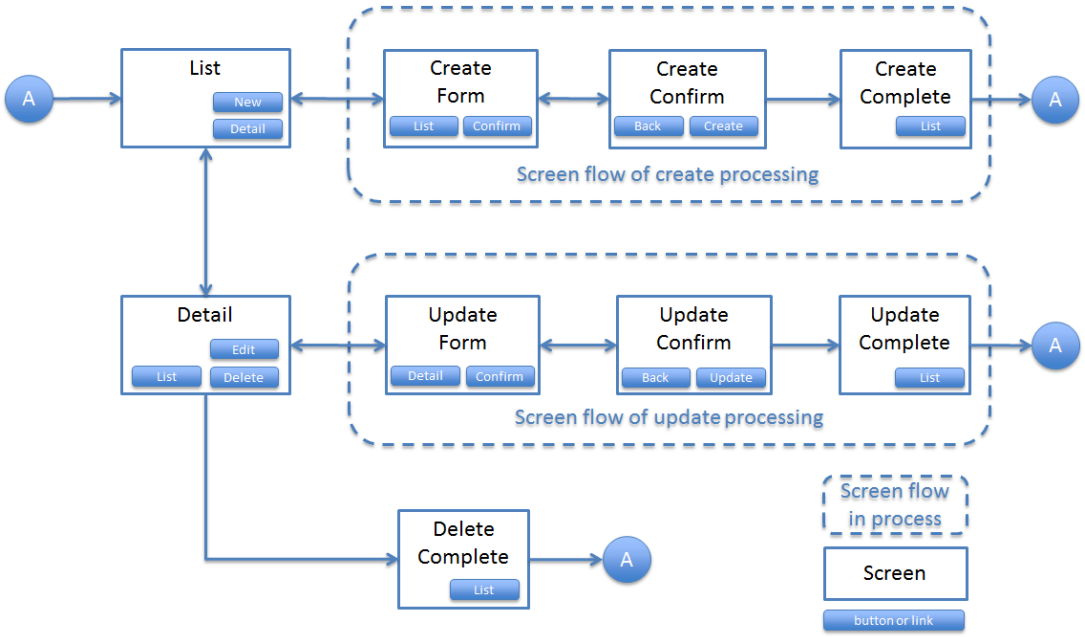


図 3.2　**Picture - Screen flow of entity management function**

リクエスト URL

必要となるリクエストの URL の設計を行う。

- 機能内で必要となるリクエストのリクエスト URL をグループ化する。

  ここでは Abc という Entity の CRUD 操作を行う機能となるので、 `/abc/` から始まる URL とする。

- 処理毎にリクエスト URL を設ける。

項番	処理名	処理毎の URL(パターン)
1.	Entity 一覧取得	/abc/list
2.	Entity 新規作成	/abc/create
3.	Entity 参照	/abc/{id}
4.	Entity 更新	/abc/{id}/update
5.	Entity 削除	/abc/{id}/delete

注釈: Entity 参照、Entity 更新、Entity 削除処理の URL 内に指定している `{id}` は、URI Template Patterns と呼ばれ、任意の値を指定する事ができる。サンプルアプリケーションでは、操作する Entity の ID を指定する。

画面フロー図に各処理に割り振られた URL をマッピングすると以下のようになる。

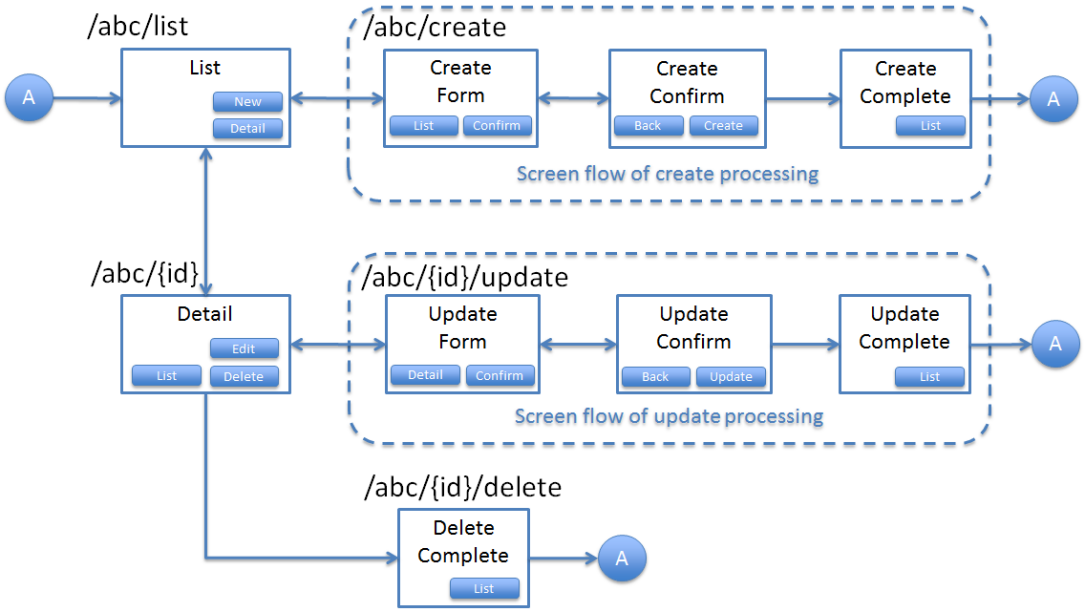


図 3.3 **Picture - Screen flow of entity management function and assigned URL**

リクエストとハンドラメソッドのマッピング

リクエストとハンドラメソッドのマッピングの設計を行う。

以下は、マッピング方針に則って設計したマッピング定義となる。

項番	処理名	URL	リクエスト名	HTTP メソッド	HTTP パラメータ	ハンドラメソッド
1.	Entity 一覧取得	/abc/list	一覧表示	GET	-	list
2.	Entity 新規作成	/abc/create	フォーム表示	-	form	createForm
3.			入力内容確認表示	POST	confirm	createConfirm
4.			フォーム再表示	POST	redo	createRedo
5.			新規作成	POST	-	create
6.			新規作成完了表示	GET	complete	createComplete
7.	Entity 参照	/abc/{id}	詳細表示	GET	-	read
8.	Entity 更新	/abc/{id}/update	フォーム表示	-	form	updateForm
9.			入力内容確認表示	POST	confirm	updateConfirm

Continued on next page

表 3.13 – continued from previous page

項番	処理名	URL	リクエスト名	HTTP メソッド	HTTP パラメータ	ハンドラメソッド
10.			フォーム再表示	POST	redo	updateRedo
11.			更新	POST	-	update
12.			更新完了表示	GET	complete	updateComplete
13.	Entity 削除	/abc/{id}/delete	削除	POST	-	delete
14.			削除完了表示	GET	complete	deleteComplete

Entity 新規作成、Entity 更新、Entity 削除処理では、処理内に複数のリクエストが存在しているため、HTTP メソッドと HTTP パラメータによってハンドラメソッドを切り替えている。

以下に、Entity 新規作成処理を例に、処理内に複数のリクエストが存在する場合のリクエストフローを示す。

URL は全て `/abc/create` で、HTTP メソッドと HTTP パラメータの組み合わせでハンドラメソッドを切り替えている点に注目すること。

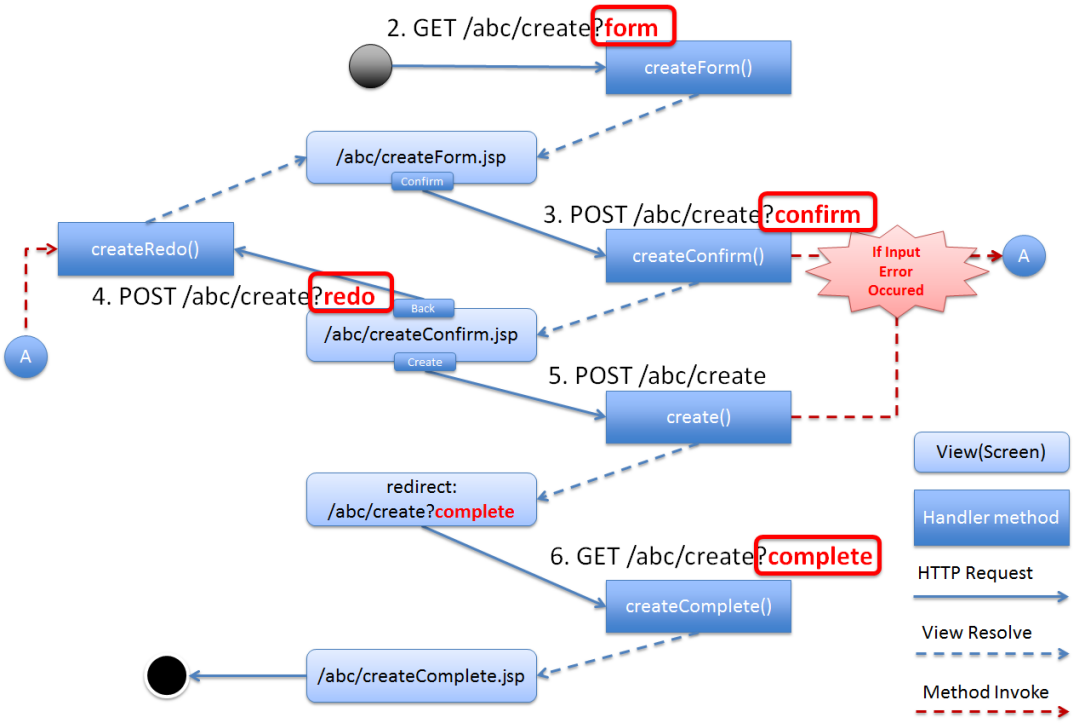


図 3.4 **Picture - Request flow of entity create processing**

以下に、Entity 新規作成処理のハンドラメソッドの実装コードを示す。

ここではリクエストとハンドラメソッドのマッピングについて理解してもらうのが目的なので、`@RequestMapping` の書き方に注目すること。

ハンドラメソッドの引数や返り値 (View 名及び View) の詳細については、次章以降で説明する。

- フォーム表示の実装
- 入力内容確認表示の実装
- フォーム再表示の実装
- 新規作成の実装
- 新規作成完了表示の実装

- *HTML form* 上に複数のボタンを配置する場合の実装

フォーム表示の実装

フォーム表示する場合は、HTTP パラメータとして `form` を指定させる。

```
@RequestMapping(value = "create", params = "form") // (1)
public String createForm(AbcForm form, Model model) {
    // omitted
    return "abc/createForm"; // (2)
}
```

項番	説明
(1)	params 属性に `form` を指定する。
(2)	フォーム画面を描画するための JSP の View 名を返却する。

注釈: この処理で HTTP メソッドを GET に限る必要がないので method 属性を指定していない。

以下に、ハンドラメソッド以外の部分の実装例についても説明しておく。

フォーム表示を行う場合、ハンドラメソッドの実装以外に、

- フォームオブジェクトの生成処理の実装。フォームオブジェクトの詳細は、 フォームオブジェクトの実装 を参照されたい。
- フォーム画面の View の実装。View の詳細は、 *View* の実装 を参照されたい。

が必要になる。

以下のフォームオブジェクトを使用する。

```
public class AbcForm implements Serializable {
    private static final long serialVersionUID = 1L;

    @NotEmpty
    private String input1;

    @NotNull
    @Min(1)
    @Max(10)
    private Integer input2;

    // omitted setter&getter
}
```

フォームオブジェクトを生成する。

```
@ModelAttribute
public AbcForm setUpAbcForm() {
    return new AbcForm();
}
```

フォーム画面の View(JSP) を作成する。

```
<h1>Abc Create Form</h1>
<form:form modelAttribute="abcForm"
    action="${pageContext.request.contextPath}/abc/create">
    <form:label path="input1">Input1</form:label>
    <form:input path="input1" />
    <form:errors path="input1" />
    <br>
    <form:label path="input2">Input2</form:label>
    <form:input path="input2" />
    <form:errors path="input2" />
    <br>
    <input type="submit" name="confirm" value="Confirm" /> <!-- (1) -->
</form:form>
```

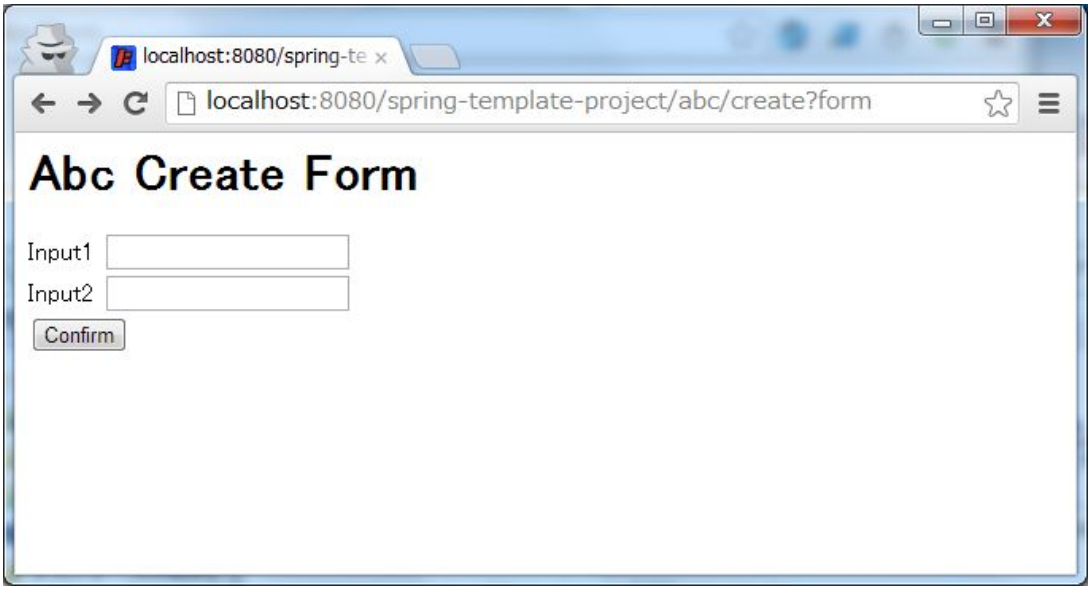
項番	説明
(1)	確認画面へ遷移するための submit ボタンには `name="confirm"` というパラメータを指定しておく。

以下に、フォーム表示の動作について説明する。

フォーム表示処理を呼び出す。

`abc/create?form` という URI にアクセスする。

`form` という HTTP パラメータの指定があるため、Controller の createForm メソッドが呼び出されフォーム画面が表示される。



入力内容確認表示の実装

フォームの入力内容を確認する場合は、POST メソッドでデータを送信し、HTTP パラメータに `confirm` を指定させる。

```
@RequestMapping(value = "create", method = RequestMethod.POST, params =
↪"confirm") // (1)
public String createConfirm(@Validated AbcForm form, BindingResult result,
        Model model) {
    if (result.hasErrors()) {
        return createRedo(form, model); // return "abc/createForm"; (2)
    }
    // omitted
    return "abc/createConfirm"; // (3)
}
```

項番	説明
(1)	method 属性に `RequestMethod.POST` 、params 属性に `confirm` を指定する。
(2)	入力チェックエラーが発生した場合の処理は、フォーム再表示用のハンドラメソッドを呼び出すことを推奨する。フォーム画面を再表示するための処理の共通化を行うことができる。
(3)	入力内容確認画面を描画するための JSP の View 名を返却する。

注釈: POST メソッドを指定させる理由は、個人情報やパスワードなどの秘密情報がブラウザのアドレスバーに現れ、他人に容易に閲覧されることを防ぐためである。(もちろんセキュリティ対策としては十分ではなく、SSL などのセキュアなサイトにする必要がある)。

以下に、ハンドラメソッド以外の部分の実装例についても説明しておく。

入力内容確認表示を行う場合、ハンドラメソッドの実装以外に、

- 入力内容確認画面の View の実装。View の詳細は、 *View の実装* を参照されたい。

が必要になる。

入力内容確認画面の View(JSP) を作成する。

```
<h1>Abc Create Form</h1>
<form:form modelAttribute="abcForm"
    action="${pageContext.request.contextPath}/abc/create">
    <form:label path="input1">Input1</form:label>
        ${f:h(abcForm.input1)}
    <form:hidden path="input1" /> <!-- (1) -->
    <br>
    <form:label path="input2">Input2</form:label>
        ${f:h(abcForm.input2)}
    <form:hidden path="input2" /> <!-- (1) -->
    <br>
    <input type="submit" name="redo" value="Back" /> <!-- (2) -->
    <input type="submit" value="Create" /> <!-- (3) -->
</form:form>
```

項番	説明
(1)	フォーム画面で入力された値は、Create ボタン及び Back ボタンが押下された際に再度サーバに送る必要があるため、HTML form の hidden 項目とする。
(2)	フォーム画面に戻るための submit ボタンには `name="redo"` というパラメータを指定しておく。
(3)	新規作成を行うための submit ボタンにはパラメータ名の指定は不要。

注釈: この例では確認項目を表示する際に HTML エスケープするため、 `f:h()` 関数を使用している。XSS 対策のため、必ず行うこと。詳細については *Cross Site Scripting* を参照されたい。

以下に、入力内容確認の動作について説明する。

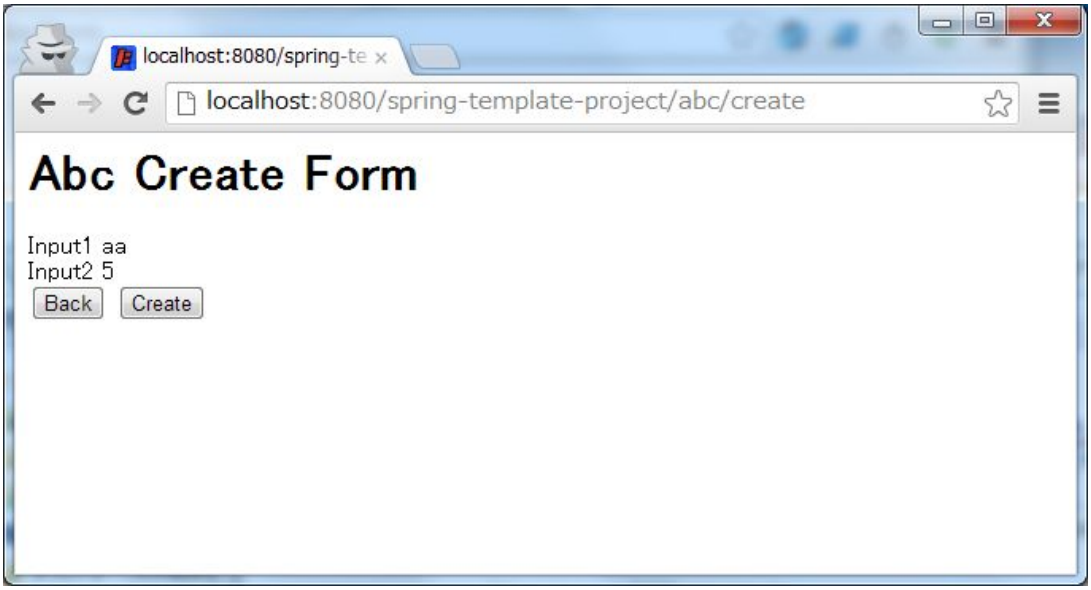
入力内容確認表示処理を呼び出す。

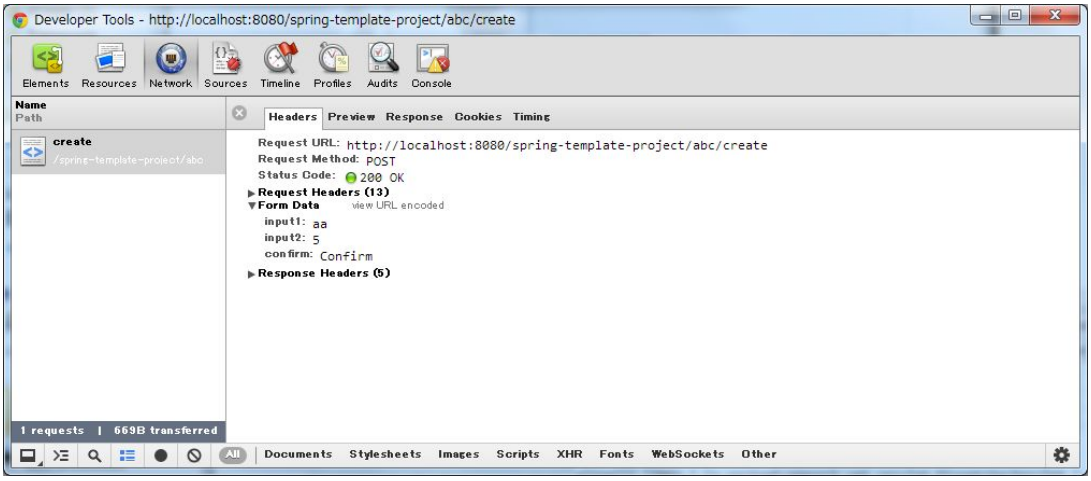
フォーム画面で Input1 に `aa` を、Input2 に 「`5`」 を入力し、Confirm ボタンを押下する。

Confirm ボタンを押下すると、 `abc/create?confirm` という URI に POST メソッドでアクセスする。

`confirm` という HTTP パラメータがあるため、Controller の createConfirm メソッドが呼び出され、入力内容確認画面が表示される。

Confirm ボタンを押下すると POST メソッドで HTTP パラメータが送信されるため、URI には現れていないが、HTTP パラメータとして `confirm` が含まれている。





フォーム再表示の実装

フォームを再表示する場合は、HTTP パラメータに redo を指定させる。

```
@RequestMapping(value = "create", method = RequestMethod.POST, params = "redo
↪") // (1)
public String createRedo(AbcForm form, Model model) {
    // omitted
    return "abc/createForm"; // (2)
}
```

項番	説明
(1)	method 属性に `RequestMethod.POST` 、params 属性に `redo` を指定する。
(2)	入力内容確認画面を描画するための JSP の View 名を返却する。

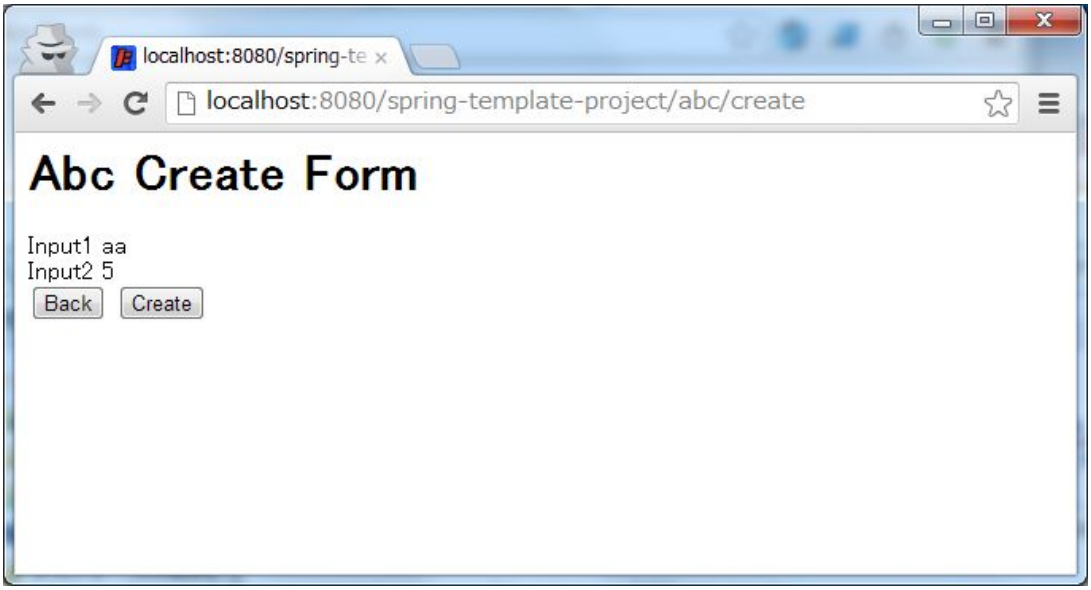
以下に、フォーム再表示の動作について説明する。

フォーム再表示リクエストを呼び出す。

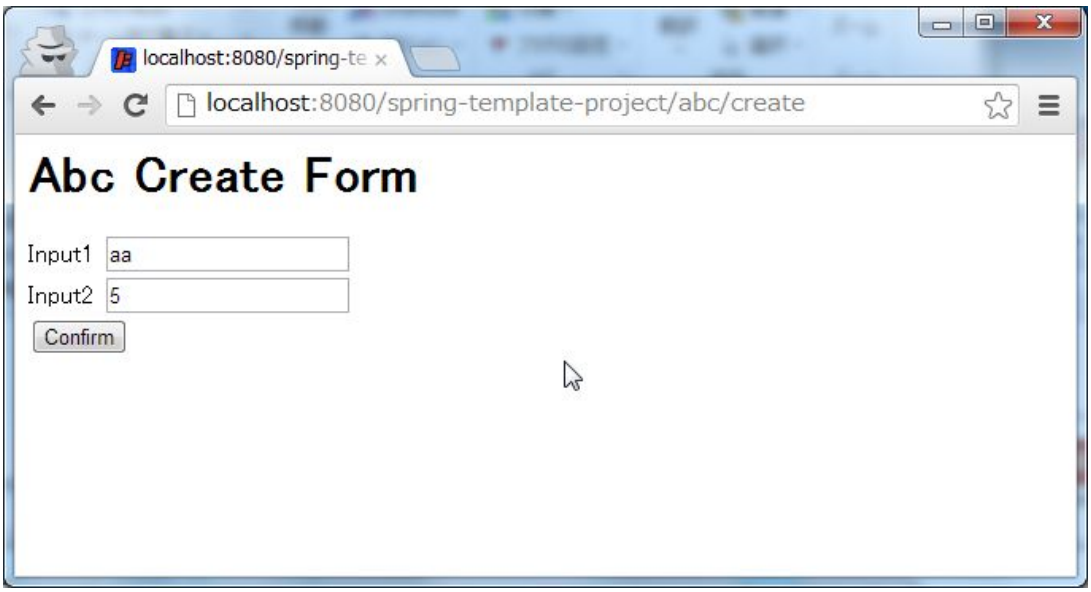
入力内容確認画面で、Back ボタンを押下する。

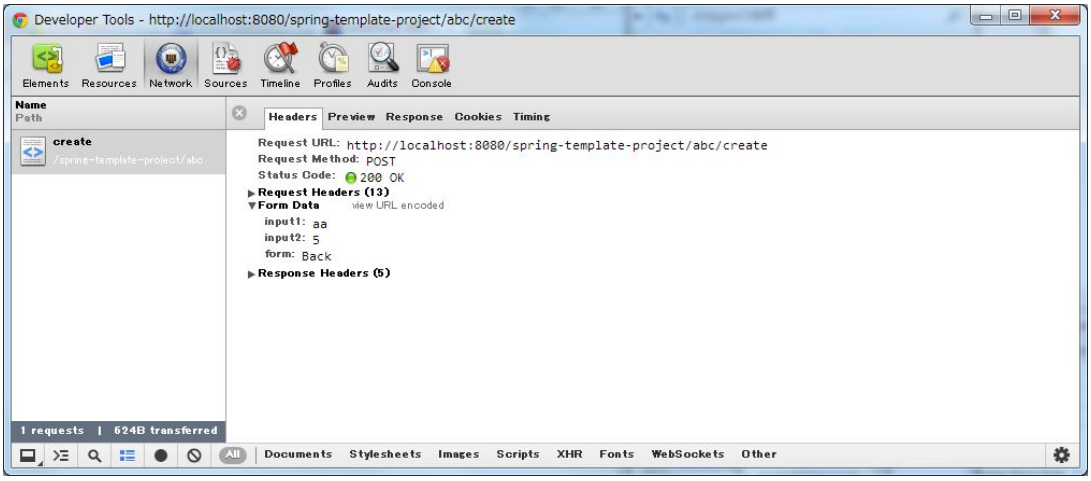
Back ボタンを押下すると、 `abc/create?redo` という URI に POST メソッドでアクセスする。

`redo` という HTTP パラメータがあるため、Controller の createRedo メソッドが呼び出され、フォーム画面が再表示される。



Back ボタンを押下すると POST メソッドで HTTP パラメータが送信されるため、URI には現れていないが、HTTP パラメータとして `redo` が含まれている。また、フォームの入力値を hidden 項目として送信されるため、フォーム画面で入力値を復元することが出来る。





注釈: 戻るボタンの実現方法には、ボタンの属性に `onclick="javascript:history.back()"` を設定する方法もある。両者では以下が異なり、要件に応じて選択する必要がある。

- ブラウザの戻るボタンを押した場合の挙動
- 戻るボタンがあるページに直接アクセスして戻るボタンを押した場合の挙動
- ブラウザの履歴

新規作成の実装

フォームの入力内容を登録する場合は、POST で登録対象のデータ (hidden パラメータ) を送信させる。

新規作成リクエストはこの処理のメインリクエストになるので、HTTP パラメータによる振り分けは行っていない。

この処理ではデータベースの状態を変更するので、二重送信によって新規作成処理が複数回実行されないように制御する必要がある。

そのため、この処理が終了した後は View(画面) を直接表示するのではなく、次の画面 (新規作成完了画面) へリダイレクトしている。このパターンを POST-Redirect-GET(PRG) パターンと呼ぶ。 PRG (Post-Redirect-Get) パターンの詳細については 二重送信防止 を参照されたい。

```
@RequestMapping(value = "create", method = RequestMethod.POST) // (1)
public String create(@Validated AbcForm form, BindingResult result, Model␣
↪model) {
    if (result.hasErrors()) {
        return createRedo(form, model); // return "abc/createForm";
    }
    // omitted
    return "redirect:/abc/create?complete"; // (2)
}
```

項番	説明
(1)	method 属性に `RequestMethod.POST` を指定し、params 属性は指定しない。
(2)	PRG パターンとするため、新規作成完了表示リクエストにリダイレクトするための URL を View 名として返却する。

注釈: 「redirect:/xxx」を返却すると」/xxx」へリダイレクトさせることができる。

警告: PRG パターンとすることで、ブラウザの F5 ボタン押下時のリロードによる二重送信を防ぐ事はできるが、二重送信の対策としては十分ではない。二重送信の対策としては、共通部品として提供している TransactionTokenCheck を行う必要がある。TransactionTokenCheck の詳細については 二重送信防止 を参照されたい。

以下に、「新規作成」の動作について説明する。

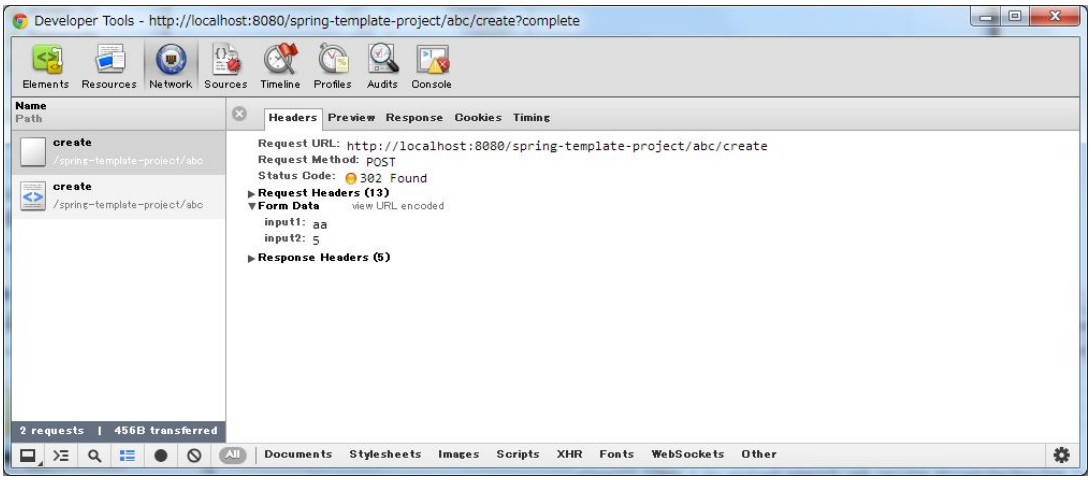
新規作成処理を呼び出す。

入力内容確認画面で、Create ボタンを押下する。

Create ボタンを押下すると、 `abc/create` という URI に POST メソッドでアクセスする。

ボタンを識別するための HTTP パラメータを送信していないので、Entity 新規作成処理のメインのリクエストと判断され、Controller の create メソッドが呼び出される。

新規作成リクエストでは、直接画面を返さず、新規作成完了表示 (`/abc/create?complete`) へリダイレクトしているため、HTTP ステータスが 302 になっている。



新規作成完了表示の実装

新規作成処理が完了した事を通知する場合は、HTTP パラメータに `complete` を指定させる。

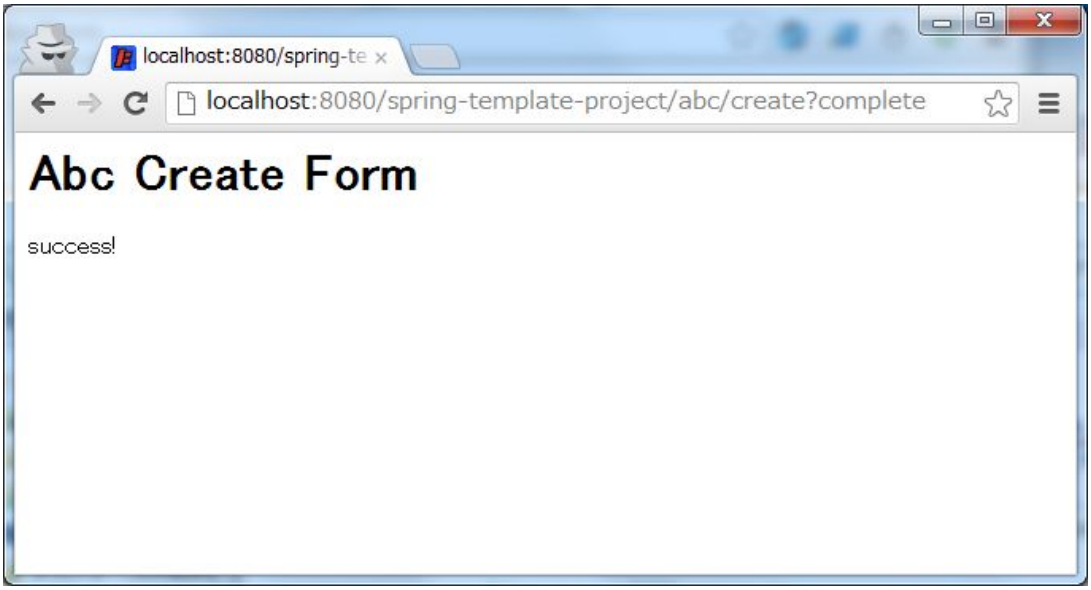
```
@RequestMapping(value = "create", params = "complete") // (1)
public String createComplete() {
    // omitted
    return "abc/createComplete"; // (2)
}
```

項番	説明
(1)	params 属性に `complete` を指定する。
(2)	新規作成完了画面を描画するため、JSP の View 名を返却する。

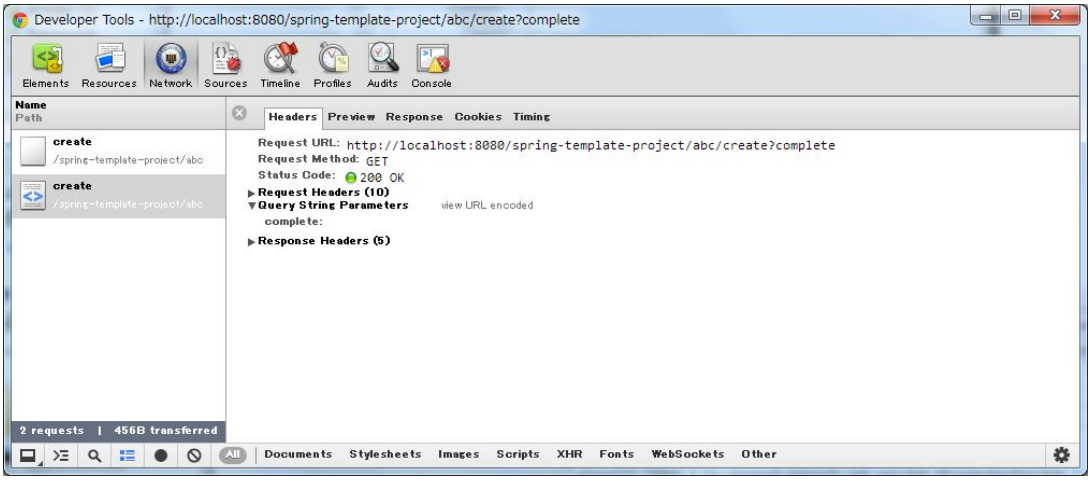
注釈: この処理も HTTP メソッドを GET に限る必要がないので method 属性を指定しなくても良い。

以下に、「新規作成完了表示」の動作について説明する。

新規作成完了後、リダイレクト先に指定された URI(`/abc/create?complete`) にアクセスする。
`complete` という HTTP パラメータがあるため、Controller の createComplete メソッドが呼び出され、新規作成完了画面が表示される。



注釈: PRG パターンを利用しているため、ブラウザをリロードしても、新規作成処理は実行されず、新規作成完了が再度表示されるだけである。



HTML form 上に複数のボタンを配置する場合の実装

1 つのフォームに対して複数のボタンを設置したい場合、ボタンを識別するための HTTP パラメータを送ることで、実行するハンドラメソッドを切り替える。ここではサンプルアプリケーションの入力内容確認画面の Create ボタンと Back ボタンを例に説明する。

下図のように、入力内容確認画面のフォームには、新規作成を行う Create ボタンと新規作成フォーム画面を再表示する Back ボタンが存在する。

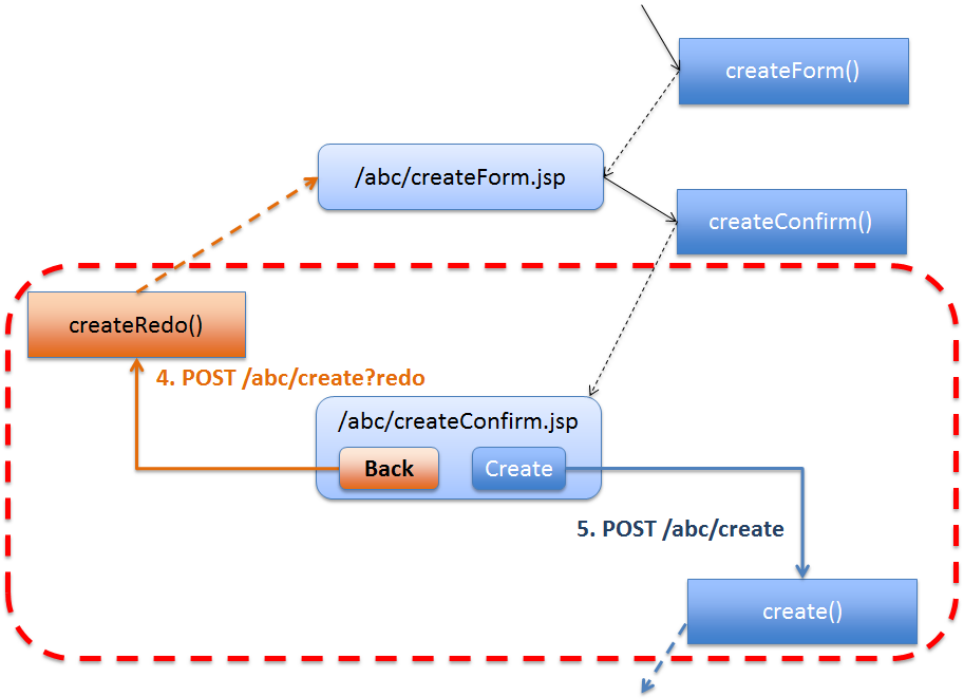


図 3.5　**Picture - Multiple button in the HTML form**

Back ボタンを押下した場合、新規作成フォーム画面を再表示するためのリクエスト (`/abc/create?redo`) を送信する必要があるため、HTML form 内に以下のコードが必要となる。

```
<input type="submit" name="redo" value="Back" /> <!-- (1) -->
<input type="submit" value="Create" />
```

項番	説明
(1)	上記のように、入力内容確認画面 (abc/createConfirm.jsp) の Back ボタンに name="redo"というパラメータを指定する。

Back ボタン押下時の動作については、 フォーム再表示の実装 を参照されたい。

サンプルアプリケーションの Controller のソースコード

以下に、サンプルアプリケーションの新規作成処理実装後の Controller の全ソースを示す。
Entity 一覧取得、Entity 参照、Entity 更新、Entity 削除も同じ要領で実装することになるが、説明は割愛する。

```
@Controller
@RequestMapping("abc")
public class AbcController {

    @ModelAttribute
    public AbcForm setUpAbcForm() {
        return new AbcForm();
    }

    // Handling request of "/abc/create?form"
    @RequestMapping(value = "create", params = "form")
    public String createForm(AbcForm form, Model model) {
        // omitted
        return "abc/createForm";
    }

    // Handling request of "POST /abc/create?confirm"
    @RequestMapping(value = "create", method = RequestMethod.POST, params =
↪"confirm")
    public String createConfirm(@Validated AbcForm form, BindingResult result,
            Model model) {
        if (result.hasErrors()) {
            return createRedo(form, model);
        }
        // omitted
        return "abc/createConfirm";
    }
```

```
    // Handling request of "POST /abc/create?redo"
    @RequestMapping(value = "create", method = RequestMethod.POST, params =
↪"redo")
    public String createRedo(AbcForm form, Model model) {
        // omitted
        return "abc/createForm";
    }

    // Handling request of "POST /abc/create"
    @RequestMapping(value = "create", method = RequestMethod.POST)
    public String create(@Validated AbcForm form, BindingResult result, Model␣
↪model) {
        if (result.hasErrors()) {
            return createRedo(form, model);
        }
        // omitted
        return "redirect:/abc/create?complete";
    }

    // Handling request of "/abc/create?complete"
    @RequestMapping(value = "create", params = "complete")
    public String createComplete() {
        // omitted
        return "abc/createComplete";
    }

}
```

ハンドラメソッドの引数について

ハンドラメソッドの引数は様々な値をとることができる が、基本的には次に挙げるものは原則として使用しないこと。

- ServletRequest
- HttpServletRequest
- org.springframework.web.context.request.WebRequest
- org.springframework.web.context.request.NativeWebRequest
- java.io.InputStream
- java.io.Reader
- java.io.OutputStream

- java.io.Writer
- java.util.Map
- org.springframework.ui.ModelMap

注釈: `HttpServletRequest` の getAttribute/setAttribute や `Map` の get/put のような汎用的なメソッドの利用を許可すると自由な値の受け渡しができてしまい、プロジェクトの規模が大きくなると保守性を著しく低下させる可能性がある。

同様の理由で、他で代替できる場合は `HttpSession` を極力使用しないことを推奨する。

共通的なパラメータ (リクエストパラメータ) を JavaBean に格納して Controller の引数に渡したい場合は後述の *HandlerMethodArgumentResolver の実装* を使用することで実現できる。

以下に、引数の使用方法について、目的別に 13 例示す。

- 画面 *(View)* にデータを渡す
- *URL* のパスから値を取得する
- リクエストパラメータを個別に取得する
- リクエストパラメータをまとめて取得する
- 入力チェックを行う
- リダイレクト先にデータを渡す
- リダイレクト先へリクエストパラメータを渡す
- リダイレクト先 *URL* のパスに値を埋め込む
- *Cookie* から値を取得する
- *Cookie* に値を書き込む
- ページネーション情報を取得する
- アップロードファイルを取得する
- 画面に結果メッセージを表示する

画面 (View) にデータを渡す

画面 (View) に表示するデータを渡したい場合は、`org.springframework.ui.Model`(以降 `Model` と呼ぶ) をハンドラメソッドの引数として受け取り、`Model` オブジェクトに渡したいデータ (オブジェクト) を追加する。

- SampleController.java

```
@RequestMapping("hello")
public String hello(Model model) { // (1)
    model.addAttribute("hello", "Hello World!"); // (2)
    model.addAttribute(new HelloBean("Bean Hello World!")); // (3)
    return "sample/hello"; // returns view name
}
```

- hello.jsp

```
Message : ${f:h(hello)}<br> <%-- (4) --%>
Message : ${f:h(helloBean.message)}<br> <%-- (5) --%>
```

- HTML of created by View(hello.jsp)

```
Message : Hello World!<br> <!-- (6) -->
Message : Bean Hello World!<br>  <!-- (6) -->
```

項番	説明
(1)	Model オブジェクトを引数として受け取る。
(2)	引数で受け取った Model オブジェクトの addAttribute メソッドを呼び出し、渡したいデータを Model オブジェクトに追加する。 例では、hello という属性名で HelloWorld! という文字列のデータを追加している。
(3)	addAttribute メソッドの第一引数を省略すると値のクラス名の先頭を小文字にした文字列が属性名になる。 例では、 model.addAttribute("helloBean", new HelloBean()); を行ったのと同じ結果となる。
(4)	View(JSP) 側では、「${属性名}」と記述することで Model オブジェクトに追加したデータを取得することができる。 例では HTML エスケープを行う EL 式の関数を呼び出しているため、「${f:h(属性名)}」としている。 HTML エスケープを行う EL 式の関数の詳細については、 *Cross Site Scripting* を参照されたい。
(5)	「${属性名.JavaBean のプロパティ名}」と記述することで Model に格納されている JavaBean から値を取得することができる。
(6)	JSP 実行後に出力される HTML。

注釈: Model は使用しない場合でも引数に指定しておいてもよい。実装初期段階では必要なくても後で使う場合がある (後々メソッドのシグニチャを変更する必要がなくなる)。

注釈: Model に addAttribute することで、 HttpServletRequest に setAttribute されるため、Spring MVC の管理下にないモジュール (例えば ServletFilter など) からも値を参照することが

出来る。

URL のパスから値を取得する

URL のパスから値を取得する場合は、引数に`@PathVariable`アノテーションを付与する。

`@PathVariable`アノテーションを使用してパスから値を取得する場合、`@RequestMapping`アノテーションの value 属性に取得したい部分を変数化しておく必要がある。

```
@RequestMapping("hello/{id}/{version}") // (1)
public String hello(
        @PathVariable("id") String id, // (2)
        @PathVariable Integer version, // (3)
        Model model) {
    // do something
    return "sample/hello"; // returns view name
}
```

項番	説明
(1)	`@RequestMapping` アノテーションの value 属性に、抜き出したい箇所をパス変数として指定する。パス変数は、「{変数名}」の形式で指定する。 上記例では、 `id` と `version` という二つのパス変数を指定している。
(2)	`@PathVariable` アノテーションの value 属性には、パス変数の変数名を指定する。 上記例では、 `sample/hello/aaaa/1` という URL にアクセスした場合、引数 id に文字列 `aaaa` が渡る。
(3)	`@PathVariable` アノテーションの value 属性は省略可能で、省略した場合は引数名がリクエストパラメータ名となる。 上記例では、 `sample/hello/aaaa/1` という URL にアクセスした場合、引数 version に数値 「`1`」 が渡る。 ただしこの方法は、 • `-g` オプション (デバッグ情報を出力するモード) • Java8 から追加された`-parameters` オプション (メソッド・パラメータにリフレクション用のメタデータを生成するモード) のどちらかを指定してコンパイルする必要がある。

注釈: バインドする引数の型は String 以外でも良い。型が合わない場合は `org.springframework.beans.TypeMismatchException` がスローされ、デフォルトの動作は 400(Bad Request) が応答される。例えば、上記例で `sample/hello/aaaa/v1` という URL でアクセスした場合、`v1` を Integer に変換できないため、例外がスローされる。

警告: `@PathVariable` アノテーションの value 属性を省略する場合、デプロイするアプリケーションは`-g` オプション又は Java8 から追加された`-parameters` オプションを指定してコンパイルする必要がある。これらのオプションを指定した場合、コンパイル後のクラスにはデバッグ時に必要となる情報や処理などが挿入されるため、メモリや処理性能に影響を与えることがあるので注意が必要である。基本的には、value 属性を明示的に指定する方法を推奨する。

リクエストパラメータを個別に取得する

リクエストパラメータを 1 つずつ取得したい場合は、引数に@RequestParam アノテーションを付与する。

```
@RequestMapping("bindRequestParams")
public String bindRequestParams(
        @RequestParam("id") String id, // (1)
        @RequestParam String name, // (2)
        @RequestParam(value = "age", required = false) Integer age, // (3)
        @RequestParam(value = "genderCode", required = false, defaultValue =
↪"unknown") String genderCode, // (4)
        Model model) {
    // do something
    return "sample/hello"; // returns view name
}
```

項番	説明
(1)	`@RequestParam` アノテーションの value 属性には、リクエストパラメータ名を指定する。 上記例では、 `sample/hello?id=aaaa` という URL にアクセスした場合、引数 id に文字列 `aaaa` が渡る。
(2)	`@RequestParam` アノテーションの value 属性は省略可能で、省略した場合は引数名がリクエストパラメータ名となる。 上記例では、 `sample/hello?name=bbbb&....` という URL にアクセスした場合、引数 name に文字列 `bbbb` が渡る。 ただしこの方法は、 • `-g` オプション (デバッグ情報を出力するモード) • Java8 から追加された`-parameters` オプション (メソッド・パラメータにリフレクション用のメタデータを生成するモード) のどちらかを指定してコンパイルする必要がある。
(3)	デフォルトの動作では、指定したリクエストパラメータが存在しないとエラーとなる。リクエストパラメータが存在しないケースを許容する場合は、required 属性を `false` に指定する。 上記例では、`age` というリクエストパラメータがない状態でアクセスした場合、引数 age に `null` が渡る。
(4)	指定したリクエストパラメータが存在しない場合にデフォルト値を使用したい場合は、defaultValue 属性にデフォルト値を指定する。 上記例では、`genderCode` というリクエストパラメータがない状態でアクセスした場合、引数 genderCode に `unknown` が渡る。

注釈: 必須パラメータを指定しないでアクセスした場合は、`org.springframework.web.bind.MissingServletRequestParameterException` がスローされ、デフォルトの動作は 400(Bad Request) が応答される。ただし、defaultValue 属性を指定している場合、例外はスローされず、defaultValue 属性で指定した値が渡る。

注釈: バインドする引数の型は String 以外でも良い。型が合わない場合は `org.springframework.`

beans.TypeMismatchException がスローされ、デフォルトの動作は 400(Bad Request) が応答される。例えば、上記例で sample/hello?age=aaaa&... という URL でアクセスした場合、aaaa を Integer に変換できないため、例外がスローされる。

以下の条件に当てはまる場合は、次に説明するフォームオブジェクトにバインドすること。

- リクエストパラメータが HTML form 内の項目である。
- リクエストパラメータは HTML form 内の項目ではないが、リクエストパラメータに必須チェック以外の入力チェックを行う必要がある。
- リクエストパラメータの入力チェックエラーのエラー詳細をパラメータ毎に出力する必要がある。
- 3 つ以上のリクエストパラメータをバインドする。(保守性、可読性の観点)

リクエストパラメータをまとめて取得する

リクエストパラメータをオブジェクトにまとめて取得する場合は、フォームオブジェクトを使用する。

フォームオブジェクトは、HTML form を表現する JavaBean である。フォームオブジェクトの詳細は フォームオブジェクトの実装 を参照されたい。

以下は、@RequestParam で個別にリクエストパラメータを受け取っていたハンドラメソッドを、フォームオブジェクトで受け取るように変更した場合の実装例である。

@RequestParam を使って個別にリクエストパラメータを受け取っているハンドラメソッドは以下の通り。

```
@RequestMapping("bindRequestParams")
public String bindRequestParams(
        @RequestParam("id") String id,
        @RequestParam String name,
        @RequestParam(value = "age", required = false) Integer age,
        @RequestParam(value = "genderCode", required = false, defaultValue =
↪"unknown") String genderCode,
        Model model) {
    // do something
    return "sample/hello"; // returns view name
}
```

フォームオブジェクトクラスを作成する。

このフォームオブジェクトに対応する HTML form の jsp は *HTML form へのバインディング方法* を参照されたい。

```
public class SampleForm implements Serializable{
    private static final long serialVersionUID = 1477614498217715937L;

    private String id;
    private String name;
    private Integer age;
    private String genderCode;

    // omit setters and getters

}
```

注釈: リクエストパラメータ名とフォームオブジェクトのプロパティ名は一致させる必要がある。

上記のフォームオブジェクトに対して `id=aaa&name=bbbb&age=19&genderCode=men?tel=01234567` というパラメータが送信された場合、`id` , `name` , `age` , `genderCode` は名前が一致するプロパティに値が格納されるが、 `tel` は名前が一致するプロパティがないため、フォームオブジェクトに取り込まれない。

`@RequestParam` を使って個別に受け取っていたリクエストパラメータをフォームオブジェクトとして受け取るようにする。

```
@RequestMapping("bindRequestParams")
public String bindRequestParams(@Validated SampleForm form, // (1)
        BindingResult result,
        Model model) {
    // do something
    return "sample/hello"; // returns view name
}
```

項番	説明
(1)	`SampleForm` オブジェクトを引数として受け取る。

注釈: フォームオブジェクトを引数に用いた場合、`@RequestParam` の場合とは異なり、必須チェックは行われない。フォームオブジェクトを使用する場合は、次に説明する 入力チェックを行う を行うこと。

警告: Entity など Domain オブジェクトをそのままフォームオブジェクトとして使うこともできるが、実際には、WEB の画面上にしか存在しないパラメータ（確認用パスワードや、規約確認チェックボックス等）が存在する。Domain オブジェクトにそのような画面項目に依存する項目を入れるべきではないので、Domain オブジェクトとは別にフォームオブジェクト用のクラスを作成することを推奨する。リクエストパラメータから Domain オブジェクトを作成する場合は、一旦フォームオブジェクトにバインドしてからプロパティ値を Domain オブジェクトにコピーすること。

入力チェックを行う

リクエストパラメータがバインドされているフォームオブジェクトに対して入力チェックを行う場合は、フォームオブジェクト引数に`@Validated` アノテーションを付け、フォームオブジェクト引数の直後に`org.springframework.validation.BindingResult`(以降 `BindingResult` と呼ぶ) を引数に指定する。

入力チェックの詳細については、 入力チェック を参照されたい。

フォームオブジェクトクラスのフィールドに入力チェックで必要となるアノテーションを付加する。

```
public class SampleForm implements Serializable {
    private static final long serialVersionUID = 1477614498217715937L;

    @NotNull
    @Size(min = 10, max = 10)
    private String id;

    @NotNull
    @Size(min = 1, max = 10)
    private String name;

    @Min(1)
    @Max(100)
    private Integer age;

    @Size(min = 1, max = 10)
    private Integer genderCode;

    // omit setters and getters
}
```

フォームオブジェクト引数に`@Validated` アノテーションを付与する。

`@Validated` アノテーションを付けた引数は、ハンドラメソッド実行前に入力チェックが行われ、チェック結果が直後の `BindingResult` 引数に格納される。

フォームオブジェクトに String 型以外を指定した場合に発生する型変換エラーも BindingResult に格納されている。

```
@RequestMapping("bindRequestParams")
public String bindRequestParams(@Validated SampleForm form, // (1)
        BindingResult result, // (2)
        Model model) {
    if (result.hasErrors()) { // (3)
        return "sample/input"; // back to the input view
    }
    // do something
    return "sample/hello"; // returns view name
}
```

項番	説明
(1)	SampleForm オブジェクトに@Validated アノテーションを付与し、入力チェック対象のオブジェクトにする。
(2)	入力チェック結果が格納される BindingResult を引数に指定する。
(3)	入力チェックエラーが存在するか判定する。エラーがある場合は、true が返却される。

リダイレクト先にデータを渡す

ハンドラメソッドを実行した後にリダイレクトする場合に、リダイレクト先で表示するデータを渡したい場合は、org.springframework.web.servlet.mvc.support.RedirectAttributes(以降 RedirectAttributes と呼ぶ) をハンドラメソッドの引数として受け取り、RedirectAttributes オブジェクトに渡したいデータを追加する。

- SampleController.java

```
@RequestMapping("hello")
public String hello(RedirectAttributes redirectAttrs) { // (1)
    redirectAttrs.addFlashAttribute("hello", "Hello World!"); // (2)
    redirectAttrs.addFlashAttribute(new HelloBean("Bean Hello World!")); //␣
↪(3)
    return "redirect:/sample/hello?complete"; // (4)
}
```

```
@RequestMapping(value = "hello", params = "complete")
public String helloComplete() {
    return "sample/complete"; // (5)
}
```

- complete.jsp

```
Message : ${f:h(hello)}<br> <%-- (6) --%>
Message : ${f:h(helloBean.message)}<br> <%-- (7) --%>
```

- HTML of created by View(complete.jsp)

```
Message : Hello World!<br> <!-- (8) -->
Message : Bean Hello World!<br>  <!-- (8) -->
```

項番	説明
(1)	`RedirectAttributes` オブジェクトを引数として受け取る。
(2)	`RedirectAttributes` オブジェクトの `addFlashAttribute` メソッドを呼び出し、渡したいデータを `RedirectAttributes` オブジェクトに追加する。 例では、 `hello` という属性名で `HelloWorld!` という文字列のデータを追加している。
(3)	`addFlashAttribute` メソッドの第一引数を省略すると値に渡したオブジェクトのクラス名の先頭を小文字にした文字列が属性名になる。 例では、 `model.addFlashAttribute("helloBean", new HelloBean());` を行ったのと同じ結果となる。
(4)	画面 (View) を直接表示せず、次の画面を表示するためのリクエストにリダイレクトする。
(5)	リダイレクト後のハンドラメソッドでは、(2)(3) で追加したデータを表示する画面の View 名を返却する。

Continued on next page

表 3.14 – continued from previous page

項番	説明
(6)	View(JSP) 側では、「${属性名}」と記述することで `RedirectAttributes` を通じて flash scope に追加したデータを取得することができる。 例では HTML エスケープを行う EL 式の関数を呼び出しているため、「${f:h(属性名)}」としている。 HTML エスケープを行う EL 式の関数の詳細については、 *Cross Site Scripting* を参照されたい。
(7)	「${属性名.JavaBean のプロパティ名}」と記述することで `RedirectAttributes` に格納されている JavaBean から値を取得することができる。
(8)	HTML の出力例。

警告: Model に追加してもリダイレクト先にデータを渡すことはできない。

注釈: Model の addAttribute メソッドに非常によく似ているが、データの生存期間が異なる。RedirectAttributes の addFlashAttribute では flash scope というスコープにデータが格納され、リダイレクト後の 1 リクエスト (PRG パターンの G) でのみ追加したデータを参照することができる。2 回目以降のリクエストの時にはデータは消えている。

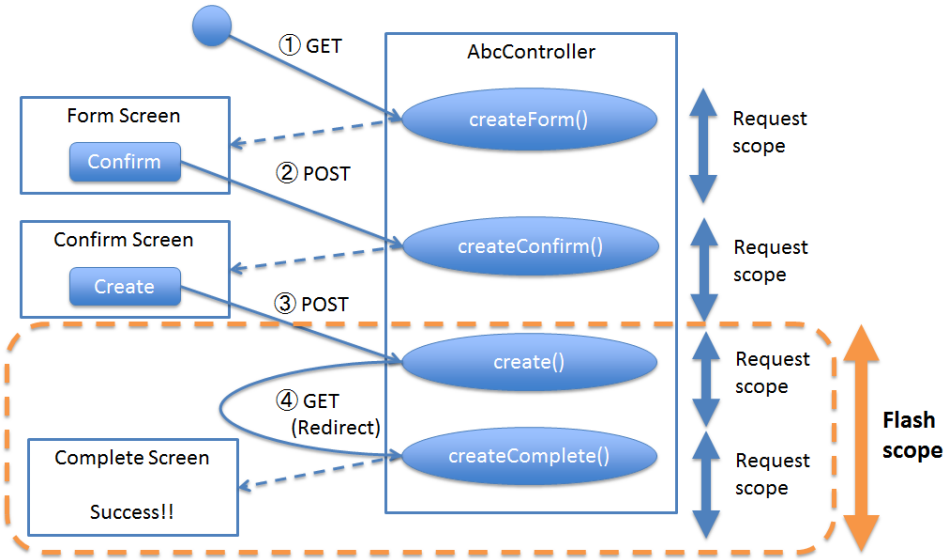


図 3.6 **Picture - Survival time of flush scope**

リダイレクト先へリクエストパラメータを渡す

リダイレクト先へ動的にリクエストパラメータを設定したい場合は、引数の RedirectAttributes オブジェクトに渡したい値を追加する。

```
@RequestMapping("hello")
public String hello(RedirectAttributes redirectAttrs) {
    String id = "aaaa";
    redirectAttrs.addAttribute("id", id); // (1)
    // must not return "redirect:/sample/hello?complete&id=" + id;
    return "redirect:/sample/hello?complete";
}
```

項番	説明
(1)	属性名にリクエストパラメータ名、属性値にリクエストパラメータの値を指定して、RedirectAttributes オブジェクトの addAttribute メソッドを呼び出す。 上記例では、 /sample/hello?complete&id=aaaa にリダイレクトされる。

警告: 上記例ではコメント化しているが、return "redirect:/sample/hello?complete&id=" + id; と結果は同じになる。ただし、 RedirectAttributes オブジェクトの addAttribute メソッドを用いると URI エンコーディングも行われるので、動的に埋め込むリクエストパラメータについては、**返り値のリダイレクト URL として組み立てるのではなく、必ず addAttribute メソッドを使用してリクエストパラメータに設定すること。**動的に埋め込まないリクエストパラメータ (上記例だと」complete」) については、返り値のリダイレクト URL に直接指定してよい。

リダイレクト先 URL のパスに値を埋め込む

リダイレクト先 URL のパスに動的に値を埋め込みたい場合は、リクエストパラメータの設定と同様引数の RedirectAttributes オブジェクトに埋め込みたい値を追加する。

```
@RequestMapping("hello")
public String hello(RedirectAttributes redirectAttrs) {
    String id = "aaaa";
    redirectAttrs.addAttribute("id", id); // (1)
    // must not return "redirect:/sample/hello/" + id + "?complete";
    return "redirect:/sample/hello/{id}?complete"; // (2)
}
```

項番	説明
(1)	属性名とパスに埋め込みたい値を指定して、RedirectAttributes オブジェクトの addAttribute メソッドを呼び出す。
(2)	リダイレクト URL の埋め込みたい箇所に「{属性名}」のパス変数を指定する。 上記例では、 /sample/hello/aaaa?complete にリダイレクトされる。

警告: 上記例ではコメント化しているが、`"redirect:/sample/hello/" + id + "?complete";` と結果は同じになる。ただし、 `RedirectAttributes` オブジェクトの `addAttribute` メソッドを用いると URL エンコーディングも行われるので、動的に埋め込むパス値については、返り値のリダイレクト **URL** として記述せずに、必ず **addAttribute** メソッドを使用し、パス変数を使って埋め込むこと。

Cookie から値を取得する

Cookie から取得したい場合は、引数に`@CookieValue` アノテーションを付与する。

```
@RequestMapping("readCookie")
public String readCookie(@CookieValue("JSESSIONID") String sessionId, Model␣
↪model) { // (1)
    // do something
    return "sample/readCookie"; // returns view name
}
```

項番	説明
(1)	`@CookieValue` アノテーションの value 属性には、Cookie 名を指定する。 上記例では、Cookie から」JSESSIONID」という Cookie 名の値が引数 sessionId に渡る。

注釈: `@RequestParam` 同様、required 属性、defaultValue 属性があり、引数の型には String 型以外の指定も可能である。詳細は、 リクエストパラメータを個別に取得する を参照されたい。

Cookie に値を書き込む

Cookie に値を書き込む場合は、`HttpServletResponse` オブジェクトの `addCookie` メソッドを直接呼び出して Cookie に追加する。

Spring MVC から Cookie に値を書き込む仕組みが提供されていないため (3.2.3 時点)、この場合に限り **HttpServletResponse** を引数に取っても良い。

```
@RequestMapping("writeCookie")
public String writeCookie(Model model,
        HttpServletResponse response) { // (1)
    Cookie cookie = new Cookie("foo", "HelloWorld!");
    response.addCookie(cookie); // (2)
    // do something
    return "sample/writeCookie";
}
```

項番	説明
(1)	Cookie を書き込むために、`HttpServletResponse` オブジェクトを引数に指定する。
(2)	`Cookie` オブジェクトを生成し、`HttpServletResponse` オブジェクトに追加する。 上記例では、 `foo` という Cookie 名で `HelloWorld!` という値を設定している。

ちなみに: `HttpServletResponse` を引数として受け取ることに変わりはないが、Cookie に値を書き込むためのクラスとして、Spring Framework から `org.springframework.web.util.CookieGenerator` というクラスが提供されている。必要に応じて使用すること。

注釈: HTTP Cookie の処理を規定する RFC 6265 では、Cookie の名前や値に一部使用できない文字があることに注意されたい。例えば、RFC 6265 に準拠した実装の Tomcat 8.5 では、Cookie の値にスペースを使用することができない。

RFC 6265(HTTP State Management Mechanism) の 4.1 SetCookie の Syntax を参照されたい。

ページネーション情報を取得する

一覧検索を行うリクエストでは、ページネーション情報が必要となる。

`org.springframework.data.domain.Pageable`(以降 `Pageable` と呼ぶ) オブジェクトをハンドラメソッドの引数に取ることで、ページネーション情報 (ページ数、取得件数) を容易に扱うことができる。

> 詳細については ページネーション を参照すること。

アップロードファイルを取得する

アップロードされたファイルを取得する方法は大きく２つある。

- フォームオブジェクトに `MultipartFile` のプロパティを用意する。
- `@RequestParam` アノテーションを付与して `org.springframework.web.multipart.MultipartFile` をハンドラメソッドの引数とする。

詳細については ファイルアップロード を参照されたい。

画面に結果メッセージを表示する

`Model` オブジェクト又は `RedirectAttributes` オブジェクトをハンドラメソッドの引数として受け取り、`ResultMessages` オブジェクトを追加することで処理の結果メッセージを表示できる。

詳細については メッセージ管理 を参照されたい。

ハンドラメソッドの返り値について

ハンドラメソッドの返り値についても様々な値をとることができる が、基本的には次に挙げるもののみを使用すること。

- String(View 名)

以下に、目的別に返り値の使用方法について説明する。

- *HTML* を応答する
- ダウンロードデータを応答する

HTML を応答する

ハンドラメソッドの実行結果を HTML として応答する場合、ハンドラメソッドの返り値は、JSP の View 名を返却する。

JSP を使って HTML を生成する場合の `ViewResolver` は、基本的には `UrlBasedViewResolver` の継承クラス (`InternalResourceViewResolver` や `TilesViewResolver` 等) となる。

以下では、JSP 用の `InternalResourceViewResolver` を使用する場合の例を記載するが、画面レイアウトがテンプレート化されている場合は `TilesViewResolver` を使用することを推奨する。

`TilesViewResolver` の使用方法については、 *Tiles による画面レイアウト* を参照されたい。

- spring-mvc.xml

```
<mvc:view-resolvers>
    <mvc:jsp prefix="/WEB-INF/views/" /> <!-- (1) -->
</mvc:view-resolvers>
```

- SampleController.java

```
@RequestMapping("hello")
public String hello() {
    // omitted
    return "sample/hello"; // (2)
}
```

項番	説明
(1)	<mvc:jsp>要素を使用して、JSP 用の `InternalResourceViewResolver` を定義する。 • `prefix` 属性には、JSP ファイルが格納されているベースディレクトリ (ファイルパスのプレフィックス) を指定する。 • `suffix` 属性には、デフォルト値として `.jsp` が適用されているため、明示的に指定する必要はない。 **注釈:** <mvc:view-resolvers>要素を使用すると、`ViewResolver` をシンプルに定義することが出来るため、本ガイドラインでは<mvc:view-resolvers>を使用することを推奨する。
(2)	ハンドラメソッドの返り値として `sample/hello` という View 名を返却した場合、 `/WEB-INF/views/sample/hello.jsp` が呼び出されて HTML が応答される。

注釈: 上記の例では JSP を使って HTML を生成しているが、Velocity や FreeMarker など他のテンプレート

エンジンを使用して HTML を生成する場合でも、ハンドラメソッドの返り値は sample/hello のままでよい。使用するテンプレートエンジンでの差分は ViewResolver によって解決される。

注釈: 単純に view 名を返すだけのメソッドを実装する場合は、<mvc:view-controller> を使用して Controller クラスの実装を代用することも可能である。

- <mvc:view-controller>を使用した Controller の定義例。

```
<mvc:view-controller path="/hello" view-name="sample/hello" />
```

警告: <mvc:view-controller>使用に関する留意点

Spring Framework 4.3 へのバージョンアップによって、<mvc:view-controller>が許可する HTTP メソッドは GET と HEAD のみに限定される様になったため (SPR-13130)、HTTP メソッドが GET と HEAD 以外 (POST など) でアクセスするページの場合、<mvc:view-controller>は使用できない。GET と HEAD 以外 (POST など) からフォワードされた場合も同様となるため、エラーページへの遷移などフォワード元の HTTP メソッドが限定できない場合には<mvc:view-controller>を使用しないよう注意されたい。

ダウンロードデータを応答する

データベースなどに格納されているデータをダウンロードデータ (application/octet-stream 等) として応答する場合、
レスポンスデータの生成 (ダウンロード処理) を行う View を作成し、処理を委譲することを推奨する。
ハンドラメソッドでは、ダウンロード対象となるデータを Model に追加し、ダウンロード処理を行う View の View 名を返却する。

View 名から View を解決する方法としては、個別の ViewResolver を作成する方法もあるが、ここでは Spring Framework から提供されている BeanNameViewResolver を使用する。
ダウンロード処理の詳細については、 ファイルダウンロード を参照されたい。

- spring-mvc.xml

```
<mvc:view-resolvers>
    <mvc:bean-name /> <!-- (1) -->
```

```
    <mvc:jsp prefix="/WEB-INF/views/" />
</mvc:view-resolvers>
```

- SampleController.java

```
@RequestMapping("report")
public String report() {
    // omitted
    return "sample/report"; // (2)
}
```

- XxxExcelView.java

```
@Component("sample/report") // (3)
public class XxxExcelView extends AbstractXlsxView { // (4)
    @Override
    protected void buildExcelDocument(Map<String, Object> model,
            Workbook workbook, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        Sheet sheet;
        Cell cell;

        sheet = workbook.createSheet("Spring");
        sheet.setDefaultColumnWidth(12);

        // write a text at A1
        cell = getCell(sheet, 0, 0);
        setText(cell, "Spring-Excel test");

        cell = getCell(sheet, 2, 0);
        setText(cell, ((Date) model.get("serverTime")).toString());
    }
}
```

項番	説明
(1)	<mvc:bean-name>要素を使用して、BeanNameViewResolver を定義する。 <mvc:view-resolvers>要素を使用して ViewResolver を定義する場合は、子要素に指定する ViewResolver の定義順が優先順位となる。上記例では、JSP用の InternalResourceViewResolver を定義するための要素 (<mvc:jsp>) より上に定義することで、JSP 用の InternalResourceViewResolver より先に BeanNameViewResolver による View 解決が行われる。 **注釈:** <mvc:view-resolvers>要素を使用すると、ViewResolver をシンプルに定義することが出来るため、本ガイドラインでは<mvc:view-resolvers>を使用することを推奨する。
(2)	ハンドラメソッドの返り値として sample/report という View 名を返却した場合、(5) で Bean 登録された View インスタンスによって生成されたデータがダウンロードデータとして応答される。
(3)	コンポーネントの名前に View 名を指定して、View オブジェクトを Bean として登録する。 上記例では、 sample/report という bean 名 (View 名) で x.y.z.app.views.XxxExcelView のインスタンスが Bean 登録される。
(4)	View の実装例。 上記例では、 org.springframework.web.servlet.view.document.AbstractXlsxView を継承し、Excel データを生成する View クラスの実装となる。

処理の実装

Controller では、業務処理の実装は行わない という点がポイントとなる。

業務処理の実装は Service で行い、Controller では業務処理が実装されている Service のメソッドを呼び出す。

業務処理の実装の詳細については ドメイン層の実装 を参照されたい。

注釈: Controller は、基本的には画面遷移の決定などの処理のルーティングと Model の設定のみ実装することに徹し、可能な限りシンプルな状態に保つこと。この方針で統一することにより、Controller で実装すべき処理が明確になり、開発規模が大きくなった場合でも Controller のメンテナンス性を保つことができる。

Controller で実装すべき処理を以下に 4 つ示す。

- 入力値の相関チェック
- 業務処理の呼び出し
- ドメインオブジェクトへの値反映
- フォームオブジェクトへの値反映

入力値の相関チェック

入力値に対する相関チェックは、`org.springframework.validation.Validator` インタフェースを実装した Validation クラス、もしくは、Bean Validation で検証を行う。
相関チェックの実装の詳細については、入力チェック を参照されたい。

相関チェックの実装自体は Controller のハンドラメソッドで行うことはないが、相関チェックを行う `Validator` を `org.springframework.web.bind.WebDataBinder` に追加する必要がある。

```
@Inject
PasswordEqualsValidator passwordEqualsValidator; // (1)

@InitBinder
protected void initBinder(WebDataBinder binder){
    binder.addValidators(passwordEqualsValidator); // (2)
}
```

項番	説明
(1)	相関チェックを行う `Validator` を Inject する。
(2)	Inject した `Validator` を `WebDataBinder` に追加する。 `WebDataBinder` に追加しておくことで、ハンドラメソッド呼び出し前に行われる入力チェック処理にて、(1) で追加した `Validator` が実行され、相関チェックを行うことが出来る。

業務処理の呼び出し

業務処理が実装されている Service を Inject し、Inject した Service のメソッドを呼び出すことで業務処理を実行する。

```
@Inject
SampleService sampleService; // (1)

@RequestMapping("hello")
public String hello(Model model){
    String message = sampleService.hello(); // (2)
    model.addAttribute("message", message);
    return "sample/hello";
}
```

項番	説明
(1)	業務処理が実装されている `Service` を Inject する。
(2)	Inject した `Service` のメソッドを呼び出し、業務処理を実行する。

ドメインオブジェクトへの値反映

本ガイドラインでは、HTML form から送信されたデータは直接ドメインオブジェクトにバインドするのではなく、フォームオブジェクトにバインドする方法を推奨している。

そのため、Controller では Service のメソッドに渡すドメインオブジェクトにフォームオブジェクトの値を反映する処理を行う必要がある。

```
@RequestMapping("hello")
public String hello(@Validated SampleForm form, BindingResult result, Model␣
↪model){
    // omitted
    Sample sample = new Sample(); // (1)
    sample.setField1(form.getField1());
    sample.setField2(form.getField2());
    sample.setField3(form.getField3());
```

```
    // ...
    // and more ...
    // ...
    String message = sampleService.hello(sample); // (2)
    model.addAttribute("message", message); // (3)
    return "sample/hello";
}
```

項番	説明
(1)	Service の引数となるドメインオブジェクトを生成し、フォームオブジェクトにバインドされている値を反映する。
(2)	Service のメソッドを呼び出し、業務処理を実行する。
(3)	業務処理から返却されたデータを `Model` に追加する。

ドメインオブジェクトへ値を反映する処理は、Controller のハンドラメソッド内で実装してもよいが、コード量が多くなる場合はハンドラメソッドの可読性を考慮して Helper クラスのメソッドに処理を委譲することを推奨する。

以下に Helper メソッドに処理を委譲した場合の例を示す。

- SampleController.java

```
@Inject
SampleHelper sampleHelper; // (1)

@RequestMapping("hello")
public String hello(@Validated SampleForm form, BindingResult result){
    // omitted
    String message = sampleHelper.hello(form); // (2)
    model.addAttribute("message", message);
    return "sample/hello";
}
```

- SampleHelper.java

```
public class SampleHelper {

    @Inject
```

```
    SampleService sampleService;

    public String hello(SampleForm form){ // (3)
        Sample sample = new Sample();
        sample.setField1(form.getField1());
        sample.setField2(form.getField2());
        sample.setField3(form.getField3());
        // ...
        // and more ...
        // ...
        String message = sampleService.hello(sample);
        return message;
    }
}
```

項番	説明
(1)	Controller に Helper クラスのオブジェクトを Inject する。
(2)	Inject した Helper クラスのメソッドを呼び出すことで、ドメインオブジェクトへの値の反映を行っている。Helper クラスに処理を委譲することで、Controller の実装をシンプルな状態に保つことができる。
(3)	ドメインオブジェクトを生成した後に Service クラスのメソッド呼び出し、業務処理を実行している。

注釈: Helper クラスに処理を委譲する以外の方法として、Bean 変換機能を使用する方法がある。Bean 変換機能の詳細は、*Bean マッピング (Dozer)* を参照されたい。

フォームオブジェクトへの値反映

本ガイドラインでは、HTML form の項目にバインドするデータはドメインオブジェクトではなく、フォームオブジェクトを使用する方法を推奨している。

そのため、Controller では Service のメソッドから返却されたドメインオブジェクトの値をフォームオブジェクトに反映する処理を行う必要がある。

```
@RequestMapping("hello")
public String hello(SampleForm form, BindingResult result, Model model){
    // omitted
    Sample sample = sampleService.getSample(form.getId()); // (1)
    form.setField1(sample.getField1()); // (2)
    form.setField2(sample.getField2());
    form.setField3(sample.getField3());
    // ...
    // and more ...
    // ...
    model.addAttribute(sample); // (3)
    return "sample/hello";
}
```

項番	説明
(1)	業務処理が実装されている Service のメソッドを呼び出し、ドメインオブジェクトを取得する。
(2)	取得したドメインオブジェクトの値をフォームオブジェクトに反映する。
(3)	表示のみ行う項目がある場合は、データを参照できるようにするために、`Model` にドメインオブジェクトを追加する。

注釈: 画面に表示のみ行う項目については、フォームオブジェクトに項目をもつのではなく、Entity などのドメインオブジェクトから直接値を参照することを推奨する。

フォームオブジェクトへの値反映処理は、Controller のハンドラメソッド内で実装してもよいが、コード量が多くなる場合はハンドラメソッドの可読性を考慮して Helper クラスのメソッドに委譲することを推奨する。

- SampleController.java

```
@RequestMapping("hello")
public String hello(@Validated SampleForm form, BindingResult result){
    // omitted
    Sample sample = sampleService.getSample(form.getId());
    sampleHelper.applyToForm(sample, form); // (1)
    model.addAttribute(sample);
    return "sample/hello";
}
```

- SampleHelper.java

```
public void applyToForm(SampleForm destForm, Sample srcSample){
    destForm.setField1(srcSample.getField1()); // (2)
    destForm.setField2(srcSample.getField2());
    destForm.setField3(srcSample.getField3());
    // ...
    // and more ...
    // ...
}
```

項番	説明
(1)	ドメインオブジェクトの値をフォームオブジェクトに反映するためのメソッドを呼び出す。
(2)	ドメインオブジェクトの値をフォームオブジェクトに反映するためのメソッドにて、ドメインオブジェクトの値をフォームオブジェクトに反映する。

注釈: Helper クラスに処理を委譲する以外の方法として、Bean 変換機能を使用する方法がある。Bean 変換機能の詳細は、*Bean マッピング (Dozer)* を参照されたい。

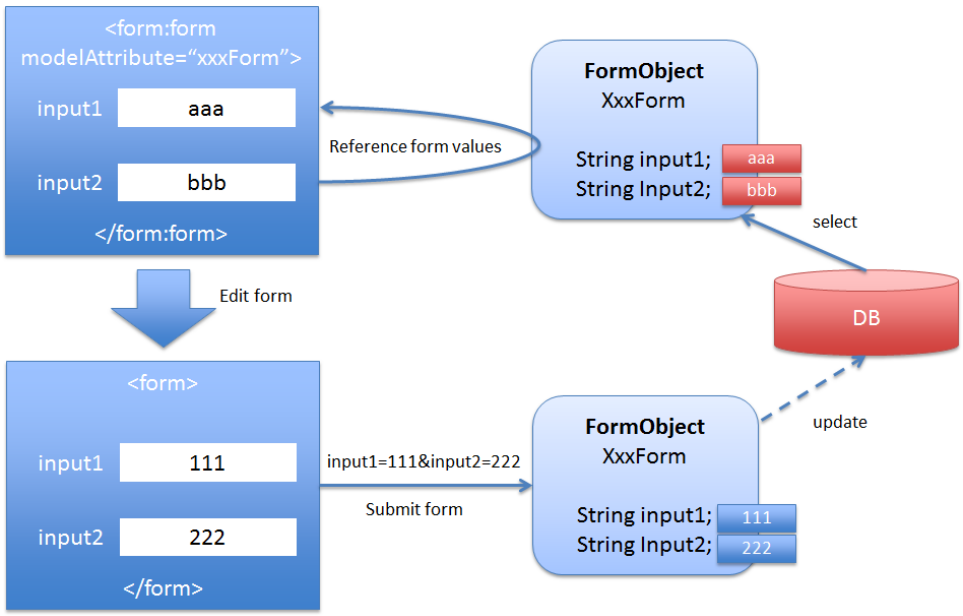
3.4.2 フォームオブジェクトの実装

フォームオブジェクトは HTML 上の form を表現するオブジェクト (JavaBean) であり、以下の役割を担う。

1. データベース等で保持している業務データを保持し、**HTML(JSP) form** から参照できるようにする。
2. **HTML form** から送信されたリクエストパラメータを保持し、ハンドラメソッドで参照できるようにする。

フォームオブジェクトの実装について、以下 4 点に着目して説明する。

- フォームオブジェクトの作成方法
- フォームオブジェクトの初期化方法



- *HTML form* へのバインディング方法
- リクエストパラメータのバインディング方法

フォームオブジェクトの作成方法

フォームオブジェクトは JavaBean として作成する。Spring Framework では、HTML form から送信されたリクエストパラメータ (文字列) を、フォームオブジェクトに定義されている型に変換してからバインドする機能を提供しているため、フォームオブジェクトに定義するフィールドの型は、`java.lang.String` だけではなく、任意の型で定義することができる。

```
public class SampleForm implements Serializable {
    private String id;
    private String name;
    private Integer age;
    private String genderCode;
    private Date birthDate;
    // ommitted getter/setter
}
```

ちなみに：Spring Framework から提供されている型変換を行う仕組みについて

Spring Framework は、以下の 3 つの仕組みを使って型変換を行っており、基本的な型への変換は標準でサポートされている。各変換機能の詳細については、リンク先のページを参照されたい。

- Spring Type Conversion
- Spring Field Formatting

- java.beans.PropertyEditor implementations

警告: フォームオブジェクトには画面に表示のみ行う項目は保持せず、HTML form の項目のみ保持することを推奨する。フォームオブジェクトに画面表示のみ行う項目の値を設定した場合、フォームオブジェクトを HTTP セッションオブジェクトに格納する際にメモリを多く消費する事になり、メモリ枯渇の原因になる可能性がある。画面表示のみの項目は、Entity などのドメイン層のオブジェクトをリクエストスコープに追加 (`Model.addAttribute`) することで HTML(JSP) にデータを渡すことを推奨する。

フィールド単位の数値型変換

`@NumberFormat` アノテーションを使用することでフィールド毎に数値の形式を指定することが出来る。

```
public class SampleForm implements Serializable {
    @NumberFormat(pattern = "#,#") // (1)
    private Integer price;
    // ommitted getter/setter
}
```

項番	説明
(1)	HTML form から送信されるリクエストパラメータの数値形式を指定する。例では、pattern として `#,#` 形式を指定しているので、「,」でフォーマットされた値をバインドすることができる。リクエストパラメータの値が `1,050` の場合、フォームオブジェクトの price には `1050` の Integer オブジェクトがバインドされる。

`@NumberFormat` アノテーションで指定できる属性は以下の通り。

項番	属性名	説明
1.	style	数値のスタイルを指定する。詳細は、NumberFormat.Style の Javadoc を参照されたい。
2.	pattern	Java の数値形式を指定する。詳細は、DecimalFormat の Javadoc を参照されたい。

フィールド単位の日時型変換

`@DateTimeFormat` アノテーションを使用することでフィールド毎に日時の形式を指定することが出来る。

```
public class SampleForm implements Serializable {
    @DateTimeFormat(pattern = "yyyyMMdd") // (1)
    private Date birthDate;
    // ommitted getter/setter
}
```

項番	説明
(1)	HTML form から送信されるリクエストパラメータの日時形式を指定する。例では、pattern として `yyyyMMdd` 形式を指定している。リクエストパラメータの値が `20131001` の場合、フォームオブジェクトの birthDate には 2013 年 10 月 1 日の Date オブジェクトがバインドされる。

`@DateTimeFormat` アノテーションで指定できる属性は以下の通り。

項番	属性名	説明
1.	iso	ISO の日時形式を指定する。詳細は、DateTimeFormat.ISO の Javadoc を参照。
2.	pattern	Java の日時形式を指定する。詳細は、SimpleDateFormat の Javadoc を参照されたい。
3.	style	日付と時刻のスタイルを 2 桁の文字列として指定する。 1 桁目が日付のスタイル、2 桁目が時刻のスタイルとなる。 スタイルとして指定できる値は以下の値となる。 S : `java.text.DateFormat.SHORT` と同じ形式となる。 M : `java.text.DateFormat.MEDIUM` と同じ形式となる。 L : `java.text.DateFormat.LONG` と同じ形式となる。 F : `java.text.DateFormat.FULL` と同じ形式となる。 - : 省略を意味するスタイル。 指定例及び変換例) MM : Dec 9, 2013 3:37:47 AM M- : Dec 9, 2013 -M : 3:41:45 AM

Controller 単位の型変換

@InitBinder アノテーションを使用することで Controller 毎に型変換の定義を指定する事も出来る。

```
@InitBinder // (1)
public void initWebDataBinder(WebDataBinder binder) {
    binder.registerCustomEditor(
            Long.class,
            new CustomNumberEditor(Long.class, new DecimalFormat("#,#"),␣
↪true)); // (2)
}
```

```
@InitBinder("sampleForm") // (3)
public void initSampleFormWebDataBinder(WebDataBinder binder) {
    // ...
}
```

項番	説明
(1)	@InitBinder アノテーション を付与したメソッド用意すると、バインド処理が行われる前にこのメソッドが呼び出され、デフォルトの動作をカスタマイズすることができる。
(2)	例では、Long 型のフィールドの数値形式を #,# に指定しているので、「,」でフォーマットされた値をバインドすることができる。
(3)	@InitBinder アノテーションの value 属性にフォームオブジェクトの属性名を指定することで、フォームオブジェクト毎にデフォルトの動作をカスタマイズすることもできる。例では、 sampleForm という属性名のフォームオブジェクトに対するバインド処理が行われる前にメソッドが呼び出される。

入力チェック用のアノテーションの指定

フォームオブジェクトのバリデーションは、Bean Validation を使用して行うため、フィールドの制約条件を示すアノテーションを指定する必要がある。入力チェックの詳細は、入力チェック を参照されたい。

フォームオブジェクトの初期化方法

HTML の form にバインドするフォームオブジェクトの事を form-backing bean と呼び、`@ModelAttribute` アノテーションを使うことで結びつけることができる。form-backing bean の初期化は、`@ModelAttribute` アノテーションを付与したメソッドで行う。このようなメソッドのことを本ガイドラインでは ModelAttribute メソッドと呼び、`setUpXxxForm` というメソッド名で定義することを推奨する。

```
@ModelAttribute // (1)
public SampleForm setUpSampleForm() {
    SampleForm form = new SampleForm();
    // populate form
    return form;
}
```

```
@ModelAttribute("xxx") // (2)
public SampleForm setUpSampleForm() {
    SampleForm form = new SampleForm();
    // populate form
    return form;
}
```

```
@ModelAttribute
public SampleForm setUpSampleForm(
        @CookieValue(value = "name", required = false) String name, // (3)
        @CookieValue(value = "age", required = false) Integer age,
        @CookieValue(value = "birthDate", required = false) Date birthDate) {
    SampleForm form = new SampleForm();
    form.setName(name);
    form.setAge(age);
    form.setBirthDate(birthDate);
    return form;
}
```

項番	説明
(1)	`Model` に追加するための属性名は、クラス名の先頭を小文字にした値（デフォルト値）が設定される。この例では `sampleForm` が属性名になる。返却したオブジェクトは、`model.addAttribute(form)` 相当の処理が実行され `Model` に追加される。
(2)	`Model` に追加するための属性名を指定したい場合は、`@ModelAttribute` アノテーションの value 属性に指定する。この例では `xxx` が属性名になる。返却したオブジェクトは、`model.addAttribute("xxx", form)` 相当の処理が実行され `Model` に追加される。デフォルト値以外の属性名を指定した場合、ハンドラメソッドの引数としてフォームオブジェクトを受け取る時に`@ModelAttribute("xxx")` の指定が必要になる。
(3)	ModelAttribute メソッドは、ハンドラメソッドと同様に初期化に必要なパラメータを渡すこともできる。例では、`@CookieValue` アノテーションを使用して Cookie の値をフォームオブジェクトに設定している。

注釈: フォームオブジェクトにデフォルト値を設定したい場合は ModelAttribute メソッドで値を設定すること。例の (3) では Cookie から値を取得しているが、定数クラスなどに定義されている固定値を直接設定してもよい。

注釈: ModelAttribute メソッドは Controller 内に複数定義することができる。各メソッドは Controller のハンドラメソッドが呼び出される前に毎回実行される。

警告: ModelAttribute メソッドはリクエストごとにメソッドが実行されるため、特定のリクエストの時のみに必要なオブジェクトについて ModelAttribute メソッドを使って生成すると、無駄なオブジェクトの生成及び初期化処理が行われる点に注意すること。特定のリクエストのみで必要なオブジェクトについては、ハンドラメソッド内で生成し `Model` に追加する方法にすること。

HTML form へのバインディング方法

`Model` に追加されたフォームオブジェクトは`<form:xxx>`タグを用いて、HTML(JSP) の form にバインドすることができる。

`<form:xxx>`タグの詳細は、 Using Spring’s form tag library を参照されたい。

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %> <!-
↪- (1) -->
```

```
<form:form modelAttribute="sampleForm"
           action="${pageContext.request.contextPath}/sample/hello"> <!-- (2)␣
↪-->
    Id         : <form:input path="id" /><form:errors path="id" /><br /> <!--␣
↪(3) -->
    Name       : <form:input path="name" /><form:errors path="name" /><br />
    Age        : <form:input path="age" /><form:errors path="age" /><br />
    Gender     : <form:input path="genderCode" /><form:errors path="genderCode
↪" /><br />
    Birth Date : <form:input path="birthDate" /><form:errors path="birthDate"␣
↪/><br />
</form:form>
```

項番	説明
(1)	<form:form>タグを使用するための taglib の定義を行う。
(2)	<form:form>タグの modelAttribute 属性には、Model に格納されているフォームオブジェクトの属性名を指定する。
(3)	<form:input>タグの path 属性には、フォームオブジェクトのプロパティ名を指定する。

リクエストパラメータのバインディング方法

HTML form から送信されたリクエストパラメータは、フォームオブジェクトにバインドし、Controller のハンドラメソッドの引数に渡すことができる。

```
@RequestMapping("hello")
public String hello(
        @Validated SampleForm form, // (1)
        BindingResult result,
        Model model) {
    if (result.hasErrors()) {
        return "sample/input";
    }
    // process form...
    return "sample/hello";
}
```

```
@ModelAttribute("xxx")
public SampleForm setUpSampleForm() {
    SampleForm form = new SampleForm();
    // populate form
    return form;
}

@RequestMapping("hello")
public String hello(
        @ModelAttribute("xxx") @Validated SampleForm form, // (2)
        BindingResult result,
        Model model) {
    // ...
}
```

項番	説明
(1)	フォームオブジェクトにリクエストパラメータが反映された状態で、Controller のハンドラメソッドの引数に渡される。
(2)	ModelAttribute メソッドにて属性名を指定した場合、`@ModelAttribute("xxx")` といった感じで、フォームオブジェクトの属性名を明示的に指定する必要がある。

警告: ModelAttribute メソッドで指定した属性名とメソッドの引数で指定した属性名が異なる場合、ModelAttribute メソッドで生成したインスタンスとは別のインスタンスが生成されるので注意が必要。ハンドラメソッドで属性名の指定を省略した場合、クラス名の先頭を小文字にした値が属性名として扱われる。

バインディング結果の判定

HTML form から送信されたリクエストパラメータをフォームオブジェクトにバインドする際に発生したエラー（入力チェックエラーも含む）は、`org.springframework.validation.BindingResult` に格納される。

```
@RequestMapping("hello")
public String hello(
        @Validated SampleForm form,
        BindingResult result, // (1)
        Model model) {
    if (result.hasErrors()) { // (2)
        return "sample/input";
    }
    // ...
}
```

項番	説明
(1)	フォームオブジェクトの直後に `BindingResult` を宣言すると、フォームオブジェクトへのバインド時のエラーと入力チェックエラーを参照することができる。
(2)	`BindingResult.hasErrors()` を呼び出すことで、フォームオブジェクトの入力値のエラー有無を判定することができる。

フィールドエラーの有無、グローバルエラー (相関チェックエラーなどのクラスレベルのエラー) の有無を個別に判定することもできるので、要件に応じて使い分けること。

項番	メソッド	説明
1.	`hasGlobalErrors()`	グローバルエラーの有無を判定するメソッド
2.	`hasFieldErrors()`	フィールドエラーの有無を判定するメソッド
3.	`hasFieldErrors(String field)`	指定したフィールドのエラー有無を判定するメソッド

3.4.3 View の実装

View は以下の役割を担う。

1. **クライアントに応答するレスポンスデータ (HTML) を生成する。**

   View はモデル（フォームオブジェクトやドメインオブジェクトなど）から必要なデータを取得し、クライアントが描画するために必要な形式でレスポンスデータを生成する。

JSP の実装

クライアントに HTML を応答する場合は、JSP を使用して View を実装する。

JSP を呼び出すための `ViewResolver` は、Spring Framework より提供されているので、提供されているクラスを利用する。`ViewResolver` の設定方法は、 *HTML を応答する* を参照されたい。

以下に、基本的な JSP の実装方法について説明する。

- インクルード用の共通 *JSP* の作成
- モデルに格納されている値を表示する
- モデルに格納されている数値を表示する
- モデルに格納されている日時を表示する
- リクエスト *URL* を生成する

- *HTML form* へフォームオブジェクトをバインドする
- 入力チェックエラーを表示する
- 処理結果のメッセージを表示する
- コードリストを表示する
- 固定文言を表示する
- 条件によって表示を切り替える
- コレクションの要素に対して表示処理を繰り返す
- ページネーション用のリンクを表示する
- 権限によって表示を切り替える

本章では代表的な JSP タグライブラリの使い方は説明しているが、全ての JSP タグライブラリの説明はしていないので、詳細な使い方については、それぞれのドキュメントを参照すること。

項番	JSP タグライブラリ名	ドキュメント
1.	Spring’s form tag library	• https://docs.spring.io/spring/docs/4.3.23.RELEASE/spring-framework-reference/html/view.html#view-jsp-formtaglib • https://docs.spring.io/spring/docs/4.3.23.RELEASE/spring-framework-reference/html/spring-form-tld.html
2.	Spring’s tag library	• https://docs.spring.io/spring/docs/4.3.23.RELEASE/spring-framework-reference/html/spring-tld.html
3.	JSTL	• http://download.oracle.com/otndocs/jcp/jstl-1.2-mrel2-eval-oth-JSpec/
4.	Common library’s tags & el functions	• 本ガイドラインの「共通ライブラリが提供する *JSP Tag Library* と *EL Functions*」

警告: terasoluna-gfw-web 1.0.0.RELEASE を使用している場合は、Spring’s form tag library から提供されている`<form:form>`タグを使う際は、必ず `action` 属性を指定すること。

terasoluna-gfw-web 1.0.0.RELEASE が依存している Spring MVC(3.2.4.RELEASE) では、`<form:form>`タグの `action` 属性を省略した場合、XSS(Cross-site scripting) の脆弱性が存在する。脆弱性に関する情報については、National Vulnerability Database (NVD) の CVE-2014-1904 を参照されたい。

> 尚、terasoluna-gfw-web 1.0.1.RELEASE 以上では、XSS 対策が行われている Spring MVC(3.2.10.RELEASE 以上) に依存しているため、本脆弱性は存在しない。

インクルード用の共通 JSP の作成

全ての JSP で必要となるディレクティブの宣言などを行うための JSP を作成する。この JSP を `web.xml` の `<jsp-config>/<jsp-property-group>/<include-prelude>` 要素に指定することで、個々の JSP で宣言する必要がなくなる。なお、このファイルはブランクプロジェクトで提供している。

- include.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%> <%-- (1) --%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%> <%--␣
↪(2) --%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec"%>

<%@ taglib uri="http://terasoluna.org/functions" prefix="f"%> <%-- (3) --%>
<%@ taglib uri="http://terasoluna.org/tags" prefix="t"%>
```

- web.xml

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <el-ignored>false</el-ignored>
        <page-encoding>UTF-8</page-encoding>
        <scripting-invalid>false</scripting-invalid>
        <include-prelude>/WEB-INF/views/common/include.jsp</include-prelude>
↪<!-- (4) -->
    </jsp-property-group>
</jsp-config>
```

項番	説明
(1)	JSTL の JSP タグライブラリを宣言している。 例では、 `core` と `fmt` を利用している。
(2)	Spring Framework の JSP タグライブラリを宣言している。 例では、 `spring` と `form` と `sec` を利用している。
(3)	共通ライブラリから提供している JSP タグライブラリを宣言している。
(4)	インクルード用の JSP(`/WEB-INF/views/common/include.jsp`) に指定した内容が、各 JSP(`<url-pattern>`で指定されているファイル) の先頭にインクルードされる。

注釈: ディレクティブの詳細は、 JavaServer Pages Specification(Version2.2) の 「JSP.1.10 Directives」 を参照されたい。

注釈: <jsp-property-group>要素の詳細は、 JavaServer Pages Specification(Version2.2) の 「JSP.3.3 JSP Property Groups」 を参照されたい。

モデルに格納されている値を表示する

モデル（フォームオブジェクトやドメインオブジェクトなど）に格納されている値を HTML に表示する場合、EL 式又は JSTL から提供されている JSP タグライブラリを使用する。

EL 式を使用して表示する。

- SampleController.java

```
@RequestMapping("hello")
public String hello(Model model) {
    model.addAttribute(new HelloBean("Bean Hello World!")); // (1)
    return "sample/hello"; // returns view name
}
```

- hello.jsp

```
Message : ${f:h(helloBean.message)} <%-- (2) --%>
```

項番	説明
(1)	Model オブジェクトに HelloBean オブジェクトを追加する。
(2)	View(JSP) 側では、「${属性名.JavaBean のプロパティ名}」と記述することで Model オブジェクトに追加したデータを取得することができる。 例では HTML エスケープを行う EL 式の関数を呼び出しているため、「${f:h(属性名.JavaBean のプロパティ名)}」としている。

注釈: 共通部品より EL 式用の HTML エスケープ関数 (f:h) を提供しているので、EL 式を使用して HTML に値を出力する場合は、必ず使用すること。HTML エスケープを行う EL 式の関数の詳細については、 *Cross Site Scripting* を参照されたい。

JSTL の JSP タグライブラリから提供されている <c:out> タグを使用して表示する。

```
Message : <c:out value="${helloBean.message}" /> <%-- (1) --%>
```

項番	説明
(1)	EL 式で取得した値を <c:out> タグの value 属性に指定する。HTML エスケープも行われる。

注釈: <c:out> の詳細は、JavaServer Pages Standard Tag Library(Version 1.2) の 「CHAPTER 4 General-Purpose Actions」 を参照されたい。

モデルに格納されている数値を表示する

数値型の値をフォーマットして出力する場合、JSTL から提供されている JSP タグライブラリを使用する。

JSTL の JSP タグライブラリから提供されている <fmt:formatNumber> タグを使用して表示する。

```
Number Item : <fmt:formatNumber value="${helloBean.numberItem}" pattern="0.00
↪" /> <%-- (1) --%>
```

項番	説明
(1)	EL 式で取得した値を <fmt:formatNumber> タグの value 属性に指定する。表示するフォーマットは pattern 属性に指定する。例では、0.00 を指定している。 仮に ${helloBean.numberItem} で取得した値が 1.2 の場合、画面には 1.20 が出力される。

注釈: <fmt:formatNumber> の詳細は、JavaServer Pages Standard Tag Library(Version 1.2) の「CHAPTER 9 Formatting Actions」 を参照されたい。

モデルに格納されている日時を表示する

日時型の値をフォーマットして出力する場合、JSTL から提供されている JSP タグライブラリを使用する。

JSTL の JSP タグライブラリから提供されている <fmt:formatDate> タグを使用して表示する。

```
Date Item : <fmt:formatDate value="${helloBean.dateItem}" pattern="yyyy-MM-dd
↪" /> <%-- (1) --%>
```

項番	説明
(1)	EL 式で取得した値を <fmt:formatDate> タグの value 属性に指定する。表示するフォーマットは pattern 属性に指定する。例では、yyyy-MM-dd を指定している。 仮に ${helloBean.dateItem} で取得した値が 2013 年 3 月 2 日の場合、画面には 2013-03-02 が出力される。

注釈: <fmt:formatDate> の詳細は、JavaServer Pages Standard Tag Library(Version 1.2) の 「CHAPTER

9 Formatting Actions」 を参照されたい。

注釈: 日時オブジェクトの型として、Joda Time から提供されている `org.joda.time.DateTime` などを利用する場合は、Jada Time から提供されている JSP タグライブラリを使用すること。Joda Time の詳細は、日付操作 *(Joda Time)* を参照されたい。

リクエスト URL を生成する

HTML の`<form>`要素 (JSP タグライブラリの`<form:form>`要素) の `action` 属性や`<a>`要素の `href` 属性などに対してリクエスト URL(Controller のメソッドを呼び出すための URL) を設定する場合は、以下のいずれかの方法を使用して URL を生成する。

- 文字列としてリクエスト URL を組み立てる
- Spring Framework 4.1 から追加された EL 関数を使用してリクエスト URL を組み立てる

注釈: どちらの方法を使用してもよいが、一つのアプリケーションの中で混在して使用することは、保守性を低下させる可能性があるので避けた方がよい。

以降の説明で使用する Controller のメソッドの実装サンプルを示す。

以降の説明では、以下に示すメソッドを呼び出すためのリクエスト URL を生成するための実装方法について説明する。

```
package com.example.app.hello;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@RequestMapping("hello")
@Controller
public class HelloController {

    // (1)
    @RequestMapping({"", "/"})
```

```
    public String hello() {
        return "hello/home";
    }

}
```

項番	説明
(1)	このメソッドに割り当てられるリクエスト URL は、{コンテキストパス}/hello となる。

文字列としてリクエスト URL を組み立てる

まず、文字列としてリクエスト URL を組み立てる方法について説明する。

```
<form action="${pageContext.request.contextPath}/hello"> <!-- (2) -->
    <!-- ... -->
</form>
```

項番	説明
(2)	pageContext(JSP の暗黙オブジェクト) から Web アプリケーションに割り振られているコンテキスパスを取得し (${pageContext.request.contextPath})、コンテキストパスの後ろに呼び出す Controller のメソッドに割り振られているサーブレットパス (上記例では、/hello) を加える。

ちなみに: URL を組み立てる JSP タグライブラリとして、

- JSTL から提供されている <c:url>
- Spring Framework から提供されている <spring:url>

が存在する。これらの JSP タグライブラリを使用して、リクエスト URL を組み立ててもよい。

リクエスト URL を動的に組み立てる必要がある場合は、これらの JSP タグライブラリを使用して URL を組み立てた方がよいケースがある。

Spring Framework 4.1 から追加された EL 関数を使用してリクエスト URL を組み立てる

つぎに、Spring Framework 4.1 から追加された EL 関数 (`spring:mvcUrl`) を使用してリクエスト URL を組み立てる方法について説明する。

`spring:mvcUrl` 関数を使用すると、Controller のメソッドのメタ情報 (メソッドシグネチャやアノテーションなど) と連携して、リクエスト URL を組み立てる事ができる。

```
<form action="${spring:mvcUrl('HC#hello').build()}"> <!-- (3) -->
    <!-- ... -->
</form>
```

項番	説明
(3)	`spring:mvcUrl` 関数の引数には、呼び出す Controller のメソッドに割り振られているリクエストマッピング名を指定する。 `spring:mvcUrl` 関数からは、リクエスト URL を組み立てるクラス (`MvcUriComponentsBuilder.MethodArgumentBuilder`) のオブジェクトが返却される。`MvcUriComponentsBuilder.MethodArgumentBuilder` クラスには、 • `arg` メソッド • `build` メソッド • `buildAndExpand` メソッド が用意されており、それぞれ、以下の役割を持つ。 • `arg` メソッドは、Controller のメソッドの引数に渡す値を指定するためのメソッドである。 • `build` メソッドは、リクエスト URL を生成するためのメソッドである。 • `buildAndExpand` メソッドは、Controller のメソッドの引数として宣言されていない動的な部分 (パス変数など) に埋め込む値を指定した上で、リクエスト URL を生成するためのメソッドである。 上記例では、リクエスト URL が静的な URL であるため、`build` メソッドのみを呼び出してリクエスト URL を生成している。リクエスト URL が動的な URL(パス変数やクエリ文字列が存在する URL) の場合は、`arg` メソッドや `buildAndExpand` メソッドを呼び出す必要がある。 `arg` メソッドと `buildAndExpand` メソッドの具体的な使用例については、「Spring Framework Reference Documentation(Building URIs to Controllers and methods from views)」を参照されたい。

注釈: リクエストマッピング名について

リクエストマッピング名は、デフォルト実装 (`org.springframework.web.servlet.mvc.method.RequestMappingInfoHandlerMethodMappingNamingStrategy` の実装) では、「クラス名の大文字部分 (クラスの短縮名) + 「`#`」 + メソッド名」となる。

リクエストマッピング名は重複しないようにする必要がある。名前が重複してしまった場合は、`@RequestMapping` アノテーションの `name` 属性に一意となる名前を指定する必要がある。

Controller のメソッドに割り当てられたリクエストマッピング名を確認したい場合は、logback.xml に以下の設定を追加すればよい。

```
<logger name="org.springframework.web.servlet.mvc.method.annotation.
↪RequestMappingHandlerMapping">
    <level value="trace" />
</logger>
```

上記設定を行った後に再起動すると、以下のようなログが出力されるようになる。

```
date:2014-12-09 18:34:29        thread:RMI TCP Connection(2)-127.0.0.1  X-
↪Track:        level:TRACE     logger:o.s.w.s.m.m.a.
↪RequestMappingHandlerMapping       message:Mapping name=HC#hello
```

HTML form へフォームオブジェクトをバインドする

HTML form へフォームオブジェクトをバインドし、フォームオブジェクトで保持している値を表示する場合、Spring Framework から提供されている JSP タグライブラリを使用する。

Spring Framework から提供されている <form:form> タグを使用してバインドする。

```
<form:form action="${pageContext.request.contextPath}/sample/hello"
           modelAttribute="sampleForm"> <%-- (1) --%>
    Id : <form:input path="id" /> <%-- (2) --%>
</form:form>
```

項番	説明
(1)	<form:form>タグの modelAttribute 属性に、Model に格納されているフォームオブジェクトの属性名を指定する。
(2)	<form:xxx>タグの path 属性に、バインドしたいプロパティのプロパティ名を指定する。xxx の部分は、入力項目のタイプによってかわる。

注釈: <form:form>、<form:xxx>タグの詳細は、 Using Spring’s form tag library を参照されたい。

入力チェックエラーを表示する

入力チェックエラーの内容を表示する場合、Spring Framework から提供されている JSP タグライブラリを使用する。

Spring Framework から提供されている <form:errors> タグを使用して表示する。
詳細は、 入力チェック を参照されたい。

```
<form:form action="${pageContext.request.contextPath}/sample/hello"
           modelAttribute="sampleForm">
    Id : <form:input path="id" /><form:errors path="id" /><%-- (1) --%>
</form:form>
```

項番	説明
(1)	<form:errors>タグの path 属性に、エラー表示したいプロパティのプロパティ名を指定する。

処理結果のメッセージを表示する

処理結果を通知するメッセージを表示する場合、共通部品から提供している JSP タグライブラリを使用する。

共通部品から提供している <t:messagesPanel> タグを使用する。
詳細は、 メッセージ管理 を参照されたい。

```
<div class="messages">
    <h2>Message pattern</h2>
    <t:messagesPanel /> <%-- (1) --%>
</div>
```

項番	説明
(1)	resultMessages という属性名で格納されているメッセージを出力する。

コードリストを表示する

共通部品から提供されているコードリストを表示する場合は、Spring Framework から提供されている JSP タグライブラリを使用する。

JSP からコードリストを参照する場合は、 `java.util.Map` インタフェースと同じ方法で参照することができる。
詳細は、 コードリスト を参照されたい。

コードリストをセレクトボックスに表示する。

```
<form:select path="orderStatus">
    <form:option value="" label="--Select--" />
    <form:options items="${CL_ORDERSTATUS}" /> <%-- (1) --%>
</form:select>
```

項番	説明
(1)	コードリスト名 (`CL_ORDERSTATUS`) を属性名として、コードリスト (`java.util.Map` インタフェース) が格納されている。そのため JSP では、EL 式を使ってコードリスト (`java.util.Map` インタフェース) にアクセスすることができる。取得した `Map` インタフェースを `<form:options>` の items 属性に渡すことで、コードリストをセレクトボックスに表示することができる。

セレクトボックスで選択した値のコード名を表示する。

```
Order Status : ${f:h(CL_ORDERSTATUS[orderForm.orderStatus])}
```

項番	説明
(1)	セレクトボックス作成時と同様に、コードリスト名 (`CL_ORDERSTATUS`) を属性名として、コードリスト (`java.util.Map` インタフェース) を取得する。取得した `Map` インタフェースのキー値として、セレクトボックスで選択した値を指定することで、コード名を表示することができる。

固定文言を表示する

画面名、項目名、ガイダンス用のメッセージなどについては、国際化の必要がない場合は JSP に直接記載してもよい。

ただし、国際化の必要がある場合は Spring Framework から提供されている JSP タグライブラリを使用して、プロパティファイルから取得した値を表示する。

Spring Framework から提供されている <spring:message> タグを使用して表示する。

詳細は、 国際化 を参照されたい。

- properties

```
# (1)
label.orderStatus=注文ステータス
```

- jsp

```
<spring:message code="label.orderStatus" text="Order Status" /> : <%-- (2) --
↪%>
    ${f:h(CL_ORDERSTATUS[orderForm.orderStatus])}
```

項番	説明
(1)	プロパティファイルにラベルの値を定義する。
(2)	<spring:message> の code 属性にプロパティファイルのキー名を指定するとキー名に一致するプロパティ値が表示される。

注釈: text 属性に指定した値は、プロパティ値が取得できなかった場合に表示される。

条件によって表示を切り替える

モデルが保持する値によって表示を切り替えたい場合は、JSTL から提供されている JSP タグライブラリを使用する。

JSTL の JSP タグライブラリから提供されている <c:if> タグ又は <c:choose> を使用して、表示の切り替えを行う。

<c:if> を使用して表示を切り替える。

```
<c:if test="${orderForm.orderStatus != 'complete'}"> <%-- (1) --%>
        <%-- ... --%>
</c:if>
```

項番	説明
(1)	<c:if> の test 属性に分岐に入る条件を実装する。例では注文ステータスが 'complete' ではない場合に分岐内の表示処理が実行される。

<c:choose> を使用して表示を切り替える。

```
<c:choose>
    <c:when test="${customer.type == 'premium'}"> <%-- (1) --%>
        <%-- ... --%>
    </c:when>
    <c:when test="${customer.type == 'general'}">
        <%-- ... --%>
    </c:when>
    <c:otherwise> <%-- (2) --%>
        <%-- ... --%>
    </c:otherwise>
</c:choose>
```

項番	説明
(1)	<c:when> タグの test 属性に分岐に入る条件を実装する。例では顧客の種別が 'premium' の場合に分岐内の表示処理が実行される。test 属性で指定した条件が false の場合は、次の <c:when> タグの処理が実行される。
(2)	全ての <c:when> タグの test 属性の結果が false の場合、 <c:otherwise> タグ内の表示処理が実行される。

注釈: 詳細は、 JavaServer Pages Standard Tag Library(Version 1.2) の 「CHAPTER 5 Conditional Actions」 を参照されたい。

コレクションの要素に対して表示処理を繰り返す

モデルが保持するコレクションに対して表示処理を繰り返したい場合は、JSTL から提供されている JSP タグライブラリを使用する。

JSTL の JSP タグライブラリから提供されている <c:forEach> を使用して表示処理を繰り返す。

```
<table>
    <tr>
        <th>No</th>
        <th>Name</th>
    </tr>
    <c:forEach var="customer" items="${customers}" varStatus="status"> <%-- (1) --%>
        <tr>
            <td>${status.count}</td> <%-- (2) --%>
            <td>${f:h(customer.name)}</td> <%-- (3) --%>
        </tr>
    </c:forEach>
</table>
```

項番	説明
(1)	<c:forEach> タグの items 属性にコレクションを指定する事で、<c:forEach> タグ内の表示処理が繰り返し実行される。処理対象となっている要素のオブジェクトを参照する場合は、var 属性にオブジェクトを格納するための変数名を指定する。
(2)	<c:forEach> タグの varStatus 属性で指定した変数から現在処理を行っている要素位置 (count) を取得している。count 以外の属性については、 javax.servlet.jsp.jstl.core.LoopTagStatus の JavaDoc を参照されたい。
(3)	<c:forEach> タグの var 属性で指定した変数に格納されているオブジェクトから値を取得している。

注釈: 詳細は、 JavaServer Pages Standard Tag Library(Version 1.2) の 「CHAPTER 6 Iterator Actions」 を参照されたい。

ページネーション用のリンクを表示する

一覧表示を行う画面にてページネーション用のリンクを表示する場合は、共通部品から提供している JSP タグライブラリを使用する。

共通部品から提供している <t:pagination> を使用してページネーション用のリンクを表示する。詳細は、ページネーション を参照されたい。

権限によって表示を切り替える

ログインしているユーザの権限によって表示を切り替える場合は、Spring Security から提供されている JSP タグライブラリを使用する。

Spring Security から提供されている <sec:authorize> を使用して表示の切り替えを行う。詳細は、 認可 を参照されたい。

JavaScript の実装

画面描画後に画面項目の制御 (表示/非表示、活性/非活性などの制御) を行う必要がある場合は、JavaScript を使用して、項目の制御を行う。

スタイルシートの実装

画面のデザインに関わる属性値の指定は JSP(HTML) に直接指定するのではなく、スタイルシート (css ファイル) に指定することを推奨する。

JSP(HTML) では、項目を一意に特定するための id 属性の指定と項目の分類を示す class 属性の指定を行い、実際の項目の配置や見た目にかかわる属性値の指定はスタイルシート (css ファイル) で指定する。

このような構成にすることで、JSP の実装からデザインに関わる処理を減らすことができる。

同時にちょっとしたデザイン変更であれば、JSP を修正せずにスタイルシート (css ファイル) の修正のみで対応可能となる。

注釈: <form:xxx> タグを使ってフォームを生成した場合、id 属性は自動で設定される。class 属性については、アプリケーション開発者によって指定が必要。

3.4.4 共通処理の実装

Controller の呼び出し前後で行う共通処理の実装

本項でいう共通処理とは、Controller を呼び出し前後に行う必要がある共通的な処理のことを指す。

Servlet Filter の実装

Spring MVC に依存しない共通処理については、Servlet Filter で実装する。

ただし、Controller のハンドラメソッドにマッピングされるリクエストに対してのみ共通処理を行いたい場合は、Servlet Filter ではなく HandlerInterceptor で実装すること。

以下に、Servlet Filter のサンプルを示す。

サンプルコードでは、クライアントの IP アドレスをログ出力するために MDC に値を格納している。

- java

```
public class ClientInfoPutFilter extends OncePerRequestFilter { // (1)

    private static final String ATTRIBUTE_NAME = "X-Forwarded-For";
    protected final void doFilterInternal(HttpServletRequest request,
            HttpServletResponse response, FilterChain filterChain) throws␣
↪ServletException, IOException {
        String remoteIp = request.getHeader(ATTRIBUTE_NAME);
        if (remoteIp == null) {
            remoteIp = request.getRemoteAddr();
        }
        MDC.put(ATTRIBUTE_NAME, remoteIp);
        try {
            filterChain.doFilter(request, response);
        } finally {
            MDC.remove(ATTRIBUTE_NAME);
        }
    }
}
```

- web.xml

```
<filter> <!-- (2) -->
    <filter-name>clientInfoPutFilter</filter-name>
    <filter-class>x.y.z.ClientInfoPutFilter</filter-class>
</filter>
<filter-mapping> <!-- (3) -->
    <filter-name>clientInfoPutFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

項番	説明
(1)	サンプルでは Spring Framework から提供されている `org.springframework.web.filter.OncePerRequestFilter` の子クラスとして Servlet Filter を作成することで、同一リクエスト内で 1 回だけ実行されることを保証している。
(2)	作成した Servlet Filter を `web.xml` に登録する。
(3)	登録した Servlet Filter を適用する URL のパターンを指定する。

Servlet Filter を Spring Framework の Bean として定義することもできる。

- web.xml

```
<filter>
    <filter-name>clientInfoPutFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</
↪filter-class> <!-- (1) -->
</filter>
<filter-mapping>
    <filter-name>clientInfoPutFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

- applicationContext.xml

```
<bean id="clientInfoPutFilter" class="x.y.z.ClientInfoPutFilter" /> <!-- (2) -
↪->
```

項番	説明
(1)	サンプルでは Spring Framework から提供されている `org.springframework.web.filter.DelegatingFilterProxy` を Servlet Filter のクラスに指定することで、(2) で定義した Servlet Filter に処理が委譲される。
(2)	作成した Servlet Filter のクラスを Bean 定義ファイル (`applicationContext.xml`) に追加する。その際に、id 属性には `web.xml` で指定したフィルター名 (`<filter-name>` タグで指定した値) にすること。

HandlerInterceptor の実装

Spring MVC に依存する共通処理については、 HandlerInterceptor で実装する。

HandlerInterceptor は、リクエストにマッピングされたハンドラメソッドが決定した後に呼び出されるので、アプリケーションが許可しているリクエストに対してのみ共通処理を行うことができる。

HandlerInterceptor では以下の 3 つのポイントで処理を実行することが出来る。

- Controller のハンドラメソッドを実行する前

  `HandlerInterceptor#preHandle` メソッドとして実装する。

- Controller のハンドラメソッドが正常終了した後

  `HandlerInterceptor#postHandle` メソッドとして実装する。

- Controller のハンドラメソッドの処理が完了した後 (正常/異常に関係なく実行される)

  `HandlerInterceptor#afterCompletion` メソッドとして実装する。

以下に、HandlerInterceptor のサンプルを示す。

サンプルコードでは、Controller の処理が正常終了した後に info レベルのログを出力している。

```
public class SuccessLoggingInterceptor extends HandlerInterceptorAdapter { //
↪(1)

    private static final Logger logger = LoggerFactory
            .getLogger(SuccessLoggingInterceptor.class);

    @Override
    public void postHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler,
            ModelAndView modelAndView) throws Exception {
        HandlerMethod handlerMethod = (HandlerMethod) handler;
        Method m = handlerMethod.getMethod();
        logger.info("[SUCCESS CONTROLLER] {}.{}", new Object[] {
                m.getDeclaringClass().getSimpleName(), m.getName()});
    }

}
```

- spring-mvc.xml

```
<mvc:interceptors>
    <!-- ... -->
    <mvc:interceptor>
        <mvc:mapping path="/**" /> <!-- (2) -->
        <mvc:exclude-mapping path="/resources/**" /> <!-- (3) -->
        <bean class="x.y.z.SuccessLoggingInterceptor" /> <!-- (4) -->
    </mvc:interceptor>
    <!-- ... -->
</mvc:interceptors>
```

項番	説明
(1)	サンプルでは Spring Framework から提供されている `org.springframework.web.servlet.handler.HandlerInterceptorAdapter` の子クラスとして HandlerInterceptor を作成している。`HandlerInterceptorAdapter` は `HandlerInterceptor` インタフェースの空実装を提供しているため、子クラスで不要なメソッドの実装をしないで済む。
(2)	作成した HandlerInterceptor を適用するパスのパターンを指定する。
(3)	作成した HandlerInterceptor を適用しないパスのパターンを指定する。
(4)	作成した HandlerInterceptor を `spring-mvc.xml` の `<mvc:interceptors>` タグ内に追加する。

Controller の共通処理の実装

ここでいう共通処理とは、すべての Controller で共通的に実装する必要がある処理のことを指す。

HandlerMethodArgumentResolver の実装

Spring Framework のデフォルトでサポートされていないオブジェクトを Controller の引数として渡したい場合は、HandlerMethodArgumentResolver を実装して Controller の引数として受け取れるようにする。

以下に、HandlerMethodArgumentResolver のサンプルを示す。

サンプルコードでは、 共通的なリクエストパラメータを JavaBean に変換して Controller のメソッドで受け取れるようにしている。

- JavaBean

```
public class CommonParameters implements Serializable { // (1)

    private String param1;
    private String param2;
    private String param3;

    // ....
```

```
}
```

- HandlerMethodArgumentResolver

```
public class CommonParametersMethodArgumentResolver implements
↪HandlerMethodArgumentResolver { // (2)

    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        return CommonParameters.class.equals(parameter.getParameterType()); //
 (3)
    }

    @Override
    public Object resolveArgument(MethodParameter parameter,
            ModelAndViewContainer mavContainer, NativeWebRequest webRequest,
            WebDataBinderFactory binderFactory) throws Exception {
        CommonParameters params = new CommonParameters(); // (4)
        params.setParam1(webRequest.getParameter("param1"));
        params.setParam2(webRequest.getParameter("param2"));
        params.setParam3(webRequest.getParameter("param3"));
        return params;
    }
```

- Controller

```
@RequestMapping(value = "home")
public String home(CommonParameters commonParams) { // (5)
    logger.debug("param1 : {}",commonParams.getParam1());
    logger.debug("param2 : {}",commonParams.getParam2());
    logger.debug("param3 : {}",commonParams.getParam3());
    // ...
    return "sample/home";

}
```

- spring-mvc.xml

```
<mvc:annotation-driven>
    <mvc:argument-resolvers>
        <!-- ... -->
        <bean class="x.y.z.CommonParametersMethodArgumentResolver" /> <!--
↪(6) -->
        <!-- ... -->
    </mvc:argument-resolvers>
</mvc:annotation-driven>
```

項番	説明
(1)	共通パラメータを保持する JavaBean。
(2)	`org.springframework.web.method.support.HandlerMethodArgumentResolver` インタフェースを実装する。
(3)	処理対象とする型を判定する。例では、共通パラメータを保持する JavaBean の型が Controller の引数として指定されていた場合に、このクラスの resolveArgument メソッドが呼び出される。
(4)	リクエストパラメータから値を取得し、共通パラメータを保持する JavaBean に設定し返却する。
(5)	Controller のハンドラメソッドの引数に共通パラメータを保持する JavaBean を指定する。 (4) で返却されるオブジェクトが渡される。
(6)	作成した HandlerMethodArgumentResolver を `spring-mvc.xml` の `<mvc:argument-resolvers>` タグ内に追加する。

注釈: 全ての Controller のハンドラメソッドで共通的に渡すパラメータがある場合は、HandlerMethodArgumentResolver を使って JavaBean に変換してから渡す方法が有効的である。ここでいうパラメータとは、リクエストパラメータに限らない。

`@ControllerAdvice` の実装

`@ControllerAdvice` アノテーションを付与したクラスでは、複数の Controller で実行したい共通的な処理を実装する。

`@ControllerAdvice` アノテーションを付与したクラスを作成すると、

- `@InitBinder` を付与したメソッド
- `@ExceptionHandler` を付与したメソッド

- `@ModelAttribute` を付与したメソッド

で実装した処理を、複数の Controller に適用する事ができる。

ちなみに: `@ControllerAdvice` アノテーションは、Spring Framework 3.2 から追加された仕組みだが、全ての Controller に処理が適用される仕組みになっていたため、アプリケーション全体の共通処理しか実装できなかった。

Spring Framework 4.0 からは、共通処理を適用する Controller を柔軟に指定する事ができるように改善されている。この改善により、様々な粒度で共通処理を実装する事ができるようになった。

以下に、共通処理を適用する Controller を指定する方法 (属性の指定方法) について説明する。

項番	属性	説明と指定例

(1)	annotations	アノテーションを指定する。
		指定したアノテーションが付与された Controller に対して共通処理が適用される。
		以下に指定例を示す。
		`@ControllerAdvice(annotations = LoginFormModelAttributeSetter.LoginFormModelAttribute.class)`
		`public class LoginFormModelAttributeSetter {`
		`  @Target(TYPE)`
		`  @Retention(RUNTIME)`
		`  public static @interface LoginFormModelAttribute {}`
		`  // ...`
		`}`
		`@LoginFormModelAttribute`
		`@Controller`
		`public class WelcomeController {`
		`  // ...`
		`}`
		`@LoginFormModelAttribute`
		`@Controller`
		`public class LoginController {`
		`  // ...`
		`}`
		上記例では、WelcomeController と LoginController に@LoginFormModelAttribute アノテーションを付与しているため、WelcomeController と LoginController に共通処理が適用される。

Continued on next page

表 3.15 – continued from previous page

項番	属性	説明と指定例

(2)	assignableTypes	クラス又はインタフェースを指定する。
		指定したクラス又はインタフェースに割り当て可能 (キャスト可能) な Controller に対して共通処理が適用される。本属性を使用する場合は、共通処理を適用する Controller であることを示すためのマーカーインタフェースを属性値に指定するスタイルを採用することを推奨する。このスタイルを採用した場合、Controller 側では、適用したい共通処理用のマーカーインタフェースを実装するだけでよい。以下の指定例を示す。
		`@ControllerAdvice(assignableTypes = ISODateInitBinder.class)`
		`public class ISODateInitBinder {`
		`  public static interface ISODateApplicable {}`
		`  // ...`
		`}`
		`@Controller`
		`public class SampleController implements ISODateApplicable {`
		`  // ...`
		`}`
		上記例では、SampleController が@ISODateApplicable インタフェース (マーカーインタフェース) を実装しているため、SampleController に共通処理が適用される。

Continued on next page

表 3.15 – continued from previous page

項番	属性	説明と指定例

(3)	basePackageClasses	クラス又はインタフェースを指定する。
		指定したクラス又はインタフェースのパッケージ配下の Controller に対して共通処理が適用される。
		本属性を使用する場合は、
		• @ControllerAdvice を付与したクラス
		• パッケージを識別するためのマーカーインタフェース
		を属性値に指定するスタイルを採用することを推奨する。以下に指定例を示す。
		`package com.example.app`
		`@ControllerAdvice(basePackageClasses = AppGlobalExceptionHandler.class)`
		`public class AppGlobalExceptionHandler { // ... }`
		`package com.example.app.sample`
		`@Controller`
		`public class SampleController { // ... }`
		上記例では、SampleController が@ControllerAdvice を付与したクラス (AppGlobalExceptionHandler) が格納されているパッケージ (com.example.app) 配下に格納されているため、SampleController に共通処理が適用される。
		`package com.example.app.common`
		`@ControllerAdvice(basePackageClasses = AppPackage.class)`
		`public class AppGlobalExceptionHandler { // ... }`
		`package com.example.app`
		`public interface AppPackage { }`
		@ControllerAdvice が付与されているクラスと Controller が格納されているクラスのパッケージ階層が異なる場合や、複数のベースパッケージに共通処理を適用したい場合は、パッケージを識別するためのマーカインタフェースを用意すればよい。

Continued on next page

表 3.15 – continued from previous page

項番	属性	説明と指定例

(4)	basePackages	パッケージ名を指定する。
		指定したパッケージ配下の Controller に対して共通処理が適用される。以下に指定例を示す。
		`@ControllerAdvice(basePackages = "com.example.app")`
		`public class AppGlobalExceptionHandler {`
		`    // ...`
		`}`

(5)	value	basePackages へのエイリアス。
		basePackages 属性を指定した際と同じ動作となる。以下に指定例を示す。
		`@ControllerAdvice("com.example.app")`
		`public class AppGlobalExceptionHandler {`
		`    // ...`
		`}`

ちなみに: basePackageClasses 属性 / basePackages 属性 / value 属性は、共通処理を適用したい Controller が格納されているベースパッケージを指定するための属性であるが、basePackageClasses 属性を使用した場合、

- 存在しないパッケージを指定してしまう事を防ぐことが出来る
- IDE 上で行ったパッケージ名変更と連動することが出来る

ため、タイプセーフな指定方法と言える。

以下に、@InitBinder メソッドの実装サンプルを示す。
サンプルコードでは、 リクエストパラメータで指定できる日付型で形式を yyyy/MM/dd に設定している。

```
@ControllerAdvice // (1)
@Order(0) // (2)
public class SampleControllerAdvice {

    // (3)
    @InitBinder
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class,
                new CustomDateEditor(dateFormat, true));
    }

}
```

項番	説明
(1)	@ControllerAdvice アノテーションを付与することで、ControllerAdvice の Bean であることを示している。
(2)	@Order アノテーションを付与することで、共通処理が適用される優先度を指定する。複数の ControllerAdvice に依存関係があるなど、ControllerAdvice に順序性を持たせたい場合は必ず指定すること。順序性を持たせる必要がなければ指定しなくてもよい。
(3)	@InitBinder メソッドを実装する。全ての Controller に対して@InitBinder メソッドが適用される。

以下に、@ExceptionHandler メソッドの実装サンプルを示す。

サンプルコードでは、 org.springframework.dao.PessimisticLockingFailureException をハンドリングしてロックエラー画面の View を返却している。

```
// (1)
@ExceptionHandler(PessimisticLockingFailureException.class)
public String handlePessimisticLockingFailureException(
        PessimisticLockingFailureException e) {
    return "error/lockError";
}
```

項番	説明
(1)	@ExceptionHandler メソッドを実装する。全ての Controller に対して@ExceptionHandler メソッドが適用される。

以下に、@ModelAttribute メソッドの実装サンプルを示す。

サンプルコードでは、 共通的なリクエストパラメータを JavaBean に変換して Model に格納している。

- ControllerAdvice

```
// (1)
@ModelAttribute
public CommonParameters setUpCommonParameters(
        @RequestParam(value = "param1", defaultValue="def1") String param1,
        @RequestParam(value = "param2", defaultValue="def2") String param2,
        @RequestParam(value = "param3", defaultValue="def3") String param3) {
    CommonParameters params = new CommonParameters();
    params.setParam1(param1);
    params.setParam2(param2);
    params.setParam3(param3);
    return params;
}
```

- Controller

```
@RequestMapping(value = "home")
public String home(@ModelAttribute CommonParameters commonParams) { // (2)
    logger.debug("param1 : {}",commonParams.getParam1());
    logger.debug("param2 : {}",commonParams.getParam2());
    logger.debug("param3 : {}",commonParams.getParam3());
    // ...
    return "sample/home";
}
```

項番	説明
(1)	`@ModelAttribute` メソッドを実装する。全ての Controller に対して`@ModelAttribute` メソッドが適用される。
(2)	`@ModelAttribute` メソッドで生成されたオブジェクトが渡る。

3.4.5 二重送信防止について

送信ボタンの複数回押下や完了画面の再読み込み (F5 ボタンによる再読み込み) などで、 同じ処理が複数回実行されてしまう可能性があるため、二重送信を防止するための対策は必ず行うこと。

対策を行わない場合に発生する問題点や対策方法の詳細は、 二重送信防止 を参照されたい。

3.4.6 セッションの使用について

Spring MVC のデフォルトの動作では、モデル（フォームオブジェクトやドメインオブジェクトなど）はセッションには格納されない。

セッションに格納したい場合は、`@SessionAttributes` アノテーションを Controller クラスに付与する必要がある。

入力フォームが複数の画面にわかれている場合は、 一連の画面遷移を行うリクエストでモデル（フォームオブジェクトやドメインオブジェクトなど）を共有できるため、 `@SessionAttributes` アノテーションの利用を検討すること。

ただし、セッションを使用する際の注意点があるので、そちらを確認した上で`@SessionAttributes` アノテーションの利用有無を判断すること。

セッションの利用指針及びセッション使用時の実装方法の詳細は、 セッション管理 を参照されたい。

3.5 開発プロジェクトのビルド

3.5.1 開発プロジェクトのビルド

アプリケーションサーバにデプロイするための war ファイル、env モジュール (ファイル環境依存ファイルを格納するモジュール) の jar ファイルを作成する方法を紹介する。

Maven Archetype で作成したプロジェクトでは、war ファイルを作成する方法として以下の 2 つの方法を提供している。

- *env モジュールの jar ファイルを war ファイルに含めないビルド方法* (推奨)
- *env モジュールの jar ファイルを war ファイルに含めるビルド方法*

注釈: 推奨するビルド方法について

本ガイドラインでは、*env モジュールの jar ファイルを war ファイルに含めないビルド方法* を推奨している。なお、ここで紹介するビルド方法は選択肢の一つであり、他のビルド方法を採用してもよい。

ただし、**試験環境や商用環境にリリースする war ファイルと jar ファイルは、Eclipse などの IDE が提供している機能を使って作成しないようにすること。**Eclipse などの一部の IDE では、開発用に最適化された独自のコンパイラを使ってクラスファイルを作成しており、コンパイラの違いが原因でアプリケーション実行時に予期しないエラーが発生するリスクが生まれる。

注釈: プロジェクト構成の原則

原則として下記のようなプロジェクト構造とする。

1. 必ずマルチプロジェクト構成にする。
2. 一つのプロジェクトに環境依存性のある設定ファイル（ex. logback.xml, jdbc.properties）をできるだけ集約する。 **以降、このプロジェクトを *-env と表現する。**
   - ex. terasoluna-tourreservation-env
3. *-env 以外のプロジェクトには環境依存性のある設定値を一切持たせない。
   - もちろん、src/test/resources 配下などにテスト用の環境依存性設定ファイルを格納しておくことは許可される。
4. 全てのソフトウェアのパッケージ済みバイナリをパッケージリポジトリ上に保管して管理する。
   - *.jar ファイルだけではなく*.war ファイルも成果物としてパッケージリポジトリにデプロイする。したがって、それらの jar/war ファイルには環境依存性が含まれていてはならない。
5. *-env プロジェクトは下記のような構造にする。
   - 開発者の PC 上での作業に必要な設定値をデフォルトとして src/main/resources 配下のファイルに格納する。

- 試験サーバ、本番サーバ等、環境毎に異なる設定ファイルを src/main/resources 以外 (ex. configs/test-server) のフォルダに格納し、maven の profile 機能を使って環境毎に自動的に設定値を差し替えながら *-env-x.y.z.jar ファイルをビルドする。

上記のような構造を取ることにより、ソフトウェアライフサイクルの全ての場面において、適切に開発をすることができるようになる。

1. ローカル開発環境では、プロジェクト本体と*-env プロジェクトの両方をチェックアウトし、env プロジェクトを本体プロジェクトのビルドパスに含めることによって、ローカル開発環境でのコーディングとテストを可能にする。
2. CI サーバ上ではビルドツール (maven) によるテストの実行とパッケージングを行い、必要に応じてパッケージリポジトリに artifact を deploy する。
3. 試験サーバ、本番サーバでは、パッケージリポジトリにあらかじめ保管しているプロジェクト本体に、リリース先環境にあわせてビルドした*-env プロジェクトを追加してリリースすることにより、アプリケーションの動作が可能になる。

詳細についてはサンプルアプリケーションを参考にされたい。

警告: ビルド環境について

ここでは Windows 環境でビルドする例になっているが、Windows 環境でビルドすることを推奨しているわけではない。本ガイドラインでは、**アプリケーションの実行環境と同じ OS と JDK のバージョンを使ってビルドすることを推奨する。**

Maven を使ってビルドする場合は、環境変数「JAVA_HOME」にコンパイル時に使用する JDK のホームディレクトリが指定されていることを確認されたい。

環境変数が設定されていない場合や異なるバージョンの JDK のホームディレクトリが指定されている場合は、環境変数に適切なホームディレクトリを指定すること。

[Windows の場合]

```
echo %JAVA_HOME%
set JAVA_HOME={Please set home directory of JDK}
```

[Linux 系の場合]

```
echo $JAVA_HOME
JAVA_HOME={Please set home directory of JDK}
```

注釈: 環境変数「JAVA_HOME」は、ビルドを実行する OS ユーザーのユーザー環境変数に設定しておくとよい。

---

env モジュールの jar ファイルを war ファイルに含めないビルド方法

war ファイルの作成

開発プロジェクトのルートディレクトリへ移動する。

```
cd C:\work\todo
```

Maven のプロファイル (`-P` パラメータ) に `warpack` を指定して、Maven install を実行する。

```
mvn -P warpack clean install
```

Maven package の実行が成功すると、web モジュールの target ディレクトリの中に、env モジュールの jar ファイルが含まれていない war ファイルが作成される。

(例：`C:\work\todo\todo-web\target\todo-web.war`)

注釈: 指定するゴールについて

上記例ではゴールに `install` を指定して war ファイルをローカルリポジトリへインストールしているが、

- war ファイルの作成のみ行う場合はゴールに `package`
- Nexus などのリモートリポジトリへデプロイする場合はゴールに `deploy`

を指定すればよい。

---

env モジュールの jar ファイルの作成

env モジュールのディレクトリへ移動する。

```
cd C:\work\todo\todo-env
```

Maven のプロファイル (-P パラメータ) に**環境を識別するプロファイル ID** を指定して、Maven package を実行する。

```
mvn -P test-server clean package
```

Maven package の実行が成功すると、env モジュールの target ディレクトリの中に、指定した環境用の jar ファイルが作成される。

(例：C:\work\todo\todo-env\target\todo-env-1.0.0-SNAPSHOT-test-server.jar)

注釈: 環境を識別するプロファイル ID について

Maven Archetype で作成したプロジェクトでは、以下のプロファイル ID がデフォルトで定義されている。

- local : 開発者のローカル環境向け (IDE 開発環境向け) のプロファイル (デフォルトのプロファイル)
- test-server : 試験環境向けのプロファイル
- production-server : 商用環境向けのプロファイル

デフォルトで用意しているプロファイルは上記の 3 つだが、開発するシステムの環境構成にあわせて追加及び修正されたい。

---

env モジュールの jar ファイルを war ファイルに含めるビルド方法

war ファイルの作成

警告: env モジュールの jar ファイルを war ファイルに含める場合の注意点

env モジュールの jar ファイルを war ファイルに含めた場合、war ファイルを他の環境にデプロイすることができないため、間違って他の環境 (特に商用環境) にデプロイされないように war ファイルを管理すること。

また、環境毎に war ファイルを作成して各環境へリリースする方法を採用した場合、商用環境へリリースされる war ファイルが厳密にいうとテスト済みの war ファイルではないという点を意識してほしい。これは、商用環境用の war ファイルを作成する際にコンパイルをしなおすためである。war ファイルを環境毎に作成してリリースする場合は、Git や Subversion などの VCS(Version Control System) の機能 (タグ機能など) を活用し、テスト済みのソースファイルを使用して商用環境や各種テスト環境へリリースする war ファイルを作成する仕組みを確立することが特に重要である。

開発プロジェクトのルートディレクトリへ移動する。

```
cd C:\work\todo
```

Maven のプロファイル (`-P` パラメータ) に `warpack-with-env` と env モジュールの中で定義している**環境を識別するプロファイル ID** を指定して、Maven package を実行する。

```
mvn -P warpack-with-env,test-server clean package
```

Maven package の実行が成功すると、web モジュールの target ディレクトリの中に、env モジュールの jar ファイルを含んだ war ファイルが作成される。

(例：`C:\work\todo\todo-web\target\todo-web.war`)

デプロイ

Tomcat へのデプロイ

Web アプリケーションを Tomcat 8.5 および Tomcat 9 上にリリースする場合は次のような手順をとる。

1. リリース対象の AP サーバ環境にあわせて maven の profile を指定し、 *-env プロジェクトを ビルドする。
2. 上記でビルドした*-env-x.y.z.jar ファイル をあらかじめ決定した AP サーバ上のフォルダに設置する。ex. /etc/foo/bar/abcd-env-x.y.z.jar
3. あらかじめパッケージリポジトリにデプロイ済みの*.war ファイルを [CATALINA_HOME]/webapps 配下で解凍 (unjar) する。
4. Tomcat のリソース機能を使用して、 /etc/foo/bar/*.jar をクラスパスに追加する。
   - [CATALINA_HOME]/conf/[contextPath].xml ファイルに下記の定義を追加する。
   - 詳しくは、 The Resources Component と terasoluna-tourreservation-env の configs フォルダを参考されたい。
   - リソースの設定例：

```
<Resources className="org.apache.catalina.webresources.StandardRoot">
  <PreResources className="org.apache.catalina.webresources.DirResourceSet"
                base="/etc/foo/bar/"
                internalPath="/"
                webAppMount="/WEB-INF/lib" />
</Resources>
```

注釈:

- [CATALINA_HOME]/conf/server.xml の Host タグ上の autoDeploy 属性を false にセットしておかなければならない。さもないと web アプリケーションの再起動のたびに [CATALINA_HOME]/conf/[contextPath].xml が自動的に削除されてしまう。
- autoDeploy を無効化している場合、[CATALINA_HOME]/webapps に war ファイルを置くだけでは Web アプリケーションは起動しない。必ず war ファイルを unjar(unzip) すること。

注釈: **Tomcat 7 および Tomcat 6 を使用する場合**

Tomcat 7 および Tomcat 6 を使用する場合は、上記手順 4. の代わりに Tomcat の VirtualWebappLoader 機能を使用して /etc/foo/bar/*.jar をクラスパスに追加する。

- [CATALINA_HOME]/conf/[contextPath].xml ファイルに下記の定義を追加する。
- 詳しくは、 VirtualWebappLoader と terasoluna-tourreservation-env の configs フォルダを参考されたい。

VirtualWebappLoader の設定例：

```
<Loader className="org.apache.catalina.loader.VirtualWebappLoader"
        virtualClasspath="/etc/foo/bar/*.jar" />
```

Tomcat 以外のアプリケーションサーバへのデプロイ

アプリケーションサーバとして Tomcat 以外のサーバを使用する際のデプロイ方法 (手順) を紹介する。

Tomcat の VirtualWebappLoader のように、Web アプリケーションごとにクラスパスを追加する手段が提供されていないアプリケーションサーバ（例： WebSphere,WebLogic,JBoss）にリリースする場合には、*-env-x.y.z.jar ファイルを war ファイル内の WEB-INF/lib 配下に追加してからリリースする方法が最も簡単である。

1. リリース対象の AP サーバ環境にあわせて maven の profile を指定し、 *-env プロジェクトを ビルドする。
2. あらかじめパッケージリポジトリにデプロイ済みの*.war ファイルを 作業ディレクトリにコピーする。
3. 下のように、ｊａｒコマンドの追加オプションを利用して、war ファイル内の WEB-INF/lib の配下に追加する。
4. foo-x.y.z.war を AP サーバにリリースする。

注釈: war ファイルをアプリケーションサーバへデプロイする方法は、使用するアプリケーションサーバのマニュアルを参照されたい。

ここでは、jar コマンドを使用して、env モジュールの jar ファイルを war ファイルに組み込む方法 (手順) を紹介する。

作業ディレクトリへ移動する。
ここでは、env プロジェクトで作業を行う例になっている。

```
cd C:\work\todo\todo-env
```

作成した war ファイルを作業ディレクトリへコピーする。

ここでは、Maven リポジトリから war ファイルを取得する例になっている。(war ファイルを `install` または `deploy` している前提とする)

```
mvn org.apache.maven.plugins:maven-dependency-plugin:2.5:get^
 -DgroupId=com.example.todo^
 -DartifactId=todo-web^
 -Dversion=1.0.0-SNAPSHOT^
 -Dpackaging=war^
 -Ddest=target/todo-web.war
```

コマンドの実行が成功すると、env モジュールの target ディレクトリの中に、指定した war ファイルがコピーされる。

(例：`C:\work\todo\todo-env\target\todo-web.war`)

注釈:

- `-DgroupId`、`-DartifactId`、`-Dversion`、`-Ddest` には、適切な値を指定すること。
- Linux 系で実行する場合は、行末の 「^」 を 「\」 に読み替えること。

作成した jar ファイルを作業ディレクト (`target\WEB-INF\lib`) へ一旦コピーし、war ファイルの中に追加する。

[Windows の場合]

```
mkdir target\WEB-INF\lib
copy target\todo-env-1.0.0-SNAPSHOT-test-server.jar target\WEB-INF\lib\.
cd target
jar -uvf todo-web.war WEB-INF\lib
```

[Linux 系の場合]

```
mkdir -p target/WEB-INF/lib
cp target/todo-env-1.0.0-SNAPSHOT-test-server.jar target/WEB-INF/lib/.
cd target
jar -uvf todo-web.war WEB-INF/lib
```

注釈: **jar コマンドが見つからない場合の対処**

jar コマンドが見つからない場合は、以下のいずれかの対処を行うことで解決することができる。

- `JAVA_HOME/bin` を環境変数「PATH」に追加する。
- jar コマンドをフルパスで指定する。Window の場合は `%JAVA_HOME%\bin\jar`、Linux 系の場合は`${JAVA_HOME}/bin/jar` を指定すればよい。

継続的なデプロイ

プロジェクト（ソースコードツリー）の構造、バージョン管理、インスペクションとビルド作業、ライフサイクル管理の工程を恒常的にループさせることによって目的のソフトウェアをリリースし続けることが、継続的デプロイメントである。

開発の途中では、SNAPSHOT バージョンのソフトウェアをパッケージリポジトリや開発用 AP サーバにリリースし、テストを実施する。ソフトウェアを正式にリリースする場合には、バージョン番号を固定したうえで VCS のソースコードツリーに対してタグづけを行う必要がある。このように、スナップショットリリースの場合と正式リリースの場合で、ビルドとデプロイのフローが少し異なる。

また、Web サービスを提供する AP サーバにアプリケーションをデプロイする場合には、スナップショットバージョンか正式リリースバージョンかに関わらず、デプロイ先の AP サーバ環境に合わせた環境依存性設定ファイル群と*.war ファイルをセットでデプロイする必要がある。

そこで、環境依存性設定を持たない状態のライブラリ (jar,war) を maven リポジトリに登録する作業と、それらを実際に AP サーバにデプロイする作業を分離することによって、デプロイ作業を簡潔に実施可能にする。

注釈: maven の世界では、pom.xml 上の<version>タグの内容によってそれが SNAPSHOT バージョンなのか RELEASE バージョンなのかが自動的に判別される。

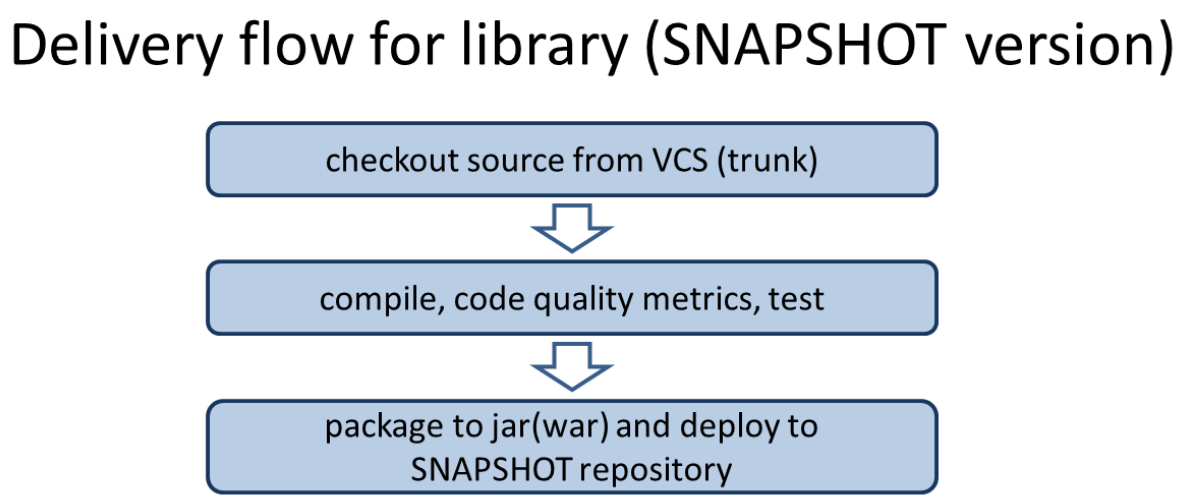
- 末尾が -SNAPSHOT である場合に SNAPSHOT とみなされる。例:<version>1.0-SNAPSHOT</version>
- 末尾が -SNAPSHOT ではない場合は RELEASE とみなされる。例 : <version>1.0</version>

また、maven パッケージリポジトリには snapshots リポジトリと release リポジトリの 2 種類があり、いくつかの制約があることに注意する。

- SNAPSHOT バージョンのソフトウェアを release リポジトリに登録することはできない。その逆も不可能。
- release リポジトリには、同一の GAV 情報を持つ artifact は 1 回しか登録できない。（GAV=groupId,artifactId,version）
- snapshot リポジトリには、同一の GAV 情報を持つ artifact を何度でも登録しなおすことができる。

SNAPSHOT バージョンの運用

SNAPSHOT バージョンのソフトウェアのデリバリーフローは下図のように簡潔である。



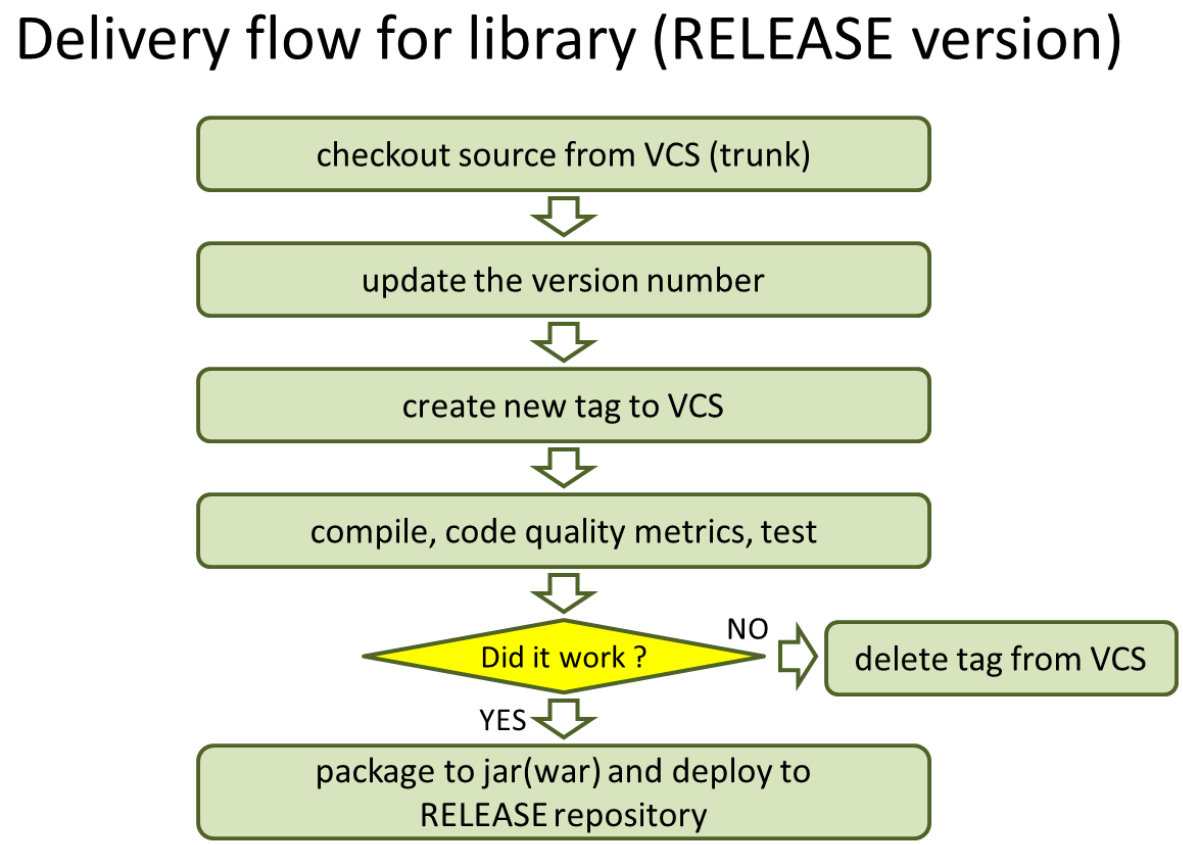
1. 開発用 trunk からソースコードをチェックアウトする。
2. コンパイル、コードメトリクスの測定、テストを実行する。
   - コンパイルエラー、コードメトリクスでの一定以上の violation の発生、テストの失敗の場合、以降の作業を中止する。
3. maven パッケージリポジトリサーバに artifact(jar,war ファイル) をアップロード (mvn deploy) する。

RELEASE バージョンの運用

正式なリリースの場合、バージョン番号の付与作業が必要なため、SNAPSHOT リリースよりもやや複雑なフローとなる。

1. リリースに与えるバージョン番号を決定する。（例：1.0.1）
2. 開発用 trunk(またはリリース用 branch) からソースコードをチェックアウトする。
3. pom.xml 上の<version>タグを変更する。（例：<version>1.0.1</version>）
4. VCS 上に tag を付与する。（例： tags/1.0.1）
5. コンパイル、コードメトリクスの測定、テストを実行する。
   - コンパイルエラー、コードメトリクスでの一定以上の violation の発生、テストの失敗の場合、以降の作業を中止する。
   - 失敗した場合は VCS 上の tag を削除する。
6. maven パッケージリポジトリサーバに artifact(jar,war ファイル) をアップロード (mvn deploy) する。

注釈: pom.xml ファイルの<version>タグの変更は versions-maven-plugin で可能である。




```
mvn versions:set -DnewVersion=1.0.0
```

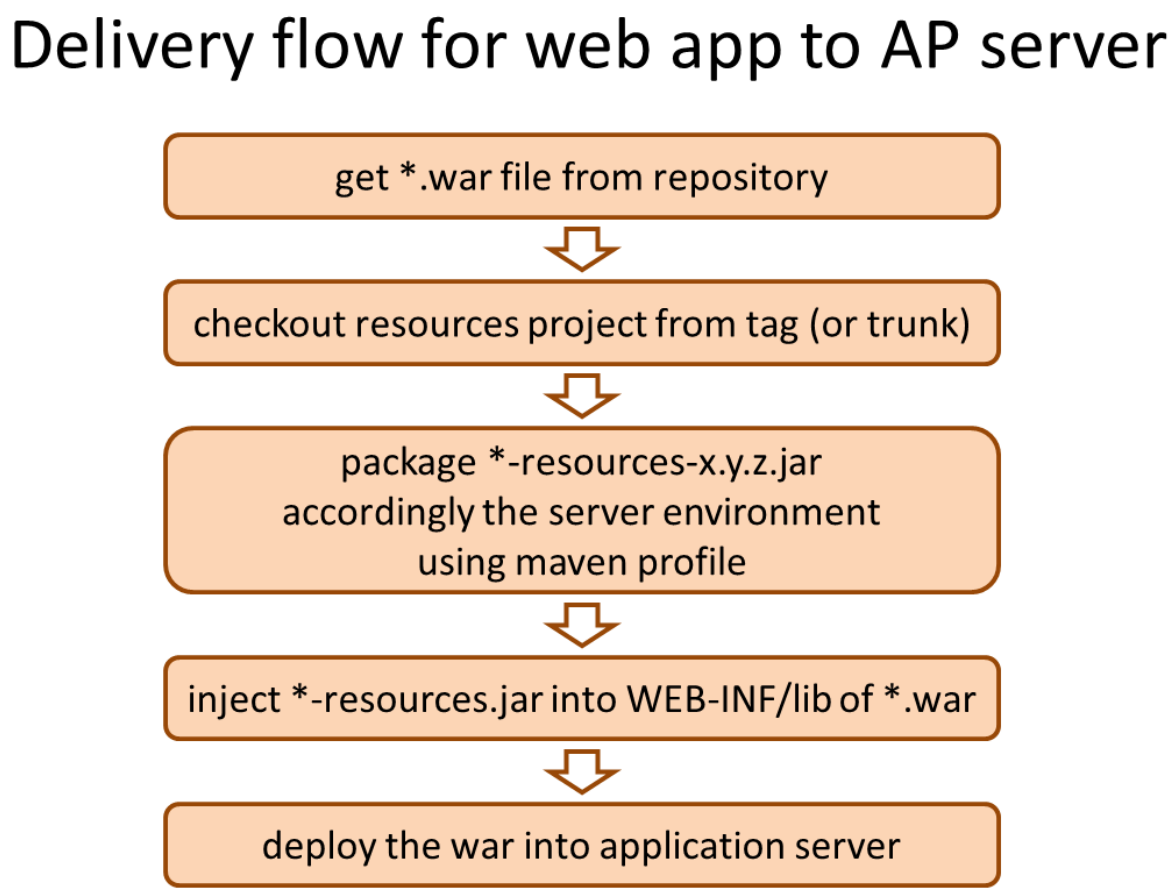
上記のようなコマンドで、pom.xml 内の version タグを<version>1.0.0</version>のように編集することができる。

アプリケーションサーバへのリリース

Web サービスを提供する AP サーバにアプリケーションをリリースする場合、あらかじめ maven パッケージリポジトリに登録済みの war ファイルと、リリース先の AP サーバ環境に合わせた環境依存性設定ファイル群とを、セットでリリースする。これはスナップショットリリースか正式リリースかに関わらず同じフローとなる。

1. リリース対象バージョンの war ファイルを maven パッケージリポジトリからダウンロードする
2. *-resources プロジェクト（環境依存性設定ファイルを集約しているプロジェクト）を VCS からチェックアウトする
3. maven の profile を機能によって、リリース先の環境に合わせた設定ファイル群で内容を差し替えて resources プロジェクトをパッケージし、*-resources-x.y.z.jar を生成する。
4. 生成した*-resources-x.y.z.jar ファイルを、war ファイル内の WEB-INF/lib フォルダ配下に追加する。

- Tomcat の場合は、*-resources-x.y.z.jar を war ファイル内部に追加するのではなく、Tomcat サーバ上の任意のパスにコピーし、そのパスを VirtualWebappLoader の拡張クラスパスに指定する。



5. war ファイルをアプリケーションサーバにデプロイする。

注釈: maven パッケージリポジトリからの war ファイルのダウンロードは、maven-dependency-plugin の get ゴールで可能である。

```
mvn org.apache.maven.plugins:maven-dependency-plugin:2.5:get^
 -DgroupId=com.example^
 -DartifactId=mywebapp^
 -Dversion=0.0.1-SNAPSHOT^
 -Dpackaging=war^
 -Ddest=${WORKSPACE}/target/mywebapp.war
```

これで、target というディレクトリ配下に mywebapp.war ファイルがダウンロードされる。

さらに、下記のようなコマンドで環境依存設定ファイルのパッケージを mywebapp.war ファイル内に追加することができる。

```
mkdir -p $WORKSPACE/target/WEB-INF/lib
cd $WORKSPACE/target
cp ./mywebapp-resources*.jar WEB-INF/lib
jar -ufv mywebapp.war WEB-INF/lib
```

レイヤ定義については、 アプリケーションのレイヤ化 を参照。

第 4 章

Web アプリ開発機能

本ガイドラインで想定している Web 機能詳細アーキテクチャについて説明する。

4.1 入力チェック

4.1.1 Overview

ユーザーが入力した値が不正かどうかを検証することは必須である。入力値の検証は大きく分けて、

1. 長さや形式など、文脈によらず入力値だけを見て、それが妥当かどうかを判定できる検証
2. システムの状態によって入力値が妥当かどうかが変わる検証

がある。

1. の例としては必須チェックや、桁数チェックがあり、2. の例としては登録済みの E-mail かどうかのチェックや、注文数が在庫数以内であるかどうかのチェックが挙げられる。

本節では、基本的には前者のことを説明し、このチェックのことを「入力チェック」を呼ぶ。後者のチェックは「業務ロジックチェック」と呼ぶ。業務ロジックチェックについてはドメイン層の実装を参照されたい。

本ガイドラインでは、基本的に入力チェックをアプリケーション層で行い、業務ロジックチェックは、ドメイン層で行うことをポリシーとする。

Web アプリケーションの入力チェックには、サーバサイドで行うチェックと、クライアントサイド (JavaScript) で行うチェックがある。サーバーサイドのチェックは必須であるが、クライアントサイドでも同じチェックを実施すると、サーバー通信なしでチェック結果が分かるため、ユーザビリティが向上する。

> **警告:** JavaScript によるクライアントサイドの処理は、改ざん可能であるため、サーバーサイドのチェックは、必ず行うこと。クライアントサイドのみでチェックを行い、サーバーサイドでチェックを省略した場合は、システムが危険な状態に晒されていることになる。

入力チェックの分類

入力チェックは、単項目チェック、相関項目チェックに分類される。

種類	説明	例	実現方法
単項目チェック	単一のフィールドで完結するチェック	入力必須チェック 桁チェック 型チェック	Bean Validation (実装ライブラリとして Hibernate Validator を使用)
相関項目チェック	複数のフィールドを比較するチェック	パスワードと確認用パスワードの一致チェック	org.springframework.validation.Validator インタフェースを実装した Validation クラス または Bean Validation

Spring は、Java 標準である Bean Validation をサポートしている。単項目チェックには、この Bean Validation を利用する。相関項目チェックの場合は、Bean Validation または Spring が提供している `org.springframework.validation.Validator` インタフェースを利用する。

4.1.2 How to use

依存ライブラリの追加

Bean Validation 1.1(Hibernate Validator 5.x) 以上を使用する場合、Bean Validation の API 仕様クラス (`javax.validation` パッケージのクラス) が格納されている jar ファイルと Hibernate Validator の jar ファイルに加えて、

- Expression Language 2.2 以上の API 仕様クラス (`javax.el` パッケージのクラス)
- Expression Language 2.2 以上のリファレンス実装クラス

が格納されているライブラリが必要となる。

アプリケーションサーバにデプロイして動かす場合は、これらのライブラリはアプリケーションサーバから提供されているため、依存ライブラリの追加は不要である。ただし、スタンドアロン環境 (JUnit など) で動かす場合は、これらのライブラリを依存ライブラリとして追加する必要がある。

スタンドアロン環境で Bean Validation 1.1 以上を動かす際に必要となるライブラリの追加例を以下に示す。

```
<!-- (1) -->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-el</artifactId>
    <scope>test</scope> <!-- (2) -->
</dependency>
```

項番	説明
(1)	スタンドアロン環境で動かすプロジェクトの `pom.xml` ファイルに、Expression Language 用のクラスが格納されているライブラリを追加する。 上記例では、組込み用の Apache Tomcat 向けに提供されているライブラリを指定している。`tomcat-embed-el` の jar ファイルには、Expression Language の API 仕様クラスとリファレンス実装クラスの両方が格納されている。
(2)	JUnit を実行するために依存ライブラリが必要になる場合は、スコープは `test` が適切である。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

単項目チェック

単項目チェックを実装するには、

- フォームクラスのフィールドに、Bean Validation 用のアノテーションを付与する
- Controller に、検証するための`@Validated` アノテーションを付与する
- JSP に、検証エラーメッセージを表示するためのタグを追加する

が必要である。

注釈: spring-mvc.xml に`<mvc:annotation-driven>`の設定が行われていれば、Bean Validation は有効になる。

基本的な単項目チェック

「新規ユーザー登録」処理を例に用いて、実装方法を説明する。ここでは「新規ユーザー登録」のフォームに、以下のチェックルールを設ける。

フィールド名	型	ルール
name	java.lang.String	入力必須 1 文字以上 20 文字以下
email	java.lang.String	入力必須 1 文字以上 50 文字以下 E-mail 形式
age	java.lang.Integer	入力必須 1 以上 200 以下

- フォームクラス

  フォームクラスの各フィールドに、Bean Validation のアノテーションを付ける。

```
package com.example.sample.app.validation;

import java.io.Serializable;

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.Email;

public class UserForm implements Serializable {

  private static final long serialVersionUID = 1L;

  @NotNull // (1)
  @Size(min = 1, max = 20) // (2)
  private String name;

  @NotNull
  @Size(min = 1, max = 50)
  @Email // (3)
  private String email;
```

```
  @NotNull // (4)
  @Min(0) // (5)
  @Max(200) // (6)
  private Integer age;

  // omitted setter/getter
}
```

項番	説明
(1)	対象のフィールドが `null` でないことを示す `javax.validation.constraints.NotNull` を付ける。 Spring MVC では、文字列の入力フィールドに未入力の状態でフォームを送信した場合、デフォルトではフォームオブジェクトに **null ではなく、空文字がバインドされる。** この`@NotNull` は、そもそもリクエストパラメータとして `name` が存在することをチェックする。
(2)	対象のフィールドの文字列長 (またはコレクションのサイズ) が指定したサイズの範囲内にあることを示す `javax.validation.constraints.Size` を付ける。 上記の通り、Spring MVC ではデフォルトで、未入力の文字列フィールドには、空文字がバインドされるため、 1 文字以上というルールが入力必須を表す。
(3)	対象のフィールドが E-mail 形式であることを示す `org.hibernate.validator.constraints.Email` を付ける。 E-mail 形式の要件が`@Email` のチェックと合致しない場合は、`javax.validation.constraints.Pattern` を用いて、正規表現を指定する必要がある。 `@Email` については、*Hibernate Validator のチェックルール*を参照されたい。
(4)	数値の入力フィールドに未入力の状態でフォームを送信した場合、フォームオブジェクトに `null` がバインドされるため、`@NotNull` が `age` の入力必須条件を表す。
(5)	対象のフィールドが指定した数値の以上であることを示す `javax.validation.constraints.Min` を付ける。
(6)	対象のフィールドが指定した数値の以下であることを示す `javax.validation.constraints.Max` を付ける。

ちなみに: Bean Validation 標準のアノテーション、Hibernate Validation が用意しているアノテーションについては、*Bean Validation のチェックルール*、*Hibernate Validator のチェックルール*を参照されたい。

ちなみに: 入力フィールドが未入力の場合に、空文字ではなく null にバインドする方法に関しては、文字列フィールドが未入力の場合に *null* をバインドするを参照されたい。

- Controller クラス

入力チェック対象のフォームクラスに、@Validated を付ける。

```
package com.example.sample.app.validation;

import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("user")
public class UserController {

  @ModelAttribute
  public UserForm setupForm() {
    return new UserForm();
  }

  @RequestMapping(value = "create", method = RequestMethod.GET, params = "form
↪")
  public String createForm() {
    return "user/createForm"; // (1)
  }

  @RequestMapping(value = "create", method = RequestMethod.POST, params =
↪"confirm")
  public String createConfirm(@Validated /* (2) */ UserForm form,␣
↪BindingResult /* (3) */ result) {
    if (result.hasErrors()) { // (4)
      return "user/createForm";
    }
    return "user/createConfirm";
  }

  @RequestMapping(value = "create", method = RequestMethod.POST)
  public String create(@Validated UserForm form, BindingResult result) { //␣
↪(5)
```

```
    if (result.hasErrors()) {
      return "user/createForm";
    }
    // omitted business logic
    return "redirect:/user/create?complete";
  }

  @RequestMapping(value = "create", method = RequestMethod.GET, params =
↪"complete")
  public String createComplete() {
    return "user/createComplete";
  }
}
```

項番	説明
(1)	「新規ユーザー登録」フォーム画面を表示する。
(2)	フォームにつけたアノテーションで入力チェックをするために、フォームの引数に `org.springframework.validation.annotation.Validated` を付ける。
(3)	(2) のチェック結果を格納する `org.springframework.validation.BindingResult` を、引数に加える。 この `BindingResult` は、フォームの直後に記述する必要がある。 直後に指定されていない場合は、検証後に結果をバインドできず、`org.springframework.validation.BindException` がスローされる。
(4)	(2) のチェック結果は、`BindingResult.hasErrors()` メソッドで判定できる。 `hasErrors()` の結果が `true` の場合は、入力値に問題があるため、フォーム表示画面に戻す。
(5)	入力内容確認画面から新規作成処理にリクエストを送る際にも、**入力チェックを必ず再実行すること。** 途中でデータを改ざんすることは可能であるため、必ず業務処理の直前で入力チェックは必要である。

注釈: `@Validated` は、Bean Validation 標準ではなく、Spring の独自アノテーションである。Bean Validation 標準の `javax.validation.Valid` アノテーションも使用できるが、`@Validated` は`@Valid` に比べて、バリデーションのグループを指定できる点で優れているため、本ガイドラインでは Controller の引数には、`@Validated` を使用することを推奨する。

- JSP

`<form:errors>`タグで、入力エラーがある場合にエラーメッセージを表示できる。

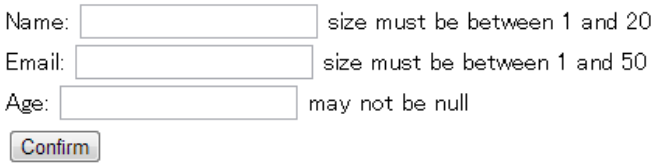
```
<!DOCTYPE html>
<html>
<%-- WEB-INF/views/user/createForm.jsp --%>
<body>
    <form:form modelAttribute="userForm" method="post"
        action="${pageContext.request.contextPath}/user/create">
        <form:label path="name">Name:</form:label>
        <form:input path="name" />
        <form:errors path="name" /><%--(1) --%>
        <br>
        <form:label path="email">Email:</form:label>
        <form:input path="email" />
        <form:errors path="email" />
        <br>
        <form:label path="age">Age:</form:label>
        <form:input path="age" />
        <form:errors path="age" />
        <br>
        <form:button name="confirm">Confirm</form:button>
    </form:form>
</body>
</html>
```

項番	説明
(1)	`<form:errors>`タグの `path` 属性に、対象のフィールド名を指定する。 この例では、フィールド毎に入力フィールドの横にエラーメッセージを表示する。

フォームは、以下のように表示される。



このフォームに対して、すべての入力フィールドを未入力のまま送信すると、以下のようにエラーメッセージが表示される。



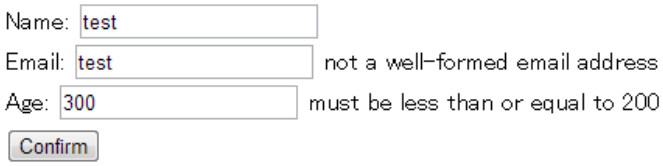
Name と Email が空文字であることに対するエラーメッセージと、Age が `null` であることに対するエラーメッセージが表示されている。

注釈: Bean Validation では、通常、入力値が `null` の場合は正常な値とみなす。ただし、以下のアノテーションを除く。

- `javax.validation.constraints.NotNull`
- `org.hibernate.validator.constraints.NotEmpty`
- `org.hibernate.validator.constraints.NotBlank`

上記の例では、Age の値は `null` であるため、`@Min` と`@Max` によるチェックは正常とみなされ、エラーメッセージは出力されていない。

次に、フィールドに何らかの値を入力してフォームを送信する。



Name の入力値は、チェック条件を満たすため、エラーメッセージが表示されない。

E-mail の入力値は文字列長に関する条件は満たすが、E-mail 形式ではないため、エラーメッセージが表示される。

Age の入力値は最大値を超えているため、エラーメッセージが表示される。

エラー時にスタイルを変更したい場合は、前述のフォームを、以下のように変更する。

```
<form:form modelAttribute="userForm" method="post"
    class="form-horizontal"
    action="${pageContext.request.contextPath}/user/create">
    <form:label path="name" cssErrorClass="error-label">Name:</form:label><%-- (1)␣
↪--%>
    <form:input path="name" cssErrorClass="error-input" /><%-- (2) --%>
    <form:errors path="name" cssClass="error-messages" /><%-- (3) --%>
```

```
    <br>
    <form:label path="email" cssErrorClass="error-label">Email:</form:label>
    <form:input path="email" cssErrorClass="error-input" />
    <form:errors path="email" cssClass="error-messages" />
    <br>
    <form:label path="age" cssErrorClass="error-label">Age:</form:label>
    <form:input path="age" cssErrorClass="error-input" />
    <form:errors path="age" cssClass="error-messages" />
    <br>
    <form:button name="confirm">Confirm</form:button>
</form:form>
```

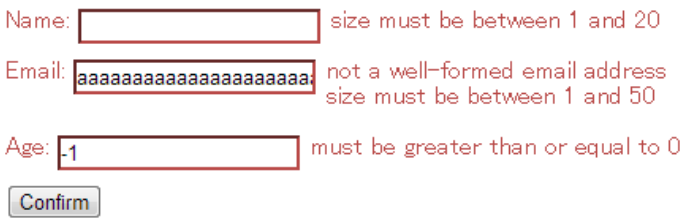
項番	説明
(1)	エラー時に<label>タグへ加えるクラス名を、cssErrorClass 属性で指定する。
(2)	エラー時に<input>タグへ加えるクラス名を、cssErrorClass 属性で指定する。
(3)	エラーメッセージに加えるクラス名を、cssClass 属性で指定する。

この JSP に対して、例えば以下の CSS を適用すると、

```
.form-horizontal input {
    display: block;
    float: left;
}

.form-horizontal label {
    display: block;
    float: left;
    text-align: right;
    float: left;
}

.form-horizontal br {
    clear: left;
}

.error-label {
    color: #b94a48;
}

.error-input {
    border-color: #b94a48;
```

```
    margin-left: 5px;
}

.error-messages {
    color: #b94a48;
    display: block;
    padding-left: 5px;
    overflow-x: auto;
}
```

エラー画面は、以下のように表示される。



画面の要件に応じて CSS をカスタマイズすればよい。

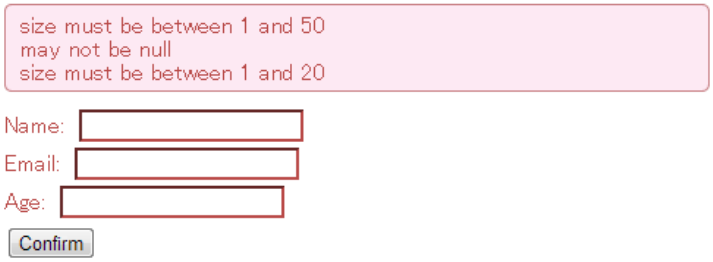
エラーメッセージを、入力フィールドの横に一件一件出力する代わりに、まとめて出力することもできる。

```
<form:form modelAttribute="userForm" method="post"
    action="${pageContext.request.contextPath}/user/create">
    <form:errors path="*" element="div" cssClass="error-message-list" /><%-- (1) --
↪%>

    <form:label path="name" cssErrorClass="error-label">Name:</form:label>
    <form:input path="name" cssErrorClass="error-input" />
    <br>
    <form:label path="email" cssErrorClass="error-label">Email:</form:label>
    <form:input path="email" cssErrorClass="error-input" />
    <br>
    <form:label path="age" cssErrorClass="error-label">Age:</form:label>
    <form:input path="age" cssErrorClass="error-input" />
    <br>
    <form:button name="confirm">Confirm</form:button>
</form:form>
```

項番	説明
(1)	<form:form>タグ内で、<form:errors>の path 属性に」*」 を指定することで、 <form:form>の modelAttribute 属性に指定した Model に関する全エラーメッセージを出力できる。 element 属性に、これらのエラーメッセージを包含するタグ名を指定できる。デフォルトでは、span であるが、 ここではエラーメッセージ一覧をブロック要素として出力するために、div を指定する。 また、CSS のクラスを cssClass 属性に指定する。

例として、以下の CSS クラスを適用した場合の、エラーメッセージ出力例を示す。

```
.form-horizontal input {
    display: block;
    float: left;
}

.form-horizontal label {
    display: block;
    float: left;
    text-align: right;
    float: left;
}

.form-horizontal br {
    clear: left;
}

.error-label {
    color: #b94a48;
}

.error-input {
    border-color: #b94a48;
    margin-left: 5px;
}

.error-message-list {
    color: #b94a48;
    padding:5px 10px;
    background-color: #fde9f3;
    border:1px solid #c98186;
    border-radius:5px;
    margin-bottom: 10px;
}
```



デフォルトでは、エラーメッセージにフィールド名は含まれず、どのフィールドのエラーメッセージなのかが分かりにくい。

そのため、エラーメッセージを一覧で表示する場合は、エラーメッセージの中にフィールド名を含めるようにメッセージを定義する必要がある。

エラーメッセージの定義方法については、「エラーメッセージの定義」を参照されたい。

注釈: エラーメッセージを一覧で表示する際の注意点

エラーメッセージの出力順序は順不同であり、標準機能で出力順序を制御することはできない。そのため、出力順序を制御する (一定に保つ) 必要がある場合は、エラー情報をソートするなどの拡張実装が必要となる。

「エラーメッセージを一覧で表示する」方式では、

- フィールド単位のエラーメッセージ定義
- エラーメッセージの出力順序を制御するための拡張実装

が必要となるため、「入力フィールドの横にエラーメッセージを表示する」方式に比べて対応コストが高くなる。**本ガイドラインでは、画面要件による制約がない場合は「入力フィールドの横にエラーメッセージを表示する」方式を推奨する。**

なお、エラーメッセージの出力順序を制御するための拡張方法としては、Spring Framework から提供されている `org.springframework.validation.beanvalidation.LocalValidatorFactoryBean` の継承クラスを作成し、`processConstraintViolations` メソッドをオーバーライドしてエラー情報をソートする方法などが考えられる。

注釈: @GroupSequence アノテーションについて

チェック順番を制御するための仕組みとして@GroupSequence アノテーションが提供されているが、この仕組みは以下のような動作になるため、エラーメッセージの出力順序を制御するための仕組みではないという点を補足しておく。

- エラーが発生した場合に後続のグループのチェックが実行されない。
- 同一グループ内のチェックで複数のエラー (複数の項目でエラー) が発生するとエラーメッセージの出力順序は順不同になる。

注釈: エラーメッセージをまとめて表示する際に、<form:form>タグの外に表示したい場合は以下のように<spring:nestedPath>タグを使用する。

```
<spring:nestedPath path="userForm">
    <form:errors path="*" element="div"
        cssClass="error-message-list" />
</spring:nestedPath>
<hr>
<form:form modelAttribute="userForm" method="post"
    action="${pageContext.request.contextPath}/user/create">
    <form:label path="name" cssErrorClass="error-label">Name:</form:label>
    <form:input path="name" cssErrorClass="error-input" />
    <br>
    <form:label path="email" cssErrorClass="error-label">Email:</form:label>
    <form:input path="email" cssErrorClass="error-input" />
    <br>
    <form:label path="age" cssErrorClass="error-label">Age:</form:label>
    <form:input path="age" cssErrorClass="error-input" />
    <br>
    <form:button name="confirm">Confirm</form:button>
</form:form>
```

日時フォーマットのチェック

日時フォーマットのチェックを行う場合には、Bean Validation の仕組みではなく、Spring が提供する日時のフォーマットを指定する@DateTimeFormat アノテーションの使用を推奨する。

@DateTimeFormat アノテーションの使用方法については、フィールド単位の日時型変換を参照されたい。

Bean Validation の@Pattern アノテーションを使用することでも日時フォーマットのチェックは可能である。

しかし、@Pattern アノテーションを使用すると、日時フォーマットを正規表現で記述する必要があり、存在しない日時をチェックする場合には、記述が煩雑化する。

そのため、@Pattern アノテーションよりも@DateTimeFormat アノテーションのほうが実装はシンプルになる。

@DateTimeFormat アノテーションは Spring が提供する型変換の仕組みのひとつであるので、入力エラーの場合には、Bean Validation のエラーメッセージではなく、型のミスマッチが発生した時にスローされる例外 (TypeMismatchException) の例外メッセージがそのまま画面へ表示される。

例外メッセージが画面に表示されることを避けるため、型のミスマッチが発生した際のエラーメッセージをプロパティファイルに設定する必要がある。

詳細は型のミスマッチを参照されたい。

ネストした Bean の単項目チェック

ネストした Bean を Bean Validation で検証する方法を説明する。

EC サイトにおける「注文」処理の例を考える。「注文」フォームでは、以下のチェックルールを設ける。

フィールド名	型	ルール	説明
coupon	java.lang.String	5 文字以下 半角英数字	クーポンコード
receiverAddress.name	java.lang.String	入力必須 1 文字以上 50 文字以下	お届け先氏名
receiverAddress.postcode	java.lang.String	入力必須 1 文字以上 10 文字以下	お届け先郵便番号
receiverAddress.address	java.lang.String	入力必須 1 文字以上 100 文字以下	お届け先住所
senderAddress.name	java.lang.String	入力必須 1 文字以上 50 文字以下	請求先氏名
senderAddress.postcode	java.lang.String	入力必須 1 文字以上 10 文字以下	請求先郵便番号
senderAddress.address	java.lang.String	入力必須 1 文字以上 100 文字以下	請求先住所

receiverAddress と senderAddress は、同じ項目であるため、同じフォームクラスを使用する。

- フォームクラス

```
package com.example.sample.app.validation;

import java.io.Serializable;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

public class OrderForm implements Serializable {
    private static final long serialVersionUID = 1L;

    @Size(max = 5)
    @Pattern(regexp = "[a-zA-Z0-9]*")
    private String coupon;

    @NotNull // (1)
    @Valid // (2)
    private AddressForm receiverAddress;

    @NotNull
    @Valid
    private AddressForm senderAddress;

    // omitted setter/getter
}
```

```
package com.example.sample.app.validation;

import java.io.Serializable;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class AddressForm implements Serializable {
    private static final long serialVersionUID = 1L;

    @NotNull
    @Size(min = 1, max = 50)
    private String name;

    @NotNull
    @Size(min = 1, max = 10)
    private String postcode;

    @NotNull
    @Size(min = 1, max = 100)
    private String address;

    // omitted setter/getter
}
```

項番	説明
(1)	子フォーム自体が必須であることを示す。 この設定がない場合、`receiverAddress` に `null` が設定されても、正常とみなされる。
(2)	ネストした Bean の Bean Validation を有効にするために、`javax.validation.Valid` アノテーションを付与する。

- Controller クラス

  前述の Controller と違いはない。

```
package com.example.sample.app.validation;

import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@RequestMapping("order")
@Controller
public class OrderController {

    @ModelAttribute
    public OrderForm setupForm() {
        return new OrderForm();
    }

    @RequestMapping(value = "order", method = RequestMethod.GET, params =
↪"form")
    public String orderForm() {
        return "order/orderForm";
    }

    @RequestMapping(value = "order", method = RequestMethod.POST, params =
↪"confirm")
    public String orderConfirm(@Validated OrderForm form, BindingResult␣
↪result) {
        if (result.hasErrors()) {
            return "order/orderForm";
        }
        return "order/orderConfirm";
    }
}
```

- JSP

```
<!DOCTYPE html>
<html>
<%-- WEB-INF/views/order/orderForm.jsp --%>
<head>
<style type="text/css">
  /* omitted (same as previous sample) */
</style>
</head>
<body>
    <form:form modelAttribute="orderForm" method="post"
        class="form-horizontal"
        action="${pageContext.request.contextPath}/order/order">
        <form:label path="coupon" cssErrorClass="error-label">Coupon Code:</
↪form:label>
        <form:input path="coupon" cssErrorClass="error-input" />
        <form:errors path="coupon" cssClass="error-messages" />
        <br>
    <fieldset>
        <legend>Receiver</legend>
        <%-- (1) --%>
        <form:errors path="receiverAddress"
            cssClass="error-messages" />
        <%-- (2) --%>
        <form:label path="receiverAddress.name"
            cssErrorClass="error-label">Name:</form:label>
        <form:input path="receiverAddress.name"
            cssErrorClass="error-input" />
        <form:errors path="receiverAddress.name"
            cssClass="error-messages" />
        <br>
        <form:label path="receiverAddress.postcode"
            cssErrorClass="error-label">Postcode:</form:label>
        <form:input path="receiverAddress.postcode"
            cssErrorClass="error-input" />
        <form:errors path="receiverAddress.postcode"
            cssClass="error-messages" />
        <br>
        <form:label path="receiverAddress.address"
            cssErrorClass="error-label">Address:</form:label>
        <form:input path="receiverAddress.address"
            cssErrorClass="error-input" />
        <form:errors path="receiverAddress.address"
            cssClass="error-messages" />
    </fieldset>
    <br>
    <fieldset>
        <legend>Sender</legend>
        <form:errors path="senderAddress"
            cssClass="error-messages" />
        <form:label path="senderAddress.name"
            cssErrorClass="error-label">Name:</form:label>
```

```
        <form:input path="senderAddress.name"
            cssErrorClass="error-input" />
        <form:errors path="senderAddress.name"
            cssClass="error-messages" />
        <br>
        <form:label path="senderAddress.postcode"
            cssErrorClass="error-label">Postcode:</form:label>
        <form:input path="senderAddress.postcode"
            cssErrorClass="error-input" />
        <form:errors path="senderAddress.postcode"
            cssClass="error-messages" />
        <br>
        <form:label path="senderAddress.address"
            cssErrorClass="error-label">Address:</form:label>
        <form:input path="senderAddress.address"
            cssErrorClass="error-input" />
        <form:errors path="senderAddress.address"
            cssClass="error-messages" />
    </fieldset>

        <form:button name="confirm">Confirm</form:button>
    </form:form>
</body>
</html>
```

項番	説明
(1)	不正な操作により、receiverAddress.name、receiverAddress.postcode、receiverAddress.address のすべて がリクエストパラメータとして送信されない場合、receiverAddress が null とみなされ、この位置にエラーメッセージが表示される。
(2)	子フォームのフィールドは、**親フィールド名.子フィールド名**で指定する。

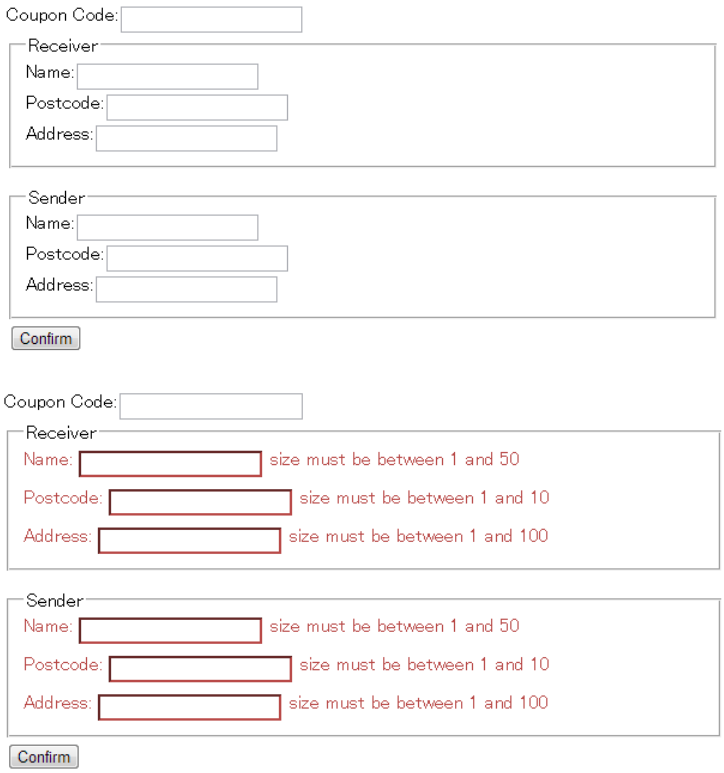
フォームは、以下のように表示される。

このフォームに対して、すべての入力フィールドを未入力のまま送信すると、以下のようにエラーメッセージが表示される。

ネストした Bean のバリデーションはコレクションに対しても有効である。

最初に説明した「ユーザー登録」フォームに住所を 3 件まで登録できるようにフィールドを追加する。住所には、前述の AddressForm を利用する。

- フォームクラス AddressForm のリストを、フィールドに追加する。




```
package com.example.sample.app.validation;

import java.io.Serializable;
import java.util.List;

import javax.validation.Valid;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.Email;

public class UserForm implements Serializable {

    private static final long serialVersionUID = 1L;

    @NotNull
    @Size(min = 1, max = 20)
    private String name;

    @NotNull
    @Size(min = 1, max = 50)
    @Email
    private String email;

    @NotNull
    @Min(0)
    @Max(200)
```

```
    private Integer age;

    @NotNull
    @Size(min = 1, max = 3) // (1)
    @Valid
    private List<AddressForm> addresses;

    // omitted setter/getter
}
```

項番	説明
(1)	コレクションのサイズチェックにも、@Size アノテーションを使用できる。

- JSP

```
<!DOCTYPE html>
<html>
<%-- WEB-INF/views/user/createForm.jsp --%>
<head>
<style type="text/css">
  /* omitted (same as previous sample) */
</style>
</head>
<body>

    <form:form modelAttribute="userForm" method="post"
        class="form-horizontal"
        action="${pageContext.request.contextPath}/user/create">
        <form:label path="name" cssErrorClass="error-label">Name:</form:label>
        <form:input path="name" cssErrorClass="error-input" />
        <form:errors path="name" cssClass="error-messages" />
        <br>
        <form:label path="email" cssErrorClass="error-label">Email:</
↪form:label>
        <form:input path="email" cssErrorClass="error-input" />
        <form:errors path="email" cssClass="error-messages" />
        <br>
        <form:label path="age" cssErrorClass="error-label">Age:</form:label>
        <form:input path="age" cssErrorClass="error-input" />
        <form:errors path="age" cssClass="error-messages" />
        <br>
        <form:errors path="addresses" cssClass="error-messages" /><%-- (1) --
↪%>
        <c:forEach items="${userForm.addresses}" varStatus="status"><%-- (2) -
↪-%>
            <fieldset class="address">
                <legend>Address${f:h(status.index + 1)}</legend>
                <form:label path="addresses[${status.index}].name"
```

```
                    cssErrorClass="error-label">Name:</form:label><%-- (3) --
↪%>
                <form:input path="addresses[${status.index}].name"
                    cssErrorClass="error-input" />
                <form:errors path="addresses[${status.index}].name"
                    cssClass="error-messages" />
                <br>
                <form:label path="addresses[${status.index}].postcode"
                    cssErrorClass="error-label">Postcode:</form:label>
                <form:input path="addresses[${status.index}].postcode"
                    cssErrorClass="error-input" />
                <form:errors path="addresses[${status.index}].postcode"
                    cssClass="error-messages" />
                <br>
                <form:label path="addresses[${status.index}].address"
                    cssErrorClass="error-label">Address:</form:label>
                <form:input path="addresses[${status.index}].address"
                    cssErrorClass="error-input" />
                <form:errors path="addresses[${status.index}].address"
                    cssClass="error-messages" />
                <c:if test="${status.index > 0}">
                    <br>
                    <button class="remove-address-button">Remove</button>
                </c:if>
            </fieldset>
            <br>
        </c:forEach>
        <button id="add-address-button">Add address</button>
        <br>
        <form:button name="confirm">Confirm</form:button>
    </form:form>
    <script type="text/javascript"
        src="${pageContext.request.contextPath}/resources/vendor/js/jquery-1.
↪10.2.min.js"></script>
    <script type="text/javascript"
        src="${pageContext.request.contextPath}/resources/app/js/
↪AddressesView.js"></script>
</body>
</html>
```

項番	説明
(1)	addresses フィールドに対するエラーメッセージを表示する。
(2)	子フォームのコレクションを、<c:forEach>タグを使ってループで処理する。
(3)	コレクション中の子フォームのフィールドは、**親フィールド名[インデックス].子フィールド名**で指定する。

- Controller クラス

```
package com.example.sample.app.validation;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("user")
public class UserController {

    @ModelAttribute
    public UserForm setupForm() {
        UserForm form = new UserForm();
        List<AddressForm> addresses = new ArrayList<AddressForm>();
        addresses.add(new AddressForm());
        form.setAddresses(addresses); // (1)
        return form;
    }

    @RequestMapping(value = "create", method = RequestMethod.GET, params =
↪"form")
    public String createForm() {
        return "user/createForm";
    }

    @RequestMapping(value = "create", method = RequestMethod.POST, params =
↪"confirm")
    public String createConfirm(@Validated UserForm form, BindingResult␣
↪result) {
        if (result.hasErrors()) {
            return "user/createForm";
        }
        return "user/createConfirm";
    }
}
```

項番	説明
(1)	「ユーザー登録」フォーム初期表示時に、一件の住所フォームを表示させるために、フォームオブジェクトを編集する。

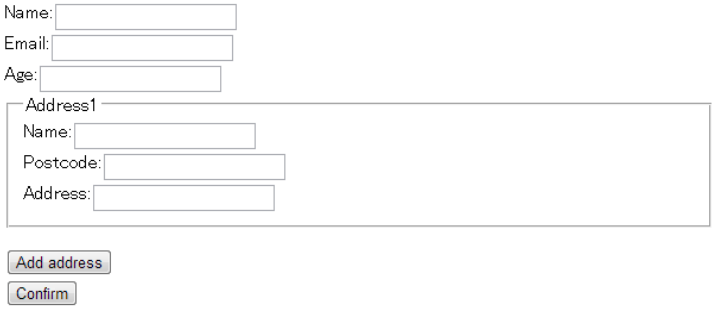
- JavaScript

  動的にアドレス入力フィールドを追加するための JavaScript も記載するが、このコードの説明は、本質的ではないため割愛する。

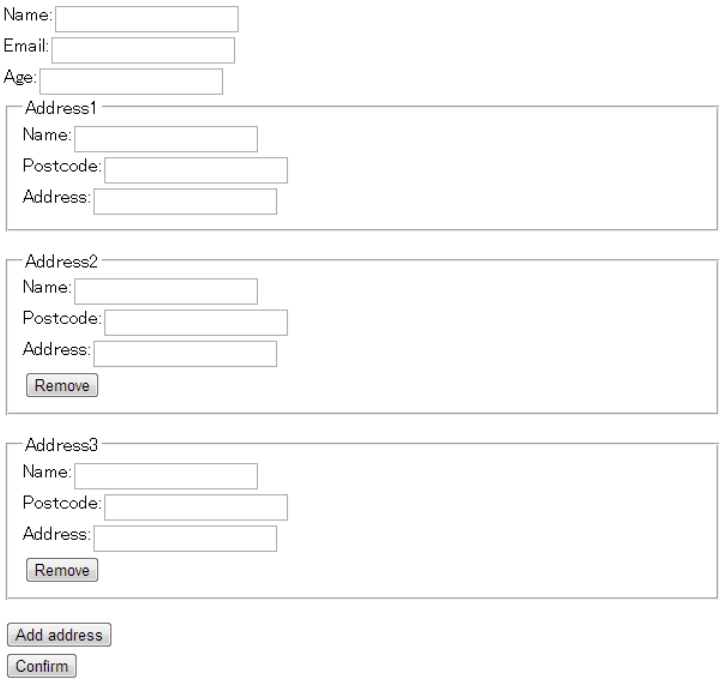
```
// webapp/resources/app/js/AddressesView.js

function AddressesView() {
  this.addressSize = $('fieldset.address').size();
};

AddressesView.prototype.addAddress = function() {
  var $address = $('fieldset.address');
  var newHtml = addressTemplate(this.addressSize++);
  $address.last().next().after($(newHtml));
};

AddressesView.prototype.removeAddress = function($fieldset) {
  $fieldset.next().remove(); // remove <br>
  $fieldset.remove(); // remove <fieldset>
};

function addressTemplate(number) {
  return '\
<fieldset class="address">\
    <legend>Address' + (number + 1) + '</legend>\
    <label for="addresses' + number + '.name">Name:</label>\
    <input id="addresses' + number + '.name" name="addresses[' + number + '].
↪name" type="text" value=""/><br>\
    <label for="addresses' + number + '.postcode">Postcode:</label>\
    <input id="addresses' + number + '.postcode" name="addresses[' + number +
↪'].postcode" type="text" value=""/><br>\
    <label for="addresses' + number + '.address">Address:</label>\
    <input id="addresses' + number + '.address" name="addresses[' + number +
↪'].address" type="text" value=""/><br>\
    <button class="remove-address-button">Remove</button>\
</fieldset>\
<br>\
';
}

$(function() {
  var addressesView = new AddressesView();

  $('#add-address-button').on('click', function(e) {
    e.preventDefault();
    addressesView.addAddress();
  });

  $(document).on('click', '.remove-address-button', function(e) {
    if (this === e.target) {
      e.preventDefault();
      var $this = $(this); // this button
```

```
      var $fieldset = $this.parent(); // fieldset
      addressesView.removeAddress($fieldset);
    }
  });

});
```

フォームは、以下のように表示される。



「Add address」ボタンを 2 回押して、住所フォームを 2 件追加する。

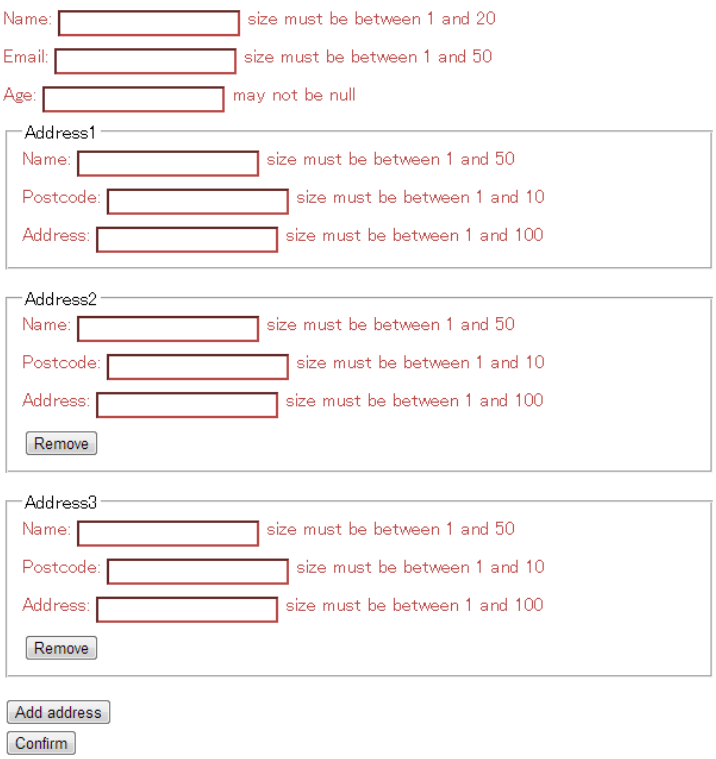


このフォームに対して、すべての入力フィールドを未入力のまま送信すると、以下のようにエラーメッセージが表示される。

バリデーションのグループ化

バリデーショングループを作成し、一つのフィールドに対して、グループごとに入力チェックルールを指定することができる。

前述の「新規ユーザー登録」の例で、age フィールドに「成年であること」というルールを追加する。「成年かどうか」は国によってルールが違うため、country フィールドも追加する。



Bean Validation でグループを指定する場合、アノテーションの `group` 属性に、グループを示す任意の `java.lang.Class` オブジェクトを設定する。

ここでは、以下の 3 グループ (interface) を作成する。

グループ	成人条件
`Chinese`	18 歳以上
`Japanese`	20 歳以上
`Singaporean`	21 歳以上

このグループをつかって、バリデーションを実行する例を示す。

- フォームクラス

```
package com.example.sample.app.validation;

import java.io.Serializable;
import java.util.List;

import javax.validation.Valid;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.Email;

public class UserForm implements Serializable {
```

```
    private static final long serialVersionUID = 1L;

    // (1)
    public static interface Chinese {
    };

    public static interface Japanese {
    };

    public static interface Singaporean {
    };

    @NotNull
    @Size(min = 1, max = 20)
    private String name;

    @NotNull
    @Size(min = 1, max = 50)
    @Email
    private String email;

    @NotNull
    @Min.List({ // (2)
            @Min(value = 18, groups = Chinese.class), // (3)
            @Min(value = 20, groups = Japanese.class),
            @Min(value = 21, groups = Singaporean.class)
            })
    @Max(200)
    private Integer age;

    @NotNull
    @Size(min = 2, max = 2)
    private String country; // (4)

    // omitted setter/getter
}
```

項番	説明
(1)	グループクラスを指定するために、各グループをインタフェースで定義する。
(2)	一つのフィールドに同じルールを複数指定するために、`@Min.List` アノテーションを使用する。 他のアノテーションを使用する場合も同様である。
(3)	グループごとにルールを定義する。グループを指定するために、`groups` 属性に対象のグループクラスを指定する。 `groups` 属性を省略した場合、`javax.validation.groups.Default` グループが使用される。
(4)	グループを振り分けるための、フィールドを追加する。

- JSP

JSP に大きな変更はない。

```
<form:form modelAttribute="userForm" method="post"
    class="form-horizontal"
    action="${pageContext.request.contextPath}/user/create">
    <form:label path="name" cssErrorClass="error-label">Name:</form:label>
    <form:input path="name" cssErrorClass="error-input" />
    <form:errors path="name" cssClass="error-messages" />
    <br>
    <form:label path="email" cssErrorClass="error-label">Email:</form:label>
    <form:input path="email" cssErrorClass="error-input" />
    <form:errors path="email" cssClass="error-messages" />
    <br>
    <form:label path="age" cssErrorClass="error-label">Age:</form:label>
    <form:input path="age" cssErrorClass="error-input" />
    <form:errors path="age" cssClass="error-messages" />
    <br>
    <form:label path="country" cssErrorClass="error-label">Country:</
↪form:label>
    <form:select path="country" cssErrorClass="error-input">
        <form:option value="cn">China</form:option>
        <form:option value="jp">Japan</form:option>
        <form:option value="sg">Singapore</form:option>
    </form:select>
```

```
    <form:errors path="country" cssClass="error-messages" />
    <br>
    <form:button name="confirm">Confirm</form:button>
</form:form>
```

- Controller クラス

  @Validated に、対象のグループを設定することで、バリデーションルールを変更できる。

```
package com.example.sample.app.validation;


import javax.validation.groups.Default;

import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.example.sample.app.validation.UserForm.Chinese;
import com.example.sample.app.validation.UserForm.Japanese;
import com.example.sample.app.validation.UserForm.Singaporean;

@Controller
@RequestMapping("user")
public class UserController {

    @ModelAttribute
    public UserForm setupForm() {
        UserForm form = new UserForm();
        return form;
    }

    @RequestMapping(value = "create", method = RequestMethod.GET, params =
↪"form")
    public String createForm() {
        return "user/createForm";
    }

    String createConfirm(UserForm form, BindingResult result) {
        if (result.hasErrors()) {
            return "user/createForm";
        }
        return "user/createConfirm";
    }

    @RequestMapping(value = "create", method = RequestMethod.POST, params = {
            "confirm",  /* (1) */ "country=cn" })
    public String createConfirmForChinese(@Validated({ /* (2) */ Chinese.
↪class,
            Default.class }) UserForm form, BindingResult result) {
```

```
        return createConfirm(form, result);
    }

    @RequestMapping(value = "create", method = RequestMethod.POST, params = {
            "confirm", "country=jp" })
    public String createConfirmForJapanese(@Validated({ Japanese.class,
            Default.class }) UserForm form, BindingResult result) {
        return createConfirm(form, result);
    }

    @RequestMapping(value = "create", method = RequestMethod.POST, params = {
            "confirm", "country=sg" })
    public String createConfirmForSingaporean(@Validated({ Singaporean.class,
            Default.class }) UserForm form, BindingResult result) {
        return createConfirm(form, result);
    }
}
```

項番	説明
(1)	グループを振り分けるためのパラメータの条件を、param 属性に追加する。
(2)	@Min 以外のアノテーションは、Default グループに属しているため、Default の指定も必要である。

この例では、各入力値の組み合わせに対するチェック結果は、以下の表の通りである。

age の値	country の値	入力チェック結果	エラーメッセージ
17	cn	NG	must be greater than or equal to 18
	jp	NG	must be greater than or equal to 20
	sg	NG	must be greater than or equal to 21
18	cn	OK	
	jp	NG	must be greater than or equal to 20
	sg	NG	must be greater than or equal to 21
20	cn	OK	
	jp	OK	
	sg	NG	must be greater than or equal to 21
21	cn	OK	
	jp	OK	
	sg	OK	

警告: この Controller の実装は、country の値が、」cn」、」jp」、」sg」のいずれでもない場合のハンドリングが行われておらず、不十分である。country の値が、想定外の場合に、400 エラーが返却される。

次にチェック対象の国が増えたため、成人条件 18 歳以上をデフォルトルールとしたい場合を考える。

ルールは、以下のようになる。

グループ	成人条件
Japanese	20 歳以上
Singaporean	21 歳以上
上記以外の国 (Default)	18 歳以上

- フォームクラス

  Default グループに意味を持たせるため、@Min 以外のアノテーションにも、明示的に全グループを指定する必要がある。

```
package com.example.sample.app.validation;

import java.io.Serializable;
import java.util.List;

import javax.validation.Valid;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.validation.groups.Default;

import org.hibernate.validator.constraints.Email;

public class UserForm implements Serializable {

    private static final long serialVersionUID = 1L;

    public static interface Japanese {
    };

    public static interface Singaporean {
    };

    @NotNull(groups = { Default.class, Japanese.class, Singaporean.class }) //
 (1)
    @Size(min = 1, max = 20, groups = { Default.class, Japanese.class,
            Singaporean.class })
    private String name;

    @NotNull(groups = { Default.class, Japanese.class, Singaporean.class })
    @Size(min = 1, max = 50, groups = { Default.class, Japanese.class,
```

```
            Singaporean.class })
    @Email(groups = { Default.class, Japanese.class, Singaporean.class })
    private String email;

    @NotNull(groups = { Default.class, Japanese.class, Singaporean.class })
    @Min.List({
            @Min(value = 18, groups = Default.class), // (2)
            @Min(value = 20, groups = Japanese.class),
            @Min(value = 21, groups = Singaporean.class) })
    @Max(value = 200, groups = { Default.class, Japanese.class, Singaporean.
↪class })
    private Integer age;

    @NotNull(groups = { Default.class, Japanese.class, Singaporean.class })
    @Size(min = 2, max = 2, groups = { Default.class, Japanese.class,
            Singaporean.class })
    private String country;

    // omitted setter/getter
}
```

項番	説明
(1)	`@Min` 以外のアノテーションにも、全グループを設定する。
(2)	`Default` グループに対するルールを設定する。

- JSP

  JSP に変更はない

- Controller クラス

```
package com.example.sample.app.validation;

import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.example.sample.app.validation.UserForm.Japanese;
import com.example.sample.app.validation.UserForm.Singaporean;

@Controller
@RequestMapping("user")
```

```
public class UserController {

    @ModelAttribute
    public UserForm setupForm() {
        UserForm form = new UserForm();
        return form;
    }

    @RequestMapping(value = "create", method = RequestMethod.GET, params =
↪"form")
    public String createForm() {
        return "user/createForm";
    }

    String createConfirm(UserForm form, BindingResult result) {
        if (result.hasErrors()) {
            return "user/createForm";
        }
        return "user/createConfirm";
    }

    @RequestMapping(value = "create", method = RequestMethod.POST, params = {
↪"confirm" })
    public String createConfirmForDefault(@Validated /* (1) */ UserForm form,
            BindingResult result) {
        return createConfirm(form, result);
    }

    @RequestMapping(value = "create", method = RequestMethod.POST, params = {
            "confirm", "country=jp" })
    public String createConfirmForJapanese(
            @Validated(Japanese.class)  /* (2) */ UserForm form,␣
↪BindingResult result) {
        return createConfirm(form, result);
    }

    @RequestMapping(value = "create", method = RequestMethod.POST, params = {
            "confirm", "country=sg" })
    public String createConfirmForSingaporean(
            @Validated(Singaporean.class) UserForm form, BindingResult␣
↪result) {
        return createConfirm(form, result);
    }
}
```

項番	説明
(1)	country フィールド指定がない場合に、Default グループが使用されるように設定する。
(2)	country フィールド指定がある場合に、Default グループが含まれないように設定する。

バリデーショングループを使用する方法について、2 パターン説明した。

前者は Default グループを Controller クラスで使用し、後者は Default グループをフォームクラスで使用した。

パターン	メリット	デメリット	使用の判断ポイント
Default グループを Controller クラスで使用	グループ化する必要のないルールは、group 属性を設定する必要がない。	グループの全パターンを定義する必要があるので、グループパターンが多いと、定義が困難になる。	グループパターンが、数種類の場合に使用すべき (新規作成グループ、更新グループ、削除グループ等)
Default グループをフォームクラスで使用	デフォルトに属さないグループのみ定義すればよいため、パターンが多くても対応できる。	グループ化する必要のないルールにも、group 属性を設定する必要があり、管理が煩雑になる。	グループパターンにデフォルト値を設定できる (グループの大多数に共通項がある) 場合に使用すべき

使用の判断ポイントのどちらにも当てはまらない場合は、Bean Validation の使用が不適切であることが考えられる。設計を見直したうえで、Spring Validator の使用または業務ロジックチェックでの実装を検討すること。

注釈: これまでの例ではバリデーショングループの切り替えは、リクエストパラメータ等、@RequestMapping アノテーションで指定できるパラメータによって行った。この方法では認証オブジェクトが有する権限情報など、@RequestMapping アノテーションでは扱えない情報でグループを切り替えることはできない。

この場合は、@Validated アノテーションを使用せず、org.springframework.validation.SmartValidator を使用し、Controller のハンドラメソッド内でグループを指定したバリデーションを行えばよい。

```
@Controller
@RequestMapping("user")
public class UserController {

    @Inject
    SmartValidator smartValidator; // (1)
```

```
    // omitted

    @RequestMapping(value = "create", method = RequestMethod.POST, params =
↪"confirm")
    public String createConfirm(/* (2) */ UserForm form, BindingResult␣
↪result) {
        // (3)
        Class<?> validationGroup = Default.class;
        // logic to determine validation group
        // if (xxx) {
        //     validationGroup = Xxx.class;
        // }
        smartValidator.validate(form, result, validationGroup); // (4)
        if (result.hasErrors()) {
            return "user/createForm";
        }
        return "user/createConfirm";
    }

}
```

項番	説明
(1)	SmartValidator をインジェクションする。SmartValidator は<mvc:annotation-driven>の設定が行われていれば使用できるため、別途 Bean 定義不要である。
(2)	@Validated アノテーションは使わない。
(3)	バリデーショングループを決定する。 バリデーショングループを決定するロジックは、Helper クラスに委譲して、Controller 内のロジックをシンプルな状態に保つことを推奨する。
(4)	SmartValidator の validate メソッドを使用して、グループを指定したバリデーションを実行する。 グループの指定は可変長引数になっており、複数指定できる。

基本的には、Controller にロジックを書くべきではないため、@RequestMapping の属性でルールを切り替

えられるのであれば、SmartValidator は使わない方がよい。

相関項目チェック

複数フィールドにまたがる相関項目チェックには、Spring Validator(org.springframework.validation.Validator インタフェースを実装した Validator)、または、Bean Validation を用いる。

それぞれ説明するが、先にそれぞれの特徴と推奨する使い分けを述べる。

方式	特徴	用途
Spring Validator	特定のクラスに対する入力チェックの作成が容易である。 Controller での利用が不便。	特定のフォームに依存した業務要件固有の入力チェック実装
Bean Validation	入力チェックの作成は Spring Validator ほど容易でない。 Controller での利用が容易。	特定のフォームに依存しない、開発プロジェクト共通の入力チェック実装

Spring Validator による相関項目チェック実装

「パスワードリセット」処理を例に実装方法を説明する。

以下のルールを実装する。ここでは「パスワードリセット」のフォームに以下のチェックルールを設ける。

フィールド名	型	ルール	説明
password	java.lang.String	入力必須 8 文字以上 **confirmPassword** と同じ値であること	パスワード
confirmPassword	java.lang.String	特になし	確認用パスワード

「confirmPassword と同じ値であること」というルールは password フィールドと confirmPassword フィールドの両方の情報が必要であるため、相関項目チェックルールである。

- フォームクラス

相関項目チェックルール以外は、これまで通り Bean Validation のアノテーションで実装する。

```
package com.example.sample.app.validation;

import java.io.Serializable;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class PasswordResetForm implements Serializable {
    private static final long serialVersionUID = 1L;

    @NotNull
    @Size(min = 8)
    private String password;

    private String confirmPassword;

    // omitted setter/getter
}
```

注釈: パスワードは、通常ハッシュ化してデータベースに保存するため、最大値のチェックは行わなくても良い。

- Validator クラス

`org.springframework.validation.Validator` インタフェースを実装して、相関項目チェックルールを実現する。

```
package com.example.sample.app.validation;

import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

@Component // (1)
public class PasswordEqualsValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return PasswordResetForm.class.isAssignableFrom(clazz); // (2)
    }

    @Override
    public void validate(Object target, Errors errors) {

        if (errors.hasFieldErrors("password")) { // (3)
            return;
```

```
        }

        PasswordResetForm form = (PasswordResetForm) target;
        String password = form.getPassword();
        String confirmPassword = form.getConfirmPassword();

        if (!password.equals(confirmPassword)) { // (4)
            errors.rejectValue(/* (5) */ "password",
            /* (6) */ "PasswordEqualsValidator.passwordResetForm.password",
            /* (7) */ "password and confirm password must be same.");
        }
    }
}
```

項番	説明
(1)	`@Component` を付与し、Validator をコンポーネントスキャン対象にする。
(2)	この Validator のチェック対象であるかどうかを判別する。ここでは、`PasswordResetForm` クラスをチェック対象とする。
(3)	単項目チェック時に対象フィールドでエラーが発生している場合は、この Validator で相関チェックは行わない。 相関チェックを必ず行う必要がある場合は、この判定ロジックは不要である。
(4)	チェックロジックを実装する。
(5)	エラー対象のフィールド名を指定する。
(6)	エラーメッセージのコード名を指定する。ここではコードを、 「バリデータ名. フォーム属性名. プロパティ名」 とする。メッセージ定義は *application-messages.properties に定義するメッセージ*を参照されたい。
(7)	エラーメッセージをコードで解決できなかった場合に使用する、デフォルトメッセージを設定する。

注釈: Spring Validator 実装クラスは、使用する Controller と同じパッケージに配置することを推奨する。

- Controller クラス

```
package com.example.sample.app.validation;

import javax.inject.Inject;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("password")
public class PasswordResetController {
    @Inject
    PasswordEqualsValidator passwordEqualsValidator; // (1)

    @ModelAttribute
    public PasswordResetForm setupForm() {
        return new PasswordResetForm();
    }

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        binder.addValidators(passwordEqualsValidator); // (2)
    }

    @RequestMapping(value = "reset", method = RequestMethod.GET, params =
↪"form")
    public String resetForm() {
        return "password/resetForm";
    }

    @RequestMapping(value = "reset", method = RequestMethod.POST)
    public String reset(@Validated PasswordResetForm form, BindingResult␣
↪result) { // (3)
        if (result.hasErrors()) {
            return "password/resetForm";
        }
        return "redirect:/password/reset?complete";
    }

    @RequestMapping(value = "reset", method = RequestMethod.GET, params =
↪"complete")
    public String resetComplete() {
        return "password/resetComplete";
    }
}
```

項番	説明
(1)	使用する Spring Validator を、インジェクションする。
(2)	`@InitBinder` アノテーションがついたメソッド内で、 `WebDataBinder.addValidators` メソッドにより、Validator を追加する。 これにより、`@Validated` アノテーションでバリデーションをする際に、追加した Validator も呼び出される。
(3)	入力チェックの実装は、これまで通りである。

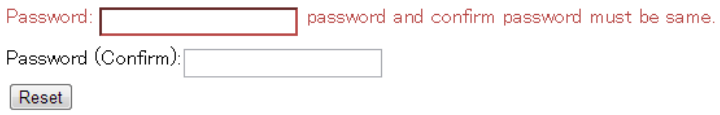
- JSP

  JSP に特筆すべき点はない。

```
<!DOCTYPE html>
<html>
<%-- WEB-INF/views/password/resetForm.jsp --%>
<head>
<style type="text/css">
/* omitted */
</style>
</head>
<body>
    <form:form modelAttribute="passwordResetForm" method="post"
        class="form-horizontal"
        action="${pageContext.request.contextPath}/password/reset">
        <form:label path="password" cssErrorClass="error-label">Password:</
↪form:label>
        <form:password path="password" cssErrorClass="error-input" />
        <form:errors path="password" cssClass="error-messages" />
        <br>
        <form:label path="confirmPassword" cssErrorClass="error-label">
↪Password (Confirm):</form:label>
        <form:password path="confirmPassword"
            cssErrorClass="error-input" />
        <form:errors path="confirmPassword" cssClass="error-messages" />
        <br>
        <form:button>Reset</form:button>
    </form:form>
</body>
</html>
```

`password` フィールドと、`confirmPassword` フィールドに、別の値を入力してフォームを送信した場合

は、以下のようにエラーメッセージが表示される。



注釈: <form:password>タグを使用すると、再表示時に、データがクリアされる。

注釈: 相関チェック対象の複数フィールドに対してエラー情報を設定することも可能である。ただし、必ずエラーメッセージの表示とスタイル適用がセットで行われ、いずれか片方のみを行うことはできない。

相関チェックエラーとなった両方のフィールドにスタイル適用したいが、エラーメッセージは 1 つだけ表示したいような場合は、エラーメッセージに空文字を設定することで実現することが可能である。以下に、password フィールドと confirmPassword フィールドにスタイルを適用し、password フィールドのみにエラーメッセージを表示する例を示す。

```
package com.example.sample.app.validation;

import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

@Component
public class PasswordEqualsValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return PasswordResetForm.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {

        // omitted
        if (!password.equals(confirmPassword)) {
            // register a field error for password
            errors.rejectValue("password",
                    "PasswordEqualsValidator.passwordResetForm.password",
                    "password and confirm password must be same.");

            // register a field error for confirmPassword
            errors.rejectValue("confirmPassword", // (1)
                      "PasswordEqualsValidator.passwordResetForm.
↪confirmPassword", // (2)
                      ""); // (3)
        }
    }
}
```

項番	説明
(1)	`confirmPassword` フィールドのエラーを登録する。
(2)	エラーメッセージのコード名を指定する。この際、対応するエラーメッセージに空文字を指定する。 メッセージ定義は *application-messages.properties に定義するメッセージ*を参照されたい。
(3)	エラーメッセージをコードで解決できなかった場合に使用する、デフォルトメッセージを設定する。 上記の例では空文字を設定している。

警告: `@InitBinder` アノテーションを付与したメソッドで Validator が登録されると、Validator の `supports` メソッドで Validator のサポート対象の型かどうか判定される。このとき、サポート対象でない場合は `java.lang.IllegalStateException` が発生する点に注意されたい。

`@InitBinder` アノテーションを付与したメソッドは、Model に独自の型のオブジェクトが追加された際に必ず実行されるが、`@InitBinder("xxx")` でモデル名を指定することで、適用範囲を限定することが可能である。

例えば以下のようなケースが該当するため、注意されたい。

- 一つの Controller で複数のフォームを扱う場合（複数のフォームオブジェクトを `@ModelAttribute` アノテーションを付与したメソッドで登録する場合や、ハンドラメソッドの引数として受け取る場合）
- フォームオブジェクトに限らず、ハンドラメソッドの引数として受け取った `Model` に、`ResultMessages` オブジェクトや `Page` オブジェクトなどの独自の型のオブジェクトを登録する場合
- 同様に `RedirectAttributes` に独自の型のオブジェクトを登録する場合

以下に、一つの Controller で複数のフォームを扱う場合の実装例を示す。

```
@Controller
@RequestMapping("xxx")
public class XxxController {
    // omitted
    @ModelAttribute("aaa")
    public AaaForm() {
        return new AaaForm();
    }

    @ModelAttribute("bbb")
    public BbbForm() {
        return new BbbForm();
    }

    @InitBinder("aaa")
    public void initBinderForAaa(WebDataBinder binder) {
        // add validators for AaaForm
        binder.addValidators(aaaValidator);
    }

    @InitBinder("bbb")
    public void initBinderForBbb(WebDataBinder binder) {
        // add validators for BbbForm
        binder.addValidators(bbbValidator);
    }
    // omitted
}
```

注釈: 相関項目チェックルールのチェック内容をバリデーショングループに応じて変更したい場合（例えば、特定のバリデーショングループが指定された場合だけ相関項目チェックを実施したい場合など）は、org.springframework.validation.Validator インターフェイスを実装する代わりに、org.springframework.validation.SmartValidator インターフェイスを実装し、validate メソッド内で処理を切り替えるとよい。

```
package com.example.sample.app.validation;

import org.apache.commons.lang3.ArrayUtils;
import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.SmartValidator;

@Component
public class PasswordEqualsValidator implements SmartValidator { //␣
↪Implements SmartValidator instead of Validator interface

    @Override
    public boolean supports(Class<?> clazz) {
        return PasswordResetForm.class.isAssignableFrom(clazz);
```

```
    }

    @Override
    public void validate(Object target, Errors errors) {
        validate(target, errors, new Object[] {});
    }

    @Override
    public void validate(Object target, Errors errors, Object...␣
↪validationHints) {
        // Check validationHints(groups) and apply validation logic only when
↪'Update.class' is specified
        if (ArrayUtils.contains(validationHints, Update.class)) {
            PasswordResetForm form = (PasswordResetForm) target;
            String password = form.getPassword();
            String confirmPassword = form.getConfirmPassword();

            // omitted...
        }
    }
}
```

Bean Validation による相関項目チェック実装

Bean Validation によって、相関項目チェックの実装するためには、独自バリデーションルールの追加を行う必要がある。

How to extend にて説明する。

エラーメッセージの定義

入力チェックエラーメッセージを変更する方法を説明する。

Spring MVC による Bean Validation のエラーメッセージは、以下の順で解決される。

1. `org.springframework.context.MessageSource` に定義されているメッセージの中に、ルールに合致するものがあればそれをエラーメッセージとして使用する (Spring のルール)。

   Spring のデフォルトのルールについては、「DefaultMessageCodesResolver の JavaDoc」を参照されたい。

2. 1. でメッセージが見つからない場合、アノテーションの `message` 属性に、指定されたメッセージからエラーメッセージを取得する (Bean Validation のルール)

   1. `message` 属性に指定されたメッセージが、」{メッセージキー}」形式でない場合、そのテキストをエラーメッセージとして使用する。

   2. `message` 属性に指定されたメッセージが、」{メッセージキー}」形式の場合、クラスパス直下の ValidationMessages.properties から、メッセージキーに対応するメッセージを探す。

      1. メッセージキーに対応するメッセージが定義されている場合は、そのメッセージを使用する

2. メッセージキーに対応するメッセージが定義されていない場合は、」{メッセージキー}」をそのままエラーメッセージとして使用する

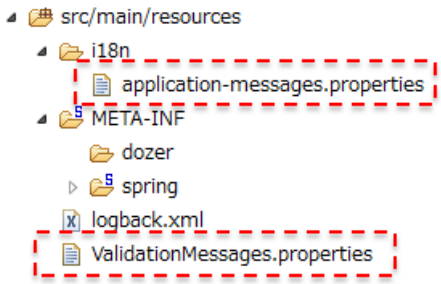
基本的にエラーメッセージは、properties ファイルに定義することを推奨する。

定義する箇所は、以下の 2 パターン存在する。

- `org.springframework.context.MessageSource` が読み込む properties ファイル
- クラスパス直下の ValidationMessages.properties

以下の説明では、applicationContext.xml に次の設定があることを前提とし、前者を」application-messages.properties」、後者を」ValidationMessages.properties」と呼ぶ。

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>i18n/application-messages</value>
        </list>
    </property>
</bean>
```



警告: `ValidationMessages.properties` ファイルは、クラスパスの直下に複数存在させてはいけない。

クラスパスの直下に複数の `ValidationMessages.properties` ファイルが存在する場合、いずれか 1 つのファイルが読み込まれ、他のファイルが読み込まれないため、適切なメッセージが表示されない可能性がある。

- マルチプロジェクト構成を採用する場合は、`ValidationMessages.properties` ファイルを複数のプロジェクトに配置しないように注意すること。
- Bean Validation 用の共通部品を jar ファイルとして配布する際に、`ValidationMessages.properties` ファイルを jar ファイルの中に含めないように注意すること。

なお、version 1.0.2.RELEASE 以降の ブランクプロジェクト からプロジェクトを生成した場合は、`xxx-web/src/main/resources` の直下に `ValidationMessages.properties` が格納されている。

本ガイドラインでは、以下のように定義を分けることを推奨する。

プロパティファイル名	定義する内容
ValidationMessages.properties	システムで定めた Bean Validation のデフォルトエラーメッセージ
application-messages.properties	個別で上書きしたい Bean Validation のエラーメッセージ Spring Validator で実装した入力チェックのエラーメッセージ

ValidationMessages.properties を用意しない場合は、*Hibernate Validator が用意するデフォルトメッセージ*が使用される。

`MessageSource` と連携することで、日本語メッセージを Native to Ascii せずに直接扱うことができる。詳細は、*Native to Ascii を行わないメッセージの読み込み*を参照されたい。

ValidationMessages.properties に定義するメッセージ

クラスパス直下 (通常 src/main/resources) の ValidationMessages.properties 内の、Bean Validation のアノテーションの `message` 属性に指定されたメッセージキーに対して、メッセージを定義する。

基本的な単項目チェックの初めに使用した、以下のフォームを用いて説明する。

- フォームクラス (再掲)

```
public class UserForm implements Serializable {

    @NotNull
    @Size(min = 1, max = 20)
    private String name;

    @NotNull
    @Size(min = 1, max = 50)
    @Email
    private String email;

    @NotNull
    @Min(0)
    @Max(200)
    private Integer age;
```

```
    // omitted getter/setter
}
```

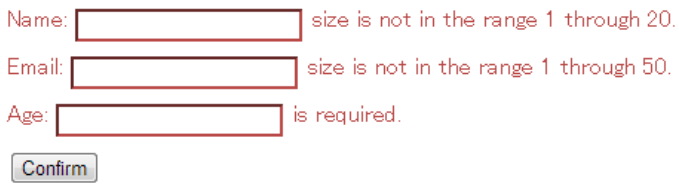
- ValidationMessages.properties

  `@NotNull`, `@Size`, `@Min`, `@Max`, `@Email` のエラーメッセージを変更する。

```
javax.validation.constraints.NotNull.message=is required.
# (1)
javax.validation.constraints.Size.message=size is not in the range {min}␣
↪through {max}.
javax.validation.constraints.Min.message=can not be less than {value}.
javax.validation.constraints.Max.message=can not be greater than {value}.
org.hibernate.validator.constraints.Email.message=is an invalid e-mail␣
↪address.
```

項番	説明
(1)	アノテーションに指定した属性値は、{**属性名**}で埋め込むことができる。

この設定を加えた状態で、すべての入力フィールドを未入力のままフォームを送信すると、以下のように変更したエラーメッセージが、表示される。



警告: Bean Validation 標準のアノテーションや Hibernate Validator 独自のアノテーションには `message` 属性に{**アノテーションの** `FQCN.message`}という値が設定されているため、

> アノテーションの `FQCN.message`=メッセージ

という形式でプロパティファイルにメッセージを定義すればよいが、すべてのアノテーションが、この形式になっているわけではないので、対象のアノテーションの Javadoc またはソースコードを確認すること。

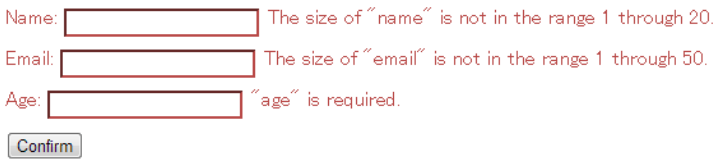
エラーメッセージに、フィールド名を含める場合は、以下のように、メッセージに`{0}`を加える。

- ValidationMessages.properties

  `@NotNull`、`@Size`、`@Min`、`@Max`、`@Email` のエラーメッセージを変更する。

```
javax.validation.constraints.NotNull.message="{0}" is required.
javax.validation.constraints.Size.message=The size of "{0}" is not in the␣
↪range {min} through {max}.
javax.validation.constraints.Min.message="{0}" can not be less than {value}.
javax.validation.constraints.Max.message="{0}" can not be greater than {value}
↪.
org.hibernate.validator.constraints.Email.message="{0}" is an invalid e-mail␣
↪address.
```

エラーメッセージは、以下のように変更される。



このままでは、フォームクラスのプロパティ名が表示されてしまい、ユーザーフレンドリではない。適切なフィールド名を表示したい場合は、**application-messages.properties** に

```
フォームのプロパティ名=表示するフィールド名
```

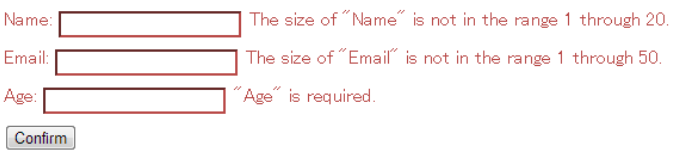
形式でフィールド名を定義すればよい。

これまでの例に、以下の設定を追加する。

- application-messages.properties

```
name=Name
email=Email
age=Age
```

エラーメッセージは、以下のように変更される。



注釈: `{0}`でフィールド名を埋め込めるのは、Bean Validation の機能ではなく、Spring の機能である。したがって、フィールド名変更の設定は、Spring 管理下の application-messages.properties(`ResourceBundleMessageSource`) に定義する必要がある。

ちなみに: Bean Validation 1.1 より、`ValidationMessages.properties` に指定するメッセージの中に Expression Language(以降、「EL 式」と呼ぶ) を使用する事ができるようになった。Hibernate Validator 5.x では、Expression Language 2.2 以上をサポートしている。

実行可能な EL 式のバージョンは、アプリケーションサーバのバージョンによって異なる。そのため、EL 式を使用する場合は、**アプリケーションサーバがサポートしている EL 式のバージョンを確認した上で使用すること。**

以下に、Hibernate Validator がデフォルトで用意している `ValidationMessages.properties` に定義されているメッセージを例に、EL 式の使用例を示す。

```
# ...
# (1)
javax.validation.constraints.DecimalMax.message  = must be less than $
↪{inclusive == true ? 'or equal to ' : ''}{value}
# ...
```

項番	説明
(1)	メッセージの中の「`${inclusive == true ? 'or equal to ' : ''}`」の部分が EL 式である。 上記のメッセージ定義から実際に生成されるメッセージのパターンは、 • must be less than or equal to {value} • must be less than {value} の 2 パターンとなる。(`{value}`の部分には、`@DecimalMax` アノテーションの `value` 属性に指定した値が埋め込まれる) 前者は`@DecimalMax` アノテーションの `inclusive` 属性に `true` を指定した場合 (又は指定しなかった場合)、後者は`@DecimalMax` アノテーションの `inclusive` 属性に `false` を指定した場合に生成される。 Bean Validation における EL 式の扱いについては、Hibernate Validator Reference Guide(Interpolation with message expressions) を参照されたい。

また、`ValidationMessages.properties` に指定するメッセージに `${validatedValue}`を使用することで、エラーメッセージにチェック対象の値を含むことができる。

以下に、 `${validatedValue}`の使用例を示す。

```
# ...
# (1)
javax.validation.constraints.Pattern.message = The value entered "$
↪{validatedValue}" is invalid.
# ...
```

項番	説明
(1)	上記のメッセージ定義から実際に生成されるメッセージは、 `${validatedValue}`の部分にフォームに入力した値が埋め込まれる。入力値に機密情報を含む場合、機密情報がメッセージに表示されないようにするため、 `${validatedValue}`を使用しないように注意すること。 詳細については、Hibernate Validator Reference Guide(Interpolation with message expressions)を参照されたい。

application-messages.properties に定義するメッセージ

ValidationMessages.properties でシステムが利用するデフォルトのメッセージを定義したが、画面によっては、デフォルトメッセージから変更したい場合が出てくる。

その場合、application-messages.properties に、以下の形式でメッセージを定義する。

```
アノテーション名．フォーム属性名．プロパティ名=対象のメッセージ
```

*ValidationMessages.properties に定義するメッセージ*の設定がある前提で、以下の設定で `email` と `age` フィールドのメッセージを上書きする。

- application-messages.properties

```
# override messages
# for email field
Size.userForm.email=The size of "{0}" must be between {2} and {1}.
# for age field
NotNull.userForm.age="{0}" is compulsory.
Min.userForm.age="{0}" must be greater than or equal to {1}.
Max.userForm.age="{0}" must be less than or equal to {1}.

# filed names
name=Name
email=Email
age=Age
```

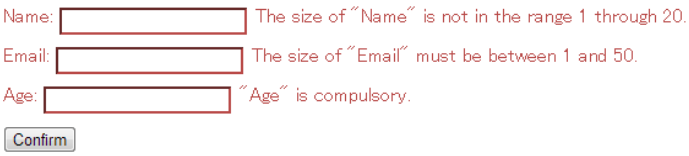
アノテーションの属性値は、`{1}`以降に埋め込まれる。なお、属性値のインデックス位置は、アノテーションの属性名のアルファベット順 (昇順) となる。

例えば、`@Size` のインデックス位置は、

- `{0}`：プロパティ名 (物理名又は論理名)
- `{1}`：`max` 属性の値
- `{2}`：`min` 属性の値

となる。仕様の詳細については SpringValidatorAdapter の JavaDoc を参照されたい。

エラーメッセージは以下のように変更される。



注釈: application-messages.properties のメッセージキーの形式は、これ以外にも用意されているが、デフォルトメッセージを一部上書きする目的で使用するのであれば、基本的に、アノテーション名．フォーム属性名．プロパティ名形式でよい。

4.1.3 How to extend

Bean Validation は標準で用意されているチェックルール以外に、独自ルール用アノテーションを作成する仕組みをもつ。

作成方法は大きく分けて、以下の観点で分かれる。

- 既存ルールの組み合わせ
- 新規ルールの作成

基本的には、以下の雛形を使用して、ルール毎にアノテーションを作成する。

```
package com.example.common.validation;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Constraint(validatedBy = {})
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface Xxx {
    String message() default "{com.example.common.validation.Xxx.message}";

    Class<?>[] groups() default {};
```

```
    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Xxx[] value();
    }
}
```

既存ルールを組み合わせた Bean Validation アノテーションの作成

システム共通で、

- 文字列は半角英数字の文字種に限定したい
- 数値は正の数に限定したい

または、ドメイン共通で、

- 「ユーザー ID」は、4 文字以上 20 文字以下の半角英字に制限したい
- 「年齢」は、1 歳以上 150 歳以下に制限したい

という制約がある場合を考える。

これらは既存ルールの`@Pattern`、`@Size`、`@Min`、`@Max` 等を組み合わせることでも実現できるが、

同じルールを複数の箇所で使用すると、設定内容が分散してしまい、メンテナンス性が悪化する。

複数のルールを組み合わせて一つのルールを作成することができる。独自アノテーションを作成すると、正規表現パターンや、最大値・最小値などの値を共通化できるだけでなく、エラーメッセージも共通化できるというメリットがある。これにより、再利用性や保守性が高まる。複数のルールの組み合わせではなくても、一つのルールの属性を特定するだけでも効果的である。

以下に、実装例を示す。

- 半角英数字の文字種に限定する`@Alphanumeric` アノテーションの実装例

```
package com.example.common.validation;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.ReportAsSingleViolation;
```

```
import javax.validation.constraints.Pattern;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Constraint(validatedBy = {}) // (1)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@ReportAsSingleViolation // (2)
@Pattern(regexp = "[a-zA-Z0-9]*") // (3)
public @interface AlphaNumeric {
    String message() default "{com.example.common.validation.AlphaNumeric.
↪message}"; // (4)

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        AlphaNumeric[] value();
    }
}
```

項番	説明
(1)	既存のアノテーションを利用して実装を行う場合、validatedBy は空にしておく必要がある。
(2)	エラーメッセージをまとめ、エラー時はこのアノテーションによるメッセージだけを変えるようにする。
(3)	このアノテーションにより使用されるルールを定義する。
(4)	エラーメッセージのデフォルト値を定義する。

- 正の数に限定する@NotNegative アノテーションの実装例

```
package com.example.common.validation;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.ReportAsSingleViolation;
import javax.validation.constraints.Min;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Constraint(validatedBy = {})
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@ReportAsSingleViolation
@Min(value = 0)
public @interface NotNegative {
    String message() default "{com.example.common.validation.NotNegative.
↪message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        NotNegative[] value();
    }
}
```

- 「ユーザー ID」のフォーマットを規定する@UserId アノテーションの実装例

```
package com.example.sample.domain.validation;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.ReportAsSingleViolation;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
```

```
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Constraint(validatedBy = {})
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@ReportAsSingleViolation
@Size(min = 4, max = 20)
@Pattern(regexp = "[a-z]*")
public @interface UserId {
    String message() default "{com.example.sample.domain.validation.UserId.
↪message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        UserId[] value();
    }
}
```

- 「年齢」の制限を規定する@Age アノテーションの実装例

```
package com.example.sample.domain.validation;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.ReportAsSingleViolation;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
```

```
@Constraint(validatedBy = {})
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@ReportAsSingleViolation
@Min(1)
@Max(150)
public @interface Age {
    String message() default "{com.example.sample.domain.validation.Age.
↪message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Age[] value();
    }
}
```

注釈: 1 つのアノテーションに複数のルールを設定した場合、それらの AND 条件が複合ルールとなる。Hibernate Validator では、OR 条件を実現するための`@ConstraintComposition` アノテーションが用意されている。詳細は、Hibernate Validator のドキュメントを参照されたい。

コレクション内の値を Bean Validation のアノテーションを使用してチェックする方法

複数選択可能な画面項目（チェックボックスや複数選択ドロップダウンなど）を扱う際は、フォームクラスで画面項目を `String` 等の基本型のコレクションとして扱うことが一般的である。Bean Validation の標準仕様ではコレクション内の各値に対しては Bean Validation のアノテーションを使いチェックすることができないが、Java SE 8 と Hibernate Validator の独自機能を使う、或いは Java SE 8 と Hibernate Validator の独自機能を使わない場合は画面項目の値に対するラッパークラスを作成しコレクションとして扱うことで、コレクション内の値を Bean Validation を使いチェックすることが可能になる。

ここでは、共通ライブラリが提供している入力値がコードリスト内に定義されたコード値であるかどうかチェックするアノテーション、`org.terasoluna.gfw.common.codelist.ExistInCodeList` を例に、コレクション内の String に対する入力チェックについて説明する。

複数選択可能な画面項目（チェックボックスや複数選択ドロップダウンなど）に`@ExistInCodeList` アノテーションを対応させるための実装方法を以下に示す。

- *Java SE 8 と Hibernate Validator 5.2+ による実装*　Java SE 8 と Hibernate Validator の独自機能を利

用し、String の List に付加できる独自のアノテーションを実装する方式。**後者と比べて簡単かつシンプルな実装で実現できるため、Java SE 8 と Hibernate Validator の独自機能が使用できる環境ではこちらの方式を推奨する。** また、この方法は将来的に Bean Validation の後続バージョンで標準化される予定である。

- *Java Bean を使った String のラッパークラスによる実装* Java Bean クラスでラップしたプロパティに対して @ExistInCodeList アノテーションを設定する方式。Java SE 8 と Hibernate Validator の独自機能を使用しないユーザ向け。Java SE 8 と Hibernate Validator の独自機能が使用できる環境では *Java SE 8 と Hibernate Validator 5.2+ による実装*を推奨する。

Java SE 8 と Hibernate Validator 5.2+ による実装

ここでは、共通ライブラリが提供している@ExistInCodeList をラップし、独自のアノテーションを作成することでコレクションに対応させる方法を紹介する。

Java SE 8 で java.lang.annotation.ElementType.TYPE_USE が追加された。これにより、従来のクラスやメソッド等の宣言に対してだけでなく、型全般（ローカル変数の型等）にアノテーションを付加できるようになり、Java SE 8 に対応した Hibernate Validator 5.2+ は、Collection, Map, Optional, などのパラメータ化された型に付与された制約アノテーションを読み取ることで、コレクション内の値に対するチェックを可能にしている。

Java SE 8 と Hibernate Validator 5.2+ を組み合わせることで、List<@NotNullForTypeArgument String>のように、リスト内の型指定部分に付加できるアノテーションを作成し、コレクション内の値の入力チェックを行うことができるようになる。詳細は、Hibernate Validator のドキュメント (Type argument constraints) を参照されたい。

共通ライブラリが提供する@ExistInCodeList は、Java SE 7 互換のため TYPE_USE に対応していないが、上記のようにリスト内の型指定部分に付加できる独自アノテーションを作成することで、コレクション内の値の入力チェックを行うことができるようになる。

主な手順は以下の通り。

- 「TYPE_USE」を使用し、型使用箇所に付加できる@ExistInCodeList を拡張したアノテーションを実装する。
- チェック対象にアノテーションを設定する。

複数項目設定可能な Role(String の List) に対する入力チェックを例に用いて説明する。

- 型使用箇所に付加できる@ExistInCodeListForTypeArgument の実装例

```
package com.example.common.validation;

import static java.lang.annotation.ElementType.TYPE_USE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.OverridesAttribute;
import javax.validation.Payload;
```

```
import javax.validation.ReportAsSingleViolation;
import org.terasoluna.gfw.common.codelist.ExistInCodeList;

@Documented
@Constraint(validatedBy = {})
@Target(TYPE_USE) // (1)
@Retention(RUNTIME)
@ReportAsSingleViolation
@ExistInCodeList(codeListId = "") // (2)
public @interface ExistInCodeListForTypeArgument {
    String message() default "{com.example.common.validation.
↪ExistInCodeListForTypeArgument.message}"; // (3)

    @OverridesAttribute(constraint = ExistInCodeList.class, name = "codeListId
↪") // (4)
    String codeListId();


    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target(TYPE_USE) // (1)
    @Retention(RUNTIME)
    @Documented
    @interface List {
        ExistInCodeListForTypeArgument[] value();
    }
}
```

項番	説明
(1)	TYPE_USE を設定し、このアノテーションが型使用箇所で付加できるようにする。
(2)	このアノテーションにより使用されるルール (@ExistInCodeList) を定義する。
(3)	エラーメッセージのデフォルト値を定義する。また、ValidationMessages.properties に任意のエラーメッセージを定義する。
(4)	@ExistInCodeList アノテーションの codeListId 属性をオーバーライドする。

- フォームクラス

```
package com.example.sample.app.validation;

import java.util.List;

import javax.validation.constraints.NotNull;

import com.example.common.validation.ExistInCodeListForTypeArgument;

public class SampleForm {
    @NotNull
    @Valid // (1)
    private List<@ExistInCodeListForTypeArgument(codeListId = "CL_ROLE")␣
↪String> roles; // (2)

    public List<String> getRoles() {
        return roles;
    }

    public void setRoles(List<String> roles) {
        this.roles = roles;
    }
}
```

項番	説明
(1)	List 内の要素値に対する入力チェックを有効にするために、 javax.validation.Valid アノテーションを付与する。
(2)	入力チェック対象となるコレクションの型指定部に対し て@ExistInCodeListForTypeArgument アノテーションを設定する。 アノテーションの codeListId パラメータにチェック元となるコードリストを指定する。

- JSP

```
<form:form modelAttribute="sampleForm">
    <!-- (1) -->
```

```
    <form:checkboxes path="roles" items="${CL_ROLE}"/>
    <form:errors path="roles*"/>
    <form:button>Submit</form:button>
</form:form>
```

項番	説明
(1)	<form:checkboxes>を実装する。

Java Bean を使った String のラッパークラスによる実装

ここで紹介する実装は Java SE 8 と Hibernate Validator の独自機能を使用しないユーザ向けである。Java SE 8 と Hibernate Validator の独自機能が使用できる環境では *Java SE 8 と Hibernate Validator 5.2+ による実装* を推奨する。

Java SE 8 と Hibernate Validator の独自機能を使用しない場合では前述したようなコレクション内の要素に対して Bean Validation のアノテーションを使用することができないため、Java Bean で `String` をラップし、ネストした Bean のプロパティに対して`@ExistInCodeList` を付加することによって入力チェックを行う。

ラッパークラスに対して Spring 提供のタグライブラリを使用する場合、フォームへバインドするためには文字列とラッパークラスとの型変換を実施する必要がある。これは Spring が提供している型変換の仕組み (Formatter) を利用して実装を行うことができる。

Formatter で `String` から `Role`、`Role` から `String` への型変換を追加することで、`List<String>`にした時と同様に、複雑な実装をすることなく <form:checkboxes>を使用した実装ができる。

主な手順は以下の通り。

- チェック対象に`@ExistInCodeList` アノテーションを設定する。
- 変換用インタフェースである `Formatter` クラスを実装したクラスを作成する。
- `ConversionServiceFactoryBean` を使用し、作成した `Formatter` を Spring に登録する。

また、<form:checkboxes>で正常に選択済みの項目を表示するためには、Formatter の実装に加えてラッパークラスの `toString` メソッドをオーバーライドする必要がある。

<form:checkboxes>は `items` 属性で指定されたコレクションの要素を選択項目として表示し、選択項目の値が `path` 属性で指定されたプロパティの値と一致する場合は、選択済みの項目として表示する。この一致性の判断には、プロパティが単項目の場合は Formatter が使用され、配列やコレクションの場合は指定されたプロパティの `toString` メソッドの結果が使用される。

正常に選択済みの項目として表示するためには、後述する例のようにラッパークラスで `toString` メソッドをオーバーライドし、ラップしている値の文字列を返却する必要がある。

注釈: 選択済みの判定方法の詳細については実際に判定を行う `org.springframework.web.servlet.tags.form.SelectedValueComparator` クラスの javadoc を参照されたい。

複数項目設定可能な `Role`(Java Bean の `List`) に対する入力チェックを例に用いて説明する。

- フォームクラス

```
package com.example.sample.app.validation;

import java.util.List;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

import com.example.sample.domain.model.Role;

public class SampleForm {
    @NotNull
    @Valid // (1)
    private List<Role> roles; // (2)

    public List<Role> getRoles() {
        return roles;
    }

    public void setRoles(List<Role> roles) {
        this.roles = roles;
    }
}
```

項番	説明
(1)	ネストした Bean の Bean Validation を有効にするために、`javax.validation.Valid` アノテーションを付与する。
(2)	`String` の `List` には`@ExistInCodeList` を付加することはできないが、Java Bean で `String` をラップすることで、ネストした Bean の `String` プロパティに対して`@ExistInCodeList` を付加することが出来るようになる。

- JavaBean クラス

```
package com.example.sample.domain.model

import org.terasoluna.gfw.common.codelist.ExistInCodeList;

public class Role {
    @ExistInCodeList(codeListId = "CL_ROLE") // (1)
    private String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    @Override // (2)
    public String toString() {
        return getValue();
    }
}
```

項番	説明
(1)	入力チェックを行うために Role クラスにラップしたプロパティに対して @ExistInCodeList アノテーションを設定し、codeListId にチェック元となるコードリストを指定する。
(2)	複数選択時の状態を正常に画面描画するためにオーバーライドし、ラップしている値の文字列を返却する。

型変換を行う Formatter クラスを実装し、Spring に登録する。

前述のとおり、入力チェックを行うために String を Role(Java Bean) でラップする必要がある。画面の入力（String）からラップした Role への変換と、その逆変換を行うために、Formatter による型変換の実装を行う。

型変換を追加することで、String と Role の相互変換が自動で行われる。Controller 側では Role の List、JSP 側では String の List として扱えるようになる。

- Formatter クラス

  String と Role の相互変換を行う Formatter の実装

```
package com.example.sample.app.validation.formatter;

import java.text.ParseException;
import java.util.Locale;

import org.springframework.format.Formatter;

import com.example.usermanagement.domain.model.Role;

public class RoleFormatter implements Formatter<Role> { //(1)

    @Override
    public String print(Role source, Locale locale) {
        return source.getValue();
    }

    @Override
    public Role parse(String source, Locale locale) throws ParseException {
        Role role = new Role();
```

```
        role.setValue(source);
        return role;
    }

}
```

項番	説明
(1)	インタフェース `org.springframework.format.Formatter<T>`を実装する。

- 独自の `Formatter` を適用するための Bean 定義

```
<!-- (1) -->
<bean id="conversionService"
    class="org.springframework.format.support.
↪FormattingConversionServiceFactoryBean">
    <property name="formatters">
        <list>
            <!-- (2) -->
            <bean class="com.example.sample.app.validation.formatter.
↪RoleFormatter" />
        </list>
    </property>
</bean>
```

```
<!-- (3) -->
<mvc:annotation-driven conversion-service="conversionService">
    <!-- omitted -->
</mvc:annotation-driven>
```

項番	説明
(1)	FormattingConversionServiceFactoryBean の Bean 定義を追加する。
(2)	作成した Formatter クラス (RoleFormatter) を設定する。
(3)	カスタマイズした型変換を使用するために、mvc:annotation-driven の conversion-service 属性に (1) で定義した Bean を設定する。

- JSP

```
<form:form modelAttribute="sampleForm">
    <!-- (1) -->
    <form:checkboxes path="roles" items="${CL_ROLE}"/>
    <form:errors path="roles*"/>
    <form:button>Submit</form:button>
</form:form>
```

項番	説明
(1)	List<String>にした時と同様に <form:checkboxes>を使用することができる。

新規ルールを実装した Bean Validation アノテーションの作成

javax.validation.ConstraintValidator インタフェースを実装し、その Validator を使用するアノテーションを作成することで、任意のルールを作成することができる。

用途としては、以下の 3 通りが挙げられる。

- 既存のルールの組み合わせでは表現できないルール
- 相関項目チェックルール

- 業務ロジックチェック

既存のルールの組み合わせでは表現できないルール

`@Pattern`、`@Size`、`@Min`、`@Max` 等を組み合わせても表現できないルールは、`javax.validation.ConstraintValidator` 実装クラスに記述する。

例として、ISBN(International Standard Book Number)-13 の形式をチェックするルールを挙げる。

- アノテーション

```
package com.example.common.validation;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Constraint(validatedBy = { ISBN13Validator.class }) // (1)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface ISBN13 {
    String message() default "{com.example.common.validation.ISBN13.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        ISBN13[] value();
    }
}
```

項番	説明
(1)	このアノテーションを使用したときに実行される `ConstraintValidator` を指定する。 複数指定することができる。

- Validator

```
package com.example.common.validation;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class ISBN13Validator implements ConstraintValidator<ISBN13, String> {
↪// (1)

    @Override
    public void initialize(ISBN13 constraintAnnotation) { // (2)
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context)
↪{ // (3)
        if (value == null) {
            return true; // (4)
        }
        return isISBN13Valid(value); // (5)
    }

    // This logic is written in http://en.wikipedia.org/wiki/International_
↪Standard_Book_Number
    static boolean isISBN13Valid(String isbn) {
        if (isbn.length() != 13) {
            return false;
        }
        int check = 0;
        try {
            for (int i = 0; i < 12; i += 2) {
                check += Integer.parseInt(isbn.substring(i, i + 1));
            }
            for (int i = 1; i < 12; i += 2) {
                check += Integer.parseInt(isbn.substring(i, i + 1)) * 3;
            }
            check += Integer.parseInt(isbn.substring(12));
        } catch (NumberFormatException e) {
            return false;
        }
        return check % 10 == 0;
    }
}
```

項番	説明
(1)	ジェネリクスのパラメータに、対象のアノテーションとフィールドの型を指定する。
(2)	`initialize` メソッドに、初期化処理を実装する。
(3)	`isValid` メソッドで入力チェック処理を実装する。
(4)	入力値が、`null` の場合は、正常とみなす。
(5)	ISBN-13 の形式のチェックを行う。

ちなみに: ファイルアップロードの *Bean Validation* の例も、ここに分類される。また共通ライブラリでは、この実装として*@ExistInCodeList* を用意している。

相関項目チェックルール

相関項目チェックで説明したように、Bean Validation によって複数のフィールドにまたがる相関項目チェックを実装できる。

Bean Validation で相関項目チェックルールを実装する場合は、汎用的なルールを対象とすることを推奨する。

以下では、「あるフィールドとその確認用フィールドの内容が一致すること」というルールを実現する例を挙げる。

ちなみに: 共通ライブラリでは、2 つのフィールドの内容を比較する相関項目チェックの実装として*@Compare* アノテーションを用意している。

@Compare アノテーションを利用することで、このルールをより簡単に実現することができる。 詳細は共通ライブラリのチェックルールの拡張方法を参照されたい。

ここでは、確認用フィールドの先頭に、「confirm」を付与する規約を設ける。

- アノテーション

相関項目チェック用のアノテーションはクラスレベルに付与できるようにする。

```
package com.example.common.validation;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Constraint(validatedBy = { ConfirmValidator.class })
@Target({ TYPE, ANNOTATION_TYPE }) // (1)
@Retention(RUNTIME)
public @interface Confirm {
    String message() default "{com.example.common.validation.Confirm.message}
↪";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    /**
     * Field name
     */
    String field(); // (2)

    @Target({ TYPE, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Confirm[] value();
    }
}
```

項番	説明
(1)	このアノテーションが、クラスまたはアノテーションにのみ付加できるように、対象を絞る。
(2)	アノテーションに渡すパラメータを定義する。

- Validator

```
package com.example.common.validation;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

import org.springframework.beans.BeanWrapper;
import org.springframework.beans.BeanWrapperImpl;
import org.springframework.util.ObjectUtils;
import org.springframework.util.StringUtils;

public class ConfirmValidator implements ConstraintValidator<Confirm, Object>
↪{
    private String field;

    private String confirmField;

    private String message;

    public void initialize(Confirm constraintAnnotation) {
        field = constraintAnnotation.field();
        confirmField = "confirm" + StringUtils.capitalize(field);
        message = constraintAnnotation.message();
    }

    public boolean isValid(Object value, ConstraintValidatorContext context) {
        BeanWrapper beanWrapper = new BeanWrapperImpl(value); // (1)
        Object fieldValue = beanWrapper.getPropertyValue(field); // (2)
        Object confirmFieldValue = beanWrapper.getPropertyValue(confirmField);
        boolean matched = ObjectUtils.nullSafeEquals(fieldValue,
                confirmFieldValue);
        if (matched) {
            return true;
        } else {
            context.disableDefaultConstraintViolation(); // (3)
            context.buildConstraintViolationWithTemplate(message)
                    .addPropertyNode(field).addConstraintViolation(); // (4)
            return false;
        }
    }

}
```

項番	説明
(1)	JavaBean のプロパティにアクセスする際に便利な `org.springframework.beans.BeanWrapper` を使用する。
(2)	`BeanWrapper` 経由で、フォームオブジェクトからプロパティ値を取得する。
(3)	デフォルトの `ConstraintViolation` オブジェクトの生成を無効にする。
(4)	独自 `ConstraintViolation` オブジェクトを生成する。 `ConstraintValidatorContext.buildConstraintViolationWithTemplate` で出力するメッセージを定義する。 `ConstraintViolationBuilder.addPropertyNode` でエラーメッセージを出力したいフィールド名を指定する。 詳細は、ConstraintValidatorContext の JavaDoc を参照されたい。

ちなみに: `ConstraintViolationBuilder.addPropertyNode` メソッドは、Bean Validation 1.1 から追加されたメソッドである。

Bean Validation 1.0 では `ConstraintViolationBuilder.addNode` というメソッドを使用していたが、Bean Validation 1.1 から非推奨の API となっている。

Bean Validation の非推奨 API については、Bean Validation API Document(Deprecated API) を参照されたい。

注釈: Spring Validator による相関項目チェックにて紹介したように、Bean Validation においても相関チェック対象の複数フィールドに対してエラー情報を設定する ことが可能である。

以下に、Bean Validation にて `password` フィールドと `confirmPassword` フィールドにスタイルを適用し、`password` フィールドのみにエラーメッセージを表示する例を示す。

```
// omitted
public class ConfirmValidator implements ConstraintValidator<Confirm, Object>
↪{
    private String field;

    private String confirmField;
```

```
    private String message;

    public void initialize(Confirm constraintAnnotation) {
        // omitted
    }

    public boolean isValid(Object value, ConstraintValidatorContext context) {
        // omitted
        if (matched) {
            return true;
        } else {
            context.disableDefaultConstraintViolation();

            //new ConstraintViolation to be generated for field
            context.buildConstraintViolationWithTemplate(message)
                    .addPropertyNode(field).addConstraintViolation();

            //new ConstraintViolation to be generated for confirmField
            context.buildConstraintViolationWithTemplate("") // (1)
                    .addPropertyNode(confirmField).addConstraintViolation();

            return false;
        }
    }

}
```

項番	説明
(1)	confirmPassword フィールドのエラーを登録する。この際、エラーメッセージに空文字を設定している。

この@Confirm アノテーションを使用して、前述の「パスワードリセット」処理を再実装すると、以下のようになる。

- フォームクラス

```
package com.example.sample.app.validation;

import java.io.Serializable;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import com.example.common.validation.Confirm;
```

```
@Confirm(field = "password") // (1)
public class PasswordResetForm implements Serializable {
    private static final long serialVersionUID = 1L;

    @NotNull
    @Size(min = 8)
    private String password;

    private String confirmPassword;

    // omitted geter/setter
}
```

項番	説明
(1)	クラスレベルに@Confirm アノテーションを付与する。 これにより ConstraintValidator.isValid の引数にはフォームオブジェクトが渡る。

- Controller クラス

  Validator のインジェクションおよび@InitBinder による Validator の追加は、不要になる。

```
package com.example.sample.app.validation;

import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("password")
public class PasswordResetController {

    @ModelAttribute
    public PasswordResetForm setupForm() {
        return new PasswordResetForm();
    }

    @RequestMapping(value = "reset", method = RequestMethod.GET, params =
↪"form")
    public String resetForm() {
        return "password/resetForm";
    }

    @RequestMapping(value = "reset", method = RequestMethod.POST)
```

```
    public String reset(@Validated PasswordResetForm form, BindingResult␣
↪result) {
        if (result.hasErrors()) {
            return "password/resetForm";
        }
        return "redirect:/password/reset?complete";
    }

    @RequestMapping(value = "reset", method = RequestMethod.GET, params =
↪"complete")
    public String resetComplete() {
        return "password/resetComplete";
    }
}
```

業務ロジックチェック

業務ロジックチェックは、基本的にはドメイン層の *Service* で実装し、結果メッセージは `ResultMessages` オブジェクトに格納することを推奨している。

したがって、通常画面の上部などに表示されることを想定している。

一方で、「入力されたユーザー名が既に登録済みかどうか」など、対象の入力フィールドに対する業務ロジックエラーメッセージを、フィールドの横に表示したい場合もある。このような場合は、Validator クラスに Service クラスをインジェクションして、業務ロジックチェックを実行し、その結果を、`ConstraintValidator.isValid` の結果に使用すればよい。

「入力されたユーザー名が既に登録済みかどうか」を Bean Validation で実現する例を示す。

- Service クラス

  実装クラス (UserServiceImpl) は割愛する。

```
package com.example.sample.domain.service.user;

public interface UserService {

    boolean isUnusedUserId(String userId);

    // omitted other methods
}
```

- アノテーション

```
package com.example.sample.domain.validation;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
```

```
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Constraint(validatedBy = { UnusedUserIdValidator.class })
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface UnusedUserId {
    String message() default "{com.example.sample.domain.validation.
↪UnusedUserId.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        UnusedUserId[] value();
    }
}
```

- Validator クラス

```
package com.example.sample.domain.validation;

import javax.inject.Inject;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

import org.springframework.stereotype.Component;

import com.example.sample.domain.service.user.UserService;

@Component // (1)
public class UnusedUserIdValidator implements
                                    ConstraintValidator<UnusedUserId, String> {

    @Inject // (2)
    UserService userService;

    @Override
    public void initialize(UnusedUserId constraintAnnotation) {
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
```

```
        if (value == null) {
            return true;
        }

        return userService.isUnusedUserId(value); // (3)
    }

}
```

項番	説明
(1)	Validator クラスをコンポーネントスキャンの対象にする。 パッケージが Bean 定義ファイルの`<context:component-scan base-package="..." />`の設定に含まれている必要がある。
(2)	呼び出す Service クラスを、インジェクションする。
(3)	業務ロジックの結果を返却する。`isValid` メソッド名で業務ロジックを記述せず、かならず Service に処理を委譲すること。

Method Validation

Bean Validation によってメソッドの実引数と返り値の妥当性を確認する方法を説明する。説明のために、本節ではこの方法を Method Validation と呼ぶ。防衛的プログラミングを行う場合などでは、Controller 以外のクラスでメソッドの入出力を確認する必要がある。このとき、Bean Validation ライブラリを利用すれば、Controller で使用した Bean Validation の制約アノテーションを再利用できる。

アプリケーションの設定

Spring Framework が提供する Method Validation を使用する場合は、Spring Framework から提供されている `org.springframework.validation.beanvalidation.MethodValidationPostProcessor` クラスを Bean 定義する必要がある。

`MethodValidationPostProcessor` を定義する Bean 定義ファイルは、Method Validation を使用する箇所によって異なる。

ここでは、本ガイドラインが推奨するマルチプロジェクト環境で Method Validation を使用するための設定例を示す。

- アプリケーション層用のプロジェクト (projectName-web)
- ドメイン層用のプロジェクト (projectName-domain)

の両プロジェクトの設定を変更する必要がある。

- projectName-domain/src/main/resources/META-INF/spring/projectName-domain.xml

```
<!-- (1) -->
<bean id="validator"
      class="org.springframework.validation.beanvalidation.
↪LocalValidatorFactoryBean"/>

<!-- (2) -->
<bean class="org.springframework.validation.beanvalidation.
↪MethodValidationPostProcessor">
    <property name="validator" ref="validator" />
</bean>
```

- projectName-web/src/main/resources/META-INF/spring/spring-mvc.xml

```
<!-- (3) -->
<mvc:annotation-driven validator="validator">
    <!-- ... -->
</mvc:annotation-driven>

<!-- (4) -->
<bean class="org.springframework.validation.beanvalidation.
↪MethodValidationPostProcessor">
    <property name="validator" ref="validator" />
</bean>
```

項番	説明
(1)	LocalValidatorFactoryBean を Bean 定義する。
(2)	MethodValidationPostProcessor を Bean 定義し、ドメイン層のクラスのメソッドに対して Method Validation が実行されるようにする。 validator プロパティには、(1) で定義した Bean を指定する。
(3)	<mvc:annotation-driven>要素の validator 属性に、(1) で定義した Bean を指定する。 この設定がない場合は (1) で作成したものとは異なる Validator インスタンスが生成されてしまう。
(4)	MethodValidationPostProcessor を Bean 定義し、アプリケーション層のクラスのメソッドに対して Method Validation が実行されるようにする。 validator プロパティには、(1) で定義した Bean を指定する。

ちなみに: `LocalValidatorFactoryBean` は、Bean Validation(Hibernate Validator) が提供する `Validator` クラスと Spring Framework を連携するためのラッパー `Validator` オブジェクトを生成するためのクラスである。

このクラスによって生成されたラッパー `Validator` を使用することで、Spring Framework が提供するメッセージ管理機能 (`MessageSource`) や DI コンテナなどとの連携が行えるようになる。

ちなみに: Spring Framework では、DI コンテナで管理されている Bean のメソッド呼び出しに対する Method Validation の実行を、AOP の仕組みを利用して行っている。

`MethodValidationPostProcessor` は、Method Validation を実行するための AOP を適用するためのクラスである。

注釈: 上記例では、各 Bean の `validator` プロパティに対して、同じ `Validator` オブジェクト (インスタンス) を設定しているが、これは必ずしも必須ではない。ただし、特に理由がない場合は、同じオブジェクト (インスタンス) を設定しておくことを推奨する。

Method Validation 対象のメソッドにするための定義方法

メソッドに Method Validation を適用するには、対象のメソッドを含むことを示したアノテーションをクラスレベルに、Bean Validation の制約アノテーションをメソッドと仮引数にそれぞれ指定する必要がある。

「アプリケーションの設定」を行っただけでは、Method Validation を実行する AOP は適用されない。Method Validation を実行する AOP を適用するためには、インタフェース又はクラスに`@ org.springframework.validation.annotation.Validated` アノテーションを付与する必要がある。

ここでは、インタフェースに対してアノテーションを指定する方法を紹介する。

```
package com.example.domain.service;

import org.springframework.validation.annotation.Validated;

@Validated // (1)
public interface HelloService {
    // ...
}
```

項番	説明
(1)	Method Validation の対象となるインタフェースに、`Validated` アノテーションを指定する。 上記例では、`HelloService` インタフェースの実装メソッドに対して、Method Validation を実行する AOP が適用される。

ちなみに: `@Validated` アノテーションの `value` 属性にグループインタフェースを指定することで、指定したグループに属する Validation のみ実行する事も可能である。

また、メソッドレベルに `Validated` アノテーションを付与することで、メソッド毎にバリデーショングループを切り替える事も可能な仕組みとなっている。

バリデーショングループについては、「バリデーションのグループ化」を参照されたい。

次に、Bean Validation の制約アノテーションをメソッドや仮引数へ指定する方法を説明する。具体的には、

- メソッドの引数
- メソッドの引数に指定された JavaBean のフィールド

に対して Bean Validation の制約アノテーションを、

- メソッドの返り値
- メソッドの返り値として返却する JavaBean のフィールド

に対して Bean Validation の制約アノテーションを指定する。

以下に、具体的な指定方法について説明する。以降の説明では、インタフェースにアノテーションを指定する方法を紹介する。

まず、メソッドのシグネチャとして基本型 (プリミティブやプリミティブラッパ型など) を使用するメソッドに対して、制約アノテーションを指定する方法について説明する。

```
package com.example.domain.service;

import org.springframework.validation.annotation.Validated;

import javax.validation.constraints.NotNull;

@Validated
public interface HelloService {

    // (2)
```

```
    @NotNull
    String hello(@NotNull /* (1) */ String message);

}
```

項番	説明
(1)	Bean Validation の制約アノテーションをメソッドの引数アノテーションとして指定する。 `@NotNull` は `message` という引数が Null 値を許可しないことを意味する制約である。引数に Null 値が指定された場合、`javax.validation.ConstraintViolationException` が発生する。
(2)	Bean Validation の制約アノテーションをメソッドアノテーションとして指定する。 上記例では、返り値が Null 値にならないことを示しており、返り値として Null 値が返却された場合、`javax.validation.ConstraintViolationException` が発生する。

次に、メソッドのシグネチャとして JavaBean を使用するメソッドに対して、Bean Validation の制約アノテーションを指定する方法について説明する。

ここでは、インタフェースに対してアノテーションを指定する方法を紹介する。

注釈: ポイントは、`@javax.validation.Valid` アノテーションを指定するという点である。以下に、サンプルコード使って指定方法を詳しく説明する。

Service インタフェース

```
package com.example.domain.service;

import org.springframework.validation.annotation.Validated;

import javax.validation.constraints.NotNull;

@Validated
public interface HelloService {

    @NotNull // (3)
    @Valid   // (4)
    HelloOutput hello(@NotNull /* (1) */ @Valid /* (2) */ HelloInput input);

}
```

項番	説明
(1)	Bean Validation の制約アノテーションをメソッドの引数アノテーションとして指定する。 `input` という引数 (JavaBean) が Null 値を許可しない事を示しており、引数に Null 値が指定された場合は、`javax.validation.ConstraintViolationException` が発生する。
(2)	`@javax.validation.Valid` アノテーションをメソッドの引数アノテーションとして指定する。 `@Valid` アノテーションを付与する事で、引数の JavaBean のフィールドに指定した Bean Validation の制約アノテーションが有効となる。JavaBean に指定された制約を満たさない場合は `javax.validation.ConstraintViolationException` が発生する。
(3)	Bean Validation の制約アノテーションをメソッドアノテーションとして指定する。 返り値の JavaBean が Null 値にならないことを示しており、返り値として Null 値が返却された場合は例外が発生する。
(4)	`@Valid` アノテーションをメソッドアノテーションとして指定する。 `@Valid` アノテーションを付与する事で、返り値の JavaBean のフィールドに指定した Bean Validation の制約アノテーションが有効となる。JavaBean に指定された制約を満たさない場合は `javax.validation.ConstraintViolationException` が発生する。

以下に JavaBean の実装サンプルを紹介する。

基本的には、Bean Validation の制約アノテーションを指定するだけだが、JavaBean が更に JavaBean をネストしている場合は注意が必要になる。

Input 用の JavaBean

```
package com.example.domain.service;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import java.util.Date;

public class HelloInput {

    @NotNull
    @Past
    private Date visitDate;

    @NotNull
    private String visitMessage;

    private String userId;
```

```
    // ...

}
```

Output 用の JavaBean

```
package com.example.domain.service;

import com.example.domain.model.User;

import java.util.Date;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;

public class HelloOutput {

    @NotNull
    @Past
    private Date acceptDate;

    @NotNull
    private String acceptMessage;

    @Valid // (5)
    private User user;

    // ...

}
```

Output 用の JavaBean 内でネストしている JavaBean

```
package com.example.domain.model;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import java.util.Date;

public class User {

    @NotNull
    private String userId;

    @NotNull
    private String userName;

    @Past
    private Date dateOfBirth;

    // ...
```

```
}
```

項番	説明
(5)	ネストした JavaBean に指定している Bean Validation の制約アノテーションを有効にする場合は、`@Valid` アノテーションをフィールドアノテーションとして指定する。 `@Valid` アノテーションを付与する事で、ネストした JavaBean のフィールドに指定した Bean Validation の制約アノテーションが有効となる。ネストした JavaBean に指定された制約を満たさない場合は `javax.validation.ConstraintViolationException` が発生する。

制約違反時の例外ハンドリング

制約に違反した場合、`javax.validation.ConstraintViolationException` が発生する。

`ConstraintViolationException` が発生した場合、スタックトレースから発生したメソッドは特定できるが、具体的な違反内容が特定できない。

違反内容を特定するためには、`ConstraintViolationException` をハンドリングしてログ出力を行う例外ハンドリングクラスを作成するとよい。

以下の例外ハンドリングクラスの作成例を示す。

```
package com.example.app;

import javax.validation.ConstraintViolationException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class ConstraintViolationExceptionHandler {

    private static final Logger logger = LoggerFactory.
↪getLogger(ConstraintViolationExceptionHandler.class);

    // (1)
    @ExceptionHandler
    public String handleConstraintViolationException(ConstraintViolationException␣
↪e){
        // (2)
        if (log.isErrorEnabled()) {
            log.error("ConstraintViolations[\n{}\n]", e.getConstraintViolations());
        }
```

```
        return "common/error/systemError";
    }

}
```

項番	説明
(1)	ConstraintViolationException をハンドリングするための@ExceptionHandler メソッドを作成する。 メソッドの引数として、ConstraintViolationException を受け取るようにする。
(2)	メソッドの引数で受け取った ConstraintViolationException が保持している違反内容 (ConstraintViolation の Set) をログに出力する。

注釈: @ControllerAdvice アノテーションの詳細については「*@ControllerAdvice の実装*」を参照されたい。

警告: ConstraintViolation#getMessage メソッドを使用することでエラーメッセージを取得することができるが、Spring の機能によるメッセージ補完は行われないため、エラーメッセージに {0} でフィールド名を埋め込むことはできない。

代わりに、フィールド名は ConstraintViolation#getPropertyPath メソッドで取得することが可能である。

Spring の機能によるメッセージ補完については、*ValidationMessages.properties に定義するメッセージ* の Note を参照されたい。

ConstraintViolation の詳細については、Hibernate Validator のリファレンスを参照されたい。

4.1.4 Appendix

Hibernate Validator が用意する入力チェックルール

Hibernate Validator は Bean Validation で定義されたアノテーションに加え、独自の検証用アノテーションを提供している。

検証に使用することができるアノテーションのリストは、こちらを参照されたい。

Bean Validation のチェックルール

Bean Validation の標準アノテーション (javax.validation.*) を以下に示す。

詳細は、Bean Validation specification の 7 章を参照されたい。

アノテーション	対象の型	説明	使用例
@NotNull	任意	対象のフィールドが、null でないことを検証する。	`@NotNull` `private String id;`

@Null	任意	対象のフィールドが、null であることを検証する。
		(例：グループ検証での使用)
			`@Null(groups=`
			`↪{Update.class})`
			`private String id;`

@Pattern	String	対象のフィールドが正規表現にマッチするかどうか
		(Hibernate Validator 実装では、任意の CharSequence インタフェースの実装クラスにも適用可能)
			`@Pattern(regexp =`
			`↪"[0-9]+")`
			`private String tel;`

@Min	BigDecimal, BigInteger, byte, short, int, long およびラッパー (Hibernate Validator 実装では、任意の Number の継承クラス,CharSequence インタフェースの実装クラスにも適用可能。ただし、文字列が数値表現の場合に限る。)	値が、最小値以上であるかどうかを検証する。	@Max 参照

@Max	BigDecimal, BigInteger, byte, short, int, long およびラッパー
	(Hibernate Validator 実装では任意の Number の継承クラス,CharSequence インタフェースの実装クラスにも適用可能。ただし、文字列が数値表現の場合に限る。)
		値が、最大値以下であるかどうかを検証する。	`@Min(1)`
			`@Max(100)`
			`private int␣`
			`↪quantity;`

Continued on next page

表 4.3 – continued from previous page

アノテーション	対象の型	説明	使用例
@DecimalMin	BigDecimal, BigInteger, String, byte, short, int, long およびラッパー (Hibernate Validator 実装では任意の Number の継承クラス,CharSequence インタフェースの実装クラスにも適用可能)	Decimal 型の値が、最小値以上であるかどうかを検証する。 inclusive = false を指定する事で、最小値より大きいかどうかを検証するように動作を変更する事ができる。	@DecimalMax 参照

@DecimalMax	BigDecimal, BigInteger, String, byte, short, int, long およびラッパー (Hibernate Validator 実装では任意の Number の継承クラス,CharSequence インタフェースの実装クラスにも適用可能)	Decimal 型の値が、最大値以下であるかどうかを検証する。
		inclusive = false を指定する事で、最大値より小さいかどうかを検証するように動作を変更する事ができる。
			`@DecimalMin("0.0")`
			`@DecimalMax("99999.99")`
			`private BigDecimal price;`

@Size	String(文字列の長さ), Collection(要素のサイズ), Map(要素のサイズ), Array(配列の長さ) (Hibernate Validator 実装では、任意の CharSequence インタフェースの実装クラスにも適用可能)	要素の長さ (要素のサイズ) が min と max の間のサイズか検証する。 min と max は省略可能であるが、デフォルトは min=0,max=Integer.MAX_VALUE となる。	`@Size(min=4, max=64)` `private String password;`
@Digits	BigDecimal, BigInteger, String, byte, short, int, long およびラッパー	値が指定された範囲内の数値であるかチェックする。 integer に最大整数の桁を指定し、fraction に最大小数桁を指定する。	`@Digits(integer=6, fraction=2)` `private BigDecimal price;`
@AssertTrue	boolean,Boolean	対象のフィールドが true であることを検証する (例：規約に同意したかどうか)	`@AssertTrue` `private boolean checked;`
@AssertFalse	boolean,Boolean	対象のフィールドが false であることを検証する	`@AssertFalse` `private boolean checked;`

Continued on next page

表 4.3 – continued from previous page

アノテーション	対象の型	説明	使用例
`@Future`	`Date`, `Calendar`(Hibernate Validator 実装では Joda-Time のクラスにも適用可能)	未来日付であるか検証する。	`@Future` `private Date eventDate;`
`@Past`	`Date`, `Calendar`(Hibernate Validator 実装では Joda-Time のクラスにも適用可能)	過去日付であるか検証する。	`@Past` `private Date eventDate;`

`@Valid`	任意の非プリミティブ型	関連付けられているオブジェクトについて、再帰的に検証を行う。	`@Valid`
			`private List<Employer> employers;`
			
			`@Valid`
			`private Dept dept;`

ちなみに: `@DecimalMin` と `@DecimalMax` アノテーションの `inclusive` 属性は、Bean Validation 1.1 から追加された属性である。

`inclusive` 属性のデフォルト値には `true`(指定した閾値と同じ値を許容する) が指定されており、Bean Validation 1.0 との互換性が保たれている。

警告: `@Size` アノテーションでは、サロゲートペアと呼ばれる char 型 2 つ（32 ビット）で表される文字に対する考慮がされていない。

サロゲートペアを含む文字列をチェック対象とした場合、カウントした文字数が実際の入力文字数より多くカウントされる可能性があるため注意すること。

サロゲートペアを含む文字列の文字列長については、 文字列長の取得 を参照されたい。

注釈: Hibernate Validator が提供する `TimeProvider` を実装することで、`@Past`、`@Future` の基準となる日付を変更することが出来る。なお、実装した `TimeProvider` を適用するには、`LocalValidatorFactoryBean` の継承クラスを作成し、`postProcessConfiguration` メソッドをオーバーライドすれば良い。`TimeProvider` を実装したクラスの例に関しては、Hibernate Validator Reference Guide(Time providers for @Future and @Past) を参照されたい。

Hibernate Validator のチェックルール

Hibernate Validator の代表的なアノテーション (`org.hibernate.validator.constraints.*`) を以下に示す。

詳細は、Hibernate Validator 仕様を参照されたい。

アノテーション	対象の型	説明	使用例
@CreditCardNumber	任意の CharSequence インタフェースの実装クラスに適用可能	Luhn アルゴリズムでクレジットカード番号が妥当かどうかを検証する。使用可能な番号かどうかをチェックするわけではない。 ignoreNonDigitCharacters = true を指定する事で、数字以外の文字を無視して検証する事ができる。	`@CreditCardNumber` `private String cardNumber;`
@Email	任意の CharSequence インタフェースの実装クラスに適用可能	E-mail アドレスとして妥当であること検証する。	`@Email` `private String email;`
@URL	任意の CharSequence インタフェースの実装クラスに適用可能	URL として妥当であること検証する。java.net.URL のコンストラクタを使用して文字列検証を行っており、URL として妥当とされるプロトコルは JVM がサポートするプロトコル (http,https,file,jar など) に依存する。	`@URL` `private String url;`
@NotBlank	任意の CharSequence インタフェースの実装クラスに適用可能	null、空文字 (""), 空白のみでないことを検証する。	`@NotBlank` `private String userId;`
@NotEmpty	Collection、Map、Array、任意の CharSequence インタフェースの実装クラスに適用可能	null、または空でないことを検証する。 @NotNull + @Min(1) の組み合わせでチェックする場合は、@NotEmpty を使用すること。	`@NotEmpty` `private String password;`

警告: E-mail の形式は RFC2822 で定義されているが、@Email は厳密に RFC2822 に準拠していることをチェックするものではない。

例えばマルチバイト文字（全角文字）を含んでいても@Email でのチェックをパスすることが確認されて

いる。また、実際に利用されている Email アドレスも、必ずしも RFC2822 に厳密に準拠しているわけではない。

これらの注意点を考慮した上で、利用・サポートする SMTP サーバなどによって適切なルールでの入力チェックを実装することを推奨する。実装の際は、既存ルールを組み合わせた *Bean Validation* アノテーションの作成を参照されたい。

ちなみに: `@URL` にて、JVM がサポートしていないプロトコルについても妥当として検証したい場合、Hibernate から提供されている `org.hibernate.validator.constraintvalidators.RegexpURLValidator` を使用する。当該クラスは`@URL` アノテーションに対応する Validator クラスで、URL 形式であるかを正規表現で検証しており、JVM がサポートしていないプロトコルについても妥当として検証可能である。

- アプリケーション全体の`@URL` のチェックルールを変更してもよい場合には、JavaDoc に記載されているように、XML にて Validator クラスを `RegexpURLValidator` に変更する。
- 一部の項目だけに正規表現による検証を適用し、`@URL` はデフォルトのルールを使用したい場合には、新規アノテーション、および `RegexpURLValidator` と同様の検証を行う `javax.validation.ConstraintValidator` 実装クラスを作成し、必要な項目に作成したアノテーションによる検証を適用する。

など、用途に応じた適用を行えばよい。

XML によるチェックルール変更の詳細については Hibernate のリファレンスを、新規アノテーションの作成方法については、新規ルールを実装した *Bean Validation* アノテーションの作成をそれぞれ参照されたい。

Hibernate Validator が用意するデフォルトメッセージ

hibernate-validator-<version>.jar 内の org/hibernate/validator に、ValidationMessages.properties のデフォルト値が定義されている。

```
javax.validation.constraints.AssertFalse.message = must be false
javax.validation.constraints.AssertTrue.message  = must be true
javax.validation.constraints.DecimalMax.message  = must be less than ${inclusive
↪== true ? 'or equal to ' : ''}{value}
javax.validation.constraints.DecimalMin.message  = must be greater than $
↪{inclusive == true ? 'or equal to ' : ''}{value}
javax.validation.constraints.Digits.message      = numeric value out of bounds (<
↪{integer} digits>.<{fraction} digits> expected)
javax.validation.constraints.Future.message      = must be in the future
javax.validation.constraints.Max.message         = must be less than or equal to
↪{value}
javax.validation.constraints.Min.message         = must be greater than or equal
↪to {value}
javax.validation.constraints.NotNull.message     = may not be null
javax.validation.constraints.Null.message        = must be null
javax.validation.constraints.Past.message        = must be in the past
```

```
javax.validation.constraints.Pattern.message     = must match "{regexp}"
javax.validation.constraints.Size.message        = size must be between {min} and
↪{max}

org.hibernate.validator.constraints.CreditCardNumber.message        = invalid␣
↪credit card number
org.hibernate.validator.constraints.EAN.message                     = invalid
↪{type} barcode
org.hibernate.validator.constraints.Email.message                   = not a well-
↪formed email address
org.hibernate.validator.constraints.Length.message                  = length must␣
↪be between {min} and {max}
org.hibernate.validator.constraints.LuhnCheck.message               = The check␣
↪digit for ${validatedValue} is invalid, Luhn Modulo 10 checksum failed
org.hibernate.validator.constraints.Mod10Check.message              = The check␣
↪digit for ${validatedValue} is invalid, Modulo 10 checksum failed
org.hibernate.validator.constraints.Mod11Check.message              = The check␣
↪digit for ${validatedValue} is invalid, Modulo 11 checksum failed
org.hibernate.validator.constraints.ModCheck.message                = The check␣
↪digit for ${validatedValue} is invalid, ${modType} checksum failed
org.hibernate.validator.constraints.NotBlank.message                = may not be␣
↪empty
org.hibernate.validator.constraints.NotEmpty.message                = may not be␣
↪empty
org.hibernate.validator.constraints.ParametersScriptAssert.message  = script␣
↪expression "{script}" didn't evaluate to true
org.hibernate.validator.constraints.Range.message                   = must be␣
↪between {min} and {max}
org.hibernate.validator.constraints.SafeHtml.message                = may have␣
↪unsafe html content
org.hibernate.validator.constraints.ScriptAssert.message            = script␣
↪expression "{script}" didn't evaluate to true
org.hibernate.validator.constraints.URL.message                     = must be a␣
↪valid URL

org.hibernate.validator.constraints.br.CNPJ.message                 = invalid␣
↪Brazilian corporate taxpayer registry number (CNPJ)
org.hibernate.validator.constraints.br.CPF.message                  = invalid␣
↪Brazilian individual taxpayer registry number (CPF)
org.hibernate.validator.constraints.br.TituloEleitoral.message      = invalid␣
↪Brazilian Voter ID card number
```

共通ライブラリが用意する入力チェックルール

共通ライブラリでは、独自の検証用アノテーションを提供している。ここでは、共通ライブラリで提供しているアノテーションを使用した入力チェックルールの指定方法について説明する。

terasoluna-gfw-common のチェックルール

terasoluna-gfw-common が提供するアノテーション (`org.terasoluna.gfw.common.codelist.*`) を以下に示す。

アノテーション	対象の型	説明	使用例
`@ExistInCodeList`	`Character` `CharSequence` の実装クラス (`String`, `StringBuilder` など) `Number` の継承クラス (`Integer`, `Long` など) **5.4.2 から追加**	値がコードリストに含まれているかどうかを検証する。	*@ExistInCodeList* 参照

terasoluna-gfw-codepoints のチェックルール

terasoluna-gfw-codepoints が提供するアノテーション (`org.terasoluna.gfw.common.codepoints.*`) を以下に示す。なお、`terasoluna-gfw-codepoints` はバージョン 5.1.0.RELEASE 以上で利用することができる。

アノテーション	対象の型	説明	使用例
`@ConsistOf`	`CharSequence` の実装クラス (`String`, `StringBuilder` など)	チェック対象の文字列が指定したコードポイント集合に全て含まれるかどうかを検証する。	*@ConsistOf* 参照

terasoluna-gfw-validator のチェックルール

terasoluna-gfw-validator が提供するアノテーション (`org.terasoluna.gfw.common.validator.constraints.*`) を以下に示す。なお、`terasoluna-gfw-validator` はバージョン 5.1.0.RELEASE 以上で利用することができる。

アノテーション	対象の型	説明	使用例
@ByteMin	CharSequence の実装クラス (String, StringBuilder など)	値のバイト長が最小値以上であることを検証する。 **[アノテーションの属性]** long value - バイト長の最小値を指定する。 String charset - 値をバイトシーケンスに符号化する際に使用する文字セットを指定する。デフォルト値は UTF-8。	`@ByteMin(value = 1, charset = "Shift_JIS")` `private String id;`
@ByteMax	CharSequence の実装クラス (String, StringBuilder など)	値のバイト長が最大値以下であることを検証する。 **[アノテーションの属性]** long value - バイト長の最大値を指定する。 String charset - 値をバイトシーケンスに符号化する際に使用する文字セットを指定する。デフォルト値は UTF-8。	`@ByteMax(100)` `private String id;`

Continued on next page

表 4.4 – continued from previous page

アノテーション	対象の型	説明	使用例
`@ByteSize`	`CharSequence` の実装クラス (`String`, `StringBuilder` など)	値のバイト長が最小値と最大値の範囲内であることを検証する。 （**5.4.2** から追加） `@ByteMin` と `@ByteMax` を組み合わせて使う場合は、こちらを使うことを推奨する。 **[アノテーションの属性]** `long min` - バイト長の最小値を指定する。デフォルト値は `0`。 `long max` - バイト長の最大値を指定する。デフォルト値は `Long.MAX_VALUE`。 `String charset` - 値をバイトシーケンスに符号化する際に使用する文字セットを指定する。デフォルト値は `UTF-8`。	`@ByteSize(min = 1, max = 100)` `private String id;`

Continued on next page

表 4.4 – continued from previous page

アノテーション	対象の型	説明	使用例

@Compare	Comparable インタフェースの実装クラスをプロパティにもつ任意の JavaBean に適用可能	指定したプロパティの値の比較結果が正しいことを検証する。
		
		[アノテーションの属性]
		String left - オブジェクト内の比較元としたいプロパティ名を指定する。検証エラーとなった場合は、このプロパティにメッセージを表示される。
		String right - オブジェクト内の比較先としたいプロパティ名を指定する。
		org.terasoluna.gfw.common.validator.constraints.Compare.Operator operator - 比較方法を示す列挙型 Operator の値を指定する。指定可能な値は以下の通り。
		• EQUAL : left = right である
		• NOT_EQUAL : left != right である
		• GREATER_THAN : left > right である
		• GREATER_THAN_OR_EQUAL : left >= right である
		• LESS_THAN : left < right である
		• LESS_THAN_OR_EQUAL : left <= right である
		
		NOT_EQUAL は、terasoluna-gfw-validator 5.3.2.RELEASE 以上で利用可能な値である。
		
		boolean requireBoth - left 属性と right 属性で指定したフィールドの両方が入力されている（null でない）必要が
			メールアドレスと確認用に入力したメールアドレスが一致することをチェックし、フォーム全体のエラーメッセージとして表示する場合、以下のように実装する。
			`@Compare(left = "email", right = "confirmEmail", operator = Compare.Operator.EQUAL, requireBoth = true, node = Compare.Node.ROOT_BEAN) public class UserRegisterForm { private String email; private String confirmEmail; }`
			
			期間の開始日と終了日が両方入力された場合は、開始日が終了日以前であることをチェックし、期間の開始日にエラーメッセージを表示する場合、以下のように実装する。
			`@Compare(left = "form", right = "to", operator = Compare.Operator.LESS_THAN_OR_EQUAL) public class Period { private Date from; private Date to; }`

表 4.4 – continued from previous page

アノテーション	対象の型	説明	使用例

注釈: 相関項目チェックにおける入力必須について

単項目チェックにおいては、入力フィールドが入力されている（ null でない）かどうかは @NotNull を併用してチェックすればよい。しかし、相関項目チェックにおいては、「どちらか一方でも入力した場合は、もう一方の入力を強制する」といった、 @NotNull の併用だけでは実現できない場合がある。このため、@Compare では、チェック対象の入力必須を制御する requireBoth 属性を提供しており、これを併用して要件に応じたチェックを実装することができる。

なお、入力フィールドが未入力の場合に null がバインドされる場合のみ、 requireBoth 属性が利用できる。Spring MVC では文字列の入力フィールドに未入力の状態でフォームを送信した場合、デフォルトでは、フォームオブジェクトに null ではなく、空文字がバインドされることに注意しなければならない。 文字列フィールドが未入力の場合に、空文字ではなく、null をフォームオブジェクトにバインドするには、文字列フィールドが未入力の場合に *null* をバインドするを参照されたい。

期間の開始日が終了日以前であることのチェックを例に、想定されるチェック要件と設定の例を以下に示す。

チェック要件	設定例

from と to がともに必須で、from と to の比較チェックを行う。	from と to に @NotNull を付与し、requireBoth 属性はデフォルト値（ false）を使用する。
	`@Compare(left = "from", right = "to", operator = Compare.Operator.LESS_THAN_OR_EQUAL)`
	`public class Period {`
	`  @NotNull`
	`  LocalDate from;`
	`  @NotNull`
	`  LocalDate to;`
	`}`

from だけ必須だが、 to も入力された時は比較チェックする。	from にだけ @NotNull を付与し、requireBoth 属性はデフォルト値（ false）を使用する。
	`@Compare(left = "from", right = "to", operator = Compare.Operator.LESS_THAN_OR_EQUAL)`
	`public class Period {`
	`  @NotNull`
	`  LocalDate from;`
	`  LocalDate to;`
	`}`

Continued on next page

表 4.5 – continued from previous page

チェック要件	設定例

from と to がともに必須ではなく、 from と to が両方入力された時だけ比較チェックする。どちらか一方だけが入力された場合は比較チェックを行わない。	@NotNull は付与せず、 requireBoth 属性はデフォルト値（ false）を使用する。
	`@Compare(left = "from", right = "to", operator = Compare.Operator.LESS_THAN_OR_EQUAL)`
	`public class Period {`
	`  LocalDate from;`
	`  LocalDate to;`
	`}`

from と to がともに必須ではないが、 from か to のどちら一方でも入力した場合は、必ず両方入力して比較チェックを行う。	@NotNull は付与せず、 requireBoth 属性に true を設定する。
	`@Compare(left = "from", right = "to", operator = Compare.Operator.LESS_THAN_OR_EQUAL, requireBoth = true)`
	`public class Period {`
	`  LocalDate from;`
	`  LocalDate to;`
	`}`

共通ライブラリのチェックルールの適用方法

以下の手順で、共通ライブラリのチェックルールを適用する。

使用したいチェックルールに応じて、依存ライブラリを追加する。terasoluna-gfw-validator を追加する例を以下に示す。

```
<dependencies>
    <dependency>
        <groupId>org.terasoluna.gfw</groupId>
        <artifactId>terasoluna-gfw-validator</artifactId>
    </dependency>
</dependencies>
```

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。

次に、*ValidationMessages.properties に定義するメッセージ*で説明したように ValidationMessages.properties に、アノテーションに対応する任意のメッセージ定義を追加する。

```
# (1)
org.terasoluna.gfw.common.validator.constraints.ByteMin.message = must be greater␣
↪than or equal to {value} bytes
org.terasoluna.gfw.common.validator.constraints.ByteMax.message = must be less␣
↪than or equal to {value} bytes
org.terasoluna.gfw.common.validator.constraints.ByteSize.message = must be between
↪{min} and {max} bytes
org.terasoluna.gfw.common.validator.constraints.Compare.message = invalid␣
↪combination of {left} and {right}
```

項番	説明
(1)	アノテーションごとにメッセージ定義を追加する。アノテーションの属性値は、プレースホルダ（{属性名}の形式）を使用してメッセージの中に埋め込むことができる。

最後に、基本的な単項目チェックで説明したように、JavaBean のプロパティにアノテーションを付与する。

注釈: Bean Validation では、アノテーションの属性値の不正により検証が実行できない場合、javax.validation.ValidationException がスローされる。スタックトレースに出力される原因を参照し、属性値を適切な値に修正すること。

詳細は、Bean Validation specification の 9 章を参照されたい。

共通ライブラリのチェックルールの拡張方法

共通ライブラリで提供しているチェックルールを利用して、任意のルールを作成することができる。

以下では、相関項目チェックルールで独自に実装した@Confirm アノテーションを、共通ライブラリで提供しているチェックルールを利用して作成する例を紹介する。

既存ルールを組み合わせた *Bean Validation* アノテーションの作成で説明したように、@Compare を利用して@Confirm アノテーションを作成する。

```
package com.example.sample.domain.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.OverridesAttribute;
import javax.validation.Payload;

import org.terasoluna.gfw.common.validator.constraints.Compare;

@Documented
@Constraint(validatedBy = {})
@Target({ TYPE, ANNOTATION_TYPE }) // (1)
@Retention(RUNTIME)
@Compare(left = "", right = "", operator = Compare.Operator.EQUAL, requireBoth =␣
↪true) // (2)
public @interface Confirm {

    @OverridesAttribute(constraint = Compare.class, name = "message") // (3)
    String message() default "{com.example.sample.domain.validation.Confirm.
↪message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @OverridesAttribute(constraint = Compare.class, name = "left") // (4)
    String field();

    @OverridesAttribute(constraint = Compare.class, name = "right") // (5)
    String confirmField();

    @Documented
    @Target({ TYPE, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @interface List {
        Confirm[] value();
```

```
    }
}
```

項番	説明
(1)	このアノテーションを付与できる場所を、クラスまたはアノテーションに限定する。
(2)	`@Compare` アノテーションの `operator` 属性に `Compare.Operator.EQUAL`(同値であること) を指定する。どちらか一方が未入力の場合はエラーとするため、`requireBoth` 属性に `true` を指定する。
(3)	`@Compare` アノテーションの `message` 属性をオーバーライドし、エラー時に `message` 属性に指定したメッセージが使用されるようにする。
(4)	`@Compare` アノテーションの `left` 属性をオーバーライドし、属性名を `field` に変更する。
(5)	同様に `right` 属性をオーバーライドし、属性名を `confirmField` に変更する。

注釈: 「既存ルールを組み合わせた *Bean Validation* アノテーションの作成」では`@ReportAsSingleViolation` を付与する方法を紹介しているが、`@ReportAsSingleViolation` を付与するとラップされた`@Compare` のエラーメッセージは使用されず、`@Confirm` のエラーメッセージのみが表示される。`@Confirm` はフォームオブジェクトに対する入力チェックであるため、エラーメッセージはフォームオブジェクトに割り当てられ、実際に表示したい `field` 属性に指定したフィールドには割り当てられない。

これを回避するためには、`@ReportAsSingleViolation` を付与せず、`@Confirm` の `message` 属性で`@Compare` の `message` 属性をオーバーライドする必要がある。これにより、`@Compare` のルールに従い `left` 属性（つまり`@Confirm` の `field` 属性）に`@Confirm` のエラーメッセージを割り当てることができるようになる。

相関項目チェックルールで実装したアノテーションの代わりに、上記で作成したアノテーションを使用する。

```
package com.example.sample.app.validation;
```

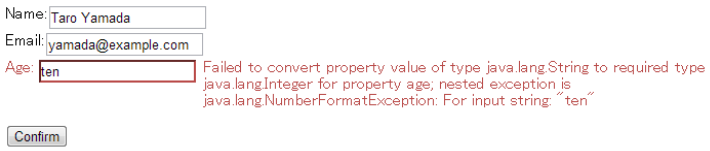
```
import java.io.Serializable;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import com.example.common.validation.Confirm;

@Confirm(field = "password", confirmField = "confirmPassword") // (1)
public class PasswordResetForm implements Serializable {
    private static final long serialVersionUID = 1L;

    @NotNull // (2)
    @Size(min = 8)
    private String password;

    @NotNull // (3)
    private String confirmPassword;

    // omitted geter/setter
}
```

項番	説明
(1)	クラスレベルに@Confirm アノテーションを付与する。
(2)	password フィールドが null の場合は@Confirm の検証はパスするため、null チェックは@NotNull アノテーションを付与して行う。
(3)	同様に confirmPassword フィールドにも、@NotNull アノテーションを付与する。

型のミスマッチ

フォームオブジェクトの String 以外のフィールドに対して、変換不可能な値を送信した場合は org.springframework.beans.TypeMismatchException がスローされる。

「新規ユーザー登録」処理の例では「Age」フィールドは Integer で定義されているが、このフィールドに対して整数に変換できない値を入力すると、以下のようなエラーメッセージが表示される。



例外の原因がそのまま表示されてしまい、エラーメッセージとしては不適切である。型がミスマッチの場合の

エラーメッセージは、org.springframework.context.MessageSource が読み込む properties ファイル (application-messages.properties) に定義できる。

以下のルールで、エラーメッセージを定義すればよい。

メッセージキー	メッセージ内容	用途
typeMismatch	型ミスマッチエラーのデフォルトメッセージ	システム全体のデフォルト値
typeMismatch.対象の FQCN	特定の型ミスマッチエラーのデフォルトメッセージ	システム全体のデフォルト値
typeMismatch.フォーム属性名.プロパティ名	特定のフォームのフィールドに対する型ミスマッチエラーのメッセージ	画面毎に変更したいメッセージ

application-messages.properties に以下の定義を行った場合、

```
# typemismatch
typeMismatch="{0}" is invalid.
typeMismatch.int="{0}" must be an integer.
typeMismatch.double="{0}" must be a double.
typeMismatch.float="{0}" must be a float.
typeMismatch.long="{0}" must be a long.
typeMismatch.short="{0}" must be a short.
typeMismatch.java.lang.Integer="{0}" must be an integer.
typeMismatch.java.lang.Double="{0}" must be a double.
typeMismatch.java.lang.Float="{0}" must be a float.
typeMismatch.java.lang.Long="{0}" must be a long.
typeMismatch.java.lang.Short="{0}" must be a short.
typeMismatch.java.util.Date="{0}" is not a date.

# filed names
name=Name
email=Email
age=Age
```

エラーメッセージは、次のように変更される。



*application-messages.properties に定義するメッセージ*で説明したように、{0}でフィールド名を埋めることができる。

基本的にデフォルトメッセージは定義しておくこと。

ちなみに: メッセージキーのルールの詳細は、DefaultMessageCodesResolver の Javadoc を参照されたい。

文字列フィールドが未入力の場合に null をバインドする

これまで説明してきたように、Spring MVC では文字列の入力フィールドに未入力の状態でフォームを送信した場合、デフォルトでは、フォームオブジェクトに `null` ではなく、空文字がバインドされる。

この場合、「未入力は許容するが、入力された場合は 6 文字以上であること」という要件を、既存のアノテーションで満たすことができない。

文字列フィールドが未入力の場合に、空文字ではなく、`null` をフォームオブジェクトにバインドするには、以下のように `org.springframework.beans.propertyeditors.StringTrimmerEditor` を使用すればよい。

```
@Controller
@RequestMapping("xxx")
public class XxxController {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        // bind empty strings as null
        binder.registerCustomEditor(String.class, new StringTrimmerEditor(true));
    }

    // omitted ...
}
```

この設定により、Controller 毎に空文字を `null` とみなすかどうかを設定できる。

プロジェクト全体で空文字を `null` にしたい場合は、プロジェクト共通設定として*@ControllerAdvice* で設定すればよい。

ちなみに: Spring Framework 4.0 より追加された@ControllerAdvice アノテーションの属性について

`@ControllerAdvice` アノテーションの属性を指定することで、`@ControllerAdvice` が付与されたクラスで実装したメソッドを適用する Controller を柔軟に指定できるように改善されている。属性の詳細については、*@ControllerAdvice の属性*を参照されたい。

```
@ControllerAdvice
public class XxxControllerAdvice {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        // bind empty strings as null
```

```
        binder.registerCustomEditor(String.class, new StringTrimmerEditor(true));
    }

    // omitted ...
}
```

この設定を行った場合は、フォームオブジェクトの文字列フィールドに設定される空文字がすべて null になる。
したがって、必須チェックに、かならず@NotNull が必要であることに注意しないといけない。

Native to Ascii を行わないメッセージの読み込み

Native to Ascii を行わずに Bean Validation のメッセージ（ValidationMessage.properties）を読み込む方法紹介する。

日本語メッセージを Native to Ascii せずに直接扱いたい場合、Spring の MessageSource と連携すると簡単に実装することができる。

以下のように定義すると、MessageSource の機能で読み込まれたメッセージが Hibernate Validator の中で使用されるようになる。

- Bean 定義

  *-domain.xml

```
<!-- (1) -->
<bean id="validator" class="org.springframework.validation.beanvalidation.
↪LocalValidatorFactoryBean">
    <property name="validationMessageSource">
        <!-- (2) -->
        <bean class="org.springframework.context.support.
↪ResourceBundleMessageSource">
            <property name="basenames">
                <list>
                    <value>ValidationMessages</value> <!-- (3) -->
                </list>
            </property>
            <property name="defaultEncoding" value="UTF-8" />
        </bean>
    </property>
</bean>

<!-- (4) -->
<bean class="org.springframework.validation.beanvalidation.
↪MethodValidationPostProcessor">
    <property name="validator" ref="validator" />
</bean>
```

  spring-mvc.xml

```
<!-- (5) -->
<mvc:annotation-driven validator="validator">
    <!-- ommited -->
</mvc:annotation-driven>

<!-- (6) -->
<bean class="org.springframework.validation.beanvalidation.
↪MethodValidationPostProcessor">
    <property name="validator" ref="validator" />
</bean>
```

項番	説明
(1)	LocalValidatorFactoryBean を Bean 定義する。
(2)	MessageSource の定義。ここでは ResourceBundleMessageSource を使用する。
(3)	ApplicationContext に読み込ませるリソースバンドルを指定する。
(4)	*Method Validation* を利用する際には、MethodValidationPostProcessor の validator プロパティに (1) で定義した Bean を指定する。 Method Validation を利用しない場合、この Bean 定義は不要である。
(5)	<mvc:annotation-driven>要素の validator 属性に、(1) で定義した Bean を指定する。
(6)	(4) と同様である。

注釈: MessageSource の機能を利用することで、プロパティファイルの配置先がクラスパス直下に制限されなくなる。また、複数のプロパティファイルを指定することもできるようになる。

OS コマンドインジェクション対策

ここでは、セキュリティ脆弱性の一種である OS コマンドインジェクションとその対策について説明する。

OS コマンドインジェクションとは

OS コマンドインジェクションとは、アプリケーション内でユーザー入力文字列からコマンド実行文字列を組み立てている箇所がある場合に、ユーザー入力文字列の中に悪意のあるコマンドが送り込まれると、コンピュータを不正に操られてしまう問題である。

ちなみに: 詳細は、OWASP の解説ページなどを参照されたい。

Java では `ProcessBuilder` クラスや、`Runtime` クラスの `exec` メソッドを用いてコマンドを実行する際に、実行するコマンドとして以下のものを利用する場合に、OS コマンドインジェクションが発生する可能性がある。

- `/bin/sh`（Unix 系の場合）や `cmd.exe`（Windows の場合）
- ユーザーが入力した文字列

以下では、`/bin/sh` を利用する場合に OS コマンドインジェクションが発生する例を示す。

```
ProcessBuilder pb = new ProcessBuilder("/bin/sh", "-c", script); // (1)
Process p = pb.start();
```

項番	説明
(1)	例えば、`script` に」exec.sh ; cat /etc/passwd」 が入ると、文字列中のセミコロンが`/bin/sh` により区切り文字として解釈され、」cat /etc/passwd」が実行される。 そのため、標準出力の扱い方によっては`/etc/passwd` が出力される可能性がある。

警告: ScriptEngine や ScriptTemplateViewResolver の利用について

Java SE 6 より追加された `ScriptEngine` や、Spring Framework 4.2 より追加された `ScriptTemplateViewResolver` では、JVM 上で別言語（`Ruby` や `Python` など）を使用することができる。

これらの機能を利用して別言語のコードを実行する場合、コードの書き方によっては OS コマンドインジェクションが発生する可能性があるため、利用には十分注意すること。

対策方法

OS コマンドインジェクションを起こさないためには、可能な限り外部プロセスの実行を避ける。ただし諸般の事情により外部プロセスの実行がどうしても必要な場合、以下の対策を行った上で外部プロセス実行を実装すること。

- 極力、`/bin/sh`（Unix 系の場合）や `cmd.exe`（Windows の場合）を使用したコマンド実行を行わない

- ユーザーにより入力された文字が、アプリケーションとして許可されたものであるか、ホワイトリスト方式を用いてチェックする

以下では、ユーザーが入力したコマンドと引数が指定された文字列で構成されているかをホワイトリスト方式でチェックするルールの例を示す。

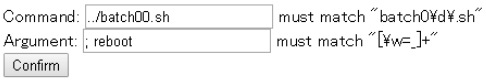
```
@Pattern(regexp = "batch0\\d\\.sh") // (1)
private String cmdStr;

@Pattern(regexp = "[\\w=_]+")  // (2)
private String arg;
```

項番	説明
(1)	コマンドとして batch0X.sh（X は 0 から 9 までの半角数字）のみ許可するルールを指定する。
(2)	引数として、無害な文字である半角英数字（\w)、」=」 、」_」 から構成された文字列のみ許可するルールを指定する。

注釈: この例では、コマンドや引数にパスが含まれないようなルールとすることで、ディレクトリトラバーサルを起こさないようにしている。

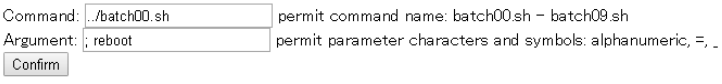
@Pattern を利用する場合、@Pattern に指定された正規表現がそのままエラーメッセージとして出力され、以下の点でメッセージとしては不適切である。

- エラーの意味が不明確となり、ユーザに優しくない
- 脆弱性への対策のためのロジックが利用者に露呈してしまう



エラーの意味を明確にし、かつ、ロジックを隠蔽するために、application-messages.properties に適切なメッセージを定義する。メッセージの定義方法については、*application-messages.properties に定義するメッセージ*を参照されたい。

```
Pattern.cmdForm.cmdStr = permit command name: batch00.sh - batch09.sh
Pattern.cmdForm.arg = permit parameter characters and symbols: alphanumeric, =, _
```



4.2 例外ハンドリング

本ガイドラインで作成する、Web アプリケーションの例外ハンドリング指針について説明する。

4.2.1 Overview

本節では、Spring MVC 配下の処理で発生する例外のハンドリングについて説明する。説明対象は、以下の通りである。

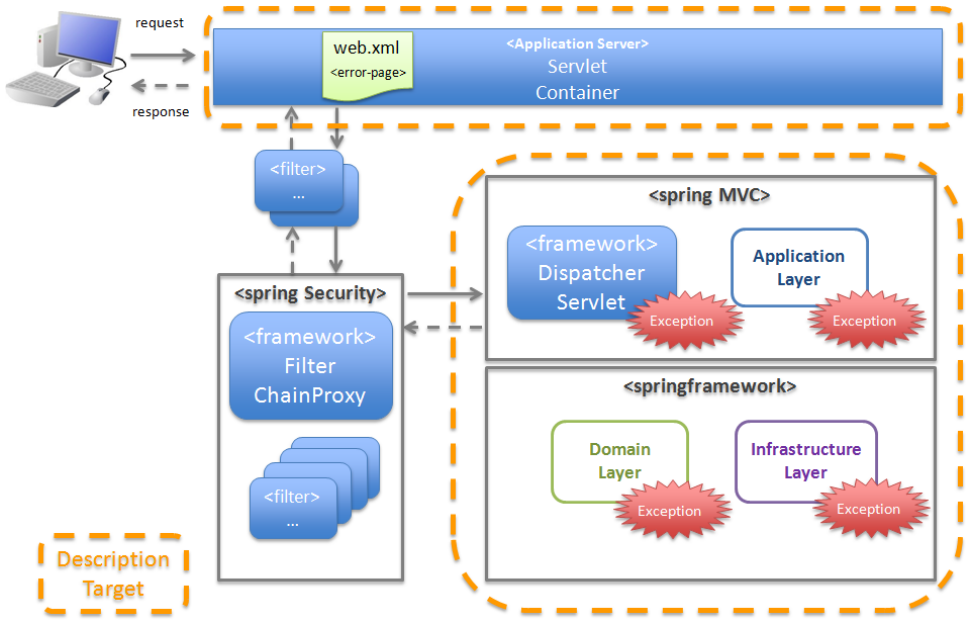


図 4.1　図-説明対象

1. 例外の分類
2. 例外のハンドリング方法

例外の分類

アプリケーション実行時に発生する例外は、以下 3 つに分類される。

表 4.6　表-アプリケーション実行時に発生する例外の分類

項番	分類	説明	例外の種類
(1)	オペレータの再操作 (入力値の変更など) によって、発生原因が解消できる例外	オペレータの再操作によって、発生原因が解消できる例外は、アプリケーションコードで例外をハンドリングし、例外処理を行う。	1. ビジネス例外 2. 正常稼働時に発生するライブラリ例外
(2)	オペレータの再操作によって、発生原因が解消できない例外	オペレータの再操作によって、発生原因が解消できない例外は、フレームワークで例外をハンドリングし、例外処理を行う。	1. システム例外 2. 予期しないシステム例外 3. 致命的なエラー
(3)	クライアントからの不正リクエストにより発生する例外	クライアントからの不正リクエストにより発生する例外は、フレームワークで例外をハンドリングし、例外処理を行う。	1. リクエスト不正時に発生するフレームワーク例外

注釈: 誰が、例外を意識する必要があるのか？

- (1) はアプリケーション開発者が意識する例外となる。
- (2) と (3) はアプリケーションアーキテクトが意識する例外となる。

例外のハンドリング方法

アプリケーション実行時に発生する例外は、以下 4 つの方法でハンドリングを行う。

ハンドリング方法毎のハンドリングフローの詳細は、例外ハンドリングの基本フローを参照されたい。

表 4.7: 表-例外のハンドリング方法

項番	ハンドリング方法	説明	例外ハンドリングのパターン
(1)	アプリケーションコードにて、`try-catch` を使い、例外ハンドリングを行う。	リクエスト (Controller のメソッド) 単位に、例外をハンドリングする場合に使用する。 詳細は、リクエスト単位で *Controller* クラスがハンドリングする場合の基本フローを参照されたい。	1. ユースケースの一部やり直し *(途中からのやり直し)* を促す場合
(2)	`@ExceptionHandler` アノテーションを使い、アプリケーションコードで例外ハンドリングを行う。	ユースケース (Controller) 単位に、例外をハンドリングする場合に使用する。 詳細は、ユースケース単位で *Controller* クラスがハンドリングする場合の基本フローを参照されたい。	1. ユースケースのやり直し *(先頭からのやり直し)* を促す場合
(3)	フレームワークから提供されている HandlerExceptionResolver の仕組みを使い、例外ハンドリングを行う。	サーブレット単位に、例外をハンドリングする場合に使用する。 HandlerExceptionResolver は、`<mvc:annotation-driven>`を指定した際に、自動的に、登録されるクラスと、共通ライブラリから提供している `SystemExceptionResolver` を使用する。 詳細は、サーブレット単位でフレームワークがハンドリングする場合の基本フローを参照されたい。	1. システム、またはアプリケーションが、正常な状態でない事を通知する場合 2. リクエスト内容が、不正であることを通知する場合

Continued on next page

表 4.7 – continued from previous page

項番	ハンドリング方法	説明	例外ハンドリングのパターン
(4)	サーブレットコンテナの error-page 機能を使い、例外ハンドリングを行う。	致命的なエラー、Spring MVC 管理外で発生する例外をハンドリングする場合に使用する。 詳細は、*Web* アプリケーション単位でサーブレットコンテナがハンドリングする場合の基本フローを参照されたい。	1. 致命的なエラーが発生したことを検知する場合 2. プレゼンテーション層 *(JSP* など*)* で、例外が発生したことを通知する場合

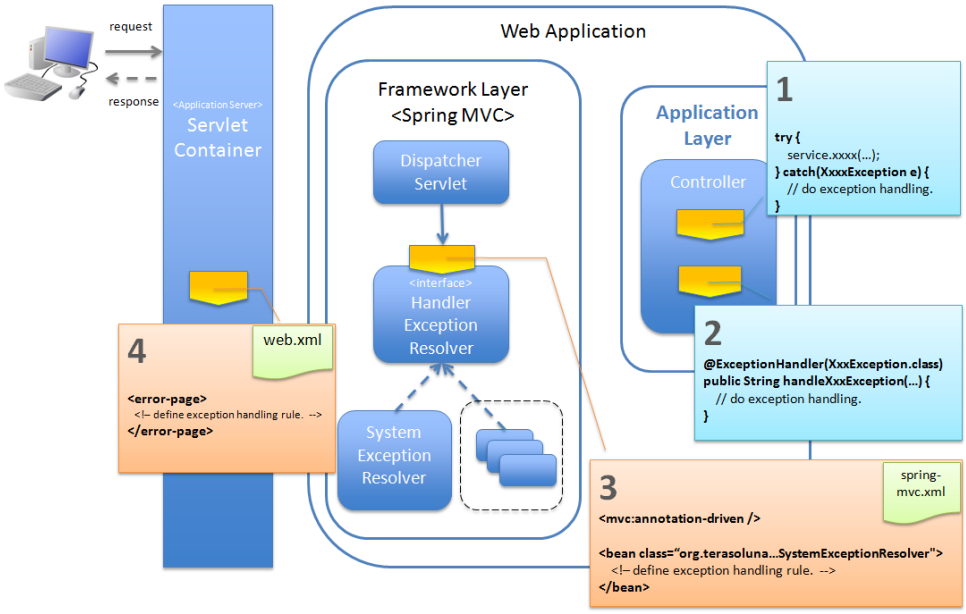


図 4.2　図-例外のハンドリング方法

注釈: 誰が例外ハンドリングを行うのか？

- (1) と (2) はアプリケーション開発者が設計・実装する。
- (3) と (4) はアプリケーションアーキテクトが設計・設定する。

注釈: 自動的に登録される **HandlerExceptionResolver** について

<mvc:annotation-driven> を指定した際に、自動的に登録される HandlerExceptionResolver の役割は、以下の通りである。

優先順は、以下の並び順の通りとなる。

項番	クラス (優先順位)	役割
(1)	ExceptionHandlerExceptionResolver (order=0)	`@ExceptionHandler` アノテーションが付与されている Controller クラスのメソッドを呼び出し、例外ハンドリングを行うためのクラス。 No.2 のハンドリング方法を実現するために必要なクラス。
(2)	ResponseStatusExceptionResolver (order=1)	クラスアノテーションとして、`@ResponseStatus` が付与されている例外をハンドリングするためのクラス。 `@ResponseStatus` に指定されている値で、`HttpServletResponse#sendError(int sc, String msg)` が呼び出される。
(3)	DefaultHandlerExceptionResolver (order=2)	Spring MVC 内で発生するフレームワーク例外を、ハンドリングするためのクラス。 フレームワーク例外に対応する HTTP レスポンスコードの値で、`HttpServletResponse#sendError(int sc)` が呼び出される。 設定される HTTP レスポンスコードの詳細は、*DefaultHandlerExceptionResolver で設定される HTTP レスポンスコードについて*を参照されたい。

注釈: 共通ライブラリから提供している SystemExceptionResolver の役割は？

<mvc:annotation-driven> を指定した際に、自動的に登録される HandlerExceptionResolver によって、ハンドリングされない例外をハンドリングするためのクラスである。そのため優先順は、DefaultHandlerExceptionResolver の後になるように設定する。

注釈: Spring Framework 3.2 より追加された@ControllerAdvice アノテーションについて

`@ControllerAdvice` の登場により、サーブレット単位で、`@ExceptionHandler` を使った例外ハンドリングを行えるようになった。`@ControllerAdvice` アノテーションが付与されたクラスで、`@ExceptionHandler` アノテーションを付与したメソッドを定義すると、サーブレット内のすべての Controller に適用される。以前のバージョンで同じことを実現する場合、`@ExceptionHandler` アノテー

ションが付与されたメソッドを、Controller のベースクラスのメソッドとして定義し、各 Controller でベースクラスを継承する必要があった。

Spring Framework 4.0 より追加された@ControllerAdvice アノテーションの属性について

`@ControllerAdvice` アノテーションの属性を指定することで、`@ControllerAdvice` が付与されたクラスで実装したメソッドを適用する Controller を柔軟に指定できるように改善されている。属性の詳細については、*@ControllerAdvice の属性*を参照されたい。

注釈: @ControllerAdvice アノテーションの使いどころ

1. サーブレット単位で行う例外ハンドリングに対して、View 名と、HTTP レスポンスコードの解決以外の処理が必要な場合。（View 名と HTTP レスポンスコードの解決のみでよい場合は、`SystemExceptionResolver` で対応できる ）
2. サーブレット単位で行う例外ハンドリングに対して、エラー応答用のレスポンスデータを JSP などのテンプレートエンジンを使わずに、エラー用のモデル（JavaBeans）を、JSON や XML 形式にシリアライズして生成したい場合（AJAX や、REST 用の Controller を作成する際の、エラーハンドリングとして使用する）。

4.2.2 Detail

1. 例外の種類
2. 例外ハンドリングのパターン
3. 例外ハンドリングの基本フロー

例外の種類

アプリケーション実行時に発生する例外は、以下 6 種類に分類される。

表 4.8　表-例外の種類

項番	例外の種類	説明
(1)	ビジネス例外	ビジネスルールの違反を検知したことを通知する例外
(2)	正常稼働時に発生するライブラリ例外	フレームワーク、およびライブラリ内で発生する例外のうち、システムが、正常稼働している時に発生する可能性のある例外
(3)	システム例外	システムが、正常稼働している時に、発生してはいけない状態を検知したことを通知する例外
(4)	予期しないシステム例外	システムが、正常稼働している時には発生しない非検査例外
(5)	致命的なエラー	システム (アプリケーション) 全体に影響を及ぼす、致命的な問題が発生していることを通知するエラー
(6)	リクエスト不正時に発生するフレームワーク例外	フレームワークが、リクエスト内容の不正を検知したことを通知する例外

ビジネス例外

ビジネスルールの違反を検知したことを通知する例外 。

本例外は、ドメイン層のロジック内で発生させる。

アプリケーションとして想定される状態なので、システム運用者による対処は、不要である。

- 旅行を予約する際に予約日が期限を過ぎている場合
- 商品を注文する際に在庫切れの場合
- etc …

注釈: 該当する例外クラス

- `org.terasoluna.gfw.common.exception.BusinessException` (共通ライブラリから提供しているクラス)。
- 細かくハンドリングする必要がある場合は、BusinessException を継承した例外クラスを作成すること。
- 共通ライブラリで用意しているビジネス例外クラスで、要件を満たせない場合は、プロジェクト毎にビジネス例外クラスを作成すること。

正常稼働時に発生するライブラリ例外

フレームワーク、およびライブラリ内で発生する例外のうち、 **システムが、正常稼働している時に発生する可能性のある例外。**
フレームワーク、およびライブラリ内で発生する例外とは、Spring Framework や、その他のライブラリ内で発生する例外クラスを対象とする。
アプリケーションとして想定される状態なので、システム運用者による対処は、不要である。

- 複数のオペレータによって、同じデータを同時に更新しようとした場合に、発生する楽観排他例外や、悲観排他例外。
- 複数のオペレータによって、同じデータを同時に登録しようとした場合に、発生する一意制約違反例外。
- etc …

注釈: 該当する例外クラスの例

- `org.springframework.dao.OptimisticLockingFailureException` (楽観排他でエラーが発生した場合に発生する例外)。
- `org.springframework.dao.PessimisticLockingFailureException` (悲観排他でエラーが発生した場合に発生する例外)。
- `org.springframework.dao.DuplicateKeyException` (一意制約違反となった場合に発生する例外)。
- etc …

課題: **JPA(Hibernate)** を使用すると、現状意図しないエラーとなることが発覚している。

- 一意制約違反が発生した場合、DuplicateKeyException ではなく、org.springframework.dao.DataIntegrityViolationException が発生する。

システム例外

システムが、正常稼働している時に、発生してはいけない状態を検知したことを通知する例外 。

本例外は、アプリケーション層、およびドメイン層のロジックで発生させる。

システム運用者による対処が必要となる。

- 事前に存在しているはずのマスタデータ、ディレクトリ、ファイルなどが存在しない場合。
- フレームワーク、ライブラリ内で発生する検査例外のうち、システム異常に分類される例外を捕捉した場合 (ファイル操作時の IOException など)。
- etc …

注釈: 該当する例外クラス

- org.terasoluna.gfw.common.exception.SystemException (共通ライブラリから提供しているクラス)。
- 遷移先のエラー画面や、HTTP レスポンスコードを細かく分ける場合は、SystemException を継承した例外クラスを作成すること。
- 共通ライブラリで用意しているシステム例外クラスだと要件を満たせない場合は、プロジェクト毎にシステム例外クラスを作成すること。

予期しないシステム例外

システムが、正常稼働している時には発生しない非検査例外。

システム運用者による対処、またはシステム開発者による解析が必要となる。

予期しないシステム例外は、アプリケーションコードでハンドリング (try-catch) すべきでない。

- アプリケーション、フレームワーク、ライブラリにバグが潜んでいる場合。
- DB サーバなどがダウンしている場合。
- etc …

注釈: 該当する例外クラスの例

- java.lang.NullPointerException(バグ起因で発生する例外)。
- org.springframework.dao.DataAccessResourceFailureException(DB サーバがダウンしている場合に発生する例外)。

- etc …

致命的なエラー

システム (アプリケーション) 全体に影響を及ぼす、致命的な問題が発生している事を通知するエラー 。

システム運用者、またはシステム開発者による対処・リカバリが必要となる。

致命的なエラー（**java.lang.Error** を継承しているエラーオブジェクト）は、アプリケーションコードでハンドリング **(try-catch)** してはいけない。

- Java 仮想マシンで使用できるメモリが不足している場合。
- etc …

注釈: 該当するエラークラスの例

- `java.lang.OutOfMemoryError`(メモリ不足時に発生するエラー)。
- etc …

リクエスト不正時に発生するフレームワーク例外

フレームワークが、リクエスト内容の不正を検知したことを通知する例外 。

本例外は、フレームワーク (Spring MVC) 内で発生する。

原因は、クライアント側に存在するため、システム運用者による対処は、不要である。

- POST メソッドのみ許容しているリクエストパスに対して、GET メソッドでアクセスした場合に発生する例外。
- `@PathVariable` アノテーションを使って、URI から値を抽出する際に、URI に型変換できない値が、指定された場合に発生する例外。
- etc …

注釈: 該当する例外クラスの例

- `org.springframework.web.HttpRequestMethodNotSupportedException`(サポート外のメソッドでアクセスされた場合に発生する例外)。
- `org.springframework.beans.TypeMismatchException`(URI に型変換できない値が指定された場合に発生する例外)。
- etc …

DefaultHandlerExceptionResolver で設定される *HTTP* レスポンスコードについての中の、HTTP ステータス

コードが「4XX」の例外が該当するクラス。

例外ハンドリングのパターン

例外ハンドリングは、目的に応じて、以下 6 種類のパターンに分類される。

(1)-(2) はユースケース毎、(3)-(6) はシステム (アプリケーション) 全体でハンドリングを行う。

表 4.9: 表-例外ハンドリングのパターン

項番	ハンドリングの目的	ハンドリング対象となり得る例外	ハンドリング方法	ハンドリング単位
(1)	ユースケースの一部やり直し (途中からのやり直し) を促す場合	1. ビジネス例外	アプリケーションコード (try-catch)	リクエスト
(2)	ユースケースのやり直し (先頭からのやり直し) を促す場合	1. ビジネス例外 2. 正常稼働時に発生するライブラリ例外	アプリケーションコード (@ExceptionHandler)	ユースケース
(3)	システム、またはアプリケーションが、正常な状態でない事を通知する場合	1. システム例外 2. 予期しないシステム例外	フレームワーク (ハンドリングルールを、`spring-mvc.xml` に指定する)	サーブレット
(4)	リクエスト内容が、不正であることを通知する場合	1. リクエスト不正時に発生するフレームワーク例外	フレームワーク	サーブレット

Continued on next page

表 4.9 – continued from previous page

項番	ハンドリングの目的	ハンドリング対象となり得る例外	ハンドリング方法	ハンドリング単位
(5)	致命的なエラーが発生したことを検知する場合	1. 致命的なエラー	サーブレットコンテナ (ハンドリングルールを、`web.xml` に指定する)	Web アプリケーション
(6)	プレゼンテーション層 *(JSP など)* で、例外が発生したことを通知する場合	1. プレゼンテーション層で発生する全ての例外及びエラー	サーブレットコンテナ (ハンドリングルールを、`web.xml` に指定する)	Web アプリケーション

ユースケースの一部やり直し (途中からのやり直し) を促す場合

ユースケースの一部やり直し (途中からのやり直し) を促す場合は、Controller クラスのアプリケーションコードで捕捉 (try-catch) し、リクエスト単位で例外処理を行う。

注釈: ユースケースの一部やり直しを促す場合の例

- ショッピングサイトで注文処理を行った際に、在庫不足を通知するビジネス例外が発生する場合。

  このケースの場合、個数を減らせば注文処理が行えるため、個数が変更できる画面に遷移し、個数変更を促すメッセージを表示する。

- etc …

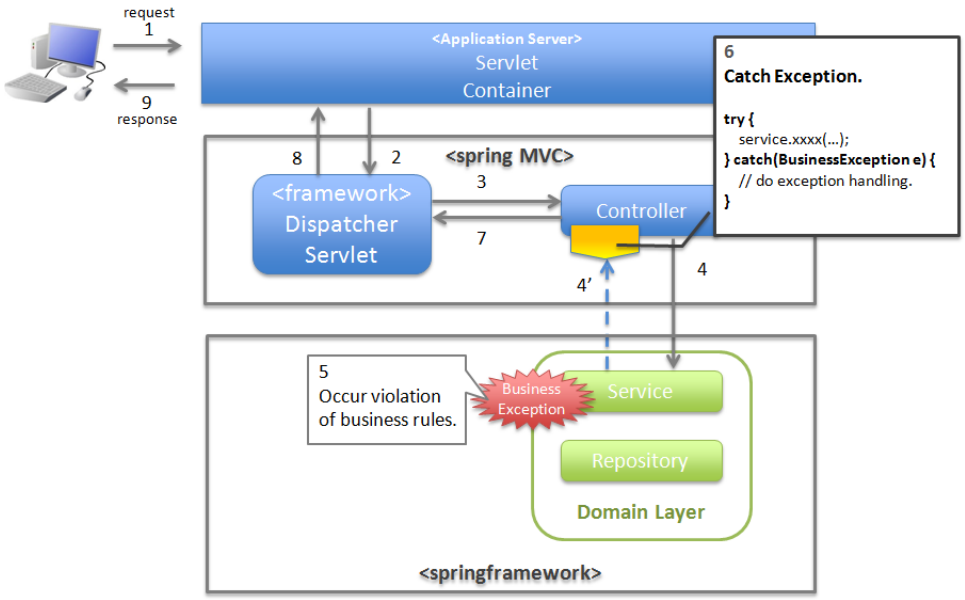


図 4.3　図-ユースケースの一部やり直し (途中からのやり直し) を促す場合のハンドリング方法

ユースケースのやり直し (先頭からのやり直し) を促す場合

ユースケースのやり直し (先頭からのやり直し) を促す場合は、`@ExceptionHandler` を使って捕捉し、ユースケース単位で例外処理を行う。

注釈: ユースケースのやり直し（先頭からのやり直し）を促す場合の例

- ショッピングサイト (管理者向けサイト) で商品マスタの変更を行った際に、変更対象の商品マスタが他のオペレータによって変更されていた場合（楽観排他例外が発生した場合）。

  このケースの場合、他のユーザが行った変更内容を確認してから操作してもらう必要があるため、ユースケースの先頭画面（例えば商品マスタの検索画面）に遷移し、再操作を促すメッセージを表示する。

- etc …

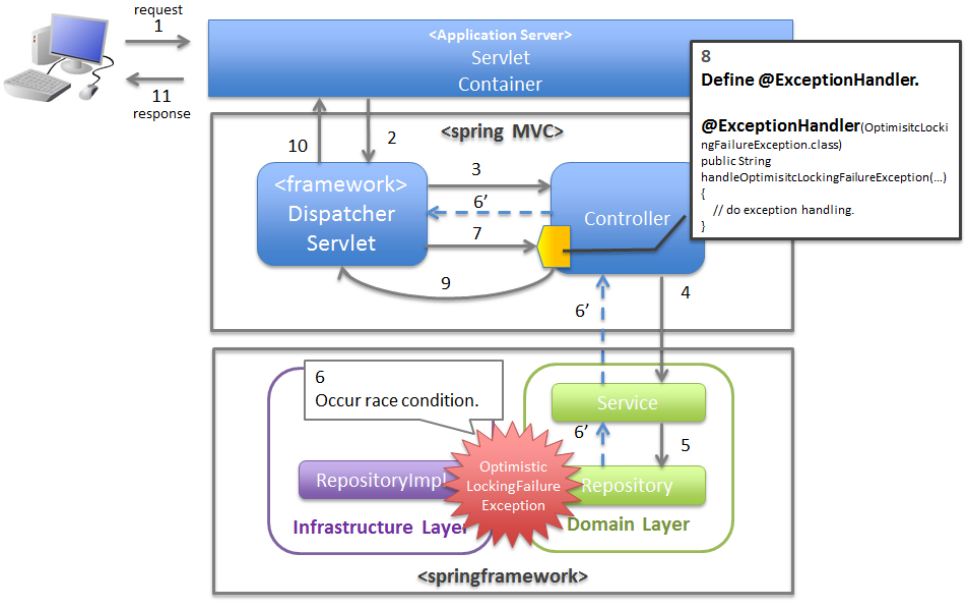


図 4.4　図-ユースケースのやり直し (先頭からのやり直し) を促す場合のハンドリング方法

システム、またはアプリケーションが、正常な状態でない事を通知する場合

システム、またはアプリケーションが、正常な状態でないことを通知する例外を検知する場合は、SystemExceptionResolver で捕捉し、サーブレット単位で例外処理を行う。

注釈: システム、またはアプリケーションが正常な状態でないことを通知する場合の例

- 外部システムとの接続を行うユースケースにて、外部システムが、閉塞中であることを通知する例外が発生した場合。

  このケースの場合、外部システムが開局するまで実行できないため、エラー画面に遷移し、外部システムが開局するまでユースケースが実行できない旨を通知する。

- アプリケーションで指定した値を、条件にマスタ情報の検索を行った際に、該当するマスタ情報が存在しない場合。

  このケースの場合、マスタメンテナンス機能のバグ又はシステム運用者によるデータ投入ミス (リリースミス) の可能性があるため、システムエラー画面に遷移し、システム異常が発生した旨を通知する。

- ファイル操作時に API から IOException が発生した場合。

  このケースの場合、ディスク異常などが考えられるため、システムエラー画面に遷移し、システム異常が発生した旨を通知する。

- etc …

リクエスト内容が、不正であることを通知する場合

フレームワークによって、検知されたリクエスト不正を通知する場合は、DefaultHandlerExceptionResolver で捕捉し、サーブレット単位で例外処理を行う。

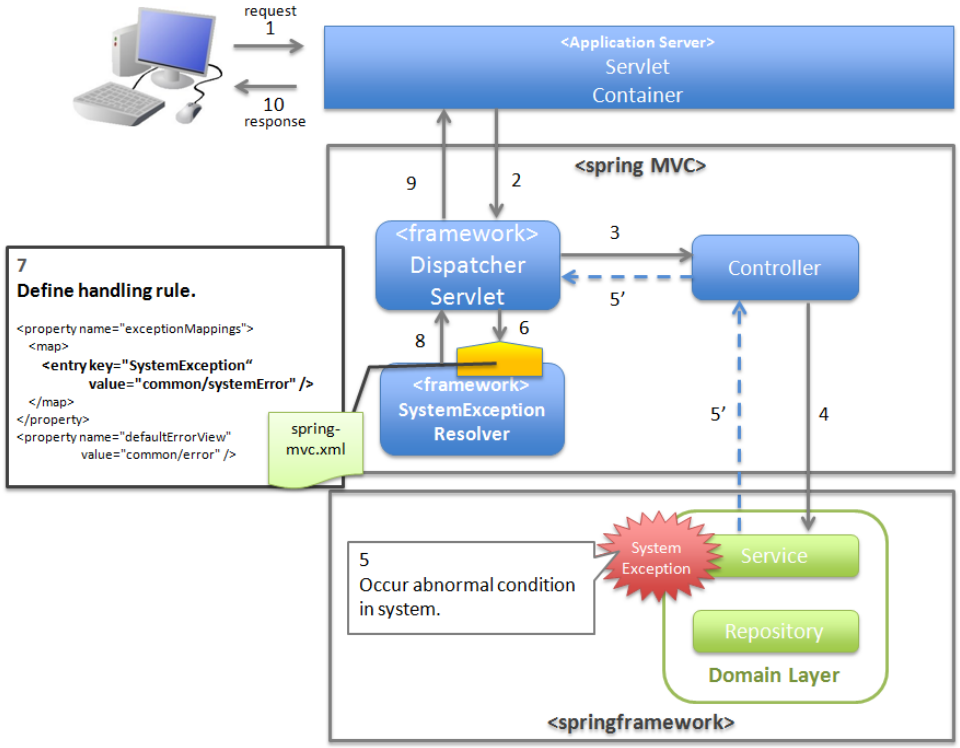


図 4.5　図-システム、またはアプリケーションが、正常な状態でないことを通知する例外を検知する場合のハンドリング方法

注釈: リクエスト内容が不正であることを通知する場合の例

- POST メソッドのみ許可されている URI で、GET メソッドを使ってアクセスした場合。

  このケースの場合、ブラウザのお気に入り機能などを使って直接アクセスしている事が考えられるため、エラー画面に遷移し、リクエスト内容が不正であることを通知する。

- `@PathVariable` アノテーションを使って URI から値を抽出する際に、URI から値を抽出できなかった場合。

  このケースの場合、ブラウザのアドレスバーの値を書き換えて、直接アクセスしている事が考えられるため、エラー画面に遷移し、リクエスト内容が不正であることを通知する。

- etc …

致命的なエラーが発生したことを検知する場合

致命的なエラーが発生したことを検知する場合、サーブレットコンテナで捕捉し、Web アプリケーション単位で例外処理を行う。

> **警告: @ExceptionHandler と SystemExceptionResolver による致命的なエラーのハンドリングついて**
>
> Spring Framework 4.3 より、致命的なエラー (`java.lang.Error` 及びそのサブクラス) や `java.lang.Throwable` がラップされた `org.springframework.web.util.NestedServletException`

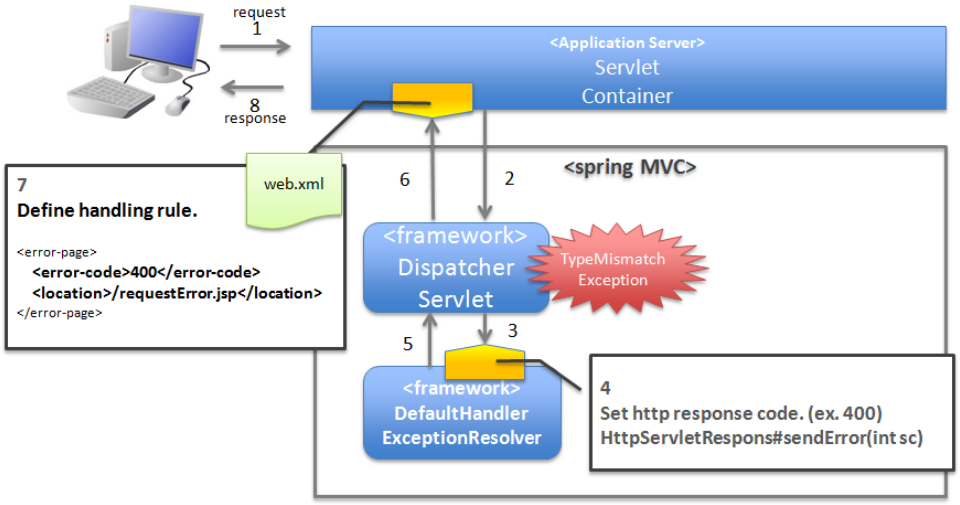


図 4.6　図-リクエスト内容が不正であることを通知する場合のハンドリング方法

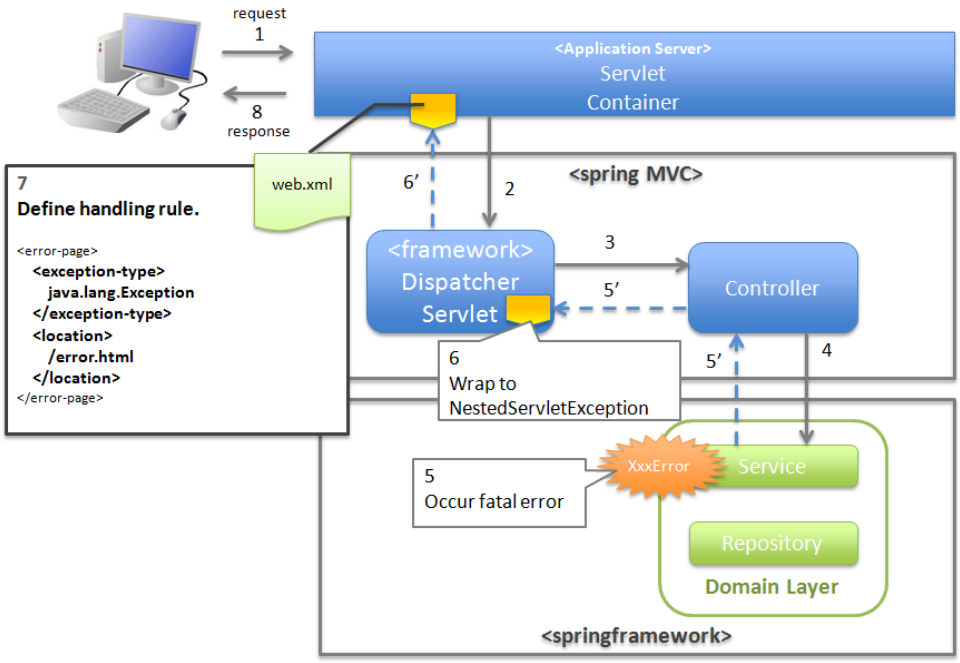


図 4.7　図-致命的なエラーが発生したことを検知する場合のハンドリング方法

を、Spring MVC の例外ハンドラ (`HandlerExceptionResolver`) を使用して捕捉できるようになった。この変更に伴い、致命的なエラーや `Throwable` を意図せず 共通ライブラリが提供する `SystemExceptionResolver`(`HandlerExceptionResolver` を継承) や`@ExceptionHandler` を付与したメソッド (`HandlerExceptionResolver` の仕組み上で動作) によって捕捉してしまう可能性がある。

致命的なエラーをサーブレットコンテナで捕捉するためには、`SystemExceptionResolver` と`@ExceptionHandler` を付与したメソッドで `NestedServletException` をハンドリングせず、サーブレットコンテナに通知する必要がある。`NestedServletException` をハンドリングしない方法については、How to use で解説している。

- `SystemExceptionResolver` については、アプリケーション層の設定を参照されたい。
- `@ExceptionHandler` については、ユースケース単位で例外をハンドリングする方法を参照されたい。

プレゼンテーション層 (JSP など) で、例外が発生したことを通知する場合

プレゼンテーション層 (JSP など) で、例外が発生したことを通知する場合、サーブレットコンテナで捕捉し、Web アプリケーション単位で例外処理を行う。

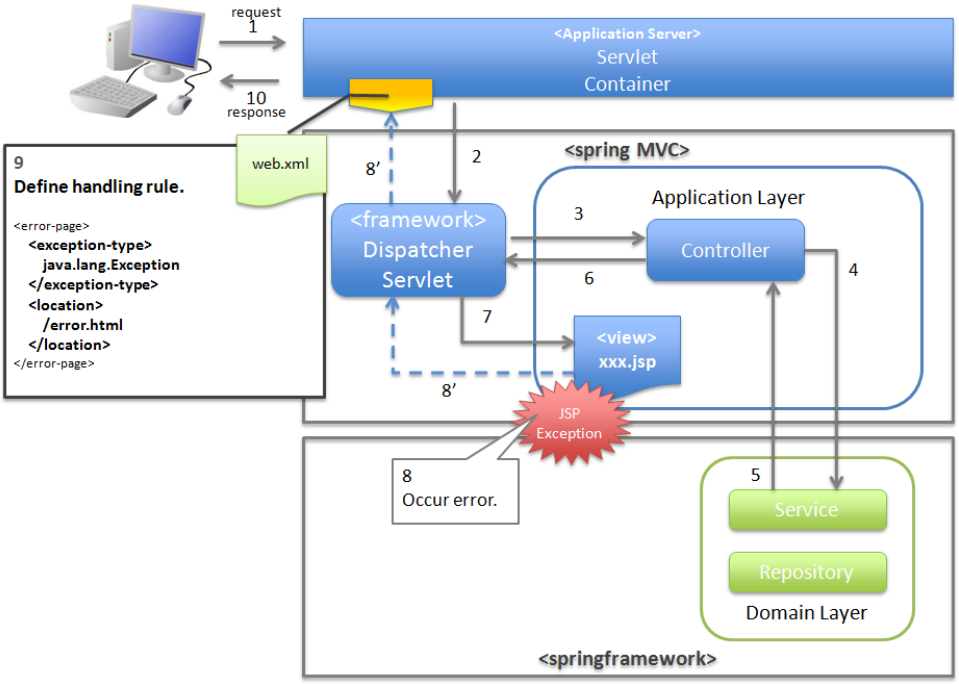


図 4.8　図-プレゼンテーション層 (JSP など) で例外が発生した事を通知する場合のハンドリング方法

例外ハンドリングの基本フロー

例外処理の基本フローを示す。

共通ライブラリから提供しているクラスの概要については、共通ライブラリから提供している例外ハンドリング用のクラスについてを参照されたい。

アプリケーションコードで行う処理 (実装が必要な処理) についての説明は、太字で表現している。

例外メッセージ、およびスタックトレースのログ出力は、共通ライブラリから提供しているクラス（Filter や Interceptor クラス）で行う。

例外メッセージ、およびスタックトレース以外の情報を、ログ出力する必要がある場合は、各ロジックで個別にログを出力すること。

例外ハンドリングのフロー説明であるため、Service クラスを呼び出すまでのフローに関する説明は、省略する。

1. リクエスト単位で *Controller* クラスがハンドリングする場合の基本フロー

2. ユースケース単位で *Controller* クラスがハンドリングする場合の基本フロー
3. サーブレット単位でフレームワークがハンドリングする場合の基本フロー
4. *Web* アプリケーション単位でサーブレットコンテナがハンドリングする場合の基本フロー

リクエスト単位で Controller クラスがハンドリングする場合の基本フロー

例外をリクエスト単位でハンドリングする場合、Controller クラスのアプリケーションコードで捕捉 (try-catch) し、例外処理を行う。

基本フローは、以下の通りである。

下記の図は、 共通ライブラリから提供しているビジネス例外 (`org.terasoluna.gfw.common.exception.BusinessException`) をハンドリングする場合の基本フローである。

ログは、結果メッセージを保持している例外が発生したことを記録するインタセプタ (`org.terasoluna.gfw.common.exception.ResultMessagesLoggingInterceptor`) を使用して、出力する。

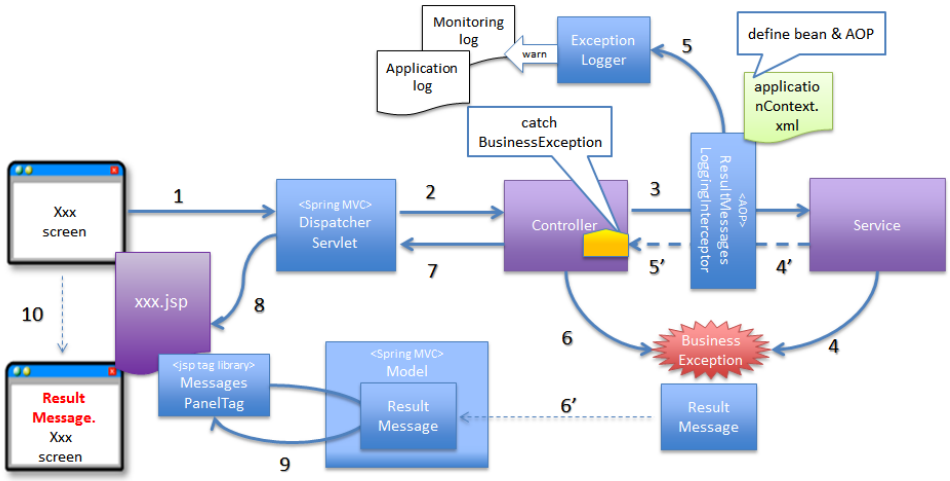


図 4.9　図-リクエスト単位で **Controller** クラスがハンドリングする場合の基本フロー

4. Service クラスにて、 BusinessException を生成し、スローする。
5. ResultMessagesLoggingInterceptor は、 ExceptionLogger を呼び出し、warn レベルのログ (監視ログとアプリケーションログ) を出力する。ResultMessagesLoggingInterceptor は ResultMessagesNotificationException のサブ例外 (BusinessException/ResourceNotFoundException) が発生した場合のみ、ログを出力するクラスである。
6. **Controller** クラスは、 **BusinessException** を捕捉し、 **BusinessException** に設定されているメッセージ情報 **(ResultMessage)** を画面表示用に **Model** に設定する **(6』)**。
7. **Controller** クラスは、遷移先の **View** 名を返却する。
8. DispatcherServlet は、返却された View 名に対応する JSP を呼び出す。

9. **JSP** は、**MessagesPanelTag** を使用して、メッセージ情報 **(ResultMessage)** を取得し、メッセージ表示用の **HTML** コードを生成する。

10. JSP で生成されたレスポンスが表示される。

ユースケース単位で Controller クラスがハンドリングする場合の基本フロー

例外をユースケース単位でハンドリングする場合、Controller クラスの`@ExceptionHandler` を使って捕捉し、例外処理を行う。

基本フローは、以下の通りである。

下記の図は、 任意の例外 (`XxxException`) をハンドリングする場合の、基本フローである。

ログは、HandlerExceptionResolver によって、例外ハンドリングすることを記録するインタセプタ (`org.terasoluna.gfw.web.exception.HandlerExceptionResolverLoggingInterceptor`) を使用して、出力する。

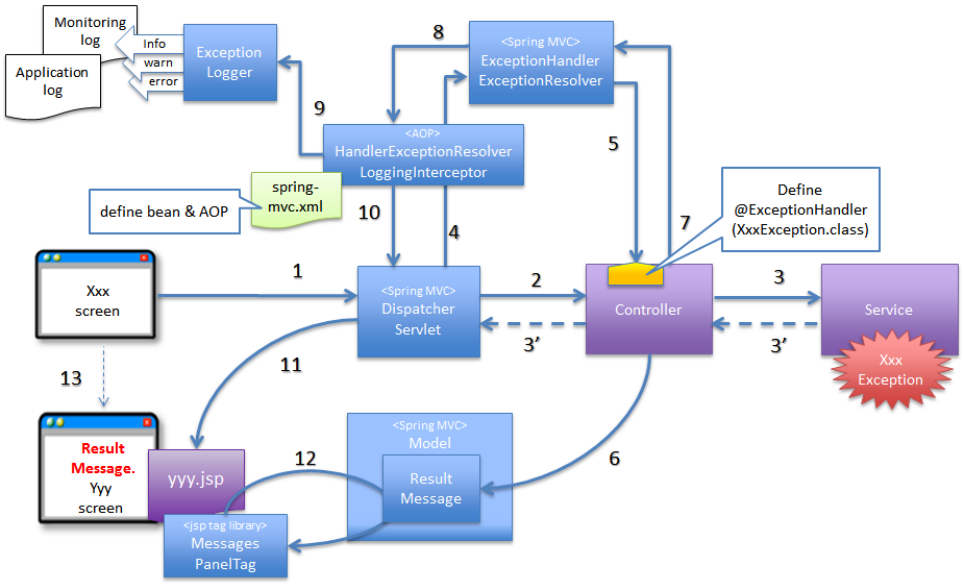


図 4.10　図-ユースケース単位で、**Controller** クラスがハンドリングする場合の基本フロー

3. Controller クラスから呼び出された Service クラスにて、例外 (XxxException) が発生する。

4. DispatcherServlet は、XxxException を捕捉し、ExceptionHandlerExceptionResolver を呼び出す。

5. ExceptionHandlerExceptionResolver は、Controller クラスに用意されている例外ハンドリングメソッドを呼び出す。

6. **Controller** クラスは、メッセージ情報 **(ResultMessage)** を生成し、画面表示用として **Model** に設定する。

7. **Controller** クラスは、遷移先の **View** 名を返却する。

8. ExceptionHandlerExceptionResolver は、Controller より返却された View 名を返却する。

9. HandlerExceptionResolverLoggingInterceptor は、ExceptionLogger を呼び出し、HTTP ステータスコー

ドに対応するレベル (info, warn, error) のログ (監視ログとアプリケーションログ) を出力する。

10. HandlerExceptionResolverLoggingInterceptor は、ExceptionHandlerExceptionResolver より返却された View 名を返却する。
11. DispatcherServlet は、返却された View 名に対応する JSP を呼び出す。
12. **JSP は、MessagesPanelTag を使用して、メッセージ情報 (ResultMessage) を取得し、メッセージ表示用の HTML コードを生成する。**
13. JSP で生成されたレスポンスが表示される。

サーブレット単位でフレームワークがハンドリングする場合の基本フロー

例外をフレームワーク (サーブレット単位) でハンドリングする場合、SystemExceptionResolver で捕捉し例外処理を行う。

基本フローは、以下の通りである。

下記の図は、 共通ライブラリから提供しているシステム例外 (`org.terasoluna.gfw.common.exception.SystemException`) を、 `org.terasoluna.gfw.web.exception.SystemExceptionResolver` を使ってハンドリングする場合の基本フローである。

ログは、例外ハンドリングメソッドの引数に指定された例外を記録するインタセプタ (`org.terasoluna.gfw.web.exception.HandlerExceptionResolverLoggingInterceptor`) を使用して、出力する。

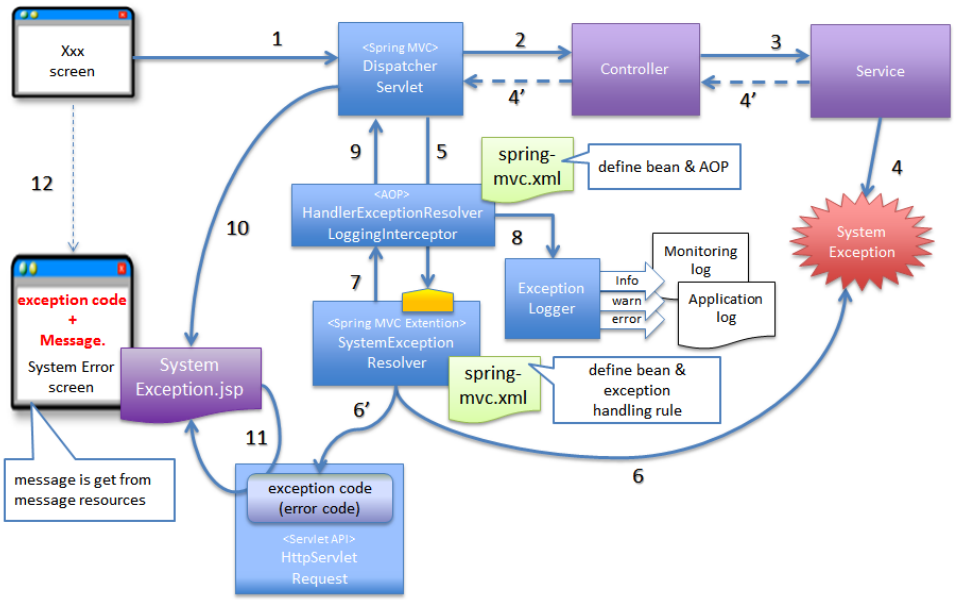


図 4.11　図-サーブレット単位でフレームワークがハンドリングする場合の基本フロー

4. Service クラスにて、システム例外に該当する状態を検知したため、SystemException を発生させる。
5. DispatcherServlet は、SystemException を捕捉し、SystemExceptionResolver を呼び出す。

6. SystemExceptionResolver は、SystemException から例外コードを取得し、画面表示用に HttpServletRequest に設定する (6』)。
7. SystemExceptionResolver は、SystemException 発生時の遷移先の View 名を返却する。
8. HandlerExceptionResolverLoggingInterceptor は、ExceptionLogger を呼び出し、HTTP ステータスコードに対応するレベル (info, warn, error) のログ (監視ログとアプリケーションログ) を出力する。
9. HandlerExceptionResolverLoggingInterceptor は、SystemExceptionResolver より返却された View 名を返却する。
10. DispatcherServlet は、返却された View 名に対応する JSP を呼び出す。
11. **JSP は、HttpServletRequest より例外コードを取得し、メッセージ表示用の HTML コードに埋め込む。**
12. JSP で生成されたレスポンスが表示される。

Web アプリケーション単位でサーブレットコンテナがハンドリングする場合の基本フロー

例外を Web アプリケーション単位でハンドリングする場合、サーブレットコンテナで捕捉し、例外処理を行う。

致命的なエラー、フレームワークでハンドリング対象となっていない例外 (JSP 内で発生した例外など)、Filter で発生した例外をハンドリングする。

基本フローは以下の通りである。

下記フローは、java.lang.Exception を、」error page」でハンドリングする場合のフローである。

ログ出力は、ハンドリングされていない例外が発生したことを記録するサーブレットフィルタ (`org.terasoluna.gfw.web.exception.ExceptionLoggingFilter`) を使用して、出力する。

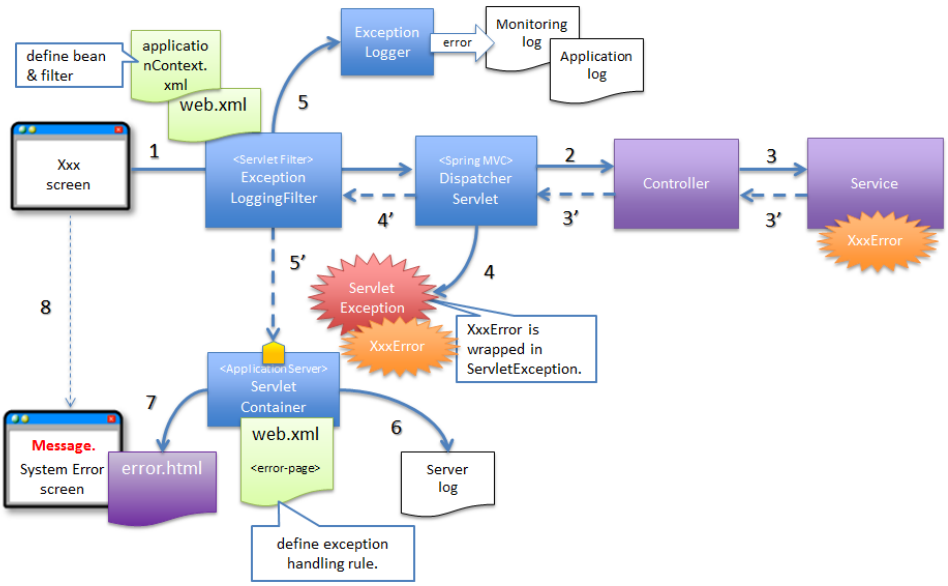


図 4.12　図-Web アプリケーション単位でサーブレットコンテナがハンドリングする場合の基本フロー

4. DispatcherServlet は、XxxError を捕捉し、ServletException にラップしてスローする。

5. ExceptionLoggingFilter は、ServletException を捕捉し、ExceptionLogger を呼び出す。ExceptionLogger は、error レベルのログ (監視ログとアプリケーションログ) を出力する。ExceptionLoggingFilter は、ServletException を再スローする。
6. ServletContainer は、ServletException を捕捉し、サーバログにログを出力する。ログのレベルは、アプリケーションサーバによって異なる。
7. ServletContainer は、`web.xml` に定義されている遷移先 (HTML など) を呼び出す。
8. 呼び出された遷移先で生成されたレスポンスが表示される。

4.2.3 How to use

例外ハンドリング機能の使用方法について説明する。

共通ライブラリから提供している例外ハンドリング用のクラスについては、共通ライブラリから提供している例外ハンドリング用のクラスについてを参照されたい。

1. アプリケーションの設定
2. コーディングポイント（*Service* 編）
3. コーディングポイント（*Controller* 編）
4. コーディングポイント（*JSP* 編）

アプリケーションの設定

例外ハンドリングを使用する際に、必要なアプリケーション設定を、以下に示す。

なお、ブランクプロジェクトは、既に設定済みの状態になっているので、**【プロジェクト毎にカスタマイズする箇所】**の部分を変更すればよい。

1. 共通設定
2. ドメイン層の設定
3. アプリケーション層の設定
4. サーブレットコンテナの設定

共通設定

1． 例外のログ出力を行うロガークラス（`ExceptionLogger`）を、bean 定義に追加する。

- **applicationContext.xml**

```
<!-- Exception Code Resolver. -->
<bean id="exceptionCodeResolver"
    class="org.terasoluna.gfw.common.exception.
↪SimpleMappingExceptionCodeResolver"> <!-- (1) -->
    <!-- Setting and Customization by project. -->
    <property name="exceptionMappings"> <!-- (2) -->
        <map>
            <entry key="ResourceNotFoundException" value="e.xx.fw.5001" />
            <entry key="BusinessException" value="e.xx.fw.8001" />
        </map>
    </property>
    <property name="defaultExceptionCode" value="e.xx.fw.9001" /> <!-- (3) -->
</bean>

<!-- Exception Logger. -->
<bean id="exceptionLogger"
    class="org.terasoluna.gfw.common.exception.ExceptionLogger"> <!-- (4) -->
    <property name="exceptionCodeResolver" ref="exceptionCodeResolver" /> <!--
↪ (5) -->
</bean>
```

項番	説明
(1)	`ExceptionCodeResolver` を、bean 定義に追加する。
(2)	ハンドリング対象とする例外名と、適用する「例外コード (メッセージ ID)」のマッピングを指定する。 上記の設定例では、例外クラス (又は親クラス) のクラス名に、」BusinessException」が含まれている場合は、」e.xx.fw.8001」、「ResourceNotFoundException」が含まれている場合は、」e.xx.fw.5001」が「例外コード (メッセージ ID)」となる。 --- **注釈: 例外コード (メッセージ ID) について** ここでは、」BusinessException」に、メッセージ ID が指定されなかった場合の対応で定義をしているが、後述の」BusinessException」を発生させる実装側で、メッセージ ID を指定することを推奨する。「BusinessException」に対する「例外コード (メッセージ ID)」の指定は、」BusinessException」発生時に指定されなかった場合の救済策である。 --- 【プロジェクト毎にカスタマイズする箇所】

Continued on next page

表 4.10 – continued from previous page

項番	説明
(3)	デフォルトの「例外コード (メッセージ ID)」を指定する。 上記の設定例では、例外クラス (または親クラス) のクラス名に」BusinessException」、または」ResourceNotFoundException」が含まれない場合、」e.xx.fw.9001」が例外コード (メッセージ ID)」となる。 【プロジェクト毎にカスタマイズする箇所】 --- **注釈: 例外コード (メッセージ ID) について** 例外コードは、ExceptionLogger によりログに出力される。(画面での取得も可能である。View(JSP) から例外コードを参照する方法については、システム例外の例外コードを、画面表示する方法を参照されたい。) またコード体系については、プロパティに定義している形式でなくともよい。例えば、MA7001 等 ---
(4)	`ExceptionLogger` を、bean 定義に追加する。
(5)	`ExceptionCodeResolver` を DI する。

2． ログ定義を追加する。

- **logback.xml**

  監視ログ用のログ定義を追加する。

```
<appender name="MONITORING_LOG_FILE" class="ch.qos.logback.core.rolling.
↪RollingFileAppender"> <!-- (1) -->
    <file>${app.log.dir:-log}/projectName-monitoring.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>${app.log.dir:-log}/projectName-monitoring-%d
↪{yyyyMMdd}.log</fileNamePattern>
        <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder>
        <charset>UTF-8</charset>
        <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tX-Track:%X{X-Track}
↪\tlevel:%-5level\tmessage:%msg%n]]></pattern>
    </encoder>
</appender>

<logger name="org.terasoluna.gfw.common.exception.ExceptionLogger.Monitoring"␣
↪additivity="false"> <!-- (2) -->
    <level value="error" /> <!-- (3) -->
    <appender-ref ref="MONITORING_LOG_FILE" /> <!-- (4) -->
</logger>
```

項番	説明
(1)	監視ログを出力するための、appender 定義を指定する。上記の設定例では、ファイルに出力する appender としているが、システム要件に一致する appender を使うこと。 【プロジェクト毎にカスタマイズする箇所】
(2)	監視ログ用の、ロガー定義を指定する。ExceptionLogger を作成する際に、任意のロガー名を指定していない場合は、上記設定のままでよい。 **警告: additivity の設定値について** `false` を指定すること。`true` を指定すると、上位のロガー (例えば、root) によって、同じログが出力されてしまう。
(3)	出力レベルを指定する。ExceptionLogger では info, warn, error の 3 種類のログを出力しているが、システム要件にあったレベルを指定すること。error レベルを推奨する。 【プロジェクト毎にカスタマイズする箇所】
(4)	出力先となる appender を指定する。 【プロジェクト毎にカスタマイズする箇所】

アプリケーションログ用のログ定義を追加する。

```
<appender name="APPLICATION_LOG_FILE" class="ch.qos.logback.core.rolling.
↪RollingFileAppender"> <!-- (1) -->
    <file>${app.log.dir:-log}/projectName-application.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>${app.log.dir:-log}/projectName-application-%d
↪{yyyyMMdd}.log</fileNamePattern>
        <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder>
        <charset>UTF-8</charset>
        <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tthread:%thread\tX-
↪Track:%X{X-Track}\tlevel:%-5level\tlogger:%-48logger{48}\tmessage:%msg%n]]>
↪</pattern>
    </encoder>
</appender>
```

```
<logger name="org.terasoluna.gfw.common.exception.ExceptionLogger"> <!-- (2) -
↪->
    <level value="info" /> <!-- (3) -->
</logger>

<root level="warn">
    <appender-ref ref="STDOUT" />
    <appender-ref ref="APPLICATION_LOG_FILE" /> <!-- (4) -->
</root>
```

項番	説明
(1)	アプリケーションログを出力するための、appender 定義を指定する。上記の設定例では、ファイルに出力する appender としているが、システム要件に一致する appender を使うこと。 【プロジェクト毎にカスタマイズする箇所】
(2)	アプリケーションログ用の、ロガー定義を指定する。ExceptionLogger を作成する際に、任意のロガー名を指定していない場合は、上記設定のままでよい。 **注釈: アプリケーションログ出力用の appender 定義について** アプリケーションログ用の appender は、例外出力用に個別に定義するのではなく、フレームワークや、アプリケーションコードで出力するログ用の appender と、同じものを使うことを推奨する。同じ出力先にすることで、例外が発生するまでの過程が追いやすくなる。
(3)	出力レベルを指定する。ExceptionLogger では、info, warn, error の 3 種類のログを出力しているが、システム要件にあったレベルを指定すること。本ガイドラインでは、info レベルを推奨する。 【プロジェクト毎にカスタマイズする箇所】
(4)	(2) で設定したロガーは、appender を指定していないので、root に流れる。そのため、出力先となる appender を指定する。ここでは、」STDOUT」と」APPLICATION_LOG_FILE」に出力される。 【プロジェクト毎にカスタマイズする箇所】

ドメイン層の設定

ResultMessages を保持する例外 (BisinessException,ResourceNotFoundException) が発生した際に、ログを出力するためのインタセプタクラス（`ResultMessagesLoggingInterceptor`）と、AOP の設定を、bean 定義に追加する。

- **xxx-domain.xml**

```
<!-- interceptor bean. -->
<bean id="resultMessagesLoggingInterceptor"
      class="org.terasoluna.gfw.common.exception.
↪ResultMessagesLoggingInterceptor"> <!-- (1) -->
      <property name="exceptionLogger" ref="exceptionLogger" /> <!-- (2) -->
</bean>

<!-- setting AOP. -->
<aop:config>
    <aop:advisor advice-ref="resultMessagesLoggingInterceptor"
                 pointcut="@within(org.springframework.stereotype.Service)" />
↪ <!-- (3) -->
</aop:config>
```

項番	説明
(1)	`ResultMessagesLoggingInterceptor` を、bean 定義に追加する。
(2)	例外のログ出力を行うロガーオブジェクトを DI する。`applicationContext.xml` に定義している 「exceptionLogger」 を指定する。
(3)	Service クラス (`@Service` アノテーションが付いているクラス) のメソッドに対して、ResultMessagesLoggingInterceptor を適用する。

アプリケーション層の設定

<mvc:annotation-driven> を指定した際に、自動的に登録される HandlerExceptionResolver によって、ハンドリングされない例外をハンドリングするためのクラス（`SystemExceptionResolver`）を、bean 定義に追加する。

- **spring-mvc.xml**

```
<!-- Setting Exception Handling. -->
<!-- Exception Resolver. -->
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver"> <!--␣
↪(1) -->
```

```
    <property name="exceptionCodeResolver" ref="exceptionCodeResolver" /> <!--
↪ (2) -->
    <!-- Setting and Customization by project. -->
    <property name="order" value="3" /> <!-- (3) -->
    <property name="exceptionMappings"> <!-- (4) -->
        <map>
            <entry key="ResourceNotFoundException" value="common/error/
↪resourceNotFoundError" />
            <entry key="BusinessException" value="common/error/businessError"␣
↪/>
            <entry key="InvalidTransactionTokenException" value="common/error/
↪transactionTokenError" />
            <entry key=".DataAccessException" value="common/error/
↪dataAccessError" />
        </map>
    </property>
    <property name="statusCodes"> <!-- (5) -->
        <map>
            <entry key="common/error/resourceNotFoundError" value="404" />
            <entry key="common/error/businessError" value="409" />
            <entry key="common/error/transactionTokenError" value="409" />
            <entry key="common/error/dataAccessError" value="500" />
        </map>
    </property>
    <property name="excludedExceptions"> <!-- (6) -->
        <array>
            <value>org.springframework.web.util.NestedServletException</value>
        </array>
    </property>
    <property name="defaultErrorView" value="common/error/systemError" /> <!--
↪ (7) -->
    <property name="defaultStatusCode" value="500" /> <!-- (8) -->
</bean>

<!-- Settings View Resolver. -->
<mvc:view-resolvers>
    <mvc:jsp prefix="/WEB-INF/views/" /> <!-- (9) -->
</mvc:view-resolvers>
```

項番	説明
(1)	`SystemExceptionResolver` を、bean 定義に追加する。
(2)	例外コード (メッセージ ID) を解決するオブジェクトを DI する。 `applicationContext.xml` に定義している、」exceptionCodeResolver」を指定する。

Continued on next page

表 4.11 – continued from previous page

項番	説明
(3)	ハンドリングの優先順位を指定する。値は、基本的に「3」で良い。 <mvc:annotation-driven>を指定した際に、自動的に、登録されるクラスの方が、優先順位が上となる。 ヒント: **DefaultHandlerExceptionResolver** で行われる例外ハンドリングを無効化する方法 DefaultHandlerExceptionResolver で例外ハンドリングされた場合、HTTP レスポンスコードは設定されるが、View の解決がされないため、View の解決は、web.xml の Error Page で行う必要がある。View の解決を web.xml ではなく、HandlerExceptionResolver で行いたい場合は、SystemExceptionResolver の優先順位を「1」にすると、DefaultHandlerExceptionResolver より前にハンドリング処理を実行することができる。DefaultHandlerExceptionResolver でハンドリングされた場合の、HTTP レスポンスコードのマッピングについては、*DefaultHandlerExceptionResolver で設定される HTTP レスポンスコードについて*を参照されたい。
(4)	ハンドリング対象とする例外名と、遷移先となる View 名のマッピングを指定する。 上記の設定では、例外クラス (または親クラス) のクラス名に」.DataAccessException」が含まれている場合、」common/error/dataAccessError」が、遷移先の View 名となる。 例外クラスが」ResourceNotFoundException」の場合、」common/error/resourceNotFoundError」が、遷移先の View 名となる。 【プロジェクト毎にカスタマイズする箇所】
(5)	遷移先となる View 名と、HTTP ステータスコードのマッピングを指定する。 上記の設定では、View 名が」common/error/resourceNotFoundError」の場合に、」404(Not Found)」が HTTP ステータスコードとなる。 【プロジェクト毎にカスタマイズする箇所】
(6)	ハンドリング対象外とする例外クラスを指定する。 SystemExceptionResolver で致命的なエラーをハンドリングせず、サーブレットコンテナに通知するため、org.springframework.web.util.NestedServletException をハンドリング対象外とする。 ハンドリング対象外にする理由は、「*@ExceptionHandler と SystemExceptionResolver による致命的なエラーのハンドリングついて*」を参照されたい。

Continued on next page

表 4.11 – continued from previous page

項番	説明
(7)	遷移するデフォルトの View 名を、指定する。 上記の設定では、例外クラスに」ResourceNotFoundException」、」BusinessException」、」InvalidTransactionTokenException」や例外クラス (または親クラス) のクラス名に、」.DataAccessException」が含まれない場合、」common/error/systemError」が、遷移先の View 名となる。 【プロジェクト毎にカスタマイズする箇所】
(8)	レスポンスヘッダに設定する HTTP ステータスコードのデフォルト値を指定する。**「500」(Internal Server Error)** を設定することを推奨する。 **警告: 指定を省略した場合の挙動** **「200」(OK)** 扱いになるので、注意すること。
(9)	実際に遷移する `View` は、`ViewResolver` の設定に依存する。 上記の設定では、 • `/WEB-INF/views/common/error/systemError.jsp` • `/WEB-INF/views/common/error/resourceNotFoundError.jsp` • `/WEB-INF/views/common/error/businessError.jsp` • `/WEB-INF/views/common/error/transactionTokenError.jsp` • `/WEB-INF/views/common/error/dataAccessError.jsp` が遷移先となる。

HandlerExceptionResolver でハンドリングされた例外を、ログに出力するためのインタセプタクラス（HandlerExceptionResolverLoggingInterceptor）と、AOP の設定を、bean 定義に追加する。

- **spring-mvc.xml**

```
<!-- Setting AOP. -->
<bean id="handlerExceptionResolverLoggingInterceptor"
    class="org.terasoluna.gfw.web.exception.
↪HandlerExceptionResolverLoggingInterceptor"> <!-- (1) -->
    <property name="exceptionLogger" ref="exceptionLogger" /> <!-- (2) -->
</bean>
<aop:config>
    <aop:advisor advice-ref="handlerExceptionResolverLoggingInterceptor"
        pointcut="execution(* org.springframework.web.servlet.
↪HandlerExceptionResolver.resolveException(..))" /> <!-- (3) -->
</aop:config>
```

項番	説明
(1)	HandlerExceptionResolverLoggingInterceptor を、bean 定義に追加する。
(2)	例外のログ出力を行うロガーオブジェクトを、DI する。applicationContext.xml に定義している」exceptionLogger」を指定する。
(3)	HandlerExceptionResolver インタフェースの resolveException メソッドに対して、HandlerExceptionResolverLoggingInterceptor を適用する。 デフォルトの設定では、共通ライブラリから提供している org.terasoluna.gfw.common.exception.ResultMessagesNotificationException のサブクラスの例外は、このクラスで行われるログ出力の対象外となっている。 ResultMessagesNotificationException のサブクラスの例外をログ出力対象外としている理由は、 org.terasoluna.gfw.common.exception.ResultMessagesLoggingInterceptor によってログ出力されるためである。 デフォルトの設定を変更する必要がある場合は、*HandlerExceptionResolverLoggingInterceptor の設定項目について* を参照されたい。

致命的なエラー、Spring MVC 管理外で発生する例外を、ログに出力するための Filter クラス（ExceptionLoggingFilter）を、bean 定義と web.xml に追加する。

- **applicationContext.xml**

```
<!-- Filter. -->
<bean id="exceptionLoggingFilter"
    class="org.terasoluna.gfw.web.exception.ExceptionLoggingFilter" > <!--
↪(1) -->
    <property name="exceptionLogger" ref="exceptionLogger" /> <!-- (2) -->
</bean>
```

項番	説明
(1)	ExceptionLoggingFilter を、bean 定義に追加する。
(2)	例外のログ出力を行うロガーオブジェクトを、DI する。applicationContext.xml に定義している」exceptionLogger」を指定する。

- **web.xml**

```
<filter>
    <filter-name>exceptionLoggingFilter</filter-name> <!-- (1) -->
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</
↪filter-class> <!-- (2) -->
</filter>
<filter-mapping>
    <filter-name>exceptionLoggingFilter</filter-name> <!-- (3) -->
    <url-pattern>/*</url-pattern> <!-- (4) -->
</filter-mapping>
```

項番	説明
(1)	フィルター名を指定する。applicationContext.xml に定義した ExceptionLoggingFilter の bean 名と、一致させる。
(2)	フィルタークラスを指定する。 org.springframework.web.filter.DelegatingFilterProxy 固定。
(3)	マッピングするフィルターのフィルター名を指定する。(1) で指定した値。
(4)	フィルターを適用する URL パターンを指定する。致命的なエラー、Spring MVC 管理外をログ出力するため、/*を推奨する。

- 出力ログ

```
date:2013-09-25 19:51:52    thread:tomcat-http--3    X-
↪Track:f94de92148f1489b9ceeac3b2f17c969        level:ERROR     logger:o.t.
↪gfw.common.exception.ExceptionLogger         message:[e.xx.fw.9001] An␣
↪exception occurred processing JSP page /WEB-INF/views/exampleJSPException.
↪jsp at line 13
```

サーブレットコンテナの設定

Spring MVC の、デフォルトの例外ハンドリング機能によって行われるエラー応答（HttpServletResponse#sendError）、致命的なエラー、Spring MVC 管理外で発生する例外をハンドリングするために、サーブレットコンテナの Error Page 定義を追加する。

- **web.xml**

  Spring MVC の、デフォルトの例外ハンドリング機能によって行われるエラー応答（HttpServletResponse#sendError）を、ハンドリングするための定義を追加する。

```
<error-page>
    <!-- (1) -->
    <error-code>404</error-code>
    <!-- (2) -->
    <location>/WEB-INF/views/common/error/resourceNotFoundError.jsp</location>
</error-page>
```

項番	説明
(1)	ハンドリング対象とする **HTTP** レスポンスコード を指定する。 【プロジェクト毎にカスタマイズする箇所】 Spring MVC の、デフォルトの例外ハンドリング機能で応答される HTTP レスポンスコードについては、*DefaultHandlerExceptionResolver* で設定される *HTTP* レスポンスコードについてを参照されたい。
(2)	遷移するファイル名を指定する。Web アプリケーションルートからのパスで、指定する。上記の設定では、」${WebAppRoot}/WEB-INF/views/common/error/resourceNotFoundError.jsp」が、遷移先のファイルとなる。 【プロジェクト毎にカスタマイズする箇所】

致命的なエラー、Spring MVC 管理外で発生する例外をハンドリングするための定義を追加する。

```
<error-page>
    <!-- (3) -->
    <location>/WEB-INF/views/common/error/unhandledSystemError.html</location>
</error-page>
```

項番	説明
(3)	遷移するファイル名を指定する。Web アプリケーションルートからのパスで指定する。上記の設定では、」${WebAppRoot}/WEB-INF/views/common/error/unhandledSystemError.html」が、遷移先のファイルとなる。 【プロジェクト毎にカスタマイズする箇所】

注釈: **location** に指定するパスについて

動的コンテンツのパスを指定した場合、致命的なエラーが発生していた場合に、別のエラーが発生する可能性が高くなるため、location には、JSP などの動的コンテンツでなく、 **HTML** などの静的コンテンツへのパスを指定することを推奨する。

注釈: 開発中に原因が特定できないエラーが発生した場合

上記の設定が行われている状態で想定外のエラー応答（HttpServletResponse#sendError）が発生した場合、どのようなエラー応答が発生したのか特定できないケースがある。

location タグに指定したエラー画面が表示されるが、ログなどからエラーの原因を特定できない場合は、上記設定をコメントアウトして動かすことで、発生したエラー応答 (HTTP レスポンスコード) を、画面で確認することできる。

Spring MVC 管理外で発生する例外を、個別にハンドリングする必要がある場合は、例外毎の定義を追加する。

```
<error-page>
    <!-- (4) -->
    <exception-type>java.io.IOException</exception-type>
    <!-- (5) -->
    <location>/WEB-INF/views/common/error/systemError.jsp</location>
</error-page>
```

項番	説明
(4)	ハンドリング対象とする **例外クラス名 (FQCN)** を指定する。
(5)	遷移するファイル名を指定する。Web アプリケーションルートからのパスで指定する。上記の設定では、」${WebAppRoot}/WEB-INF/views/common/error/systemError.jsp」が遷移先のファイルとなる。 **【プロジェクト毎にカスタマイズする箇所】**

コーディングポイント（Service 編）

例外ハンドリングを行う際の、Service でのコーディングポイントを、以下に示す。

1. ビジネス例外を発生させる
2. システム例外を発生させる
3. 例外をキャッチして、処理を継続させる

ビジネス例外を発生させる

ビジネス例外 (BusinessException) の発生方法を、以下に示す。

注釈: ビジネス例外の発生方法に関する注意事項

- 基本的には、ロジックでビジネスルールの違反を検知して、ビジネス例外を発生させる方法を推奨する。
- 既存資材や、基盤機能 (FW や共通機能) の API 仕様として、ビジネスルールの違反が、例外によって

通知される場合のみ、例外を捕捉してビジネス例外を発生させてもよい。

例外を、処理フローを制御するために使用すると、処理全体の見通しが悪くなり、保守性を低下させる可能性がある。

ロジックでビジネスルールの違反を検知して、ビジネス例外を発生させる。

警告:

- デフォルトでは、ビジネス例外は、Service で発生させることを想定している。*AOP の設定*で、`@Service` アノテーションを付与したクラスで発生したビジネス例外のログを出力としている。Controller などでビジネス例外は、ログを出力しない。プロジェクトでの考えがある場合は変更すること。

- xxxService.java

```
...
@Service
public class ExampleExceptionServiceImpl implements ExampleExceptionService {
    @Override
    public String throwBisinessException(String test) {
        ...
        // int stockQuantity = 5;
        // int orderQuantity = 6;

        if (stockQuantity < orderQuantity) {                    // (1)
            ResultMessages messages = ResultMessages.error(); // (2)
            messages.add("e.ad.od.5001", stockQuantity);       // (3)
            throw new BusinessException(messages);             // (4)
        }
        ...
```

項番	説明
(1)	ビジネスルールの違反がないか、チェックを行う。
(2)	違反している場合、ResultMessages を生成する。上記の実装例では、error レベルの ResultMessages を生成している。 ResultMessages の生成方法の詳細については、メッセージ管理を参照されたい。
(3)	ResultMessages に、ResultMessage を追加する。第 1 引数 (必須) にメッセージ ID を、第 2 引数 (任意) にメッセージ埋め込み値を指定する。 メッセージ埋め込み値は、可変長パラメータなので、複数指定することができる。
(4)	ResultMessages を指定して、BusinessException を発生させる。

ちなみに: 上記の `xxxService.java` は説明用に (2)-(4) に分けて処理をしているが、1 ステップで実装することができる。

```
throw new BusinessException(ResultMessages.error().add(
     "e.ad.od.5001", stockQuantity));
```

- xxx.properties

  参考としてプロパティの設定を記述する。

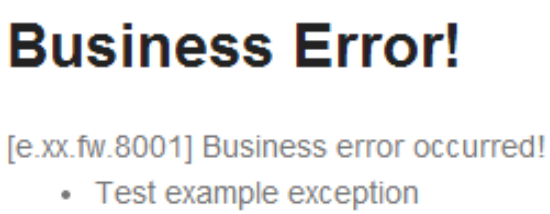
```
e.ad.od.5001 = Order number is higher than the stock quantity={0}. Change the␣
↪order number.
```

下記のようなアプリケーションログが出力される。

```
date:2013-09-17 22:25:55    thread:tomcat-http--8    X-
↪Track:6cfb0b378c124b918e40ac0c32a1fac7         level:WARN       logger:o.t.
↪gfw.common.exception.ExceptionLogger          message:[e.xx.fw.8001]␣
↪ResultMessages [type=error, list=[ResultMessage [code=e.ad.od.5001,␣
↪args=[5], text=null]]]
org.terasoluna.gfw.common.exception.BusinessException: ResultMessages␣
↪[type=error, list=[ResultMessage [code=e.ad.od.5001, args=[5], text=null]]]

// stackTarace omitted
```

```
...

date:2013-09-17 22:25:55    thread:tomcat-http--8   X-
↪Track:6cfb0b378c124b918e40ac0c32a1fac7        level:DEBUG     logger:o.t.
↪gfw.web.exception.SystemExceptionResolver    message:Resolving exception␣
↪from handler [public java.lang.String org.terasoluna.exception.app.example.
↪ExampleExceptionController.home(java.util.Locale,org.springframework.ui.
↪Model)]: org.terasoluna.gfw.common.exception.BusinessException:␣
↪ResultMessages [type=error, list=[ResultMessage [code=e.ad.od.5001,␣
↪args=[5], text=null]]]
date:2013-09-17 22:25:55    thread:tomcat-http--8   X-
↪Track:6cfb0b378c124b918e40ac0c32a1fac7        level:DEBUG     logger:o.t.
↪gfw.web.exception.SystemExceptionResolver    message:Resolving to view
↪'common/error/businessError' for exception of type [org.terasoluna.gfw.
↪common.exception.BusinessException], based on exception mapping␣
↪[BusinessException]
date:2013-09-17 22:25:55    thread:tomcat-http--8   X-
↪Track:6cfb0b378c124b918e40ac0c32a1fac7        level:DEBUG     logger:o.t.
↪gfw.web.exception.SystemExceptionResolver    message:Applying HTTP status␣
↪code 409
date:2013-09-17 22:25:55    thread:tomcat-http--8   X-
↪Track:6cfb0b378c124b918e40ac0c32a1fac7        level:DEBUG     logger:o.t.
↪gfw.web.exception.SystemExceptionResolver    message:Exposing Exception as␣
↪model attribute 'exception'
```

表示される画面



警告: ビジネス例外は、Controller でハンドリングし、各業務画面でメッセージを表示させることを推奨する。上記例は、Controller でハンドリングしなかった場合に、表示される画面となる。

例外を捕捉して、ビジネス例外を発生させる

```
try {
    order(orderQuantity, itemId );
} catch (StockNotEnoughException e) {                    // (1)
    throw new BusinessException(ResultMessages.error().add(
            "e.ad.od.5001", e.getStockQuantity()), e); // (2)
}
```

項番	説明
(1)	ビジネスルールに違反した際に、発生する例外を捕捉する。
(2)	ResultMessages と、原因例外 (e) を指定して、BusinessException を発生させる。

システム例外を発生させる

システム例外 (SystemException) の発生方法を、以下に示す。

ロジックで、システム異常を検知し、システム例外を発生させる。

```
if (itemEntity == null) {                                          // (1)
    throw new SystemException("e.ad.od.9012",
        "not found item entity. item code [" + itemId + "]."); // (2)
}
```

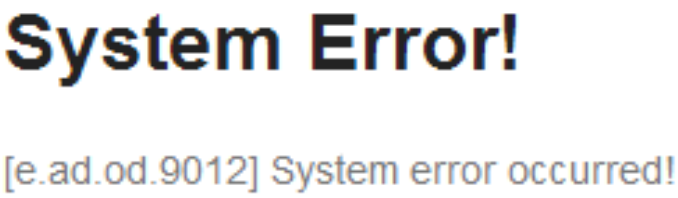
項番	説明
(1)	システムが、正常な状態であることをチェックする。 ここでは、例として、リクエストされた商品コード（itemId）が、商品マスタ（Item Mastar）上に存在するかチェックし、 存在しない場合、システムで用意するべきリソースがないと判断して、システムエラーにしている。
(2)	システムが異常な状態の場合、第 1 引数に例外コード (メッセージ ID) を指定する。第 2 引数に例外メッセージを指定して、SystemException を発生させる。 上記の実装例では、メッセージ本文に、変数」itemId」の値を埋め込んでいる。

下記のような、アプリケーションログが出力される。

```
date:2013-09-19 21:03:06      thread:tomcat-http--3   X-
↪Track:c19eec546b054d54a13658f94292b24f        level:DEBUG     logger:o.t.
↪gfw.web.exception.SystemExceptionResolver    message:Resolving exception␣
↪from handler [public java.lang.String org.terasoluna.exception.app.example.
↪ExampleExceptionController.home(java.util.Locale,org.springframework.ui.
↪Model)]: org.terasoluna.gfw.common.exception.SystemException: not found␣
↪item entity. item code [10-123456].
date:2013-09-19 21:03:06      thread:tomcat-http--3   X-
↪Track:c19eec546b054d54a13658f94292b24f        level:DEBUG     logger:o.t.
↪gfw.web.exception.SystemExceptionResolver    message:Resolving to default␣
↪view 'common/error/systemError' for exception of type [org.terasoluna.gfw.
↪common.exception.SystemException]
```

```
date:2013-09-19 21:03:06      thread:tomcat-http--3   X-
↪Track:c19eec546b054d54a13658f94292b24f        level:DEBUG     logger:o.t.
↪gfw.web.exception.SystemExceptionResolver    message:Applying HTTP status␣
↪code 500
date:2013-09-19 21:03:06      thread:tomcat-http--3   X-
↪Track:c19eec546b054d54a13658f94292b24f        level:DEBUG     logger:o.t.
↪gfw.web.exception.SystemExceptionResolver    message:Exposing Exception as␣
↪model attribute 'exception'
date:2013-09-19 21:03:06      thread:tomcat-http--3   X-
↪Track:c19eec546b054d54a13658f94292b24f        level:ERROR     logger:o.t.
↪gfw.common.exception.ExceptionLogger         message:[e.ad.od.9012] not␣
↪found item entity. item code [10-123456].
org.terasoluna.gfw.common.exception.SystemException: not found item entity.␣
↪item code [10-123456].
      at org.terasoluna.exception.domain.service.ExampleExceptionServiceImpl.
↪throwSystemException(ExampleExceptionServiceImpl.java:14) ~
↪[ExampleExceptionServiceImpl.class:na]
...
// stackTarace omitted
```

表示される画面



注釈: システムエラー画面は、個別に用意せず、共通的に決めることを推奨する。

本ガイドラインの画面では、システムエラーのためのメッセージ ID（業務毎）を表示し、文言は固定にしている。その理由は、オペレータに対して、エラーの細かい内容を知らせる必要がなく、システムに異常があることだけを伝えればよいためである。そこで、開発側では、解析を簡易にするために、キーとなるメッセージ ID を画面に表示して、システム異常の問い合わせに対するレスポンスを向上しようとしている。表示される画面については、各プロジェクトで UI 規約に従い、用意すること。

例外を捕捉して、システム例外を発生させる

```
try {
    return new File(preUploadDir.getFile(), key);
} catch (FileNotFoundException e) { // (1)
    throw new SystemException("e.ad.od.9007",
        "not found upload file. file is [" + preUploadDir.getDescription() +
↪"]."
        e); // (2)
}
```

項番	説明
(1)	システム異常に分類される検査例外を捕捉する。
(2)	例外コード (メッセージ ID)、メッセージ、**原因例外 (e)** を指定して、SystemException を発生させる。

例外をキャッチして、処理を継続させる

例外をキャッチして、処理を継続させる必要がある場合、発生した例外をログに出力してから、処理を継続するようにする。

> **注釈: 監視ログの出力について**
>
> 発生した例外をログに出力する際には、例外の種類に応じて監視ログを出力する事を検討する。共通ライブラリでは、アプリケーションログと監視ログを同時に出力する機能を持つ `org.terasoluna.gfw.common.exception.ExceptionLogger` を提供している。アプリケーションログと合わせて、監視ログの出力も必要となる場合には、`ExceptionLogger` を使用する事を推奨する。

下記は、外部システムから、顧客対応履歴の取得に失敗した場合に、顧客対応履歴以外の情報を取得する処理を、継続する場合の例である。
この例では、顧客対応履歴の情報が取得できなくても、業務は継続できるため、処理を継続している。

```
@Inject
ExceptionLogger exceptionLogger; // (1)

// ...
```

```
InteractionHistory interactionHistory = null;
try {
    interactionHistory = restTemplete.getForObject(uri, InteractionHistory.
↪class, customerId);
} catch (RestClientException e) { // (2)
    exceptionLogger.log(e); // (3)
}

// (4)
Customer customer = customerRepository.findOne(customerId);
```

```
// ...
```

項番	説明
(1)	ログ出力のため、共通ライブラリで提供している `org.terasoluna.gfw.common.exception.ExceptionLogger` を DI する。
(2)	ハンドリング対象の例外をキャッチする。
(3)	`ExceptionLogger` を利用して、キャッチした例外をログに出力する。例では、例外コードに応じた出力レベルでログ出力するため log メソッドを呼び出しているが、出力レベルが決まっており、 後に変更する可能性がない場合は、info、warn、error メソッドを直接呼び出してもよい。
(4)	(3) でログを出力したのみで、処理を継続する。

上記例の場合、以下のような、アプリケーションログ、及び監視ログが出力される。

なお、この挙動は、`ExceptionCodeResolver` の設定がデフォルトの場合を前提としている。

`ExceptionCodeResolver` については、共通ライブラリから提供している例外ハンドリング用のクラスについてを参照されたい。

- アプリケーションログ

```
date:2013-09-19 21:31:47      thread:tomcat-http--3   X-
↪Track:df5271ece2304b12a2c59ff494806397        level:ERROR     logger:o.t.
↪gfw.common.exception.ExceptionLogger         message:[e.xx.fw.9001] Test␣
↪example exception
org.springframework.web.client.RestClientException: Test example exception
...
// stackTarace omitted
```

- 監視ログ

```
date:2013-09-19 21:31:47  X-Track:df5271ece2304b12a2c59ff494806397       ␣
↪level:ERROR     message:[e.xx.fw.9001] Test example exception
```

ExceptionLogger を利用してログ出力する場合、デフォルトの設定では、error レベルのログが監視ログに出力される。

その為、処理を継続させて問題ない場合など、ExceptionLogger を利用してログ出力する際に監視ログへの出力対象外にするには、error 以外のログレベルで出力すれば良い。

これには、以下のいずれかの方法を取れば良い。

- info または warn メソッドでログ出力する。
- ExceptionCodeResolver で該当する例外の例外コードの先頭を e（error）以外に設定し、log メソッドでログ出力する。
- ログ出力する例外が SystemException である場合には、セットする例外コードの先頭を e（error）以外に設定し、log メソッドでログ出力する。

次の例では、info メソッドでログ出力する例を示す。

```
} catch (RestClientException e) {
    exceptionLogger.info(e);
}
```

上記例の場合は、以下のようにアプリケーションログのみが出力される。

```
date:2013-09-19 22:17:53    thread:tomcat-http--3   X-
↪Track:999725b111b4445b8d10b4ea44639c61        level:INFO      logger:o.t.
↪gfw.common.exception.ExceptionLogger         message:[e.xx.fw.9001] Test␣
↪example exception
org.springframework.web.client.RestClientException: Test example exception
```

コーディングポイント（Controller 編）

例外ハンドリングを行う際の、Controller でのコーディングポイントを、以下に示す。

1. リクエスト単位で例外をハンドリングする方法
2. ユースケース単位で例外をハンドリングする方法

リクエスト単位で例外をハンドリングする方法

例外をリクエスト単位でハンドリングし、引き継ぎ情報（メッセージ情報）を、Model に設定する。

その後、遷移する画面を表示するためのメソッドを呼び出すことで、遷移先で必要なモデルを生成し、View 名を決定する。

```
@RequestMapping(value = "change", method = RequestMethod.POST)
public String change(@Validated UserForm userForm,
                     BindingResult result,
                     RedirectAttributes redirectAttributes,
                     Model model) {          // (1)

    // omitted
```

```
    User user = userHelper.convertToUser(userForm);
    try {
        userService.change(user);
    } catch (BusinessException e) {                                    // (2)
        model.addAttribute(e.getResultMessages());                     // (3)
        return viewChangeForm(user.getUserId(), model);                // (4)
    }

    // omitted

}
```

項番	説明
(1)	エラー情報を、View と連携するためのオブジェクトとして、Model を引数に定義する。
(2)	ハンドリング対象となる例外を、アプリケーションコードで捕捉する。
(3)	ResultMessages オブジェクトを、Model に追加する。
(4)	エラー時の遷移先を表示するためのメソッドを呼び出し、View 表示に必要なモデルと、View 名を取得した後に、表示する View 名を返却する。

ユースケース単位で例外をハンドリングする方法

例外を、ユースケース単位でハンドリングし、引き継ぎ情報（メッセージ情報など）が格納された ModelMap（ExtendedModelMap）を生成する。

その後、遷移する画面を表示するためのメソッドを呼び出すことで、遷移先で必要なモデルを生成し、View 名を決定する。

```
@ExceptionHandler(BusinessException.class) // (1)
@ResponseStatus(HttpStatus.CONFLICT) // (2)
public ModelAndView handleBusinessException(BusinessException e) {
    ExtendedModelMap modelMap = new ExtendedModelMap();                // (3)
    modelMap.addAttribute(e.getResultMessages());                      // (4)
    String viewName = top(modelMap);                                   // (5)
    return new ModelAndView(viewName, modelMap);                       // (6)
}
```

項番	説明
(1)	`@ExceptionHandler` アノテーションの value 属性に、ハンドリング対象とする例外クラスを指定する。ハンドリング対象とする例外は、複数指定することもできる。
(2)	`@ResponseStatus` アノテーションの、value 属性に返却する HTTP ステータスコードを指定する。例では、「409:Conflict」を指定している。
(3)	エラー情報と、モデル情報を、View と連携するためのオブジェクトとして、ExtendedModelMap を生成する。
(4)	ResultMessages オブジェクトを、ExtendedModelMap に追加する。
(5)	エラー時の遷移先を表示するためのメソッドを呼び出し、View 表示に必要なモデルと、View 名を取得する。
(6)	(3)-(5) の処理で取得した View 名と、Model が格納されている ModelAndView を生成し、返却する。

警告: `@ExceptionHandler` を付与したメソッドで `java.lang.Exception` や `javax.servlet.ServletException` をハンドリングしている場合は、致命的なエラーをラップしている `NestedServletException` を意図せずハンドリングしてしまうため、サーブレットコンテナに致命的なエラーを通知することができない。詳細は、*「@ExceptionHandler と SystemExceptionResolver による致命的なエラーのハンドリングついて」*を参照されたい。

このようなケースで致命的なエラーをサーブレットコンテナに通知するためには、*SystemExceptionResolver で NestedServletException をハンドリング対象外とする*ことに加えて、`@ExceptionHandler` を付与したメソッドで `NestedServletException` をハンドリングし、再スローするように実装すればよい。以下に実装例を示す。

```
@ExceptionHandler(NestedServletException.class) // (1)
public void handleNestedServletException(NestedServletException e)␣
↪throws NestedServletException {
    throw e; // (2)
}
```

項番	説明
(1)	`@ExceptionHandler` アノテーションを付与し、`NestedServletException.class` を指定する。
(2)	ハンドリングした `NestedServletException` を再スローする。

複数の Controller で Exception や ServletException を捕捉している場合について

複数の Controller で `NestedServletException` を再スローする`@ExceptionHandler` を記述する必要がある場合は、`@ControllerAdvice` の使用を検討した方がよい。`@ControllerAdvice` の詳細は、*@ControllerAdvice の実装*を参照されたい。

コーディングポイント（JSP 編）

例外ハンドリングを行う際の、JSP でのコーディングポイントを、以下に示す。

1. *MessagesPanelTag* を使用して、メッセージを画面表示する方法
2. システム例外の例外コードを、画面表示する方法

ちなみに： アプリケーションで Internet Explorer/Microsoft Edge をサポートする場合、エラー画面の応答として生成される HTML のサイズに注意する必要がある。

Internet Explorer/Microsoft Edge では、応答された HTML のサイズが規定値以下だと、アプリケーションが用意したエラー画面の代わりに、Internet Explorer/Microsoft Edge が用意した簡易メッセージが表示されるためである。

参考までに、Internet Explorer での詳細な条件は、「Friendly HTTP Error Pages」を参照されたい。

MessagesPanelTag を使用して、メッセージを画面表示する方法

任意の場所に、ResultMessages を出力する際の実装例を、以下に示す。

```
<t:messagesPanel /> <!-- (1) -->
```

項番	説明
(1)	メッセージを出力したい場所に、<t:messagesPanel>タグを指定する。 <t:messagesPanel>タグの使用方法の詳細については、メッセージ管理を参照されたい。

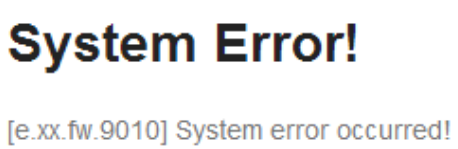
システム例外の例外コードを、画面表示する方法

任意の場所に、例外コード (メッセージ ID) と、固定メッセージを表示する際の実装例を、以下に示す。

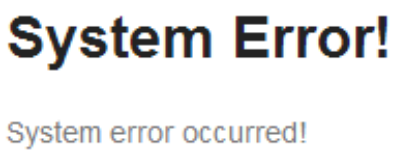
```
<p>
    <c:if test="${!empty exceptionCode}">  <!-- (1) -->
        [${f:h(exceptionCode)}]            <!-- (2) -->
    </c:if>
    <spring:message code="e.cm.fw.9999" /> <!-- (3) -->
</p>
```

項番	説明
(1)	例外コード (メッセージ ID) の存在チェックを行う。上記の実装例のように、記号などで例外コード (メッセージ ID) を囲む場合は、存在チェックを行うこと。
(2)	例外コード (メッセージ ID) を出力する。
(3)	メッセージ定義より取得した、固定メッセージを出力する。

- 出力画面（exceptionCode 有り）



- 出力画面（exceptionCode 無し）



注釈: システム例外時に出力するメッセージについて

- システム例外が発生した場合、エラー原因が特定できる、または推測できる詳細メッセージを出力せず、システム例外が発生したことだけを伝えるメッセージを表示することを推奨する。

- エラー原因が特定できる、または推測できる詳細メッセージを表示した場合、システムの脆弱性を公開してしまう可能性がある。

注釈: 例外コード (メッセージ ID) について

- システム例外が発生した場合、詳細メッセージの代わりに、例外コード (メッセージ ID) を出力することを推奨する。
- 例外コード (メッセージ ID) を出力することで、システム利用者からの問い合わせに、素早く対応することができる。
- 例外コード (メッセージ ID) からエラー原因を特定できるのは、システム管理者だけなので、システムの脆弱性を公開する危険性は少なくなる。

4.2.4 How to use (Ajax)

Ajax の例外ハンドリングについては、*Ajax* を参照されたい。

4.2.5 Appendix

1. 共通ライブラリから提供している例外ハンドリング用のクラスについて
2. *SystemExceptionResolver* の設定項目について
3. *DefaultHandlerExceptionResolver* で設定される *HTTP* レスポンスコードについて

共通ライブラリから提供している例外ハンドリング用のクラスについて

Spring MVC が提供しているクラスとは別に、共通ライブラリより例外ハンドリングを行うためのクラスを提供している。

クラスの役割は、以下の通りである。

表 4.13: 表- org.terasoluna.gfw.common.exception パッケージ配下のクラス

項番	クラス	役割
(1)	ExceptionCode Resolver	例外クラスに対応する例外コード（メッセージ ID）を解決するためのインタフェース。 例外コードとは、どのような例外が発生したのかを識別するためのコードで、システムエラー画面や、ログに出力することを想定している。 `ExceptionLogger`、`SystemExceptionResolver` などから参照される。
(2)	SimpleMapping ExceptionCode Resolver	`ExceptionCodeResolver` の実装クラスで、例外クラスの名前と、例外コードのマッピングを保持することで、例外コードの解決を実現する。 例外クラスの名前は、FQCN ではなく、FQCN の一部や、親クラスの名前でもよい。 **警告:** • FQCN の一部を指定した場合、場合によっては想定していなかったクラスとマッチしてしまうことがあるので注意が必要である。 • 親クラスの名前を指定した場合、全ての子クラスがマッチするので注意が必要である。

Continued on next page

表 4.13 – continued from previous page

項番	クラス	役割
(3)	enums. ExceptionLevel	例外クラスに対応する例外レベルを表現する enum。 INFO, WARN, ERROR が定義されている。
(4)	ExceptionLevel Resolver	例外クラスに対応する例外レベル（ログレベル）を解決するためのインタフェース。 例外レベルとは、どのようなレベルの例外が発生したのかを識別するためのコードで、ログの出力レベルを切り替えるために使われる。 `ExceptionLogger` から参照される。
(5)	DefaultException LevelResolver	`ExceptionLevelResolver` の実装クラスで、例外コードの先頭 1 文字で、例外レベルを解決している。 先頭の 1 文字目（case insensitive）が、 1. 「i」の場合は `ExceptionLevel.INFO` 2. 「w」の場合は `ExceptionLevel.WARN` 3. 「e」の場合は `ExceptionLevel.ERROR` 4. 上記以外の場合は `ExceptionLevel.ERROR` レベルとして扱う。 本クラスは、メッセージのガイドラインに記載されている、メッセージ ID のルールに則った実装となっている。
(6)	ExceptionLogger	例外をログ出力するためのクラス。 監視ログ (メッセージのみ) と、アプリケーションログ (メッセージと、スタックトレースの両方) を出力することができる。 本クラスは、フレームワークより提供している Filter や、Interceptor クラスから使用されている。 アプリケーションコードで、例外をハンドリングして処理を継続する場合、本クラスを用いて、ログを出力すること。

Continued on next page

表 4.13 – continued from previous page

項番	クラス	役割
(7)	ResultMessages LoggingInterceptor	`ResultMessages` を保持している例外 (`ResultMessagesNotificationException` のサブ例外) が発生した事をログに出力するための Interceptor クラス。 ログは全て WARN レベルで出力する。 本 Interceptor は、 `@Service` アノテーションが付与されているクラスのメソッドに対して適用することを想定している。 ログは、`ExceptionLogger` を使用して出力している。
(8)	BusinessException	ビジネスルールの違反を検知したことを通知するための例外クラスで、ドメイン層のロジックで発生させる例外である。 `java.lang.RuntimeException` を継承しているため、デフォルトの動作として、トランザクションは、ロールバックされる。 トランザクションをコミットしたい場合は、`@Transactional` アノテーションの noRollbackFor 、または noRollbackForClassName に、本例外クラスを指定する必要がある。
(9)	Resource NotFoundException	指定されたリソース（データ）が、システム内に存在しないことを通知するための例外クラスで、主に、ドメイン層のロジックで発生させる例外である。 `java.lang.RuntimeException` を継承しているため、デフォルトの動作として、トランザクションは、ロールバックされる。
(10)	ResultMessages Notification Exception	結果メッセージ（`ResultMessages`）を保持している例外であることを通知するための抽象例外クラスで、共通ライブラリでは、`BusinessException` と、`ResourceNotFoundException` が継承している。 `java.lang.RuntimeException` を継承しているため、デフォルトの動作としてトランザクションはロールバックされる。 本例外クラスを継承すると、`ResultMessagesLoggingInterceptor` によって、warn レベルのログが出力される。

Continued on next page

表 4.13 – continued from previous page

項番	クラス	役割
(11)	SystemException	システム又はアプリケーションの異常を検知した事を通知するための例外クラスで、アプリケーション層又はドメイン層のロジックで発生させる例外である。 `java.lang.RuntimeException` を継承しているため、デフォルトの動作として、トランザクションは、ロールバックされる。
(12)	ExceptionCodeProvider	例外コードを保持する役割があることを示すインタフェースで、共通ライブラリでは、`SystemException` が実装している。 本インタフェースを実装した例外クラスを作成すると、共通ライブラリから提供している例外ハンドリング処理にて、例外で保持している例外コードで、そのまま使われる。

表 4.14　**表- org.terasoluna.gfw.web.exception** パッケージ配下のクラス

項番	クラス	役割
(13)	SystemException Resolver	<mvc:annotation-driven>を指定した際に、自動的に登録される HandlerExceptionResolver によって、ハンドリングされない例外をハンドリングするためのクラス。 Spring MVC より提供されている SimpleMappingExceptionResolver を継承し、例外コード及び ResultMessages を、View から参照できるように機能追加を行っている。
(14)	HandlerException ResolverLogging Interceptor	HandlerExceptionResolver でハンドリングされた例外を、ログに出力するための Interceptor クラス。 本 Interceptor クラスでは、HandlerExceptionResolver で解決された HTTP レスポンスコードの分類に応じて、ログの出力レベルを切り替えている。 1. 「100-399」の場合は、 INFO レベルで出力する。 2. 「400-499」の場合は、 WARN レベルで出力する。 3. 「500-「の場合は ERROR レベルで出力する。 4. 「-99」の場合は ログ出力しない。 本 Interceptor を使用することで、Spring MVC 管理下で発生する全ての例外を、ログに出力することができる。 ログは、ExceptionLogger を使用して出力している。 プロジェクトの要件に応じて log メソッドを拡張することで、デフォルトの挙動を変更してログ出力することが可能である。
(15)	ExceptionLogging Filter	致命的なエラー、Spring MVC 管理外で発生する例外を、ログに出力するための Filter クラス。 ログは、すべて ERROR レベルで出力する。 本 Filter を使用した場合、致命的なエラー、および Spring MVC 管理外で発生するすべての例外を、ログに出力することができる。 ログは、ExceptionLogger を使用して出力している。

SystemExceptionResolver の設定項目について

本編で説明していない設定項目について、説明する。要件に応じて、設定を行うこと。

表 4.15: 本編で説明していない設定項目一覧

項番	項目名	プロパティ名	説明	デフォルト値
(1)	結果メッセージの属性名	resultMessagesAttribute	ビジネス例外に設定されているメッセージ情報と、モデルに設定する際の属性名 (String) を指定する。 View(JSP) から結果メッセージにアクセスする際の、属性名となる。	resultMessages
(2)	例外コード (メッセージ ID) の属性名	exceptionCodeAttribute	例外コード (メッセージ ID) として、HttpServletRequest に設定する際の属性名 (String) を指定する。 View(JSP) から例外コード (メッセージ ID) にアクセスする際の属性名となる。	exceptionCode
(3)	例外コード (メッセージ ID) のヘッダ名	exceptionCodeHeader	例外コード (メッセージ ID) として、HttpServletResponse のレスポンスヘッダに設定する際のヘッダ名 (String) を指定する。	X-Exception-Code
(4)	例外オブジェクトの属性名	exceptionAttribute	ハンドリングした例外オブジェクトとして、モデルに設定する際の属性名 (String) を指定する。 View(JSP) から例外オブジェクトにアクセスする際の属性名となる。	exception
(5)	本 ExceptionResolver として、使用するハンドラー (Controller) のオブジェクト一覧	mappedHandlers	本 ExceptionResolver を使用するハンドラーの、オブジェクト一覧 (Set) を指定する。 指定したハンドラーオブジェクトで発生した例外のみ、ハンドリングが行われる。 この設定項目は指定してはいけない。	指定なし 指定した場合の動作は、保証しない。

Continued on next page

表 4.15 – continued from previous page

項番	項目名	プロパティ名	説明	デフォルト値
(6)	本 ExceptionResolver を使用するハンドラー (Controller) のクラス一覧	mappedHandlerClasses	本 ExceptionResolver を使用するハンドラーのクラス一覧 (Class[]) を指定する。 指定したハンドラークラスで発生した例外のみハンドリングが行われる。 この設定項目は指定してはいけない。	指定なし 指定した場合の動作は、保証しない。
(7)	HTTP レスポンスのキャッシュ制御有無	preventResponseCaching	HTTP レスポンス時のキャッシュ制御の有無 (true:有 false:無) を指定する。 true:有を指定すると、キャッシュを無効にするための HTTP レスポンスヘッダが追加される。	false:無

(1)-(3) は、org.terasoluna.gfw.web.exception.SystemExceptionResolver の設定項目。

(4) は、org.springframework.web.servlet.handler.SimpleMappingExceptionResolver の設定項目。

(5)-(7) は、org.springframework.web.servlet.handler.AbstractHandlerExceptionResolver の設定項目。

結果メッセージの属性名

SystemExceptionResolver でハンドリングして設定したメッセージと、アプリケーションコードでハンドリングして設定したメッセージを、View(JSP) で別の messagesPanel として出力したい場合は、SystemExceptionResolver 専用の属性名を指定する。

下記に示す例は、デフォルト値から「resultMessagesForExceptionResolver」に変更する場合の、設定&実装例である。

- **spring-mvc.xml**

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">

    <!-- omitted -->

    <property name="resultMessagesAttribute" value=
↪"resultMessagesForExceptionResolver" /> <!-- (1) -->

    <!-- omitted -->
</bean>
```

- **jsp**

```
<t:messagesPanel messagesAttributeName="resultMessagesForExceptionResolver"/>
↪<!-- (2) -->
```

項番	説明
(1)	結果メッセージの属性名 (resultMessagesAttribute) に、」resultMessagesForExceptionResolver」を指定する。
(2)	メッセージ属性名 (messagesAttributeName) に、SystemExceptionResolver で設定した属性名を指定する。

例外コード (メッセージ ID) の属性名

デフォルトの属性名をアプリケーションコードで使用している場合は、重複を避けるために、別の値を設定すること。重複がない場合は、デフォルト値を変更する必要はない。

下記は、デフォルト値から、「exceptionCodeForExceptionResolver」に変更する場合の、設定&実装例である。

- **spring-mvc.xml**

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">

    <!-- omitted -->

    <property name="exceptionCodeAttribute" value=
↪"exceptionCodeForExceptionResolver" /> <!-- (1) -->

    <!-- omitted -->
</bean>
```

- **jsp**

```
<p>
    <c:if test="${!empty exceptionCodeForExceptionResolver}">  <!-- (2) -->
        [${f:h(exceptionCodeForExceptionResolver)}]            <!-- (3) -->
    </c:if>
    <spring:message code="e.cm.fw.9999" />
</p>
```

項番	説明
(1)	例外コード (メッセージ ID) の属性名 (exceptionCodeAttribute) に、」exceptionCodeForExceptionResolver」を指定する。
(2)	SystemExceptionResolver に設定した値 (exceptionCodeForExceptionResolver) を、テスト対象 (空チェック対応) の変数名として指定する。
(3)	SystemExceptionResolver に設定した値 (exceptionCodeForExceptionResolver) を、出力対象の変数名として指定する。

例外コード (メッセージ ID) のヘッダ名

デフォルトのヘッダ名が使用されている場合、重複を避けるために、別の値を設定すること。重複がない場合は、デフォルト値を変更する必要はない。

下記は、デフォルト値から「X-Exception-Code-ForExceptionResolver」に変更する場合の、設定&実装例である。

- **spring-mvc.xml**

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">

    <!-- omitted -->

    <property name="exceptionCodeHeader" value="X-Exception-Code-
↪ForExceptionResolver" /> <!-- (1) -->

    <!-- omitted -->
</bean>
```

項番	説明
(1)	例外コード (メッセージ ID) のヘッダ名 (exceptionCodeHeader) に、」 X-Exception-Code-ForExceptionResolver」を指定する。

例外オブジェクトの属性名

デフォルトの属性名をアプリケーションコードで使用している場合は、重複を避けるために、別の値を設定すること。重複がない場合は、デフォルト値を変更する必要はない。

下記は、デフォルト値から「exceptionForExceptionResolver」に変更する場合の、設定&実装例である。

- **spring-mvc.xml**

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">

    <!-- omitted -->

    <property name="exceptionAttribute" value="exceptionForExceptionResolver"␣
↪/> <!-- (1) -->

    <!-- omitted -->
</bean>
```

- **jsp**

```
<p>[Exception Message]</p>
<p>${f:h(exceptionForExceptionResolver.message)}</p> <!-- (2) -->
```

項番	説明
(1)	例外オブジェクトの属性名 (exceptionAttribute) に、」exceptionForExceptionResolver」を指定する。
(2)	SystemExceptionResolver に設定した値 (exceptionForExceptionResolver) を、例外オブジェクトからメッセージを取得するための変数名として、指定する。

HTTP レスポンスのキャッシュ制御有無

HTTP レスポンスに、キャッシュ制御用のヘッダを追加したい場合は、true:有を指定する。

- **spring-mvc.xml**

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">

    <!-- omitted -->

    <property name="preventResponseCaching" value="true" /> <!-- (1) -->

    <!-- omitted -->
</bean>
```

項番	説明
(1)	HTTP レスポンスのキャッシュ制御有無 (preventResponseCaching) に、true:有を指定する。

注釈: 有を指定した場合の HTTP レスポンスヘッダ

HTTP レスポンスのキャッシュ制御有無を有にすると、以下の HTTP レスポンスヘッダが出力される。

Cache-Control:no-store

`SystemExceptionResolver` によるキャッシュ制御用のヘッダ追加はブラウザキャッシュによる意図しないエラー画面の表示を抑止するためのオプションであるが、Spring Security の機能を使用してセキュリティの観点からキャッシュ制御用のヘッダを追加することも可能である。Spring Security の機能については、ブラウザのセキュリティ対策機能との連携を参照されたい。

> **警告: SpringSecurity の Cache-Control ヘッダを利用する場合の注意点**
>
> SystemExceptionResolver のキャッシュ制御と Spring Security の Cache-Control ヘッダを有効にした場合、SystemExceptionResolver のキャッシュ制御が優先される。これにより、正常時は Spring Security では no-store 以外も付与されるが、例外時は no-store のみ付与されるため、意図したとおりにキャッシュを制御できない恐れがあることに注意されたい。

HandlerExceptionResolverLoggingInterceptor の設定項目について

本編で説明していない設定項目について、説明する。要件に応じて、設定を行うこと。

表 4.16　本編で説明していない設定項目一覧

項番	項目名	プロパティ名	説明	デフォルト値
(1)	ログ出力対象から除外する例外クラスの一覧	ignoreExceptions	HandlerExceptionResolver によってハンドリングされた例外のうち、ログ出力しない例外クラスをリスト形式で指定する。 指定した例外クラス及びサブクラスの例外が発生した場合、 本クラスでログの出力は行われない。 本項目に指定する例外クラスは、別の場所 (別の仕組み) でログ出力される例外のみ指定すること。	ResultMessagesNoti class ResultMessagesNoti class 及びサブクラスの例外は、 ResultMessagesLogg でログ出力されるため、デフォルト設定として除外している。

ログ出力対象から除外する例外クラスの一覧

プロジェクトで用意した例外クラスをログ出力対象から除外したい場合は、以下のような設定となる。

- **spring-mvc.xml**

```
<bean id="handlerExceptionResolverLoggingInterceptor"
    class="org.terasoluna.gfw.web.exception.
↪HandlerExceptionResolverLoggingInterceptor">
    <property name="exceptionLogger" ref="exceptionLogger" />
    <property name="ignoreExceptions">
        <set>
            <!-- (1) -->
            <value>org.terasoluna.gfw.common.exception.
↪ResultMessagesNotificationException</value>
            <!-- (2) -->
            <value>com.example.common.XxxException</value>
        </set>
```

```
    </property>
</bean>
```

項番	説明
(1)	共通ライブラリのデフォルト設定で指定されている `ResultMessagesNotificationException` を除外対象に指定する。
(2)	プロジェクトで用意した例外クラスを除外対象に指定する。

全ての例外クラスをログ出力対象とする場合は、以下のような設定となる。

- **spring-mvc.xml**

```
<bean id="handlerExceptionResolverLoggingInterceptor"
    class="org.terasoluna.gfw.web.exception.
↪HandlerExceptionResolverLoggingInterceptor">
    <property name="exceptionLogger" ref="exceptionLogger" />
    <!-- (3) -->
    <property name="ignoreExceptions"><null /></property>
</bean>
```

項番	説明
(3)	ignoreExceptions プロパティに `null` を指定する。 `null` を指定すると、全ての例外クラスがログ出力対象となる。

DefaultHandlerExceptionResolver で設定される HTTP レスポンスコードについて

DefaultHandlerExceptionResolver でハンドリングされるフレームワーク例外と、HTTP ステータスコードのマッピングを、以下に記載する。

項番	ハンドリングされるフレームワーク例外	HTTP ステータスコード
(1)	org.springframework.web.servlet.mvc.multiaction.NoSuchRequestHandlingMethodException	404
(2)	org.springframework.web.HttpRequestMethodNotSupportedException	405
(3)	org.springframework.web.HttpMediaTypeNotSupportedException	415
(4)	org.springframework.web.HttpMediaTypeNotAcceptableException	406
(5)	org.springframework.web.bind.MissingPathVariableException	500
(6)	org.springframework.web.bind.MissingServletRequestParameterException	400
(7)	org.springframework.web.bind.ServletRequestBindingException	400
(8)	org.springframework.beans.ConversionNotSupportedException	500
(9)	org.springframework.beans.TypeMismatchException	400
(10)	org.springframework.http.converter.HttpMessageNotReadableException	400
(11)	org.springframework.http.converter.HttpMessageNotWritableException	500

Continued on next page

表 4.17 – continued from previous page

項番	ハンドリングされるフレームワーク例外	HTTP ステータスコード
(12).	org.springframework.web.bind.MethodArgumentNotValidException	400
(13)	org.springframework.web.multipart.support.MissingServletRequestPartException	400
(14)	org.springframework.validation.BindException	400
(15)	org.springframework.web.servlet.NoHandlerFoundException	404
(16)	org.springframework.web.context.request.async.AsyncRequestTimeoutException	503

4.3 セッション管理

4.3.1 Overview

本節では、Web アプリケーションのセッション管理について説明する。

Web アプリケーションは、HTTP を利用して、クライアントとサーバ間でのデータのやり取りを行う。

HTTP 自体には、物理的にセッションを維持する仕組みはないが、セッションを識別するための値 (セッション ID) を、クライアントとサーバとの間で連携することで、論理的にセッションを維持する仕組みが提供されている。

クライアントと、サーバとの間で、セッション ID の連携する方法としては、Cookie、またはリクエストパラメータが使用される。

以下に、論理的なセッションの確立イメージを示す。

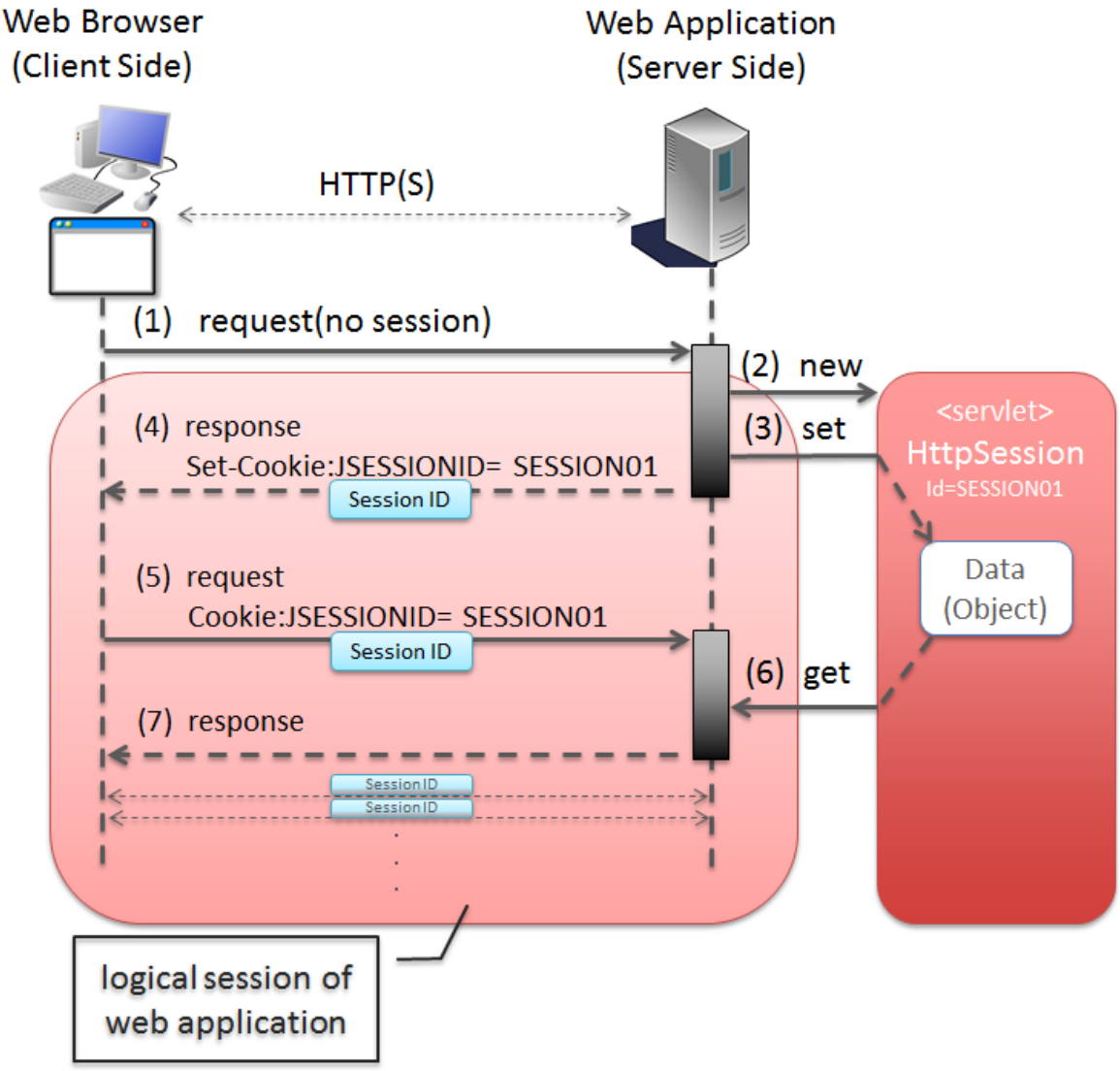


図 4.13 **Picture - Establishment of logical session**

項番	説明
(1)	Web ブラウザ (Client) は、セッションが確立していない状態で、Web アプリケーション (Server) にアクセスする。
(2)	Web アプリケーションは、Web ブラウザとのセッションを管理するために、`HttpSession` オブジェクトを生成する。`HttpSession` オブジェクトを生成したタイミングで、セッション ID が払い出される。
(3)	Web アプリケーションは、Web ブラウザから送信されたデータを、`HttpSession` オブジェクトに格納する。
(4)	Web アプリケーションは、Web ブラウザにレスポンスを返却する。レスポンスの「Set-Cookie」ヘッダに、「JSESSIONID = 払い出されたセッション ID」を設定することで、セッション ID を Web ブラウザに連携する。 連携したセッション ID は Cookie に格納される。
(5)	Web ブラウザは、リクエストの「Cookie」ヘッダに、「JSESSIONID = 払い出されたセッション ID」を設定することで、セッション ID を Web アプリケーションと連携する。 Web アプリケーションがデプロイされているアプリケーションサーバは、Web ブラウザから連携されたセッション ID に対応する `HttpSession` オブジェクトを取得し、リクエストに関連づける。
(6)	Web アプリケーションは、リクエストに関連付けられた `HttpSession` オブジェクトから、(1) のリクエストで格納したデータを取得する。 リクエストをまたいで、同じデータにアクセスすることができる。
(7)	Web アプリケーションは、Web ブラウザにレスポンスを返却する。

注釈: セッション ID を連携するためのパラメータ名について

Java EE の Serlvet の仕様では、セッション ID を連携するためのパラメータ名のデフォルトは、「JSESSIONID」となっている。

セッションのライフサイクル

セッションのライフサイクルの制御 (生成、破棄、タイムアウト検知) は、Controller の処理として実装するのではなく、
フレームワークや共通ライブラリから提供されている処理を使用して行う。

注釈: 以降の説明で登場する **セッション** は、Servlet API より提供されている `javax.servlet.http.HttpSession` オブジェクトの事である。`HttpSession` オブジェクトは、上記で説明した論理的なセッションを表現する Java オブジェクトである。

セッションの生成

本ガイドラインで推奨している方法で Web アプリケーションを作成した場合、以下のいずれかの処理でセッションが生成される。

項番	説明
1.	Spring Security から提供されている認証・認可を行う処理。 Spring Security の設定により、セッションの生成有無や、生成タイミングを指定することができる。 Spring Security で行われるセッション管理についての詳細は、*How to use* を参照されたい。
2.	Spring Security から提供されている CSRF トークンチェックを行う処理。 既にセッションが確立されている場合は、新たなセッションは生成されない。 CSRF トークンチェックの詳細については、*CSRF 対策*を参照されたい。
3.	共通ライブラリから提供されているトランザクショントークンチェックを行う処理。 既にセッションが確立されている場合は、新たなセッションは生成されない。 トランザクショントークンチェックの詳細については、二重送信防止を参照されたい。

Continued on next page

表 4.19 – continued from previous page

項番	説明
4.	`RedirectAttributes` インタフェースの addFlashAttribute メソッドを使用して、リダイレクト先のリクエストにモデル（フォームオブジェクトやドメインオブジェクトなど）を引き渡す処理。 既にセッションが確立されている場合は、新たなセッションは生成されない。 `RedirectAttributes` および Flash scope についての詳細は、リダイレクト先にデータを渡すを参照されたい。
5.	`@SessionAttributes` アノテーションを使用して、モデル（フォームオブジェクトや、ドメインオブジェクトなど）をセッションに格納する処理。 指定したモデル（フォームオブジェクトや、ドメインオブジェクトなど）がセッションに格納される。既にセッションが確立されている場合は、新たなセッションは生成されない。 `@SessionAttributes` アノテーションの使用方法については、*@SessionAttributes アノテーションの使用*を参照されたい。
6.	Spring Framework の、session スコープの Bean を使用する処理。 既にセッションが確立されている場合は、新たなセッションは生成されない。 session スコープの Bean の使用方法については、*Spring Framework の session スコープの Bean の使用*を参照されたい。

注釈: 上記の項番 4, 5, 6 については、セッションの使用有無は Controller の実装によって指定するが、セッションの生成タイミングは、フレームワークによって制御される。つまり、Controller の処理として `HttpSession` の API を直接使用する必要はない。

セッションへの属性格納

本ガイドラインで推奨している方法で Web アプリケーションを作成した場合、以下のいずれかの処理でセッションに属性 (オブジェクト) が格納される。

項番	説明
1.	Spring Security から提供されている認証を行う処理。 認証されたユーザ情報がセッションに格納される。 Spring Security で行われる認証処理の詳細は、認証を参照されたい。
2.	Spring Security から提供されている CSRF トークンチェックを行う処理。 払い出されたトークン値がセッションに格納される。 CSRF トークンチェックの詳細については、*CSRF 対策*を参照されたい。
3.	共通ライブラリから提供されているトランザクショントークンチェックを行う処理。 払い出されたトークン値がセッションに格納される。 トランザクショントークンチェックの詳細については、二重送信防止を参照されたい。
4.	`RedirectAttributes` インタフェースの addFlashAttribute メソッドを使用して、リダイレクト先のリクエストにモデル（フォームオブジェクトやドメインオブジェクトなど）を引き渡す処理。 `RedirectAttributes` インタフェースの addFlashAttribute メソッドの引数に指定したオブジェクトが、セッション上に存在する Flash scope という領域に格納される。 `RedirectAttributes` および Flash scope についての詳細は、リダイレクト先にデータを渡すを参照されたい。

Continued on next page

表 4.20 – continued from previous page

項番	説明
5.	`@SessionAttributes` アノテーションを使用して、モデル（フォームオブジェクトや、ドメインオブジェクトなど）をセッションに格納する処理。 指定したモデル（フォームオブジェクトや、ドメインオブジェクトなど）がセッションに格納される。 `@SessionAttributes` アノテーションの使用方法については、*@SessionAttributes アノテーションの使用*を参照されたい。
6.	Spring Framework の、session スコープの Bean を使用する処理。 session スコープの Bean がセッションに格納される。 session スコープの Bean の使用方法については、*Spring Framework の session スコープの Bean の使用*を参照されたい。

注釈: オブジェクトをセッションに格納するタイミングはフレームワークによって制御されるため、Controller の処理として `HttpSession` オブジェクトの setAttribute メソッドを呼び出すことはない。

セッションからの属性削除

本ガイドラインで推奨している方法で、Web アプリケーションを作成した場合、以下のいずれかの処理でセッションから属性 (オブジェクト) が削除される。

項番	説明
1.	Spring Security から提供されているログアウトを行う処理。 認証されたユーザ情報がセッションから削除される。 Spring Security で行われるログアウト処理についての詳細は、認証を参照されたい。
2.	共通ライブラリから提供されているトランザクショントークンチェックを行う処理。 払い出されたトークン値が、ネームスペースに割り振られている上限値を超えた場合、使用されていないトークン値がセッションから削除される。 トランザクショントークンチェックの詳細については、二重送信防止を参照されたい。
3.	Flash scope にオブジェクトを格納した後のリダイレクト処理。 `RedirectAttributes` インタフェースの addFlashAttribute メソッドの引数に指定したオブジェクトが、セッション上に存在する Flash scope という領域から削除される。
4.	Controller の処理として、 `SessionStatus` オブジェクトの setComplete メソッドを呼び出した後のフレームワークの処理。 `@SessionAttributes` アノテーションで指定したオブジェクトがセッションから削除される。

注釈: セッションからオブジェクトを削除するタイミングはフレームワークによって制御されるため、Controller の処理として `HttpSession` オブジェクトの removeAttribute メソッドを呼び出すことはない。

セッションの破棄

本ガイドラインで推奨している方法で、Web アプリケーションを作成した場合、以下のいずれかの処理でセッションが破棄される。

項番	説明
1.	Spring Security から提供されているログアウト処理。 Spring Security で行われるログアウト処理についての詳細は、認証を参照されたい。
2.	アプリケーションサーバのセッションタイムアウト検知処理。

明示的に破棄する際のイメージを、以下に示す。

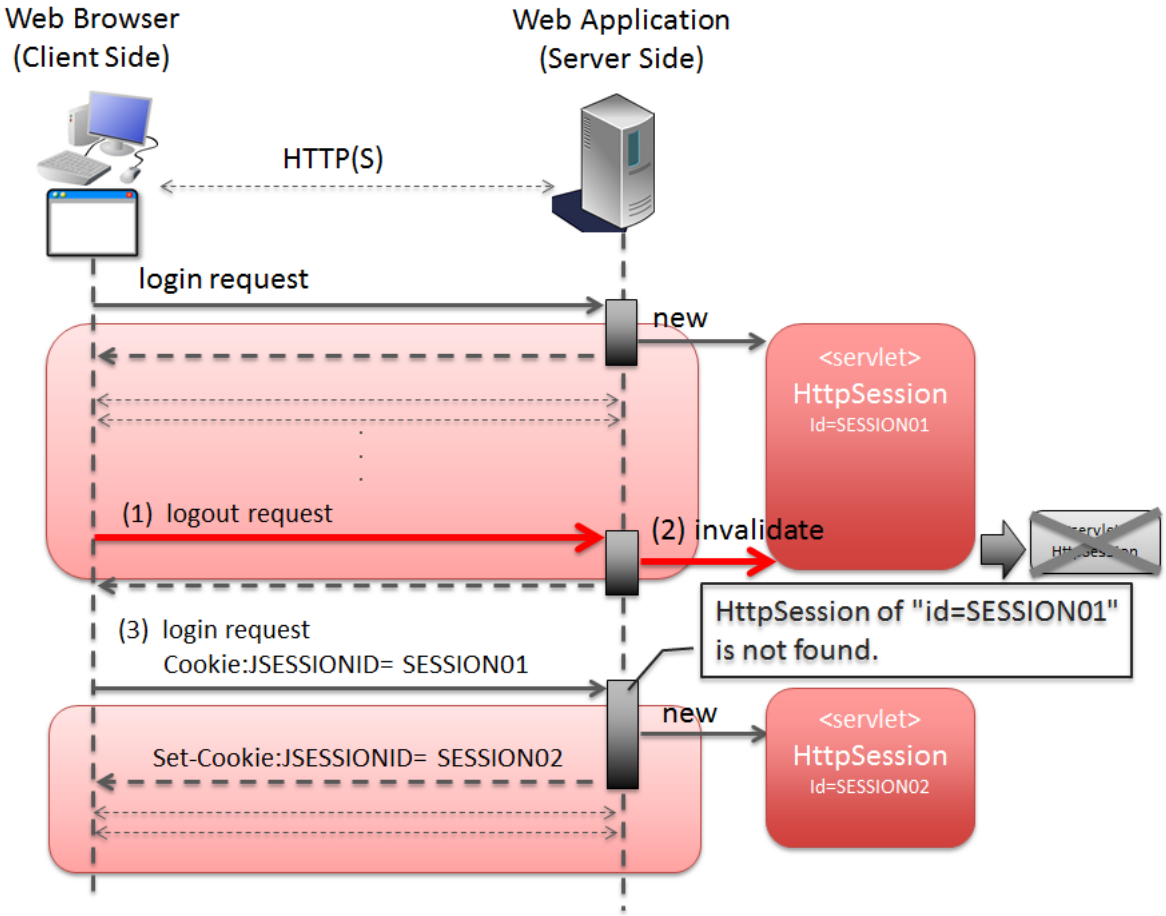


図 4.14 **Picture - Invalidate session by processing of Web Application**

項番	説明
(1)	Web ブラウザからセッションを破棄する処理に、アクセスする。 Spring Security を使用する場合は、Spring Security から提供されているログアウト処理が、セッションを破棄する処理を行っている。 Spring Security で行われるログアウト処理についての詳細は、認証を参照されたい。
(2)	Web アプリケーションは、Web ブラウザから連携されたセッション ID に対応する `HttpSession` オブジェクトを破棄する。 この時点でサーバ側には、 `SESSION01` という ID の `HttpSession` オブジェクトが消滅する。
(3)	Web ブラウザから破棄されたセッションのセッション ID を使ってアクセスされた場合、セッション ID に対応する `HttpSession` オブジェクトが存在しないため、別のセッションを生成する。 上記例では、セッション ID が、 `SESSION02` のセッションを生成している。

タイムアウトによって、自動的に破棄される際のイメージを、以下に示す。

項番	説明
(1)	確立されたセッションに対して一定時間アクセスがない場合、アプリケーションサーバは、セッションタイムアウトを検知する。
(2)	アプリケーションサーバは、セッションタイムアウトが検知されたセッションを破棄する。
(3)	セッションタイムアウト発生後に、Web ブラウザからアクセスされた場合、Web ブラウザから送られてきたセッション ID に対応する `HttpSession` オブジェクトが存在しないため、セッションタイムアウトエラーを Web ブラウザに返却する。

注釈: セッションタイムアウトの設計

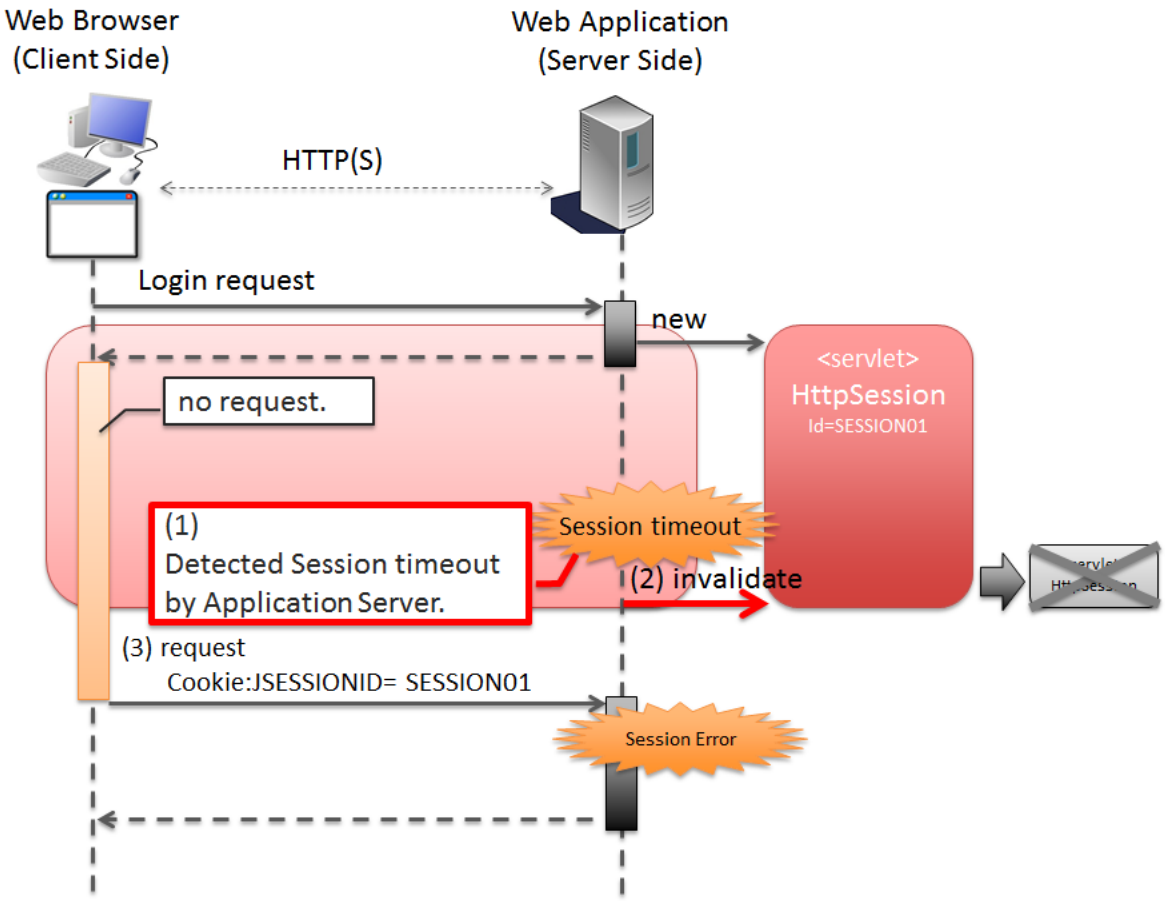


図 4.15 **Picture - Invalidate session by Application Server**

セッションにデータを格納する場合は、必ずセッションタイムアウトの設計を行うこと。特に、格納するデータのサイズが大きくなる場合は、タイムアウトは、可能な限り短く設定することを推奨する。

注釈: デフォルトのセッションタイムアウト時間について

デフォルトのセッションタイムアウト時間は、アプリケーションサーバによって異なる。

- Tomcat : 1800 秒 (30 分)
- WebLogic : 3600 秒 (60 分)
- WebSphere : 1800 秒 (30 分)
- JBoss : 1800 秒 (30 分)

セッションタイムアウト後のリクエスト検知

本ガイドラインで推奨している方法で Web アプリケーションを作成した場合、以下のいずれかの処理で、セッションタイムアウト後のリクエストを検知する。

項番	説明
1.	Spring Security から提供されているセッションのタイムアウトチェック処理。 Spring Security のデフォルトの設定では、セッションのタイムアウトチェックは行われない。 そのため、セッションにデータを格納する場合は、Spring Security のセッションのタイムアウトチェック処理を有効化するための設定が、必要となる。 Spring Security で行われるセッションのタイムアウトチェック処理の詳細は、*How to use* を参照されたい。
2.	Spring Security を使用しない場合は、Servlet Filter、または、Spring MVC の `HandlerInterceptor` にて、セッションのタイムアウトチェックを行う処理を実装する必要がある。

Spring Security から提供されているセッションチェック処理を使用して、セッションタイムアウトを検知する際のイメージについて、以下に示す。

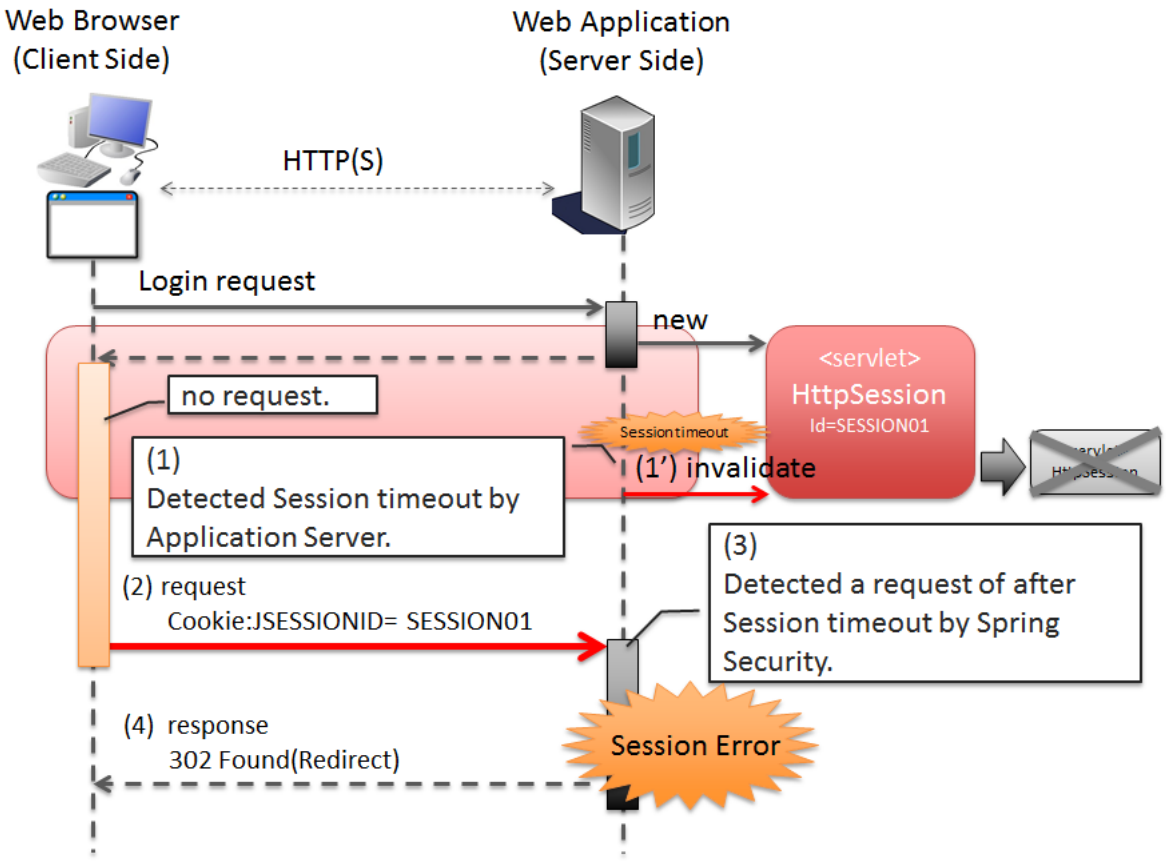


図 4.16 **Picture - Detected a request of after session timeout by Spring Security**

項番	説明
(1)	確立されたセッションに対して、一定時間アクセスがない場合、アプリケーションサーバは、セッションタイムアウトを検知し、セッションを破棄する。
(2)	セッションタイムアウト発生後に、Web ブラウザからアクセスが発生する。
(3)	Spring Security から提供されているセッションの存在チェック処理は、クライアントから連携されたセッション ID に対応する `HttpSession` オブジェクトが存在しないため、セッションタイムアウトエラーとする。 Spring Security のデフォルト実装では、エラー画面を表示するための、URL へのリダイレクト要求が応答される。

注釈: セッションのタイムアウトチェックの必要性

「セッションにデータが格納されていること」が事前条件となる処理については、必ずセッションのタイムアウトチェックを行うこと。セッションのタイムアウトチェックを行わないと、処理で必要なデータが取得できないため、予期しないシステムエラーの発生や、想定外の動作を引き起こす可能性がある。

セッションの利用について

複数の画面 (複数のリクエスト) をまたがって、データの持ち回りが必要になる場合は、持ち回り対象のデータをセッションに格納することで、簡単にデータを持回ることができる。

ただし、セッションにデータを格納すると、データの持ち回りが簡単になるというメリットがある反面、アプリケーション上の制約などが発生するというデメリットもあるため、アプリケーションおよびシステム要件を考慮して、使用有無を決めること。

注釈: 本ガイドラインでは、安易にセッションにデータを格納するのではなく、まずはセッションを使わない方針で検討し、本当に必要なデータのみセッションに格納することを推奨する。

注釈: 以下の条件にあてはまるデータについては、セッションにデータを格納した方がよい場合がある。

- ユースケース間で連携はしないが、別のユースケースに移って戻った際に、状態を保持しておく必要があるデータ。

例えば、一覧画面の検索条件が、このパターンに該当する。

一覧画面の検索条件は、別のユースケース（例えば、「検索したデータを変更する」ユースケース）から戻った際に、別のユースケースに移る前の状態を保持することが機能要件となる事が多い。

検索条件を hidden で持ち回る方法もあるが、ユースケース間に余計な依存関係が生まれ、アプリケーションの実装も複雑になることが予想される。

- ユースケース間で連携が必要なデータ。

  たとえば、ショッピングサイトのカートに格納するデータが、このパターンに該当する。

  ショッピングサイトのカートに格納するデータは、「商品をカートに追加する」ユースケース、「カートを表示する」ユースケース、「カートの状態を変更する」ユースケース、「カートにいれた商品を購入する」ユースケースでデータの連携が必要となるためである。

  ただし、スケラビリティを考慮する必要がある場合は、セッションではなくデータベースに格納した方がよいケースがある。

セッション利用時のメリットとデメリット

セッション利用時のメリットとデメリットは、以下の通りである。

- メリット
  - 複数の画面 (複数のリクエスト) をまたいで、データを持ち回ることができるため、ウィザード画面のような複数の画面で、1 つ処理を構成する場合に、簡単にデータが持ち回れる。
  - 取得したデータをセッションに格納しておくことで、データの取得処理の実行回数を、減らすことができる。
- デメリット
  - 同一処理の画面を、複数のブラウザやタブで立ち上げた場合、互いの操作がセッション上に格納しているデータに干渉しあうため、データの整合性を保つことができなくなる。

    データの整合性を保つためには、同一処理の画面を複数立ち上げることができないように、制御する必要がある。

    データの整合性を保つための制御は、共通ライブラリから提供しているトランザクショントークンチェックを使用することで実現する事ができるが、ユーザビリティの低いアプリケーションとなってしまう。
  - セッションは、通常アプリケーションサーバ上のメモリとして管理されるため、セッションに格納するデータの量に比例して、メモリの使用量も増大する。

    処理で使用されなくなったデータを残したままにすると、ガベージコレクションの対象外となり、メモリ枯渇の原因となるため、不要になった段階でセッションから削除する必要がある。

    セッションから不要となったデータを削除するタイミングについて、別途設計を行う必要がある。
  - 処理で扱うデータをセッションに格納すると、AP サーバのスケーラビリティを低下させる要因となりうる。

注釈: AP サーバをスケールアウトする場合、以下のいずれかの仕組みが必要となる。

1. セッションをレプリケーションし、すべての AP サーバでセッション情報を共有する。

   セッションをレプリケーションする場合は、セッションに格納されるデータの量とレプリケーション対象となる AP サーバの数に比例してレプリケーション処理にかかる負荷が高くなる。

   そのため、スケールアウトすることで、レスポンスタイムなどが劣化する可能性がある点に注意が必要となる。

2. ロードバランサによって、同一セッション内で発生するリクエストを全て同じ AP サーバに振り分ける。

   同じ AP サーバに振り分ける場合は、AP サーバがダウンした場合に別の AP サーバで処理を継続することができない。

   そのため、高い可用性 (サービスレベル) が求められるアプリケーションでは使用できない可能性がある点に注意が必要となる。

それぞれの注意点を考慮した上で、スケールアウトする方法を判断すること。

セッションを利用しない時のメリットとデメリット

セッション使用時のデメリットを回避するためには、サーバの処理で必要となるすべてのデータを、リクエストパラメータとして連携することで、実現することができる。
セッションを利用しない時の、メリットとデメリットは、以下の通りである。

- メリット
  - サーバ側でデータを保持しないため、複数ブラウザや複数タブを使用しても、互いの操作が干渉することはない。そのため、同一処理の画面を複数立ち上げることもできるので、ユーザビリティが損なわれることはない。
  - サーバ側でデータを保持しないため、持続的に使用するメモリの使用量を、抑えることができる。
  - AP サーバのスケーラビリティを低下させる要因が少なくなる。
- デメリット
  - サーバの処理で必要となるデータを、リクエストパラメータとして送信する必要があるため、画面表示に表示していない項目についても、hidden 項目に指定する必要がある。

    そのため、JSP の実装コードが増える。

    これは、JSP タグライブラリを作成することで、最小限に抑えることが可能である。
  - サーバの処理で必要となるデータを、すべてのリクエストで送信する必要があるため、ネットワーク上に流れるデータ量が増える。
  - 画面表示に必要なデータを、その都度取得する必要があるため、データの取得処理の実行回数が増える。

セッションに格納するデータについて

セッションに格納するデータは、以下の点を考慮する必要がある。

- シリアライズすることができるオブジェクト (`java.io.Serializable` を実装しているオブジェクト) であること。
- メモリ枯渇の原因となるような容量の大きいオブジェクトでないこと。

シリアライズ可能なオブジェクト

セッションに格納するデータは、特定の条件下において、ディスク、またはネットワークへの入出力が行われる可能性がある。
そのため、シリアライズすることができるオブジェクトである必要がある。

ディスクへの入出力が発生するケースは、以下の通りである。

- アクティブなセッションが存在する状態で、アプリケーションサーバが停止された場合、セッションおよびセッションに格納されていたデータは、ディスクに退避される。
  退避されたセッション、および格納されていたデータは、アプリケーションサーバ起動時に復元される。
  データの復元に関するこの動作は、アプリケーションサーバによってサポート状況が異なる。
- セッションの格納領域が溢れそうになった場合や、最終アクセスから一定時間アクセスがない場合、セッションのスワップアウトが発生する可能性がある。
  スワップアウトされたセッションは、アクセスが発生した際にスワップインされる。
  スワップアウトの発生条件などは、アプリケーションサーバによって異なる。

ネットワークへの入出力が発生するケースは、以下の通りである。

- セッションを、別のアプリケーションサーバにレプリケーションする場合、セッションに格納したデータが、ネットワークを経由して、別のアプリケーションサーバに送信される。

セッションに格納するデータの容量

セッションに格納するデータは、できる限りコンパクトにすることを推奨する。

セッションに格納されているデータの容量が大きい場合は、致命的なパフォーマンス低下を引き起こす原因となるので、容量の大きいデータは、セッションに格納しないように設計することを推奨する。

パフォーマンス低下を引き起こす主な原因は、以下の通り。

- セッションに容量の大きいデータを格納する場合、メモリ枯渇が発生しないようにするために、アプリケーションサーバの設定をスワップアウトが発生しやすい設定にしておく必要がある。
  スワップアウト処理は、「重い」処理であるため、スワップアウトが頻繁に発生すると、アプリケーション全体のパフォーマンスに影響を与える可能性がある。
  スワップアウトに関する動作や設定方法は、アプリケーションサーバによってサポート状況が異なる。
- セッションをレプリケーションする場合、オブジェクトのシリアライズとデシリアライズが行われる。
  容量の大きいオブジェクトのシリアライズとデシリアライズ処理は、「重い」処理であるため、レスポンスタイムなどのパフォーマンスに影響を与える可能性がある。

セッションに格納するデータをコンパクトにするために、以下の条件にあてはまるデータについては、セッションスコープではなく、リクエストスコープに格納することを検討すること。

- 画面操作で編集することができない読み取り専用のデータ。

  データが必要になったタイミングで最新のデータを取得し、取得したデータをリクエストスコープへ格納した上で View(JSP) へ表示するようにすれば、セッションへ格納する必要はない。

- 画面操作で編集できるが、生存期間がユースケース内の画面操作に閉じているデータ。

  HTML フォームの hidden 項目として、全ての画面遷移でデータを引き回せば、セッションに格納する必要はない。

AP サーバ多重化時の考慮点について

通常のシステム構成では、アプリケーションサーバが 1 台で構成されることはほとんどなく、可用性要件、性能要件などを考慮して、複数台で構成することになる。

そのため、セッションにデータを格納する場合は、システム要件にあわせて以下の何れかの仕組みを適用する必要がある。

1. 高い可用性 (サービスレベル) が求められるシステムの場合は、AP サーバダウン時に別の AP サーバで処理が継続できるようにする必要がある。

   AP サーバダウン時に別の AP サーバで処理を継続するためには、全ての AP サーバでセッション情報を共有しておく必要があるので、アプリケーションサーバをクラスタ構成としてセッションをレプリケーションする必要がある。

   セッション情報を共有する別の方法としては、セッションの保存先を Oracle Coherence のようなキャッシュサーバやデータベースにすることで実現することも可能である。

   AP サーバの台数、セッションに格納されるデータの容量、同時に貼らせるセッション数が大量になる場合は、セッションの保存先を Oracle Coherence のようなキャッシュサーバやデータベースにすることを検討した方がよい。

2. 高い可用性 (サービスレベル) が求められないシステムの場合は、AP サーバダウン時に別の AP サーバで処理を継続できるようにする必要はない。

   そのため、全ての AP サーバでセッション情報を共有する必要はないので、ロードバランサの機能を使って同一セッション内で発生するリクエストを全て同じ AP サーバに振り分けるようにすればよい。

警告: 本ガイドラインで推奨している方法で Web アプリケーションを作成した場合、以下のデータがセッションに格納されるため、何れかの仕組みを適用する必要がある。

- Spring Security の認証処理で認証されたユーザ情報。
- Spring Security の CSRF トークンチェックで払い出されたトークン値。
- 共通ライブラリから提供しているトランザクショントークンチェックで払い出されたトークン値。

セッションの保存先について

セッションの保存先は、AP サーバのメモリだけではなく、Key-Value Store や Oracle Coherence のようなインメモリデータグリッドにすることも可能である。

スケーラビリティが求められる場合は検討の余地がある。

セッションの保存先を変更する際の実装方法については、AP サーバーや保存先によって異なるため、本ガイドラインでは説明は割愛する。

4.3.2 How to use

本ガイドラインでは、セッションにデータを格納する場合は、以下のいずれかの方法を使用して行うことを推奨している。

1. *@SessionAttributes アノテーションの使用*
2. *Spring Framework の session スコープの Bean の使用*

> **警告:** Controller のハンドラメソッドの引数に `HttpSession` オブジェクトを指定することで、`HttpSession` の API を直接呼び出すことができるが、**原則としては HttpSession の API を直接使用しないことを強く推奨する。**
>
> `HttpSession` を直接使わないと実現できない処理については、`HttpSession` の API を直接使用してもよいが、多くの業務処理において、HttpSession の API を直接使用する必要はないため、原則 Controller のハンドラメソッドの引数として、 `HttpSession` オブジェクトを指定しないようにすること。

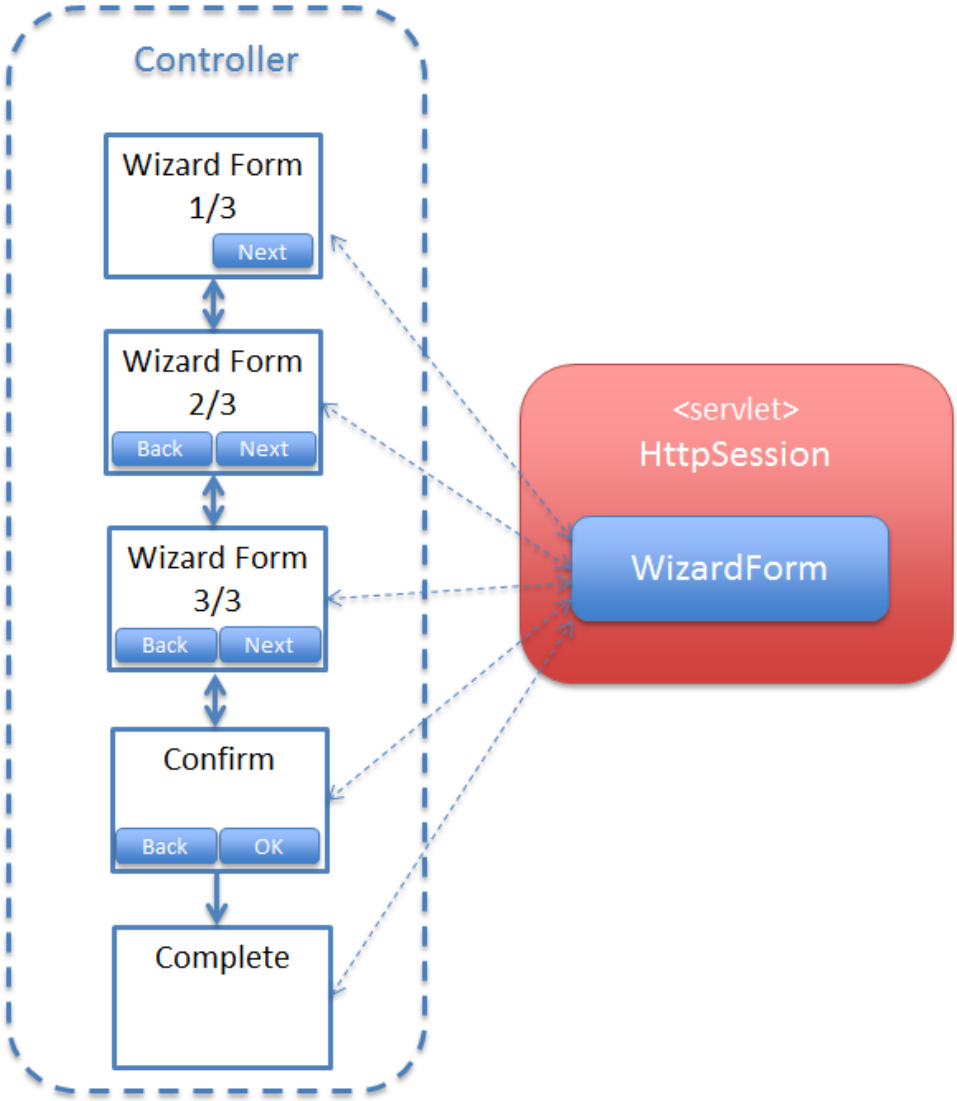
`@SessionAttributes` アノテーションの使用

`@SessionAttributes` アノテーションは、Controller 内で行われる画面遷移において、データを持ち回る場合に使用する。

ただし、入力画面、確認画面、完了画面がそれぞれ 1 ページで構成されるような場合は、セッションを使わずに HTML フォームの hidden を使ってデータを持ち回った方がよい。

入力画面が複数のページで構成されるような場合や、複雑な画面遷移を伴う場合は、
`@SessionAttributes` アノテーションを使用してフォームオブジェクトをセッションに格納する方法を採用すべきか検討すること。

フォームオブジェクトをセッションに格納することで、アプリケーションの設計及び実装がシンプルになる可能性がある。



セッションに格納するオブジェクトの指定

`@SessionAttributes` アノテーションをクラスに指定し、セッションに格納するオブジェクトを指定する。

```
@Controller
@RequestMapping("wizard")
@SessionAttributes(types = { WizardForm.class, Entity.class }) // (1)
public class WizardController {
    // ...
}
```

項番	説明
(1)	`@SessionAttributes` アノテーションの types 属性に、セッションに格納するオブジェクトの型を指定する。 `@ModelAttribute` アノテーション、または `Model` の addAttribute メソッドを使用して、`Model` オブジェクトに追加されたオブジェクトのうち、types 属性で指定した型に一致するオブジェクトが、セッションに格納される。 上記例では、 `WizardForm` クラスと `Entity` クラスのオブジェクトが、セッションに格納される。

注釈: ライフサイクルの管理単位

`@SessionAttributes` アノテーションを使って、セッションに格納したオブジェクトは、Controller 単位で、ライフサイクルが管理される。

`SessionStatus` オブジェクトの setComplete メソッドを呼び出すと、`@SessionAttributes` アノテーションで指定したオブジェクトが、すべてセッションから削除される。そのため、ライフサイクルが異なるオブジェクトを、セッションに格納する場合は、Controller を分割する必要がある。

警告: @SessionAttributes アノテーション使用時の注意点

Controller 単位で、ライフサイクルが管理されると上で説明したが、複数の Controller で同じ属性名のオブジェクトを、`@SessionAttributes` アノテーションを使って、セッションに格納した場合は、Controller をまたいで、ライフサイクルが管理される。

別ウィンドウやタブを開いて、同時に画面操作できる処理の場合は、同じオブジェクトに対してアクセスすることになるため、不具合を引き起こす原因になりうる。そのため、複数の Controller で、同じフォームオブジェクトのクラスを使用する場合は、`@ModelAttribute` アノテーションの value 属性に、それぞれ別の値 (属性名) を指定した上で、 `@SessionAttributes` アノテーションの value 属性に `@ModelAttribute` アノテーションの value 属性に指定した値と同じ値を指定すること。

セッションに格納するオブジェクトの指定は、属性名で指定することも出来る。

以下に、指定方法を示す。

```
@Controller
@RequestMapping("wizard")
@SessionAttributes(value = { "wizardCreateForm" }) // (2)
```

```
public class WizardController {

    // ...

    @ModelAttribute(value = "wizardCreateForm")
    public WizardForm setUpWizardForm() {
        return new WizardForm();
    }

    // ...
}
```

項番	説明
(2)	`@SessionAttributes` アノテーションの value 属性に、セッションに格納するオブジェクトの属性名を指定する。 `@ModelAttribute` アノテーション、または `Model` の addAttribute メソッドを使用して、`Model` オブジェクトに追加されたオブジェクトのうち、value 属性で指定した属性名に一致するオブジェクトが、セッションに格納される。 上記例では、属性名が `wizardCreateForm` のオブジェクトが、セッションに格納される。

セッションにオブジェクトを追加

セッションにオブジェクトを追加する場合、以下 2 つの方法を使用する。

- `@ModelAttribute` アノテーションが付与されたメソッドにて、セッションに追加するオブジェクトを返却する。
- `Model` オブジェクトの addAttribute メソッドを使用して、セッションに格納するオブジェクトを追加する。

`Model` オブジェクトに追加されたオブジェクトは、`@SessionAttributes` アノテーションの types と、value 属性の属性値にしたがって、セッションに格納されるため、Controller のハンドラメソッドで、セッションを意識した実装を行う必要はない。

`@ModelAttribute` アノテーションが付与されたメソッドを使用して、セッションに格納するオブジェクトを返却する方法について、説明する。
フォームオブジェクトをセッションに格納する場合は、この方法を使用して、オブジェクトを生成することを推奨する。

```
@ModelAttribute(value = "wizardForm") // (1)
public WizardForm setUpWizardForm() {
```

```
    return new WizardForm();
}
```

項番	説明
(1)	Model オブジェクトに格納する属性名を、value 属性に指定する。 上記例では、返却したオブジェクトが、wizardForm という属性名でセッションに格納される。 value 属性を指定した場合、セッションにオブジェクトを格納した後のリクエストで、@ModelAttribute アノテーションの付与されたメソッドが呼び出されなくなるため、無駄なオブジェクトの生成が行われないというメリットがある。

警告: @ModelAttribute アノテーションの value 属性を省略した場合の動作について

value 属性を省略した場合、デフォルトの属性名を生成するために、すべてのリクエストで、@ModelAttribute アノテーションの付与されたメソッドが呼ばれる。そのため、無駄なオブジェクトが生成されるというデメリットがあるので、 **セッションに格納する場合は、この方法は原則使用しないこと。**

```
@ModelAttribute // (1)
public WizardForm setUpWizardForm() {
    return new WizardForm();
}
```

項番	説明
(1)	@ModelAttribute アノテーションが付与されたメソッドにて、セッションに追加するオブジェクトを生成し、返却する。 上記例では、wizardForm という属性名で返却したオブジェクトが、セッションに格納される。

Model オブジェクトの addAttribute メソッドを使用し、セッションにオブジェクトを追加する方法について、説明する。

Domain オブジェクトをセッションに格納する場合は、この方法を使用して、オブジェクトを追加することになる。

```
@RequestMapping(value = "update/{id}", params = "form1")
public String updateForm1(@PathVariable("id") Integer id, WizardForm form,
```

```
        Model model) {
    Entity loadedEntity = entityService.getEntity(id);
    model.addAttribute(loadedEntity); // (3)
    beanMapper.map(loadedEntity, form);
    return "wizard/form1";
}
```

項番	説明
(3)	Model オブジェクトの addAttribute メソッドを使用して、セッションに格納するオブジェクトを追加する。 上記例では、entity という属性名で、ドメイン層から取得したオブジェクトを、セッションに格納している。

セッションに格納されているオブジェクトの取得

セッションに格納されているオブジェクトは、Controller のハンドラメソッドの引数として、受け取ることができる。

セッションに格納されているオブジェクトは、@SessionAttributes アノテーションの属性値にしたがって、Model オブジェクトに格納されるため、Controller のハンドラメソッドでは、セッションを意識した実装を行う必要はない。

```
@RequestMapping(value = "save", method = RequestMethod.POST)
public String save(@Validated({ Wizard1.class, Wizard2.class,
        Wizard3.class }) WizardForm form,    // (1)
        BindingResult result,
        Entity entity,                       // (2)
        RedirectAttributes redirectAttributes) {
    // ...
    return "redirect:/wizard/save?complete";
}
```

項番	説明
(1)	Model オブジェクトに格納されているオブジェクトを取得する。 上記例では、wizardForm という属性名でセッションスコープに格納されているオブジェクトが、引数 form に渡される。 @Validated アノテーションで指定している Wizard1.class , Wizard2.class , Wizard3.class については、 Appendix の *@SessionAttributes アノテーションを使ったウィザード形式の画面遷移の実装例* を参照されたい。
(2)	上記例では、entity という属性名でセッションスコープに格納されているオブジェクトが、引数 entity に渡される。

注釈: セッションスコープに格納しているオブジェクトを受け取る際にリクエストパラメータのバインドを防止する方法

セッションスコープに格納しているオブジェクトをハンドラメソッドの引数として受け取る際、その引数にリクエストパラメータがバインドされる可能性がある。

リクエストパラメータがバインドされない様にするためには、セッションスコープに格納しているオブジェクトをハンドラメソッドの引数から受け取らず、ハンドラメソッド内で Model オブジェクトから取得する方法があるが、取得するオブジェクトの属性名を文字列で指定する必要があるためタイプセーフではない。

これに対し、Spring Framework 4.3 では@ModelAttribute アノテーションに binding 属性が追加され、引数にリクエストパラメータをバインドするか否かを指定できる様になった。引数に@ModelAttribute アノテーションを付与し、binding 属性に false を指定することで、リクエストパラメータのバインドを防止しつつ、タイプセーフにセッションスコープに格納しているオブジェクトを取得することができる。

下記の例は、entity という属性名でセッションスコープに格納しているオブジェクトをリクエストパラメータのバインドを防止して取得している。

```
@RequestMapping(value = "save", method = RequestMethod.POST)
public String save(@Validated({ Wizard1.class, Wizard2.class,
        Wizard3.class }) WizardForm form,
        BindingResult result,
        @ModelAttribute(binding = false) Entity entity,
        RedirectAttributes redirectAttributes) {
    // ...
    return "redirect:/wizard/save?complete";
}
```

Controller のハンドラメソッドの引数に渡すオブジェクトが、Model オブジェクトに存在しない場合、@ModelAttribute アノテーションの指定の有無で、動作が変わる。

- @ModelAttribute アノテーションを指定していない場合は、新しいオブジェクトが生成されて引数に渡される。生成されたオブジェクトは Model オブジェクトに格納されるため、セッションにも格納される。

注釈: リダイレクト時の動作について

遷移先をリダイレクトにした場合は、生成されたオブジェクトは、セッションに格納されない。そのため、生成されたオブジェクトを、リダイレクト先の処理で参照したい場合は、RedirectAttributes の addFlashAttribute メソッドを使用して、Flash スコープにオブジェクトを格納する必要がある。

- @ModelAttribute アノテーションを指定している場合は、org.springframework.web.HttpSessionRequiredException が発生する。

```
@RequestMapping(value = "save", method = RequestMethod.POST)
public String save(@Validated({ Wizard1.class, Wizard2.class,
        Wizard3.class }) WizardForm form, // (3)
        BindingResult result,
        @ModelAttribute Entity entity, // (4)
        RedirectAttributes redirectAttributes) {
    // ...
    return "redirect:/wizard/save?complete";
}
```

項番	説明
(3)	@Validated アノテーションで、特定の検証グループ (Wizard1.class, Wizard2.class, Wizard3.class) を設定して入力チェックを行っている。 入力チェックの詳細については、入力チェックを参照されたい。
(4)	引数に、@ModelAttribute アノテーションを指定している場合、セッションに対象のオブジェクトが存在しない時に呼び出されると、HttpSessionRequiredException が発生する。 HttpSessionRequiredException は、ブラウザバックや、URL 直接指定のアクセスなどの、クライアントの操作に起因して発生する例外になるため、クライアントエラーとして、例外ハンドリングを行う必要がある。

HttpSessionRequiredException をクライアントエラーとするための設定は、以下の通りである。

- spring-mvc.xml

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">
    <property name="exceptionCodeResolver" ref="exceptionCodeResolver" />
    <!-- ... -->
    <property name="exceptionMappings">
        <map>
            <!-- ... -->
            <entry key="HttpSessionRequiredException "
                   value="common/error/operationError" /> <!-- (5) -->
        </map>
    </property>
    <property name="statusCodes">
        <map>
            <!-- ... -->
            <entry key="common/error/operationError" value="400" /> <!-- (6) -
↪->
        </map>
    </property>
    <!-- ... -->
</bean>
```

項番	説明
(5)	共通ライブラリから提供している SystemExceptionResolver の exceptionMappings に、HttpSessionRequiredException の例外ハンドリングの定義を追加する。 上記例では、 例外発生時の遷移先の View 名として、common/error/operationError を指定している。
(6)	SystemExceptionResolver の statusCodes に、HttpSessionRequiredException 発生時の、HTTP レスポンスコードを指定する。 上記例では、 例外発生時の HTTP レスポンスコードとして、 Bad Request(400) を指定している。

- applicationContext.xml

```
<bean id="exceptionCodeResolver"
    class="org.terasoluna.gfw.common.exception.
↪SimpleMappingExceptionCodeResolver">
    <!-- Setting and Customization by project. -->
    <property name="exceptionMappings">
        <map>
            <!-- ... -->
            <entry key="HttpSessionRequiredException" value="w.xx.0003" /> <!-
↪- (7) -->
        </map>
```

```
    </property>
    <property name="defaultExceptionCode" value="e.xx.0001" /> <!-- (8) -->
</bean>
```

項番	説明
(7)	共通ライブラリから提供している SimpleMappingExceptionCodeResolver の exceptionMappings に、HttpSessionRequiredException の例外ハンドリングの定義を追加する。 上記例では、 例外発生時の例外コードとして、w.xx.0003 を指定している。 この設定を追加しない場合は、デフォルトの例外コードが、ログに出力される。
(8)	例外発生時のデフォルトの例外コード。

セッションに格納したオブジェクトの削除

@SessionAttributes を用いてセッションに格納したオブジェクトを削除する場合、org.springframework.web.bind.support.SessionStatus の setComplete メソッドを、Controller のハンドラメソッドから呼び出す。

SessionStatus オブジェクトの setComplete メソッドを呼び出すと、 @SessionAttributes アノテーションの属性値に指定されているオブジェクトが、セッションから削除される。

注釈: セッションから削除されるタイミングについて

SessionStatus オブジェクトの setComplete メソッドを呼び出すことで、@SessionAttributes アノテーションの属性値に指定されているオブジェクトが、セッションから削除される。ただし、実際に削除されるタイミングは、setComplete メソッドを呼び出したタイミングではない。

SessionStatus オブジェクトの setComplete メソッド自体は、内部のフラグを変更しているだけなので、実際の削除は、Controller のハンドラメソッドの処理が終了した後に、フレームワークによって行われる。

注釈: View(JSP) からのオブジェクトの参照について

SessionStatus オブジェクトの setComplete メソッドを呼び出すことで、セッションから削除されるが、同じオブジェクトが、Model オブジェクトに残っているため、View(JSP) から参照することができる。

セッションに格納したオブジェクトの削除は、以下 3 カ所で行う必要がある。

- 完了画面を表示するためのリクエスト。**(必須)**

  完了画面を表示した後に、セッションに格納したオブジェクトにアクセスすることはないため、不要になったオブジェクトを削除する。

> **警告: 削除が必要な理由**
>
> セッションに格納されているオブジェクトは、ガベージコレクションの対象とならないため、不要になったオブジェクトを削除しないと、メモリ枯渇の原因になりうる。また、不要なオブジェクトがセッションに格納されていると、セッションのスワップアウトが発生した際の処理が重くなり、アプリケーション全体の性能に影響を与える可能性がある。

- 一連の画面操作を中止するためのリクエスト。**(必須)**

  「メニューへ戻る」や「中止」などの、一連の画面操作を中止するためのイベントについても、セッションに格納したオブジェクトにアクセスすることはないため、不要になったオブジェクトを削除すること。

- 入力画面を初期表示するためのリクエスト。(任意)

> **警告: 削除が必要な理由**
>
> 画面操作の途中でブラウザやタブを閉じた場合、セッションに格納されているフォームオブジェクトに入力途中の情報が残るため、初期表示時に削除しないと、入力途中の情報が画面に表示されてしまう。ただし、入力途中の情報が画面に表示されてもよい場合は、初期表示するためのリクエストで削除は必須ではない。

完了画面を表示するためのリクエストで削除する際の実装例は、以下の通りである。

```
// (1)
@RequestMapping(value = "save", method = RequestMethod.POST)
public String save(@ModelAttribute @Validated({ Wizard1.class,
        Wizard2.class, Wizard3.class }) WizardForm form,
        BindingResult result, Entity entity,
        RedirectAttributes redirectAttributes) {
    // ...
    return "redirect:/wizard/save?complete"; // (2)
}
```

```
// (3)
@RequestMapping(value = "save", params = "complete", method = RequestMethod.
↪GET)
public String saveComplete(SessionStatus sessionStatus) {
    sessionStatus.setComplete(); // (4)
    return "wizard/complete";
}
```

項番	説明
(1)	更新処理を行うためのハンドラメソッド。
(2)	完了画面を表示するためのリクエスト (3) へ、リダイレクトする。
(3)	完了画面を表示するためのハンドラメソッド。
(4)	`SessionStatus` オブジェクトの setComplete メソッドを呼び出し、オブジェクトをセッションから削除する。 `Model` オブジェクトに同じオブジェクトが残っているため、直接、View(JSP) の表示処理に影響は与えない。

一連の画面操作を中止するためのリクエストで削除する際の実装例は、以下の通りである。

```
// (1)
@RequestMapping(value = "save", params = "cancel", method = RequestMethod.
↪POST)
public String saveCancel(SessionStatus sessionStatus) {
    sessionStatus.setComplete(); // (2)
    return "redirect:/wizard/menu"; // (3)
}
```

項番	説明
(1)	一連の画面操作を中止するためのハンドラメソッド。
(2)	SessionStatus オブジェクトの setComplete メソッドを呼び出し、オブジェクトをセッションから削除する。
(3)	上記例では、メニュー画面へ、リダイレクトしている。

入力画面を、初期表示するためのリクエストで削除する際の実装例は、以下の通りである。

```
// (1)
@RequestMapping(value = "create", method = RequestMethod.GET)
public String initializeCreateWizardForm(SessionStatus sessionStatus) {
    sessionStatus.setComplete();              // (2)
    return "redirect:/wizard/create?form1";   // (3)
}

// (4)
@RequestMapping(value = "create", params = "form1")
public String createForm1() {
    return "wizard/form1";
}
```

項番	説明
(1)	入力画面を初期表示するためのハンドラメソッド。
(2)	`SessionStatus` オブジェクトの setComplete メソッドを呼び出す。
(3)	入力画面を表示するためのリクエスト (4) へ、リダイレクトする。 `SessionStatus` オブジェクトの setComplete メソッドを呼び出すことで、 セッションからは削除されるが、`Model` オブジェクトに同じオブジェクトが残っているため、 直接 View(JSP) を呼び出してしまうと、入力途中の情報が表示されてしまう。 そのため、**セッションから削除したうえで、入力画面を表示するためのリクエストへ、リダイレクトする必要がある。**
(4)	入力画面を表示するためのハンドラメソッド。

@SessionAttributes を使った処理の実装例

より具体的な実装例については、Appendix の*@SessionAttributes アノテーションを使ったウィザード形式の画面遷移の実装例*を参照されたい。

Spring Framework の session スコープの Bean の使用

Spring Framework の session スコープの Bean は、
複数の Controller をまたいだ画面遷移において、データを持ち回る場合に使用する。

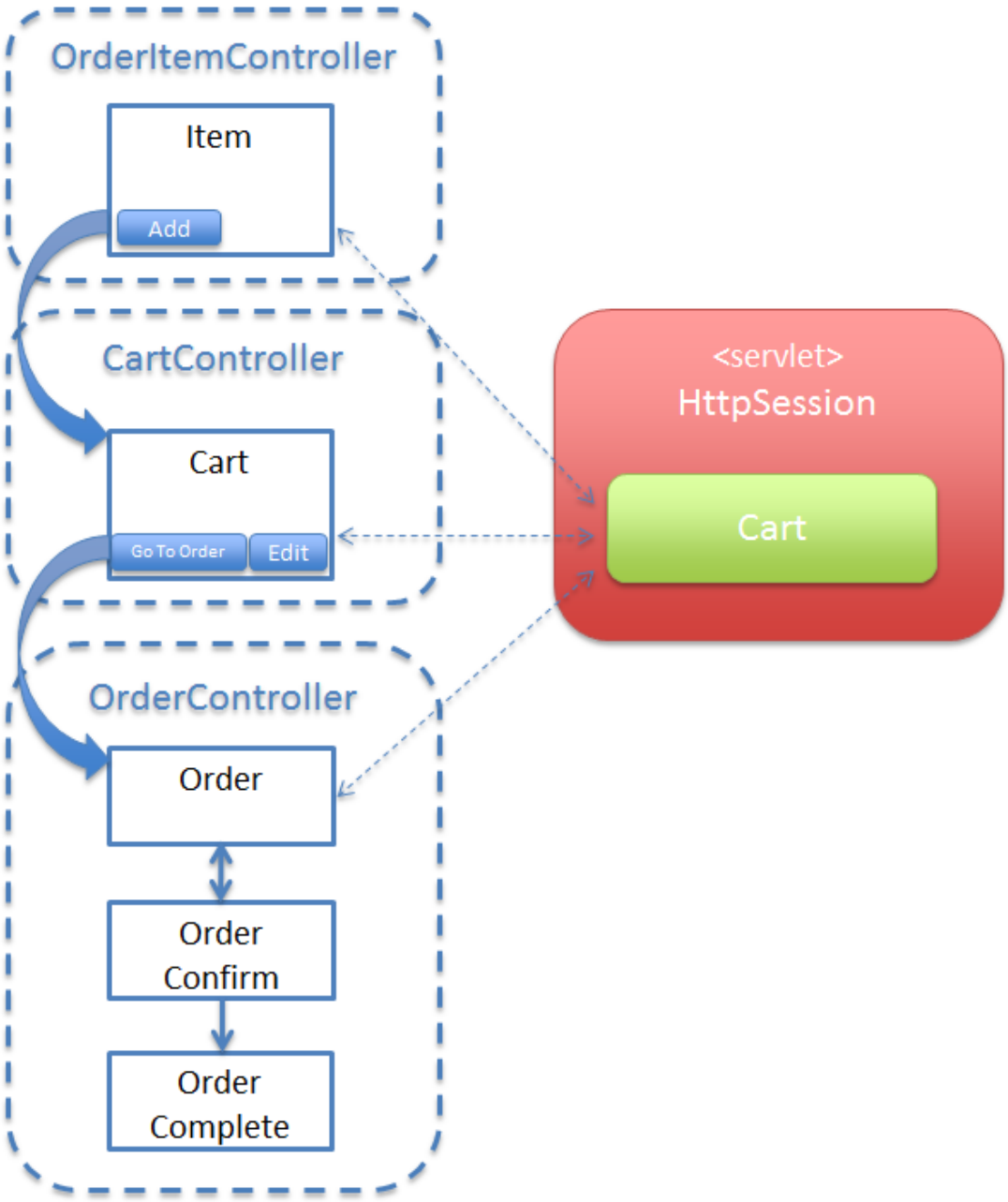
session スコープの Bean 定義

Spring Framework の session スコープの Bean を、定義する。

session スコープの Bean を定義する方法は、以下 2 種類の方法がある。

- component-scan を使用して bean を定義する。
- Bean 定義ファイル (XML) に bean を定義する。

component-scan を使用する方法を、以下に示す。



- クラス

```
@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS) // (1)
public class SessionCart implements Serializable {

    private static final long serialVersionUID = 1L;

    private Cart cart;

    public Cart getCart() {
        if (cart == null) {
            cart = new Cart();
        }
        return cart;
```

```
    }

    public void setCart(Cart cart) {
        this.cart = cart;
    }

    public void clearCart() { // (2)
        cart.clearCart();
    }
}
```

項番	説明
(1)	Bean のスコープを `session` にする。また、proxyMode 属性で `ScopedProxyMode.TARGET_CLASS` を指定し、scoped-proxy を有効にする。
(2)	注文が完了した際にカートの状態をクリア (カート内の商品を削除) するためのメソッドを用意する。

注釈: JPA で扱う Entity クラスを session スコープの Bean として定義したい場合は、直接 session スコープの Bean として定義するのではなく、ラッパークラスを用意することを推奨する。

JPA で扱う Entity クラスを session スコープの Bean として定義すると、JPA の API で session スコープの Bean を直接扱うことができない (直接あつかうと、エラーとなる)。そのため、JPA で扱うことができる Entity オブジェクトへの変換処理が、必要になってしまう。

上記例では、`Cart` という JPA の Entity クラスを、`SessionCart` というラッパークラスに包んで、session スコープの Bean としている。こうすることで、JPA で扱うことができる Entity オブジェクトへの変換処理が不要となるため、Controller で行う処理がシンプルになる。

注釈: scoped-proxy を有効化する理由について

session スコープの Bean を singleton スコープの Controller に Inject するために、scoped-proxy を有効化する必要がある。

- spring-mvc.xml

```
<context:component-scan base-package="xxx.yyy.zzz.app" /> // (2)
```

項番	説明
(2)	<context:component-scan> 要素でベースとなるパッケージを指定する。

Bean 定義ファイル (XML) に定義する方法を、以下に示す。

- JavaBean

```
<beans:bean id="sessionCart" class="xxx.yyy.zzz.app.SessionCart"
            scope="session"> <!-- (3) -->
    <aop:scoped-proxy /> <!-- (4) -->
</beans:bean>
```

項番	説明
(3)	Bean のスコープを session にする。
(4)	<aop:scoped-proxy /> 要素を指定し、scoped-proxy を有効にする。

session スコープの Bean の利用

session スコープの Bean を利用して、オブジェクトをセッションに格納・取得する場合は、
session スコープの Bean を、Controller に Inject する。

```
@Inject
SessionCart sessionCart; // (1)

@RequestMapping(value = "add")
public String addCart(@Validated ItemForm form, BindingResult result) {
    if (result.hasErrors()) {
        return "item/item";
    }
    CartItem cartItem = beanMapper.map(form, CartItem.class);
    Cart addedCart = cartService.addCartItem(sessionCart.getCart(), // (2)
            cartItem);
    sessionCart.setCart(addedCart); // (3)
```

```
    return "redirect:/cart";
}
```

項番	説明
(1)	session スコープの Bean を、Controller に Inject する。
(2)	session スコープの Bean のメソッド呼び出しを行うと、セッションに格納されているオブジェクトが返却される。 セッションにオブジェクトが格納されていない場合は、新たに生成されたオブジェクトが返却され、セッションにも格納される。 上記例では、カートに追加する前に在庫数などのチェックを行うため、Service のメソッドを呼び出している。
(3)	上記例では、`CartService` の addCartItem メソッドの引数に渡した `Cart` オブジェクトと、 返り値で返却される `Cart` オブジェクトが、別のインスタンスになる可能性があるため、 返却された `Cart` オブジェクトを session スコープの Bean に設定している。

注釈: **View(JSP) から session スコープの Bean を参照する方法**

SpEL(Spring Expression Language) 式を用いることで Controller において `Model` オブジェクトへ Bean を追加しなくても、JSP から session スコープの Bean を参照することができる。

```
<spring:eval var="cart" expression="@sessionCart.cart" />      <%-- (1) --%>

<%-- omitted --%>

<c:forEach var="item" items="${cart.cartItems}">      <%-- (2) --%>
    <tr>
        <td>${f:h(item.id)}</td>
        <td>${f:h(item.itemCode)}</td>
        <td>${f:h(item.quantity)}</td>
    </tr>
</c:forEach>
```

項番	説明
(1)	session スコープの Bean を参照する
(2)	session スコープの Bean を表示する

セッションに格納したオブジェクトの削除

不要になったオブジェクトをセッション上から削除する場合は、session スコープの Bean のフィールドをリセットする。

注釈: session スコープの Bean は、セッションが切れる時に DI コンテナによって破棄される。

DI コンテナが session スコープの Bean のライフサイクルを管理しているので、Bean 自体の破棄は DI コンテナにまかせる。

```
@Controller
@RequestMapping("order")
public class OrderController {

    @Inject
    SessionCart sessionCart; // (1)

    // ...

    @RequestMapping(method = RequestMethod.POST)
    public String order() {
        // ...
        return "redirect:/order?complete";
    }

    @RequestMapping(params = "complete", method = RequestMethod.GET)
    public String complete(Model model, SessionStatus sessionStatus) {
        sessionCart.clearCart(); // (2)
        return "order/complete";
    }

}
```

項番	説明
(1)	session スコープの Bean をインジェクションする。
(2)	session スコープの Bean の状態をクリアし、注文済みの商品をカートから削除する

session スコープの Bean を使った処理の実装例

より具体的な実装例については、Appendix の *session スコープの Bean を使った複数の Controller を跨いだ画面遷移の実装例*を参照されたい。

セッション操作のデバッグログ出力

セッションに対して行われた操作を、デバッグログに出力するクラスを、共通ライブラリとして提供している。セッションに対する操作が、想定通りに動作しているか確認する必要がある場合に、このクラスで出力するログが有効である。

共通ライブラリの詳細は、*HttpSessionEventLoggingListener* を参照されたい。

JSP の暗黙オブジェクト sessionScope を使用する

JSP の暗黙オブジェクトである `sessionScope` を使用する場合は、page ディレクティブの session 属性の値を `true` にする必要がある。ブランクプロジェクトから提供している `include.jsp` では、 `false` となっている。

`include.jsp` は、 `src/main/webapp/WEB-INF/views/common` ディレクトリに格納されている。

- `include.jsp`

```
<%@ page session="true"%>      <%-- (1) --%>

<%-- omitted --%>
```

項番	説明
(1)	page ディレクティブの session 属性の値を `true` にする。

4.3.3 How to extend

同一セッション内のリクエストの同期化

`@SessionAttributes` アノテーション、または session スコープの Bean を使用する場合は、同一セッション内のリクエストを同期化することを推奨する。

同期化しない場合、セッションに格納されているオブジェクトに、同時にアクセスする可能性があるため、想定外のエラーや、動作を引き起こす原因になりうる。

例えば、入力チェック済みのフォームオブジェクトに対して、不正な値が設定される可能性がある。

これを防ぐ方法として、`org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter` の、synchronizeOnSession を true にして、同一セッション内のリクエストを同期化することを、強く推奨する。

以下のような BeanPostProcessor を作成し、Bean 定義することで実現できる。

- コンポーネント

```
package com.example.app.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.web.servlet.mvc.method.annotation.
↪RequestMappingHandlerAdapter;

public class EnableSynchronizeOnSessionPostProcessor
    implements BeanPostProcessor {
    private static final Logger logger = LoggerFactory
            .getLogger(EnableSynchronizeOnSessionPostProcessor.class);

    @Override
    public Object postProcessBeforeInitialization(Object bean, String␣
↪beanName)
        throws BeansException {
        // NO-OP
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        if (bean instanceof RequestMappingHandlerAdapter) {
            RequestMappingHandlerAdapter adapter =
                (RequestMappingHandlerAdapter) bean;
            logger.info("enable synchronizeOnSession => {}", adapter);
            adapter.setSynchronizeOnSession(true); // (1)
        }
```

```
        return bean;
    }

}
```

項番	説明
(1)	org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter の setSynchronizeOnSession メソッドの引数に、true を指定すると、同一セッション内でのリクエストが同期化される。

- spring-mvc.xml

```
<bean class="com.example.app.config.EnableSynchronizeOnSessionPostProcessor" /
↪> <!-- (2) -->
```

項番	説明
(2)	(1) で作成した、BeanPostProcessor を Bean 定義する。

4.3.4 Appendix

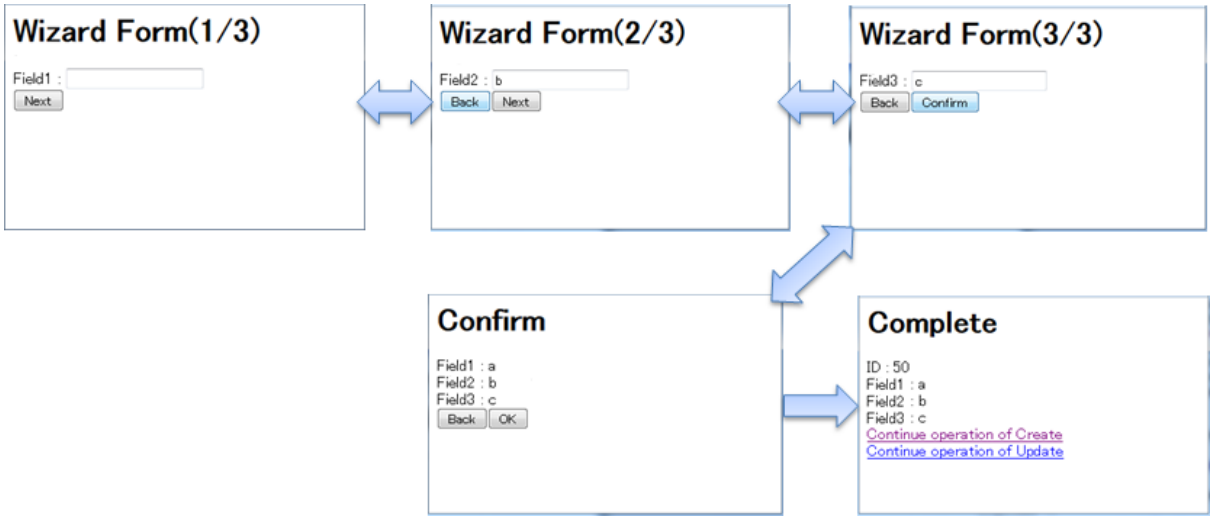
@SessionAttributes アノテーションを使ったウィザード形式の画面遷移の実装例

ウィザード形式の画面遷移を行う処理を例に、@SessionAttributes アノテーションを使った実装の説明を行う。

処理の仕様は、以下の通りとする。

- Entity の登録と、更新を行うための画面を提供する。
- 入力画面は、3 画面で構成され、各画面で 1 項目ずつ入力を行う。
- 入力した値は、保存 (登録/更新) する前に、確認画面で確認できる。
- 入力チェックは、画面遷移するタイミングで行い、エラーがある場合は、入力画面に戻る。
- 保存 (登録/更新) する前に、すべての入力値に対する入力チェックを再度行い、エラーがある場合は、不正操作を通知するエラー画面を表示する。
- すべての入力値に対するチェックが妥当な場合は、入力データをデータベースに保存する。

基本的な画面遷移は、以下の通りとする。



実装例は、以下の通りである。

- フォームオブジェクト

```
public class WizardForm implements Serializable {

    private static final long serialVersionUID = 1L;

    // (1)
    @NotEmpty(groups = { Wizard1.class })
    private String field1;

    // (2)
    @NotEmpty(groups = { Wizard2.class })
    private String field2;

    // (3)
    @NotEmpty(groups = { Wizard3.class })
    private String field3;

    // ...

    // (4)
    public static interface Wizard1 {
    }

    // (5)
    public static interface Wizard2 {
    }

    // (6)
    public static interface Wizard3 {
    }
```

```
}
```

項番	説明
(1)	1 ページ目の入力画面で入力するフィールド。
(2)	2 ページ目の入力画面で入力するフィールド。
(3)	3 ページ目の入力画面で入力するフィールド。
(4)	1 ページ目の入力画面で入力されるフィールドであることを示すための、検証グループインタフェース。
(5)	2 ページ目の入力画面で入力されるフィールドであることを示すための、検証グループインタフェース。
(6)	3 ページ目の入力画面で入力されるフィールドであることを示すための、検証グループインタフェース。

注釈: 検証グループについて

画面遷移時の入力チェックでは､該当ページのフィールドのみチェックする必要がある。Bean Validation では、検証グループを表すクラス、またはインタフェースを設けることで、検証するルールをグループ化することができる。今回の実装例のケースでは、画面毎に検証グループを用意することで、画面毎の入力チェックを実現している。

- Controller

```
@Controller
@RequestMapping("wizard")
@SessionAttributes(types = { WizardForm.class, Entity.class }) // (7)
public class WizardController {
```

```
    @Inject
    WizardService wizardService;

    @Inject
    Mapper beanMapper;
```

項番	説明
(7)	上記例では、フォームオブジェクト (WizardForm.class) と、エンティティ (Entity.class) のオブジェクトを、セッションに格納する。

```
@ModelAttribute("wizardForm") // (8)
public WizardForm setUpWizardForm() {
    return new WizardForm();
}
```

項番	説明
(8)	上記例では、セッションに格納するフォームオブジェクト (WizardForm) を生成している。無駄なオブジェクトの生成をなくすために、@ModelAttribute アノテーションの value 属性を指定している。

```
// (9)
@RequestMapping(value = "create", method = RequestMethod.GET)
public String initializeCreateWizardForm(SessionStatus sessionStatus) {
    sessionStatus.setComplete();
    return "redirect:/wizard/create?form1";
}

// (10)
@RequestMapping(value = "create", params = "form1")
public String createForm1() {
    return "wizard/form1";
}
```

項番	説明
(9)	登録用入力画面を、初期表示するためのハンドラメソッド。 操作途中のオブジェクトが、セッションに格納されている可能性があるため、このハンドラメソッドで、セッションに格納されているオブジェクトを削除しておく。
(10)	1 ページ目の登録用入力画面を、表示するためのハンドラメソッド。

```
// (11)
@RequestMapping(value = "{id}/update", method = RequestMethod.GET)
public String initializeUpdateWizardForm(@PathVariable("id") Integer id,
        RedirectAttributes redirectAttributes, SessionStatus sessionStatus) {
    sessionStatus.setComplete();
    return "redirect:/wizard/{id}/update?form1";
}

// (12)
@RequestMapping(value = "{id}/update", params = "form1")
public String updateForm1(@PathVariable("id") Integer id, WizardForm form,
        Model model) {
    Entity loadedEntity = wizardService.getEntity(id);
    beanMapper.map(loadedEntity, form); // (13)
    model.addAttribute(loadedEntity); // (14)
    return "wizard/form1";
}
```

項番	説明
(11)	更新用入力画面を、初期表示するためのハンドラメソッド。
(12)	1 ページ目の更新用入力画面を、表示するためのハンドラメソッド。
(13)	取得したエンティティの状態をフォームオブジェクトに設定する。上記例では、Dozer という Bean マッパーライブラリを使用している。
(14)	取得したエンティティを `Model` オブジェクトに追加し、セッションに格納する。 上記例では、`entity` という属性名で、セッションに格納される。

```
// (15)
@RequestMapping(value = "save", params = "form2", method = RequestMethod.POST)
public String saveForm2(@Validated(Wizard1.class) WizardForm form, // (16)
        BindingResult result) {
    if (result.hasErrors()) {
        return saveRedoForm1();
    }
    return "wizard/form2";
}

// (17)
@RequestMapping(value = "save", params = "form3", method = RequestMethod.POST)
public String saveForm3(@Validated(Wizard2.class) WizardForm form, // (18)
        BindingResult result) {
    if (result.hasErrors()) {
        return saveRedoForm2();
    }
    return "wizard/form3";
}

// (19)
@RequestMapping(value = "save", params = "confirm", method = RequestMethod.
↪POST)
public String saveConfirm(@Validated(Wizard3.class) WizardForm form, // (20)
        BindingResult result) {
    if (result.hasErrors()) {
        return saveRedoForm3();
    }
    return "wizard/confirm";
}
```

項番	説明
(15)	2 ページ目の入力画面を、表示するためのハンドラメソッド。
(16)	1 ページ目の入力画面で入力された値のみ、入力チェックするために、`@Validated` アノテーションの value 属性に、1 ページ目の入力画面の検証グループ (`Wizard1.class`) を指定する。
(17)	3 ページ目の入力画面を、表示するためのハンドラメソッド。
(18)	2 ページ目の入力画面で入力された値のみ、入力チェックするために、`@Validated` アノテーションの value 属性に、2 ページ目の入力画面の検証グループ (`Wizard2.class`) を指定する。
(19)	確認画面を表示するためのハンドラメソッド。
(20)	3 ページ目の入力画面で入力された値のみ、入力チェックするために、`@Validated` アノテーションの value 属性に、3 ページ目の入力画面の検証グループ (`Wizard3.class`) を指定する。

```
// (21)
@RequestMapping(value = "save", method = RequestMethod.POST)
public String save(@ModelAttribute @Validated({ Wizard1.class,
        Wizard2.class, Wizard3.class }) WizardForm form, // (22)
        BindingResult result,
        Entity entity, // (23)
        RedirectAttributes redirectAttributes) {
    if (result.hasErrors()) {
        throw new InvalidRequestException(result); // (24)
    }

    beanMapper.map(form, entity);

    entity = wizardService.saveEntity(entity); // (25)
```

```
    redirectAttributes.addFlashAttribute(entity); // (26)

    return "redirect:/wizard/save?complete";
}

// (27)
@RequestMapping(value = "save", params = "complete", method = RequestMethod.
↪GET)
public String saveComplete(SessionStatus sessionStatus) {
    sessionStatus.setComplete();
    return "wizard/complete";
}
```

項番	説明
(21)	保存処理を実行するためのハンドラメソッド。
(22)	入力画面で入力された値を全てチェックするために、`@Validated` アノテーションの value 属性に、各入力画面の検証グループインタフェース (`Wizard1.class`, `Wizard2.class`, `Wizard3.class`) を指定する。
(23)	保存する `Entity.class` のオブジェクトを取得する。 登録処理の場合は、新たに生成されたオブジェクト、更新処理の場合は、(14) の処理でセッションに格納したオブジェクトが取得される。
(24)	アプリケーションが提供しているボタンを使って、画面遷移を行っていれば、このタイミングでエラーは発生しないので、不正な操作が行われた場合に `InvalidRequestException` が throw される。 なお、`InvalidRequestException` は共通ライブラリから提供している例外クラスではないため、別途作成する必要がある。
(25)	入力値が反映された `Entity.class` のオブジェクトを保存する。
(26)	リダイレクト先のハンドラメソッドで保存した `Entity.class` のオブジェクトを参照できるようにするために、Flash スコープに格納する。
(27)	完了画面を表示するためのハンドラメソッド。

```
    // (28)
    @RequestMapping(value = "save", params = "redoForm1")
    public String saveRedoForm1() {
        return "wizard/form1";
    }

    // (29)
    @RequestMapping(value = "save", params = "redoForm2")
```

```
    public String saveRedoForm2() {
        return "wizard/form2";
    }

    // (30)
    @RequestMapping(value = "save", params = "redoForm3")
    public String saveRedoForm3() {
        return "wizard/form3";
    }

}
```

項番	説明
(28)	1 ページ目の入力画面を、再表示するためのハンドラメソッド。
(29)	2 ページ目の入力画面を、再表示するためのハンドラメソッド。
(30)	3 ページ目の入力画面を、再表示するためのハンドラメソッド。

- Controller の全ソース

```
@Controller
@RequestMapping("wizard")
@SessionAttributes(types = { WizardForm.class, Entity.class })
// (7)
public class WizardController {

    @Inject
    EntityService wizardService;

    @Inject
    Mapper beanMapper;

    @ModelAttribute("wizardForm")
    // (8)
    public WizardForm setUpWizardForm() {
        return new WizardForm();
    }

    // (9)
    @RequestMapping(value = "create", method = RequestMethod.GET)
    public String initializeCreateWizardForm(SessionStatus sessionStatus) {
        sessionStatus.setComplete();
```

```
        return "redirect:/wizard/create?form1";
    }

    // (10)
    @RequestMapping(value = "create", params = "form1")
    public String createForm1() {
        return "wizard/form1";
    }

    // (11)
    @RequestMapping(value = "{id}/update", method = RequestMethod.GET)
    public String initializeUpdateWizardForm(@PathVariable("id") Integer id,
            RedirectAttributes redirectAttributes, SessionStatus␣
↪sessionStatus) {
        sessionStatus.setComplete();
        return "redirect:/wizard/{id}/update?form1";
    }

    // (12)
    @RequestMapping(value = "{id}/update", params = "form1")
    public String updateForm1(@PathVariable("id") Integer id, WizardForm form,
            Model model) {
        Entity loadedEntity = wizardService.getEntity(id);
        beanMapper.map(loadedEntity, form); // (13)
        model.addAttribute(loadedEntity); // (14)
        return "wizard/form1";
    }

    // (15)
    @RequestMapping(value = "save", params = "form2", method = RequestMethod.
↪POST)
    public String saveForm2(@Validated(Wizard1.class) WizardForm form, // (16)
            BindingResult result) {
        if (result.hasErrors()) {
            return saveRedoForm1();
        }
        return "wizard/form2";
    }

    // (17)
    @RequestMapping(value = "save", params = "form3", method = RequestMethod.
↪POST)
    public String saveForm3(@Validated(Wizard2.class) WizardForm form, // (18)
            BindingResult result) {
        if (result.hasErrors()) {
            return saveRedoForm2();
        }
        return "wizard/form3";
    }

    // (19)
    @RequestMapping(value = "save", params = "confirm", method =␣
↪RequestMethod.POST)
```

```
    public String saveConfirm(@Validated(Wizard3.class) WizardForm form, //␣
↪(20)
            BindingResult result) {
        if (result.hasErrors()) {
            return saveRedoForm3();
        }
        return "wizard/confirm";
    }

    // (21)
    @RequestMapping(value = "save", method = RequestMethod.POST)
    public String save(@ModelAttribute @Validated({ Wizard1.class,
            Wizard2.class, Wizard3.class }) WizardForm form, // (22)
            BindingResult result, Entity entity, // (23)
            RedirectAttributes redirectAttributes) {
        if (result.hasErrors()) {
            throw new InvalidRequestException(result); // (24)
        }

        beanMapper.map(form, entity);

        entity = wizardService.saveEntity(entity); // (25)

        redirectAttributes.addFlashAttribute(entity); // (26)

        return "redirect:/wizard/save?complete";
    }

    // (27)
    @RequestMapping(value = "save", params = "complete", method =␣
↪RequestMethod.GET)
    public String saveComplete(SessionStatus sessionStatus) {
        sessionStatus.setComplete();
        return "wizard/complete";
    }

    // (28)
    @RequestMapping(value = "save", params = "redoForm1")
    public String saveRedoForm1() {
        return "wizard/form1";
    }

    // (29)
    @RequestMapping(value = "save", params = "redoForm2")
    public String saveRedoForm2() {
        return "wizard/form2";
    }

    // (30)
    @RequestMapping(value = "save", params = "redoForm3")
    public String saveRedoForm3() {
        return "wizard/form3";
    }
```

```
}
```

- 1 ページ目の入力画面 (JSP)

```
<html>
<head>
<title>Wizard Form(1/3)</title>
</head>
<body>
    <h1>Wizard Form(1/3)</h1>
    <form:form action="${pageContext.request.contextPath}/wizard/save"
        modelAttribute="wizardForm">
        <form:label path="field1">Field1</form:label> :
        <form:input path="field1" />
        <form:errors path="field1" />
        <div>
            <form:button name="form2">Next</form:button>
        </div>
    </form:form>
</body>
</html>
```

- 2 ページ目の入力画面 (JSP)

```
<html>
<head>
<title>Wizard Form(2/3)</title>
</head>
<body>
    <h1>Wizard Form(2/3)</h1>
    <%-- (31) --%>
    <form:form action="${pageContext.request.contextPath}/wizard/save"
        modelAttribute="wizardForm">
        <form:label path="field2">Field2</form:label> :
        <form:input path="field2" />
        <form:errors path="field2" />
        <div>
            <form:button name="redoForm1">Back</form:button>
            <form:button name="form3">Next</form:button>
        </div>
    </form:form>
</body>
</html>
```

項番	説明
(31)	フォームオブジェクトをセッションに格納しているため、1 ページ目の入力画面のフィールドを、hidden 項目にする必要はない。

- 3 ページ目の入力画面 (JSP)

```
<html>
<head>
<title>Wizard Form(3/3)</title>
</head>
<body>
    <h1>Wizard Form(3/3)</h1>
    <%-- (32) --%>
    <form:form action="${pageContext.request.contextPath}/wizard/save"
        modelAttribute="wizardForm">
        <form:label path="field3">Field3</form:label> :
        <form:input path="field3" />
        <form:errors path="field3" />
        <div>
            <form:button name="redoForm2">Back</form:button>
            <form:button name="confirm">Confirm</form:button>
        </div>
    </form:form>
</body>
</html>
```

項番	説明
(32)	フォームオブジェクトをセッションに格納しているため、1 ページ目と 2 ページ目の入力画面のフィールドを、hidden 項目にする必要はない。

- 確認画面 (JSP)

```
<html>
<head>
<title>Confirm</title>
</head>
<body>
    <h1>Confirm</h1>
    <%-- (33) --%>
    <form:form action="${pageContext.request.contextPath}/wizard/save"
        modelAttribute="wizardForm">
        <div>
            Field1 : ${f:h(wizardForm.field1)}
        </div>
        <div>
            Field2 : ${f:h(wizardForm.field2)}
        </div>
        <div>
            Field3 : ${f:h(wizardForm.field3)}
        </div>
        <div>
            <form:button name="redoForm3">Back</form:button>
```

```
            <form:button>OK</form:button>
        </div>
    </form:form>
</body>
</html>
```

項番	説明
(33)	フォームオブジェクトをセッションに格納しているため、入力画面のフィールドを、hidden 項目にする必要はない。

- 完了画面 (JSP)

```
<html>
<head>
<title>Complete</title>
</head>
<body>
    <h1>Complete</h1>
    <div>
        <div>
            ID : ${f:h(entity.id)}
        </div>
        <div>
            Field1 : ${f:h(entity.field1)}
        </div>
        <div>
            Field2 : ${f:h(entity.field2)}
        </div>
        <div>
            Field3 : ${f:h(entity.field3)}
        </div>
    </div>
    <div>
        <a href="${pageContext.request.contextPath}/wizard/create">
            Continue operation of Create
        </a>
    </div>
    <div>
        <a href="${pageContext.request.contextPath}/wizard/${entity.id}/update
↪">
            Continue operation of Update
        </a>
    </div>
</body>
</html>
```

- spring-mvc.xml

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">
    <property name="exceptionCodeResolver" ref="exceptionCodeResolver" />
    <!-- ... -->
    <property name="exceptionMappings">
        <map>
            <!-- ... -->
            <entry key="InvalidRequestException"
                   value="common/error/operationError" /> <!-- (34) -->
        </map>
    </property>
    <property name="statusCodes">
        <map>
            <!-- ... -->
            <entry key="common/error/operationError" value="400" /> <!-- (35)
↪-->
        </map>
    </property>
    <!-- ... -->
</bean>
```

項番	説明
(34)	共通ライブラリから提供している SystemExceptionResolver の exceptionMappings に、保存処理実行時に不正なリクエストを検知したことを、通知する例外 InvalidRequestException の、例外ハンドリングの定義を追加する。 上記例では、 例外発生時の遷移先の View 名として、/WEB-INF/views/common/error/operationError.jsp を指定している。
(35)	SystemExceptionResolver の statusCodes に、HttpSessionRequiredException 発生時の HTTP レスポンスコードを指定する。 上記例では、 例外発生時の HTTP レスポンスコードとして、Bad Request(400) を指定している。

- applicationContext.xml

```
<bean id="exceptionCodeResolver"
    class="org.terasoluna.gfw.common.exception.
↪SimpleMappingExceptionCodeResolver">
    <!-- Setting and Customization by project. -->
    <property name="exceptionMappings">
        <map>
            <!-- ... -->
            <entry key="InvalidRequestException" value="w.xx.0004" /> <!--
↪(36) -->
        </map>
```

```
    </property>
    <property name="defaultExceptionCode" value="e.xx.0001" /> <!-- (37) -->
</bean>
```

項番	説明
(36)	共通ライブラリから提供している SimpleMappingExceptionCodeResolver の exceptionMappings に、InvalidRequestException の例外ハンドリングの定義を追加する。 上記例では、 例外発生時の例外コードとして、w.xx.0004 を指定している。 この設定を追加しない場合は、デフォルトの例外コードが、ログに出力される。
(37)	例外発生時のデフォルトの例外コード。

session スコープの Bean を使った複数の Controller を跨いだ画面遷移の実装例

複数の Controller をまたいで画面遷移を行う処理を例に、session スコープの Bean を使った実装の説明を行う。

処理の仕様は、以下の通りとする。

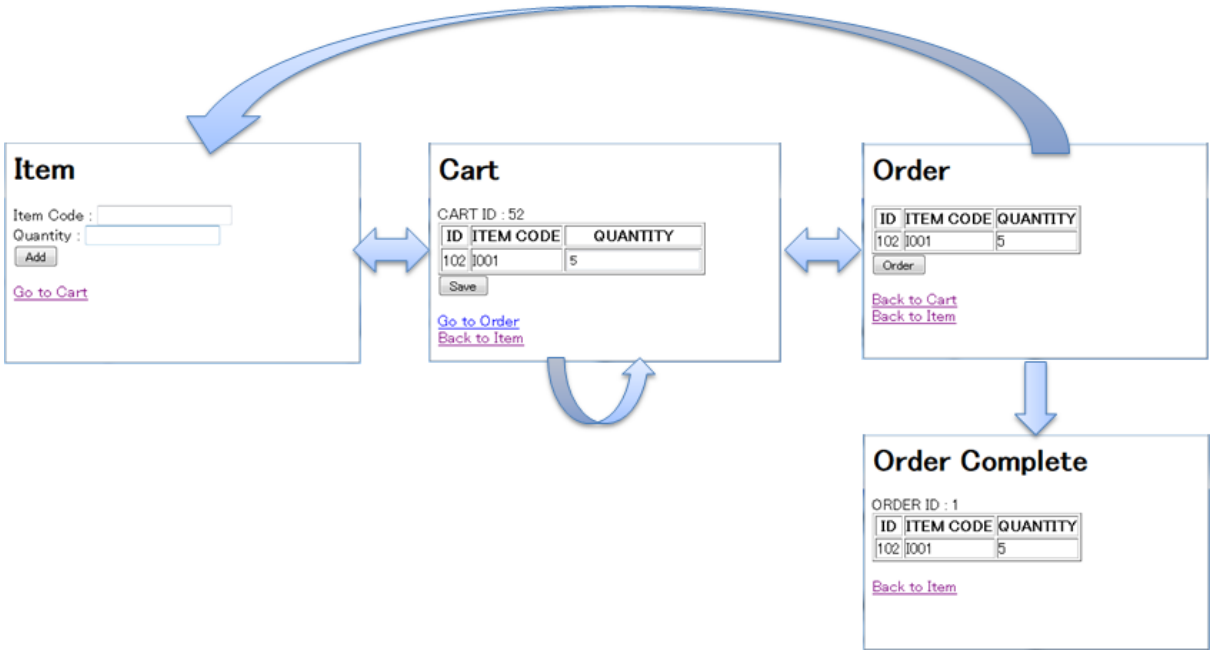
- 商品をカートに追加する処理を提供する。
- カートに追加されている商品の、数量変更を行う処理を提供する。
- カートに格納されている商品を、注文する処理を提供する。
- 上記 3 つの処理は、それぞれ独立した機能として提供するため、別 Controller(ItemController, CartController, OrderController) とする。
- カートは、上記 3 つの処理で共有するため、セッションに格納する。
- 商品をカートに追加した場合は、カート画面に遷移する。

画面遷移は、以下の通りとする。

実装例は、以下の通りである。

- session スコープの Bean として定義する JavaBean

```
@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class SessionCart implements Serializable {

    private static final long serialVersionUID = 1L;
```




```
    private Cart cart; // (1)

    public Cart getCart() {
        if (cart == null) {
            cart = new Cart();
        }
        return cart;
    }

    public void setCart(Cart cart) {
        this.cart = cart;
    }

    public void clearCart() { // (2)
        cart.clearCart();
    }
}
```

項番	説明
(1)	`Cart` という Entity(Domain オブジェクト) をラップしている。
(2)	カートに追加された商品のオブジェクトを `cart` から削除し，カートが空の状態にする。

- ItemController

```
@Controller
@RequestMapping("item")
public class ItemController {

    @Inject
    SessionCart sessionCart;

    @Inject
    CartService cartService;

    @Inject
    Mapper beanMapper;

    @ModelAttribute
    public ItemForm setUpItemForm() {
        return new ItemForm();
    }

    // (3)
    @RequestMapping
    public String view(Model model) {
        return "item/item";
    }

    // (4)
    @RequestMapping(value = "add")
    public String addCart(@Validated ItemForm form, BindingResult result) {
        if (result.hasErrors()) {
            return "item/item";
        }
        CartItem cartItem = beanMapper.map(form, CartItem.class);
        Cart cart = cartService.addCartItem(sessionCart.getCart(), // (5)
                cartItem);
        sessionCart.setCart(cart); // (6)
        return "redirect:/cart"; // (7)
    }
}
```

項番	説明
(3)	商品画面を、表示するためのハンドラメソッド。
(4)	指定された商品を、カートに追加するためのハンドラメソッド。
(5)	セッションに格納されている `Cart` オブジェクトを、Service のメソッドに渡す。
(6)	Service のメソッドから返却された `Cart` オブジェクトを、session スコープの Bean に反映する。 session スコープの Bean に反映することで、`Cart` オブジェクトがセッションに格納される。
(7)	商品をカートに追加した後に、カート画面を表示するためのリクエストに、リダイレクトする。 **別 Controller の画面に遷移する場合は、直接 View(JSP) を呼び出すのではなく、画面を表示するためのリクエストにリダイレクトすることを推奨する。**

- CartController

```
@Controller
@RequestMapping("cart")
public class CartController {

    @Inject
    SessionCart sessionCart;

    @Inject
    CartService cartService;

    @Inject
    Mapper beanMapper;

    @ModelAttribute
    public CartForm setUpCartForm() {
        return new CartForm();
    }

    // (8)
    @RequestMapping
```

```
    public String cart(CartForm form) {
        beanMapper.map(sessionCart.getCart(), form);
        return "cart/cart";
    }

    // (9)
    @RequestMapping(params = "edit", method = RequestMethod.POST)
    public String edit(@Validated CartForm form, BindingResult result,
            Model model) {
        if (result.hasErrors()) {
            return "cart/cart";
        }

        Cart cart = sessionCart.getCart();
        Iterator<CartItemForm> itemForm = form.getCartItems().iterator();
        for (CartItem item : cart.getCartItems()) {
            beanMapper.map(itemForm.next(), item);
        }

        cart = cartService.saveCart(cart);
        sessionCart.setCart(cart); // (10)

        return "redirect:/cart"; // (11)
    }


}
```

項番	説明
(8)	カート画面 (数量変更画面) を表示するためのハンドラメソッド。
(9)	数量変更を、行うためのハンドラメソッド。
(10)	Service のメソッドから返却された `Cart` オブジェクトを session スコープの Bean に反映する。 session スコープの Bean に反映することで、セッションに反映される。
(11)	数量変更を行った後に、カート画面 (数量変更画面) を表示するためのリクエストに、リダイレクトする。 **更新処理を行った場合は、直接 View(JSP) を呼び出すのではなく、画面を表示するためのリクエストにリダイレクトすることを推奨する。**

- OrderController

```
@Controller
@RequestMapping("order")
public class OrderController {

    @Inject
    SessionCart sessionCart;

    @ModelAttribute
    public OrderForm setUpOrderForm() {
        return new OrderForm();
    }

    // (12)
    @RequestMapping
    public String view() {
        return "order/order";
    }

    // (13)
    @RequestMapping(method = RequestMethod.POST)
    public String order() {
        // ...
        return "redirect:/order?complete";
    }
```

```
    // (14)
    @RequestMapping(params = "complete", method = RequestMethod.GET)
    public String complete(Model model, SessionStatus sessionStatus) {
        sessionCart.clearCart();
        return "order/complete";
    }

}
```

項番	説明
(12)	注文画面を、表示するためのハンドラメソッド。
(13)	注文処理を行うためのハンドラメソッド。
(14)	注文完了画面を表示するためのハンドラメソッド。

- 商品画面 (JSP)

```
<html>
<head>
<title>Item</title>
</head>
<body>
    <h1>Item</h1>
    <form:form action="${pageContext.request.contextPath}/item/add"
        modelAttribute="itemForm">
        <form:label path="itemCode">Item Code</form:label> :
        <form:input path="itemCode" />
        <form:errors path="itemCode" />
        <br>
        <form:label path="quantity">Quantity</form:label> :
        <form:input path="quantity" />
        <form:errors path="quantity" />
        <div>
            <%-- (15) --%>
            <form:button>Add</form:button>
        </div>
    </form:form>
    <div>
        <a href="${pageContext.request.contextPath}/cart">Go to Cart</a>
    </div>
</body>
```

```
</html>
```

項番	説明
(15)	商品を追加するためのボタン。

- カート画面 (JSP)

```
<html>
<head>
<title>Cart</title>
</head>
<body>
    <%-- (16) --%>
    <spring:eval var="cart" experssion="@sessionCart.cart" />
    <h1>Cart</h1>
    <c:choose>
        <c:when test="${ empty cart.cartItems }">
            <div>Cart is empty.</div>
        </c:when>
        <c:otherwise>
            CART ID :
            ${f:h(cart.id)}
            <form:form modelAttribute="cartForm">
                <table border="1">
                    <thead>
                        <tr>
                            <th>ID</th>
                            <th>ITEM CODE</th>
                            <th>QUANTITY</th>
                        </tr>
                    </thead>
                    <tbody>
                        <c:forEach var="item"
                            items="${cart.cartItems}"
                            varStatus="rowStatus">
                            <tr>
                                <td>${f:h(item.id)}</td>
                                <td>${f:h(item.itemCode)}</td>
                                <td>
                                    <form:input
                                        path="cartItems[${rowStatus.index}].
↪quantity" />
                                    <form:errors
                                        path="cartItems[${rowStatus.index}].
↪quantity" />
                                </td>
                            </tr>
                        </c:forEach>
```

```
                    </tbody>
                </table>
                <%-- (17) --%>
                <form:button name="edit">Save</form:button>
            </form:form>
            <div>
                <%-- (18) --%>
                <a href="${pageContext.request.contextPath}/order">Go to Order
↪</a>
            </div>
        </c:otherwise>
    </c:choose>
    <div>
        <a href="${pageContext.request.contextPath}/item">Back to Item</a>
    </div>
</body>
</html>
```

項番	説明
(16)	SpEL 式を用いて session スコープの Bean を参照する。
(17)	数量を更新するためのボタン。
(18)	注文画面を表示するためのリンク。

- 注文画面 (JSP)

```
<html>
<head>
<title>Order</title>
</head>
<body>
    <spring:eval var="cart" experssion="@sessionCart.cart" />
    <h1>Order</h1>
    <table border="1">
        <thead>
            <tr>
                <th>ID</th>
                <th>ITEM CODE</th>
                <th>QUANTITY</th>
            </tr>
        </thead>
        <tbody>
```

```
            <c:forEach var="item" items="${cart.cartItems}"
                varStatus="rowStatus">
                <tr>
                    <td>${f:h(item.id)}</td>
                    <td>${f:h(item.itemCode)}</td>
                    <td>${f:h(item.quantity)}</td>
                </tr>
            </c:forEach>
        </tbody>
    </table>
    <form:form modelAttribute="orderForm">
        <%-- (19) --%>
        <form:button>Order</form:button>
    </form:form>
    <div>
        <a href="${pageContext.request.contextPath}/cart">Back to Cart</a>
    </div>
    <div>
        <a href="${pageContext.request.contextPath}/item">Back to Item</a>
    </div>
</body>
</html>
```

項番	説明
(19)	注文するためのボタン。

- 注文完了画面 (JSP)

```
<html>
<head>
<title>Order Complete</title>
</head>
<body>
    <h1>Order Complete</h1>
    ORDER ID :
    ${f:h(order.id)}
    <table border="1">
        <thead>
            <tr>
                <th>ID</th>
                <th>ITEM CODE</th>
                <th>QUANTITY</th>
            </tr>
        </thead>
        <tbody>
            <c:forEach var="item" items="${order.orderItems}"
                varStatus="rowStatus">
                <tr>
```

```
                    <td>${f:h(item.id)}</td>
                    <td>${f:h(item.itemCode)}</td>
                    <td>${f:h(item.quantity)}</td>
                </tr>
            </c:forEach>
        </tbody>
    </table>
    <br>
    <div>
        <a href="${pageContext.request.contextPath}/item">Back to Item</a>
    </div>
</body>
</html>
```

4.4 ページネーション

4.4.1 Overview

本章では、検索条件に一致するデータをページ分割して表示する方法 (ページネーション) について説明する。

検索条件に一致するデータが大量になる場合は、ページネーション機能を使用することを推奨する。

一度に大量のデータを取得し画面に表示すると、以下 3 点の問題が発生する可能性がある。

- サーバ側のメモリ枯渇の発生。

  単発のリクエストで問題が発生しなくても、同時に複数実行された場合に `java.lang.OutOfMemoryError` が発生する可能性がある。

- ネットワーク負荷の発生。

  不要なデータがネットワークに流れることで、ネットワーク全体にかかる負荷が高くなり、システム全体のレスポンスタイムに影響を与える可能性がある。

- 画面のレスポンス遅延の発生。

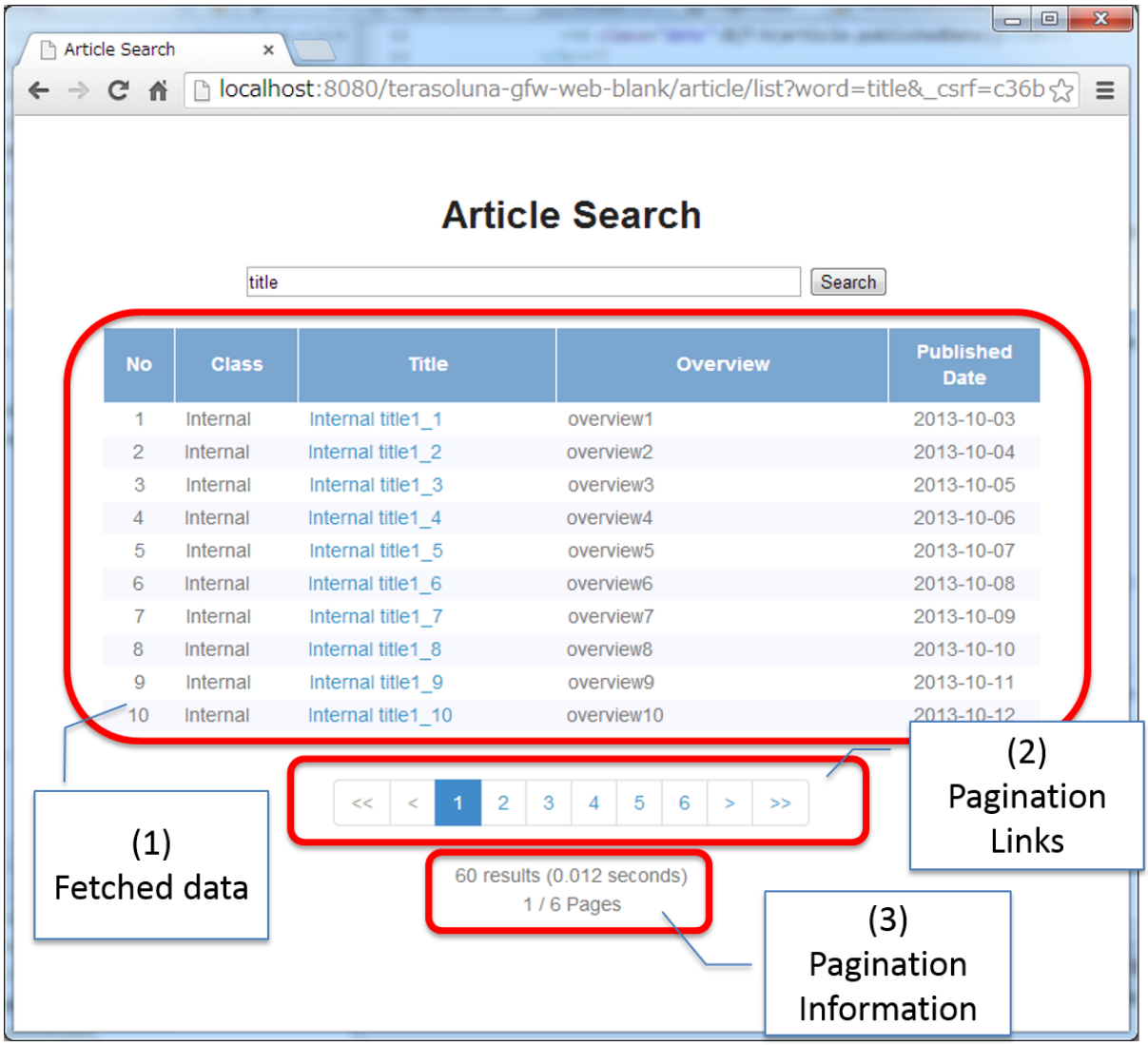
  大量のデータを扱う場合、サーバの処理、ネットワークのトラフィック処理、クライアントの描画処理の全てで時間がかかるため、画面のレスポンスが遅くなる可能性がある。

ページ分割時の一覧画面の表示について

ページネーション機能を利用してページ分割した場合、以下のような画面になる。

各要素の表示にはサーバ側で行う検索処理をページ検索に対応させる必要がある。ページ検索機能については、「ページ検索について 」を参照されたい。

また、各要素の概要及び HTML 構造等については、「ページネーションの表示について」を参照されたい。



項番	説明
(1)	ページ検索処理で取得したデータを表示する。
(2)	ページを移動するためのリンクを表示する。 リンク押下時には、該当ページを表示するためのリクエストを送信する。
(3)	ページネーションに関連する情報 (合計件数、合計ページ数、表示ページ数など) を表示する。

ページ検索について

ページネーションを実現する際には、まずサーバ側で行う検索処理をページ検索できるように実装する必要がある。

本ガイドラインでは、サーバ側のページ検索は、 Spring Data から提供されている仕組みを利用することを前提としている。

Spring Data 提供のページ検索機能について

Spring Data より提供されているページ検索用の機能は、以下の通り。

項番	説明
1	リクエストパラメータよりページ検索に必要な情報 (検索対象のページ位置、取得件数、ソート条件) を抽出し、抽出した情報を `org.springframework.data.domain.Pageable` のオブジェクトとして Controller の引数に引き渡す。 この機能は、 `org.springframework.data.web.PageableHandlerMethodArgumentResolver` クラスとして提供されており、`spring-mvc.xml` の `<mvc:argument-resolvers>` 要素に追加することで有効となる。 リクエストパラメータについては、「*Note* 欄 」を参照されたい。
2	ページ情報 (合計件数、該当ページのデータ、検索対象のページ位置、取得件数、ソート条件) を保持する。 この機能は、 `org.springframework.data.domain.Page` インタフェースとして提供されており、デフォルトの実装クラスとして `org.springframework.data.domain.PageImpl` が提供されている。 **共通ライブラリより提供しているページネーションリンクを出力するための JSP タグライブラリでは、 Page オブジェクトから必要なデータを取得する仕様となっている。**
3	データベースアクセスとして Spring Data JPA を使用する場合は、Repository の Query メソッドの引数に `Pageable` オブジェクトを指定することで、該当ページの情報が `Page` オブジェクトとして返却される。 合計件数を取得する SQL の発行、ソート条件の追加、該当ページに一致するデータの抽出などの処理が全て自動で行われる。 データベースアクセスとして、MyBatis を使用する場合は、Spring Data JPA が自動で行ってくれる処理を、Java(Service) 及び SQL マッピングファイル内で実装する必要がある。

注釈: ページ検索用のリクエストパラメータについて

Spring Data より提供されているページ検索用のリクエストパラメータは以下の 3 つとなる。

項番	パラメータ名	説明
1.	page	検索対象のページ位置を指定するためのリクエストパラメータ。 値には、0 以上の数値を指定する。 デフォルトの設定では、ページ位置の値は 「0」 から開始する。そのため、1 ページ目のデータを取得する場合は 「0」 を、2 ページ目のデータを取得する場合は 「1」 を指定する必要がある。
2.	size	取得する件数を指定するためのリクエストパラメータ。 値には、1 以上の数値を指定する。 `PageableHandlerMethodArgumentResolver` の `maxPageSize` に指定された値より大きい値が指定された場合は、 `maxPageSize` の値が `size` の値となる。
3.	sort	ソート条件を指定するためのパラメータ (複数指定可能)。 値には、{**ソート項目名** (**,** **ソート順**)} の形式で指定する。 ソート順には、`ASC` 又は `DESC` のどちらかの値を指定し、省略した場合は `ASC` が適用される。 項目名は 「,」 区切りで複数指定することが可能である。 例えば、クエリ文字列として `sort=lastModifiedDate,id,DESC&sort=subId` が指定された場合、 `ORDER BY lastModifiedDate DESC, id DESC, subId ASC` のような Order By 句を Query に追加することになる。

ページネーションの表示について

「ページ分割時の一覧画面の表示について」にて説明した画面の各要素について説明する。

取得データの表示について

ページ検索処理で検索条件 (検索対象のページ位置、取得件数、ソート条件等) を指定して取得したデータを表示する。

ページ検索については、「ページ検索について 」を参照されたい。

ページネーションリンクの表示について

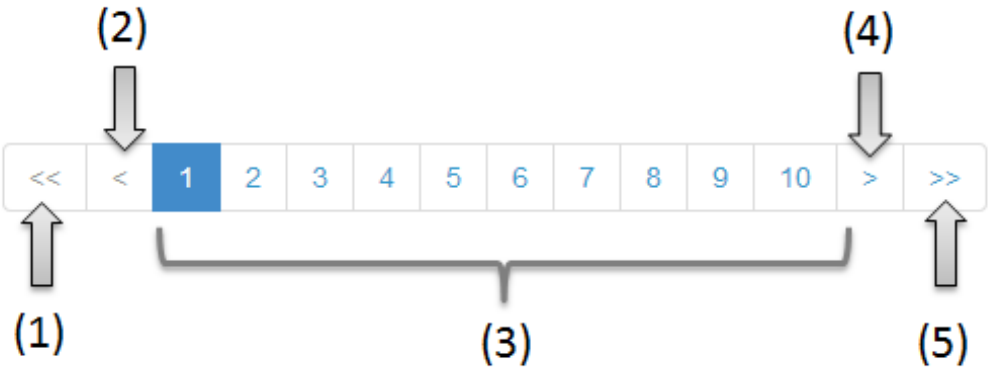
共通ライブラリから提供している JSP タグライブラリを使って出力されるページネーションリンクについて以下の流れで説明する。

1. ページネーションリンクの構成
2. ページネーションリンクの *HTML* 構造
3. *JSP* タブライブラリのパラメータについて

共通ライブラリからはページネーションリンクを表示するためのスタイルシートの提供は行っていないため、各プロジェクトにて用意すること。
以降の説明で使用する画面は、Bootstrap v3.0.0 のスタイルシートを適用している。

ページネーションリンクの構成

ページネーションリンクは、以下の要素から構成される。



項番	説明
(1)	最初のページに移動するためのリンク。
(2)	前のページに移動するためのリンク。
(3)	指定したページに移動するためのリンク。
(4)	次のページに移動するためのリンク。
(5)	最後のページに移動するためのリンク。

ページネーションリンクは、以下の状態をもつ。



項番	説明
(6)	現在表示しているページで操作することができないリンクであることを示す状態。 具体的には、1 ページ目を表示している時の「最初のページに移動するためのリンク」「前のページに移動するためのリンク」と、最終ページを表示している時の「次のページに移動するためのリンク」「最後のページに移動するためのリンク」がこの状態となる。 共通ライブラリから提供している JSP タグライブラリでは、この状態を `disabled` と定義している。
(7)	現在表示しているページであることを示す状態。 共通ライブラリから提供している JSP タグライブラリでは、この状態を `active` と定義している。

共通ライブラリを使って出力される HTML は、以下の構造となる。
図中の番号は、上記で説明した「ページネーションリンクの構成」と「ページネーションリンクの状態」の項番に対応させている。

- JSP

```
<t:pagination page="${page}" />
```

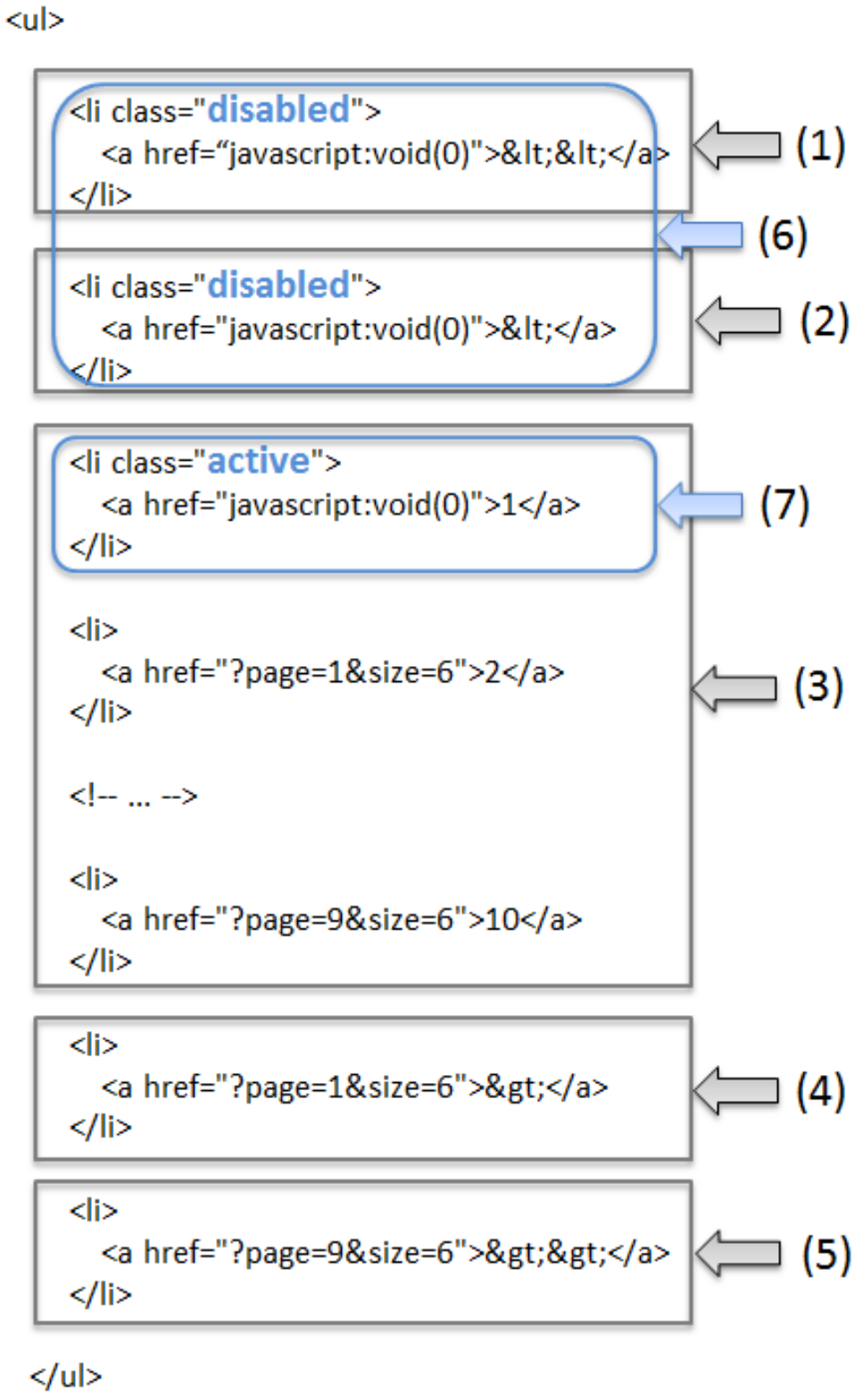
- 出力される HTML

ページネーションリンクの HTML 構造

共通ライブラリを使って出力されるページネーションリンクの HTML は、以下の構造となる。

- HTML

- 画面イメージ



<li class="disabled">
<<

(1)
(6)
<li class="disabled">
<

(2)
<li class="active">
1

(7)

2

(3)
<!-- ... -->

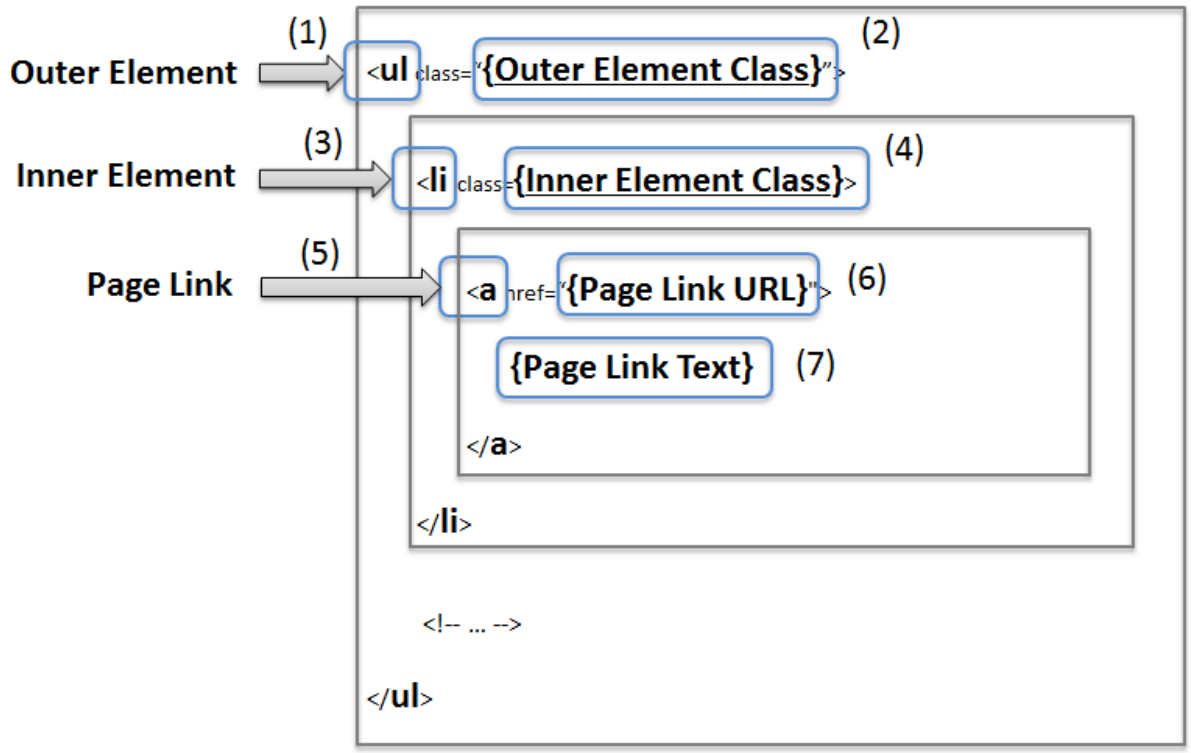
10

>

(4)

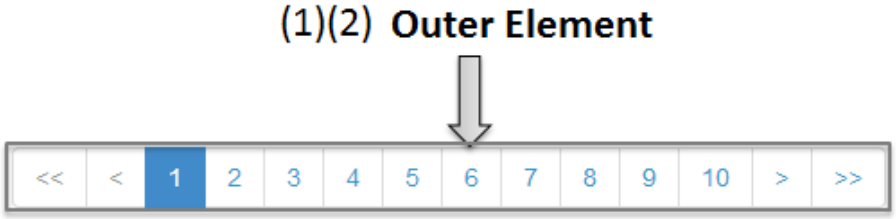
>>

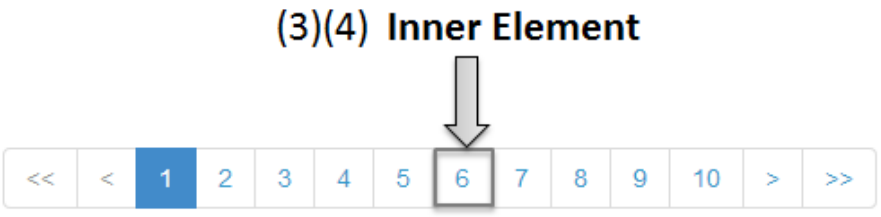
(5)

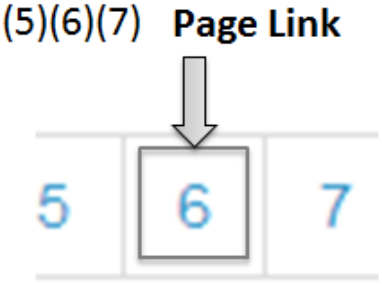

(1)
Outer Element
<ul class="{Outer Element Class}">
(2)
(3)
Inner Element
<li class="{Inner Element Class}">
(4)
(5)
Page Link

(6)
{Page Link Text}
(7)

<!-- ... -->


(1)(2) Outer Element
<< < 1 2 3 4 5 6 7 8 9 10 > >>


(3)(4) Inner Element
<< < 1 2 3 4 5 6 7 8 9 10 > >>


(5)(6)(7) Page Link
5
6
7

項番	説明	デフォルト値
(1)	ページネーションリンクの構成要素をまとめるための要素。 共通ライブラリでは、この部分を「Outer Element」と呼び、複数の「Inner Element」を保持する。 使用する要素は、JSP タグライブラリのパラメータによって変更することが出来る。	`<ul>` 要素
(2)	「Outer Element」のスタイルクラスを指定するための属性。 共通ライブラリでは、この部分を「Outer Element Class」と呼び、属性値は JSP タグライブラリのパラメータによって指定する。	指定なし
(3)	ページネーションリンクを構成するための要素。 共通ライブラリでは、この部分を「Inner Element」と呼び、 ページ移動するためのリクエストを送信するための `<a>` 要素を保持する。 使用する要素は、JSP タグライブラリのパラメータによって変更することが出来る。	`<li>` 要素
(4)	「Inner Element」のスタイルクラスを指定するための属性。 共通ライブラリでは、この部分を「Inner Element Class」と呼び、属性値は表示しているページ位置によって JSP タグライブラリ内の処理で切り替わる。	「*Note* 欄 」を参照されたい。
(5)	ページ移動するためのリクエストを送信するための要素。 共通ライブラリでは、この部分を「Page Link」と呼ぶ。	*<a>* 要素固定
(6)	ページ移動するための URL を指定するための属性。 共通ライブラリでは、この部分を「Page Link URL」と呼ぶ。	下記の「*Note* 欄 」を参照されたい。
(7)	ページ移動するためのリンクの表示テキストを指定する。 共通ライブラリでは、この部分を「Page Link Text」と呼ぶ。	下記の「*Note* 欄 」を参照されたい。

注釈: 「Inner Element」の数について

デフォルトの設定では、「Inner Element」は最大で 14 個となる。内訳は以下の通り。

- 最初のページに移動するためのリンク：1
- 前のページに移動するためのリンク：1
- 指定したページに移動するためのリンク：最大 10
- 次のページに移動するためのリンク：1
- 最後のページに移動するためのリンク：1

「Inner Element」の数は、JSP タグライブラリのパラメータの指定によって変更することができる。

注釈: 「Inner Element Class」の設定値について

デフォルトの設定では、ページ位置によって、以下 3 つの値となる。

- `disabled`: 現在表示しているページでは操作することができないリンクであることを示すためのスタイルクラス。
- `active`: 現在表示しているページのリンクであることを示すためのスタイルクラス。
- 指定なし：上記以外のリンクであることを示す。

`disabled` と `active` は、JSP タグライブラリのパラメータの指定によって別の値に変更することができる。

注釈: 「Page Link URL」のデフォルト値について

リンクの状態が `disabled` と `active` の場合は `javascript:void(0)`、それ以外の場合は`?page={page}&size={size}`となる。

「Page Link URL」は、JSP タグライブラリのパラメータの指定によって別の値に変更することができる。

terasoluna-gfw-web 5.0.0.RELEASE より、`active` 状態のリンクのデフォルト値を`?page={page}&size={size}`から `javascript:void(0)` に変更している。これは、メジャーな Web サイトのページネーションリンクの実装やメジャーな CSS ライブラリ (Bootstrap など) の実装に合わせるためである。

注釈: 「Page Link Text」のデフォルト値について

項番	リンク名	デフォルト値
1.	最初のページに移動するためのリンク	<<
2.	前のページに移動するためのリンク	「<」
3.	指定したページに移動するためのリンク	該当ページのページ番号 (変更不可)
4.	次のページに移動するためのリンク	「>」
5.	最後のページに移動するためのリンク	>>

「指定したページに移動するためのリンク」以外は、JSP タグライブラリのパラメータの指定によって、別の値に変更することができる。

JSP タブライブラリのパラメータについて

JSP タグライブラリのパラメータに値を指定することで、デフォルト動作を変更することができる。

以下にパラメータの一覧を示す。

レイアウトを制御するためのパラメータ

項番	パラメータ名	説明
1.	outerElement	「Outer Element」として使用する HTML 要素名を指定する。 例) div
2.	outerElementClass	「Outer Element Class」に設定するスタイルシートのクラス名を指定する。 例) pagination

Continued on next page

表 4.23 – continued from previous page

項番	パラメータ名	説明
3.	innerElement	「Inner Element」として使用する HTML 要素名を指定する。 例) span
4.	disabledClass	`disabled` 状態と判断された「Inner Element」の class 属性に設定する値を指定する。 例) hiddenPageLink
5.	activeClass	`active` 状態の「Inner Element」の class 属性に設定する値を指定する。 例) currentPageLink
6.	firstLinkText	「最初のページに移動するためのリンク」の「Page Link Text」に設定する値を指定する。 "" を指定すると、「最初のページに移動するためのリンク」自体が出力されなくなる。 例) First
7.	previousLinkText	「前のページに移動するためのリンク」の「Page Link Text」に設定する値を指定する。 "" を指定すると、「前のページに移動するためのリンク」自体が出力されなくなる。 例) Prev
8.	nextLinkText	「次のページに移動するためのリンク」の「Page Link Text」に設定する値を指定する。 "" を指定すると、「次のページに移動するためのリンク」自体が出力されなくなる。 例) Next

Continued on next page

表 4.23 – continued from previous page

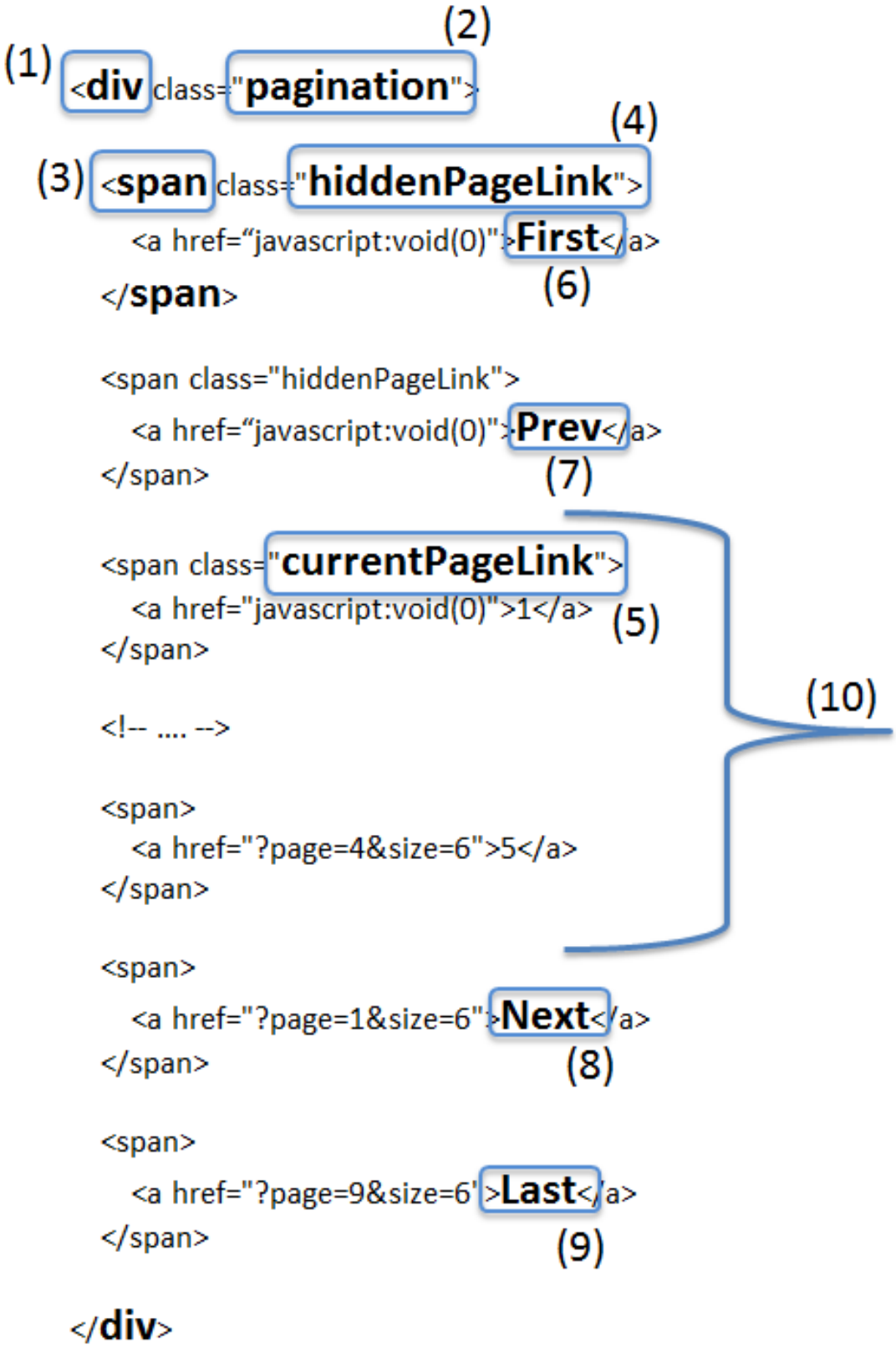
項番	パラメータ名	説明
9.	lastLinkText	「最後のページに移動するためのリンク」の「Page Link Text」に設定する値を指定する。 `""` を指定すると、「次のページに移動するためのリンク」自体が出力されなくなる。 例) Last
10.	maxDisplayCount	「指定したページに移動するためのリンク」の最大表示数を指定する。 「`0`」 を指定すると、「指定したページに移動するためのリンク」自体が出力されなくなる。 例) 5

レイアウトを制御するためのパラメータを、全てデフォルトから変更した時に出力される HTML は以下の通り。図中の番号は、上記で説明したパラメータ一覧の項番に対応している。

- JSP

```
<t:pagination page="${page}"
    outerElement="div"
    outerElementClass="pagination"
    innerElement="span"
    disabledClass="hiddenPageLink"
    activeClass="currentPageLink"
    firstLinkText="First"
    previousLinkText="Prev"
    nextLinkText="Next"
    lastLinkText="Last"
    maxDisplayCount="5"
    />
```

- 出力される HTML


(1)
(2)

(3)
(4)

First

(6)

Prev

(7)

1
(5)

<!-- .... -->

5

(10)

Next

(8)

Last

(9)

動作を制御するためのパラメータ

項番	パラメータ名	説明
1.	disabledHref	`disabled` 状態のリンクの「Page Link URL」に設定する値を指定する。
2.	pathTmpl	「Page Link URL」に設定するリクエストパスのテンプレートを指定する。 ページ表示時のリクエストパスとページ移動するためのリクエストパスが異なる場合は、このパラメータにページ移動用のリクエストパスを指定する必要がある。 指定するリクエストパスのテンプレートには、ページ位置 (page) や取得件数 (size) などをパス変数 (プレースホルダ) として指定することができる。 指定した値は UTF-8 で URL エンコーディングされる。
3.	queryTmpl	「Page Link URL」のクエリ文字列のテンプレートを指定する。 ページ移動する際に必要となるページネーション用のクエリ文字列 (page,size,sort パラメータ) を生成するためのテンプレートを指定する。 ページ位置や取得件数のリクエストパラメータ名をデフォルト以外の値にする場合は、このパラメータにクエリ文字列を指定する必要がある。 指定するクエリ文字列のテンプレートには、ページ位置 (page) や取得件数 (size) などをパス変数 (プレースホルダ) として指定することができる。 指定した値は UTF-8 で URL エンコーディングされる。 この属性は、ページネーション用のクエリ文字列 (page,size,sort パラメータ) を生成するための属性であるため、検索条件を引き継ぐためのクエリ文字列は criteriaQuery 属性に指定すること。

Continued on next page

表 4.24 – continued from previous page

項番	パラメータ名	説明
4.	criteriaQuery	「Page Link URL」に追加する検索条件用のクエリ文字列を指定する。 **「Page Link URL」に検索条件を引き継ぐ場合は、このパラメータに検索条件用のクエリ文字列を指定すること。** **指定した値は URL エンコーディングされないため、URL エンコーディング済みのクエリ文字列を指定する必要がある。** フォームオブジェクトに格納されている検索条件を URL エンコーディング済みのクエリ文字列に変換する場合は、共通ライブラリから提供している EL ファンクション (`f:query(Object)`) を使用すると、簡単に条件を引き継ぐことができる。 terasoluna-gfw-web 1.0.1.RELEASE 以上で利用可能なパラメータである。
5.	disableHtmlEscapeOfCriteriaQuery	`criteriaQuery` パラメータに指定された値に対する HTML エスケープ処理を無効化するためのフラグ。 `true` を指定する事で、`criteriaQuery` パラメータに指定された値に対して HTML エスケープ処理が行われなくなる。(デフォルト値は `false`) **true を指定する場合は、XSS 対策が必要な文字がクエリ文字列内に含まれない事が保証されていること。** terasoluna-gfw-web 1.0.1.RELEASE 以上で利用可能なパラメータである。

Continued on next page

表 4.24 – continued from previous page

項番	パラメータ名	説明
6.	enableLinkOfCurrentPage	`active` 状態のページリンクを押下した際に、該当ページを再表示するためのリクエストを送信するためのフラグ。 `true` を指定する事で、「Page Link URL」に該当ページを再表示するための URL(デフォルト値は`?page={page}&size={size}`)が設定される。(デフォルト値は `false` で、「Page Link URL」には `disabledHref` 属性の値が設定される) terasoluna-gfw-web 5.0.0.RELEASE 以上で利用可能なパラメータである。

注釈: **disabledHref** の設定値について

デフォルトでは、`disabledHref` 属性には `javascript:void(0)` が設定されている。ページリンク押下時の動作を無効化するだけであれば、デフォルトのままでよい。

ただし、デフォルトの状態でページリンクにフォーカスを移動又はマウスオーバーした場合、ブラウザのステータスバーに `javascript:void(0)` が表示されることがある。この挙動を変えたい場合は、JavaScript を使用してページリンク押下時の動作を無効化する必要がある。実装例については、「*JavaScript を使用したページリンクの無効化*」を参照されたい。

terasoluna-gfw-web 5.0.0.RELEASE より、`disabledHref` 属性のデフォルト値を」#」から `javascript:void(0)` に変更している。この変更を行うことで、`disabled` 状態のページリンクを押下した際に、フォーカスがページのトップへ移動しないようになっている。

注釈: パス変数 (プレースホルダ) について

`pathTmpl` 及び `queryTmpl` に指定できるパス変数は、以下の通り。

項番	パス変数名	説明
1.	page	ページ位置を埋め込むためのパス変数。
2.	size	取得件数を埋め込むためのパス変数。
3.	sortOrderProperty	ソート条件のソート項目を埋め込むためのパス変数。
4.	sortOrderDirection	ソート条件のソート順を埋め込むためのパス変数。

パス変数は、{**パス変数名**} の形式で指定する。

警告: ソート条件の制約事項

ソート条件のパス変数に設定される値は、ひとつのソート条件のみとなっている。そのため、複数のソート条件を指定して検索した結果を、ページネーション表示する必要がある場合は、共通ライブラリから提供している JSP タグライブラリを拡張する必要がある。

動作を制御するためのパラメータを変更した時に出力される HTML は、以下の通り。図中の番号は、上記で説明したパラメータ一覧の項番に対応している。

- JSP

```
<t:pagination page="${page}"
    disabledHref="#"
    pathTmpl="${pageContext.request.contextPath}/article/list/{page}/{size}
↪"
    queryTmpl="sort={sortOrderProperty},{sortOrderDirection}"
    criteriaQuery="${f:query(articleSearchCriteriaForm)}"
    enableLinkOfCurrentPage="true" />
```

- 出力される HTML




```
<ul>
  <li class="disabled">
    <a href="#">&lt;&lt;</a>
  </li>
  <li class="disabled">
    <a href="#">&lt;</a>
  </li>
  <li class="active">
    <a href="/webapp/article/list/0/6?sort=publishedDate,DESC&word=title">1</a>
  </li>

  <!-- ... -->

  <li>
    <a href="/webapp/article/list/9/6?sort=publishedDate,DESC&word=title">10</a>
  </li>
  <li>
    <a href="/webapp/article/list/1/6?sort=publishedDate,DESC&word=title">&gt;</a>
  </li>
  <li>
    <a href="/webapp/article/list/9/6?sort=publishedDate,DESC&word=title">&gt;&gt;</a>
  </li>
</ul>
```

ページネーション情報の表示について

ページネーションに関する情報の表示を行う。

Spring Data 提供のページ検索機能を使用することで、以下の情報を画面に表示することができる。

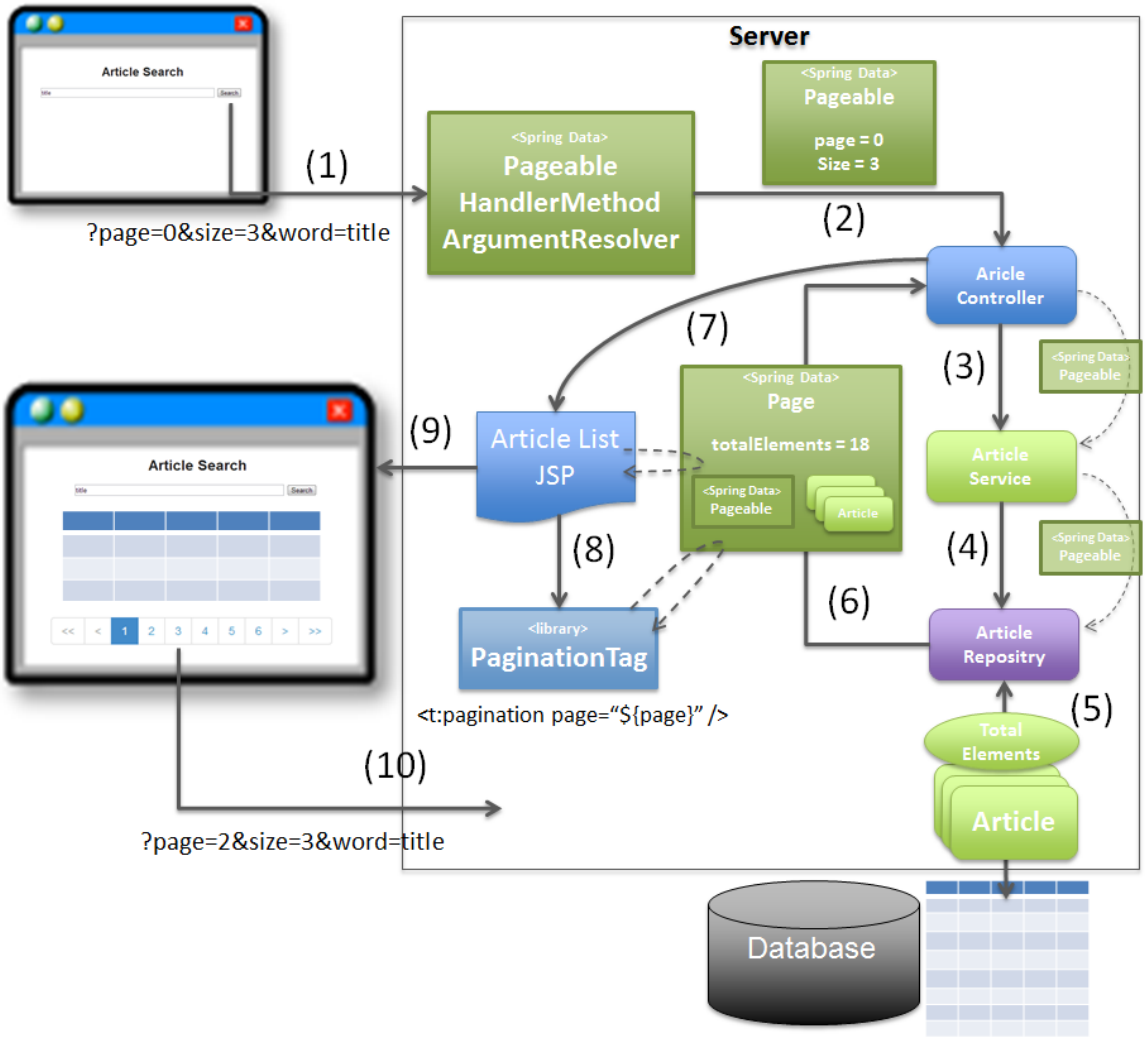
- 合計件数

- 検索対象のページ位置
- 取得件数
- ソート条件

ページ検索については、「ページ検索について 」を参照されたい。

ページネーション機能使用時の処理フロー

Spring Data より提供されているページネーション機能と、共通ライブラリから提供している JSP タグライブラリを利用した際の処理フローは、以下の通り。



項番	説明
(1)	検索条件と共に、リクエストパラメータとして検索対象のページ位置 (page) と取得件数 (size) を指定してリクエストを送信する。
(2)	`PageableHandlerMethodArgumentResolver` は、リクエストパラメータに指定されている検索対象のページ位置 (page) と取得件数 (size) を取得し、 `Pageable` オブジェクトを生成する。 生成された `Pageable` オブジェクトは、Controller のハンドラメソッドの引数に設定される。
(3)	Controller は、引数で受け取った `Pageable` オブジェクトを、Service のメソッドに引き渡す。
(4)	Service は、引数で受け取った `Pageable` オブジェクトを、 Repository の Query メソッドに引き渡す。
(5)	Repository は、検索条件に一致するデータの合計件数 (totalElements) と、引数で受け取った `Pageable` オブジェクトに指定されているページ位置 (page) と取得件数 (size) の範囲に存在するデータを、データベースより取得する。
(6)	Repository は、取得した合計件数 (totalElements)、取得データ (content)、引数で受け取った `Pageable` オブジェクトより `Page` オブジェクトを作成し、Service 及び Controller へ返却する。
(7)	Controller は、返却された `Page` オブジェクトを、 `Model` オブジェクトに格納後、JSP に遷移する。
(8)	JSP は、 `Model` オブジェクトに格納されている `Page` オブジェクトを取得し、共通ライブラリから提供されているページネーション用の JSP タグライブラリ (`<t:pagination>`) を呼び出す。 ページネーション用の JSP タグライブラリは `Page` オブジェクトを参照し、ページネーションリンクを生成する。

Continued on next page

表 4.25 – continued from previous page

項番	説明
(9)	JSP で生成した HTML を、クライアント (ブラウザ) に返却する。
(10)	ページネーションリンクを押下すると、該当ページを表示するためリクエストが送信される。

注釈: **Repository** の実装について

上記フローの (5) と (6) の処理は、使用する O/R Mapper によって実装方法が異なる。

- MyBatis3 を使用する場合は、Java(Service) 及び SQL マッピングファイルの実装が必要である。
- Spring Data JPA を使用する場合は、Spring Data JPA の機能で自動的で行われるため実装は不要である。

具体的な実装例については、

- データベースアクセス（*MyBatis3* 編）
- データベースアクセス（*JPA* 編）

を参照されたい。

4.4.2 How to use

ページネーション機能の具体的な使用方法を以下に示す。

アプリケーションの設定

Spring Data のページネーション機能を有効化するための設定

リクエストパラメータに指定された検索対象のページ位置 (page)、取得件数 (size)、ソート条件 (sort) を、`Pageable` オブジェクトとして Controller の引数に設定するための機能を有効化する。
下記の設定は、ブランクプロジェクトでは設定済みの状態になっている。

`spring-mvc.xml`

```
<mvc:annotation-driven>
    <mvc:argument-resolvers>
        <!-- (1) -->
        <bean
            class="org.springframework.data.web.
↪PageableHandlerMethodArgumentResolver" />
    </mvc:argument-resolvers>
</mvc:annotation-driven>
```

項番	説明
(1)	<mvc:argument-resolvers> に org.springframework.data.web.PageableHandlerMethodArgumentResolver を指定する。 PageableHandlerMethodArgumentResolver で指定できるプロパティについては、「*PageableHandlerMethodArgumentResolver のプロパティ値について* 」を参照されたい。

ページ検索の実装

ページ検索を実現するための実装方法を以下に示す。

アプリケーション層の実装

ページ検索に必要な情報 (検索対象のページ位置、取得件数、ソート条件) を、Controller の引数として受け取り、Service のメソッドに引き渡す。

- Controller

```
@RequestMapping("list")
public String list(@Validated ArticleSearchCriteriaForm form,
        BindingResult result,
        Pageable pageable, // (1)
        Model model) {

    ArticleSearchCriteria criteria = beanMapper.map(form,
            ArticleSearchCriteria.class);

    Page<Article> page = articleService.searchArticle(criteria, pageable); //␣
↪(2)

    model.addAttribute("page", page); // (3)

    return "article/list";
}
```

項番	説明
(1)	ハンドラメソッドの引数として Pageable を指定する。 Pageable オブジェクトには、ページ検索に必要な情報 (検索対象のページ位置、取得件数、ソート条件) が格納されている。
(2)	Service のメソッドの引数に Pageable オブジェクトを指定して呼び出す。
(3)	Service から返却された検索結果 (Page オブジェクト) を Model に追加する。 Model に追加することで、View(JSP) から参照できるようになる。

注釈: リクエストパラメータにページ検索に必要な情報の指定がない場合の動作について

ページ検索に必要な情報 (検索対象のページ位置、取得件数、ソート条件) がリクエストパラメータに指定されていない場合は、デフォルト値が適用される。デフォルト値は、以下の通り。

- 検索対象のページ位置 : *0* (1 ページ目)
- 取得件数 : *20*
- ソート条件 : *null* (ソート条件なし)

デフォルト値は、以下の 2 つの方法で変更することができる。

- ハンドラメソッドの Pageable の引数に、 @org.springframework.data.web.PageableDefault アノテーションを指定してデフォルト値を定義する。
- PageableHandlerMethodArgumentResolver の fallbackPageable プロパティにデフォルト値を定義した Pageable オブジェクトを指定する。

@PageableDefault アノテーションを使用してデフォルト値を指定する方法について説明する。

ページ検索処理毎にデフォルト値を変更する必要がある場合は、@PageableDefault アノテーションを使ってデフォルト値を指定する。

```
@RequestMapping("list")
public String list(@Validated ArticleSearchCriteriaForm form,
        BindingResult result,
        @PageableDefault( // (1)
                page = 0,     // (2)
                size = 50,    // (3)
                direction = Direction.DESC,  // (4)
                sort = {      // (5)
                    "publishedDate",
                    "articleId"
                    }
                ) Pageable pageable,
        Model model) {
    // ...
    return "article/list";
}
```

項番	説明	デフォルト値
(1)	Pageable の引数に @PageableDefault アノテーションを指定する。	-
(2)	ページ位置のデフォルト値を変更する場合は、 @PageableDefault の page 属性に値を指定する。 通常変更する必要はない。	「0」 (1 ページ目)
(3)	取得件数のデフォルト値を変更する場合は、 @PageableDefault の size 又は value 属性に値を指定する。	10
(4)	ソート条件のデフォルト値を変更する場合は、 @PageableDefault の direction 属性に値を指定する。	Direction.ASC (昇順)
(5)	ソート条件のソート項目を指定する場合は、 @PageableDefault の sort 属性にソート項目を指定する。 複数の項目でソートする場合は、ソートするプロパティ名を配列で指定する。	空の配列 (ソート項目なし)

指定したソート条件のクエリへの反映法は O/RMapper により異なる。

上記例では、JPA の場合、`ORDER BY publishedDate DESC, articleId DESC` というソート条件が Query に追加される。Mybatis の場合、`ORDER BY publishedDate DESC, articleId DESC` のような Order By 句を Query に追加することになる。

詳細については

- ドメイン層の実装 *(MyBatis3 編)*
- ドメイン層の実装 *(JPA 編)*

を参照されたい。

注釈: @PageableDefault アノテーションで指定できるソート順について

`@PageableDefault` アノテーションで指定できるソート順は昇順か降順のどちらか一つなので、項目ごとに異なるソート順を指定したい場合は `@org.springframework.data.web.SortDefaults` アノテーションを使用する必要がある。具体的には、`ORDER BY publishedDate DESC, articleId ASC` というソート順にしたい場合である。

ちなみに: 取得件数のデフォルト値のみ変更する場合の指定方法

取得件数のデフォルト値のみ変更する場合は、`@PageableDefault(50)` と指定することもできる。これは `@PageableDefault(size = 50)` と同じ動作となる。

`@SortDefaults` アノテーションを使用してデフォルト値を指定する方法について説明する。

`@SortDefaults` アノテーションは、ソート項目が複数あり、項目ごとに異なるソート順を指定したい場合に使用する。

```
@RequestMapping("list")
public String list(
        @Validated ArticleSearchCriteriaForm form,
        BindingResult result,
        @PageableDefault(size = 50)
        @SortDefaults(  // (1)
                {
                    @SortDefault(  // (2)
                                 sort = "publishedDate",    // (3)
                                 direction = Direction.DESC // (4)
                                ),
```

```
                    @SortDefault(
                                 sort = "articleId"
                                )
                }) Pageable pageable,
        Model model) {
    // ...
    return "article/list";
}
```

項番	説明	デフォルト値
(1)	Pageable の引数に @SortDefaults アノテーションを指定する。 @SortDefaults アノテーションには、複数の @org.springframework.data.web.SortDefault アノテーションを配列として指定することができる。	-
(2)	@SortDefaults アノテーションの value 属性に、 @SortDefault アノテーションを指定する。 複数指定する場合は配列として指定する。	-
(3)	@PageableDefault の sort 又は value 属性にソート項目を指定する。 複数の項目を指定する場合は配列として指定する。	空の配列 (ソート項目なし)
(4)	ソート条件のデフォルト値を変更する場合は、@PageableDefault の direction 属性に値を指定する。	Direction.ASC (昇順)

指定したソート条件のクエリへの反映法は O/RMapper により異なるため、詳細については、

- ドメイン層の実装 *(MyBatis3 編)*
- ドメイン層の実装 *(JPA 編)*

を参照されたい。

ちなみに：ソート項目のデフォルト値のみ指定する場合の指定方法

取得項目のみ指定する場合は、 @PageableDefault("articleId") と指定することもできる。これは @PageableDefault(sort = "articleId") や @PageableDefault(sort =

`"articleId", direction = Direction.ASC)` と同じ動作となる。

アプリケーション全体のデフォルト値を変更する必要がある場合は、 `spring-mvc.xml` に定義した `PageableHandlerMethodArgumentResolver` の `fallbackPageable` プロパティにデフォルト値を定義した `Pageable` オブジェクトを指定する。

`fallbackPageable` の説明や具体的な設定例については、「*PageableHandlerMethodArgumentResolver のプロパティ値について* 」を参照されたい。

ドメイン層の実装 (MyBatis3 編)

MyBatis を使用してデータベースにアクセスする場合は、Controller から受け取った `Pageable` オブジェクトより、必要な情報を抜き出して Repository に引き渡す。
該当データを抽出するための SQL やソート条件については、SQL マッピングで実装する必要がある。

ドメイン層で実装するページ検索処理の詳細については、

- *Entity のページネーション検索 (MyBatis3 標準方式)*
- *Entity のページネーション検索 (SQL 絞り込み方式)*
- *Entity のページネーション検索 (検索結果のソート)*

を参照されたい。

ドメイン層の実装 (JPA 編)

JPA(Spring Data JPA) を使用してデータベースにアクセスする場合は、Controller から受け取った `Pageable` オブジェクトを Repository に引き渡す。

ドメイン層で実装するページ検索処理の詳細については、

- *条件に一致する Entity のページ検索*

を参照されたい。

JSP の実装 (基本編)

ページ検索処理で取得した Page オブジェクトを一覧画面に表示し、ページネーションリンク及びページネーション情報 (合計件数、合計ページ数、表示ページ数など) を表示する方法を以下に示す。

取得データの表示

ページ検索処理で取得したデータを表示するための実装例を以下に示す。

- Controller

```
@RequestMapping("list")
public String list(@Validated ArticleSearchCriteriaForm form, BindingResult
↪result,
        Pageable pageable, Model model) {

    if (!StringUtils.hasLength(form.getWord())) {
        return "article/list";
    }

    ArticleSearchCriteria criteria = beanMapper.map(form,
            ArticleSearchCriteria.class);

    Page<Article> page = articleService.searchArticle(criteria, pageable);

    model.addAttribute("page", page); // (1)

    return "article/list";
}
```

項番	説明
(1)	page という属性名で Page オブジェクトを Model に格納する。 JSP では page という属性名を指定して Page オブジェクトにアクセスすることになる。

- JSP

```
<%-- ... --%>

<%-- (1) --%>
<c:when test="${page != null && page.totalPages != 0}">

    <table class="maintable">
```

```
        <thead>
            <tr>
                <th class="no">No</th>
                <th class="articleClass">Class</th>
                <th class="title">Title</th>
                <th class="overview">Overview</th>
                <th class="date">Published Date</th>
            </tr>
        </thead>

        <%-- (2) --%>
        <c:forEach var="article" items="${page.content}" varStatus="rowStatus
↪">
            <tr>
                <td class="no">
                    ${(page.number * page.size) + rowStatus.count}
                </td>
                <td class="articleClass">
                    ${f:h(article.articleClass.name)}
                </td>
                <td class="title">
                    ${f:h(article.title)}
                </td>
                <td class="overview">
                    ${f:h(article.overview)}
                </td>
                <td class="date">
                    ${f:h(article.publishedDate)}
                </td>
            </tr>
        </c:forEach>

    </table>

    <div class="paginationPart">

    <%-- ... --%>

    </div>
</c:when>

<%-- ... --%>
```

項番	説明
(1)	上記例では、条件に一致するデータが存在するかチェックを行い、一致するデータがない場合はヘッダ行も含めて表示していない。 一致するデータがない場合でもヘッダ行は表示させる必要がある場合は、この分岐は不要となる。
(2)	JSTL の `<c:forEach>` タグを使用して、取得したデータの一覧を表示する。 取得したデータは、 `Page` オブジェクトの `content` プロパティにリスト形式で格納されている。

- 上記 JSP で出力される画面例

No	Class	Title	Overview	Published Date
1	Internal	Internal title1_20	overview20	2013-10-29
2	International	International title2_20	overview20	2013-10-29
3	Economy	Economy title3_20	overview20	2013-10-29
4	Internal	Internal title1_19	overview19	2013-10-28
5	International	International title2_19	overview19	2013-10-28
6	Economy	Economy title3_19	overview19	2013-10-28

.
.
.

ページネーションリンクの表示

ページ移動するためのリンク (ページネーションリンク) を表示するための実装例を以下に示す。

共通ライブラリより提供している JSP タグライブラリを使用して、ページネーションリンクを出力する。

- `include.jsp`

  共通ライブラリより提供している JSP タグライブラリの宣言を行う。ブランクプロジェクトでは設定済みの状態となっている。

```
<%@ taglib uri="http://terasoluna.org/tags" prefix="t"%>        <%-- (1) --%>
<%@ taglib uri="http://terasoluna.org/functions" prefix="f"%>  <%-- (2) --%>
```

項番	説明
(1)	ページネーションリンクを表示するための JSP タグが格納されている。
(2)	ページネーションリンクを使う際に利用する JSP の EL ファンクションが格納されている。

- JSP

```
<t:pagination page="${page}" /> <%-- (3) --%>
```

項番	説明
(3)	<t:pagination> タグを使用する。 page 属性には、 Controller で Model に格納した Page オブジェクトを指定する。

- 出力される HTML

下記の出力例は、?page=0&size=6 を指定して検索した際の結果である。

```
<ul>
    <li class="disabled"><a href="javascript:void(0)">&lt;&lt;</a></li>
    <li class="disabled"><a href="javascript:void(0)">&lt;</a></li>
    <li class="active"><a href="javascript:void(0)">1</a></li>
    <li><a href="?page=1&size=6">2</a></li>
    <li><a href="?page=2&size=6">3</a></li>
    <li><a href="?page=3&size=6">4</a></li>
    <li><a href="?page=4&size=6">5</a></li>
    <li><a href="?page=5&size=6">6</a></li>
    <li><a href="?page=6&size=6">7</a></li>
    <li><a href="?page=7&size=6">8</a></li>
    <li><a href="?page=8&size=6">9</a></li>
    <li><a href="?page=9&size=6">10</a></li>
    <li><a href="?page=1&size=6">&gt;</a></li>
    <li><a href="?page=9&size=6">&gt;&gt;</a></li>
</ul>
```

ページネーションリンク用のスタイルシートを用意しないと以下のような表示となる。
見てわかる通り、ページネーションリンクとして成立していない。

- <<
- <
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- >
- >>

ページネーションリンクとして成立する最低限のスタイルシートの定義の追加と、JSP の変更を行うと、以下のような表示となる。

- 画面イメージ

<< < 1 2 3 4 5 6 7 8 9 10 > >>

- JSP

```
<%-- ... --%>

<t:pagination page="${page}"
```

```
    outerElementClass="pagination" /> <%-- (4) --%>

<%-- ... --%>
```

項番	説明
(4)	ページネーションリンクであることを示すクラス名を指定する。 クラス名を指定することによってスタイルシートで指定するスタイルの適用範囲をページネーションリンクに限定することができる。

- スタイルシート

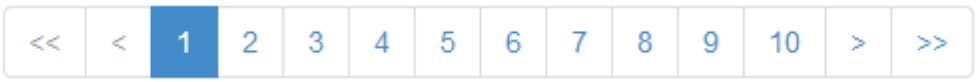
```
.pagination li {
    display: inline;
}

.pagination li>a {
    margin-left: 10px;
}
```

ページネーションリンクとして成立したが、以下 2 つの問題が残る。

- 押下できるリンクと押下できないリンクの区別ができない。
- 現在表示しているページ位置がわからない。

上記の問題を解決する手段として、Bootstrap v3.0.0 のスタイルシートと適用すると、以下のような表示となる。

- 画面イメージ



- スタイルシート

  bootstrap v3.0.0 の css ファイルを `$WEB_APP_ROOT/resources/vendor/bootstrap-3.0.0/css/bootstrap.css` に配置する。

  以下、ページネーション関連のスタイル定義の抜粋。

```
.pagination {
  display: inline-block;
  padding-left: 0;
  margin: 20px 0;
  border-radius: 4px;
}

.pagination > li {
  display: inline;
}

.pagination > li > a,
.pagination > li > span {
  position: relative;
  float: left;
  padding: 6px 12px;
  margin-left: -1px;
  line-height: 1.428571429;
  text-decoration: none;
  background-color: #ffffff;
  border: 1px solid #dddddd;
}

.pagination > li:first-child > a,
.pagination > li:first-child > span {
  margin-left: 0;
  border-bottom-left-radius: 4px;
  border-top-left-radius: 4px;
}

.pagination > li:last-child > a,
.pagination > li:last-child > span {
  border-top-right-radius: 4px;
  border-bottom-right-radius: 4px;
}

.pagination > li > a:hover,
.pagination > li > span:hover,
.pagination > li > a:focus,
.pagination > li > span:focus {
  background-color: #eeeeee;
}

.pagination > .active > a,
.pagination > .active > span,
.pagination > .active > a:hover,
```

```
.pagination > .active > span:hover,
.pagination > .active > a:focus,
.pagination > .active > span:focus {
  z-index: 2;
  color: #ffffff;
  cursor: default;
  background-color: #428bca;
  border-color: #428bca;
}

.pagination > .disabled > span,
.pagination > .disabled > a,
.pagination > .disabled > a:hover,
.pagination > .disabled > a:focus {
  color: #999999;
  cursor: not-allowed;
  background-color: #ffffff;
  border-color: #dddddd;
}
```

- JSP

  JSP では配置した css ファイルを読み込む定義を追加する。

```
<link rel="stylesheet"
    href="${pageContext.request.contextPath}/resources/vendor/bootstrap-3.0.0/
↪css/bootstrap.css"
    type="text/css" media="screen, projection">
```

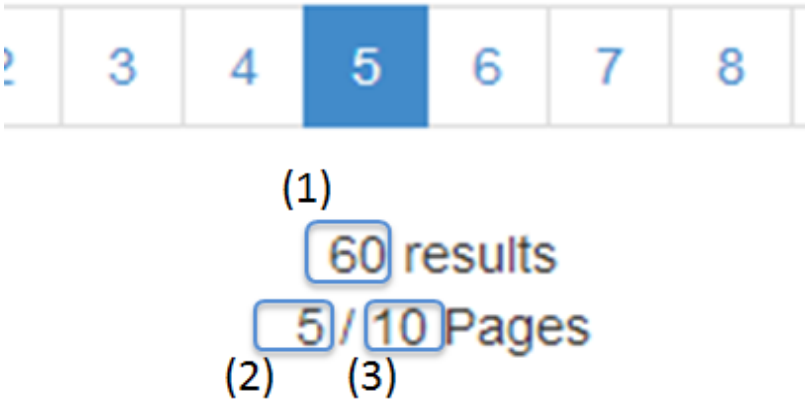
ページネーション情報の表示

ページネーションに関連する情報 (合計件数、合計ページ数、表示ページ数など) を表示するための実装例を以下に示す。

- 画面例

- JSP

```
<div>
    <fmt:formatNumber value="${page.totalElements}" /> results <%-- (1) --%>
</div>
<div>
    ${f:h(page.number + 1) } /       <%-- (2) --%>
    ${f:h(page.totalPages)} Pages    <%-- (3) --%>
</div>
```

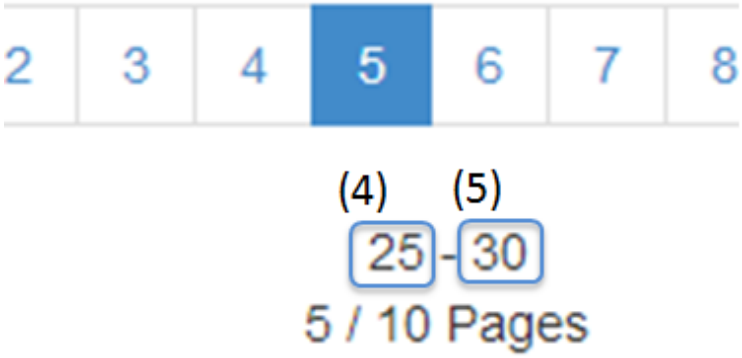


項番	説明
(1)	検索条件に一致するデータの合計件数を表示する場合は、 `Page` オブジェクトの `totalElements` プロパティから値を取得する。
(2)	表示しているページのページ数を表示する場合は、 `Page` オブジェクトの `number` プロパティから値を取得し、`+1` する。 `Page` オブジェクトの `number` プロパティは「`0`」開始のため、ページ番号を表示する際は `+1` が必要となる。
(3)	検索条件に一致するデータの合計ページ数を表示する場合は、 `Page` オブジェクトの `totalPages` プロパティから値を取得する。

該当ページの表示データ範囲を表示するための実装例を以下に示す。

- 画面例

- JSP




```
<div>
    <%-- (4) --%>
    <fmt:formatNumber value="${(page.number * page.size) + 1}" /> -
    <%-- (5) --%>
    <fmt:formatNumber value="${(page.number * page.size) + page.
↪numberOfElements}" />
</div>
```

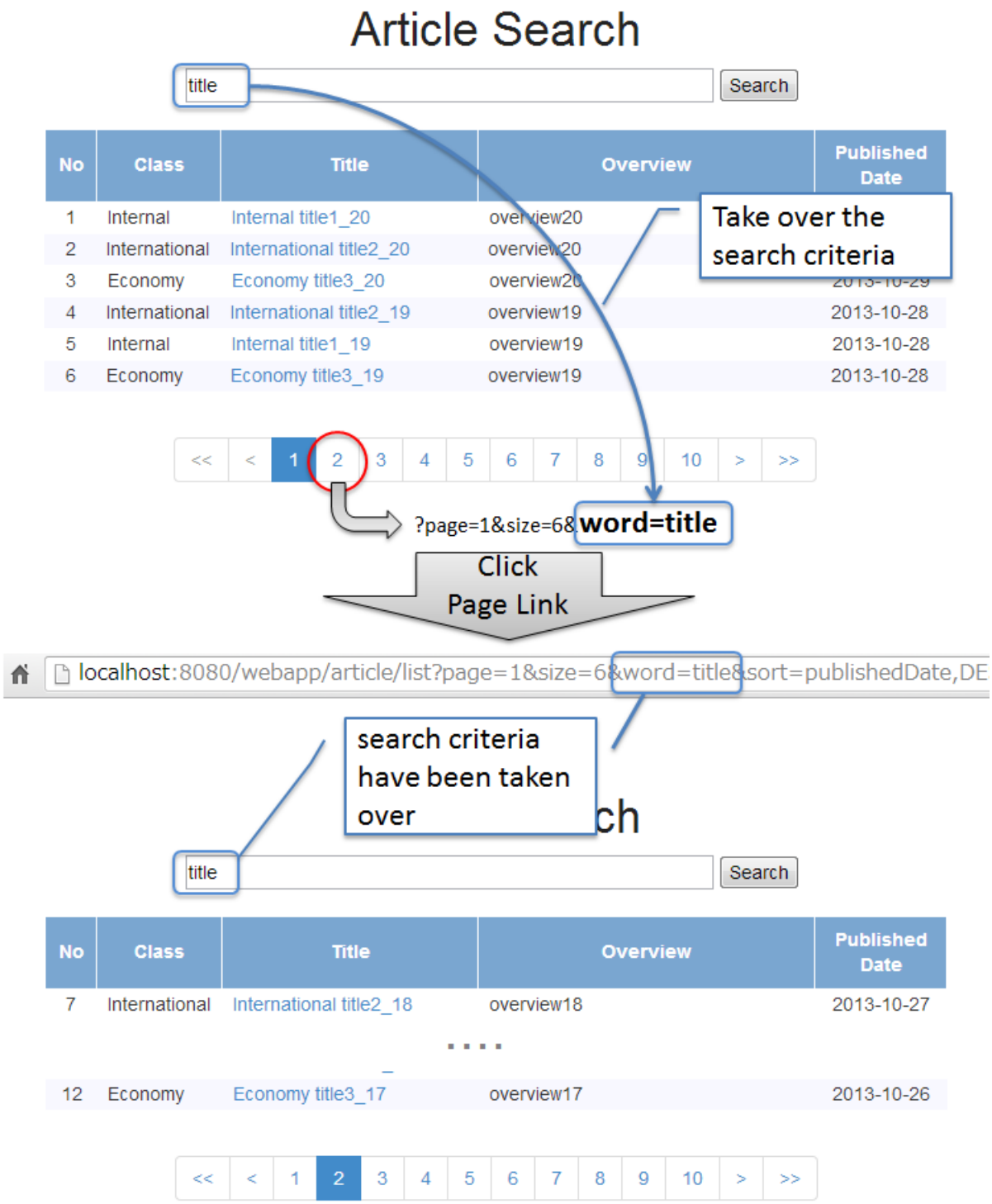
項番	説明
(4)	開始位置を表示する場合は、 Page オブジェクトの number プロパティと size プロパティを使って計算する。 Page オブジェクトの number プロパティは 「0」 開始のため、データ開始位置を表示する際は +1 が必要となる。
(5)	終了位置を表示する場合は、 Page オブジェクトの number プロパティ、 size プロパティ、 numberOfElements プロパティ を使って計算する。 最終ページは端数となる可能性があるので、 numberOfElements を加算する必要がある。

ちなみに: 数値のフォーマットについて

表示する数値をフォーマットする必要がある場合は、 JSTL から提供されているタグライブラリ (<fmt:formatNumber>) を使用する。

ページリンクで検索条件を引き継ぐ

検索条件をページ移動時のリクエストに引き継ぐ方法を、以下に示す。



- JSP

```
<%-- (1) --%>
<div id="criteriaPart">
    <form:form action="${pageContext.request.contextPath}/article/list"
↪method="get"
        modelAttribute="articleSearchCriteriaForm">
        <form:input path="word" />
        <form:button>Search</form:button>
        <br>
    </form:form>
</div>

<%-- ... --%>

<t:pagination page="${page}"
    outerElementClass="pagination"
    criteriaQuery="${f:query(articleSearchCriteriaForm)}" /> <%-- (2) --%>
```

項番	説明
(1)	検索条件を指定するフォーム。 検索条件として word が存在する。
(2)	ページ移動時のリクエストに検索条件を引き継ぐ場合は、 criteriaQuery 属性に **URL エンコーディング済みのクエリ文字列**を指定する。 検索条件をフォームオブジェクトに格納する場合は、共通ライブラリから提供している EL ファンクション (f:query(Object)) を使用すると、簡単に条件を引き継ぐことができる。 上記例の場合、 ?page=**ページ位置**&size=**取得件数**&word=**入力値**という形式のクエリ文字列が生成される。 criteriaQuery 属性は、terasoluna-gfw-web 1.0.1.RELEASE 以上で利用可能な属性である。

注釈: f:query(Object) の仕様について

f:query の引数には、 フォームオブジェクトなどの JavaBean と Map オブジェクトを指定することができる。JavaBean の場合はプロパティ名がリクエストパラメータ名となり、 Map オブジェクトの場合はマップのキー名がリクエストパラメータとなる。生成されるクエリ文字列は、UTF-8 の URL エンコーディングが行われる。

terasoluna-gfw-web 5.0.1.RELEASE より、ネスト構造をもつ JavaBean と Map オブジェクトを指定で

きるように改善されている。

f:query の詳細な仕様 (URL エンコーディングの仕様など) については、「*f:query()*」を参照されたい。

警告: f:query を使用して生成したクエリ文字列を queryTmpl 属性に指定した際の動作について

f:query を使用して生成したクエリ文字列を queryTmpl 属性に指定すると、URL エンコーディングが重複してしまい、特殊文字の引き継ぎが正しく行われないことが判明している。

URL エンコーディングが重複してしまう事象については、terasoluna-gfw-web 1.0.1.RELEASE 以上で利用可能な criteriaQuery 属性を使用することで回避する事が出来る。

ページリンクでソート条件を引き継ぐ

ソート条件をページ移動時のリクエストに引き継ぐ方法を、以下に示す。

- JSP

```
<t:pagination page="${page}"
    outerElementClass="pagination"
    queryTmpl="page={page}&size={size}&sort={sortOrderProperty},
↪{sortOrderDirection}" />  <%-- (1) --%>
```

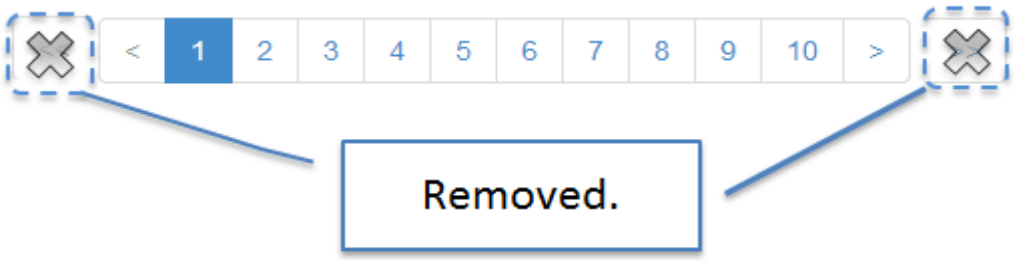
項番	説明
(1)	ページ移動時のリクエストにソート条件を引き継ぐ場合は、 queryTmpl を指定し、クエリ文字列にソート条件を追加する。 ソート条件を指定するためのパラメータの仕様については、「 ページ検索用のリクエストパラメータについて 」を参照されたい。 上記例の場合、 ?page=0&size=20&sort=ソート項目,ソート順 (ASC or DESC) がクエリ文字列となる。

JSP の実装 (レイアウト変更編)

先頭ページと最終ページに移動するリンクの削除

「最初のページに移動するためのリンク」と「最後のページに移動するためのリンク」を削除するための実装例を、以下に示す。

- 画面例



- JSP

```
<t:pagination page="${page}"
    outerElementClass="pagination"
    firstLinkText=""
    lastLinkText="" /> <%-- (1) (2) --%>
```

項番	説明
(1)	「最初のページに移動するためのリンク」を非表示にする場合は、 <t:pagination> タグの firstLinkText 属性に "" を指定する。
(2)	「最後のページに移動するためのリンク」を非表示にする場合は、 <t:pagination> タグの lastLinkText 属性に "" を指定する。

前ページと次ページに移動するリンクの削除

「前のページに移動するためのリンク」と「次のページに移動するためのリンク」を削除するための実装例を、以下に示す。

- 画面例



- JSP

```
<t:pagination page="${page}"
    outerElementClass="pagination"
    previousLinkText=""
    nextLinkText="" /> <%-- (1) (2) --%>
```

項番	説明
(1)	「前のページに移動するためのリンク」を非表示にする場合は、 <t:pagination> タグの previousLinkText 属性に "" を指定する。
(2)	「次のページに移動するためのリンク」を非表示にする場合は、 <t:pagination> タグの nextLinkText 属性に "" を指定する。

disabled 状態のリンクの削除

disabled 状態のリンクを削除するための実装例を、以下に示す。
disabled 時のスタイルシートに、以下の定義を追加する。

- 画面例

- スタイルシート



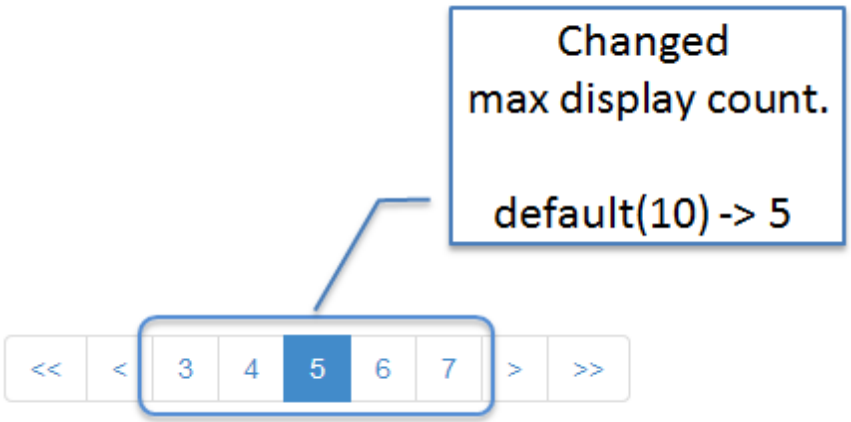

```
.pagination .disabled {
    display: none;  /* (1) */
}
```

項番	説明
(1)	disabled クラスの属性値として、 display: none; を指定する。

指定ページへ移動するリンクの最大表示数の変更

指定したページに移動するためのリンクの最大表示数を変更するための実装例を、以下に示す。

- 画面例



- JSP

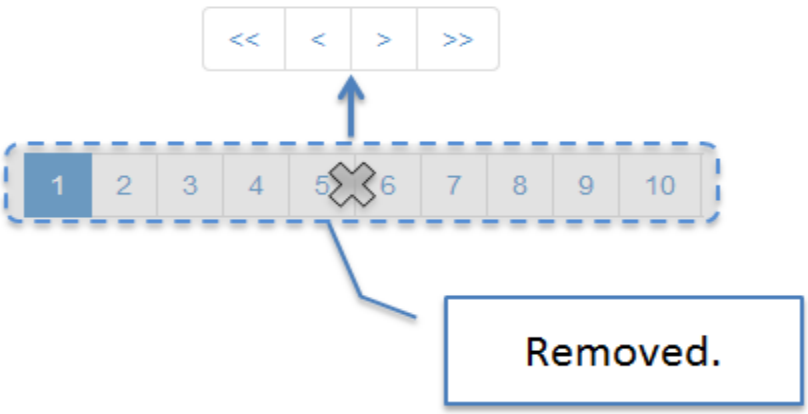
```
<t:pagination page="${page}"
    outerElementClass="pagination"
    maxDisplayCount="5" /> <%-- (1) --%>
```

項番	説明
(1)	指定したページに移動するためのリンクの最大表示数を変更する場合は、 <t:pagination> タグの maxDisplayCount 属性に値を指定する。

指定ページへ移動するリンクの削除

指定したページに移動するためのリンクを削除するための実装例を、以下に示す。

- 画面例



- JSP

```
<t:pagination page="${page}"
    outerElementClass="pagination"
    maxDisplayCount="0" /> <%-- (1) --%>
```

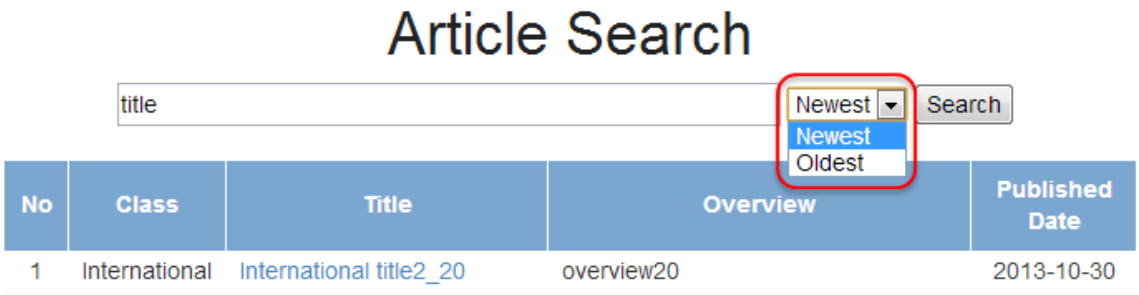
項番	説明
(1)	指定したページに移動するためのリンクを非表示にする場合は、 `<t:pagination>` タグの maxDisplayCount 属性に 「`0`」 を指定する。

JSP の実装 (動作編)

ソート条件の指定

クライアントからソート条件を指定するための実装例を、以下に示す。

- 画面例



- JSP

```
<div id="criteriaPart">
    <form:form
        action="${pageContext.request.contextPath}/article/search"
        method="get" modelAttribute="articleSearchCriteriaForm">
        <form:input path="word" />
        <%-- (1) --%>
        <form:select path="sort">
            <form:option value="publishedDate,DESC">Newest</form:option>
            <form:option value="publishedDate,ASC">Oldest</form:option>
        </form:select>
        <form:button>Search</form:button>
        <br>
    </form:form>
</div>
```

項番	説明
(1)	クライアントからソート条件を指定する場合は、 ソート条件を指定するためのパラメータを追加する。 ソート条件を指定するためのパラメータの仕様については、「 ページ検索用のリクエストパラメータについて 」を参照されたい。 上記例では、publishedDate の昇順と降順をプルダウンで選択できるようにしている。

JavaScript を使用したページリンクの無効化

デフォルトでは、disabled 状態と active 状態のページリンク押下時の動作を無効化するために、<t:pagination>タグの disabledHref 属性に javascript:void(0) を設定している。この状態でページリンクにフォーカスを移動又はマウスオーバーすると、 ブラウザのステータスバーに javascript:void(0) が表示されることがある。この挙動を変えたい場合は、JavaScript を使用してページリンク押下時の動作を無効化する必要がある。

以下に実装例を示す。

JSP

```
<%-- (1) --%>
<script type="text/javascript"
        src="${pageContext.request.contextPath}/resources/vendor/js/jquery.js"></
↪script>

<%-- (2) --%>
<script type="text/javascript">
    $(function(){
        $(document).on("click", ".disabled a, .active a", function(){
            return false;
        });
    });
</script>

<%-- ... --%>

<%-- (3) --%>
<t:pagination page="${page}" disabledHref="#" />
```

項番	説明
(1)	jQuery の js ファイルを読み込む。 上記例では、JavaScript を使用してページリンク押下時の動作を無効化するために jQuery の API を利用する。
(2)	jQuery の API を使用して、`disabled` 状態と `active` 状態のページリンクのクリックイベントを無効化する。 ただし、`<t:pagination>`タグの `enableLinkOfCurrentPage` 属性に `true` を指定している場合は、`active` 状態のページリンクのクリックイベントを無効化してはいけない。
(3)	`disabledHref` 属性に」#」 を指定する。

4.4.3 Appendix

`PageableHandlerMethodArgumentResolver` のプロパティ値について

`PageableHandlerMethodArgumentResolver` で指定できるプロパティは以下の通り。

アプリケーションの要件に応じて、値を変更すること。

項番	プロパティ名	説明	デフォルト値
1.	maxPageSize	取得件数として許可する最大値を指定する。 指定された取得件数が `maxPageSize` を超えていた場合は、 `maxPageSize` が取得件数となる。	*2000*
2.	fallbackPageable	アプリケーション全体のページ位置、取得件数、ソート条件のデフォルト値を指定する。 ページ位置、取得件数、ソート条件が指定されていない場合は、fallbackPageable に設定されている値が適用される。	ページ位置 : *0* 取得件数 : *20* ソート条件 : *null*

Continued on next page

表 4.26 – continued from previous page

項番	プロパティ名	説明	デフォルト値
3.	oneIndexedParameters	ページ位置の開始値を指定する。 *false* を指定した場合はページ位置の開始値は *0* となり、*true* を指定した場合はページ位置の開始値は *1* となる。	*false*
4.	pageParameterName	ページ位置を指定するためのリクエストパラメータ名を指定する。	`page`
5.	sizeParameterName	取得件数を指定するためのリクエストパラメータ名を指定する。	`size`
6.	prefix	ページ位置と取得件数を指定するためのリクエストパラメータの接頭辞 (ネームスペース) を指定する。 デフォルトのパラメータ名がアプリケーションで使用するパラメータと衝突する場合は、ネームスペースを指定することでリクエストパラメータ名の衝突を防ぐことが出来る。 prefix を指定すると、ページ位置を指定するためのリクエストパラメータ名は `prefix + pageParameterName` 、取得件数を指定するためのリクエストパラメータ名は `prefix + sizeParameterName` となる。	`""` (ネームスペースなし)
7.	qualifierDelimiter	同一リクエストで複数のページ検索が必要になる場合、ページ検索に必要な情報 (検索対象のページ位置、取得件数など) を区別するために、リクエストパラメータ名は `qualifier + delimiter +` **標準パラメータ名** の形式で指定する。 本プロパティは、上記形式の中の `delimiter` の値を設定する。 この設定を変更する場合は、`SortHandlerMethodArgumentResolver` の `qualifierDelimiter` 設定も合わせて変更する必要がある。	「_」

注釈: **maxPageSize** の設定値について

デフォルト値は 2000 であるが、アプリケーションが許容する最大値に設定を変更することを推奨する。アプリケーションが許可する最大値が *100* ならば、maxPageSize も *100* に設定する。

注釈: **fallbackPageable** の設定方法について

アプリケーション全体に適用するデフォルト値を変更する場合は、 `fallbackPageable` プロパティにデフォルト値が定義されている `Pageable` (`org.springframework.data.domain.PageRequest`) オブジェクトを設定する。ソート条件のデフォルト値を変更する場合は、`SortHandlerMethodArgumentResolver` の `fallbackSort` プロパティにデフォルト値が定義されている `org.springframework.data.domain.Sort` オブジェクトを設定する。

開発するアプリケーション毎に変更が想定される以下の項目について、デフォルト値を変更する際の設定例を以下に示す。

- 取得件数として許可する最大値 (`maxPageSize`)
- アプリケーション全体のページ位置、取得件数のデフォルト値 (`fallbackPageable`)
- ソート条件のデフォルト値 (`fallbackSort`)

```
<mvc:annotation-driven>
    <mvc:argument-resolvers>
        <bean
            class="org.springframework.data.web.
↪PageableHandlerMethodArgumentResolver">
            <!-- (1) -->
            <property name="maxPageSize" value="100" />
            <!-- (2) -->
            <property name="fallbackPageable">
                <bean class="org.springframework.data.domain.PageRequest">
                    <!-- (3) -->
                    <constructor-arg index="0" value="0" />
                    <!-- (4) -->
                    <constructor-arg index="1" value="50" />
                </bean>
            </property>
            <!-- (5) -->
            <constructor-arg index="0">
                <bean class="org.springframework.data.web.
↪SortHandlerMethodArgumentResolver">
```

```
                    <!-- (6) -->
                    <property name="fallbackSort">
                        <bean class="org.springframework.data.domain.Sort">
                            <!-- (7) -->
                            <constructor-arg index="0">
                                <list>
                                    <!-- (8) -->
                                    <bean class="org.springframework.data.
↪domain.Sort.Order">
                                        <!-- (9) -->
                                        <constructor-arg index="0" value="DESC
↪" />
                                        <!-- (10) -->
                                        <constructor-arg index="1" value=
↪"lastModifiedDate" />
                                    </bean>
                                    <!-- (8) -->
                                    <bean class="org.springframework.data.
↪domain.Sort.Order">
                                        <constructor-arg index="0" value="ASC
↪" />
                                        <constructor-arg index="1" value="id"␣
↪/>
                                    </bean>
                                </list>
                            </constructor-arg>
                        </bean>
                    </property>
                </bean>
            </constructor-arg>
        </bean>
    </mvc:argument-resolvers>
</mvc:annotation-driven>
```

項番	説明
(1)	上記例では取得件数の最大値を *100* に設定している。 取得件数 (size) に *101* 以上が指定された場合は、 *100* に切り捨てて検索が行われる。
(2)	`org.springframework.data.domain.PageRequest` のインスタンスを生成し、`fallbackPageable` に設定する。
(3)	`PageRequest` のコンストラクタの第 1 引数に、ページ位置のデフォルト値を指定する。 上記例では *0* を指定しているため、デフォルト値は変更していない。

Continued on next page

表 4.27 – continued from previous page

項番	説明
(4)	`PageRequest` のコンストラクタの第 2 引数に、取得件数のデフォルト値を指定する。 上記例ではリクエストパラメータに取得件数の指定がない場合の取得件数は *50* となる。
(5)	`PageableHandlerMethodArgumentResolver` のコンストラクタとして、`SortHandlerMethodArgumentResolver` のインスタンスを設定する。
(6)	`Sort` のインスタンスを生成し、 `fallbackSort` に設定する。
(7)	`Sort` のコンストラクタの第 1 引数に、 デフォルト値として使用する `Order` オブジェクトのリストを設定する。
(8)	`Order` のインスタンスを生成し、 デフォルト値として使用する `Order` オブジェクトのリストに追加する。 ソート条件のクエリへの反映法は O/RMapper により異なるため、詳細については ドメイン層の実装 *(MyBatis3 編)* ドメイン層の実装 *(JPA 編)* を参照されたい。
(9)	`Order` のコンストラクタの第 1 引数に、ソート順 (ASC/DESC) を指定する。
(10)	`Order` のコンストラクタの第 2 引数に、ソート項目を指定する。

`SortHandlerMethodArgumentResolver` のプロパティ値について

`SortHandlerMethodArgumentResolver` で指定できるプロパティは以下の通り。

アプリケーションの要件に応じて、値を変更すること。

項番	プロパティ名	説明	デフォルト値
1.	fallbackSort	アプリケーション全体のソート条件のデフォルト値を指定する。 ソート条件が指定されていない場合は、fallbackSort に設定されている値が適用される。	*null* (ソート条件なし)
2.	sortParameter	ソート条件を指定するためのリクエストパラメータ名を指定する。 デフォルトのパラメータ名がアプリケーションで使用するパラメータと衝突する場合は、リクエストパラメータ名を変更することで衝突を防ぐことができる。	`sort`
3.	propertyDelimiter	ソート項目及びソート順 (ASC,DESC) の区切り文字を指定する。	「`,`」
4.	qualifierDelimiter	同一リクエストで複数のページ検索が必要になる場合、ページ検索に必要な情報 (ソート条件) を区別するために、リクエストパラメータ名は `qualifier + delimiter + sortParameter` の形式で指定する。 本プロパティは、上記形式の中の `delimiter` の値を設定する。	「`_`」

4.5 二重送信防止

4.5.1 Overview

Problems

画面を提供する Web アプリケーションでは、以下の操作が行われると、同じ処理が複数回実行されてしまうことがある。

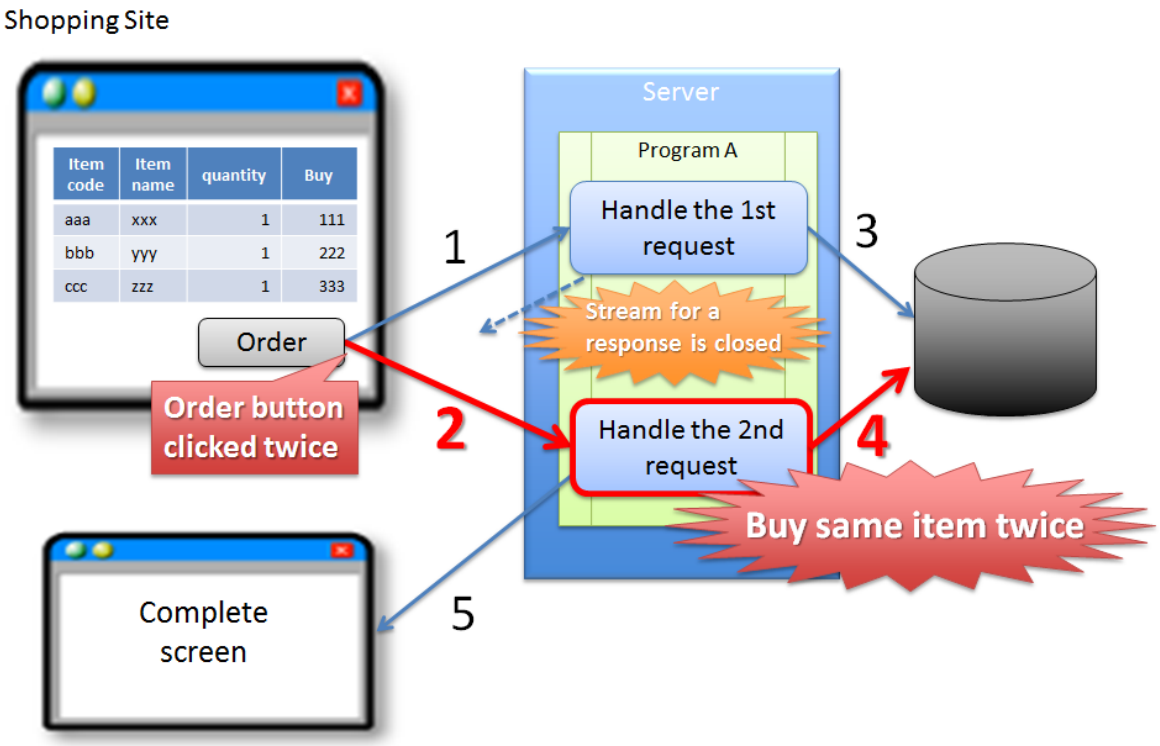
項番	操作	操作概要
(1)	更新系ボタンの二重クリック	更新処理を行うボタンを連続してクリックする。
(2)	更新処理完了後の画面の再読み込み	ブラウザの更新ボタンを使用することで、更新処理完了後の画面の再読み込みを行う。
(3)	ブラウザの戻るボタンを使用した不正な画面遷移	更新処理の完了画面からブラウザの戻るボタンを使用してページを戻し、更新処理を行うボタンを再度クリックする。

それぞれ具体的な問題点を、以下に示す。

更新系ボタンの二重クリック

更新処理を行うボタンを連続してクリックすると、以下のような問題が発生する。

以下では、ショッピングサイトの商品購入を例として、対策を行わない場合にどのような問題が発生するのかを説明する。



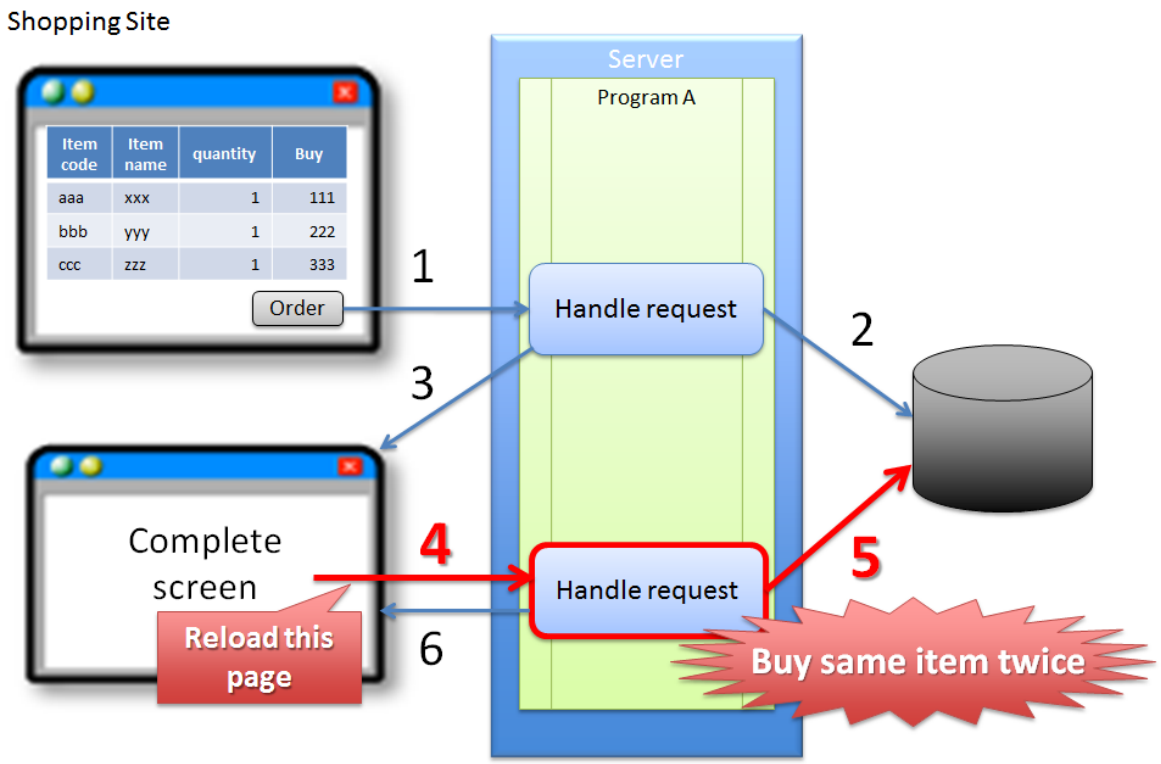
項番	説明
(1)	購買者が、商品購入画面で注文ボタンをクリックする。
(2)	(1) のレスポンスが返る前に、購買者が誤って注文ボタンをもう一度クリックする。
(3)	サーバは、(1) のリクエストで受けた商品の購入処理を DB に対して反映する。
(4)	サーバは、(2) のリクエストで受けた商品の購入処理を DB に対して反映する。
(5)	サーバは、(2) のリクエストで受けた商品の購入完了画面を応答する。

> **警告:** 上記のケースでは、購入者が誤って注文ボタンを押下することで、まったく同じ商品の購入が２回行われてしまうことになる。購入者の操作ミスが原因ではあるが、アプリケーションとして上記の問題が発生しないように制御する事が望ましい。

更新処理完了後の画面の再読み込み

更新処理完了後の画面の再読み込みを行うと、以下のような問題が発生する。

以下では、ショッピングサイトの商品購入を例として、対策を行わない場合にどのような問題が発生するのかを説明する。



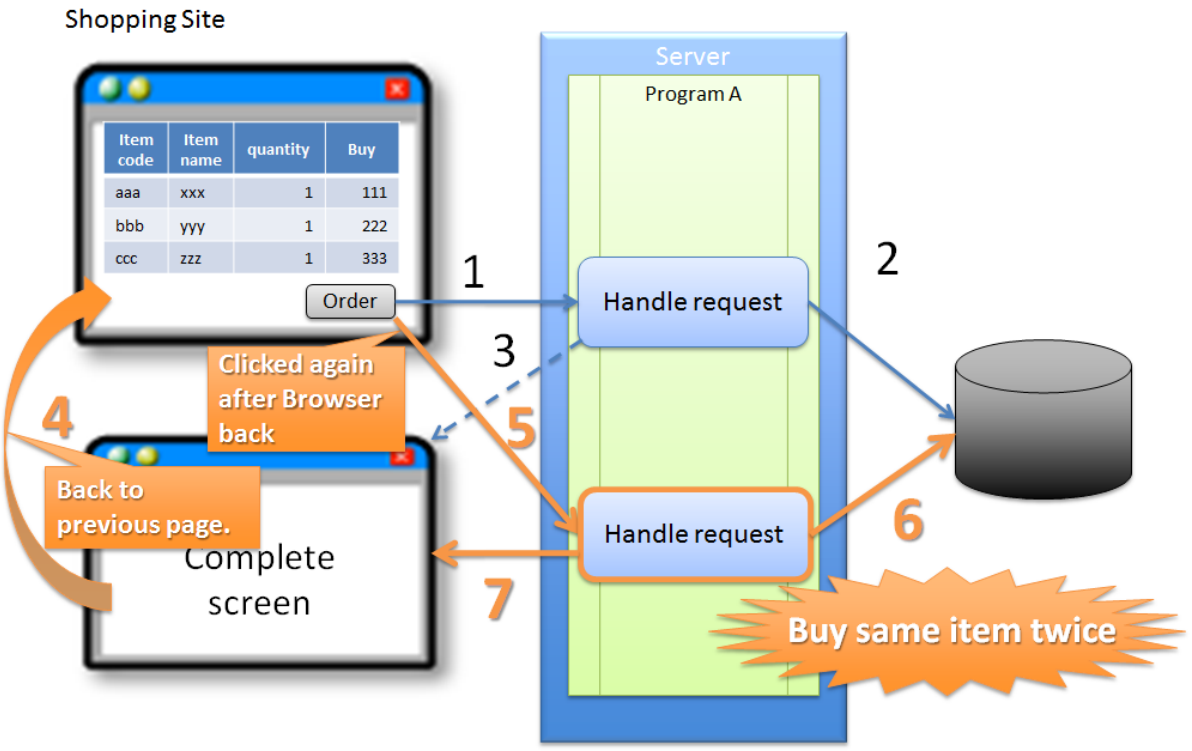
項番	説明
(1)	購買者が、商品購入画面で注文ボタンをクリックする。
(2)	サーバは、(1) のリクエストで受けた商品の購入処理を DB に対して反映する。
(3)	サーバは、(1) のリクエストで受けた商品の購入完了画面を応答する。
(4)	購買者が、誤ってブラウザのリロード機能を実行する。
(5)	サーバは、(4) のリクエストで受けた商品の購入処理を DB に対して反映する。
(6)	サーバは、(4) のリクエストで受けた商品の購入完了画面を応答する。

警告: 上記のケースでは、購入者が誤ってブラウザのリロード機能を実行することで、**まったく同じ商品の購入が２回行われてしまうことになる。**購入者の操作ミスが原因ではあるが、アプリケーションとして上記の問題が発生しないように制御する事が望ましい。

ブラウザの戻るボタンを使用した不正な画面遷移

ブラウザの戻るボタンを使用した不正な画面遷移を行うと、以下のような問題が発生する。

以下では、ショッピングサイトの商品購入を例として、対策を行わない場合にどのような問題が発生するのかを説明する。



項番	説明
(1)	購買者が、商品購入画面で注文ボタンをクリックする。
(2)	サーバは、(1) のリクエストで受けた商品の購入処理を DB に対して反映する。
(3)	サーバは、(1) のリクエストで受けた商品の購入完了画面を応答する。
(4)	購買者が、ブラウザの戻るボタンを使って購入画面を再度表示する。
(5)	購買者が、ブラウザの戻るボタンを使って表示した購入画面で注文ボタンを再度クリックする。
(6)	サーバは、(5) のリクエストで受けた商品の購入処理を DB に対して反映する。

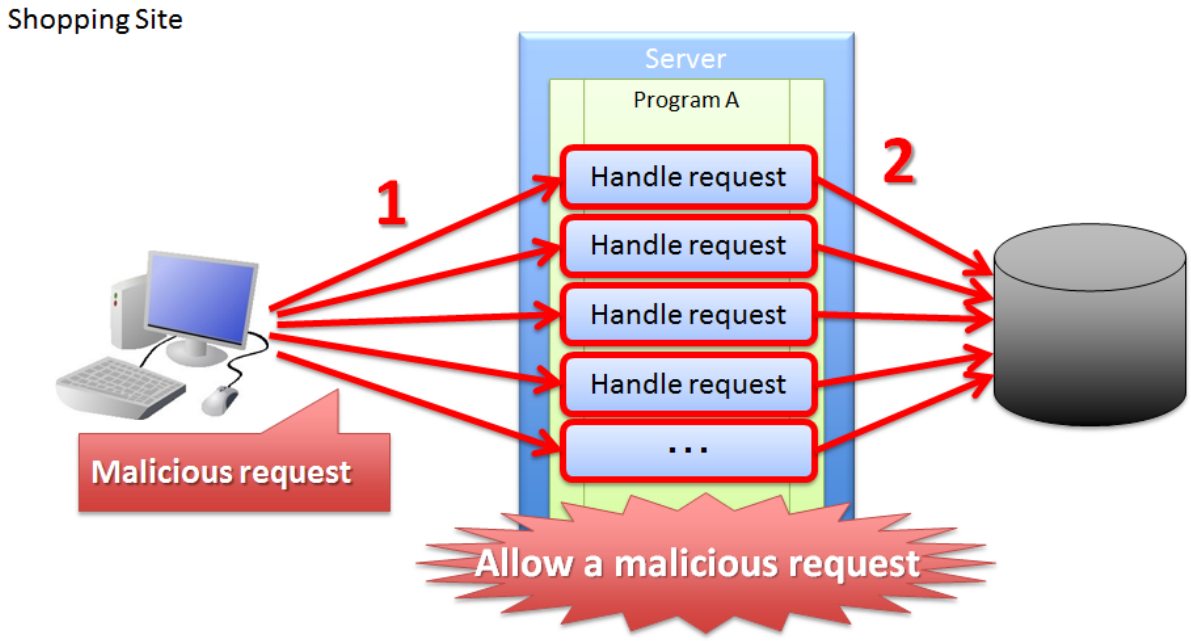
Continued on next page

表 4.28 – continued from previous page

項番	説明
(7)	サーバは、(5) のリクエストで受けた商品の購入完了画面を応答する。

注釈: 上記のケースでは、購入者の操作ミスではないため、購入者に対して問題が発生することはない。

ただし、不正な画面操作を行った後でも更新処理が実行できてしまうと、以下のような問題が発生する。



警告: 上記のケースのように、不正な画面操作を行った後でも更新処理が実行できてしまうと、悪意のある攻撃者によって、正規のルートを経由せずに直接更新処理が実行される危険度が高まる。

項番	説明
(1)	攻撃者が、正規の画面遷移を行わずに、直接商品の購入を行う処理に対してリクエストを実行する。
(2)	サーバは、不正なルートでリクエストが行われていることを検知することができないため、リクエストで受けた商品の購入処理を DB に対して反映してしまう。

不正なリクエストによって購入処理を実行することで、各サーバの負荷が高くなったり、正規のルートで商品が購入できなくなるなどの問題が発生してしまう。結果的に、正規のルートで購入している利用者に対して問題が波及する事になるため、アプリケーションとして上記の問題が発生しないように制御する事が望ましい。

Solutions

上記の問題を解決する方法として、下記の対策が必要になる。

リクエストの改竄など悪意あるオペレーションを考慮すると、**(3) の「トランザクショントークンチェックの適用」は必須である。**

項番	Solution	概要
(1)	JavaScript によるボタンの 2 度押し防止	更新処理を行うボタンを押下した際に、JavaScript によるボタン制御を行うことで、2 度押しされた際にリクエストが送信されないようにする。
(2)	PRG(Post-Redirect-Get) パターンの適用	更新処理を行うリクエスト (POST メソッドによるリクエスト) に対する応答としてリダイレクトを返却し、その後ブラウザから自動的にリクエストされる GET メソッドの応答として遷移先の画面を返却するようにする。 PRG パターンを適用することで、画面表示後にページの再読み込みを行った場合に発生するリクエストが GET メソッドになるため、更新処理の再実行を防ぐことが出来る。
(3)	トランザクショントークンチェックの適用	画面遷移毎にトークン値を払い出し、ブラウザから送信されたトークン値とサーバ上で保持しているトークン値を比較することで、トランザクション内で不正な画面操作が行われないようにする。 トランザクショントークンチェックを適用することで、ブラウザの戻るボタンを使ってページを移動した後の更新処理の再実行を防ぐことが出来る。 また、トークン値のチェックを行った後にサーバで管理しているトークン値を破棄することで、サーバ側の処理として二重送信を防ぐことも出来る。

注釈: 「トランザクショントークンチェックの適用」のみの対策だと、単純な操作ミスを行った場合でもトランザクショントークンエラーとなるため、利用者に対してユーザビリティの低いアプリケーショ

ンになってしまう。

ユーザビリティを確保しつつ、二重送信で発生する問題を防止するためには、「JavaScript によるボタンの 2 度押し防止」及び「PRG(Post-Redirect-Get) パターンの適用」が必要となる。

本ガイドラインでは、全ての対策を行うことを推奨するが、アプリケーションの要件によって対策の有無は判断すること。

> **警告:** Ajax と Web サービスでは、リクエスト毎に変更されるトランザクショントークンの受け渡しを行いにくいため、トランザクショントークンチェックを使用しなくてよい。Ajax の場合は、JavaScript によるボタンの 2 度押し防止のみで二重送信防止を行う。

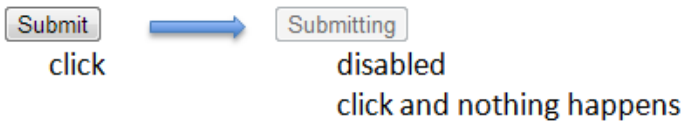
JavaScript によるボタンの 2 度押し防止について

更新処理を行うボタンや、時間のかかる検索処理を行うボタンなどに対して、ボタンの二重クリックを防止する。
ボタンが押された際に、JavaScript を使用してボタンやリンクの無効化の制御を行う。
無効化するための代表的な制御例としては、

1. ボタンやリンクを非活性化することで、ボタンやリンクを押下できないように制御する。
2. 処理状態をフラグとして保持しておき、処理中にボタンやリンクが押された場合に処理中であることを通知するメッセージを表示する。

などがあげられる。

下記は、ボタンを非活性化した際のイメージとなる。

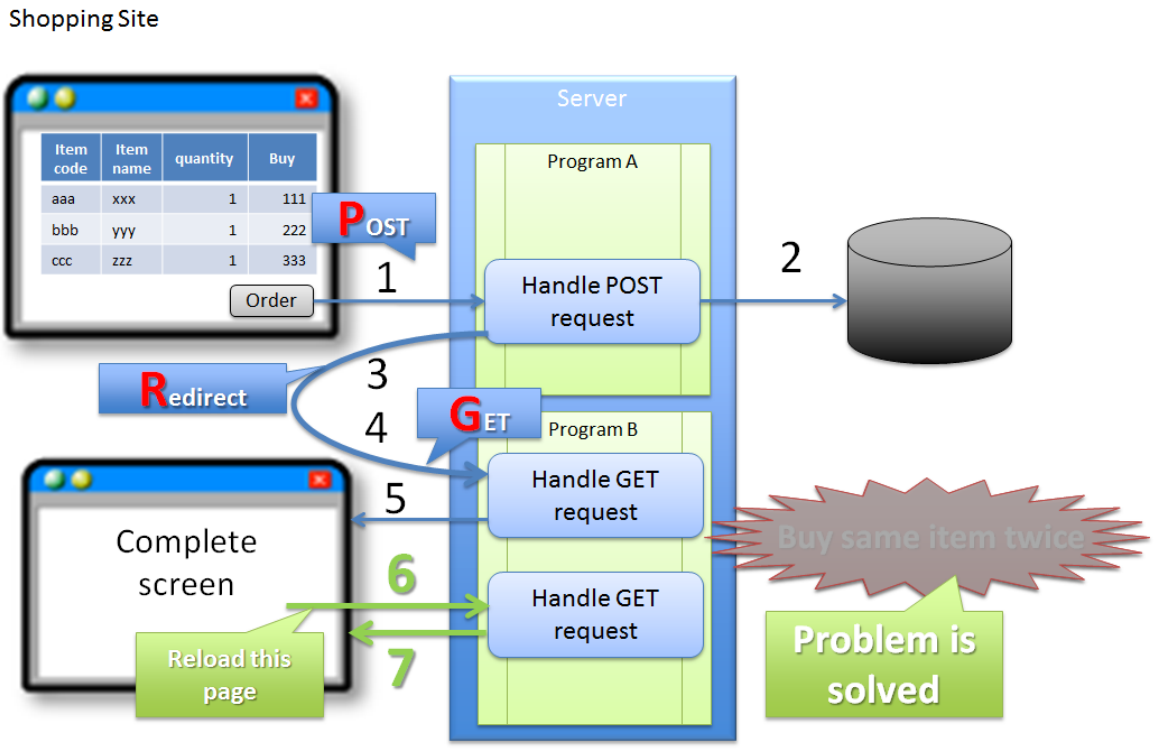


> **警告:** 画面上に存在する全てのボタン及びリンクを無効化してしまうと、サーバからの応答がない場合に、画面操作が行えなくなってしまう。そのため、「前画面に戻る」や「トップ画面へ移動」などのイベントを実行するボタンやリンクは無効化しないようにすることを推奨する。

PRG(Post-Redirect-Get) パターンについて

更新処理を行うリクエスト (POST メソッドによるリクエスト) に対する応答としてリダイレクトを返却し、その後ブラウザから自動的にリクエストされる GET メソッドの応答として遷移先の画面を返却するようにする。

PRG パターンを適用することで、画面表示後にページの再読み込みを行った場合に発生するリクエストが GET メソッドになるため、更新処理の再実行を防ぐことが出来る。



項番	説明
(1)	購買者が、商品購入画面で注文ボタンをクリックする。 **リクエストは、POST メソッドを使って送信される。**
(2)	サーバは、(1) のリクエストで受けた商品の購入処理を DB に対して反映する。
(3)	**サーバは、商品の購入完了画面を表示するための URL に対するリダイレクト応答を行う。**
(4)	ブラウザは、商品の購入完了画面を表示するための URL にリクエストを送信する。 **リクエストは、GET メソッドを使って送信される。**
(5)	サーバは、商品の購入完了画面を応答する。
(6)	購買者が、誤ってブラウザのリロード機能を実行する。 リロード機能によって要求されるリクエストは、商品の購入完了画面を表示するためのリクエストとなるため、 **更新処理が再実行されることはない。**
(7)	サーバは、商品の購入完了画面を応答する。

注釈: 更新処理を伴う処理の場合は、PRG パターンを適用し、ブラウザの更新ボタンが押された際に、GET メソッドのリクエストが送信されるように制御することを推奨する。

警告: PRG パターンでは、完了画面でブラウザの戻るボタンを押下することで、更新処理を再度実行されることを防ぐことはできない。ブラウザの戻るボタンを使用して不正な遷移をした画面から、更新処理の再実行を防ぐ場合は、トランザクショントークンチェックを行う必要がある。

トランザクショントークンチェックについて

トランザクショントークンチェックは、

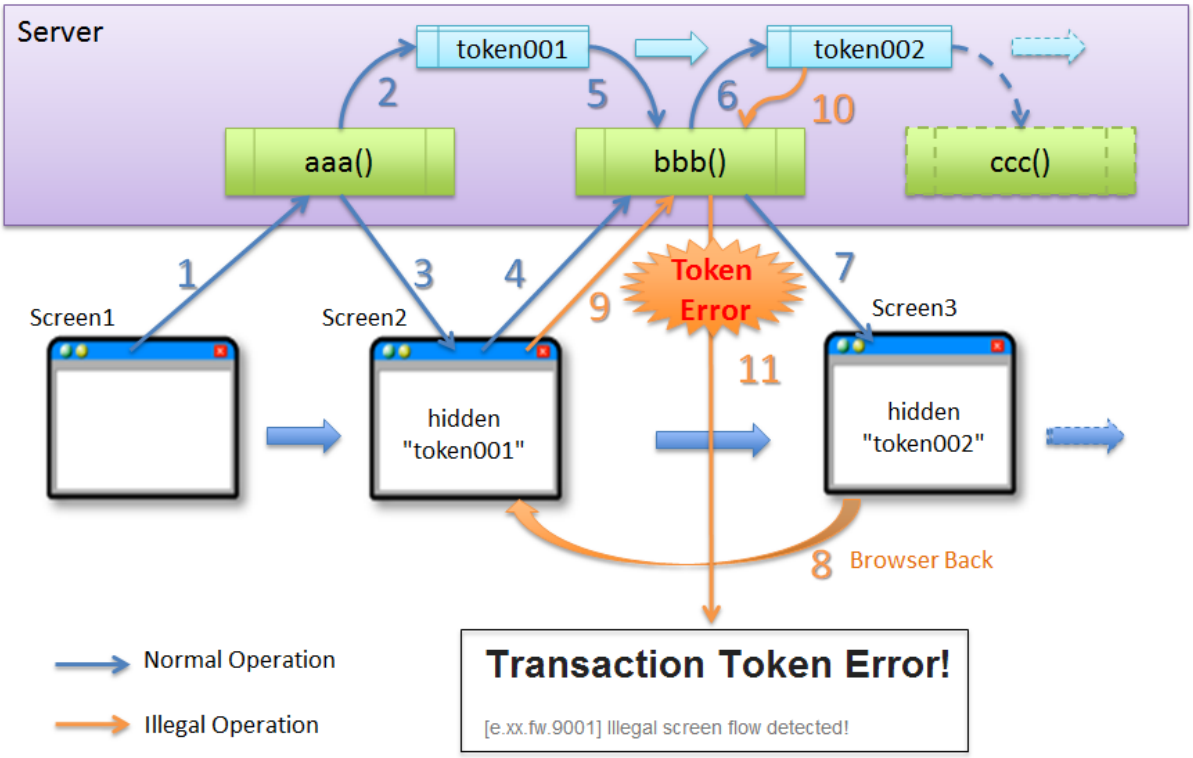
- サーバは、クライアントからリクエストが来た際に、サーバ上にトランザクションを一意に識別するための値（以下、トランザクショントークン）を保持する。
- サーバは、クライアントへトランザクショントークンを引き渡す。画面を提供する Web アプリケーションの場合は、form の hidden タグを使用してクライアントにトランザクショントークンを引き渡す。
- クライアントは次のリクエストを送信する際に、サーバから引き渡されたトランザクショントークンを送る。サーバは、クライアントから受け取ったトランザクショントークンと、サーバ上で管理しているトランザクショントークンを比較する。

という、３つの処理で構成され、リクエストで送信されてきたトランザクショントークン値と、サーバ上で保持しているトランザクショントークン値が一致していない場合は、不正なリクエストとみなしてエラーを返す。

> **警告:** トランザクショントークンチェックの濫用は、アプリケーションのユーザビリティ低下につながるため、以下の点を考慮して、適用範囲を決めること。
>
> - データの更新を伴わない参照系のリクエストや、単に画面遷移のみ行うリクエストについては、トランザクショントークンチェックの範囲に含める必要はない。
>
> 必要以上にトランザクションの範囲を広げてしまうと、トランザクショントークンエラーが発生しやすくなるため、アプリケーションのユーザビリティを低下させる事になる。
> - ビジネス観点で何回更新されても問題ないような処理（ユーザー情報更新など）では、トランザクショントークンチェックは必須ではない。
> - 入金処理や商品の購入処理など、処理が二重で実行されると問題がある場合は、トランザクショントークンチェックが必須である。

以下に、トランザクショントークンチェック使用時において、想定通りの操作を行った場合の処理フローと、想定外の操作を行った場合の処理フローについて説明する。

想定通りの操作を行った場合の処理フローについて説明する。



項番	説明
(1)	クライアントから、リクエストを送信する。
(2)	サーバは、トランザクショントークン (token001) を作成し、サーバ上で保持する。
(3)	サーバは、作成したトランザクショントークン (token001) を、クライアントに引き渡す。
(4)	クライアントから、トランザクショントークン (token001) を含めたリクエストを送信する。
(5)	サーバは、サーバ上で保持しているトランザクショントークン (token001) と、クライアントから送信されたトランザクショントークン (token001) が同一かチェックする。 値が同一なので、正規のリクエストと判断される。
(6)	サーバは、次のリクエストで使用するトランザクショントークン (token002) を生成し、サーバ上で管理している値を更新する。 この時点で、トランザクショントークン (token001) は破棄される。

Continued on next page

表 4.30 – continued from previous page

項番	説明
(7)	サーバは、更新したトランザクショントークン (token002) を、クライアントに引き渡す。

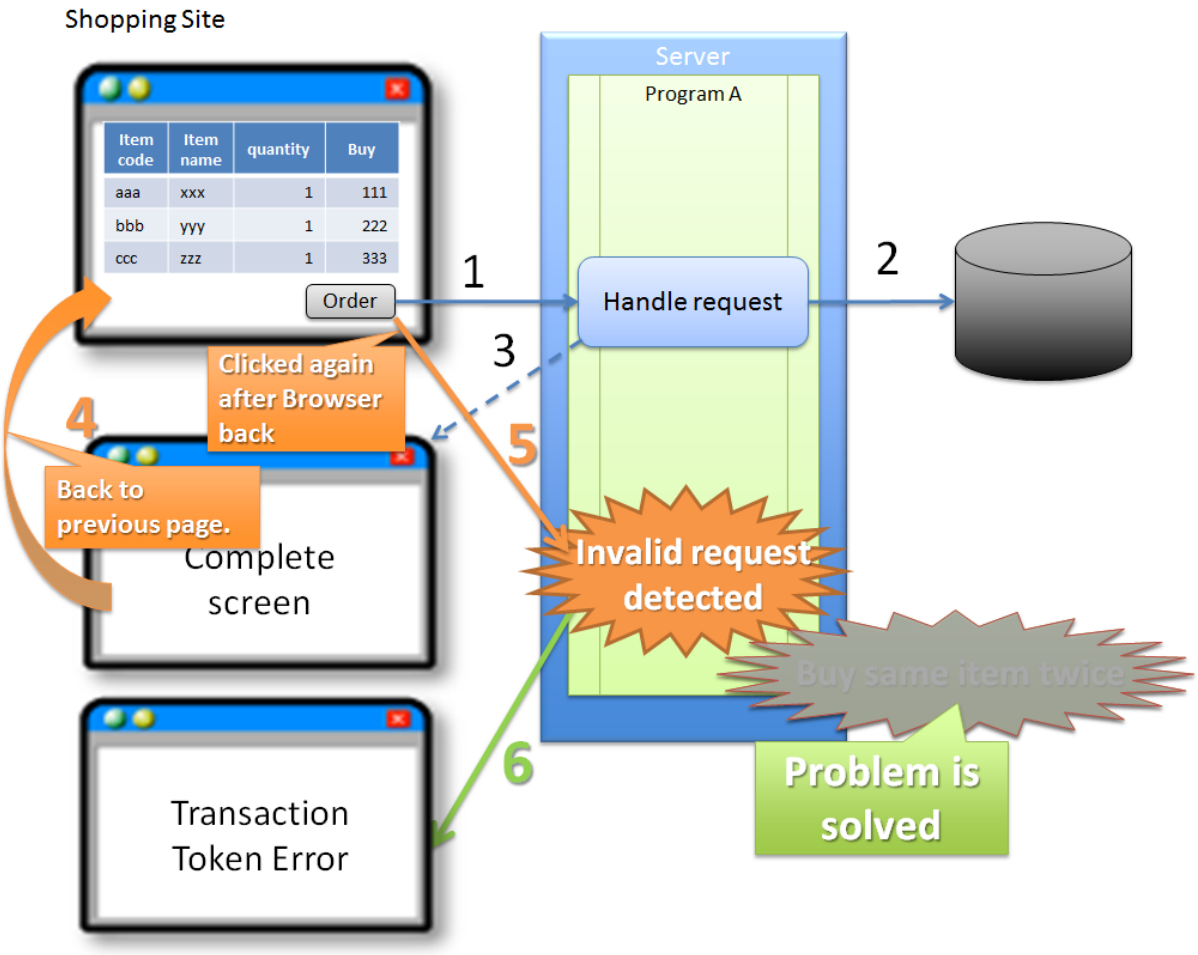
想定外の操作を行った場合の処理フローについて説明する。
ここではブラウザの戻るボタンを例にしているが、ショートカットからの直接リクエストなどでも同様である。

項番	説明
(8)	クライアントでブラウザの戻るボタンをクリックする。
(9)	クライアントから戻った画面にあるトランザクショントークン (token001) を含めたリクエストを送信する。
(10)	サーバは、サーバ上に保持しているトランザクショントークン (token002) と、クライアントから送信されたトランザクショントークン (token001) が同一かチェックする。 値が同一ではないので、 不正なリクエストと判断し、トランザクショントークンエラーとする。
(11)	サーバは、トランザクショントークンエラーが発生した事を通知するエラー画面を応答する。

トランザクショントークンチェックで防ぐことが出来るのは、以下の 3 つの事象である。

- 決められた画面遷移を行うことが求められる業務において、不正な画面遷移が行われる。
- 正規の画面遷移を伴わない不正なリクエストによって、データが更新される。
- 二重送信によって、更新処理が重複して実行される。

以下のフローによって、決められた画面遷移を行うことが求められる業務において、不正な画面遷移が行われる事を防ぐ事ができる。



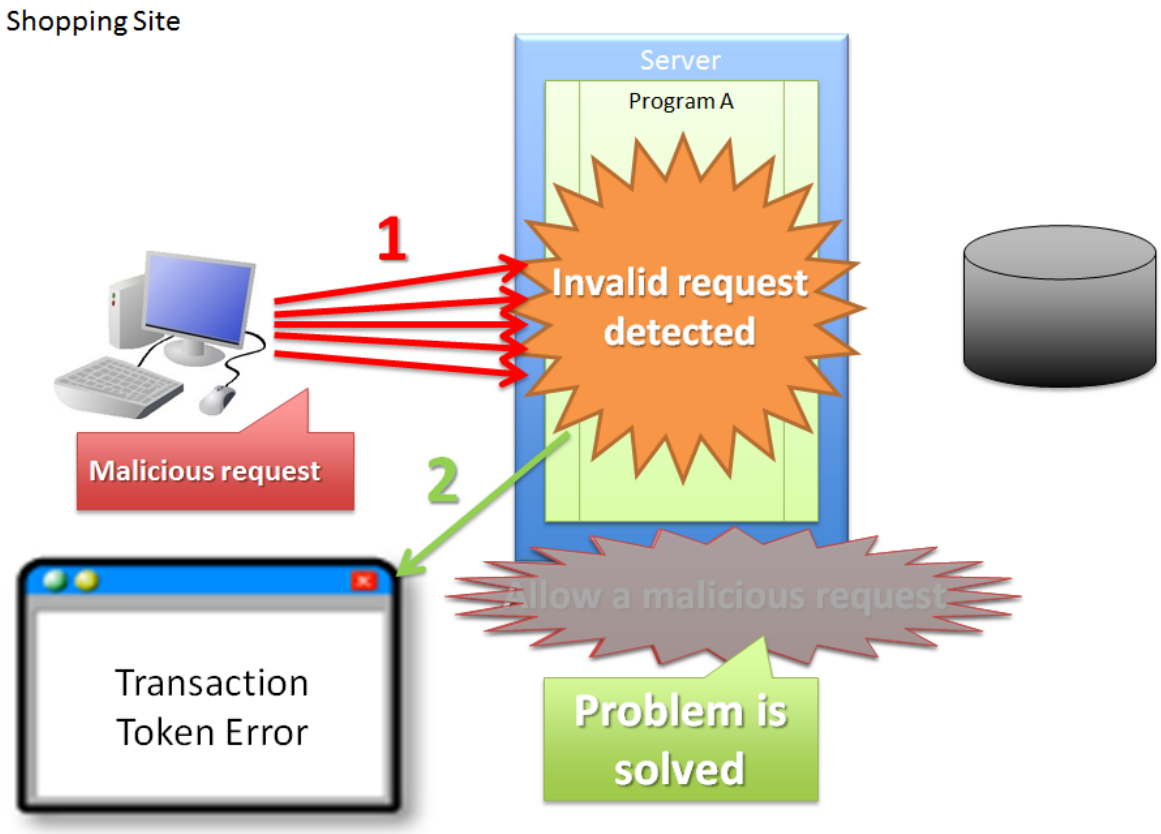
項番	説明
(1)	購買者が、商品購入画面で注文ボタンをクリックする。 サーバ上で保持しているトランザクショントークンと、クライアントから送信されたトランザクショントークンが一致するため、商品を購入する処理を実行する。 このタイミングで、サーバ上で保持していたトランザクショントークの値が破棄され、新しいトークン値に更新される。
(2)	サーバは、(1) のリクエストで受けた商品の購入処理を DB に対して反映する。
(3)	サーバは、(1) のリクエストで受けた商品の購入完了画面を応答する。

Continued on next page

表 4.31 – continued from previous page

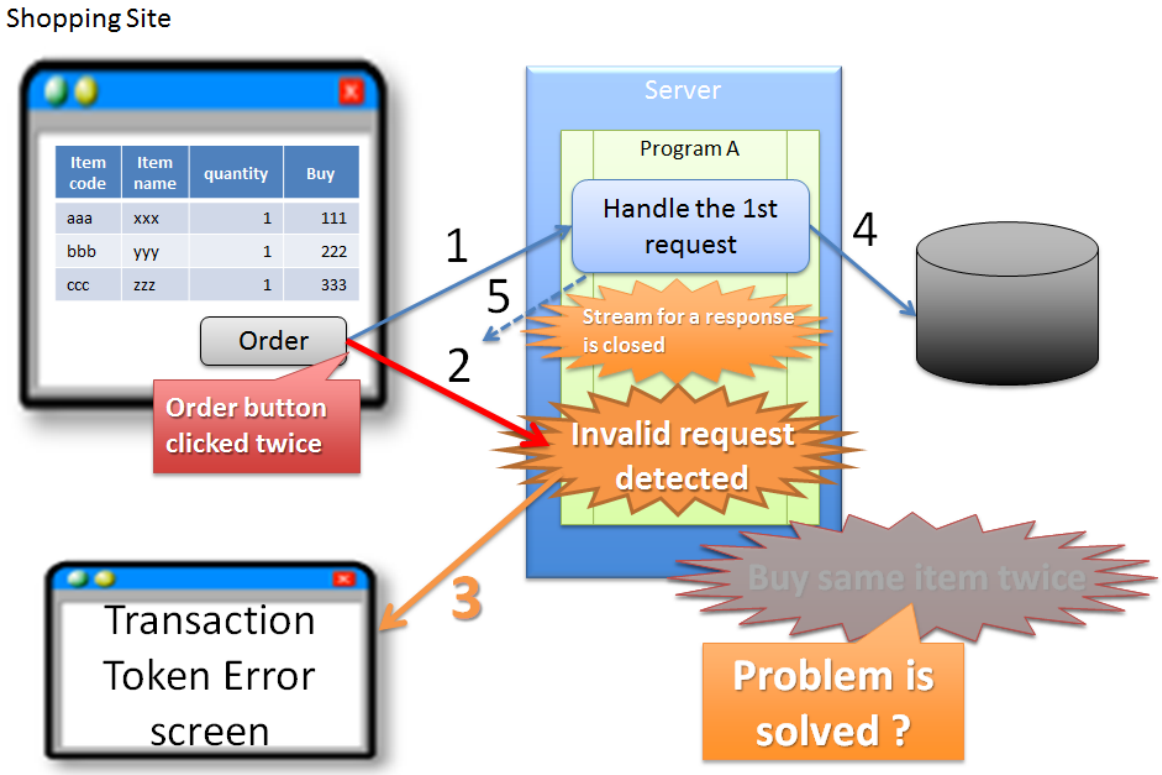
項番	説明
(4)	購買者が、ブラウザの戻るボタンを使って購入画面を再度表示する。
(5)	購買者が、ブラウザの戻るボタンを使って表示した購入画面で注文ボタンを再度クリックする。 クライアントから送信されたトランザクショントークンは既に破棄された値のため、トランザクショントークンエラーとなる。
(6)	サーバは、トランザクショントークンエラーが発生した事を通知するエラー画面を応答する。

以下のフローによって、正規の画面遷移を伴わない不正なリクエストでデータが更新される事を防ぐことができる。



項番	説明
(1)	攻撃者が、正規の画面遷移を行わずに、直接商品の購入を行う処理に対してリクエストを実行する。 トランザクショントークンを生成するためのリクエストを実行していないため、トランザクショントークンエラーとなる。
(2)	サーバは、トランザクショントークンエラーが発生した事を通知するエラー画面を応答する。

以下のフローによって、二重送信発生時に更新処理が重複して実行される事を防ぐことができる。



項番	説明
(1)	購買者が、商品購入画面で注文ボタンをクリックする。 サーバ上で保持しているトランザクショントークンと、クライアントから送信されたトランザクショントークンが一致するため、商品を購入する処理を実行する。 **このタイミングで、サーバ上で保持していたトランザクショントークの値が破棄され、新しいトークン値に更新される。**
(2)	(1) のレスポンスが返る前に、購買者が誤って注文ボタンをもう一度クリックする。 (1) の処理が実行されることによって、 **クライアントから送信されたトランザクショントークンは既に破棄された値のため、トランザクショントークンエラーとなる。**
(3)	サーバは、(2) のリクエストに対して、 **トランザクショントークンエラーが発生した事を通知するエラー画面を応答する。**
(4)	サーバは、(1) のリクエストで受けた商品の購入処理を DB に対して反映する。
(5)	サーバは、(1) のリクエストで受けた商品の購入完了画面を応答しようとするが、(2) のリクエストが送信された事により、(1) のリクエストに対する応答を行うためのストリームが閉じられているため、購入完了画面を応答することができない。

警告: 二重送信発生時に更新処理が重複して実行される事は防ぐことが出来るが、処理が完了した事を通知する画面を応答することが出来ないという問題が残る。そのため、JavaScript によるボタンの 2 度押し防止も合わせて対応することを推奨する。

トランザクショントークンのネームスペースについて

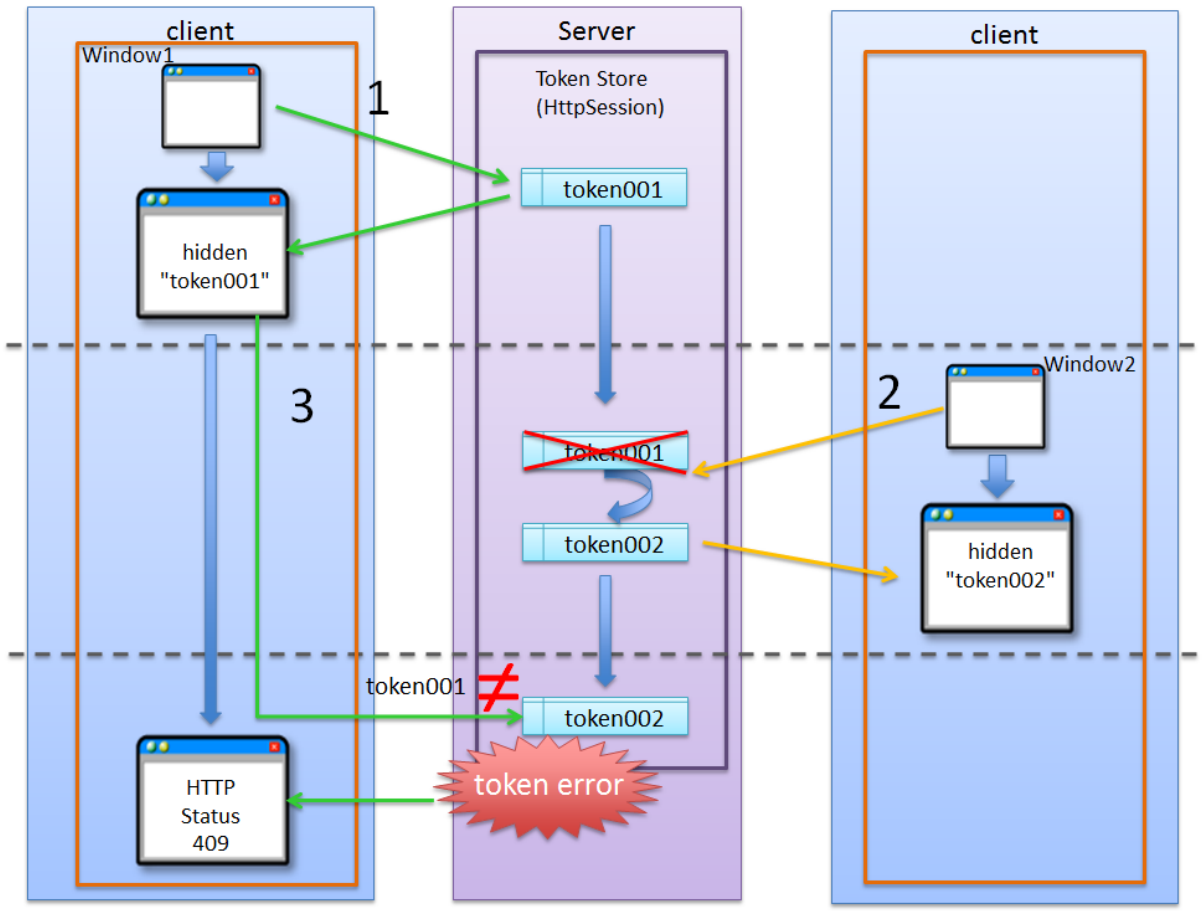
共通ライブラリから提供しているトランザクショントークンチェック機能では、トランザクショントークンを管理するための器にネームスペースを設けることが出来る。これは、タブブラウザや複数ウィンドウを使用して、更新処理を並行して操作できるようにするための仕組みである。

ネームスペースがない場合の問題点について

まず、ネームスペースがない場合の問題点について説明する。

以下の図では、client が左右にわかれているが、実際は同一マシン上に 2 つの Window を立ち上げた際の例と

なる。


client
Window1
1
hidden
"token001"
3
HTTP
Status
409
Server
Token Store
(HttpSession)
token001
token001
token002
token001
≠
token002
token error
client
2
Window2
hidden
"token002"

項番	説明
(1)	Window1 からリクエストを送信し、応答されたトランザクショントークン (token001) をブラウザに保持する。 サーバ上で保持しているトランザクショントークンは token001 となる。
(2)	Window2 からリクエストを送信し、応答されたトランザクショントークン (token002) をブラウザに保持する。 **サーバ上で保持しているトランザクショントークンは token002 となる。このタイミングで (1) の処理で生成されたトランザクショントークン (token001) は破棄される。**
(3)	Window1 からブラウザで保持しているトランザクショントークン (token001) を含めてリクエストを送信する。 サーバ上で保持しているトランザクショントークン (token002) と、リクエストで送信されたトランザクショントークン (token002) が一致しないため、不正なリクエストと判断され、トランザクショントークンエラーとなる。

警告: ネームスペースがない場合は、更新処理を並行して操作することができないため、ユーザビリティの低いアプリケーションとなってしまう。

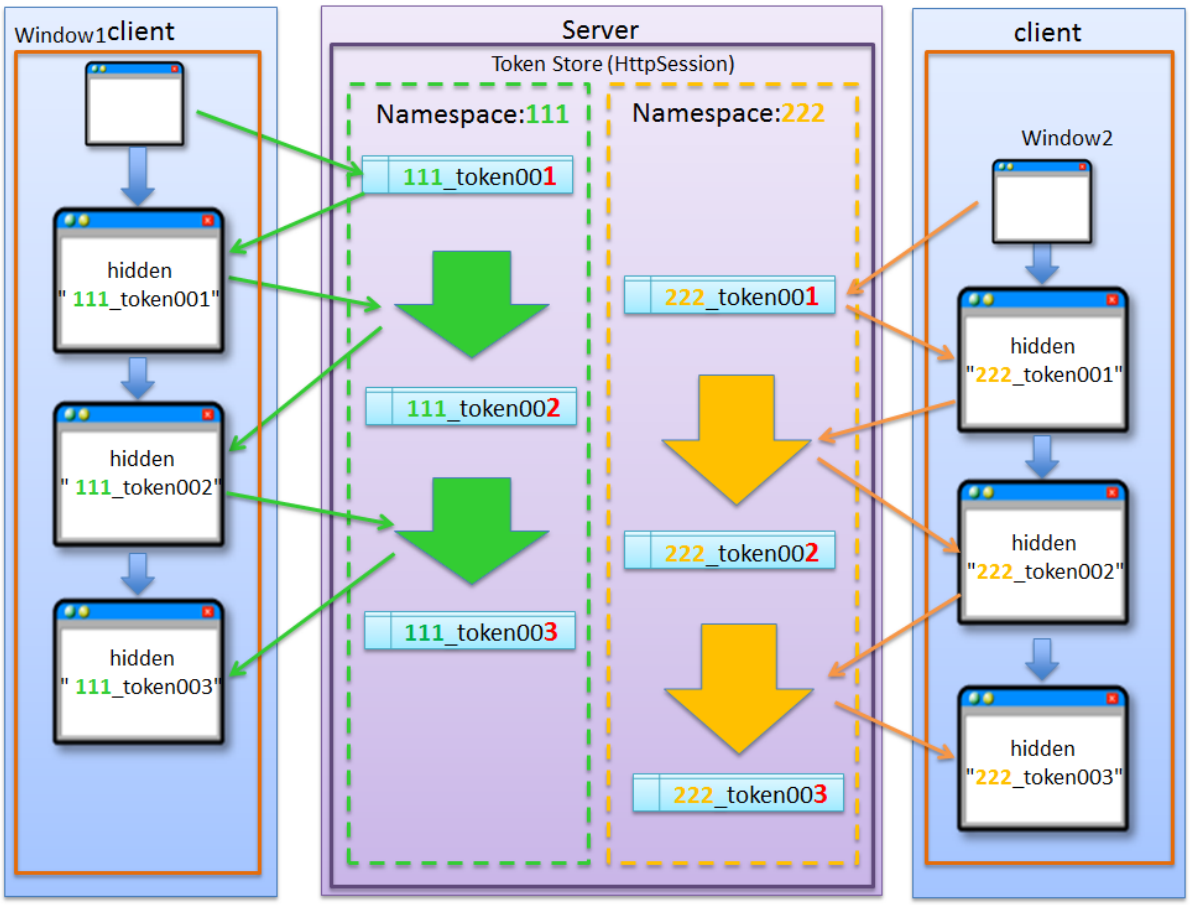
ネームスペース指定時の動作について

次に、ネームスペースを付与した際の動作について説明する。

ネームスペースがない場合は、更新処理を並行して操作することができないという問題があったが、ネームスペースも設けることで、この問題を解決することが出来る。

以下の図では、client が左右にわかれているが、実際は同一マシン上に 2 つの Window を立ち上げた際の例となる。

上記の図の、111, 222 の部分が、ネームスペースとなる。



ネームスペースを与えることで、トランザクションに割り振られたネームスペース内に存在するトランザクショントークンのみを操作するため、別のネームスペースのトランザクションに対して影響を与えない。

ここでは、ブラウザを別の Window で記述しているが、タブブラウザでも同じである。生成されるキーや使用方法については、トランザクショントークンチェックの適用で説明する。

4.5.2 How to use

JavaScript によるボタンの 2 度押し防止の適用

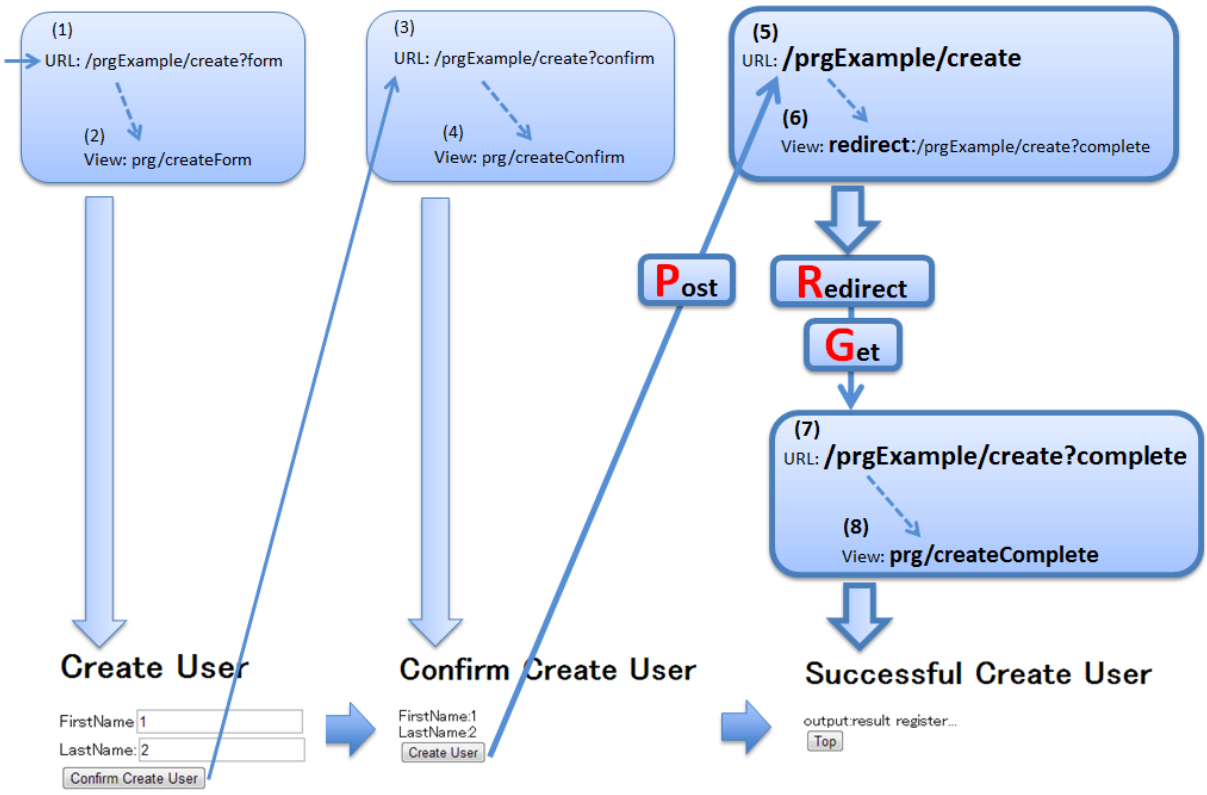
クライアントでのボタンの二重クリック防止は、JavaScript で実現することになる。

ボタンをクリックした後は、再描画するまでクリックできないようにする。

PRG(Post-Redirect-Get) パターンの適用

PRG(Post-Redirect-Get) パターンを適用する場合の実装例について説明する。

以降では、入力画面 -> 確認画面 -> 完了画面 というシンプルな画面遷移を行うアプリケーションを例に説明する。



画像の番号と、ソースのコメント番号を連動させている。

ただし、(1)～(4) については、PRG パターンと直接関係ないため、説明は省略する。

- Controller

```
@Controller
@RequestMapping("prgExample")
public class PostRedirectGetExampleController {

    @Inject
    UserService userService;

    @ModelAttribute
    public PostRedirectGetForm setUpForm() {
        PostRedirectGetForm form = new PostRedirectGetForm();
        return form;
    }

    @RequestMapping(value = "create",
                    method = RequestMethod.GET,
                    params = "form") // (1)
    public String createForm(
        PostRedirectGetForm postRedirectGetForm,
```

```
        BindingResult bindingResult) {
        return "prg/createForm"; // (2)
    }

    @RequestMapping(value = "create",
                    method = RequestMethod.POST,
                    params = "confirm") // (3)
    public String createConfirm(
        @Validated PostRedirectGetForm postRedirectGetForm,
        BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return "prg/createForm";
        }
        return "prg/createConfirm"; //  (4)
    }

    @RequestMapping(value = "create",
                    method = RequestMethod.POST) // (5)
    public String create(
        @Validated PostRedirectGetForm postRedirectGetForm,
        BindingResult bindingResult,
        RedirectAttributes redirectAttributes) {
        if (bindingResult.hasErrors()) {
            return "prg/createForm";
        }

        // omitted

        String output = "result register..."; // (6)
        redirectAttributes.addFlashAttribute("output", output); // (6)
        return "redirect:/prgExample/create?complete"; // (6)
    }

    @RequestMapping(value = "create",
                    method = RequestMethod.GET,
                    params = "complete") // (7)
    public String createComplete() {
        return "prg/createComplete"; // (8)
    }
}
```

項番	説明
(5)	確認画面の登録ボタン (Create User ボタン) が押下時の処理を行うハンドラメソッド。 **POST メソッドでリクエストを受け取る。**
(6)	**完了画面を表示するための URL へリダイレクトする。** 上記例では、`prgExample/create?complete` という URL に対して `GET` メソッドでリクエストされる。 リダイレクト先にデータを引き渡す場合は、`RedirectAttributes` の addFlashAttribute メソッドを呼び出し、引き渡すデータを追加する。 `Model` の addAttribute メソッドは、リダイレクト先にデータを引き渡すことはできない。
(7)	完了画面を表示するためのハンドラメソッド。 **GET メソッドでリクエストを受け取る。**
(8)	完了画面を表示する View(JSP) を呼び出し、完了画面を応答する。 JSP の拡張子は `spring-mvc.xml` に定義されている `ViewResolver` によって付与されるため、ハンドラメソッドの返却値からは省略している。

注釈:

- リダイレクトする際は、ハンドラメソッドの返り値として返却する遷移情報のプレフィックスとして「redirect:」を付与する。
- リダイレクト先の処理にデータを引き渡したい場合は、`RedirectAttributes` の addFlashAttribute メソッドを呼び出し、引き渡すデータを追加する。

- `createForm.jsp`

```
<h1>Create User</h1>
<div id="prgForm">
  <form:form
    action="${pageContext.request.contextPath}/prgExample/create"
    method="post" modelAttribute="postRedirectGetForm">
    <form:label path="firstName">FirstName</form:label>
    <form:input path="firstName" /><br>
    <form:label path="lastName">LastName:</form:label>
```

```
    <form:input path="lastName" /><br>
    <form:button name="confirm">Confirm Create User</form:button>
  </form:form>
</div>
```

- createConfirm.jsp

```
<h1>Confirm Create User</h1>
<div id="prgForm">
  <form:form
    action="${pageContext.request.contextPath}/prgExample/create"
    method="post"
    modelAttribute="postRedirectGetForm">
    FirstName:${f:h(postRedirectGetForm.firstName)}<br>
    <form:hidden path="firstName" />
    LastName:${f:h(postRedirectGetForm.lastName)}<br>
    <form:hidden path="lastName" />
    <form:button>Create User</form:button> <%-- (6) --%>
  </form:form>
</div>
```

項番	説明
(6)	更新処理を行うためのボタンが押下された場合は、 **POST** メソッドでリクエストを送る。

- createComplete.jsp

```
<h1>Successful Create User Completion</h1>
<div id="prgForm">
  <form:form
    action="${pageContext.request.contextPath}/prgExample/create"
    method="get" modelAttribute="postRedirectGetForm">
    output:${f:h(output)}<br> <%-- (7) --%>
    <form:button name="backToTop">Top</form:button>
  </form:form>
</div>
```

項番	説明
(7)	リダイレクト先にて、更新処理から引き渡したデータを参照する場合は、 RedirectAttributes の **addFlashAttribute** メソッドで追加したデータの属性名を指定する。 上記例では、 output が引き渡したデータを参照するための属性名となる。

トランザクショントークンチェックの適用

トランザクショントークンチェックを適用する場合の実装例について説明する。

トランザクショントークンチェックは、Spring MVC から提供されている機能ではなく、共通ライブラリから提供している機能となる。

共通ライブラリから提供しているトランザクショントークンチェックについて

共通ライブラリから提供しているトランザクショントークンチェック機能では、

- トランザクショントークンのネームスペース化
- トランザクションの開始
- トランザクション内のトークン値チェック
- トランザクションの終了

を行うために、`@org.terasoluna.gfw.web.token.transaction.TransactionTokenCheck` アノテーションを提供している。

トランザクショントークンチェックを行う場合は、Controller クラス及び Controller クラスのハンドラメソッドに対して、 `@TransactionTokenCheck` アノテーションを付与することで、宣言的にトランザクショントークンチェックを行うことが出来る。

`@TransactionTokenCheck` アノテーションの属性について

`@TransactionTokenCheck` アノテーションに指定できる属性について説明する。

表 4.32: `@TransactionTokenCheck` アノテーションパラメタ一覧

項番	属性名	内容	default	例
1.	value	任意文字列。NameSpace として使用される。	無	`value = "create"` 引数が 1 つのみの場合は、`value =`部分は省略できる。

Continued on next page

表 4.32 – continued from previous page

項番	属性名	内容	default	例
2.	namespace	任意文字列。NameSpace として使用される。 value 属性のエイリアスである。	無	`namespace = "create"` `value = "create"`と同義である。 `@TransactionTokenCheck`をメタアノテーションとして利用する際には value 属性が使用できないため、value 属性の代わりとして利用する。

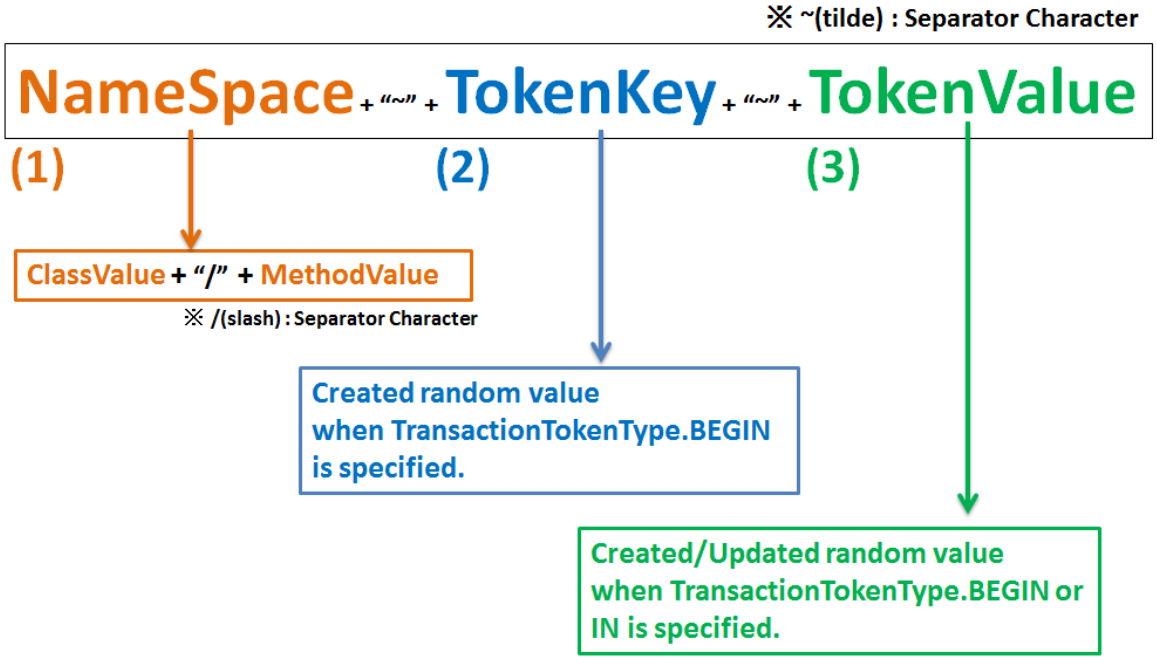
3.	type	**BEGIN**
		トランザクショントークンを作成し、新たなトランザクションを開始する。
		
		IN
		トランザクショントークンの妥当性チェックを実施する。
		リクエストされたトークン値とサーバ上で管理しているトークン値が一致している場合は、トランザクショントークンのトークン値を更新する。
		
		CHECK
		トランザクショントークンの妥当性チェックを実施する。
		リクエストされたトークン値とサーバ上で管理しているトークン値が一致している場合でも、トランザクショントークンのトークン値を更新しない。
		利用ケースは「ユースケース内にファイルダウンロード処理等の画面を更新しない処理が含まれる場合」を参照。
			IN	`type = TransactionTokenType.BEGIN`
				
				`type = TransactionTokenType.IN`
				
				`type = TransactionTokenType.CHECK`

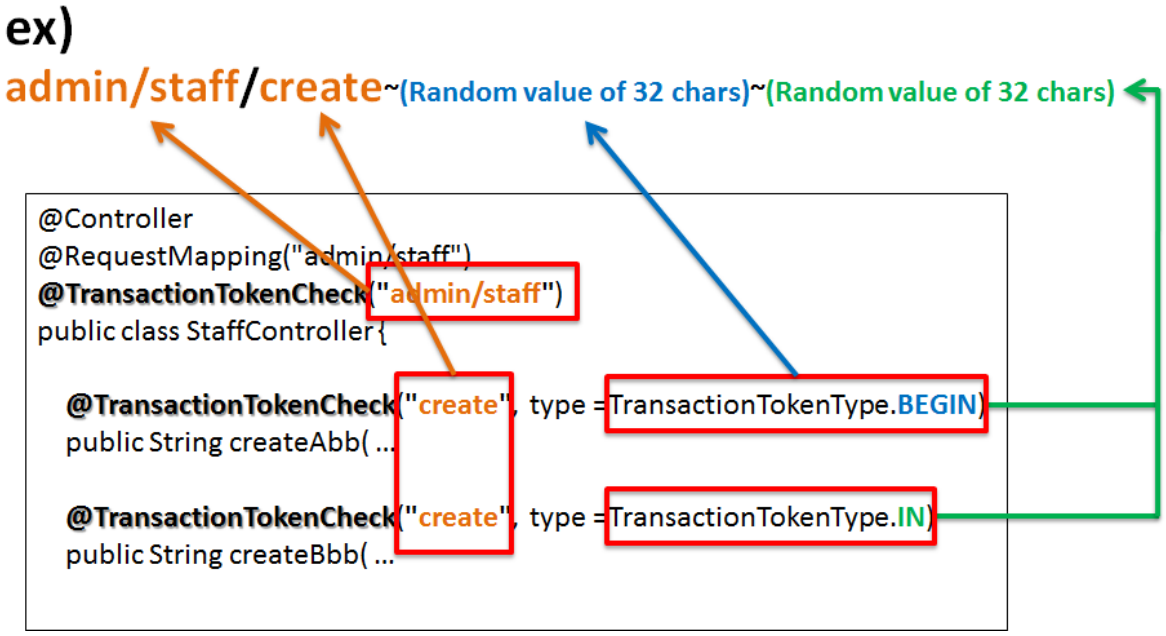
注釈: value 属性または namespace 属性に設定する値は、`@RequestMapping` アノテーションの value 属性の設定値と、同じ値を設定することを推奨する。

注釈: type 属性には、 **NONE** 及び **END** を指定することが出来るが、通常使用することはないため、説明は省略する。

トランザクショントークンの形式について

共通ライブラリから提供しているトランザクショントークンチェックで使用するトランザクショントークンは、以下の形式となる。





項番	構成要素	説明
(1)	NameSpace	• NameSpace は、一連の画面遷移を識別するための論理的な名称を付与するための要素となる。 • NameSpace を設けることで、異なる NameSpace に属するリクエストが干渉しあう事を防ぐ事が出来るため、並行して操作を行うことができる画面遷移を増やすことが出来る。 • NameSpace として使用する値は、`@TransactionTokenCheck` アノテーションの value 属性で指定した値が使用される。 • クラスアノテーションの value 属性とメソッドアノテーションの value 属性の両方を指定した場合は、両方の値を」/」で連結した値が NameSpace となる。複数のメソッドで同じ値を指定した場合は、同じ NameSpace に属するメソッドとなる。 • クラスアノテーションにのみ value 属性を指定した場合は、そのクラスで生成されるトランザクショントークンの NameSpace は、全てクラスアノテーションで指定した値となる。 • メソッドアノテーションにのみ value 属性を指定した場合は、生成されるトランザクショントークンの NameSpace はメソッドアノテーションで指定した値となる。複数のメソッドで同じ値を指定した場合は、同じ NameSpace に属するメソッドとなる。 • クラスアノテーションの value 属性とメソッドアノテーションの value 属性の両方を省略した場合は、グローバルトークンに属するメソッドとなる。グローバルトークンについては、グローバルトークンを参照されたい。

Continued on next page

表 4.33 – continued from previous page

項番	構成要素	説明
(2)	TokenKey	• TokenKey は、ネームスペース内で管理されているトランザクションを識別するための要素となる。 • TokenKey は、`@TransactionTokenCheck` アノテーションの type 属性に `TransactionTokenType.BEGIN` が宣言されているメソッドが実行されたタイミングで生成される。 • 複数の TokenKey を同時に保持することが出来る数には上限数があり、デフォルト 10 である。TokenKey の保持数は NameSpace 毎に管理される。 • `TransactionTokenType.BEGIN` 時に NameSpace 毎に管理されている保持数が最大値に達している場合は、実行された日時が最も古い TokenKey を破棄することで (Least Recently Used (LRU))、新しいトランザクションを有効なトランザクションとして管理する仕組みとなっている。 • 破棄されたトランザクショントークンを使ってアクセスした場合は、トランザクショントークンエラーとなる。
(3)	TokenValue	• TokenValue は、トランザクションのトークン値を保持するための要素となる。 • TokenValue は、`@TransactionTokenCheck` アノテーションの type 属性に `TransactionTokenType.BEGIN` 又は `TransactionTokenType.IN` が宣言されているメソッドが実行されたタイミングで生成される。

> **警告:** メソッドアノテーションにのみ value 属性を指定した場合、他の Controller で同じ値を指定している場合に、一連の画面遷移を行うためのリクエストとして扱われる点に注意する必要がある。この方法での指定は、Controller を跨いだ画面遷移を同一トランザクションとして扱いたい場合にのみ、使用すること。
>
> 原則的には、メソッドアノテーションにのみ value 属性を指定する方法は使用しない事を推奨する。

注釈: NameSpace の指定方法として、

- クラスアノテーションの value 属性とメソッドアノテーションの value 属性の両方を指定する場合
- クラスアノテーションにのみ value 属性を指定する場合

の使い分けについては、Controller の作成粒度に応じて使い分ける。

1. Controller に、複数のユースケースに対応するハンドラメソッドを実装する場合は、クラスアノテーションの value 属性とメソッドアノテーションの value 属性の両方を指定する。

   例えば、ユーザの登録、変更、削除を一つの Controller で実装する場合は、このパターンとなる。

2. Controller に、一つのユースケースに対応するハンドラメソッドを実装する場合は、クラスアノテーションにのみ value 属性を指定する。

   例えば、ユーザの登録、変更、削除毎に Controller を実装する場合は、このパターンとなる。

トランザクショントークンのライフサイクルについて

トランザクショントークンのライフサイクル (生成、更新、破棄) 制御は、以下のタイミングで行われる。

項番	ライフサイクル制御	説明
(1)	トークンの生成	`@TransactionTokenCheck` アノテーションの type 属性に `TransactionTokenType.BEGIN` が指定されたメソッドの処理が終了したタイミングで新たなトークンが生成され、トランザクションが開始される。

Continued on next page

表 4.34 – continued from previous page

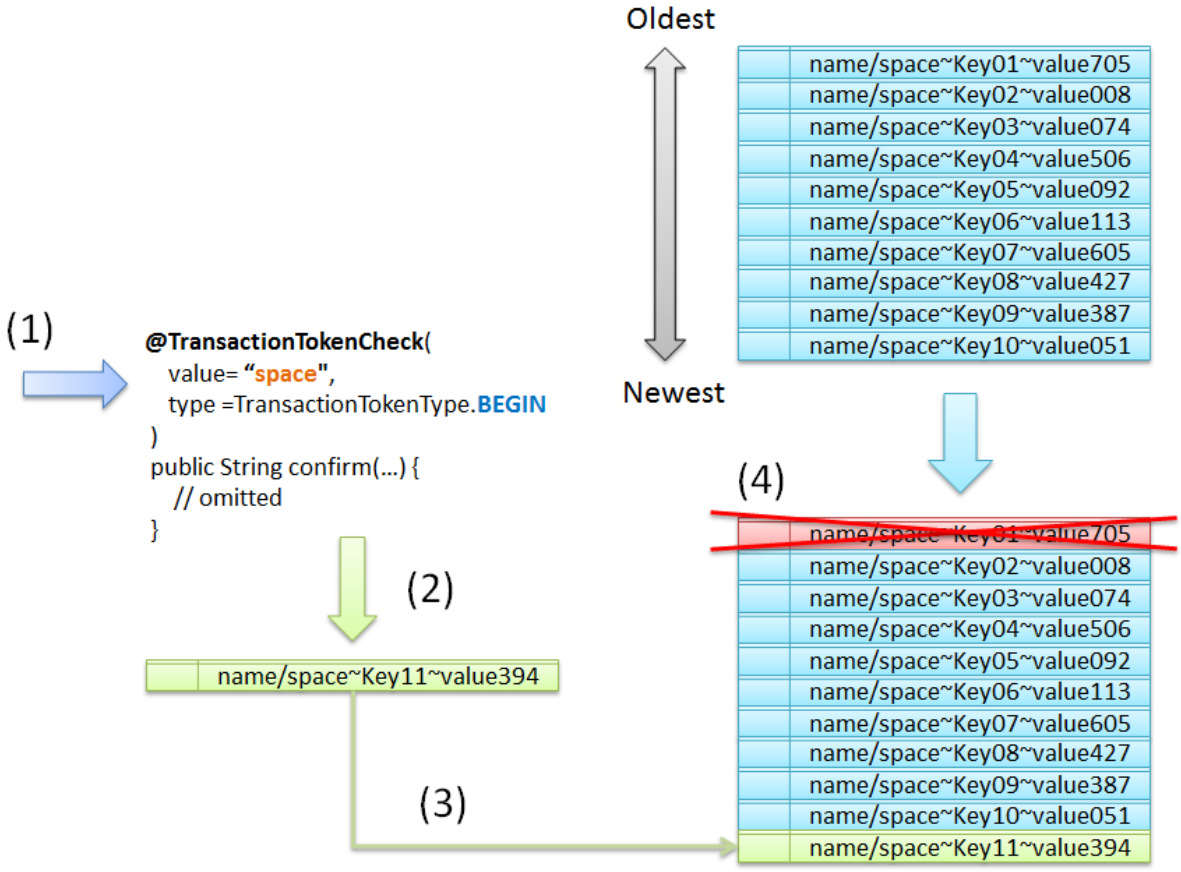
項番	ライフサイクル制御	説明
(2)	トークンの更新	`@TransactionTokenCheck` アノテーションの type 属性に `TransactionTokenType.IN` が指定されたメソッドの処理が終了したタイミングでトークン (TokenValue) が更新され、トランザクションが継続される。
(3)	トークンの破棄	以下の何れかのタイミングで破棄され、トランザクションが終了される。 [1] `@TransactionTokenCheck` アノテーションの type 属性に `TransactionTokenType.BEGIN` が指定されたメソッドを呼び出すタイミングで、リクエストパラメータに指定されているトランザクショントークンが破棄され、不要なトランザクションが終了される。 [2] NameSpace 内で保持することが出来るトランザクショントークン (TokenKey) の数が上限数に達している状態で、新たにトランザクションが開始される場合、実行された日時が最も古いトランザクショントークンが破棄され、トランザクションが強制終了される。 [3] システムエラーなどの例外が発生した場合、リクエストパラメータに指定されているトランザクショントークンが破棄され、トランザクションを終了される。
(4)	トークンの引継	`@TransactionTokenCheck` アノテーションの type 属性に `TransactionTokenType.CHECK` が指定されたメソッドの処理が終了したタイミングでサーバ上のトークンが引継がれ、トランザクションが継続される。

注釈: NameSpace 内で保持することが出来るトランザクショントークン (TokenKey) の数には上限数が設けられており、新たにトランザクショントークンを生成する際に上限値に達していた場合は、実行された日時が最も古い TokenKey をもつトランザクショントークンを破棄 (Least Recently Used (LRU)) することで、新しいトランザクションを有効なトランザクションとして管理する仕組みとなっている。

NameSpace ごとに保持できるトランザクショントークンの上限数はデフォルトで 10 となっている。上限値を変更する場合は、トランザクショントークンの上限数の変更方法についてを参照されたい。

以下に、新たにトランザクショントークンを生成する際に上限値に達していた場合の動作について説明する。
前提条件は以下の通りとする。

- NameSpace 内で保持することが出来るトランザクショントークンの数には上限数は、デフォルト値 (10) が指定されている。
- Controller のクラスアノテーションとして、 `@TransactionTokenCheck("name")` が指定されている。
- 同じ NameSpace のトランザクショントークンが上限値に達している状態である。



項番	説明
(1)	同じ NameSpace のトランザクショントークンが上限値に達している状態で、新たなトランザクションを開始するリクエストを受け付ける。
(2)	新たにトランザクショントークンを生成する。
(3)	生成したトランザクショントークンをトークン格納先に追加する。 この時点で上限数を超えるトランザクショントークンが **NameSpace** 内に存在する状態となる。
(4)	NameSpace 内で保持することが出来るトランザクショントークンの数には上限数を超える分のトランザクショントークンを削除する。 トランザクショントークンを削除する際は、実行された日時が最も古いものから順に削除する。

トランザクショントークンチェックを使用するための設定

共通ライブラリから提供しているトランザクショントークンチェックを使用するための設定を、以下に示す。

- spring-mvc.xml

```
<mvc:interceptors>
    <mvc:interceptor> <!-- (1) -->
        <mvc:mapping path="/**" /> <!-- (2) -->
        <mvc:exclude-mapping path="/resources/**" /> <!-- (2) -->
        <!-- (3) -->
        <bean
            class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>

<bean id="requestDataValueProcessor"
    class="org.terasoluna.gfw.web.mvc.support.
↪CompositeRequestDataValueProcessor">
    <constructor-arg>
        <util:list>
            <!-- (4) -->
            <bean class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenRequestDataValueProcessor" />
            <!-- omitted -->
        </util:list>
    </constructor-arg>
</bean>
```

項番	説明
(1)	トランザクショントークンの生成及びチェックを行うための HandlerInterceptor を設定する。
(2)	HandlerInterceptor を適用するリクエストパスを指定する。 上記例では、 /resources 配下へのリクエストへのリクエストを除く、全てのリクエストに対して適用している。
(3)	@TransactionTokenCheck アノテーションを使用して、トランザクショントークンの生成及びチェックを実施するためのクラス (TransactionTokenInterceptor) を指定する。
(4)	トランザクショントークンを、Spring MVC の<form:form>タグを使用して Hidden 領域に自動的に埋め込むためのクラス (TransactionTokenRequestDataValueProcessor) を設定する。

トランザクショントークンエラーをハンドリングするための設定

トランザクショントークンエラーが発生した場合は、
org.terasoluna.gfw.web.token.transaction.InvalidTransactionTokenException が発生する。

そのため、トランザクショントークンエラーをハンドリングするためには、

- applicationContext.xml に定義されている ExceptionCodeResolver
- spring-mvc.xml に定義されている SystemExceptionResolver

の設定に対して、 InvalidTransactionTokenException のハンドリング定義を追加する必要がある。

設定の追加方法については、

- 共通設定
- アプリケーション層の設定

を参照されたい。

トランザクショントークンチェックの Controller での利用方法

トランザクショントークンチェックを行う場合、Controller ではトランザクションを開始するメソッドの定義、チェックを行うメソッドの定義が必要となる。

以下では、1 つの controller で、1 つのユースケースで必要となるハンドラメソッドを実装する場合の説明となる。

- Controller

```
@Controller
@RequestMapping("transactionTokenCheckExample")
@TransactionTokenCheck("transactionTokenCheckExample") // (1)
public class TransactionTokenCheckExampleController {

    @RequestMapping(params = "first", method = RequestMethod.GET)
    public String first() {
        return "transactionTokenCheckExample/firstView";
    }

    @RequestMapping(params = "second", method = RequestMethod.POST)
    @TransactionTokenCheck(type = TransactionTokenType.BEGIN) // (2)
    public String second() {
        return "transactionTokenCheckExample/secondView";
    }

    @RequestMapping(params = "third", method = RequestMethod.POST)
    @TransactionTokenCheck // (3)
    public String third() {
        return "transactionTokenCheckExample/thirdView";
    }

    @RequestMapping(params = "fourth", method = RequestMethod.POST)
    @TransactionTokenCheck // (3)
    public String fourth() {
        return "transactionTokenCheckExample/fourthView";
    }

    @RequestMapping(params = "fifth", method = RequestMethod.POST)
    @TransactionTokenCheck // (3)
    public String fifth() {
        return "redirect:/transactionTokenCheckExample?complete";
    }

    @RequestMapping(params = "complete", method = RequestMethod.GET)
    public String complete() { // (4)
        return "transactionTokenCheckExample/fifthView";
    }

}
```

項番	説明
(1)	クラスアノテーションの value 属性で NameSpace を指定する。 上記例では、本ガイドラインの推奨パターンである `@RequestMapping` の value 属性と同じ値を指定している。
(2)	トランザクションを開始し、新しいトランザクショントークンを払い出す。 ここでは、Controller 単位でトランザクショントークンを管理するため、メソッドアノテーションの value 属性を指定しない。
(3)	トランザクショントークンをチェックし、トランザクショントークンのトークン値を更新する。 type 属性を省略した場合は、`@TransactionTokenCheck(type = TransactionTokenType.IN)` を指定した時と同じ動作となる。
(4)	ユースケースの完了を通知する画面を表示するためのリクエストでは、トランザクショントークンチェックを行う必要はないため`@TransactionTokenCheck` アノテーションの指定は行っていない。

注釈:

- `@TransactionTokenCheck` アノテーションの type 属性に BEGIN を指定した場合は、新しく TokenKey が生成されるため、トランザクショントークンのチェックは行われない。
- `@TransactionTokenCheck` アノテーションの type 属性に IN が指定された場合は、リクエストで指定されたトークン値とサーバ上で保持しているトークン値が同一のものがあるかをチェックする。

トランザクショントークンチェックの View(JSP) での利用方法

トランザクショントークンチェックを行う場合、払い出されたトランザクショントークンが、リクエストパラメータとして送信されるように View(JSP) を実装する必要がある。

リクエストパラメータとして送信されるようにする方法としては、トランザクショントークンチェックを使用するための設定を行った上で、`<form:form>`タグを使用して自動的にトランザクショントークンを hidden 要素に埋め込む方法を推奨する。

- firstView.jsp

```
<h1>First</h1>
<form:form method="post" action="transactionTokenCheckExample">
  <input type="submit" name="second" value="second" />
</form:form>
```

- secondView.jsp

```
<h1>Second</h1>
<form:form method="post" action="transactionTokenCheckExample"><!-- (1) -->
  <input type="submit" name="third" value="third" />
</form:form>
```

- thirdView.jsp

```
<h1>Third</h1>
<form:form method="post" action="transactionTokenCheckExample"><!-- (1) -->
  <input type="submit" name="fourth" value="fourth" />
</form:form>
```

- fourthView.jsp

<form:form>タグを使用する場合

```
<h1>Fourth</h1>
<form:form method="post" action="transactionTokenCheckExample"><!-- (1) -->
  <input type="submit" name="fifth" value="fifth" />
</form:form>
```

HTML の<form>タグを使用する場合

```
<h1>Fourth</h1>
<form method="post" action="transactionTokenCheckExample">
  <t:transaction /><!-- (2) -->
  <!-- (3) -->
  <input type="hidden" name="${f:h(_csrf.parameterName)}"
                       value="${f:h(_csrf.token)}"/>
  <input type="submit" name="fifth" value="fifth" />
</form>
```

- fifthView.jsp

```
<h1>Fifth</h1>
<form:form method="get" action="transactionTokenCheckExample">
  <input type="submit" name="first" value="first" />
</form:form>
```

項番	説明
(1)	JSP で、<form:form>タグを使用した場合は、@TransactionTokenCheck アノテーションの type 属性に BEGIN か IN を指定すると、name="_TRANSACTION_TOKEN"に対する Value が、hidden タグとして自動的に埋め込まれる。
(2)	HTML の<form>タグを使用する場合は、<t:transaction /> を使用することで、(1)と同様の hidden タグが埋め込まれる。
(3)	HTML の<form>タグを使用する場合は、Spring Security から提供されている CSRF トークンチェックで必要となる CSRF トークンを hidden 項目として埋め込む必要がある。 CSRF トークンチェックで必要となる CSRF トークンについては、*Spring MVC を使用した連携*を参照されたい。

注釈: <form:form>タグを使用すると、CSRF トークンチェックで必要となるパラメータも自動的に埋め込まれる。 CSRF トークンチェックで必要となるパラメータについては、*HTML フォーム使用時の連携*を参照されたい。

注釈: <t:transaction />は、共通ライブラリから提供している JSP タグライブラリである。(2)で使用している「t:」については、*インクルード用の共通 JSP の作成*を参照されたい。

- HTML の出力例

出力された HTML を確認すると、

- NameSpace は、クラスアノテーションの value 属性で指定した値が設定される。
  上記例だと、 transactionTokenCheckExample(橙色の下線) が NameSpace となる。
- TokenKey は、トランザクション開始時に払い出された値が引き回されて設定される。
  上記例だと、 c0123252d531d7baf730cd49fe0422ef(青色の下線) が TokenKey となる。
- TokenValue は、リクエスト毎に値が変化している。
  上記例だと、 3f610684e1cb546a13b79b9df30a7523、
  da770ed81dbca9a694b232e84247a13b、
  bd5a2d88ec446b27c06f6d4f486d4428(緑色の下線) が TokenValue となる。



ことが、わかる。

1 つの Controller 内で複数のユースケースを実施する場合

1 つの Controller 内で複数のユースケースの処理を実装する場合のトランザクショントークンチェックの実装例を以下に示す。

下記の例では、(2),(3),(4) を別々のユースケースの画面遷移として扱っている。

- Controller

```
@Controller
@RequestMapping("transactionTokenChecFlowkExample")
@TransactionTokenCheck("transactionTokenChecFlowkExample") // (1)
public class TransactionTokenCheckFlowExampleController {

    @RequestMapping(value = "flowOne",
                    params = "first",
                    method = RequestMethod.GET)
    public String flowOneFirst() {
        return "transactionTokenChecFlowkExample/flowOneFirstView";
    }

    @RequestMapping(value = "flowOne",
```

```
                params = "second",
                method = RequestMethod.POST)
    @TransactionTokenCheck(value = "flowOne",
                           type = TransactionTokenType.BEGIN) // (2)
    public String flowOneSecond() {
        return "transactionTokenChecFlowkExample/flowOneSecondView";
    }

    @RequestMapping(value = "flowOne",
                params = "third",
                method = RequestMethod.POST)
    @TransactionTokenCheck(value = "flowOne",
                           type = TransactionTokenType.IN)    // (2)
    public String flowOneThird() {
        return "transactionTokenChecFlowkExample/flowOneThirdView";
    }

    @RequestMapping(value = "flowTwo",
                params = "first",
                method = RequestMethod.GET)
    public String flowTwoFirst() {
        return "transactionTokenChecFlowkExample/flowTwoFirstView";
    }

    @RequestMapping(value = "flowTwo",
                params = "second",
                method = RequestMethod.POST)
    @TransactionTokenCheck(value = "flowTwo",
                           type = TransactionTokenType.BEGIN) // (3)
    public String flowTwoSecond() {
        return "transactionTokenChecFlowkExample/flowTwoSecondView";
    }

    @RequestMapping(value = "flowTwo",
                params = "third",
                method = RequestMethod.POST)
    @TransactionTokenCheck(value = "flowTwo",
                           type = TransactionTokenType.IN) // (3)
    public String flowTwoThird() {
        return "transactionTokenChecFlowkExample/flowTwoThirdView";
    }

    @RequestMapping(value = "flowThree",
                params = "first",
                method = RequestMethod.GET)
    public String flowThreeFirst() {
        return "transactionTokenChecFlowkExample/flowThreeFirstView";
    }

    @RequestMapping(value = "flowThree",
                params = "second",
                method = RequestMethod.POST)
    @TransactionTokenCheck(value = "flowThree",
```

```
                            type = TransactionTokenType.BEGIN) // (4)
    public String flowThreeSecond() {
        return "transactionTokenChecFlowkExample/flowThreeSecondView";
    }

    @RequestMapping(value = "flowThree",
                    params = "third",
                    method = RequestMethod.POST)
    @TransactionTokenCheck(value = "flowThree",
                            type = TransactionTokenType.IN) // (4)
    public String flowThreeThird() {
        return "transactionTokenChecFlowkExample/flowThreeThirdView";
    }

}
```

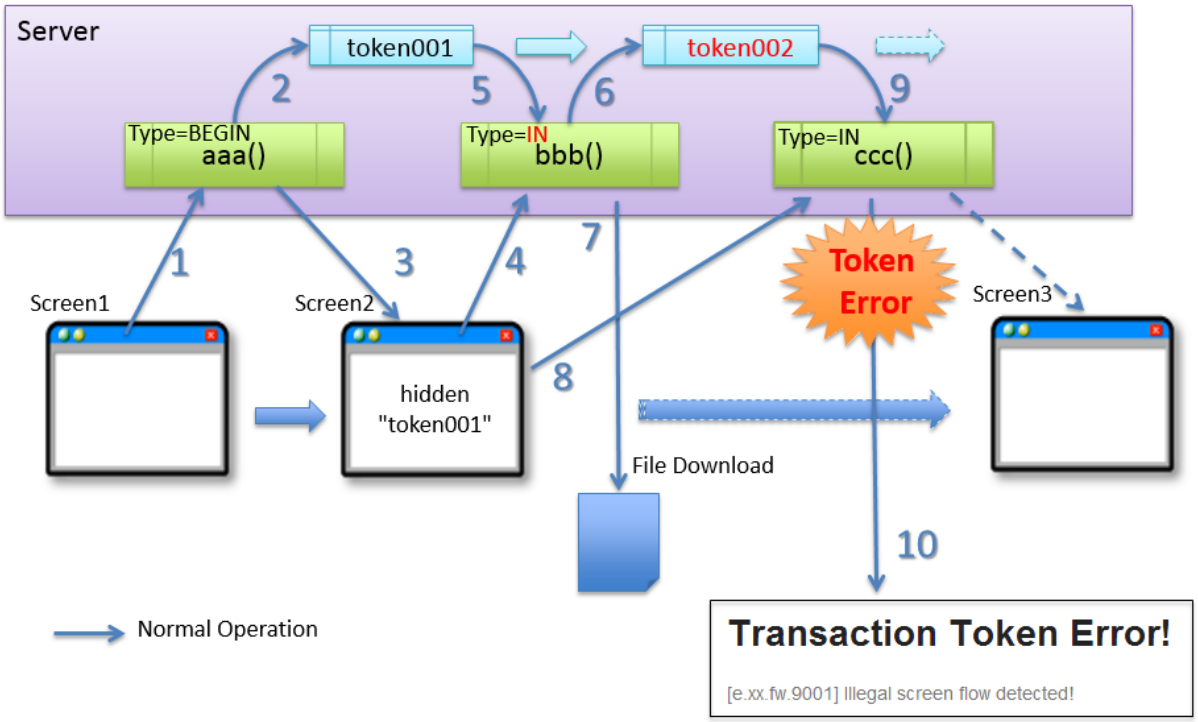
項番	説明
(1)	クラスアノテーションの value 属性で NameSpace を指定する。 上記例では、本ガイドラインの推奨パターンである `@RequestMapping` の value 属性と同じ値を指定している。
(2)	`flowOne` という名前を持つユースケースの処理に対して、トランザクショントークンチェックを行う。 上記例では、本ガイドラインの推奨パターンである `@RequestMapping` の value 属性と同じ値を指定している。
(3)	`flowTwo` という名前を持つユースケースの処理に対して、トランザクショントークンチェックを行う。 上記例では、本ガイドラインの推奨パターンである `@RequestMapping` の value 属性と同じ値を指定している。
(4)	`flowThree` という名前を持つユースケースの処理に対して、トランザクショントークンチェックを行う。 上記例では、本ガイドラインの推奨パターンである `@RequestMapping` の value 属性と同じ値を指定している。

注釈: ユースケースごとに NameSpace を割り振ることで、ユースケースごとのトランザクショントークンチェックを行うことが出来る。

ユースケース内にファイルダウンロード処理等の画面を更新しない処理が含まれる場合

ファイルダウンロード処理等のクライアントへ画面を返却しない処理を含むユースケースにおいて TransactionTokenType を正しく設定しない場合、通常のオペレーションでもトランザクショントークンエラーが発生するので注意が必要である。

不適切な TransactionTokenType の指定により通常のオペレーションでトランザクショントークンエラーが発生する例を以下に示す。



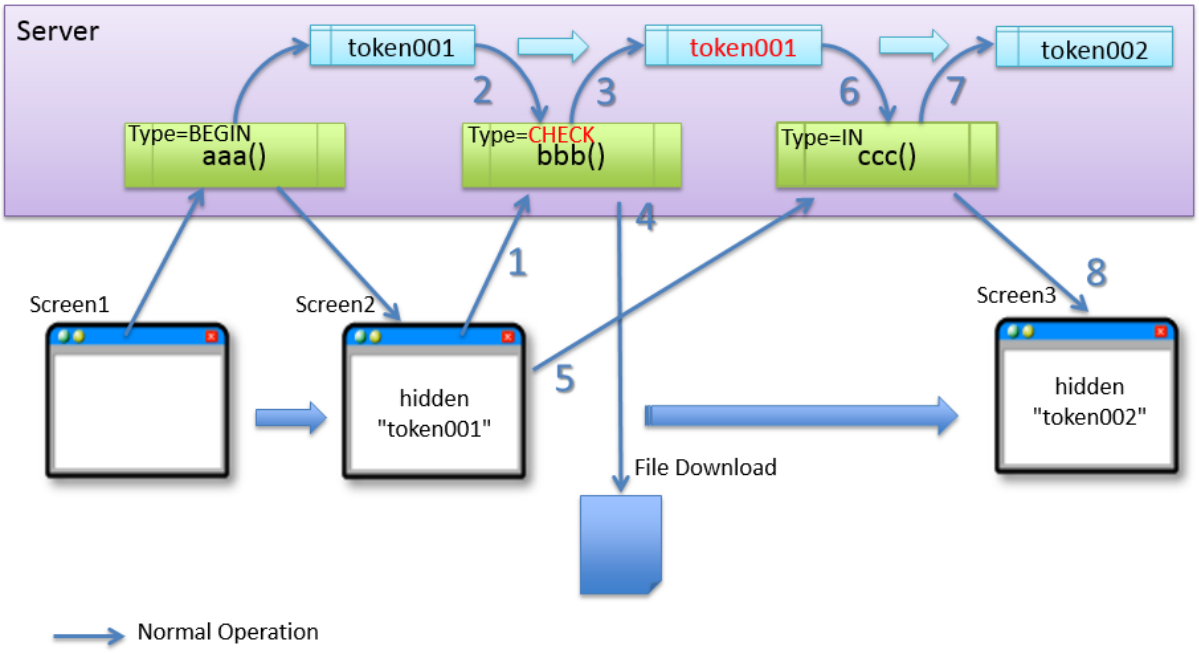
項番	説明
(1)	クライアントから、リクエストを送信する。
(2)	サーバは、トークン (token001) を作成し、サーバ上で保持する。
(3)	サーバは、作成したトークン (token001) を、クライアントに引き渡す。

Continued on next page

表 4.36 – continued from previous page

項番	説明
(4)	クライアントから、トークン (token001) を含めたファイルダウンロードのリクエストを送信する。
(5)	サーバは、サーバ上で保持しているトークン (token001) と、クライアントから送信されたトークン (token001) が同一かチェックする。 **値が同一なので、正規のリクエストと判断される。**
(6)	`TransactionTokenType` に `IN` を設定しているため、サーバは、次のリクエストで使用するトークン (token002) を生成し、サーバ上で管理している値を更新する。 この時点で、トークン (token001) は破棄される。
(7)	サーバは、要求のあったファイルを返却する。 **更新されたトークンをクライアントに返却しないため、この時点で画面のトークン (token001) とサーバのトークン (token002) が不一致となる**
(8)	クライアントから、トークン (token001) を含めたリクエストを送信する。
(9)	サーバは、サーバ上で保持しているトークン (token001) と、クライアントから送信されたトークン (token002) が同一かチェックする。 **値が不一致なので、不正なリクエストと判断される。**

上記のように、ファイルダウンロード処理を行うことができる画面（screen2）から次画面（screen3）への遷移の処理に対してトランザクショントークンを適用したい場合にファイルダウンロード処理の `TransactionTokenType` に `IN` を使用すると通常オペレーションの範囲でトークンの不一致を引き起こす。このようなケースにおいては `TransactionTokenType` に `CHECK` を使用する。



項番	説明
(1)	クライアントから、トークン (token001) を含めたファイルダウンロードのリクエストを送信する。
(2)	サーバは、サーバ上で保持しているトークン (token001) と、クライアントから送信されたトークン (token001) が同一かチェックする。 **値が同一なので、正規のリクエストと判断される。**
(3)	`TransactionTokenType` に `CHECK` を設定しているため、サーバ上で管理しているトークンを更新しない。
(4)	サーバは、要求のあったファイルを返却する。
(5)	クライアントから、トークン (token001) を含めたリクエストを送信する。

Continued on next page

表 4.37 – continued from previous page

項番	説明
(6)	サーバは、サーバ上で保持しているトークン (token001) と、クライアントから送信されたトークン (token001) が同一かチェックする。 値が同一なので、正規のリクエストと判断される。
(7)	サーバは、次のリクエストで使用するトークン (token002) を生成し、サーバ上で管理している値を更新する。 この時点で、トークン (token001) は破棄される。
(8)	サーバは、更新したトークン (token002) を、クライアントに引き渡す。

警告: サーバ側のトークンを更新させない方法として、@TransactionTokenCheck を Controller のファイルダウンロードメソッドに付与しないという方法もある。しかしながら、@TransactionTokenCheck を付与しない場合、クライアントにトークンが返却されないことを十分考慮して選択すること。

例えば、ファイルダウンロード処理中に再実行可能なエラーが発生した場合にエラーメッセージを画面に表示するなど、処理結果によって画面とファイルのいずれかを返却する可能性がある場合において、@TransactionTokenCheck を付与しない方法を選択してしまうと、エラーメッセージを画面に返却した際に画面のトークンが失われてしまう。結果として通常のオペレーションにおいてトランザクショントークンエラーを発生させてしまうことになる。

トランザクショントークンチェックの代表的な適用例

「入力画面 -> 確認画面 -> 完了画面」といったシンプルな画面遷移を行うユースケースに対して、トランザクショントークンチェックを適用する際の実装例を以下に示す。

- Controller

```
@Controller
@RequestMapping("user")
@TransactionTokenCheck("user") // (1)
public class UserController {

    // omitted

    @RequestMapping(value = "create", params = "form")
    public String createForm(UserCreateForm form) { // (2)
      return "user/createForm";
    }

    @RequestMapping(value = "create",
                  params = "confirm",
                  method = RequestMethod.POST)
    @TransactionTokenCheck(value = "create",
                         type = TransactionTokenType.BEGIN) // (3)
    public String createConfirm(@Validated
    UserCreateForm form, BindingResult result) {

        // omitted

        return "user/createConfirm";
    }

    @RequestMapping(value = "create", method = RequestMethod.POST)
    @TransactionTokenCheck(value = "create") // (4)
    public String create(@Validated
    UserCreateForm form, BindingResult result) {

        // omitted
```

```
        return "redirect:/user/create?complete";
    }

    @RequestMapping(value = "create", params = "complete")
    public String createComplete() { // (5)
        return "user/createComplete";
    }

    // omitted

}
```

項番	説明
(1)	クラスアノテーションとして、user という NameSpace を設定している。 上記例では、推奨パターンの@RequestMapping アノテーションの value 属性と同じ値を指定している。
(2)	入力画面の表示するためのハンドラメソッド。 ユースケースを開始するための画面ではあるが、データの更新を伴わない表示のみの処理であるため、トランザクションを開始する必要はない。 そのため、上記例では @TransactionTokenCheck アノテーションを指定していない。
(3)	入力チェックを行い、確認画面を表示するためのハンドラメソッド。 確認画面には更新処理を実行するためのボタンが配置されているため、このタイミングでトランザクションを開始する。 遷移先には、View（JSP）を指定する。
(4)	更新処理を実行するためのハンドラメソッド。 更新処理を行うメソッドなので、トランザクショントークンのチェックを行う。
(4)	完了画面を表示するためのハンドラメソッド。 完了画面を表示するだけなので、トランザクショントークンのチェックは不要である。 そのため、上記例では @TransactionTokenCheck アノテーションを指定していない。

> **警告:** `@TransactionTokenCheck` アノテーションを定義したハンドラメソッドの遷移先は、View(JSP) を指定する必要がある。リダイレクト先などの View（JSP）以外を遷移先に指定すると、次の処理で TransactionToken の値が変わっており、必ず TransactionToken エラーが発生する。

セッション使用時の並行処理の排他制御について

`@SessionAttributes` アノテーションを使用してフォームオブジェクトなどをセッションに格納した場合、同じ処理の画面遷移を複数並行して行うと、互いの画面操作が干渉しあい、画面に表示されている値とセッション上で保持している値が一致しなくなってしまう事がある。

こうような不整合な状態になっている画面からのリクエストを不正なリクエストとして防ぐ方法として、トランザクショントークンチェック機能を使用することができる。

NameSpace ごとに保持できるトランザクショントークンの上限数に 1 を設定する。

- `spring-mvc.xml`

```
<mvc:interceptor>
    <mvc:mapping path="/**" />
    <!-- omitted -->
    <bean
        class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenInterceptor">
        <constructor-arg value="1"/> <!-- (1) -->
    </bean>
</mvc:interceptor>
```

項番	説明
(1)	NameSpace ごとのトランザクショントークンの保持数を、」1」に設定する。

> **注釈:** `@SessionAttributes` アノテーションを使用してフォームオブジェクトなどをセッションに格納した場合は、 NameSpace ごとのトランザクショントークンの保持数を」1」に設定するとこで、古いデータを表示している画面からのリクエストを不正なリクエストとして防ぐことが可能となる。

4.5.3 How to extend

トランザクショントークンの上限数の変更方法について

以下の設定を行うことで、1 つの NameSpace 上で保持する事ができるトランザクショントークンの上限数を変更することができる。

- spring-mvc.xml

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**" />
        <mvc:exclude-mapping path="/resources/**" />
        <bean
            class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenInterceptor" />
        <constructor-arg value="5"/> <!-- (1) -->
    </mvc:interceptor>
</mvc:interceptors>
```

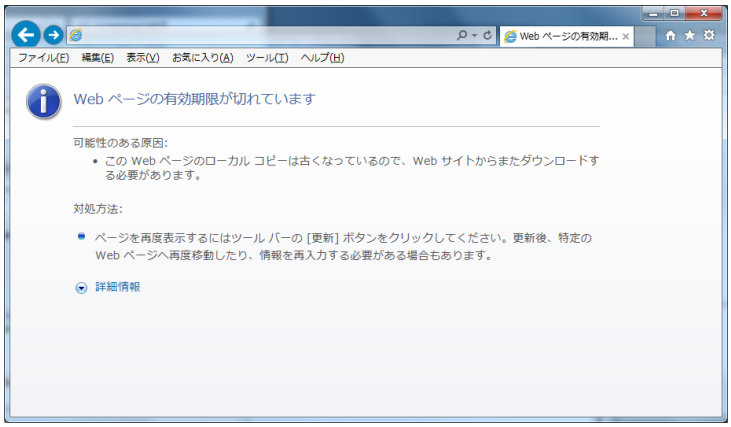
項番	説明
(1)	TransactionTokenInterceptor のコンストラクタの値として、1 つの NameSpace 上で保持する事ができるトランザクショントークンの上限数を指定する。 デフォルト値 (デフォルトコンストラクタ使用時に設定される値) は、10 となっている。 上記例では、 デフォルト値 (10) から 5 に変更している。

4.5.4 Appendix

ブラウザキャッシュ無効時のトランザクショントークンチェック

HTTP レスポンスヘッダの Cache-Control の設定により、ブラウザキャッシュが無効になっている場合は、「トランザクショントークンチェックについて」の想定外の操作を行った際に、トランザクショントークンエラーが発生する前に Web ブラウザの有効期限切れメッセージが表示される。

具体的には (8) のブラウザの戻るボタンをクリックすると以下の画面が表示される。図は Internet Explorer 11 を使用した場合である。



この場合でも二重送信自体は防止されているため、問題はない。バージョン 5.0.0.RELEASE 以降の雛形プロジェクトでは、*Spring Security* の機能でキャッシュが無効になる設定が行われている。

もしこの画面の表示が出る代わりにトランザクショントークンエラー画面を表示したい場合は、`<sec:cache-control />`の設定を除外する必要があるが、セキュリティ観点では`<sec:cache-control />`を設定しておくことを推奨する。

グローバルトークン

`@TransactionTokenCheck` アノテーションの value 属性の指定を省略すると、グローバルなトランザクショントークンとして扱われる。

グローバルなトランザクショントークンの NameSpace には、`globalToken`(固定値) が使用される。

> **注釈:** アプリケーション全体として、単一の画面遷移のみを許容する場合は、NameSpace ごとに保持できるトランザクショントークンの上限数を 1 に設定し、グルーバルトークンを使用することで実現することが出来る。

アプリケーション全体として、単一の画面遷移のみを許容する場合の設定及び実装例を以下に示す。

NameSpace ごとに保持できるトランザクショントークンの上限数の変更

NameSpace ごとに保持できるトランザクショントークンの上限数に 1 を設定する。

- `spring-mvc.xml`

```
<mvc:interceptor>
    <mvc:mapping path="/**" />
    <!-- omitted -->
    <bean
        class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenInterceptor">
        <constructor-arg value="1"/> <!-- (1) -->
    </bean>
</mvc:interceptor>
```

項番	説明
(1)	NameSpace ごとのトランザクショントークンの保持数を、」1」に設定する。

Controller の実装

グルーバルトークン用の NameSpace となるようにするために、`@TransactionTokenCheck` アノテーションの value 属性には、値を指定しない。

- Controller

```
@Controller
@RequestMapping("globalTokenCheckExample")
public class GlobalTokenCheckExampleController { // (1)

    @RequestMapping(params = "first", method = RequestMethod.GET)
    public String first() {
        return "globalTokenCheckExample/firstView";
    }

    @RequestMapping(params = "second", method = RequestMethod.POST)
    @TransactionTokenCheck(type = TransactionTokenType.BEGIN) // (2)
    public String second() {
        return "globalTokenCheckExample/secondView";
    }

    @RequestMapping(params = "third", method = RequestMethod.POST)
    @TransactionTokenCheck // (2)
    public String third() {
        return "globalTokenCheckExample/thirdView";
    }

    @RequestMapping(params = "fourth", method = RequestMethod.POST)
    @TransactionTokenCheck // (2)
    public String fourth() {
        return "globalTokenCheckExample/fourthView";
    }

    @RequestMapping(params = "fifth", method = RequestMethod.POST)
    public String fifth() {
        return "globalTokenCheckExample/fifthView";
    }

}
```

項番	説明
(1)	クラスアノテーションとして、`@TransactionTokenCheck` アノテーションを指定しない。
(2)	メソッドアノテーションとして指定する `@TransactionTokenCheck` アノテーションの value 属性を指定しない。

- HTML の出力例

JSP は、トランザクショントークンチェックの *View(JSP)* での利用方法で用意した JSP と同等のものを

用意する。

action を、`transactionTokenCheckExample` から `globalTokenCheckExample` に変更したのみで、他は同じである。



出力された HTML を確認すると、

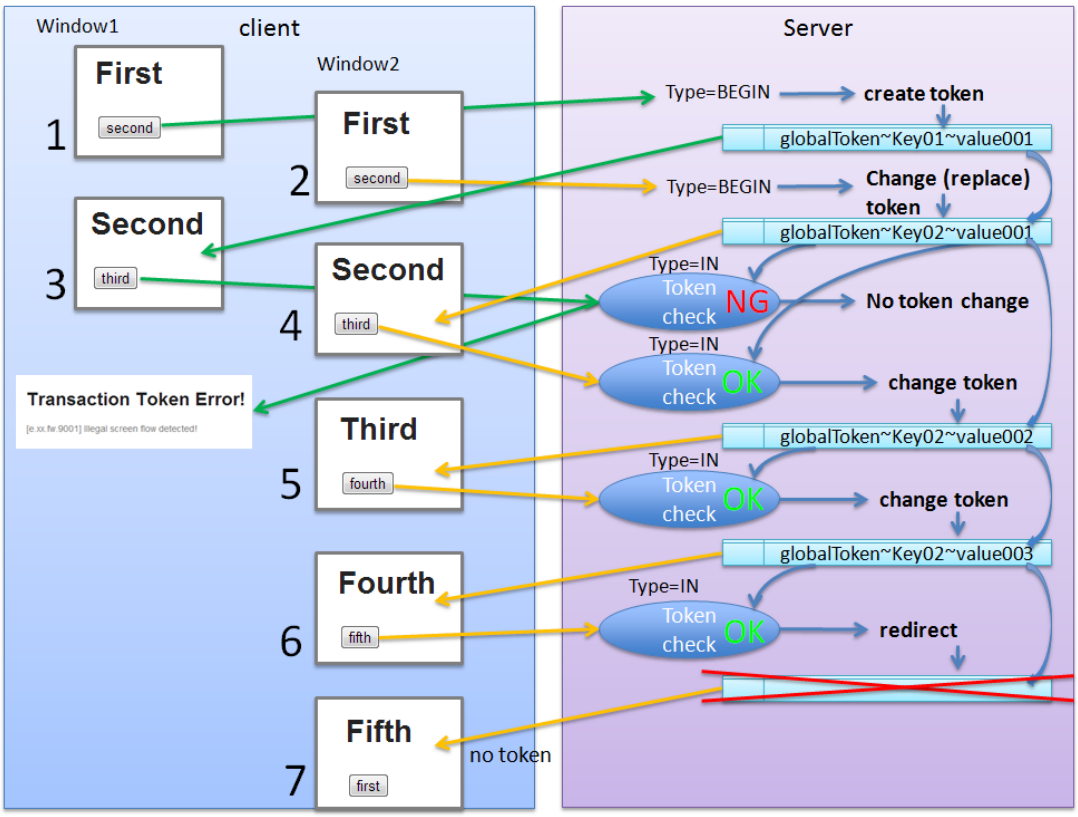
- NameSpace は、`globalToken` という固定値が設定される。
- TokenKey は、トランザクション開始時に払い出された値が引き回されて設定される。

  上記例だと、 `9d937be4adc2f5dd2032292d153f1133`(青色の下線) が TokenKey となる。
- TokenValue は、リクエスト毎に値が変化している。

  上記例だと、 `9204d7705ce7a17f16ca6cec24cfd88b`、
  `69c809fefcad541dbd00bd1983af2148`、
  `6b83f33b365f1270ee1c1b263f046719`(緑色の下線) が TokenValue となる。

ことが、わかる。

以下に、NameSpace ごとのトランザクショントークンの上限数を 1 に設定して、グローバルトークンを使用した場合の動作について説明する。



項番	説明
(1)	window1 の処理にて、TransactionTokenType.BEGIN を行い、グローバルトークンを生成する。
(2)	window2 の処理にて、TransactionTokenType.BEGIN で token を更新する。 内部的に更新ではなく入れ替えとなるが、サーバ上保持できるトランザクショントークンは 1 つなので、トークンが更新されるイメージとなる。
(3)	window1 の処理の TransactionTokenType.IN にて、token の値をチェックする。 **1 の処理で生成したトランザクショントークンをリクエストパラメータとして送信するが、サーバ上に指定したトークンが存在しないため、トランザクショントークンエラーとなる。**
(4)	window2 の処理の TransactionTokenType.IN にて、token の値をチェックする。 2 の処理で生成したトランザクショントークンをリクエストパラメータとして送信し、サーバ上で保持しているトークン値と一致することをチェックする。 一致している場合は、処理が継続される。

Continued on next page

表 4.38 – continued from previous page

項番	説明
(5)	(4) と同様。
(6)	(4) と同様。
(7)	リダイレクトを使用して画面を表示する場合は、トランザクショントークン用の hidden タグは存在しない。

注釈: サーバ上に残っているトランザクショントークンは、グローバルトークンが新たに生成されたタイミングで自動的に削除される。

Quick Reference

以下の表では、Account と Customer を管理する業務アプリケーションを例として、トランザクショントークンに関する設定をどのようにすべきか、また、その際の業務的な制限を示す。

例で示す業務アプリケーションで想定するユースケースは、Account,Customer の create,update,delete とする。

下記の表を参考に、システム要件にあったトークンの上限数と、Namespace の設定を行うこと。

番号	Namespace 毎に保持するトークン数	class で指定した namespace 値	メソッドで指定した namespace 値	生成されるトークンの例	業務制限
(1)	10 (Default)	account	指定無し	account~key~value	Account ユースケース全体 (create/update/delete) の同時実行数は、10 に制限される。
(2)	10 (Default)	account	create	account/create~key~value	Account ユースケースの create 業務の同時実行数は、10 に制限される。
(3)	10 (Default)	account	update	account/update~key~value	Account ユースケースの update 業務の同時実行数は、10 に制限される。

Continued on next page

表 4.39 – continued from previous page

番号	Namespace 毎に保持するトークン数	class で指定した namespace 値	メソッドで指定した namespace 値	生成されるトークンの例	業務制限
(4)	10 (Default)	account	delete	account/delete~key~value	Account ユースケースの delete 業務の同時実行数は、10 に制限される。((2),(3),(4) の指定で、account ユースケース全体の同時実行数は、30 になること。ほとんどのアプリケーションに対して、この設定は広過ぎるため、デフォルトの 10 より少ない値でも十分である。)
(5)	10 (Default)	指定無し	create	create~key~value	アプリケーション全体で、create という同一の Namespace が作成され、その中の同時実行数は、10 に制限される。Account と、Customer という業務が、別にあり、その中でも、create メソッドで TransactionToken の NameSpace に」create」と指定した場合、Account と、Customer の create の合計同時実行数は、10 に制限される。

Continued on next page

表 4.39 – continued from previous page

番号	Namespace 毎に保持するトークン数	class で指定した namespace 値	メソッドで指定した namespace 値	生成されるトークンの例	業務制限
(6)	10 (Default)	指定無し	update	update~key~value	(5) と同じ
(7)	10 (Default)	指定無し	delete	delete~key~value	(5) と同じ
(8)	10 (Default)	指定無し	指定無し	globalToken~key~value	Account と Customer ユースケース全体の合計同時実行数は 10 に制限される。
(9)	1 (Custom Setting in spring-mvc.xml)	account	指定無し	account~key~value	Account ユースケース全体の同時実行数は 1 に制限されること。Account の create/update/delete は同時には一つの業務しか出来ない。1 画面のみを使用した画面遷移を想定した場合、有効。
(10)	1 (Custom Setting in spring-mvc.xml)	account	create	account/create~key~value	Account ユースケースの create 業務の同時実行数は、1 に制限されること。Account の create は、2 画面開いての実行が、同時にできない。

Continued on next page

表 4.39 – continued from previous page

番号	Namespace 毎に保持するトークン数	class で指定した namespace 値	メソッドで指定した namespace 値	生成されるトークンの例	業務制限
(11)	1 (Custom Setting in spring-mvc.xml)	account	update	account/update~key~value	(10) と同じ
(12)	1 (Custom Setting in spring-mvc.xml)	account	delete	account/delete~key~value	(10) と同じ
(13)	1 (Custom Setting in spring-mvc.xml)	指定無し	create	create~key~value	アプリケーション全体で create という同一の Namespace が作成され、その中の同時実行数は、1 に制限されること。Account と Customer という業務が別にあり、create メソッドで TransactionToken の NameSpace に」create」と指定した場合、Account と Customer の create は同時に行えない。
(14)	1 (Custom Setting in spring-mvc.xml)	指定無し	update	update~key~value	(13) と同じ
(15)	1 (Custom Setting in spring-mvc.xml)	指定無し	delete	delete~key~value	(13) と同じ

Continued on next page

表 4.39 – continued from previous page

番号	Namespace 毎に保持するトークン数	class で指定した namespace 値	メソッドで指定した namespace 値	生成されるトークンの例	業務制限
(16)	1 (Custom Setting in spring-mvc.xml)	指定無し	指定無し	globalToken~key~value	アプリケーション全体で同時に実行できる業務は、1 に制限される。1 セッションでは 1 つの操作のみを許容するプロジェクトで使用すること。

4.6 メッセージ管理

4.6.1 Overview

メッセージとは、画面や帳票等に表示する固定文言、またはユーザの画面操作の結果に応じて表示する動的文言を指す。

また、エラーメッセージは、できるだけ細かく定義することを推奨する。

> **警告:** 以下の場合において、運用中、あるいは運用前の試験の際、エラーの原因を究明できなくなるリスクが生じる。(開発中は、特に困らないかもしれない。)
>
> - エラーメッセージを、1 つのみ定義している
> - エラーメッセージを、「重要」と「警告」の 2 つしか定義していない
>
> その結果、開発メンバが少ない中で、メッセージの定義変更を行い、開発が進むにつれて、修正コストが増えることになる。そのため、あらかじめメッセージは、細かい粒度で定義しておくことを推奨する。

メッセージタイプ

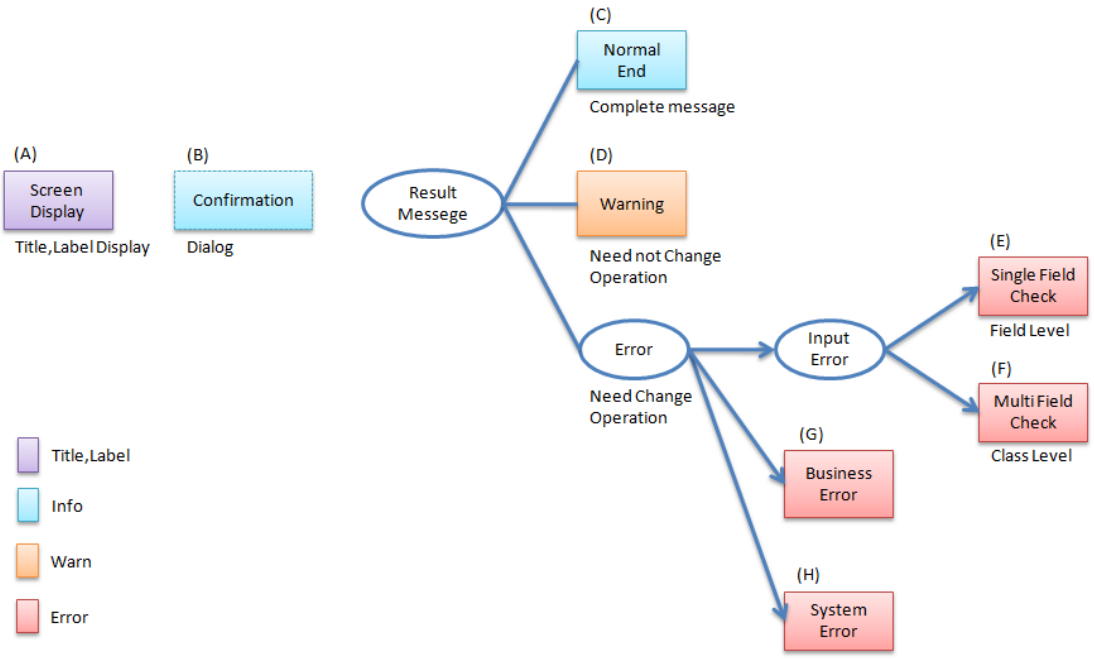
ユーザの画面操作の結果に応じて表示するメッセージは、内容に応じて、以下 3 種類のメッセージタイプに分けて管理する。

メッセージを定義する際は、出力するメッセージが、どのタイプに属するか意識すること。

メッセージタイプ	カテゴリ	概要
info	情報メッセージ	ユーザの操作による処理が正常に実行された後、画面に表示するメッセージ。
warn	警告メッセージ	処理は継続できるが、注意喚起が必要な状態である場合に表示するメッセージ。（例：パスワード有効期限切れが近い場合の通知メッセージ）
error	入力エラーメッセージ	ユーザの入力値が不正な場合に、入力画面に表示するメッセージ。
	業務エラーメッセージ	業務ロジックでエラーと判定された場合に表示するメッセージ
	システムエラーメッセージ	システム起因のエラー（データベースとの接続失敗等）が発生し、ユーザの操作でリカバリできない場合に表示するメッセージ

パターン別メッセージタイプの分類

メッセージの出力パターンを、以下に示す。



メッセージパターンとメッセージの表示内容、及びメッセージタイプを、以下に示す。

記号	パターン	表示内容	メッセージタイプ	例
(A)	タイトル	画面のタイトル	-	• 従業員登録画面
	ラベル	画面の項目名 帳票の項目名 コメント ガイダンス	-	• ユーザー名 • パスワード
(B)	ダイアログ	確認メッセージ	info	• 登録してよろしいでしょうか？ • 削除してよろしいでしょうか？
(C)	結果メッセージ	正常終了	info	• 登録しました。 • 削除しました。

Continued on next page

表 4.40 – continued from previous page

記号	パターン	表示内容	メッセージタイプ	例
(D)		警告	warn	• パスワードの有効期限切れが間近です。パスワードを変更して下さい。 • サーバが混み合っています。時間をおいてから再度実行して下さい。
(E)		単項目チェックエラー	error	•「ユーザー名」は必須です。 •「名前」は 20 桁以内で入力してください。 •「金額」には数字を入力してください。
(F)		相関チェックエラー	error	•「パスワード」と」パスワード (確認用)」が一致しません。
(G)		業務エラー	error	• キャンセル可能期間を過ぎているため、予約を取り消せません。 • 登録可能件数を超えているため、登録できません。
(H)		システムエラー	error	• XXX システム閉塞中のため、しばらく経ってから再度実行して下さい • タイムアウトが発生しました。 • システムエラーが発生しました。

メッセージ ID 体系

メッセージは、メッセージ ID をつけて管理することを推奨する。

主な理由は、以下 3 つの利点を得るためである。

- メッセージ変更時に、ソースコードを修正することなくメッセージを変更するため
- メッセージの出力箇所を特定しやすくするため
- 国際化に対応できるため

メッセージ ID の決め方は、メンテナンス性向上のため、規約を作って統一することを強く推奨する。

メッセージパターン毎のメッセージ ID 規約例を以下に示す。

開発プロジェクトでメッセージ ID 規約が定まっていない場合は、参考にされたい。

タイトル

画面のタイトルに使用する、メッセージ ID の決め方について説明する。

- フォーマット

接頭句	区切り	業務名	区切り	画面名
title	.	nnn*	.	nnn*

- 記述内容

項目	位置	内容	備考
接頭句	1-5 桁目 (5 桁)	「title」 (固定)	
業務名	可変長：任意	spring-mvc.xml で定義した viewResolver の prefix の下のディレクトリ（JSP の上位ディレクトリ）	
画面名	可変長：任意	JSP 名	ファイル名が」aaa.jsp」の場合」aaa」の部分

- 定義例

```
# "/WEB-INF/views/admin/top.jsp"の場合
title.admin.top=Admin Top
# "/WEB-INF/views/staff/createForm.jsp"の場合
title.staff.createForm=Staff Register Input
```

ちなみに: 本例は、Tiles を利用する場合に有効である。詳細は *Tiles による画面レイアウト* を参照されたい。Tiles を利用しない場合は、次に説明するラベルの規約を利用しても良い。

ラベル

画面のラベル、帳票の固定文言に使用する、メッセージ ID の決め方について説明する。

- フォーマット

接頭句	区切り	プロジェクト区分	区切り	業務名	区切り	項目名
label	.	xx	.	nnn*	.	nnn*

- 記述内容

項目	位置	内容	備考
接頭句	1-5 桁目 (5 桁)	「label」 (固定)	
プロジェクト区分	7-8 桁名 (2 桁)	プロジェクト名のアルファベット 2 桁表記	
業務名	可変長：任意		
項目名	可変長：任意	ラベル名、説明文名	

注釈: 入力チェックエラーのメッセージに項目名 (論理名) を含める場合は、

- フォームのモデル名 + 「.」 + フィールド名

```
staffForm.staffName = Staff name
```

- フィールド名

```
staffName = Staff name
```

にする必要がある。

- 使用例

```
# スタッフ登録画面のフォームの項目名
# プロジェクト区分=em (Event Management System)
label.em.staff.staffName=Staff name
# ツアー検索画面に表示する説明文の場合
# プロジェクト区分=tr (Tour Reservation System)
label.tr.tourSearch.tourSearchMessage=You can search tours with the␣
↪specified conditions.
```

注釈: プロジェクトが複数存在する場合に、メッセージ ID が重複しないようにプロジェクト区分を定義する。単一プロジェクトの場合でも、将来を見据えてプロジェクト区分を定義することを推奨する。

結果メッセージ

業務間で共通して使用するメッセージ

同一メッセージを定義しないように、複数の業務間で共有するメッセージについて説明する。

- フォーマット

メッセージタイプ	区切り	プロジェクト区分	区切り	共通メッセージ区分	区切り	エラーレベル	連番
x	.	xx	.	fw	.	9	999

- 記述内容

項目	位置	内容	備考
メッセージタイプ	1 桁目 (1 桁)	info : i warn : w error : e	
プロジェクト区分	3-4 桁目 (2 桁)	プロジェクト名のアルファベット 2 桁表記	
共通メッセージ区分	6-7 桁目 (2 桁)	「fw」 (固定)	
エラーレベル	9 桁 (1 桁)	0-1 : 正常メッセージ 2-4 : 業務エラー（準正常） 5-7 : 入力チェックエラー 8 : 業務エラー（エラー） 9 : システムエラー	
連番	10-12 桁目 (3 桁)	連番で利用する (000-999)	メッセージ削除となっても連番は空き番として、削除しない

- 使用例

```
# 登録が成功した場合（正常メッセージ）
i.ex.fw.0001=Registered successfully.
# サーバリソース不足
w.ex.fw.9002=Server busy. Please, try again.
# システムエラー発生の場合（システムエラー）
e.ex.fw.9001=A system error has occurred.
```

各業務で個別に使用するメッセージ

業務で個別に使用するメッセージについて説明する。

- フォーマット

メッセージタイプ	区切り	プロジェクト区分	区切り	業務メッセージ区分	区切り	エラーレベル	連番
x	.	xx	.	xx	.	9	999

- 記述内容

項目	位置	内容	備考
メッセージタイプ	1 桁目 (1 桁)	info : i warn : w error : e	
プロジェクト区分	3-4 桁目 (2 桁)	プロジェクト名のアルファベット 2 桁表記	
業務メッセージ区分	6-7 桁目 (2 桁)	業務 ID など業務毎に決める 2 桁の文字	
エラーレベル	9 桁 (1 桁)	0-1 : 正常メッセージ 2-4 : 業務エラー（準正常） 5-7 : 入力チェックエラー 8 : 業務エラー（エラー） 9 : システムエラー	
連番	10-12 桁目 (3 桁)	連番で利用する (000-999)	メッセージ削除となっても連番は空き番として、削除しない

- 使用例

```
# ファイルのアップロードが成功した場合
i.ex.an.0001={0} upload completed.
# パスワードの推奨変更期間が過ぎている場合
w.ex.an.2001=The recommended change interval of password has passed.␣
↪Please change your password.
# ファイルサイズが制限を超えている場合
e.ex.an.8001=Cannot upload, Because the file size must be less than {0}MB.
# データに不整合がある場合
```

```
e.ex.an.9001=There are inconsistencies in the data.
```

入力チェックエラーメッセージ

入力チェックでエラーがある場合に出力するメッセージについては、エラーメッセージの定義を参照されたい。

注釈: 入力チェックエラーの出力場所に関する基本方針を、以下に示す。

- 単項目入力チェックエラーのメッセージは、対象の項目がわかるように項目の横に表示させる。
- 相関入力チェックエラーのメッセージは、ページ上部などにまとめて表示させる。
- 単項目チェックでもメッセージを項目の横に表示させにくい場合は、ページ上部に表示させる。
  その場合は、メッセージに項目名を含める。

4.6.2 How to use

プロパティファイルに設定したメッセージの表示
プロパティを使用する際の設定

メッセージ管理を行う `org.springframework.context.MessageSource` の実装クラスの定義を行う。

- applicationContext.xml

```
<!-- Message -->
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource
↪"> <!-- (1) -->
    <property name="basenames"> <!-- (2) -->
        <list>
            <value>i18n/application-messages</value>
        </list>
    </property>
</bean>
```

項番	説明
(1)	MessageSource の定義。ここでは ResourceBundleMessageSource を使用する。
(2)	使用するメッセージプロパティの基底名を定義する。クラスパス相対で指定する。 この例では」src/main/resources/i18n/application-messages.properties」を読み込む。

プロパティに設定したメッセージの表示

- application-messages.properties

  ここでは、application-messages.properties にメッセージを定義する例を示す。

```
label.aa.bb.year=Year
label.aa.bb.month=Month
label.aa.bb.day=Day
```

注釈: 文字コード「ISO-8859-1」では表現できない文字 (日本語など) は native2ascii コマンドで ISO-8859-1 に変換して使用することが多かった。しかし、JDK 6 からは文字コードを指定できるようになったため、変換する必要はない。文字コード UTF-8 にすることで、properties ファイルに直接日本語等を使用できる。

  - application-messages.properties

```
label.aa.bb.year=年
label.aa.bb.month=月
label.aa.bb.day=日
```

  この場合、以下のように、ResourceBundleMessageSource にも読み込む文字コードを指定する必要がある。

  - applicationContext.xml

```
<bean id="messageSource"
    class="org.springframework.context.support.
↪ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>i18n/application-messages</value>
        </list>
    </property>
    <property name="defaultEncoding" value="UTF-8" />
</bean>
```

デフォルトでは ISO-8859-1 が使用されるため、日本語等を properties ファイルに直接記述したい場合は、必ず `defaultEncoding` を設定すること。

- JSP

上記で設定したメッセージを JSP からは、`<spring:message>`タグを用いて表示できる。インクルード用の共通 *JSP* の作成の設定が必要である。

```
<spring:message code="label.aa.bb.year" />
<spring:message code="label.aa.bb.month" />
<spring:message code="label.aa.bb.day" />
```

フォームのラベルと使用する場合は、以下のように使用すれば良い。

```
<form:form modelAttribute="sampleForm">
    <form:label path="year">
        <spring:message code="label.aa.bb.year" />
    </form:label>: <form:input path="year" />
    <br>
    <form:label path="month">
        <spring:message code="label.aa.bb.month" />
    </form:label>: <form:input path="month" />
    <br>
    <form:label path="day">
        <spring:message code="label.aa.bb.day" />
    </form:label>: <form:input path="day" />
</form:form>
```

ブラウザで表示すると以下のように出力される。



ちなみに: 国際化に対応する場合は、

```
src/main/resources/i18n
                    ├ application-messages.properties （英語メッセージ）
                    ├ application-messages_fr.properties （フランス語メッセージ）
                    ├ ...
                    └ application-messages_ja.properties （日本語メッセージ）
```

というように各言語用の properties ファイルを作成すればよい。詳細は、国際化を参照されたい。

結果メッセージの表示

サーバサイドでの処理の成功や、失敗を示す結果メッセージを格納するクラスとして、

共通ライブラリでは、`org.terasoluna.gfw.common.message.ResultMessages`、および `org.terasoluna.gfw.common.message.ResultMessage` を提供している。

クラス名	説明
`ResultMessages`	結果メッセージの一覧とメッセージタイプを持つクラス。 結果メッセージの一覧は `List<ResultMessage>`、メッセージタイプは `org.terasoluna.gfw.common.message.ResultMessageType` インタフェースで表現される。
`ResultMessage`	結果メッセージのメッセージ ID、または、メッセージ本文を持つクラス。

この結果メッセージを JSP で表示するための JSP タグライブラリとして、`<t:messagesPanel>`タグも提供される。

基本的な結果メッセージの使用方法

Controller で `ResultMessages` を生成して画面に渡し、JSP で`<t:messagesPanel>`タグを使用して、結果メッセージを表示する方法を説明する。

- Controller クラス

  `ResultMessages` オブジェクトの生成方法、および画面へメッセージを渡す方法を示す。
  application-messages.proerties には、各業務で個別に使用するメッセージの例が定義されていることとする。

```
package com.example.sample.app.message;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.terasoluna.gfw.common.message.ResultMessages;

@Controller
@RequestMapping("message")
public class MessageController {
```

```
  @RequestMapping(method = RequestMethod.GET)
  public String hello(Model model) {
    ResultMessages messages = ResultMessages.error().add("e.ex.an.9001"); /
↪/ (1)
    model.addAttribute(messages); // (2)
    return "message/index";
  }
}
```

項番	説明
(1)	メッセージタイプが」error」である ResultMessages を作成し、 メッセージ ID が」e.ex.an.9001」である結果メッセージを設定する。 この処理は次と同義である。 ResultMessages.error().add(ResultMessage.fromCode("e.ex.an.9001")); メッセージ ID を指定する場合は、ResultMessage オブジェクトの生成を省略できるため、省略することを推奨する。
(2)	ResultMessages を Model に追加する。 属性は指定しなくてよい。(属性名は」resultMessages」になる)

- JSP

  WEB-INF/views/message/index.jsp を、以下のように記述する。

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Result Message Example</title>
</head>
<body>
    <h1>Result Message</h1>
    <t:messagesPanel /><!-- (1) -->
</body>
</html>
```

項番	説明
(1)	<t:messagesPanel> タグをデフォルト設定で使用する。 デフォルトでは、属性名が」resultMessages」のオブジェクトを表示する。 そのため、デフォルトでは Controller から Model に ResultMessages を設定する際に、属性名を設定する必要がない。

ブラウザで表示すると、以下のように出力される。



<t:messagesPanel> によって出力される HTML を、以下に示す (説明しやすくするために整形している)。

```
<div class="alert alert-error"><!-- (1) -->
  <ul><!-- (2) -->
    <li>There are inconsistencies in the data.</li><!-- (3) -->
  </ul>
</div>
```

項番	説明
(1)	メッセージタイプに対応して」alert-error」クラスが付与されている。デフォルトでは タグの class に」error error-[メッセージタイプ]」が付与される。
(2)	結果メッセージのリストが タグで出力される。
(3)	メッセージ ID に対応するメッセージが MessageSource から解決される。

<t:messagesPanel>は class を付けた HTML を出力するだけであるため、見栄えは出力された class に合わせて CSS でカスタマイズする必要がある (後述する)。

注 釈: ResultMessages.error().add(ResultMessage.fromText("There are inconsistencies in the data.")); というように、メッセージの本文をハードコー

ドすることもできるが、保守性を高めるため、メッセージキーを使用して ResultMessage オブジェクトを作成し、メッセージ本文はプロパティファイルから取得することを推奨する。

メッセージのプレースホルダに値を埋める場合は、次のように add メソッドの第二引数以降に設定すればよい。

```
ResultMessages messages = ResultMessages.error().add("e.ex.an.8001", 1024);
model.addAttribute(messages);
```

この場合、<t:messagesPanel />タグにより、以下のような HTML が出力される。

```
<div class="alert alert-error">
  <ul>
    <li>Cannot upload, Because the file size must be less than 1,024MB.</li>
  </ul>
</div>
```

警告: terasoluna-gfw-web 1.0.0.RELEASE を使用してプレースホルダに値を埋める場合の注意点

terasoluna-gfw-web 1.0.0.RELEASE を使用している場合、**プレースホルダにユーザの入力値を埋め込むと XSS 脆弱性の危険がある。**ユーザの入力値に XSS 対策が必要な文字が含まれる可能性がある場合は、プレースホルダに値を埋め込まないようにすること。

terasoluna-gfw-web 1.0.1.RELEASE 以上を使用している場合は、ユーザの入力値をプレースホルダに埋め込んでも XSS 脆弱性は発生しない。

注釈: ResourceBundleMessageSource はメッセージを生成する際に java.text.MessageFormat を使用するため、1024 はカンマ区切りで 1,024 と表示される。カンマが不要な場合は、プロパティファイルには以下のように設定する。

```
e.ex.an.8001=Cannot upload, Because the file size must be less than {0,
↪number,#}MB.
```

詳細は、Javadoc を参照されたい。

以下のように、複数の結果メッセージを設定することもできる。

```
ResultMessages messages = ResultMessages.error()
    .add("e.ex.an.9001")
    .add("e.ex.an.8001", 1024);
model.addAttribute(messages);
```

この場合は、次のような HTML が出力される (JSP の変更は、不要である)。

```
<div class="alert alert-error">
  <ul>
    <li>There are inconsistencies in the data.</li>
    <li>Cannot upload, Because the file size must be less than 1,024MB.</li>
  </ul>
</div>
```

info メッセージを表示したい場合は、次のように ResultMessages.info() メソッドで ResultMessages オブジェクトを作成すればよい。

```
ResultMessages messages = ResultMessages.info().add("i.ex.an.0001", "XXXX");
model.addAttribute(messages);
```

以下のような HTML が、出力される。

```
<div class="alert alert-info"><!-- (1) -->
  <ul>
    <li>XXXX upload completed.</li>
  </ul>
</div>
```

項番	説明
(1)	メッセージタイプに対応して、出力される class 名が」alert alert-**info**」に変わっている。

標準では、以下のメッセージタイプが用意されている。

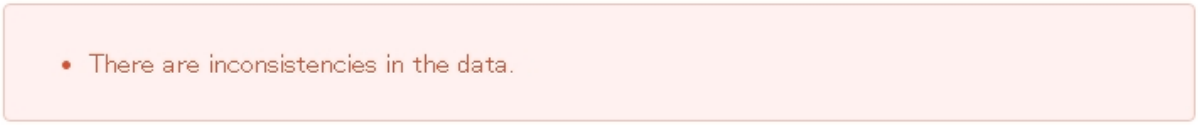
メッセージタイプ	ResultMessages オブジェクトの作成	デフォルトで出力される class 名	備考
success	`ResultMessages.success()`	alert alert-success	-
info	`ResultMessages.info()`	alert alert-info	-
warn	`ResultMessages.warn()`	alert alert-warn	メッセージタイプ「warning」の追加に伴い、terasoluna-gfw-common 5.0.0.RELEASE から非推奨。 このメッセージタイプは将来削除される可能性がある。
warning	`ResultMessages.warning()`	alert alert-warning	CSS フレームワークである Bootstrap の Alerts コンポーネントで用意されているメッセージタイプをデフォルトでサポートするために、terasoluna-gfw-common 5.0.0.RELEASE から追加。
error	`ResultMessages.error()`	alert alert-error	-
danger	`ResultMessages.danger()`	alert alert-danger	-

メッセージタイプに応じて CSS を定義されたい。以下に、CSS を適用した場合の例を示す。

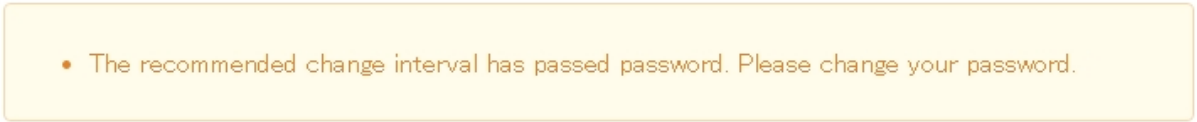
```
.alert {
  margin-bottom: 15px;
  padding: 10px;
  border: 1px solid;
  border-radius: 4px;
```

```
  text-shadow: 0 1px 0 #ffffff;
}
.alert-info {
  background: #ebf7fd;
  color: #2d7091;
  border-color: rgba(45, 112, 145, 0.3);
}
.alert-warning {
  background: #fffceb;
  color: #e28327;
  border-color: rgba(226, 131, 39, 0.3);
}
.alert-error {
  background: #fff1f0;
  color: #d85030;
  border-color: rgba(216, 80, 48, 0.3);
}
```

- `ResultMessages.error().add("e.ex.an.9001")` を`<t:messagesPanel />`で出力した例



- `ResultMessages.warning().add("w.ex.an.2001")` を`<t:messagesPanel />`で出力した例



- `ResultMessages.info().add("i.ex.an.0001", "XXXX")` を`<t:messagesPanel />`で出力した例



注釈: success と danger は、スタイルに多様性を持たせるために用意されている。本ガイドラインでは、success と info、error と danger は同義である。

ちなみに: CSS フレームワークである Bootstrap の Alerts コンポーネントは、`<t:messagesPanel />`のデフォルト設定で利用できる。

警告: 本例では、メッセージキーをハードコードで設定している。しかしながら、保守性を高めるためにも、メッセージキーは、定数クラスにまとめることを推奨する。

メッセージキー定数クラスの自動生成ツールを参照されたい。

結果メッセージの属性名指定

`ResultMessages` を Model に追加する場合、基本的には属性名を省略できる。

ただし、`ResultMessages` は一つのメッセージタイプしか表現できない。

1 画面に異なるメッセージタイプの `ResultMessages` を**同時に**表示したい場合は、明示的に属性名を指定して Model に設定する必要がある。

- Controller (MessageController に追加)

```
@RequestMapping(value = "showMessages", method = RequestMethod.GET)
public String showMessages(Model model) {

    model.addAttribute("messages1",
                ResultMessages.warning().add("w.ex.an.2001")); // (1)
    model.addAttribute("messages2",
                ResultMessages.error().add("e.ex.an.9001")); // (2)

    return "message/showMessages";
}
```

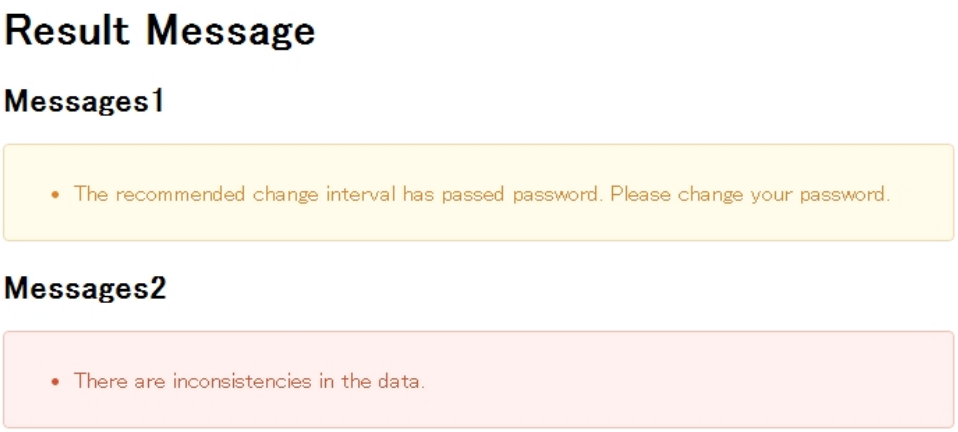
項番	説明
(1)	メッセージタイプが」warning」である、`ResultMessages` を属性名」messages1」で Model に追加する。
(2)	メッセージタイプが」info」である、`ResultMessages` を属性名」messages2」で Model に追加する。

- JSP (WEB-INF/views/message/showMessages.jsp)

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Result Message Example</title>
<style type="text/css">
.alert {
    margin-bottom: 15px;
    padding: 10px;
    border: 1px solid;
    border-radius: 4px;
    text-shadow: 0 1px 0 #ffffff;
}

.alert-info {
    background: #ebf7fd;
    color: #2d7091;
    border-color: rgba(45, 112, 145, 0.3);
}

.alert-warning {
    background: #fffceb;
    color: #e28327;
    border-color: rgba(226, 131, 39, 0.3);
}

.alert-error {
    background: #fff1f0;
    color: #d85030;
    border-color: rgba(216, 80, 48, 0.3);
}
</style>
</head>
<body>
    <h1>Result Message</h1>
    <h2>Messages1</h2>
    <t:messagesPanel messagesAttributeName="messages1" /><!-- (1) -->
    <h2>Messages2</h2>
    <t:messagesPanel messagesAttributeName="messages2" /><!-- (2) -->
</body>
</html>
```

項番	説明
(1)	属性名が」messages1」である `ResultMessages` を表示する。
(2)	属性名が」messages2」である `ResultMessages` を表示する。

ブラウザで表示すると、以下のように出力される。



業務例外メッセージの表示

`org.terasoluna.gfw.common.exception.BusinessException` と `org.terasoluna.gfw.common.exception.ResourceNotFoundException` は内部で `ResultMessages` を保持している。

業務例外メッセージを表示する場合は、Service クラスで `ResultMessages` を設定した `BusinessException` をスローすること。
Controller クラスでは `BusinessException` をキャッチし、例外中の結果メッセージを Model に追加する。

- Service クラス

```
@Service
@Transactional
public class UserServiceImpl implements UserService {
    // omitted

    public void create(...) {

        // omitted...

        if (...) {
            // illegal state!
            ResultMessages messages = ResultMessages.error()
                                                    .add("e.ex.an.9001"); /
↪/ (1)
            throw new BusinessException(messages);
        }
    }

}
```

項番	説明
(1)	エラーメッセージを ResultMessages で作成し、BusinessException に設定する。

- Controller クラス

```
@RequestMapping(value = "create", method = RequestMethod.POST)
public String create(@Validated UserForm form, BindingResult result, Model
↪model) {
    // omitted

    try {
        userService.create(user);
    } catch (BusinessException e) {
        ResultMessages messages = e.getResultMessages(); // (1)
        model.addAttribute(messages);

        return "user/createForm";
    }

    // omitted
}
```

項番	説明
(1)	BusinessException が保持する ResultMessages を取得し、Model に追加する。

通常、エラーメッセージ表示する場合は、Controller で ResultMessages オブジェクトを作成するのではなく、こちらの方法を使用する。

4.6.3 How to extend

独自メッセージタイプを作成する

メッセージタイプを追加したい場合の、独自メッセージタイプ作成方法について説明する。

通常は、用意されているメッセージタイプのみで十分であるが、採用している CSS ライブラリによってはメッセージタイプを追加したい場合がある。例えば」notice」というメッセージタイプを追加する場合を説明する。

まず、以下のように org.terasoluna.gfw.common.message.ResultMessageType インタフェースを実装した
独自メッセージタイプクラスを作成する。

```
import org.terasoluna.gfw.common.message.ResultMessageType;

public enum ResultMessageTypes implements ResultMessageType { // (1)
    NOTICE("notice");

    private ResultMessageTypes(String type) {
        this.type = type;
    }

    private final String type;

    @Override
    public String getType() { // (2)
        return this.type;
    }

    @Override
    public String toString() {
        return this.type;
    }
}
```

項番	説明
(1)	ResultMessageType インタフェースを実装した Enum を定義する。定数オブジェクトで作成してもよいが、Enum で作成することを推奨する。
(2)	getType の返り値が出力される CSS の class 名に対応する。

このメッセージタイプを使用して以下のように ResultMessages を作成する。

```
ResultMessages messages = new ResultMessages(ResultMessageTypes.NOTICE) // (1)
        .add("w.ex.an.2001");
model.addAttribute(messages);
```

項番	説明
(1)	ResultMessages のコンストラクタに対象の ResultMessageType を指定する。

この場合、<t:messagesPanel /> で以下のような HTML が出力される。

```
<div class="alert alert-notice">
  <ul>
    <li>The recommended change interval has passed password. Please change your␣
↪password.</li>
  </ul>
</div>
```

> **ちなみに:** 拡張方法は、org.terasoluna.gfw.common.message.StandardResultMessageType が参考になる。

4.6.4 Appendix

<t:messagesPanel>タグの属性変更

<t:messagesPanel>タグには、表示形式を変更する属性がいくつか用意されている。

表 4.41 <t:messagesPanel>タグ 属性一覧

オプション	内容	default の設定値
panelElement	結果メッセージ表示パネルの要素	div
panelClassName	結果メッセージ表示パネルの CSS class 名。	alert
panelTypeClassPrefix	CSS class 名の接頭辞	alert-
messagesType	メッセージタイプ。この属性が設定された場合。設定されたメッセージタイプが ResultMessages がもつメッセージタイプより優先されて使用される。	
outerElement	結果メッセージ一覧を構成する HTML の外側のタグ	ul
innerElement	結果メッセージ一覧を構成する HTML の内側のタグ	li
disableHtmlEscape	HTML エスケープ処理を無効化するためのフラグ。 true を指定する事で、出力するメッセージに対して HTML エスケープ処理が行われなくなる。 この属性は、出力するメッセージに HTML を埋め込むことで、メッセージの装飾などができるようにするために用意している。 **true を指定する場合は、XSS 対策が必要な文字がメッセージ内に含まれない事が保証されていること。** terasoluna-gfw-web 1.0.1.RELEASE 以上で利用可能な属性である。	false

例えば、CSS フレームワーク」BlueTrip」では以下のような CSS が用意されている。

```
.error,.notice,.success {
    padding: .8em;
    margin-bottom: 1.6em;
    border: 2px solid #ddd;
}

.error {
    background: #FBE3E4;
    color: #8a1f11;
    border-color: #FBC2C4;
}

.notice {
    background: #FFF6BF;
    color: #514721;
    border-color: #FFD324;
}

.success {
```

```
    background: #E6EFC2;
    color: #264409;
    border-color: #C6D880;
}
```

この CSS を使用したい場合、 ... というようにメッセージが出力されてほしい。

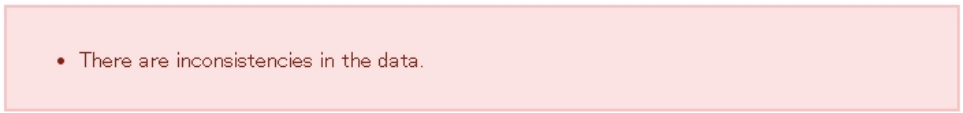
この場合、<t:messagesPanel>タグを以下のように使用すればよい (Controller は修正不要である)。

```
<t:messagesPanel panelClassName="" panelTypeClassPrefix="" />
```

出力される HTML は以下のようになる。

```
<div class="error">
    <ul>
        <li>There are inconsistencies in the data.</li>
    </ul>
</div>
```

ブラウザで表示すると、以下のように出力される。



メッセージ一覧を表示するために<ul>タグを使用したくない場合は、outerElement 属性と innerElement 属性を使用することでカスタマイズできる。

以下のように属性を設定した場合は、

```
<t:messagesPanel outerElement="" innerElement="span" />
```

次のように HTML が出力される。

```
<div class="alert alert-error">
    <span>There are inconsistencies in the data.</span>
    <span>Cannot upload, Because the file size must be less than 1,024MB.</span>
</div>
```

以下のよう CSS を設定することで、

```
.alert > span {
    display: block; /* (1) */
}
```

項番	説明
(1)	「alert」クラスの要素の子となるタグをブロックレベル要素にする。

ブラウザで次のように表示される。

There are inconsistencies in the data.
Cannot upload, Because the file size must be less than 1,024MB.

disableHtmlEscape 属性を true にした場合、以下のような出力イメージにする事ができる。
下記の例では、メッセージの一部のフォントを「16px の赤字」に装飾している。

- jsp

```
<spring:message var="informationMessage" code="i.ex.od.0001" />
<t:messagesPanel messagesAttributeName="informationMessage"
    messagesType="alert alert-info"
    disableHtmlEscape="true" />
```

- properties

```
i.ex.od.0001 = Please confirm order content. <font style="color: red; font-
↪size: 16px;">If this orders submitted, cannot cancel.</font>
```

- 出力イメージ

- Please confirm order content. If this orders submitted, cannot cancel.

disableHtmlEscape 属性が false(デフォルト) の場合は、HTML エスケープされて以下のような出力となる。

- Please confirm order content. If this orders submitted, cannot cancel.

ResultMessages を使用しない結果メッセージの表示

<t:messagesPanel>タグは ResultMessages オブジェクト以外にも

- java.lang.String
- java.lang.Exception
- java.util.List

オブジェクトも出力できる。

通常は<t:messagesPanel>タグは ResultMessages オブジェクトの出力用に使用するが、
フレームワークがリクエストスコープに設定した文字列 (エラーメッセージなど) を表示する場合にも使用できる。

例えば、Spring Security のデフォルトの設定で使用される、org.springframework.security.web.authentication.SimpleUrlAuthenticationFailureHandler では、認証エラー時に発生した例外オブジェクトを SPRING_SECURITY_LAST_EXCEPTION という属性名で、リダイレクト時はセッション、フォワード時はリクエストスコープに格納する。
この例外メッセージを、結果メッセージ同様に<t:messagesPanel>タグで出力したい場合は、以下のように設定すればよい。

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Login</title>
<style type="text/css">
/* (1) */
.alert {
    margin-bottom: 15px;
    padding: 10px;
    border: 1px solid;
    border-radius: 4px;
    text-shadow: 0 1px 0 #ffffff;
}

.alert-error {
    background: #fff1f0;
    color: #d85030;
    border-color: rgba(216, 80, 48, 0.3);
}
</style>
</head>
<body>
    <c:if test="${param.containsKey('error')}">
        <t:messagesPanel messagesType="error"
            messagesAttributeName="SPRING_SECURITY_LAST_EXCEPTION" /><!-- (2) -->
    </c:if>
    <form:form
        action="${pageContext.request.contextPath}/authentication"
        method="post">
        <fieldset>
            <legend>Login Form</legend>
            <div>
```

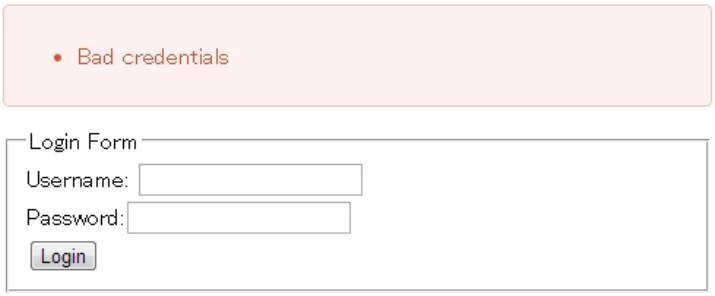
```
                <label for="username">Username: </label><input
                    type="text" id="username" name="username">
            </div>
            <div>
                <label for="username">Password:</label><input
                    type="password" id="password" name="password">
            </div>
            <div>
                <input type="submit" value="Login" />
            </div>
        </fieldset>
    </form:form>
</body>
</html>
```

項番	説明
(1)	結果メッセージ表示用の CSS を再掲する。実際は CSS ファイルに記述することを強く推奨する。
(2)	`Exception` オブジェクトが格納されている属性名を `messagesAttributeName` 属性で指定する。 また、`ResultMessages` オブジェクトとは異なり、メッセージタイプの情報をもたないため、`messagesType` 属性で、明示的に、メッセージタイプを指定する必要がある。

認証エラー時に出力される HTML は

```
<div class="alert alert-error"><ul><li>Bad credentials</li></ul></div>
```

であり、ブラウザでは以下のように出力される。



ちなみに: ログイン用の JSP の内容については、認証を参照されたい。

メッセージキー定数クラスの自動生成ツール

これまでの例ではメッセージキーを文字列のハードコードで設定していたが、
メッセージキーは定数クラスにまとめることを推奨する。

ここでは、簡易ツールとして、properties ファイルからメッセージキー定数クラスを
自動生成するプログラムおよび使用方法を紹介する。必要に応じてカスタマイズして利用されたい。

1. メッセージキー定数クラスの作成

   まず空のメッセージキー定数クラスを作成する。ここでは com.example.common.message.MessageKeys とする。

```
package com.example.common.message;

public class MessageKeys {

}
```

2. 自動生成クラスの作成

   次に MessageKeys クラスと同じパッケージに MessageKeysGen クラスを作成し、以下のように記述する。

```
package com.example.common.message;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.regex.Pattern;

import org.apache.commons.io.FileUtils;
import org.apache.commons.io.IOUtils;

public class MessageKeysGen {
    public static void main(String[] args) throws IOException {
        // message properties file
        InputStream inputStream = new FileInputStream("src/main/resources/
↪i18n/application-messages.properties");
        BufferedReader br = new BufferedReader(new␣
↪InputStreamReader(inputStream));
        Class<?> targetClazz = MessageKeys.class;
        File output = new File("src/main/java/"
                + targetClazz.getName().replaceAll(Pattern.quote("."), "/")
                + ".java");
        System.out.println("write " + output.getAbsolutePath());
        PrintWriter pw = new PrintWriter(FileUtils.
↪openOutputStream(output));

        try {
            pw.println("package " + targetClazz.getPackage().getName() + ";
↪");
            pw.println("/**");
            pw.println(" * Message Id");
            pw.println(" */");
            pw.println("public class " + targetClazz.getSimpleName() + " {
↪");
```

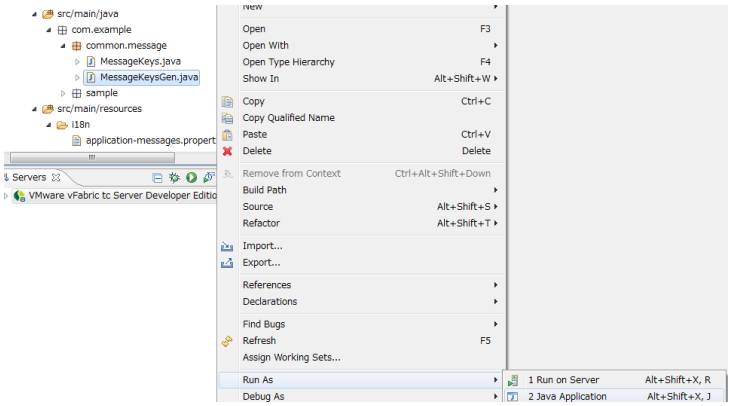
```
            String line;
            while ((line = br.readLine()) != null) {
                String[] vals = line.split("=", 2);
                if (vals.length > 1) {
                    String key = vals[0].trim();
                    String value = vals[1].trim();
                    pw.println("    /** " + key + "=" + value + " */");
                    pw.println("    public static final String "
                            + key.toUpperCase().replaceAll(Pattern.quote(".
↪"),
                                    "_").replaceAll(Pattern.quote("-"), "_
↪")
                            + " = \"" + key + "\";");
                }
            }
            pw.println("}");
            pw.flush();
        } finally {
            IOUtils.closeQuietly(br);
            IOUtils.closeQuietly(pw);
        }
    }
}
```

3. メッセージプロパティファイルの用意

   src/main/resource/i18m/application-messages.properties にメッセージを定義する。ここでは例として、以下のように設定する。

```
i.ex.an.0001={0} upload completed.
w.ex.an.2001=The recommended change interval has passed password. Please␣
↪change your password.
e.ex.an.8001=Cannot upload, Because the file size must be less than {0}MB.
e.ex.an.9001=There are inconsistencies in the data.
```

4. 自動生成クラスの実行



   MessageKeys クラスが、以下のように上書きされる。

```
package com.example.common.message;
/**
 * Message Id
 */
public class MessageKeys {
    /** i.ex.an.0001={0} upload completed. */
    public static final String I_EX_AN_0001 = "i.ex.an.0001";
    /** w.ex.an.2001=The recommended change interval has passed password.␣
↪Please change your password. */
    public static final String W_EX_AN_2001 = "w.ex.an.2001";
    /** e.ex.an.8001=Cannot upload, Because the file size must be less␣
↪than {0}MB. */
    public static final String E_EX_AN_8001 = "e.ex.an.8001";
    /** e.ex.an.9001=There are inconsistencies in the data. */
    public static final String E_EX_AN_9001 = "e.ex.an.9001";
}
```

4.7 国際化

4.7.1 Overview

国際化とは、アプリケーションで表示するラベルやメッセージを、特定の言語のみに固定せず、ロケール (Locale) と呼ばれる言語や国・地域を表す単位の指定により、複数言語の切り替えに対応させることである。

本節では、画面に表示するメッセージを国際化する方法について説明する。

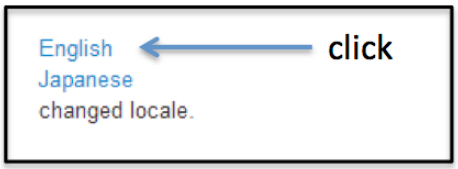
国際化するためには、以下の対応が必要となる。

- 画面内のテキスト要素（コード値の名称、メッセージ、GUI コンポーネントのラベルなど）は、プログラム内でハードコードせずに、プロパティファイルなどの外部定義から取得する。
- クライアントから Locale を指定する仕組みを提供する。

クライアントから Locale を指定する方法は以下の通りである。

- 標準のリクエストヘッダを使用する。(ブラウザの言語設定で指定)
- リクエストパラメータを使用して Cookie に保存する。
- リクエストパラメータを使用して Session に保存する。

Locale の切り替えイメージを以下に示す。





注釈: Codelist の国際化方法については、 コードリスト を参照されたい。

注釈: エラー画面を国際化する必要がある場合、Spring MVC の Controller を使用してエラー画面に遷移すること。Spring MVC を介さずエラー画面に直接遷移した場合、メッセージが意図した言語で出力されない場合がある。

詳細については国際化が適用されない場合の対処方法を参照されたい。

ちなみに: 国際化は i18n という略称が広く知られている。i18n という記述は、internationalization の先頭の i と語尾の n の間に nternationalizatio の 18 文字があることに起因する。

4.7.2 How to use

メッセージ定義の設定

画面に表示するメッセージを国際化する場合は、メッセージを管理するためのコンポーネント(MessageSource)として、

- org.springframework.context.support.ResourceBundleMessageSource
- org.springframework.context.support.ReloadableResourceBundleMessageSource

のどちらかを使用する。

ここでは、ResourceBundleMessageSource を使用する場合の設定例を紹介する。

applicationContext.xml

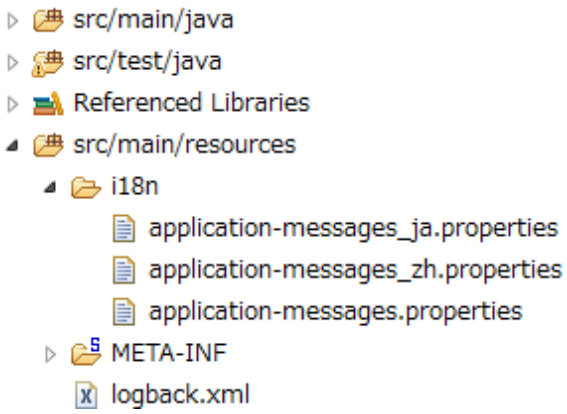
```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>i18n/application-messages</value>  <!-- (1) -->
        </list>
    </property>
</bean>
```

項番	説明
(1)	プロパティファイルの基底名として、i18n/application-messages を指定する。 国際化対応を行う場合、i18n ディレクトリ配下にメッセージプロパティファイルを格納することを推奨する。 MessageSource の詳細や定義方法は、 メッセージ管理 を参照されたい。

プロパティファイルの格納例

プロパティファイルは、以下のルールに則って作成する。

- Locale 毎のファイル名は、application-messages_XX.properties という形式で作成する。(XX 部分は Locale を指定)



- `application-messages.properties` は **必ず作成する** 。もし存在しない場合、`MessageSource` からメッセージを取得できず、JSP にメッセージを設定する際に、`JspTagException` が発生する。
- `application-messages.properties` に定義するメッセージは、デフォルトで使用する言語で作成する。

上記ルールに則ってプロパティファイルを作成すると、以下のような動作になる。

- クライアントの Locale が zh の場合、`application-messages_zh.properties` が使用される。
- クライアントの Locale が ja の場合、`application-messages_ja.properties` が使用される。
- クライアントの Locale に対応するプロパティファイルが存在しない場合、デフォルトとして、`application-messages.properties` が使用される。(ファイル名に」_XX」部分が存在しないファイル)

注釈: Locale の判別方法は、以下の順番で該当する Locale のプロパティファイルが発見されるまで、Locale を確認していくことである。

1. クライアントから指定された Locale
2. アプリケーションサーバの JVM に指定されている Locale(設定されていない場合あり)
3. アプリケーションサーバの OS に指定されている Locale

よく間違える例として、 クライアントから指定された Locale のプロパティファイルが存在しない場合、デフォルトのプロパティファイルが使用されるとの誤解が挙げられる。実際は、次にアプリケーションサーバに指定されている Locale を確認して、それでも該当する Locale のプロパティファイルが見つからない場合に、デフォルトのプロパティファイルが使用されるので注意する。

ちなみに: メッセージプロパティファイルの記載方法については、 メッセージ管理 を参照されたい。

Locale をブラウザの設定により切り替える

AcceptHeaderLocaleResolver の設定

ブラウザの設定を使用して Locale を切り替える場合は、`AcceptHeaderLocaleResolver` を使用する。

spring-mvc.xml

```
<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver"> <!--
↪(1) -->
    <property name="defaultLocale" value="en" /> <!-- (2) -->
    <property name="supportedLocales"> <!-- (3) -->
        <list value-type="java.util.Locale">
            <value>en</value>
            <value>ja</value>
        </list>
    </property>
</bean>
```

項番	説明
(1)	bean タグの id 属性」localeResolver」に `org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver` を指定する。 この `LocaleResolver` を使用すると、リクエスト毎に設定される HTTP ヘッダー (”accept-language”) に指定されている Locale が使用される。
(2)	デフォルトの Locale を設定するため、`defaultLocale` プロパティを設定する。 上記の例では、アプリケーションがサポートしていない Locale がリクエストされた場合、Locale は」en」に設定される。
(3)	アプリケーションがサポートする Locale を設定するため、`supportedLocales` プロパティを設定する。 リクエストが要求する Locale と、サポートする Locale が一致する場合はその Locale が使用される。 国と言語を組み合わせた Locale（例：ja_JP）が要求され、サポートする Locale と一致しない場合、対応する言語のみの Locale（例：ja）で一致を確認する。 上記の例では、アプリケーションがサポートする Locale として、」en」と」ja」を指定している。

注釈: `LocaleResolver` が設定されていない場合、デフォルトで `org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver` が使用されるため、`LocaleResolver` の設定は、省略することもできる。

メッセージの設定

以下に、メッセージの設定例を示す。

application-messages.properties

```
title.admin.top = Admin Top
```

application-messages_ja.properties

```
title.admin.top = 管理画面 Top
```

JSP の実装

以下に、JSP の実装例を示す。

include.jsp(インクルード用の共通 jsp ファイル)

```
<%@ page session="false"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>  <!-- (1) --
↪>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec"%>
<%@ taglib uri="http://terasoluna.org/functions" prefix="f"%>
<%@ taglib uri="http://terasoluna.org/tags" prefix="t"%>
```

項番	説明
(1)	JSP で出力する場合、Spring のタグライブラリを用いてメッセージ出力を行うため、カスタムタグを定義する必要がある。 <%@taglib uri="http://www.springframework.org/tags" prefix="spring"%> を定義すること。

注釈: インクルード用の共通 jsp ファイルの詳細は インクルード用の共通 *JSP* の作成 を参照されたい。

画面表示用 JSP ファイル

```
<spring:message code="title.admin.top" />  <!-- (2) -->
```

項番	説明
(2)	JSP では、Spring のタグライブラリである、 <spring:message> を用いてメッセージ出力を行う。 code 属性に、プロパティで指定したキーを設定する。 本例では、Locale が、ja の場合、」管理画面 Top」、それ以外の Locale の場合、」Admin Top」が出力される。

Locale を画面操作等で動的に変更する

Locale を画面操作等で動的に変更する方法は、ユーザ端末（ブラウザ）の設定に関係なく、特定の言語を選択させたい場合に有効である。

画面操作で Locale を変更する場合のイメージを以下に示す。

ユーザが使用する言語を選択する場合は、org.springframework.web.servlet.i18n.LocaleChangeInterceptor を用いることで実現する事ができる。

LocaleChangeInterceptor は、リクエストパラメータに指定された Locale の値を、org.springframework.web.servlet.LocaleResolver の API を使用してサーバ又はクライアントに保存するためのインタセプターである。

使用する LocaleResolver の実装クラスを、以下の表から選択する。

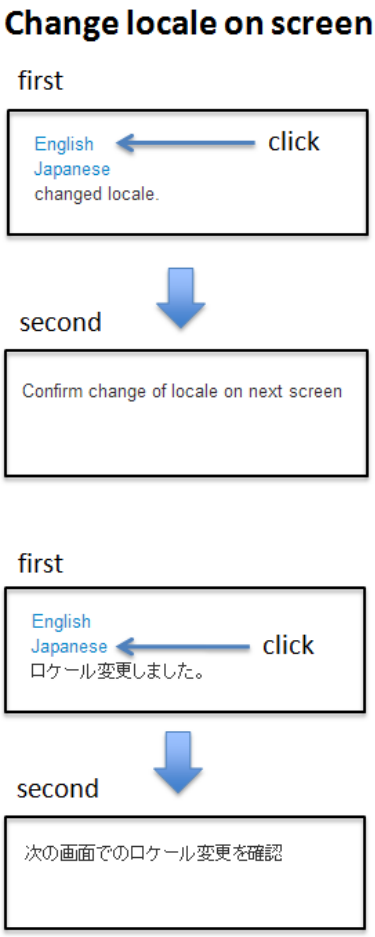


表 4.42　**LocaleResolver** の種類

No	実装クラス	Locale の保存方法
1.	`org.springframework.web.servlet.i18n.SessionLocaleResolver`	サーバーに保存 (`HttpSession` を使用)
2.	`org.springframework.web.servlet.i18n.CookieLocaleResolver`	クライアントに保存 (`Cookie` を使用)

注　釈: `LocaleResolver` に `org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver` を使用する場合、`org.springframework.web.servlet.i18n.LocaleChangeInterceptor` を使用して Locale を動的に変更することはできない。

LocaleChangeInterceptor の設定

リクエストパラメータを使用して Locale を切り替える場合は、`LocaleChangeInterceptor` を使用する。

spring-mvc.xml

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**" />
    <mvc:exclude-mapping path="/resources/**" />
    <bean
      class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">  <!--␣
↪(1) -->
    </bean>
    <!-- omitted -->
  </mvc:interceptor>
</mvc:interceptors>
```

項番	説明
(1)	Spring MVC のインタセプターに、`org.springframework.web.servlet.i18n.LocaleChangeInterceptor` を定義する。 この設定により、」リクエスト URL?locale=xx」で 使用可能 となる。

注釈: **Locale** を指定するリクエストパラメータ名の変更方法

```
<bean
    class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="lang"/>  <!-- (2) -->
</bean>
```

項番	説明
(2)	`paramName` プロパティにリクエストパラメータ名を指定する。上記例では、」リクエスト URL?lang=xx」となる。

注釈: リクエストパラメータに Locale として不正な値が指定された場合は例外がスローされる。例外とせず、リクエストパラメータによる Locale の設定をスキップしたい場合は `ignoreInvalidLocale` プロパティに `true` を指定すれば良い。これにより Locale の設定がスキップされた場合は、`LocaleResolver` で指定されたデフォルトのロケールが使用される。

なお、`LocaleChangeInterceptor` には `AcceptHeaderLocaleResolver` の `supportedLocales` のように、サポートする Locale を限定する仕組みはない。

SessionLocaleResolver の設定

Locale をサーバに保存する場合は、`SessionLocaleResolver` を使用する。

spring-mvc.xml

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.
↪SessionLocaleResolver">  <!-- (1) -->
    <property name="defaultLocale" value="en"/>  <!-- (2) -->
</bean>
```

項番	説明
(1)	bean タグの id 属性を」localeResolver」で定義し、`org.springframework.web.servlet.LocaleResolver` を実装したクラスを指定する。 本例では、セッションに Locale を保存する `org.springframework.web.servlet.i18n.SessionLocaleResolver` を指定している。 bean タグの id 属性は」localeResolver」と設定すること。 この設定により、 `LocaleChangeInterceptor` 内の処理で `SessionLocaleResolver` が使用される。
(2)	`defaultLocale` プロパティに Locale を指定する。セッションから Locale が取得できない場合、`value` の設定値が有効になる。 **注釈:** `defaultLocale` プロパティを省略した場合、ユーザ端末（ブラウザ）に設定された Locale が有効になる。

CookieLocaleResolver の設定

Locale をクライアントに保存する場合は、`CookieLocaleResolver` を使用する。

spring-mvc.xml

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.
↪CookieLocaleResolver">  <!-- (1) -->
      <property name="defaultLocale" value="en"/>  <!-- (2) -->
      <property name="cookieName" value="localeCookie"/>  <!-- (3) -->
</bean>
```

項番	説明
(1)	bean タグの id 属性」localeResolver」に `org.springframework.web.servlet.i18n.CookieLocaleResolver` を指定する。 bean タグの id 属性は」localeResolver」と設定すること。 この設定により、 `LocaleChangeInterceptor` 内の処理で `CookieLocaleResolver` が使用される。
(2)	`defaultLocale` プロパティに Locale を指定する。Cookie から Locale が取得できない場合、`value` の設定値が有効になる。 **注釈:** `defaultLocale` プロパティを省略した場合、ユーザ端末（ブラウザ）に設定された Locale が有効になる。
(3)	`cookieName` プロパティに指定した値が、cookie 名となる。指定しない場合、`org.springframework.web.servlet.i18n.CookieLocaleResolver.LOCALE` となる。**Spring Framework** を使用していることがわかるため、変更することを推奨する。

メッセージの設定

以下に、メッセージの設定例を示す。

application-messages.properties

```
i.xx.yy.0001 = changed locale
i.xx.yy.0002 = Confirm change of locale at next screen
```

application-messages_ja.properties

```
i.xx.yy.0001 = Locale を変更しました。
i.xx.yy.0002 = 次の画面での Locale 変更を確認
```

JSP の実装

以下に、JSP の実装例を示す。

画面表示用 JSP ファイル

```
<a href='${pageContext.request.contextPath}?locale=en'>English</a>  <!-- (1) -->
<a href='${pageContext.request.contextPath}?locale=ja'>Japanese</a>
<spring:message code="i.xx.yy.0001" />
```

項番	説明
(1)	Locale を切り替えるためのパラメータを送信する。 リクエストパラメータ名は、`LocaleChangeInterceptor` の `paramName` プロパティに指定した値となる。（上記例では、デフォルトのパラメータ名を使用している） 上記例の場合、English リンクで英語 Locale、Japanese リンクで日本語 Locale に変更している。 以降は、選択した Locale が有効になる。 英語 Locale は」en」用のプロパティファイルが存在しないため、デフォルトのプロパティファイルから読み込まれる。

ちなみに:

- インクルード用の共通 jsp に Spring のタグライブラリを定義する必要がある。
- インクルード用の共通 jsp ファイルの詳細は *インクルード用の共通 JSP の作成* を参照されたい。

国際化が適用されない場合の対処方法

`LocaleChangeInterceptor` は Spring MVC の `Controller` の処理実行時に呼ばれるインターセプタであるため、`Controller` を経由しない遷移の場合は国際化が適用されないことに注意されたい。

例えば、エラー画面への遷移設定に直接 JSP ファイルを指定するような場合、エラー画面への遷移には `Controller` が使用されない。この場合、エラー画面を国際化するには、エラー画面へ遷移するための

Controller を作成し、エラー画面への遷移に使用することで LocaleChangeInterceptor が使用されるように設定する必要がある。

注釈: 同様に、JSP を直接指定した遷移の場合 *Tiles による画面レイアウト*で使用する ViewResolver を経由しないため Tiles が適用されない。

設定方法について、Spring Security の認可の実装を例に以下に示す。

LocaleChangeInterceptor が適用されないエラー画面への遷移例

- spring-security.xml

```
<sec:http>
    <!-- omitted -->
    <sec:access-denied-handler
        error-page="/WEB-INF/views/common/error/accessDeniedError.jsp" /> <!-- (1)
↪-->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	<sec:access-denied-handler>タグの error-page 属性に認可エラー用のエラー画面を JSP で指定する。

LocaleChangeInterceptor が適用されるエラー画面への遷移例

- spring-security.xml

```
<sec:http>
    <!-- omitted -->
    <sec:access-denied-handler
        error-page="/common/error/accessDeniedError" /> <!-- (1) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	<sec:access-denied-handler>タグの error-page 属性に認可エラー用のエラー画面へ遷移するためのパスを設定する。

- Controller クラス

```
@Controller
@RequestMapping("/common/error") // (1)
public class ErrorController {

    @RequestMapping("accessDeniedError") // (1)
    public String accessDeniedError() {
        return "common/error/accessDeniedError"; // (2)
    }

}
```

項番	説明
(1)	エラー画面へ遷移するためにリクエストマッピングを定義する。
(2)	遷移するエラー画面の View 名を返却する。

警告: 一般的に、エラー画面には GET リクエストだけでなく POST リクエストからも遷移する可能性があるため、<mvc:view-controller>は使用しないことを推奨する。

<mvc:view-controller>使用時の留意点については *HTML を応答する*を参照されたい。

4.8 コードリスト

4.8.1 Overview

コードリストとは、「コード値 (value) とその表示名 (label)」の集合である。

画面のセレクトボックスなどコード値を画面で表示する際のラベルへのマッピング表として利用される。

共通ライブラリでは、

- xml ファイルや DB に定義されたコードリストをアプリケーション起動時に読み込みキャッシュする機能
- JSP や Java クラスからコードリストを参照する機能
- コードリストを用いて入力チェックする機能

を提供している。

また、応用的な使い方として、

- コードリストの国際化対応
- キャッシュされたコードリストのリロード

もサポートしている。

注釈: 標準でリロードが可能なのは、DB に定義されたコードリストを使用する場合のみである。

共通ライブラリでは、以下のコードリスト実装クラスを提供している。

表 4.44　コードリスト種類一覧

種類	内容	Reloadable
org.terasoluna.gfw.common.codelist.SimpleMapCodeList	xml ファイルに直接記述した内容を使用する。	NO
org.terasoluna.gfw.common.codelist.NumberRangeCodeList	数値の範囲のリストを作成する際に使用する。	NO
org.terasoluna.gfw.common.codelist.JdbcCodeList	DB から対象のコードを SQL で取得して使用する。	YES
org.terasoluna.gfw.common.codelist.EnumCodeList	Enum クラスに定義した定数からコードリストを作成する際に使用する。	NO
org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList	国際化に対応し、java.util.Locale に応じたコードリストを使用する。	NO
org.terasoluna.gfw.common.codelist.i18n.SimpleReloadableI18nCodeList	国際化に対応し、java.util.Locale に応じた更新可能なコードリストを使用する。(**5.4.2** から追加)	YES

上記コードリストのインタフェースについて、共通ライブラリに
org.terasoluna.gfw.common.codelist.CodeList を提供している。
また国際化に対応しているコードリストのインタフェースについて、
org.terasoluna.gfw.common.codelist.i18n.I18nCodeList を提供している。

共通ライブラリで提供しているコードリストのクラス図構成を以下に示す。

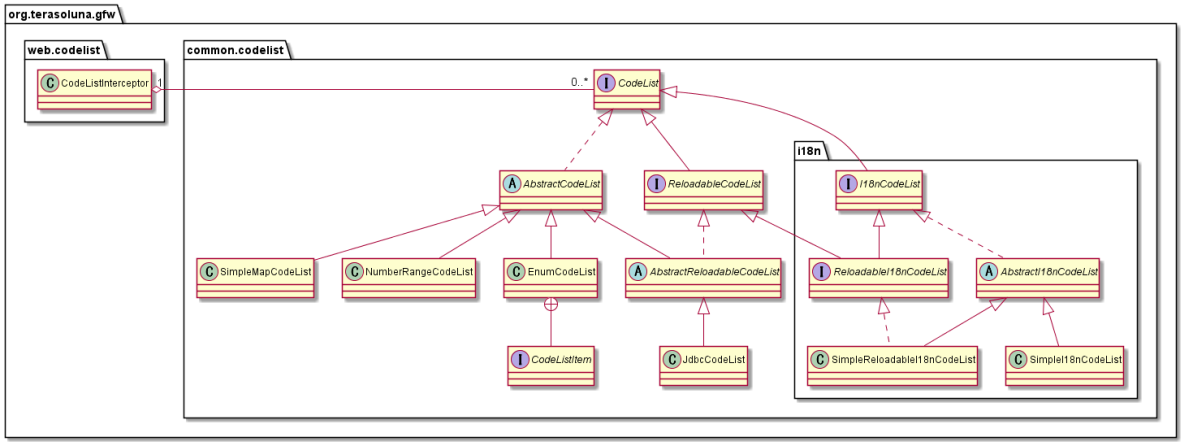


図 4.17　**Picture - Image of codelist class diagram**

4.8.2 How to use

本項では、各種コードリストを使用する上での設定や実装方法を記述する。

- *SimpleMapCodeList* の使用方法
- *NumberRangeCodeList* の使用方法
- *JdbcCodeList* の使用方法
- *EnumCodeList* の使用方法
- *I18nCodeList* の使用方法
- 特定のコード値からコード名を表示する
- コードリストを用いたコード値の入力チェック

SimpleMapCodeList の使用方法

`org.terasoluna.gfw.common.codelist.SimpleMapCodeList` とは、xml ファイルに定義したコード値をアプリケーション起動時に読み込み、そのまま使用するコードリストである。

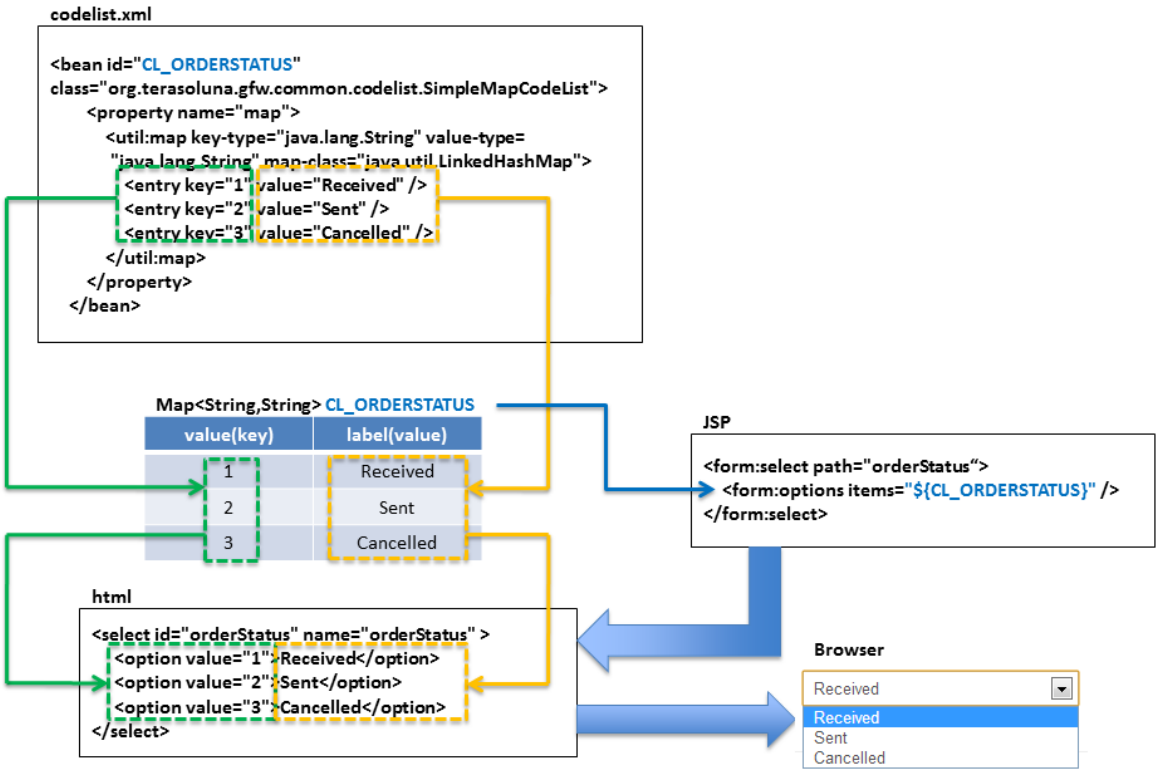
SimpleMapCodeList のイメージ

コードリスト設定例

bean 定義ファイル (xxx-codelist.xml) の定義

bean 定義ファイルは、コードリスト用に作成することを推奨する。

```
<bean id="CL_ORDERSTATUS" class="org.terasoluna.gfw.common.codelist.
↪SimpleMapCodeList"> <!-- (1) -->
    <property name="map">
        <util:map>
            <entry key="1" value="Received" /> <!-- (2) -->
            <entry key="2" value="Sent" />
            <entry key="3" value="Cancelled" />
        </util:map>
    </property>
</bean>
```



項番	説明
(1)	SimpleMapCodeList クラスを bean 定義する。 beanID は、後述する `org.terasoluna.gfw.web.codelist.CodeListInterceptor` の ID パターンに合致する名称にすること。
(2)	Map の Key、Value を定義する。 map-class 属性を省略した場合、 `java.util.LinkedHashMap` で登録されるため、上記例では、「名前と値」が、登録順に Map へ保持される。

bean 定義ファイル (xxx-domain.xml) の定義

コードリスト用 bean 定義ファイルを作成後、既存 bean 定義ファイルに import を行う必要がある。

```
<import resource="classpath:META-INF/spring/projectName-codelist.xml" /> <!-- (3) -->
<context:component-scan base-package="com.example.domain" />

<!-- omitted -->
```

項番	説明
(3)	コードリスト用 bean 定義ファイルを import する。 component-scan している間に import 先の情報が必要な場合があるため、 import は <context:component-scan base-package="com.example.domain" /> より上で設定する必要がある。

JSP でのコードリスト使用

共通ライブラリから提供しているインタセプターを用いることで、リクエストスコープに自動的に設定し、JSP からコードリストを容易に参照できる。

bean 定義ファイル (spring-mvc.xml) の定義

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**" /> <!-- (1) -->
    <bean
      class="org.terasoluna.gfw.web.codelist.CodeListInterceptor"> <!-- (2) -->
      <property name="codeListIdPattern" value="CL_.+" /> <!-- (3) -->
    </bean>
  </mvc:interceptor>

  <!-- omitted -->

</mvc:interceptors>
```

項番	説明
(1)	適用対象のパスを設定する。
(2)	CodeListInterceptor クラスを bean 定義する。
(3)	自動でリクエストスコープに設定する、コードリストの beanID のパターンを設定する。 パターンには `java.util.regex.Pattern` で使用する正規表現を設定すること。 上記例では、id が」CL_XXX」形式で定義されているデータのみを対象とする。その場合、id が」CL_」で始まらない bean 定義は取り込まれない。 「CL_」で定義した beanID は、リクエストスコープに設定されるため、JSP で容易に参照できる。 `codeListIdPattern` プロパティは省略可能である。 `codeListIdPattern` を省略した場合は、すべてのコードリスト (`org.terasoluna.gfw.common.codelist.CodeList` インタフェースを実装している bean) がリクエストスコープに設定される。

警告: 例外発生時のコードリスト利用について

terasoluna-gfw-common 5.4.2.RELEASE より、Controller のハンドラメソッドで例外が発生し `@ExceptionHandler` や `SystemExceptionResolver` で例外ハンドリングを行なった場合は、コードリストがリクエストスコープに登録されなくなった。これは、 `CodeListInterceptor` が `HandlerInterceptor#postHandle` メソッドでコードリストの登録を行うように変更されたためである。

例外時に遷移する画面でコードリストを利用したい場合は、ハンドラメソッドで例外を捕捉 (try-catch) するか、 *JSP から直接コードリスト Bean を参照する* を利用してコードリストを取得することを検討されたい。

例外ハンドリングの方法については、 例外のハンドリング方法 を参照されたい。

jsp の実装例

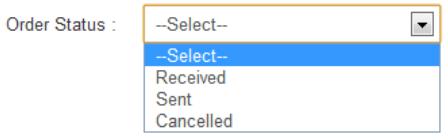
```
<form:select path="orderStatus">
  <form:option value="" label="--Select--" /> <!-- (4) -->
  <form:options items="${CL_ORDERSTATUS}" /> <!-- (5) -->
</form:select>
```

項番	説明
(4)	セレクトボックスの先頭にダミーの値を設定する場合、value に空文字を指定すること。
(5)	コードリストを定義した beanID を指定する。

出力 **HTML**

```
<select id="orderStatus" name="orderStatus">
   <option value="">--Select--</option>
   <option value="1">Received</option>
   <option value="2">Sent</option>
   <option value="3">Cancelled</option>
</select>
```

出力画面



Java クラスでのコードリスト使用

Java クラスでコードリストを利用する場合、 `javax.inject.Inject` アノテーションと、`javax.inject.Named` アノテーションを設定してコードリストをインジェクションする。`@Named` にコードリスト名を指定する。

```
import javax.inject.Named;

import org.terasoluna.gfw.common.codelist.CodeList;

@Service
public class OrderServiceImpl implements OrderService {
```

```
    @Inject
    @Named("CL_ORDERSTATUS")
    CodeList orderStatusCodeList; // (1)

    public boolean existOrderStatus(String target) {
        return orderStatusCodeList.asMap().containsKey(target); // (2)
    }
}
```

項番	説明
(1)	beanID が、」CL_ORDERSTATUS」であるコードリストをインジェクションする。
(2)	CodeList#asMap メソッドでコードリストを `java.util.Map` 形式で取得する。

NumberRangeCodeList の使用方法

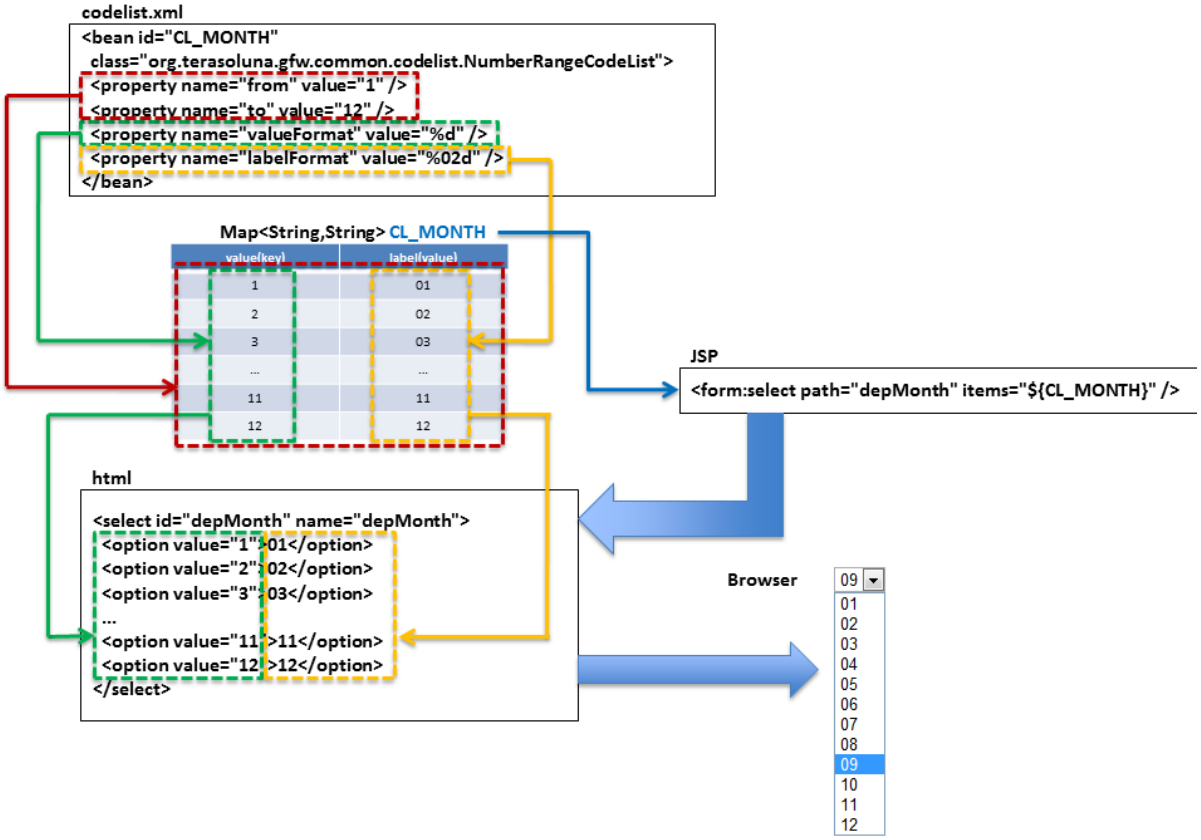
`org.terasoluna.gfw.common.codelist.NumberRangeCodeList` とは、アプリケーション起動時に、指定した数値の範囲をリストにするコードリストである。主に数だけのセレクトボックス、月や日付などのセレクトボックスに使用することを想定している。

NumberRangeCodeList のイメージ

ちなみに: NumberRangeCodeList はアラビア数字のみ対応しており、漢数字やローマ数字には対応していない。漢数字やローマ数字を表示したい場合は JdbcCodeList、SimpleMapCodeList に定義することで対応可能である。

NumberRangeCodeList には、以下の特徴がある。

1. From の値を To の値より小さくする場合、昇順に interval 分増加した値を From～To の範囲分リストにする。



2. To の値を From の値より小さくする場合、降順に interval 分減少した値を To～From の範囲分リストにする。

3. 増加分 (減少分) は interval を設定することで変更できる。

ここでは、昇順の `NumberRangeCodeList` について説明をする。降順の `NumberRangeCodeList` とインターバルの変更方法については、「*NumberRangeCodeList のバリエーション*」を参照されたい。

コードリスト設定例

From の値を To の値より小さくする (From < To) 場合の実装例を、以下に示す。

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_MONTH"
    class="org.terasoluna.gfw.common.codelist.NumberRangeCodeList"> <!-- (1) -->
    <property name="from" value="1" /> <!-- (2) -->
    <property name="to" value="12" /> <!-- (3) -->
    <property name="valueFormat" value="%d" /> <!-- (4) -->
    <property name="labelFormat" value="%02d" /> <!-- (5) -->
    <property name="interval" value="1" /> <!-- (6) -->
</bean>
```

項番	説明
(1)	NumberRangeCodeList を bean 定義する。
(2)	範囲開始の値を指定する。省略した場合、」0」が設定される。
(3)	範囲終了の値を設定する。指定必須。
(4)	コード値のフォーマット形式を設定する。フォーマット形式は `java.lang.String.format` の形式が使用される。 省略した場合、」%s」が設定される。
(5)	コード名のフォーマット形式を設定する。フォーマット形式は `java.lang.String.format` の形式が使用される。 省略した場合、」%s」が設定される。
(6)	増加する値を設定する。省略した場合、」1」が設定される。

JSP でのコードリスト使用

設定例の詳細は、前述した *JSP でのコードリスト使用* を参照されたい。

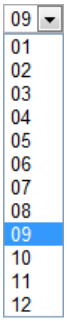
jsp の実装例

```
<form:select path="depMonth" items="${CL_MONTH}" />
```

出力 **HTML**

```
<select id="depMonth" name="depMonth">
  <option value="1">01</option>
  <option value="2">02</option>
  <option value="3">03</option>
  <option value="4">04</option>
  <option value="5">05</option>
  <option value="6">06</option>
  <option value="7">07</option>
  <option value="8">08</option>
  <option value="9">09</option>
  <option value="10">10</option>
  <option value="11">11</option>
  <option value="12">12</option>
</select>
```

出力画面



Java クラスでのコードリスト使用

設定例の詳細は、前述した *Java クラスでのコードリスト使用* を参照されたい。

JdbcCodeList の使用方法

`org.terasoluna.gfw.common.codelist.JdbcCodeList` とは、アプリケーション起動時に DB から値を取得し、コードリストを作成するクラスである。

JdbcCodeList はアプリケーション起動時にキャッシュを作るので、リスト表示時は DB アクセスによる遅延がない。

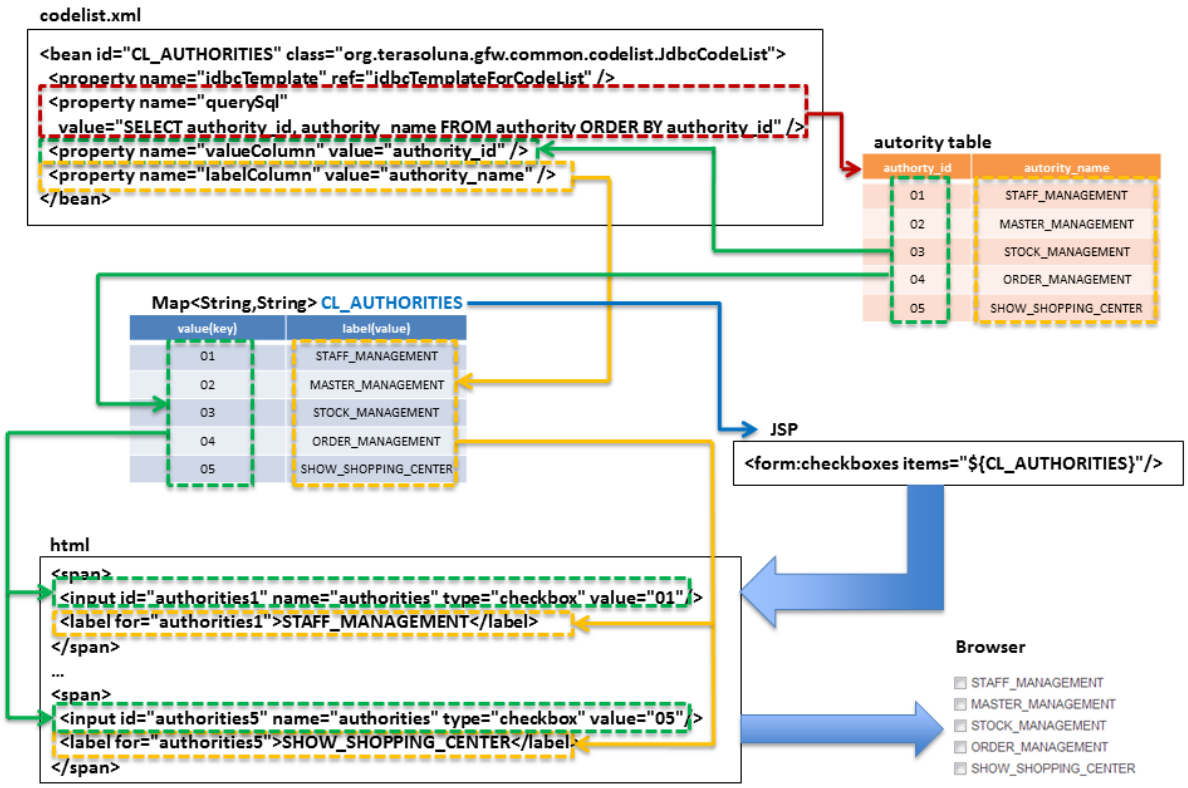
起動時の読み込み時間を抑えたいならば、取得数の上限を設定するとよい。

JdbcCodeList には org.springframework.jdbc.core.JdbcTemplate を設定するフィールドがある。

JdbcTemplate の fetchSize に上限を設定すれば、その分だけのレコードが起動時に読み込まれる。

なお、取得する値はリロードにより動的に変更できる。詳細は コードリストをリロードする場合 参照されたい。

JdbcCodeList のイメージ



コードリスト設定例

テーブル (authority) の定義

authority_id	authority_name
01	STAFF_MANAGEMENT
02	MASTER_MANAGEMENT
03	STOCK_MANAGEMENT
04	ORDER_MANAGEMENT
05	SHOW_SHOPPING_CENTER

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="jdbcTemplateForCodeList" class="org.springframework.jdbc.core.
↪JdbcTemplate" > <!-- (1) -->
    <property name="dataSource" ref="dataSource" />
    <property name="fetchSize" value="${codelist.jdbc.fetchSize:1000}" /> <!-- (2)␣
↪-->
</bean>

<bean id="AbstractJdbcCodeList"
    class="org.terasoluna.gfw.common.codelist.JdbcCodeList" abstract="true"> <!--␣
↪(3) -->
    <property name="jdbcTemplate" ref="jdbcTemplateForCodeList" /> <!-- (4) -->
</bean>

<bean id="CL_AUTHORITIES" parent="AbstractJdbcCodeList" > <!-- (5) -->
    <property name="querySql"
        value="SELECT authority_id, authority_name FROM authority ORDER BY␣
↪authority_id" /> <!-- (6) -->
    <property name="valueColumn" value="authority_id" /> <!-- (7) -->
    <property name="labelColumn" value="authority_name" /> <!-- (8) -->
</bean>
```

項番	説明
(1)	`org.springframework.jdbc.core.JdbcTemplate` クラスを bean 定義する。 独自に `fetchSize` を設定するために必要となる。
(2)	`fetchSize` を設定する。 `fetchSize` のデフォルト設定が、全件取得になっている場合があるため適切な値を設定すること。 `fetchSize` の設定が全件取得のままだと、 `JdbcCodeList` の読み込む件数が大きい場合に、DB からリストを取得する際の処理性能が落ちてしまい、アプリケーションの起動時間が長期化する可能性がある。
(3)	`JdbcCodeList` の共通 bean 定義。 他の `JdbcCodeList` の共通部分を設定している。そのため、基本 `JdbcCodeList` の bean 定義はこの bean 定義を親クラスに設定する。 abstract 属性を true にすることで、この bean はインスタンス化されない。
(4)	(1) で設定した `jdbcTemplate` を設定。 `fetchSize` を設定した `JdbcTemplate` を、 `JdbcCodeList` に格納している。
(5)	`JdbcCodeList` の bean 定義。 parent 属性を (3) の bean 定義を親クラスとして設定することで、 `fetchSize` を設定した `JdbcCodeList` が設定される。 この bean 定義では、クエリに関する設定のみを行い、必要な CodeList 分作成する。
(6)	querySql プロパティに取得する SQL を記述する。その際、 **必ず「ORDER BY」を指定し、順序を確定させること。** 「ORDER BY」を指定しないと、取得する度に順序が変わってしまう。
(7)	valueColumn プロパティに、Map の Key に該当する値を設定する。この例では authority_id を設定している。

Continued on next page

表 4.45 – continued from previous page

項番	説明
(8)	labelColumn プロパティに、Map の Value に該当する値を設定する。この例では authority_name を設定している。

JSP でのコードリスト使用

下記に示す設定の詳細について、前述した *JSP でのコードリスト使用* を参照されたい。

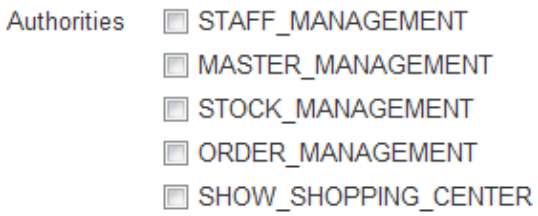
jsp の実装例

```
<form:checkboxes items="${CL_AUTHORITIES}"/>
```

出力 HTML

```
<span>
  <input id="authorities1" name="authorities" type="checkbox" value="01"/>
  <label for="authorities1">STAFF_MANAGEMENT</label>
</span>
<span>
  <input id="authorities2" name="authorities" type="checkbox" value="02"/>
  <label for="authorities2">MASTER_MANAGEMENT</label>
</span>
<span>
  <input id="authorities3" name="authorities" type="checkbox" value="03"/>
  <label for="authorities3">STOCK_MANAGEMENT</label>
</span>
<span>
  <input id="authorities4" name="authorities" type="checkbox" value="04"/>
  <label for="authorities4">ORDER_MANAGEMENT</label>
</span>
<span>
  <input id="authorities5" name="authorities" type="checkbox" value="05"/>
  <label for="authorities5">SHOW_SHOPPING_CENTER</label>
</span>
```

出力画面



Java クラスでのコードリスト使用

下記に示す設定の詳細について、前述した *Java クラスでのコードリスト使用* を参照されたい。

EnumCodeList の使用方法

`org.terasoluna.gfw.common.codelist.EnumCodeList` は、Enum クラスに定義した定数からコードリストを作成するクラスである。

注釈: 以下の条件に一致するアプリケーションでコードリストを扱う場合は、`EnumCodeList` を使用して、コードリストのラベルを Enum クラスで管理することを検討してほしい。コードリストのラベルを Enum クラスで管理することで、コード値に紐づく情報と操作を Enum クラスに集約する事ができる。

- コード値を Enum クラスで管理する必要がある (つまり、Java のロジックでコード値を意識した処理を行う必要がある)
- UI の国際化 (多言語化) の必要がない

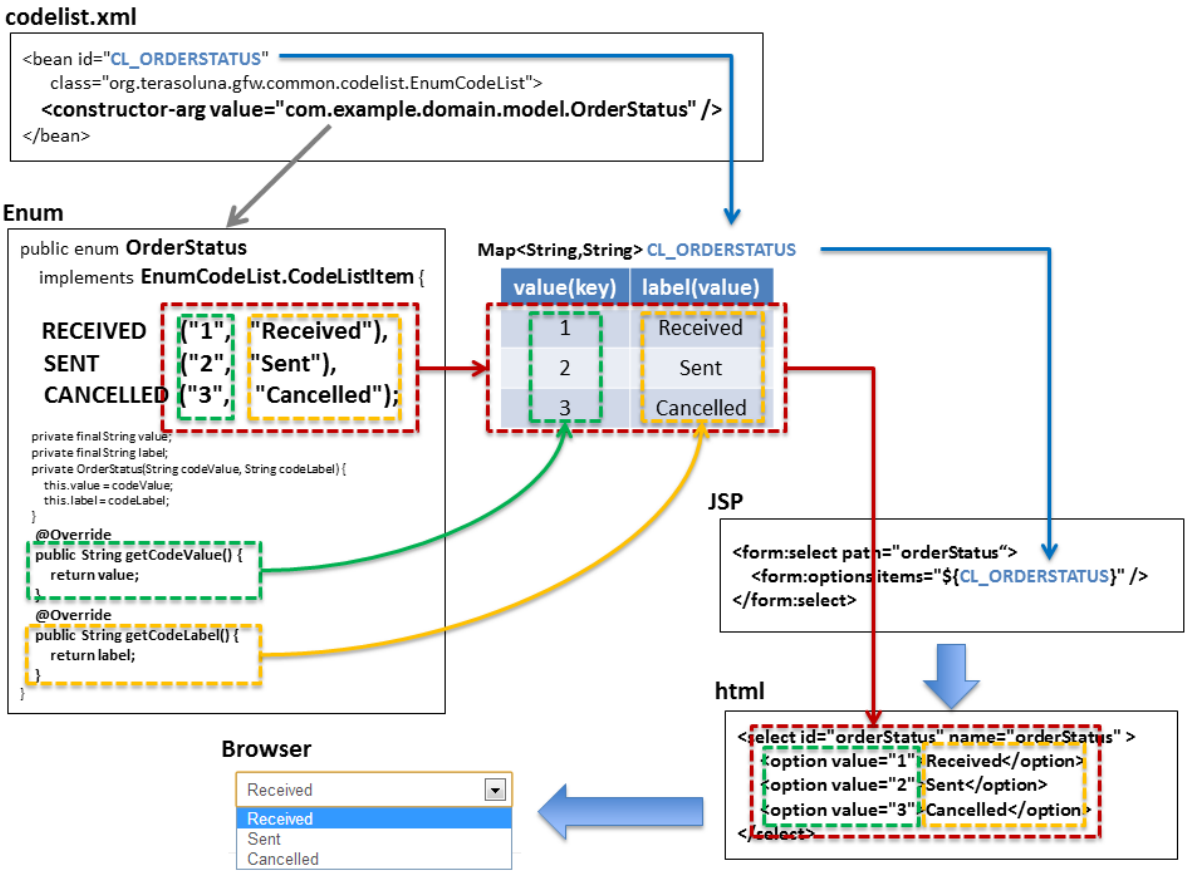
以下に、`EnumCodeList` の使用イメージを示す。

注釈: `EnumCodeList` では、Enum クラスからコードリストを作成するために必要な情報 (コード値とラベル) を取得するためのインタフェースとして、`org.terasoluna.gfw.common.codelist.EnumCodeList.CodeListItem` インタフェースを提供している。

`EnumCodeList` を使用する場合は、作成する Enum クラスで `EnumCodeList.CodeListItem` インタフェースを実装する必要がある。

コードリスト設定例

Enum クラスの作成



EnumCodeList を使用する場合は、EnumCodeList.CodeListItem インタフェースを実装した Enum クラスを作成する。以下に作成例を示す。

```
package com.example.domain.model;

import org.terasoluna.gfw.common.codelist.EnumCodeList;

public enum OrderStatus
    // (1)
    implements EnumCodeList.CodeListItem {

    // (2)
    RECEIVED  ("1", "Received"),
    SENT      ("2", "Sent"),
    CANCELLED ("3","Cancelled");

    // (3)
    private final String value;
    private final String label;

    // (4)
    private OrderStatus(String codeValue, String codeLabel) {
        this.value = codeValue;
        this.label = codeLabel;
    }
```

```
    // (5)
    @Override
    public String getCodeValue() {
        return value;
    }

    // (6)
    @Override
    public String getCodeLabel() {
        return label;
    }

}
```

項番	説明
(1)	コードリストとして使用する Enum クラスでは、共通ライブラリから提供している org.terasoluna.gfw.common.codelist.EnumCodeList.CodeListItem インタフェースを実装する。 EnumCodeList.CodeListItem インタフェースには、コードリストを作成するために必要な情報 (コード値とラベル) を取得するためのメソッドとして、 • コード値を取得する getCodeValue メソッド • ラベルを取得する getCodeLabel メソッド が定義されている。
(2)	定数を定義する。 定数を生成する際に、コードリストを作成するために必要な情報 (コード値とラベル) を指定する。 上記例では、以下の 3 つの定数を定義している。 • RECEIVED(コード値=」1」 , ラベル=Received) • SENT (コード値=」2」 , ラベル=Sent) • CANCELLED (コード値=」3」 , ラベル=Cancelled) **注釈:** EnumCodeList を使用した際のコードリストの並び順は、定数の定義順となる。
(3)	コードリストを作成するために必要な情報 (コード値とラベル) を保持するプロパティを用意する。
(4)	コードリストを作成するために必要な情報 (コード値とラベル) を受け取るコンストラクタを用意する。
(5)	定数が保持するコード値を返却する。 このメソッドは、EnumCodeList.CodeListItem インタフェースで定義されているメソッドであり、EnumCodeList が定数からコード値を取得する際に呼び出す。
(6)	定数が保持するラベルを返却する。 このメソッドは、EnumCodeList.CodeListItem インタフェースで定義されているメソッドであり、EnumCodeList が定数からラベルを取得する際に呼び出す。

bean 定義ファイル (xxx-codelist.xml) の定義

コードリスト用の bean 定義ファイルに、EnumCodeList を定義する。以下に定義例を示す。

```
<bean id="CL_ORDERSTATUS"
      class="org.terasoluna.gfw.common.codelist.EnumCodeList"> <!-- (7) -->
    <constructor-arg value="com.example.domain.model.OrderStatus" /> <!-- (8) -->
</bean>
```

項番	説明
(7)	コードリストの実装クラスとして、EnumCodeList クラスを指定する。
(8)	EnumCodeList クラスのコンストラクタに、EnumCodeList.CodeListItem インタフェースを実装した Enum クラスの FQCN を指定する。

JSP でのコードリスト使用

JSP でコードリストを使用する方法については、前述した *JSP でのコードリスト使用* を参照されたい。

Java クラスでのコードリスト使用

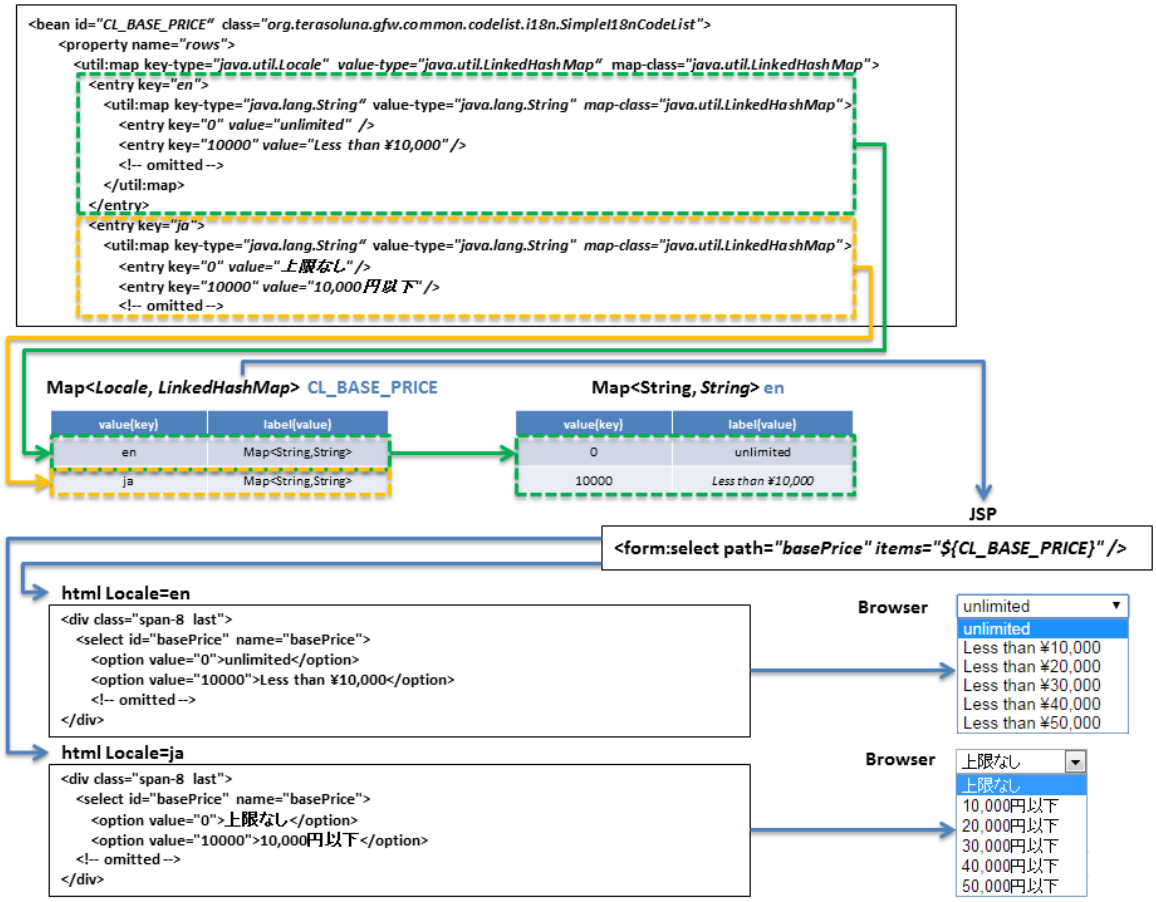
Java クラスでコードリストを使用する方法については、前述した *Java クラスでのコードリスト使用* を参照されたい。

I18nCodeList の使用方法

org.terasoluna.gfw.common.codelist.i18n.I18nCodeList は、国際化に対応しているコードリストである。ロケール毎にコードリストを設定することで、ロケールに対応したコードリストを返却できる。

I18nCodeList の実装クラスとして、org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList および org.terasoluna.gfw.common.codelist.i18n.SimpleReloadableI18nCodeList を提供している。

I18nCodeList（SimpleI18nCodeList）のイメージ



コードリスト設定例

`I18nCodeList` は行が `Locale`、列がコード値、セルの内容がラベルである 2 次元のテーブルをイメージすると理解しやすい。

料金を選択するセレクトボックスの場合を例に挙げると以下のようなテーブルができる。

row=Locale, column=Code	0	10000	20000	30000	40000	50000
en	unlimited	Less than \10,000	Less than \20,000	Less than \30,000	Less than \40,000	Less than \50,000
ja	上限なし	10,000 円以下	20,000 円以下	30,000 円以下	40,000 円以下	50,000 円以下

この国際化対応コードリストのテーブルを構築するために `SimpleI18nCodeList` は 3 つの設定方法を用意している。

- 行単位で Locale 毎の `CodeList` を設定する
- 行単位で Locale 毎の `java.util.Map`(key=コード値, value=ラベル) を設定する
- 列単位でコード値毎の `java.util.Map`(key=Locale, value=ラベル) を設定する

基本的には、「行単位で **Locale** 毎の **CodeList** を設定する」方法でコードリストを設定することを推奨する。

`SimpleReloadableI18nCodeList` は更新可能なコードリストを行に持つ以下の設定方法を用意している。

- 行単位で Locale 毎の `ReloadableCodeList`（`JdbcCodeList`）を設定する

注釈: terasoluna-gfw-common 5.4.2.RELEASE から リロードに対応した `SimpleReloadableI18nCodeList` が追加された。更新可能なコードリストを行に持つ `SimpleI18nCodeList` を利用している場合は、`SimpleReloadableI18nCodeList` に置き換えることを推奨する。

上記例の料金を選択するセレクトボックスの場合を行単位で Locale 毎の `CodeList` を設定する方法について説明する。他の設定方法については *SimpleI18nCodeList のコードリスト設定方法* 参照されたい。

Bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_I18N_PRICE"
    class="org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList">
    <property name="rowsByCodeList"> <!-- (1) -->
        <util:map>
            <entry key="en" value-ref="CL_PRICE_EN" />
            <entry key="ja" value-ref="CL_PRICE_JA" />
        </util:map>
    </property>
</bean>
```

項番	説明
(1)	rowsByCodeList プロパティに key が `java.lang.Locale` の Map を設定する。 Map には、key にロケール、value-ref にロケールに対応したコードリストクラスの参照先を指定する。 Map の value は各ロケールに対応したコードリストクラスを参照する。

Locale 毎に SimpleMapCodeList を用意する場合の Bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_I18N_PRICE"
    class="org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList">
    <property name="rowsByCodeList">
        <util:map>
            <entry key="en" value-ref="CL_PRICE_EN" />
            <entry key="ja" value-ref="CL_PRICE_JA" />
        </util:map>
    </property>
</bean>

<bean id="CL_PRICE_EN" class="org.terasoluna.gfw.common.codelist.SimpleMapCodeList
↪">  <!-- (2) -->
    <property name="map">
        <util:map>
            <entry key="0" value="unlimited" />
            <entry key="10000" value="Less than \\10,000" />
            <entry key="20000" value="Less than \\20,000" />
            <entry key="30000" value="Less than \\30,000" />
            <entry key="40000" value="Less than \\40,000" />
            <entry key="50000" value="Less than \\50,000" />
        </util:map>
    </property>
</bean>

<bean id="CL_PRICE_JA" class="org.terasoluna.gfw.common.codelist.SimpleMapCodeList
↪">  <!-- (3) -->
    <property name="map">
        <util:map>
            <entry key="0" value="上限なし" />
            <entry key="10000" value="10,000 円以下" />
            <entry key="20000" value="20,000 円以下" />
            <entry key="30000" value="30,000 円以下" />
            <entry key="40000" value="40,000 円以下" />
            <entry key="50000" value="50,000 円以下" />
        </util:map>
    </property>
</bean>
```

項番	説明
(2)	ロケールが」en」である bean 定義 CL_PRICE_EN について、コードリストクラスを SimpleMapCodeList で設定している。
(3)	ロケールが」ja」である bean 定義 CL_PRICE_JA について、コードリストクラスを SimpleMapCodeList で設定している。

Locale 毎に JdbcCodeList を用意する場合の Bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_I18N_PRICE"
    class="org.terasoluna.gfw.common.codelist.i18n.SimpleReloadableI18nCodeList">
↪<!-- (4) -->
    <property name="rowsByCodeList">
        <util:map>
            <entry key="en" value-ref="CL_PRICE_EN" />
            <entry key="ja" value-ref="CL_PRICE_JA" />
        </util:map>
    </property>
</bean>

<bean id="CL_PRICE_EN" parent="AbstractJdbcCodeList">  <!-- (5) -->
    <property name="querySql"
        value="SELECT code, label FROM price WHERE locale = 'en' ORDER BY code" />
    <property name="valueColumn" value="code" />
    <property name="labelColumn" value="label" />
</bean>

<bean id="CL_PRICE_JA" parent="AbstractJdbcCodeList">  <!-- (6) -->
    <property name="querySql"
        value="SELECT code, label FROM price WHERE locale = 'ja' ORDER BY code" />
    <property name="valueColumn" value="code" />
    <property name="labelColumn" value="label" />
</bean>
```

項番	説明
(4)	更新可能なコードリストを行に持つ場合は、SimpleReloadableI18nCodeList を利用する。
(5)	ロケールが」en」である bean 定義 CL_PRICE_EN について、コードリストクラスを JdbcCodeList で設定している。
(6)	ロケールが」ja」である bean 定義 CL_PRICE_JA について、コードリストクラスを JdbcCodeList で設定している。

テーブル定義 (price テーブル) には以下のデータを投入する。

locale	code	label
en	0	unlimited
en	10000	Less than \10,000
en	20000	Less than \20,000
en	30000	Less than \30,000
en	40000	Less than \40,000
en	50000	Less than \50,000
ja	0	上限なし
ja	10000	10,000 円以下
ja	20000	20,000 円以下
ja	30000	30,000 円以下
ja	40000	40,000 円以下
ja	50000	50,000 円以下

JSP でのコードリスト使用

基本的な設定は、前述した *JSP でのコードリスト使用* と同様のため、説明は省略する。

ここでは、 リクエストのロケールがコードリスト定義されていなかった場合に対応するため、インターセプターでリクエストスコープに `I18nCodeList` を設定し、JSP に渡す方法を紹介する。

bean 定義ファイル (spring-mvc.xml) の定義

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**" />
    <bean
      class="org.terasoluna.gfw.web.codelist.CodeListInterceptor">
      <property name="codeListIdPattern" value="CL_.+" />
      <property name="fallbackTo" value="en" />  <!-- (1) -->
    </bean>
  </mvc:interceptor>

  <!-- omitted -->

</mvc:interceptors>
```

項番	説明
(1)	リクエストのロケールがコードリスト定義されていなかった場合、 fallbackTo プロパティに設定されたロケールでコードリストを取得する。 fallbackTo プロパティが設定されていない場合、JVM のデフォルトロケールが fallbackTo プロパティとして使用される。 fallbackTo プロパティに設定されたロケールでも、コードリストが取得されない場合、WARN ログを出力し、空の Map を返却する。

jsp の実装例

```
<form:select path="basePrice" items="${CL_I18N_PRICE}" />
```

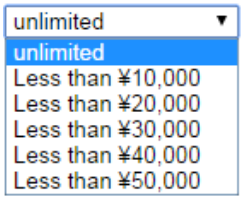
出力 **HTML lang=en**

```
<select id="basePrice" name="basePrice">
  <option value="0">unlimited</option>
  <option value="1">Less than \\10,000</option>
  <option value="2">Less than \\20,000</option>
  <option value="3">Less than \\30,000</option>
  <option value="4">Less than \\40,000</option>
  <option value="5">Less than \\50,000</option>
</select>
```

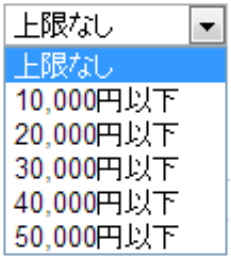
出力 **HTML lang=ja**

```
<select id="basePrice" name="basePrice">
  <option value="0">上限なし</option>
  <option value="1">10,000 円以下</option>
  <option value="2">20,000 円以下</option>
  <option value="3">30,000 円以下</option>
  <option value="4">40,000 円以下</option>
  <option value="5">50,000 円以下</option>
</select>
```

出力画面 **lang=en**



出力画面 **lang=ja**



Java クラスでのコードリスト使用

基本的な設定は、前述した *Java クラスでのコードリスト使用* と同様のため、説明は省略する。

```
@RequestMapping("orders")
@Controller
public class OrderController {
```

```
    @Inject
    @Named("CL_I18N_PRICE")
    I18nCodeList priceCodeList;

    // ...

    @RequestMapping(method = RequestMethod.POST, params = "confirm")
    public String confirm(OrderForm form, Locale locale) {
        // ...
        String priceMassage = getPriceMessage(form.getPriceCode(), locale);
        // ...
    }

    private String getPriceMessage(String targetPrice, Locale locale) {
         return priceCodeList.asMap(locale).get(targetPrice);  // (1)
    }

}
```

項番	説明
(1)	I18nCodeList#asMap(Locale) で対応したロケールの Map を取得することができる。

特定のコード値からコード名を表示する

JSP からコードリストを参照する場合は、 `CodeListInterceptor` がリクエストスコープにコードリストを `java.util.Map` で格納しているため、`Map` インターフェースと同じ方法で参照することができる。

コードリストを用いて特定のコード値からコード名を表示する方法について、以下に実装例を示す。

jsp の実装例

```
Order Status : ${f:h(CL_ORDERSTATUS[orderForm.orderStatus])}
```

項番	説明
(1)	コードリストを定義した beanID(この例では `CL_ORDERSTATUS`) を属性名として、コードリスト (`java.util.Map` インタフェース) を取得する。取得した `Map` インタフェースのキーとしてコード値 (この例では `orderStatus` に格納された値) を指定することで、対応するコード名を表示することができる。

コードリストを用いたコード値の入力チェック

入力値がコードリスト内に定義されたコード値であるかどうかチェックするような場合、共通ライブラリでは、BeanValidation 用のアノテーション、`org.terasoluna.gfw.common.codelist.ExistInCodeList` を提供している。

BeanValidation や、メッセージ出力方法の詳細については、 入力チェック を参照されたい。

`@ExistInCodeList` アノテーションを使用して入力チェックを行う場合は、`@ExistInCodeList` 用の「エラーメッセージの定義」を行う必要がある。

ブランクプロジェクト からプロジェクトを生成した場合は、`xxx-web/src/main/resources` の直下の `ValidationMessages.properties` ファイルの中に以下のメッセージが定義されている。メッセージは、アプリケーションの要件に合わせて変更すること。

```
org.terasoluna.gfw.common.codelist.ExistInCodeList.message = Does not exist in
↪{codeListId}
```

注釈: terasoluna-gfw-common 5.0.0.RELEASE より、メッセージのプロパティキーの形式を、Bean Validation のスタンダードな形式 (アノテーションの FQCN + `.message`) に変更している。

バージョン	メッセージのプロパティキー
version 5.0.0.RELEASE 以降	`org.terasoluna.gfw.common.codelist.ExistInCodeList.message`
version 1.0.x.RELEASE	`org.terasoluna.gfw.common.codelist.ExistInCodeList`

version 1.0.x.RELEASE から version 5.0.0.RELEASE 以降にバージョンアップする際に、アプリケーション要件に合わせてメッセージを変更している場合は、プロパティキーの変更が必要になる。

注釈: terasoluna-gfw-common 1.0.2.RELEASE より、`@ExistInCodeList` のメッセージを定義した `ValidationMessages.properties` を、jar ファイルの中に含めないようにしている。これは、「ValidationMessages.properties が複数存在する場合にメッセージが表示されないバグ」を修正するためである。

version 1.0.1.RELEASE 以前から version 1.0.2.RELEASE 以降にバージョンアップする際に、terasoluna-gfw-common の jar の中に含まれる `ValidationMessages.properties` に定義しているメッセージを使用し

ている場合は、ValidationMessages.properties を作成してメッセージを定義する必要がある。

@ExistInCodeList の設定例

コードリストを用いた入力チェック方法について、以下に実装例を示す。

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_GENDER" class="org.terasoluna.gfw.common.codelist.SimpleMapCodeList">
    <property name="map">
        <map>
            <entry key="M" value="Male" />
            <entry key="F" value="Female" />
        </map>
    </property>
</bean>
```

Form オブジェクト

```
public class Person {
    @ExistInCodeList(codeListId = "CL_GENDER")  // (1)
    private String gender;

    // getter and setter omitted
}
```

項番	説明
(1)	入力チェックを行いたいフィールドに対して、 @ExistInCodeList アノテーションを設定し、codeListId にチェック元となる、コードリストを指定する。

上記の結果、 gender に M、F 以外の文字が格納されている場合、エラーになる。

注釈: terasoluna-gfw-common 5.4.2.RELEASE から、@ExistInCodeList の入力チェックの対象として、CharSequence インタフェースの実装クラス (String など) または Character に加え、Number 継承クラス（Integer など）をサポートするよう変更された。

NumberRangeCodeList の valueFormat プロパティを指定している場合、 Number 型フィールドの値を当該プロパティを利用してフォーマットした値がコードリストに存在することをチェックする。

また、`@ExistInCodeList` はコレクション内の値には対応していないため、複数選択可能な画面項目（チェックボックスや複数選択ドロップダウンなど）に`@ExistInCodeList` アノテーションを対応させるためには実装を工夫する必要がある。詳細については コレクション内の値を *Bean Validation* のアノテーションを使用してチェックする方法 を参照されたい。

4.8.3 How to extend

コードリストをリロードする場合

前述した共通ライブラリで提供しているコードリストは、アプリケーション起動時に読み込まれ、それ以降は、基本的に更新されない。しかし、コードリストのマスタデータを更新した時、コードリストも更新したい場合がある。

例：JdbcCodeList を使用して、DB のマスタを変更した時にコードリストの更新を行う場合。

共通ライブラリでは、コードリストを更新可能とするインタフェースを提供している。

1. `org.terasoluna.gfw.common.codelist.ReloadableCodeList`：コードリストを更新する
2. `org.terasoluna.gfw.common.codelist.i18n.ReloadableI18nCodeList` ：行に持つコードリストを含むコードリストを更新する

コードリストの更新方法としては、以下 2 点の方法がある。

1. Task Scheduler で実現する方法
2. Controller(Service) クラスで refresh メソッドを呼び出す方法

本ガイドラインでは、Spring から提供されている Task Scheduler を使用して、コードリストを定期的にリロードする方式を基本的に推奨する。

ただし、任意のタイミングでコードリストをリフレッシュする必要がある場合は Controller クラスで refresh メソッドを呼び出す方法で実現すればよい。

注釈: `ReloadableCodeList` および `ReloadableI18nCodeList` インターフェースを実装しているコードリストについては、 コードリスト種類一覧 を参照されたい。

Task Scheduler で実現する方法

Task Scheduler の設定例について、以下に示す。

bean 定義ファイル (xxx-codelist.xml) の定義

```
<task:scheduler id="taskScheduler" pool-size="10"/>  <!-- (1) -->

<task:scheduled-tasks scheduler="taskScheduler">  <!-- (2) -->
    <task:scheduled ref="CL_AUTHORITIES" method="refresh" cron="${cron.codelist.
↪refreshTime}"/>  <!-- (3) -->
</task:scheduled-tasks>

<bean id="CL_AUTHORITIES" parent="AbstractJdbcCodeList">
    <property name="querySql"
        value="SELECT authority_id, authority_name FROM authority ORDER BY␣
↪authority_id" />
    <property name="valueColumn" value="authority_id" />
    <property name="labelColumn" value="authority_name" />
</bean>
```

項番	説明
(1)	<task:scheduler> の要素を定義する。pool-size 属性にスレッドのプールサイズを指定する。 pool-size 属性を指定しない場合、」1」 が設定される。
(2)	<task:scheduled-tasks> の要素を定義し、scheduler 属性に、 <task:scheduler> の ID を設定する。
(3)	<task:scheduled> 要素を定義する。method 属性に、refresh メソッドを指定する。 cron 属性に、 org.springframework.scheduling.support.CronSequenceGenerator でサポートされた形式で記述すること。 cron 属性は開発環境、商用環境など環境によってリロードするタイミングが変わることが想定されるため、プロパティファイルや、環境変数等から取得することを推奨する。 **cron 属性の設定例** 「秒 分 時 月 年 曜日」で指定する。 毎秒実行 「* * * * * *」 毎時実行 「0 0 * * * *」 平日の 9-17 時の毎時実行 「0 0 9-17 * * MON-FRI」 詳細は JavaDoc を参照されたい。 https://docs.spring.io/spring/docs/4.3.23.RELEASE/javadoc-api/org/springframework/scheduling/support/CronSequenceGenerator.html

Controller(Service) クラスで refresh メソッドを呼び出す方法

refresh メソッドを直接呼び出す場合について、JdbcCodeList の refresh メソッドを Service クラスで呼び出す場合の実装例を、以下に示す。

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_AUTHORITIES" parent="AbstractJdbcCodeList">
    <property name="querySql"
        value="SELECT authority_id, authority_name FROM authority ORDER BY␣
↪authority_id" />
    <property name="valueColumn" value="authority_id" />
    <property name="labelColumn" value="authority_name" />
</bean>
```

Controller クラス

```
@Controller
@RequestMapping(value = "codelist")
public class CodeListController {

    @Inject
    CodeListService codeListService; // (1)

    @RequestMapping(method = RequestMethod.GET, params = "refresh")
    public String refreshJdbcCodeList() {
        codeListService.refresh(); // (2)
        return "codelist/jdbcCodeList";
    }
}
```

項番	説明
(1)	ReloadableCodeList クラスの refresh メソッドを実行する Service クラスをインジェクションする。
(2)	ReloadableCodeList クラスの refresh メソッドを実行する Service クラスの refresh メソッドを実行する。

Service クラス

以下は実装クラスのみ記述し、インターフェースクラスは省略。

```
@Service
public class CodeListServiceImpl implements CodeListService { // (3)

    @Inject
    @Named(value = "CL_AUTHORITIES") // (4)
    ReloadableCodeList codeListItem; // (5)

    @Override
    public void refresh() { // (6)
        codeListItem.refresh(); // (7)
    }
```

```
}
```

項番	説明
(3)	実装クラス `CodeListServiceImpl` は、インターフェース `CodeListService` を実装する。
(4)	コードリストをインジェクションするとき、 `@Named` で、該当するコードリストを指定する。 value 属性に取得したい bean の ID を指定すること。 Bean 定義ファイルに定義されている bean タグの ID 属性」CL_AUTHORITIES」のコードリストがインジェクションされる。
(5)	フィールドの型に ReloadableCodeList インターフェースを定義すること。 (4) で指定した Bean は ReloadableCodeList インターフェースを実装していること。
(6)	Service クラスで定義した refresh メソッド。 Controller クラスから呼び出されている。
(7)	ReloadableCodeList インターフェースを実装したコードリストの refresh メソッド。 refresh メソッドを実行することで、コードリストが更新される。

注釈: terasoluna-gfw-common 5.4.2.RELEASE で追加された `SimpleReloadableI18nCodeList` では、refresh メソッドで行に持つすべての ReloadableCodeList を更新することが可能である。

アプリケーションの実装によっては、行に持つ ReloadableCodeList が更新されている前提で `SimpleReloadableI18nCodeList` のみ更新すれば良い場合もあり得る。この場合は、`ReloadableI18nCodeList#refresh(boolean)` メソッドの引数に `false` をセットして実行すれば良い。

コードリストを独自カスタマイズする方法

共通ライブラリで提供している 4 種類のコードリストで実現できないコードリストを作成したい場合、コードリストを独自にカスタマイズすることができる。独自カスタマイズする場合、作成できるコードリストの種類と実装方法について、以下の表に示す。

項番	Reloadable	継承するクラス	実装箇所
(1)	不要	`org.terasoluna.gfw.common.codelist.AbstractCodeList`	`asMap` をオーバライド
(2)	必要	`org.terasoluna.gfw.common.codelist.AbstractReloadableCodeList`	`retrieveMap` をオーバライド

`org.terasoluna.gfw.common.codelist.CodeList` 、 `org.terasoluna.gfw.common.codelist.ReloadableCodeList` インターフェースを直接実装しても実現はできるが、共通ライブラリで提供されている抽象クラスを拡張することで、最低限の実装で済む。

以下に、独自カスタマイズの実例について示す。例として、今年と来年の年のリストを作るコードリストについて説明する。(例：今年が 2013 の場合、コードリストには、」2013、2014」の順で格納される。)

コードリストクラス

```
package com.example.sample.domain.codelist;

...

public class DepYearCodeList extends AbstractCodeList { // (1)

    private JodaTimeDateFactory dateFactory;

    public void setDateFactory(JodaTimeDateFactory dateFactory) { //(2)
        this.dateFactory = dateFactory;
    }

    @Override
    public Map<String, String> asMap() {  // (3)
        DateTime dateTime = dateFactory.newDateTime();
        DateTime nextYearDateTime = dateTime.plusYears(1);

        Map<String, String> depYearMap = new LinkedHashMap<String, String>();

        String thisYear = dateTime.toString("Y");
        String nextYear = nextYearDateTime.toString("Y");
        depYearMap.put(thisYear, thisYear);
```

```
        depYearMap.put(nextYear, nextYear);

        return Collections.unmodifiableMap(depYearMap);
    }
}
```

項番	説明
(1)	AbstractCodeList を継承する。 今年と来年の年のリストを作る時、動的にシステム日付から算出して作成しているため、リロードは不要。
(2)	システム日付の Date クラスを作成する org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory をインジェクションするためのセッターを用意する。 JodaTimeDateFactory を利用して今年と来年の年を取得することができる。
(3)	asMap メソッドをオーバライドして、今年と来年の年のリストを作成する。 作成したいコードリスト毎に実装が異なる。

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_YEAR" class="com.example.sample.domain.codelist.DepYearCodeList"> <!--
↪ (1) -->
    <property name="dateFactory" ref="dateFactory" /> <!-- (2) -->
</bean>
```

項番	説明
(1)	作成したコードリストクラスを bean 定義する。 id に CL_YEAR を指定することで、bean 定義で設定した CodeListInterceptor によりコードリストをコンポーネント登録する。
(2)	システム日付の Date クラスを作成する JodaTimeDateFactory を設定する。 事前に、bean 定義ファイルに DataFactory 実装クラスを設定する必要がある。

jsp の実装例

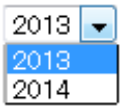
```
<form:select path="mostRecentYear" items="${CL_YEAR}" /> <!-- (1) -->
```

項番	説明
(1)	items 属性にコンポーネント登録した CL_YEAR を ${} プレースホルダー で指定することで、該当のコードリストを取得することができる。

出力 HTML

```
<select id="mostRecentYear" name="mostRecentYear">
   <option value="2013">2013</option>
   <option value="2014">2014</option>
</select>
```

出力画面



注釈: リロード可能である CodeList を独自カスタマイズする場合、スレッドセーフになるように実装すること。

4.8.4 Appendix

SimpleI18nCodeList のコードリスト設定方法

SimpleI18nCodeList のコードリスト設定について、 *I18nCodeList の使用方法* で設定されているコードリスト設定の他に 2 つ設定方法がある。料金を選択するセレクトボックスの場合の例を用いて、それぞれの設定方法を説明する。

行単位で Locale 毎の `java.util.Map`(key=コード値, value=ラベル) を設定する

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_I18N_PRICE"
    class="org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList">
    <property name="rows"> <!-- (1) -->
        <util:map>
            <entry key="en">
                <util:map>
                    <entry key="0" value="unlimited" />
                    <entry key="10000" value="Less than \\10,000" />
                    <entry key="20000" value="Less than \\20,000" />
                    <entry key="30000" value="Less than \\30,000" />
                    <entry key="40000" value="Less than \\40,000" />
                    <entry key="50000" value="Less than \\50,000" />
                </util:map>
            </entry>
            <entry key="ja">
                <util:map>
                    <entry key="0" value="上限なし" />
                    <entry key="10000" value="10,000 円以下" />
                    <entry key="20000" value="20,000 円以下" />
                    <entry key="30000" value="30,000 円以下" />
                    <entry key="40000" value="40,000 円以下" />
                    <entry key="50000" value="50,000 円以下" />
                </util:map>
            </entry>
        </util:map>
    </property>
</bean>
```

項番	説明
(1)	rows プロパティに対して、」Map の Map」を設定する。外側の Map の key は `java.lang.Locale` である。 内側の Map の key はコード値、value はロケールに対応したラベルである。

列単位でコード値毎の **`java.util.Map`(key=Locale, value=ラベル)** を設定する

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_I18N_PRICE"
    class="org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList">
    <property name="columns"> <!-- (1) -->
        <util:map>
            <entry key="0">
```

```
                <util:map>
                    <entry key="en" value="unlimited" />
                    <entry key="ja" value="上限なし" />
                </util:map>
            </entry>
            <entry key="10000">
                <util:map>
                    <entry key="en" value="Less than \\10,000" />
                    <entry key="ja" value="10,000 円以下" />
                </util:map>
            </entry>
            <entry key="20000">
                <util:map>
                    <entry key="en" value="Less than \\20,000" />
                    <entry key="ja" value="20,000 円以下" />
                </util:map>
            </entry>
            <entry key="30000">
                <util:map>
                    <entry key="en" value="Less than \\30,000" />
                    <entry key="ja" value="30,000 円以下" />
                </util:map>
            </entry>
            <entry key="40000">
                <util:map>
                    <entry key="en" value="Less than \\40,000" />
                    <entry key="ja" value="40,000 円以下" />
                </util:map>
            </entry>
            <entry key="50000">
                <util:map>
                    <entry key="en" value="Less than \\50,000" />
                    <entry key="ja" value="50,000 円以下" />
                </util:map>
            </entry>
        </util:map>
    </property>
</bean>
```

項番	説明
(1)	columns プロパティに対して、」Map の Map」を設定する。外側の Map の key はコード値である。内側の Map の key は `java.lang.Locale`、value はロケールに対応したラベルである。

NumberRangeCodeList のバリエーション

降順の NumberRangeCodeList の作成

次に、To の値を From の値より小さくする (To < From) 場合の実装例を、以下に示す。

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_BIRTH_YEAR"
    class="org.terasoluna.gfw.common.codelist.NumberRangeCodeList">
    <property name="from" value="2013" /> <!-- (1) -->
    <property name="to" value="2000" /> <!-- (2) -->
</bean>
```

項番	説明
(1)	範囲開始の値を指定する。name 属性」to」の value 属性の値より大きい値を指定する。 この指定によって、interval 分減少した値を、To～From の範囲分のリストとして、降順に表示する。 interval は設定していないため、デフォルトの値 1 が適用される。
(2)	範囲終了の値を設定する。 本例では、2000 を指定することにより、リストには 2013～2000 までの範囲で 1 ずつ減少して格納される。

jsp の実装例

```
<form:select path="birthYear" items="${CL_BIRTH_YEAR}" />
```

出力 HTML

```
<select id="birthYear" name="birthYear">
  <option value="2013">2013</option>
  <option value="2012">2012</option>
  <option value="2011">2011</option>
  <option value="2010">2010</option>
  <option value="2009">2009</option>
  <option value="2008">2008</option>
  <option value="2007">2007</option>
  <option value="2006">2006</option>
  <option value="2005">2005</option>
  <option value="2004">2004</option>
```

```
  <option value="2003">2003</option>
  <option value="2002">2002</option>
  <option value="2001">2001</option>
  <option value="2000">2000</option>
</select>
```

出力画面



NumberRangeCodeList のインターバルの変更

次に、interval 値を設定する場合の実装例を、以下に示す。

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_BULK_ORDER_QUANTITY_UNIT"
    class="org.terasoluna.gfw.common.codelist.NumberRangeCodeList">
    <property name="from" value="10" />
    <property name="to" value="50" />
    <property name="interval" value="10" /> <!-- (1) -->
</bean>
```

項番	説明
(1)	増加 (減少) 値を指定する。この指定によって、interval 値を増加 (減少) した値を、From～To の範囲内でコードリストとして格納する。 上記の例だと、コードリストには 10,20,30,40,50 の順で格納される。

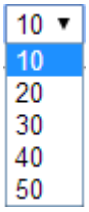
jsp の実装例

```
<form:select path="quantity" items="${CL_BULK_ORDER_QUANTITY_UNIT}" />
```

出力 HTML

```
<select id="quantity" name="quantity">
    <option value="10">10</option>
    <option value="20">20</option>
    <option value="30">30</option>
    <option value="40">40</option>
    <option value="50">50</option>
</select>
```

出力画面



注釈: interval 値分増加 (減少) した値が、Form～To の値が範囲を超えた場合は、コードリストに格納されない。

具体的には、

```
<bean id="CL_BULK_ORDER_QUANTITY_UNIT"
    class="org.terasoluna.gfw.common.codelist.NumberRangeCodeList">
    <property name="from" value="10" />
    <property name="to" value="55" />
    <property name="interval" value="10" />
</bean>
```

という定義を行った場合、

コードリストには 10,20,30,40,50 の計 5 つが格納される。次の interval である 60 及び範囲の閾値である 55 はコードリストに格納されない。

JSP から直接コードリスト Bean を参照する

JSP でのコードリスト使用 では、Spring MVC を経由する全てのリクエストに対して、CodeListIntercepter がコードリストの Bean をリクエスト属性として登録するため、コードリストの数が多くなるとリクエスト毎のオーバーヘッドの増加が懸念される。

ここでは、リクエスト毎のオーバーヘッドの増加を防ぐ方法の一つとして、コードリスト Bean を JSP から直接参照する方法を紹介する。JSP では SpEL を利用して直接 Bean を参照することができるが、こちらを利用することでオーバーヘッドの増加を防止することができる。いずれの方法を利用するかは、プロジェクトの要件によって適切に検討されたい。

注釈: 国際化対応のため SimpleI18nCodeList を使用している場合は *SimpleI18nCodeList を JSP から直接参照する方法* で紹介している方法を参照されたい。*JSP から直接コードリスト Bean を参照する* を使用する場合は、CodeListInterceptor が実施しているような、SimpleI18nCodeList の asMap メソッドに渡すロケールを決定するロジックを独自に実装する必要があるためである。

---

bean 定義ファイル (spring-mvc.xml) の定義

```
  <mvc:interceptors>
      <mvc:interceptor>
          <mvc:mapping path="/**" />
          <bean class="org.terasoluna.gfw.web.codelist.CodeListInterceptor"> <!--␣
↪(1) -->
              <property name="codeListIdPattern" value="CL_.+" />
          </bean>
      </mvc:interceptor>

  <!-- omitted -->

  </mvc:interceptors>
```

項番	説明
(1)	CodeListInterceptor の設定があれば、削除する。

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_ORDERSTATUS" class="org.terasoluna.gfw.common.codelist.␣
↪SimpleMapCodeList">
    <property name="map">
        <util:map>
            <entry key="1" value="Received" />
            <entry key="2" value="Sent" />
            <entry key="3" value="Cancelled" />
        </util:map>
    </property>
```

```
</bean>
```

jsp の実装例

```
<spring:eval var="statuses" expression="@CL_ORDERSTATUS.asMap()"/> <!-- (1) -->
<form:select items="${statuses}" path="orderStatus" />
```

項番	説明
(1)	SpEL により取得した CodeList 型 Bean の asMap メソッドにより、Map 形式で取得することができる。

出力 HTML

```
<select id="orderStatus" name="orderStatus">
    <option value="1">Received</option>
    <option value="2">Sent</option>
    <option value="3">Cancelled</option>
</select>
```

SimpleI18nCodeList を JSP から直接参照する方法

ここでは、CodeListInterceptor の実装と同様に、リクエストのロケールに対するコードリストに定義されていなかった場合、デフォルトで設定したロケールに対するコードリストを表示する例を紹介する。

bean 定義ファイル (spring-mvc.xml) の定義

bean 定義ファイル (spring-mvc.xml) の定義 と同様なため割愛する。

bean 定義ファイル (xxx-codelist.xml) の定義

```
<bean id="CL_I18N_PRICE"
    class="org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList">
    <property name="rowsByCodeList">
        <util:map>
            <entry key="en" value-ref="CL_PRICE_EN" />
            <entry key="ja" value-ref="CL_PRICE_JA" />
        </util:map>
    </property>
</bean>

<bean id="CL_PRICE_EN" class="org.terasoluna.gfw.common.codelist.SimpleMapCodeList
↪">
```

```
    <property name="map">
        <util:map>
            <entry key="0" value="unlimited" />
            <entry key="10000" value="Less than \\10,000" />
            <entry key="20000" value="Less than \\20,000" />
            <entry key="30000" value="Less than \\30,000" />
            <entry key="40000" value="Less than \\40,000" />
            <entry key="50000" value="Less than \\50,000" />
        </util:map>
    </property>
</bean>

<bean id="CL_PRICE_JA" class="org.terasoluna.gfw.common.codelist.SimpleMapCodeList
↪">
    <property name="map">
        <util:map>
            <entry key="0" value="上限なし" />
            <entry key="10000" value="10,000 円以下" />
            <entry key="20000" value="20,000 円以下" />
            <entry key="30000" value="30,000 円以下" />
            <entry key="40000" value="40,000 円以下" />
            <entry key="50000" value="50,000 円以下" />
        </util:map>
    </property>
</bean>
```

プロパティファイル

```
simpleI18nCodeList.fallback.locale = en
```

Controller クラス

```
...

@Controller
public class OrderController {

    @Value("${simpleI18nCodeList.fallback.locale}") // (1)
    private Locale fallBackLocale;

    @RequestMapping(value = "price", method = RequestMethod.GET)
    public String price(Model model, HttpServletRequest request) {
        model.addAttribute("requestLocale", RequestContextUtils
            .getLocale(request)); // (2)
        model.addAttribute("fallBackLocale",fallBackLocale); // (3)

        return "order/price";
    }
}
```

項番	説明
(1)	リクエストで指定したロケールがコードリストに定義されていなかった場合に、どのロケールのコードリストを取得するかをプロパティファイルから取得し、fallBackLocale 変数に設定する。
(2)	org.springframework.web.servlet.support.RequestContextUtils 利用してリクエストで指定されたロケールを取得し、Model に登録する。RequestContextUtils の getLocale メソッドは、引数に javax.servlet.http.HttpServletRequest を取るため、この場合は HttpServletRequest をハンドラメソッドの引数にとっても良い。
(3)	(1) で取得した fallBackLocale を Model に登録する。

jsp の実装例

```
<spring:eval var="price" expression="@CL_I18N_PRICE.asMap(requestLocale).isEmpty()␣
↪? @CL_I18N_PRICE.asMap(fallBackLocale) : @CL_I18N_PRICE.asMap(requestLocale)" />
↪<!-- (1) -->
<form:select path="basePrice" items="${price}" />
```

項番	説明
(1)	リクエストで指定したロケールに対応するコードリストを Map 形式で取得する。 リクエストで指定したロケールがコードリストに定義されていなかった場合、fallbackLocale 変数に設定したロケールで対応するコードリストを Map 形式で取得する。

出力 HTML lang=en

```
<select id="basePrice" name="basePrice">
    <option value="0">unlimited</option>
    <option value="1">Less than \\10,000</option>
    <option value="2">Less than \\20,000</option>
    <option value="3">Less than \\30,000</option>
    <option value="4">Less than \\40,000</option>
    <option value="5">Less than \\50,000</option>
</select>
```

出力 HTML lang=ja

```
<select id="basePrice" name="basePrice">
    <option value="0">上限なし</option>
    <option value="1">10,000 円以下</option>
    <option value="2">20,000 円以下</option>
    <option value="3">30,000 円以下</option>
    <option value="4">40,000 円以下</option>
    <option value="5">50,000 円以下</option>
</select>
```

出力 **HTML lang=undefined**

```
<select id="basePrice" name="basePrice">
    <option value="0">unlimited</option> <!-- (1) -->
    <option value="1">Less than \\10,000</option>
    <option value="2">Less than \\20,000</option>
    <option value="3">Less than \\30,000</option>
    <option value="4">Less than \\40,000</option>
    <option value="5">Less than \\50,000</option>
</select>
```

項番	説明
(1)	リクエストで指定したロケールがコードリストに定義されていなかった場合に、fallbackLocale 変数で指定した」en」 が設定されるため、ロケールが」en」である CL_PRICE_EN コードリストが表示される。

4.9 ファイルアップロード

4.9.1 Overview

本節では、ファイルをアップロードする方法について、説明する。

ファイルのアップロードは、Servlet 3.0 からサポートされたファイルアップロード機能と、Spring Web から提供されているクラスを利用して行う。

> **注釈:** 本節では、Servlet 3.0 でサポートされたファイルアップロード機能を使用しているため、Servlet のバージョンは、3.0 以上であることが前提となる。

> **注釈:** 一部のアプリケーションサーバ上で Servlet 3.0 のファイルアップロード機能を使用すると、リクエストパラメータやファイル名のマルチバイト文字が文字化けすることがある。

version 5.4.2.RELEASE 時点で問題の発生が確認されているアプリケーションサーバは以下の通りである。

- WebLogic 12.1.3
- JBoss EAP 7.0
- JBoss EAP 6.4.0.GA

このうち JBoss EAP 7.0 では、アプリケーションサーバ独自の設定を追加することで問題を回避することができる。詳細は、JBoss EAP 7.0 を利用する際の注意点を参照されたい。なお、JBoss EAP 7.2 では問題が解消され、設定の追加は不要であることを確認している。

その他の問題が発生するアプリケーションサーバを使用する場合は、Commons FileUpload を使用することで問題を回避することができる。Commons FileUpload を使用するための設定方法については、「*Commons FileUpload* を使用したファイルのアップロード」を参照されたい。

警告: 使用するアプリケーションサーバのファイルアップロードの実装が、Apache Commons FileUpload の実装に依存している場合、CVE-2014-0050 および CVE-2016-3092 で報告されているセキュリティの脆弱性が発生する可能性がある。使用するアプリケーションサーバに同様の脆弱性がない事を確認されたい。

Tomcat を使用する場合、7.0 系は 7.0.70 以上、8.5 系は 8.5.3 以上を使用する必要がある。Tomcat 9.0 系ではこの脆弱性は対策済みである。

アップロード処理の基本フロー

Servlet 3.0 からサポートされたファイルアップロード機能と、Spring Web のクラスを使って、ファイルをアップロードする際の基本フローを、以下に示す。

項番	説明
(1)	アップロードするファイルを選択し、アップロードを実行する。
(2)	サーブレットコンテナは、`multipart/form-data` リクエストを受け取り、`org.springframework.web.multipart.support.MultipartFilter` を呼び出す。

Continued on next page

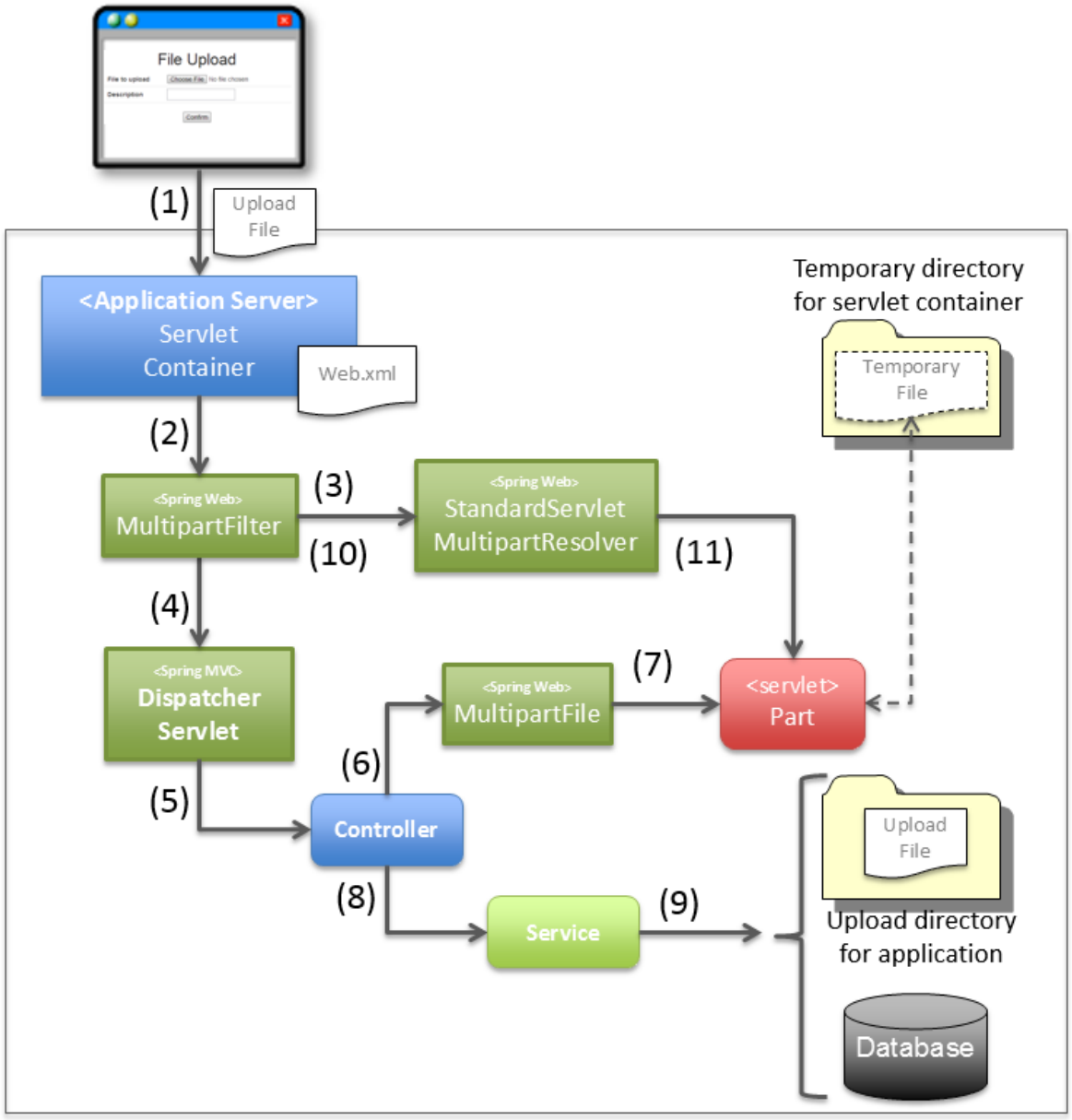
表 4.47 – continued from previous page

項番	説明
(3)	MultipartFilter は、 org.springframework.web.multipart.support.StandardServletMultipartResolver のメソッドを呼び出し、Servlet 3.0 のファイルアップロード機能を、Spring MVC で扱えるようにする。 StandardServletMultipartResolver は、Servlet 3.0 から導入された API(javax.servlet.http.Part) をラップする org.springframework.web.multipart.MultipartFile のオブジェクトを生成する。
(4)	MultipartFilter から DispatcherServlet にフィルタチェーンする。
(5)	DispatcherServlet は、Controller のハンドラメソッドを呼び出す。 (3) で生成された MultipartFile オブジェクトは、 Controller の引数またはフォームオブジェクトに、バインドされる。
(6)	Controller は、 MultipartFile オブジェクトのメソッドを呼び出し、アップロードされたファイルの中身と、メタ情報 (ファイル名など) を取得する。
(7)	MultipartFile は、Servlet 3.0 から導入された Part オブジェクトのメソッドを呼び出し、アップロードされたファイルの中身と、メタ情報 (ファイル名など) を取得し、Controller に返却する。
(8)	Controller は、Service のメソッドを呼び出し、アップロード処理を実行する。 MultipartFile オブジェクトより取得した、ファイルの中身と、メタ情報 (ファイル名など) は、Service のメソッドの引数として、引き渡す。
(9)	Service は、アップロードされたファイルの中身と、メタ情報 (ファイル名など) を、ファイルまたはデータベースに格納する。
(10)	MultipartFilter は、 StandardServletMultipartResolver を呼び出し、Servlet 3.0 のファイルアップロード機能で使用される一時ファイルを削除する。

Continued on next page

表 4.47 – continued from previous page

項番	説明
(11)	`StandardServletMultipartResolver` は、Servlet 3.0 から導入された `Part` オブジェクトのメソッドを呼び出し、ディスクに保存されている一時ファイルを削除する。



注釈: Controller では、Spring Web から提供されている `MultipartFile` オブジェクトに対して処理を行うため、Servlet 3.0 から提供されたファイルアップロード用の API に依存した実装を、排除することができる。

Spring Web から提供されているクラスについて

Spring Web から提供されているファイルアップロード用のクラスについて、説明する。

項番	クラス名	説明
1.	org.springframework.web.multipart.MultipartFile	アップロードされたファイルであることを示すインタフェース。 利用するファイルアップロード機能で扱うファイルオブジェクトを、抽象化する役割をもつ。
2.	org.springframework.web.multipart.support.StandardMultipartHttpServletRequest$StandardMultipartFile	Servlet 3.0 から導入されたファイルアップロード機能用の `MultipartFile` クラス。 Servlet 3.0 から導入された `Part` オブジェクトに、処理を委譲している。
3.	org.springframework.web.multipart.MultipartResolver	`multipart/form-data` リクエストの解析方法を解決するためのインタフェース。 ファイルアップロード機能の、実装に対応する `MultipartFile` オブジェクトを生成する役割をもつ。
4.	org.springframework.web.multipart.support.StandardServletMultipartResolver	Servlet 3.0 から導入されたファイルアップロード機能用の `MultipartResolver` クラス。
5.	org.springframework.web.multipart.support.MultipartFilter	multipart/form-data リクエストの時に、DI コンテナから MultipartResolver を実装するクラスを呼び出し、MultipartFile を生成するクラス。 このクラスを使用しないと、ファイルアップロードで許容する最大サイズを超えた場合に、Servlet Filter の処理内でリクエストパラメータを取得できない。 そのため、本ガイドラインでは MultipartFilter を使用することを推奨している。

ちなみに: 本ガイドラインでは、Servlet 3.0 から導入されたファイルアップロード機能を使うことを前提としているが、Spring Web では、「Apache Commons FileUpload」用の実装クラスも提供している。アップロード処理の実装の違いは、MultipartResolver と、MultipartFile オブジェクトによって吸収されるため、Controller の実装に影響を与えることはない。

4.9.2 How to use

アプリケーションの設定

Servlet 3.0 のアップロード機能を有効化するための設定

Servlet 3.0 のアップロード機能を有効化するために、以下の設定を行う。

- web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.
↪com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0"> <!-- (1) (2) -->

    <servlet>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <!-- omitted -->
        <multipart-config> <!-- (3) -->
            <max-file-size>5242880</max-file-size> <!-- (4) -->
            <max-request-size>27262976</max-request-size> <!-- (5) -->
            <file-size-threshold>0</file-size-threshold> <!-- (6) -->
        </multipart-config>
    </servlet>

    <!-- omitted -->

</web-app>
```

項番	説明
(1)	<web-app>要素の xsi:schemaLocation 属性に、Servlet 3.0 以上の XSD ファイルを指定する。

Continued on next page

表 4.48 – continued from previous page

項番	説明
(2)	<web-app>要素の version 属性に、3.0 以上のバージョンを指定する。
(3)	ファイルアップロードを使用する Servlet の<servlet>要素に、 <multipart-config>要素を追加する。
(4)	アップロードを許可する 1 ファイルの最大バイト数を指定する。 指定がない場合、-1 (制限なし) が設定される。 指定した値を超えた場合、 org.springframework.web.multipart.MultipartException が発生する。 上記例では、 5MB を指定している。
(5)	multipart/form-data リクエストの Content-Length の最大値を指定する。 指定がない場合、-1 (制限なし) が設定される。 指定した値を超えた場合、 org.springframework.web.multipart.MultipartException が発生する。 本パラメータに設定する値は、以下の計算式で算出される値を設定する必要がある。 **(「アップロードを許可する 1 ファイルの最大バイト数」 * 「同時にアップロードを許可するファイル数」) + 「その他のフォーム項目のデータサイズ」 + 「multipart/form-data リクエストのメタ情報サイズ」** 上記例では、 26MB を指定している。 内訳は、25MB(5MB * 5 files) と、1MB(メタ情報のバイト数 + フォーム項目のバイト数) である。
(6)	アップロードされたファイルの中身を、一時ファイルとして保存するかの閾値 (1 ファイルのバイト数) を指定する。 このパラメータを明示的に指定しないと <max-file-size> 要素や <max-request-size> 要素で指定した値が有効にならないアプリケーションサーバが存在するため、デフォルト値 (0) を明示的に指定している。

警告: DoS 攻撃に対する攻撃耐性を高めるため、max-file-size と、max-request-size は、かならず指定すること。

DoS 攻撃については、アップロード機能に対する *DoS* 攻撃を参照されたい。

注釈: デフォルトの設定では、アップロードされたファイルは必ず一時ファイルに出力されるが、<multipart-config>の子要素である<file-size-threshold>要素の設定値によって、出力有無を制御することができる。

```
<!-- omitted -->

<multipart-config>
    <!-- omitted -->
    <file-size-threshold>32768</file-size-threshold> <!-- (7) -->
</multipart-config>

<!-- omitted -->
```

項番	説明
(7)	アップロードされたファイルの中身を、一時ファイルとして保存するかの閾値 (1 ファイルのバイト数) を指定する。 指定がない場合、0 が設定される。 指定値を超えるサイズのファイルがアップロードされた場合、アップロードされたファイルは、 一時ファイルとしてディスクに出力され、リクエストが完了した時点で削除される。 上記例では、 32KB を指定している。

警告: 本パラメータは、以下の点でトレードオフの関係となっているため、**システム特性にあった設定値を指定すること。**

- 設定値を大きくすると、メモリ内で処理が完結するため、処理性能は向上するが、 DoS 攻撃などによって OutOfMemoryError が発生する可能性が高くなる。
- 設定値を小さくすると、メモリの使用率を最小限に抑えることができるため、DoS 攻撃などによって OutOfMemoryError が発生する可能性を抑えることができるが、ディスク IO の発生頻度が高くなるため、性能劣化が発生する可能性が高くなる。

一時ファイルの出力ディレクトリを変更したい場合は、<multipart-config>の子要素であ

る<location>要素にディレクトリパスを指定する。

```
<!-- omitted -->

<multipart-config>
    <location>/tmp</location> <!-- (8) -->
    <!-- omitted -->
</multipart-config>

<!-- omitted -->
```

項番	説明
(8)	一時ファイルを出力するディレクトリのパスを指定する。 省略した場合、アプリケーションサーバの一時ファイルを格納するためのディレクトに出力される。 上記例では、/tmp を指定している。

警告: <location>要素で指定するディレクトリは、アプリケーションサーバ (サーブレットコンテナ) が利用するディレクトリであり、**アプリケーションからアクセスする場所ではない。**

アプリケーションとしてアップロードされたファイルを一時的なファイルとして保存しておきたい場合は、<location>要素で指定するディレクトリとは、別のディレクトリに出力すること。

Servlet Filter の設定

multipart/form-data リクエストの時、ファイルアップロードで許容する最大サイズを超えた場合の動作は、アプリケーションサーバによって異なる。アプリケーションサーバによっては、許容サイズを超えたアップロードの際に発生する MultipartException が検知されず、後述する例外ハンドリングの設定が有効にならない場合がある。

この動作は MiltipartFilter を設定することで回避できるため、本ガイドラインでは MiltipartFilter の設定を前提として説明を行う。
以下に、設定例を示す。

- web.xml

```
<!-- (1) -->
<filter>
    <filter-name>MultipartFilter</filter-name>
    <filter-class>org.springframework.web.multipart.support.MultipartFilter</
↪filter-class>
</filter>
<!-- (2) -->
<filter-mapping>
    <filter-name>MultipartFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

項番	説明
(1)	Servlet Fliter として `MultipartFilter` を定義する。
(2)	`MultipartFilter` を適用する URL のパターンを指定する。

警告: Spring Security 使用時の注意点

Spring Security を使ってセキュリティ対策を行う場合は、`springSecurityFilterChain` より前に定義すること。また、プロジェクト独自で作成する Servlet Filter でリクエストパラメータにアクセスするものがある場合は、その Servlet Filter より前に定義すること。

ただし、`springSecurityFilterChain` より前に定義することで、認証又は認可されていないユーザーからのアップロード (一時ファイル作成) を許容することになる。この動作を回避する方法が Spring Security Reference -Cross Site Request Forgery (CSRF)-の中で紹介されているが、セキュリティ上のリスクを含む回避方法になるため、本ガイドラインでは回避策の適用は推奨していない。

警告: ファイルアップロードの許容サイズを超過した場合の注意点

ファイルアップロードの許容サイズを超過した場合、WebLogic など一部のアプリケーションサーバでは、CSRF トークンを取得する前にサイズ超過のエラーが検知され、CSRF トークンチェックが行われないことがある。

注釈: MultipartResolver のデフォルト呼び出し

`MultipartFilter` を使用すると、デフォルトで `org.springframework.web.`

multipart.support.StandardServletMultipartResolver が呼び出される。StandardServletMultipartResolver は、アップロードされたファイルを org.springframework.web.multipart.MultipartFile として生成し、Controller の引数およびフォームオブジェクトのプロパティとして、受け取ることができるようにする。

例外ハンドリングの設定

許可されないサイズのファイルやマルチパートのリクエストが行われた際に発生する MultipartException の例外ハンドリングの定義を追加する。

MultipartException は、クライアントが指定するファイルサイズに起因して発生する例外なので、クライアントエラー (HTTP レスポンスコード=4xx) として扱うことを推奨する。

例外ハンドリングを個別に追加しないと、システムエラー扱いとなってしまうので、かならず定義を追加すること。

MultipartException をハンドリングするための設定は、MultipartFilter を使用するか否かによって異なる。

MultipartFilter を使用する場合は、サーブレットコンテナの<error-page>機能を使って例外ハンドリングを行う。

以下に、設定例を示す。

- web.xml

```
<error-page>
    <!-- (1) -->
    <exception-type>org.springframework.web.multipart.MultipartException</
↪exception-type>
    <!-- (2) -->
    <location>/WEB-INF/views/common/error/fileUploadError.jsp</location>
</error-page>
```

項番	説明
(1)	ハンドリング対象の例外クラスとして、MultipartException を指定する。
(2)	MultipartException が発生した際に表示するファイルを指定する。 上記例では、/WEB-INF/views/common/error/fileUploadError.jsp を指定している。

- fileUploadError.jsp

```
<%-- (3) --%>
<% response.setStatus(HttpServletResponse.SC_BAD_REQUEST); %>
<!DOCTYPE html>
<html>

    <!-- omitted -->

</html>
```

項番	説明
(3)	HTTP ステータスコードは、HttpServletResponse の API を呼び出して設定する。 上記例では、400(Bad Request) を設定している。 明示的に設定しない場合、HTTP ステータスコードは 500(Internal Server Error) となる。

MultipartFilter を使用しない場合は、SystemExceptionResolver を使用して例外ハンドリングを行う。

以下に、設定例を示す。

- spring-mvc.xml

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">
    <!-- omitted -->
    <property name="exceptionMappings">
        <map>
            <!-- omitted -->
            <!-- (4) -->
            <entry key="MultipartException"
                   value="common/error/fileUploadError" />

        </map>
    </property>
    <property name="statusCodes">
        <map>
            <!-- omitted -->
            <!-- (5) -->
            <entry key="common/error/fileUploadError" value="400" />
        </map>
    </property>
    <!-- omitted -->
</bean>
```

項番	説明
(4)	SystemExceptionResolver の exceptionMappings に、MultipartException が発生した際に表示する View(JSP) の定義を追加する。 上記例では、common/error/fileUploadError を指定している。
(5)	MultipartException が発生した際に応答する HTTP ステータスコードの定義を追加する。 上記例では、400(Bad Request) を指定している。 クライアントエラー (HTTP レスポンスコード = 4xx) を指定することで、 共通ライブラリの例外ハンドリング機能から提供しているクラス (HandlerExceptionResolverLoggingInterceptor) によって出力されるログは、ERROR レベルではなく、WARN レベルとなる。

MultipartException に対する例外コードを設ける場合は、例外コードの設定を追加する。

例外コードは、共通ライブラリのログ出力機能により出力されるログに、出力される。

例外コードは、View(JSP) から参照することもできる。

View(JSP) から例外コードを参照する方法については、システム例外の例外コードを、画面表示する方法を参照されたい。

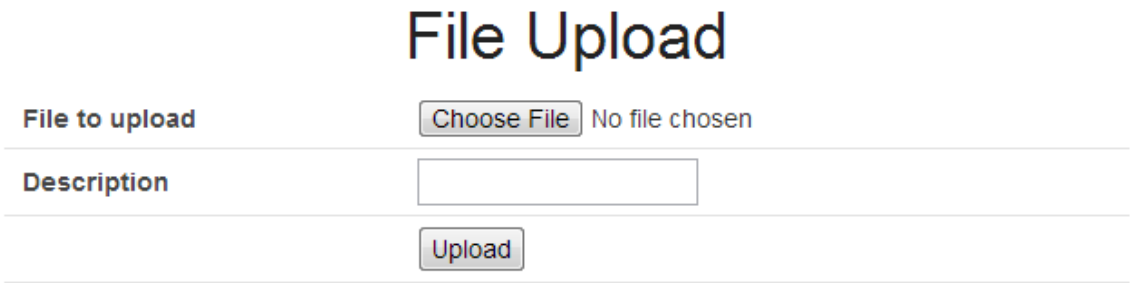
- applicationContext.xml

```
<bean id="exceptionCodeResolver"
    class="org.terasoluna.gfw.common.exception.
↪SimpleMappingExceptionCodeResolver">
    <property name="exceptionMappings">
        <map>
            <!-- (6) -->
            <entry key="MultipartException" value="e.xx.fw.6001" />
            <!-- omitted -->
        </map>
    </property>
    <property name="defaultExceptionCode" value="e.xx.fw.9001" />
    <!-- omitted -->
</bean>
```

項番	説明
(6)	SimpleMappingExceptionCodeResolver の exceptionMappings に、MultipartException が発生した際に適用する、例外コードを追加する。 上記例では、e.xx.fw.6001 を指定している。 個別に定義を行わない場合は、defaultExceptionCode に指定した例外コードが適用される。

単一ファイルのアップロード

単一ファイルをアップロードする方法について、説明する。



単一ファイルの場合は、org.springframework.web.multipart.MultipartFile オブジェクトを、フォームオブジェクトにバインドして受け取る方法と、Controller の引数として直接受け取る 2 つの方法があるが、本ガイドラインでは、フォームオブジェクトにバインドして受け取る方法を推奨する。

その理由は、アップロードされたファイルの単項目チェックを、Bean Validation の仕組みを使って行うことができるためである。

以下に、フォームオブジェクトにバインドして受け取る方法について、説明する。

フォームの実装

```
public class FileUploadForm implements Serializable {

    // omitted

    private MultipartFile file; // (1)

    @NotNull
    @Size(min = 0, max = 100)
    private String description;

    // omitted getter/setter methods.

}
```

項番	説明
(1)	フォームオブジェクトに、org.springframework.web.multipart.MultipartFile のプロパティを定義する。

JSP の実装

```
<form:form
  action="${pageContext.request.contextPath}/article/upload" method="post"
  modelAttribute="fileUploadForm" enctype="multipart/form-data"> <!-- (1) (2)␣
↪-->
  <table>
    <tr>
      <th width="35%">File to upload</th>
      <td width="65%">
        <form:input type="file" path="file" /> <!-- (3) -->
        <form:errors path="file" />
```

```
      </td>
    </tr>
    <tr>
      <th width="35%">Description</th>
      <td width="65%">
        <form:input path="description" />
        <form:errors path="description" />
      </td>
    </tr>
    <tr>
      <td> </td>
      <td><form:button>Upload</form:button></td>
    </tr>
  </table>
</form:form>
```

項番	説明
(1)	<form:form>要素の enctype 属性に、multipart/form-data を指定する。
(2)	<form:form>要素の modelAttribute 属性に、フォームオブジェクトの属性名を指定する。 上記例では、fileUploadForm を指定している。
(3)	<form:input>要素 type 属性に、file を指定し、path 属性に、MultipartFile プロパティ名を指定する。 上記例では、アップロードされたファイルは、FileUploadForm オブジェクトの file プロパティに格納される。

Controller の実装

```
@RequestMapping("article")
@Controller
public class ArticleController {

    @Value("${upload.allowableFileSize}")
    private int uploadAllowableFileSize;

    // omitted

    // (1)
    @ModelAttribute
    public FileUploadForm setFileUploadForm() {
```

```
        return new FileUploadForm();
    }

    // (2)
    @RequestMapping(value = "upload", method = RequestMethod.GET, params =
↪"form")
    public String uploadForm() {
        return "article/uploadForm";
    }

    // (3)
    @RequestMapping(value = "upload", method = RequestMethod.POST)
    public String upload(@Validated FileUploadForm form,
            BindingResult result, RedirectAttributes redirectAttributes) {

        if (result.hasErrors()) {
            return "article/uploadForm";
        }

        MultipartFile uploadFile = form.getFile();

        // (4)
        if (!StringUtils.hasLength(uploadFile.getOriginalFilename())) {
            result.rejectValue(uploadFile.getName(), "e.xx.at.6002");
            return "article/uploadForm";
        }

        // (5)
        if (uploadFile.isEmpty()) {
            result.rejectValue(uploadFile.getName(), "e.xx.at.6003");
            return "article/uploadForm";
        }

        // (6)
        if (uploadAllowableFileSize < uploadFile.getSize()) {
            result.rejectValue(uploadFile.getName(), "e.xx.at.6004",
                    new Object[] { uploadAllowableFileSize }, null);
            return "article/uploadForm";
        }

        // (7)
        // omit processing of upload.

        // (8)
        redirectAttributes.addFlashAttribute(ResultMessages.success().add(
                "i.xx.at.0001"));

        // (9)
        return "redirect:/article/upload?complete";
    }

    @RequestMapping(value = "upload", method = RequestMethod.GET, params =
↪"complete")
```

```
    public String uploadComplete() {
        return "article/uploadComplete";
    }

    // omitted

}
```

項番	説明
(1)	ファイルアップロード用のフォームオブジェクトを、Model に格納するためのメソッド。 上記例では、Model に格納するための属性名は、fileUploadForm となる。
(2)	アップロード画面を表示するためのハンドラメソッド。
(3)	ファイルをアップロードするためのハンドラメソッド。
(4)	アップロードファイルが選択されているかのチェックを行っている。 ファイルが選択されたかチェックする場合は、 MultipartFile#getOriginalFilename メソッドを呼び出し、ファイル名の指定有無で判断する。 上記例では、ファイルが選択されていない場合は、入力チェックエラーとしている。
(5)	空のファイルが選択されているかのチェックを行っている。 選択されたファイルの中身が空でないことをチェックする場合は、 MultipartFile#isEmpty メソッドを呼び出し、中身の存在チェックを行う。 上記例では、 空のファイルが選択されている場合は、入力チェックエラーとしている。
(6)	ファイルのサイズが、許容サイズ内かどうかのチェックを行っている。 選択されたファイルのサイズをチェックする場合は、MultipartFile#getSize メソッドを呼び出し、サイズが許容範囲内かチェックを行う。 上記例では、 ファイルのサイズが許容サイズを超えている場合は、入力チェックエラーとしている。

Continued on next page

表 4.49 – continued from previous page

項番	説明
(7)	アップロード処理を実装する。 上記例では、具体的な実装は省略しているが、共有ディスクやデータベースへ保存する処理を行うことになる。
(8)	要件に応じて、アップロードが成功したことを通知する、処理結果メッセージを格納する。
(9)	アップロード処理完了後の画面表示は、リダイレクトして表示する。

注釈: 重複アップロードの防止

ファイルのアップロードを行う場合は、PRG パターンによる画面遷移と、トランザクショントークンチェックを行うことを推奨する。PRG パターンによる画面遷移と、トランザクショントークンチェックを行うことで、重複送信に伴う、同一ファイルのアップロードを防ぐことができる。

重複送信の防止方法について、詳細は、二重送信防止を参照されたい。

注釈: **MultipartFile** について

MultipartFile には、アップロードされたファイルを操作するためのメソッドが用意されている。各メソッドの利用方法については、MultipartFile クラスの JavaDoc を参照されたい。

ファイルアップロードの Bean Validation

上記の実装例では、アップロードファイルのバリデーションを Controller の処理として行っていたが、ここでは、Bean Validation の仕組みを使ってバリデーションする方法について説明する。

バリデーションの詳細は、入力チェックを参照されたい。

注釈: Bean Validation の仕組みでチェックすることで、Controller の処理をシンプルに保つことができるため、Bean Validation の仕組みを使うことを推奨する。

ファイルが選択されていることを検証するためのバリデーションの実装

```
// (1)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Constraint(validatedBy = UploadFileRequiredValidator.class)
public @interface UploadFileRequired {
    String message() default "{com.examples.upload.UploadFileRequired.message}
↪";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        UploadFileRequired[] value();
    }

}
```

```
// (2)
public class UploadFileRequiredValidator implements
    ConstraintValidator<UploadFileRequired, MultipartFile> {

    @Override
    public void initialize(UploadFileRequired constraint) {
    }

    @Override
    public boolean isValid(MultipartFile multipartFile,
        ConstraintValidatorContext context) {
        return multipartFile != null &&
            StringUtils.hasLength(multipartFile.getOriginalFilename());
    }

}
```

項番	説明
(1)	ファイルが、選択されていることを検証するための、アノテーションを作成する。
(2)	ファイルが、選択されていることを検証するための、実装を行うクラスを作成する。

ファイルが空でないことを検証するためのバリデーションの実装

```
// (3)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Constraint(validatedBy = UploadFileNotEmptyValidator.class)
public @interface UploadFileNotEmpty {
    String message() default "{com.examples.upload.UploadFileNotEmpty.message}
↪";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        UploadFileNotEmpty[] value();
    }

}
```

```
// (4)
public class UploadFileNotEmptyValidator implements
    ConstraintValidator<UploadFileNotEmpty, MultipartFile> {

    @Override
    public void initialize(UploadFileNotEmpty constraint) {
    }

    @Override
    public boolean isValid(MultipartFile multipartFile,
        ConstraintValidatorContext context) {
        if (multipartFile == null ||
            !StringUtils.hasLength(multipartFile.getOriginalFilename())) {
            return true;
        }
        return !multipartFile.isEmpty();
    }

}
```

項番	説明
(3)	ファイルが、空でないことを検証するための、アノテーションを作成する。
(4)	ファイルが、空でないことを検証するための、実装を行うクラスを作成する。

ファイルのサイズが許容サイズ内であることを検証するためのバリデーションの実装

```
// (5)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Constraint(validatedBy = UploadFileMaxSizeValidator.class)
public @interface UploadFileMaxSize {
    String message() default "{com.examples.upload.UploadFileMaxSize.message}
↪";
    long value() default (1024 * 1024);
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        UploadFileMaxSize[] value();
    }

}
```

```
// (6)
public class UploadFileMaxSizeValidator implements
    ConstraintValidator<UploadFileMaxSize, MultipartFile> {

    private UploadFileMaxSize constraint;

    @Override
    public void initialize(UploadFileMaxSize constraint) {
        this.constraint = constraint;
    }

    @Override
    public boolean isValid(MultipartFile multipartFile,
        ConstraintValidatorContext context) {
        if (constraint.value() < 0 || multipartFile == null) {
            return true;
        }
        return multipartFile.getSize() <= constraint.value();
    }

}
```

項番	説明
(5)	ファイルのサイズが、許容サイズ内であることを検証するための、アノテーションを作成する。
(6)	ファイルのサイズが、許容サイズ内であることを検証するための、実装を行うクラスを作成する。

フォームの実装

```
public class FileUploadForm implements Serializable {

    // omitted

    // (7)
    @UploadFileRequired
    @UploadFileNotEmpty
    @UploadFileMaxSize
    private MultipartFile file;

    @NotNull
    @Size(min = 0, max = 100)
    private String description;

    // omitted getter/setter methods.

}
```

項番	説明
(7)	MultipartFile のフィールドに、アップロードファイルのバリデーションを行うための、アノテーションを付与する。

Controller の実装

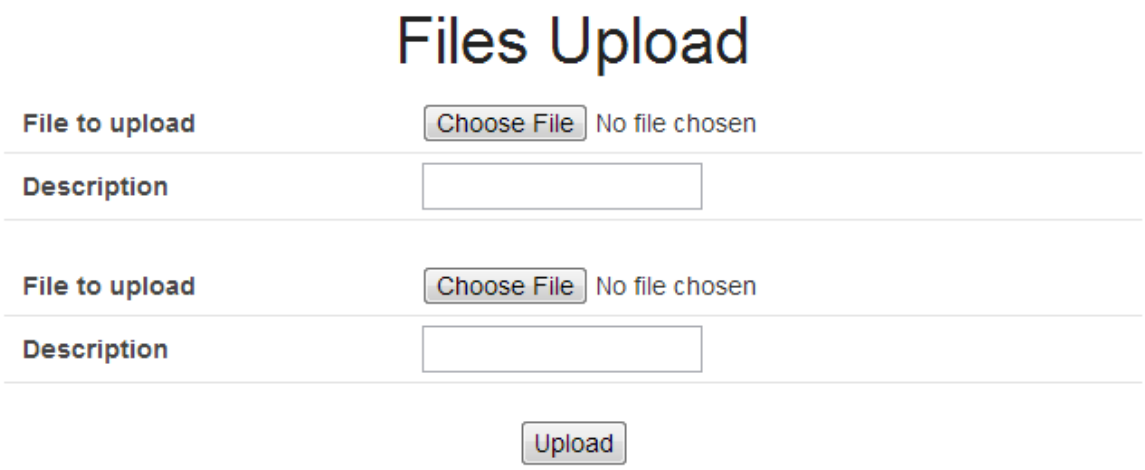
```
@RequestMapping(value = "upload", method = RequestMethod.POST)
public String uploadFile(@Validated FileUploadForm form,
        BindingResult result, RedirectAttributes redirectAttributes) {

    // (8)
    if (result.hasErrors()) {
        return "article/uploadForm";
```

```
    }

    MultipartFile uploadFile = form.getFile();

    // omit processing of upload.

    redirectAttributes.addFlashAttribute(ResultMessages.success().add(
            "i.xx.at.0001"));

    return "redirect:/article/upload";
}
```

項番	説明
(8)	アップロードファイルのバリデーションの結果は、BindingResult に格納される。

複数ファイルのアップロード

複数ファイルを同時にアップロードする方法について説明する。



複数ファイルを同時にアップロードする場合は、org.springframework.web.multipart.MultipartFile オブジェクトを、フォームオブジェクトにバインドして受け取る必要がある。

以降の説明では、単一ファイルのアップロードと重複する箇所の説明については、省略する。

フォームの実装

```
// (1)
public class FileUploadForm implements Serializable {

    // omitted
```

```
    @UploadFileRequired
    @UploadFileNotEmpty
    @UploadFileMaxSize
    private MultipartFile file;

    @NotNull
    @Size(min = 0, max = 100)
    private String description;

    // omitted getter/setter methods.

}
```

```
public class FilesUploadForm implements Serializable {

    // omitted

    @Valid // (2)
    private List<FileUploadForm> fileUploadForms; // (3)

    // omitted getter/setter methods.

}
```

項番	説明
(1)	ファイル単位の情報 (アップロードファイル自体と、関連するフォーム項目) を保持するクラス。 上記例では、単一ファイルのアップロードの説明で作成したフォームオブジェクトを再利用している。
(2)	リスト内で保持しているオブジェクトに対して、Bean Validation による入力チェックを行うために、`@Valid` アノテーションを付与する。
(3)	ファイル単位の情報 (アップロードファイル自体と、関連するフォーム項目) を保持するオブジェクトを、List 型のプロパティとして定義する。

注釈: ファイルのみアップロードする場合は、`MultipartFile` オブジェクトを、List 型のプロパティとして定義することもできるが、Bean Validation を使用してアップロードファイルの入力チェックを行う場合は、ファイル単位の情報を保持するオブジェクトを、List 型のプロパティとして定義する方

が相性がよい。

JSP の実装

```
<form:form
  action="${pageContext.request.contextPath}/article/uploadFiles" method="post
↪"
  modelAttribute="filesUploadForm" enctype="multipart/form-data">
  <c:forEach var="i" begin="0" end="1" step="1">
    <table>
      <tr>
        <th width="35%">File to upload</th>
        <td width="65%">
          <form:input type="file" path="fileUploadForms[${i}].file" /> <!--␣
↪(1) -->
          <form:errors path="fileUploadForms[${i}].file" />
        </td>
      </tr>
      <tr>
        <th width="35%">Description</th>
        <td width="65%">
          <form:input path="fileUploadForms[${i}].description" />
          <form:errors  path="fileUploadForms[${i}].description" />
        </td>
      </tr>
    </table>
  </c:forEach>
  <div>
    <form:button>Upload</form:button>
  </div>
</form:form>
```

項番	説明
(1)	アップロードファイルをバインドする List 内の位置を指定する。 バインドするリスト内の位置は、[] の中に指定する。開始位置は、」0」 開始となる。

Controller の実装

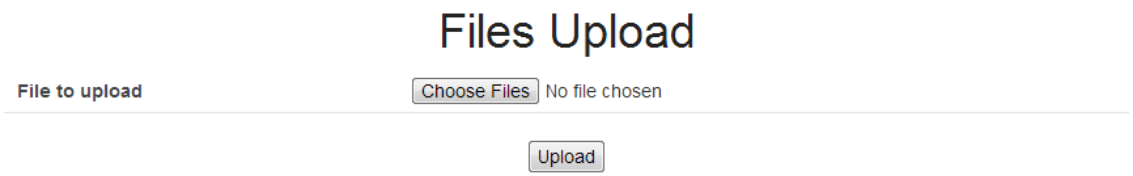
```
@RequestMapping(value = "uploadFiles", method = RequestMethod.POST)
public String uploadFiles(@Validated FilesUploadForm form,
        BindingResult result, RedirectAttributes redirectAttributes) {

    if (result.hasErrors()) {
        return "article/uploadForm";
    }
```

```
    // (1)
    for (FileUploadForm fileUploadForm : form.getFileUploadForms()) {

        MultipartFile uploadFile = fileUploadForm.getFile();

        // omit processing of upload.

    }

    redirectAttributes.addFlashAttribute(ResultMessages.success().add(
            "i.xx.at.0001"));

    return "redirect:/article/upload?complete";
}
```

項番	説明
(1)	ファイル単位の情報 (アップロードファイル自体と関連するフォーム項目) を保持するオブジェクトから `MultipartFile` を取得し、アップロード処理を実装する。 上記例では、具体的な実装は省略しているが、共有ディスクやデータベースへ保存する処理を行うことになる。

HTML5 の multiple 属性を使った複数ファイルのアップロード

HTML5 でサポートされた input タグの multiple 属性を使用して、複数ファイルを同時にアップロードする方法について説明する。



以降の説明では、単一ファイルのアップロード及び複数ファイルのアップロードと重複する箇所の説明については、省略する。

フォームの実装

HTML5 の input タグの multiple 属性を使用して、複数ファイルを同時にアップロードする場合は、`org.springframework.web.multipart.MultipartFile` オブジェクトのコレクションを、フォームオブジェクトにバインドして受け取る必要がある。

```
// (1)
public class FilesUploadForm implements Serializable {

    // omitted

    // (2)
    @UploadFileNotEmpty
    private List<MultipartFile> files;

    // omitted getter/setter methods.

}
```

項番	説明
(1)	複数のアップロードファイルを保持するためのフォームオブジェクト。
(2)	`MultipartFile` クラスをリストとして宣言する。 上記例では、入力チェックとして、ファイルが空でないことを検証するためのアノテーションを指定している。 本来は他の必須チェックやファイルのサイズチェックなども必要であるが、上記例では割愛している。

Validator の実装

コレクションに格納されている複数の `MultipartFile` オブジェクトに対して入力チェックを行う場合は、コレクション用の Validator を実装する必要がある。

以下では、単一ファイル用に作成した Validator を利用してコレクション用の Validator を作成する方法について説明する。

```
// (1)
public class UploadFileNotEmptyForCollectionValidator implements
    ConstraintValidator<UploadFileNotEmpty, Collection<MultipartFile>> {

    // (2)
    private final UploadFileNotEmptyValidator validator =
        new UploadFileNotEmptyValidator();

    // (3)
    @Override
    public void initialize(UploadFileNotEmpty constraintAnnotation) {
        validator.initialize(constraintAnnotation);
    }
```

```
    // (4)
    @Override
    public boolean isValid(Collection<MultipartFile> values,
            ConstraintValidatorContext context) {
        for (MultipartFile file : values) {
            if (!validator.isValid(file, context)) {
                return false;
            }
        }
        return true;
    }

}
```

項番	説明
(1)	全てのファイルが空でないことを検証するための実装を行うクラス。 検証対象となる値の型として、 `Collection<MultipartFile>` を指定する。
(2)	実際の処理は単一ファイル用の Validator に委譲するため、単一ファイル用の Validator のインスタンスを作成しておく。
(3)	Validator を初期化する。 上記例では、実際の処理を行う単一ファイル用の Validator の初期化を行っている。
(4)	全てのファイルが空でないことを検証する。 上記例では、単一ファイル用の Validator のメソッドを呼び出して、 1 ファイルずつ検証を行っている。

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Constraint(validatedBy =
    {UploadFileNotEmptyValidator.class,
     UploadFileNotEmptyForCollectionValidator.class}) // (5)
public @interface UploadFileNotEmpty {

    // omitted

}
```

項番	説明
(5)	複数のファイルに対してチェックを行う Validator クラスを、検証用アノテーションに追加する。 `@Constraint` アノテーションの validatedBy 属性に、(1) で作成したクラスを指定する。 こうすることで、`@UploadFileNotEmpty` アノテーションを付与したプロパティに対する妥当性チェックを行う際に、(1) で作成したクラスが実行される。

JSP の実装

```
<form:form
  action="${pageContext.request.contextPath}/article/uploadFiles" method="post
↪"
  modelAttribute="filesUploadForm" enctype="multipart/form-data">
  <table>
    <tr>
      <th width="35%">File to upload</th>
      <td width="65%">
        <form:input type="file" path="files" multiple="multiple" /> <!-- (1) -
↪->
        <form:errors path="files" />
      </td>
    </tr>
  </table>
  <div>
    <form:button>Upload</form:button>
  </div>
</form:form>
```

項番	説明
(1)	path 属性には フォームオブジェクトのプロパティ名を指定し、 multiple 属性を指定する。 multiple 属性を指定すると、HTML5 をサポートしているブラウザで複数のファイルを選択しアップロードすることができる。

Controller の実装

```
@RequestMapping(value = "uploadFiles", method = RequestMethod.POST)
public String uploadFiles(@Validated FilesUploadForm form,
        BindingResult result, RedirectAttributes redirectAttributes) {
    if (result.hasErrors()) {
        return "article/uploadForm";
    }
```

```
    // (1)
    for (MultipartFile file : form.getFiles()) {

        // omit processing of upload.

    }

    redirectAttributes.addFlashAttribute(ResultMessages.success().add(
            "i.xx.at.0001"));

    return "redirect:/article/upload?complete";
}
```

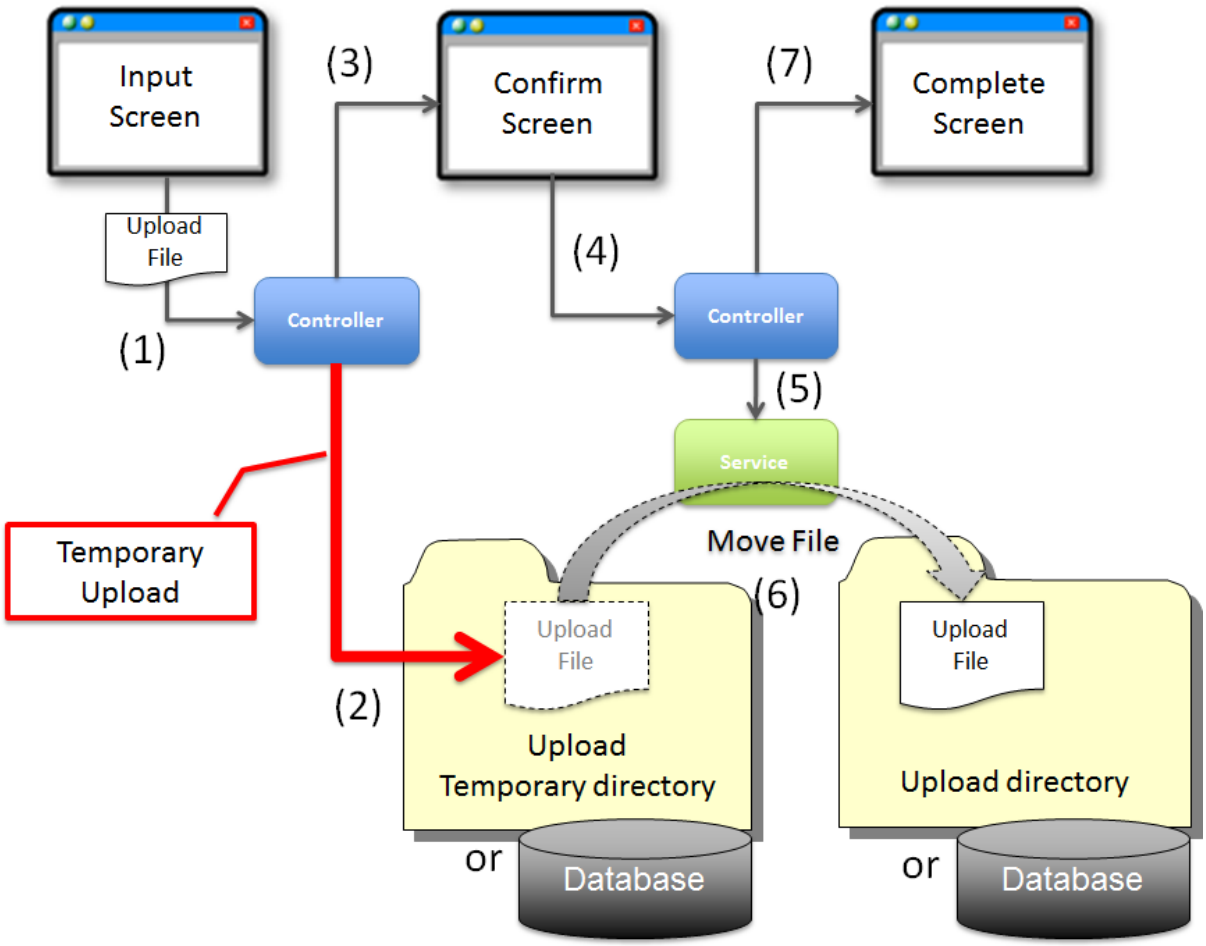
項番	説明
(1)	フォームオブジェクトから MultipartFile オブジェクトが格納されているリストを取得し、アップロード処理を実装する。 上記例では、具体的な実装は省略しているが、共有ディスクやデータベースへ保存する処理を行うことになる。

仮アップロード

アップロード結果の確認画面など、画面遷移の途中でファイルをアップロードする場合、仮アップロードという考え方が必要になる。

注釈: MultipartFile オブジェクトで保持しているファイルの中身は、アップロードしたリクエストが完了した時点で消滅する可能性がある。そのため、ファイルの中身についてリクエストを跨いで扱いたい場合は、MultipartFile オブジェクトで保持しているファイルの中身や、メタ情報 (ファイル名など) をファイルやフォームに退避する必要がある。

MultipartFile オブジェクトで保持しているファイルの中身は、下記処理フローの (3) が完了した時点で、消滅する。



項番	説明
(1)	入力画面にて、アップロードするファイルを選択し、確認画面に遷移するためのリクエストを送信する。
(2)	Controller は、アップロードされたファイルの中身を、アプリケーション用の仮ディレクトリに一時保存する。
(3)	Controller は、確認画面の View 名を返却し、確認画面に遷移する。
(4)	確認画面にて、処理を実行するためのリクエストを送信する。

Continued on next page

表 4.50 – continued from previous page

項番	説明
(5)	Controller は、Service のメソッドを呼び出し、処理を実行する。
(6)	Service は、仮ディレクトリに格納されている一時ファイルを、本ディレクトリまたはデータベースに移動する。
(7)	Controller は、完了画面を表示するための View 名を返却し、完了画面に遷移する。

> **注釈:** 仮アップロードの処理は、アプリケーション層の役割なので、Controller 又は Helper クラスで実装することになる。

Controller の実装

以下に、アップロードされたファイルを仮ディレクトリに一時保存する実装例を示す。

```
@Component
public class UploadHelper {

    // (2)
    @Value("${app.upload.temporaryDirectory}")
    private File uploadTemporaryDirectory;

    // (1)
    public String saveTemporaryFile(MultipartFile multipartFile)
        throws IOException {

        String uploadTemporaryFileId = UUID.randomUUID().toString();
        File uploadTemporaryFile =
            new File(uploadTemporaryDirectory, uploadTemporaryFileId);

        // (2)
        FileUtils.copyInputStreamToFile(multipartFile.getInputStream(),
                uploadTemporaryFile);

        return uploadTemporaryFileId;
    }

}
```

項番	説明
(1)	仮アップロードを行うためのメソッドを Helper クラスに作成する。 ファイルアップロードを行う処理が複数ある場合は、共通的な Helper メソッドを用意し、仮アップロード処理を共通化することを推奨する。
(2)	アップロードしたファイルを一時ファイルとして保存する。 上記例では、`org.apache.commons.io.FileUtils` クラスの copyInputStreamToFile メソッドを呼び出し、アップロードしたファイルの中身をファイルに保存している。

```
// omitted

@Inject
UploadHelper uploadHelper;

@RequestMapping(value = "upload", method = RequestMethod.POST, params =
↪"confirm")
public String uploadConfirm(@Validated FileUploadForm form,
        BindingResult result) throws IOException {

    if (result.hasErrors()) {
        return "article/uploadForm";
    }

    // (3)
    String uploadTemporaryFileId = uploadHelper.saveTemporaryFile(form
            .getFile());

    // (4)
    form.setUploadTemporaryFileId(uploadTemporaryFileId);
    form.setFileName(form.getFile().getOriginalFilename());

    return "article/uploadConfirm";
}
```

項番	説明
(3)	アップロードファイルを一時保存するための Helper メソッドを呼び出す。 上記例では、一時保存したファイルを識別するための ID が Helper メソッドの返り値として返却される。
(4)	アップロードしたファイルのメタ情報（ファイルを識別するための ID、ファイル名など）をフォームオブジェクトに格納する。 上記例では、アップロードファイルのファイル名と一時保存したファイルを識別するための ID をフォームオブジェクトに格納している。

注釈: 仮ディレクトリのディレクトリは、アプリケーションをデプロイする環境によって異なる可能性があるため、外部プロパティから取得すること。外部プロパティの詳細については、プロパティ管理を参照されたい。

> **警告:** 上記例では、アプリケーションサーバ上のローカルディスクに一時保存する例としているが、アプリケーションサーバがクラスタ化されている場合は、データベース又は共有ディスクに保存する必要がでてくるので、非機能要件も考慮して保存先を設計する必要がある。
>
> データベースに保存する場合は、トランザクション管理が必要となるため、 データベースに保存する処理を Service のメソッドに委譲することになる。

4.9.3 How to extend

仮アップロード時の不要ファイルの Housekeeping

仮アップロードの仕組みを使用してファイルのアップロードを行う場合、仮ディレクトリに不要なファイルが残るケースがある。
具体的には、以下のようなケースである。

- 仮アップロード後の画面操作を中止した場合
- 仮アップロード後の画面操作中にシステムエラーが発生した場合
- 仮アップロード後の画面操作中にサーバが停止した場合
- etc …

> **警告:** 不要なファイルを残したままにすると、ディスクを圧迫する可能性があるため、必ず不要なファイルを削除する仕組みを用意すること。

本ガイドラインでは、Spring Framework から提供されている「Task Scheduler」機能を使用して、不要なファイルを削除する方法について説明する。「Task Scheduler」の詳細については、公式リファレンスの」Task Execution and Scheduling」を参照されたい。

> **注釈:** ガイドラインとしては、Spring Framework から提供されている「Task Scheduler」機能を使用する方法について説明するが、使用を強制するものではない。実際のプロジェクトでは、インフラチームによって不要なファイルを削除するバッチアプリケーション (Shell アプリケーション) が提供されるケースがある。その場合は、インフラチーム作成のバッチアプリケーションを使用して不要なファイルを削除することを推奨する。

不要ファイルを削除するコンポーネントクラスの実装

不要なファイルを削除するコンポーネントクラスを実装する。

```
package com.examples.common.upload;

import java.io.File;
import java.util.Collection;
import java.util.Date;

import javax.inject.Inject;

import org.apache.commons.io.FileUtils;
import org.apache.commons.io.filefilter.FileFilterUtils;
import org.apache.commons.io.filefilter.IOFileFilter;
import org.springframework.beans.factory.annotation.Value;
import org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory;

// (1)
public class UnnecessaryFilesCleaner {

    @Inject
    JodaTimeDateFactory dateFactory;

    @Value("${app.upload.temporaryFileSavedPeriodMinutes}")
    private int savedPeriodMinutes;

    @Value("${app.upload.temporaryDirectory}")
    private File targetDirectory;

    // (2)
    public void cleanup() {

        // calculate cutoff date.
        Date cutoffDate = dateFactory.newDateTime().minusMinutes(
                savedPeriodMinutes).toDate();

        // collect target files.
        IOFileFilter fileFilter = FileFilterUtils.ageFileFilter(cutoffDate);
        Collection<File> targetFiles = FileUtils.listFiles(targetDirectory,
                fileFilter, null);

        if (targetFiles.isEmpty()) {
            return;
        }

        // delete files.
        for (File targetFile : targetFiles) {
            FileUtils.deleteQuietly(targetFile);
        }

    }

}
```

項番	説明
(1)	不要なファイルを削除するためのコンポーネントクラスを作成する。
(2)	不要なファイルを削除するメソッドを実装する。 上記例では、ファイルの最終更新日時から、一定期間更新がないファイルを、不要ファイルとして削除している。

注釈: 削除対象ファイルが格納されているディレクトリのパスや、削除基準となる時間などは、アプリケーションをデプロイする環境によって異なる可能性があるため、外部プロパティから取得すること。外部プロパティの詳細については、プロパティ管理を参照されたい。

不要ファイルを削除する処理のスケジューリング設定

不要ファイルを削除する POJO クラスを、bean 登録とタスクスケジュールの設定を行う。

- applicationContext.xml

```
<!-- omitted -->

<!-- (3) -->
<bean id="uploadTemporaryFileCleaner"
    class="com.examples.common.upload.UnnecessaryFilesCleaner" />

<!-- (4) -->
<task:scheduler id="fileCleanupTaskScheduler" />

<!-- (5) -->
<task:scheduled-tasks scheduler="fileCleanupTaskScheduler">
    <!-- (6)(7)(8) -->
    <task:scheduled ref="uploadTemporaryFileCleaner"
                    method="cleanup"
                    cron="${app.upload.temporaryFilesCleaner.cron}"/>
</task:scheduled-tasks>

<!-- omitted -->
```

項番	説明
(3)	不要ファイルを削除する POJO クラスを bean 登録する。 上記例では、 `uploadTemporaryFileCleaner` という ID で登録している。
(4)	不要ファイルを削除する処理を、実行するためのタスクスケジューラの bean を、登録する。 上記例では、pool-size 属性を省略しているため、このタスクスケジュールは、シングルスレッドでタスクを実行する。 複数のタスクを同時に実行する必要がある場合は、 pool-size 属性に任意の数字を指定すること。
(5)	不要ファイルを削除するタスクスケジューラに、タスクを追加する。 上記例では、(4) で bean 登録したタスクスケジューラに対して、タスクを追加している。
(6)	ref 属性に、不要ファイルを削除する処理が実装されている bean を、指定する。 上記例では、(3) で登録した bean を指定している。
(7)	method 属性に、不要ファイルを削除する処理が実装されているメソッド名を、指定する。 上記例では、(3) で登録した bean の cleanup メソッドを指定している。
(8)	cron 属性に、不要ファイルを削除する処理の実行タイミングを指定する。 上記例では、外部プロパティより cron 定義を取得している。

注釈: cron 属性の設定値は、「秒 分 時 月 年 曜日」の形式で指定する。

設定例）

- `0 */15 * * * *` : 毎時 0 分,15 分,30 分,45 分に実行される。
- `0 0 * * * *` : 毎時 0 分に実行される。
- `0 0 9-17 * * MON-FRI` : 平日 9 時～17 時の間の毎時 0 分に実行される。

cron の指定値の詳細については、CronSequenceGenerator の JavaDoc を参照されたい。

実行タイミングは、アプリケーションをデプロイする環境によって異なる可能性があるため、外部プロパティから取得すること。外部プロパティの詳細については、プロパティ管理を参照されたい。

ちなみに： 上記例では、タスクの実行トリガーとして cron を使用しているが、cron 以外に、fixed-delay と fixed-rate が、デフォルトで用意されているので、要件に応じて使い分けること。

デフォルトで用意されているトリガーでは要件を満たせない場合は、trigger 属性に `org.springframework.scheduling.Trigger` を実装した bean を指定することで、独自のトリガーを設けることもできる。

4.9.4 Appendix

ファイルアップロードに関するセキュリティ問題への考慮

ファイルのアップロード機能を提供する場合、以下のようなセキュリティ問題を考慮する必要がある。

1. アップロード機能に対する *DoS* 攻撃
2. アップロードしたファイルを *Web* サーバ上で実行する攻撃
3. ディレクトリトラバーサル攻撃

以下に、対策方針について説明する。

アップロード機能に対する DoS 攻撃

アップロード機能に対する DoS 攻撃とは、巨大なサイズのファイルを連続してアップロードしてサーバに対して負荷を掛けることで、サーバのダウンや、レスポンス速度の低下を狙った攻撃方法のことである。

アップロード可能なファイルのサイズに制限がない場合や、マルチパートリクエストのサイズに制限がない場合、DoS 攻撃への耐性が脆弱となる。

DoS 攻撃の耐性を高めるためには、アプリケーションの設定で説明した`<multipart-config>`要素を用いて、リクエストのサイズ制限を設ける必要がある。

アップロードしたファイルを Web サーバ上で実行する攻撃

アップロードしたファイルを Web サーバ上で実行する攻撃とは、Web サーバ (アプリケーションサーバ) で実行可能なスクリプトファイル (php, asp, aspx, jsp など) をアップロードし実行することで、Web サーバ内のファイルの閲覧・改竄・削除を行う攻撃方法のことである。

また、Web サーバを踏み台とすることで、Web サーバと同一ネットワーク上に存在する別のサーバに対して、攻撃することもできる。

この攻撃への対策方法は、以下の通りである。

- アップロードされたファイルを、Web サーバ (アプリケーションサーバ) 上の公開ディレクトリに配置せず、ファイルの中身を表示するための処理を経由して、アップロードしたファイルの中身を閲覧させる。
- アップロード可能なファイルの拡張子を制限し、Web サーバ (アプリケーションサーバ) で実行可能なスクリプトファイルが、アップロードされないようにする。

いずれかの対策を行うことで攻撃を防ぐことができるが、両方とも対策しておくことを推奨する。

ディレクトリトラバーサル攻撃

ディレクトリトラバーサル攻撃とは、」../」 などの文字列が含まれる入力を用いてファイルシステムにアクセスすることにより、サーバ上の本来アクセスさせるべきでないファイルにアクセスされてしまう攻撃である。

例えば、ユーザからアップロードされたファイルをサーバ上の所定のディレクトリに配置する Web アプリケーションでは、実装方法によっては」../../../../somewhere/attack」 というファイル名のファイルがアップロードされた際に所定外のディレクトリにファイルが配置されてしまう。

その場合、攻撃者からアップロードされたファイルによってサーバ上のファイルが改ざんされてしまう恐れがある。

ファイルアップロード機能を提供する場合の他、ファイルダウンロード機能を提供する際にもディレクトリトラバーサル攻撃のリスクがある。

これは例えば、ユーザからの入力されたファイル名に従ってファイルをダウンロードする Web アプリケーションにおいて、」../../../../etc/passwd」 と入力されることで攻撃者に」/etc/passwd」 の内容を取得されてしまうといった攻撃が考えられる。

この攻撃への対策方法は、以下の通りである。

- アップロードされたファイルをサーバ上に保存する際には、オリジナルのファイル名やユーザからの入力値を使用せず、別の名前で保存する。オリジナルのファイル名についてはサーバ上のファイル名との対応関係を DB 等の外部に保存するなど、実際のファイルアクセスに利用されない形で保存しておく。

- サーバ上のファイルにアクセスさせる際は、実際のファイル名ではなくリクエスト用の識別名を介してリクエストさせ、サーバ側で対応するファイル名に変換する。例えば、実際のファイル名」file_A」, 「file_B」 に対してそれぞれ」id01」, 「id02」 という識別子を対応させ、クライアント側から」id01」へのリクエストがあればサーバ側で対応する」file_A」 というファイル名に変換してアクセスする。

ちなみに: 入力されたファイルパスを正規化（「./」 や 「../」 等、ファイルシステム上で特別な意味を持つ文字列を含まない形式に展開すること）し、あらかじめ定めておいたパスと前方一致するかどうかをチェックすることでアクセスを許可するかどうか判断するという対策方法も考えられる。しかしながら、入力値のエンコーディングや OS ごとのパス形式の違いを考慮すると、あらゆる場合において正しく正規化されるかどうかを確認することは困難である。そのため、基本的にはユーザからの入力値を使用したファイルシステムへのアクセスは回避することが望ましい。

Commons FileUpload を使用したファイルのアップロード

一部のアプリケーションサーバ上で Servlet 3.0 のファイルアップロード機能を使用すると、リクエストパラメータやファイル名のマルチバイト文字が文字化けすることがある。

具体例としては、WebLogic 12.1.3 で Servlet 3.0 のファイルアップロード機能を使用すると、ファイルと一緒に送信するフィールドのマルチバイト文字が文字化けすることが確認されている。なお、WebLogic 12.2.1 では修正されている。

この問題は、Commons FileUpload を使用することで回避できるため、問題が発生する特定環境向けの暫定対処として、Commons FileUpload を使用したファイルのアップロードについて説明する。問題が発生しない環境での Commons FileUpload の使用は推奨しない。

Commons FileUpload を使用する場合は以下の設定を行う。

xxx-web/pom.xml

```
<!-- (1) -->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
</dependency>
```

項番	説明
(1)	commons-fileupload への依存関係を追加する。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

警告: Apache Commons FileUpload を使用する場合、CVE-2014-0050 および CVE-2016-3092 で報告されているセキュリティの脆弱性が発生する可能性がある。使用する Apache Commons FileUpload のバージョンに脆弱性がない事を確認されたい。

Apache Commons FileUpload を使用する場合、1.3.2 以上を使用する必要がある。

なお、TERASOLUNA Server Framework for Java version 5.4.2.RELEASE が準拠している Spring IO Platform Brussels-SR17.RELEASE で管理されているバージョンを使用すれば、CVE-2014-0050 および CVE-2016-3092 で報告されている脆弱性は発生しない。意図的に Apache Commons FileUpload のバージョンを変更する場合は、当該脆弱性が対処されているバージョンを指定すること。

xxx-web/src/main/resources/META-INF/spring/applicationContext.xml

```
<!-- (1) -->
<bean id="filterMultipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="maxUploadSize" value="10240000" /><!-- (2) -->
</bean>

<!-- ... -->
```

項番	説明
(1)	Commons FileUpload を使用した `MultipartResolver` 実装である `CommonsMultipartResolver` の bean 定義を行う。 bean ID には `filterMultipartResolver` を指定する。
(2)	ファイルアップロードで許容する最大サイズを設定する。 Commons FileUpload の場合、最大値は HTTP ヘッダを含めたリクエスト全体のサイズであることに注意すること。 また、**デフォルト値は-1(無制限) なので、必ず値を設定すること。** その他のプロパティは JavaDoc を参照されたい。

> **警告:** Commons Fileupload を使用する場合は、`MultipartResolver` の定義を `spring-mvc.xml` ではなく、`applicationContext.xml` に行う必要がある。`spring-mvc.xml` に定義がある場合は削除すること。

`xxx-web/src/main/webapp/WEB-INF/web.xml`

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/
↪ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <servlet>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
↪class>
        <!-- omitted -->
        <!-- (1) -->
        <!-- <multipart-config>...</multipart-config> -->
    </servlet>

    <!-- (2) -->
    <filter>
```

```
        <filter-name>MultipartFilter</filter-name>
        <filter-class>org.springframework.web.multipart.support.MultipartFilter</
↪filter-class>
    </filter>
    <filter-mapping>
        <filter-name>MultipartFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- omitted -->

</web-app>
```

項番	説明
(1)	Commons FileUpload を使用する場合、Servlet 3.0 のアップロード機能を無効にする必要がある。 DispatcherServlet の定義の中に<multipart-config>要素がある場合は、必ず削除すること。
(2)	Commons Fileupload を使用する場合、Spring Security を使ったセキュリティ対策を有効にするために MultipartFilter を定義する必要がある。 MultipartFilter のマッピング定義は、springSecurityFilterChain(Spring Security の Servlet Filter) の定義より前に行うこと。

ちなみに: MultipartFilter は、DI コンテナ (applicationContext.xml) から filterMultipartResolver という bean ID で登録されている MultipartResolver を取得して、ファイルアップロード処理を行う仕組みになっている。

4.10 ファイルダウンロード

4.10.1 Overview

本節では、Spring でクライアントにサーバからファイルをダウンロードする機能について説明する。

Spring MVC の View で、ファイルのレンダリングを行うことを推奨する。

注釈: コントローラクラスで、ファイルレンダリングのロジックを持たせることは推奨しない。

理由としては、コントローラの役割から逸脱するためである。また、コントローラから分離することで、View の入れ替えが、容易にできる。

ファイルのダウンロード処理の概要を、以下に示す。

1. DispatchServlet は、コントローラへファイルダウンロードのリクエストを送信する。
2. コントローラは、ファイル表示の情報を取得する。
3. コントローラは、View を選択する。
4. ファイルレンダリングは、View で行われる。

Spring ベースの Web アプリケーションで、ファイルをレンダリングするため、
本ガイドラインでは、カスタムビューを実装することを推奨する。
Spring フレームワークでは、カスタムビューの実装に
`org.springframework.web.servlet.View` インタフェースを提供している。

PDF ファイルの場合

Spring から提供されている
`org.springframework.web.servlet.view.document.AbstractPdfView`
クラスは、model の情報を用いて PDF ファイルをレンダリングするときに、サブクラスとして利用するクラスである。

Excel ファイルの場合

Spring から提供されている
`org.springframework.web.servlet.view.document.AbstractXlsxView`
クラスは、model の情報を用いて Excel ファイルをレンダリングするときに、サブクラスとして利用するクラスである。

Spring では上記以外にも、いろいろな View の実装を提供している。

View の技術詳細は、Spring Reference View technologies を参照されたい。

共通ライブラリから提供している、
`org.terasoluna.gfw.web.download.AbstractFileDownloadView` は、
任意のファイルをダウンロードするために使用する抽象クラスである。
PDF や Excel 形式以外のファイルをレンダリングする際に、本クラスをサブクラスに定義する。

ちなみに: ファイルダウンロード機能を提供する際には、ディレクトリトラバーサル攻撃への対策が必要な場合がある。ディレクトリトラバーサル攻撃については、ディレクトリトラバーサル攻撃 を参照すること。

4.10.2 How to use

PDF ファイルのダウンロード

PDF ファイルのレンダリングには、Spring から提供されている、
`org.springframework.web.servlet.view.document.AbstractPdfView` を継承したクラスを作成する必要がある。
コントローラで PDF ダウンロードを実装するための手順は、以下で説明する。

カスタム View の実装

AbstractPdfView を継承したクラスの実装例

```
@Component  // (1)
public class SamplePdfView extends AbstractPdfView {  // (2)

  @Override
  protected void buildPdfDocument(Map<String, Object> model,
          Document document, PdfWriter writer, HttpServletRequest request,
          HttpServletResponse response) throws Exception {  // (3)

      document.add(new Paragraph((Date) model.get("serverTime")).toString());
  }
}
```

項番	説明
(1)	本例では、`@Component` アノテーションを使用して、component-scan の対象としている。 後述する、`org.springframework.web.servlet.view.BeanNameViewResolver` の対象とすることができる。
(2)	`AbstractPdfView` を継承する。
(3)	`buildPdfDocument` メソッドを実装する。

警告: iText 2.1.7 に存在する XXE (XML External Entity) 脆弱性について

TERASOLUNA Server Framework for Java 5.4.2.RELEASE でサポートしている iText 2.1.7 では、外部から取得した XML を読み込んで PDF を作成するような実装を行っている場合、CVE-2017-9096 で報告されている脆弱性が発生する可能性がある。

`AbstractPdfView` は、PDF のレンダリングに、iText を利用している。

そのため、Maven の pom.xml に itext の定義を追加する必要がある。

```
<dependencies>
    <!-- omitted -->
    <dependency>
        <groupId>com.lowagie</groupId>
        <artifactId>itext</artifactId>
    </dependency>
</dependencies>
```

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の itext は terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

ViewResolver の定義

org.springframework.web.servlet.view.BeanNameViewResolver とは、Spring のコンテキストで管理された bean 名を用いて実行する View を選択するクラスである。

BeanNameViewResolver を使用する際は、通常使用する、

- JSP 用の ViewResolver(InternalResourceViewResolver)
- Tiles 用の ViewResolver(TilesViewResolver)

より先に BeanNameViewResolver が実行されるように定義する事を推奨する。

注釈: Spring Framework はさまざまな ViewResolver を提供しており、複数の ViewResolver をチェーンすることができる。そのため、特定の状況では、意図しない View が選択されてしまうことがある。

この動作は、<mvc:view-resolvers>要素の子要素に、優先したい ViewResolver を上から順に定義する事で防ぐことができる。

---

bean 定義ファイル

```
<mvc:view-resolvers>
    <mvc:bean-name /> <!-- (1) (2) -->
    <mvc:jsp prefix="/WEB-INF/views/" />
</mvc:view-resolvers>
```

項番	説明
(1)	<mvc:bean-name>要素を使用して、BeanNameViewResolver を定義する。
(2)	<mvc:bean-name>要素を先頭に定義し、通常使用する ViewResolver(JSP 用の ViewResolver) より優先度を高くする。

コントローラでの View の指定

BeanNameViewResolver により、コントローラで」samplePdfView」を返却することで、
Spring のコンテキストで管理された BeanID により、」samplePdfView」である View が使用される。

Java ソースコード

```
@RequestMapping(value = "home", params= "pdf", method = RequestMethod.GET)
public String homePdf(Model model) {
    model.addAttribute("serverTime", new Date());
    return "samplePdfView";    // (1)
}
```

項番	説明
(1)	「samplePdfView」 をメソッドの戻り値として返却することで、 Spring のコンテキストで管理された、SamplePdfView クラスが実行される。

上記の手順を実行した後、以下に示すような PDF を開くことができる。

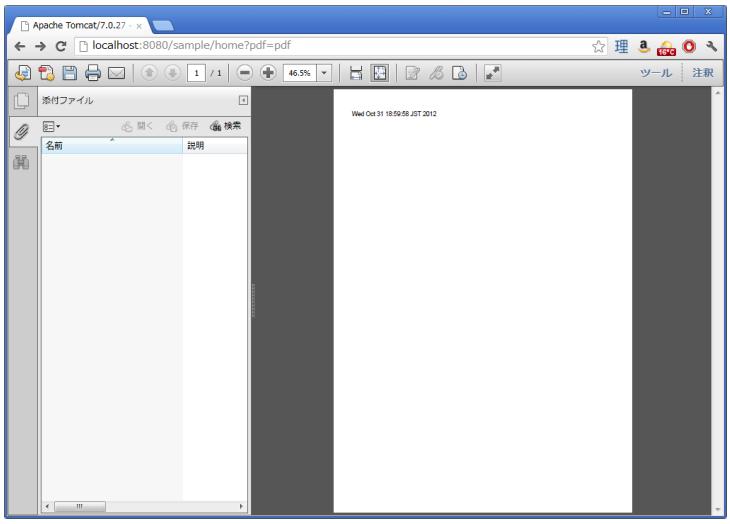


図 4.18　**Picture - FileDownload PDF**

Excel ファイルのダウンロード

EXCEL ファイルのレンダリングには、Spring から提供されている、

`org.springframework.web.servlet.view.document.AbstractXlsxView` を継承したクラスを作成する必要がある。

コントローラで EXCEL ファイルをダウンロードさせるための実装手順は、以下で説明する。

カスタム View の実装

AbstractXlsxView を継承したクラスの実装例

```
@Component  // (1)
public class SampleExcelView extends AbstractXlsxView {  // (2)

    @Override
    protected void buildExcelDocument(Map<String, Object> model,
            Workbook workbook, HttpServletRequest request,
            HttpServletResponse response) throws Exception {  // (3)
        Sheet sheet;
        Cell cell;

        sheet = workbook.createSheet("Spring");
        sheet.setDefaultColumnWidth(12);

        // write a text at A1
        cell = getCell(sheet, 0, 0);
        setText(cell, "Spring-Excel test");

        cell = getCell(sheet, 2, 0);
        setText(cell, ((Date) model.get("serverTime")).toString());
    }

    private Cell getCell(Sheet sheet, int rowNumber, int cellNumber) {
        Row row = sheet.createRow(rowNumber);
        return row.createCell(cellNumber);
    }

    private void setText(Cell cell, String text) {
        cell.setCellValue(text);
    }
}
```

項番	説明
(1)	本例では、`@Component` アノテーションを使用して、component-scan の対象としている。 前述した、`org.springframework.web.servlet.view.BeanNameViewResolver` の対象とすることができる。
(2)	`AbstractXlsxView` を継承する。
(3)	`buildExcelDocument` メソッドを実装する。

`AbstractXlsxView` は、EXCEL のレンダリングに、Apache POI を利用している。
そのため、Maven の pom.xml に POI の定義を追加する必要がある。

```
<dependencies>
    <!-- omitted -->
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi-ooxml</artifactId>
        <exclusions>
            <exclusion>
                <groupId>stax</groupId>
                <artifactId>stax-api</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

注釈: poi-ooxml が依存している stax-api は Java SE より標準で提供されるようになったため、不要なライブラリである。また、ライブラリの競合が発生する可能性があることから、 `<exclusions>`要素を追加し、当該ライブラリがアプリケーションに含まれないよう考慮する必要がある。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で

管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

注釈: `AbstractExcelView` は Spring Framework 4.2 から非推奨となった。そのため、xls ファイル形式をサポートしたい場合は `AbstractXlsView` を使用されたい。詳細は、AbstractXlsView の JavaDoc を参照されたい。

ViewResolver の定義

設定は、PDF ファイルをレンダリングする場合と同様である。詳しくは、*ViewResolver の定義*を参照されたい。

コントローラでの View の指定

`BeanNameViewResolver` により、コントローラで」sampleExcelView」を返却することで、
Spring のコンテキストで管理された BeanID により、”sampleExcelView”である View が使用される。

Java ソース

```
@RequestMapping(value = "home", params= "excel", method = RequestMethod.GET)
public String homeExcel(Model model) {
    model.addAttribute("serverTime", new Date());
    return "sampleExcelView";  // (1)
}
```

項番	説明
(1)	「sampleExcelView」 をメソッドの戻り値として返却することで、 Spring のコンテキストで管理された、`SampleExcelView` クラスが実行される。

上記の手順を実行した後、以下に示すような EXCEL を開くことができる。

任意のファイルのダウンロード

前述した、PDF や EXCEL ファイル以外のファイルのダウンロードを行う場合、
共通ライブラリが提供している、
`org.terasoluna.gfw.web.download.AbstractFileDownloadView` を継承したクラスを実装す

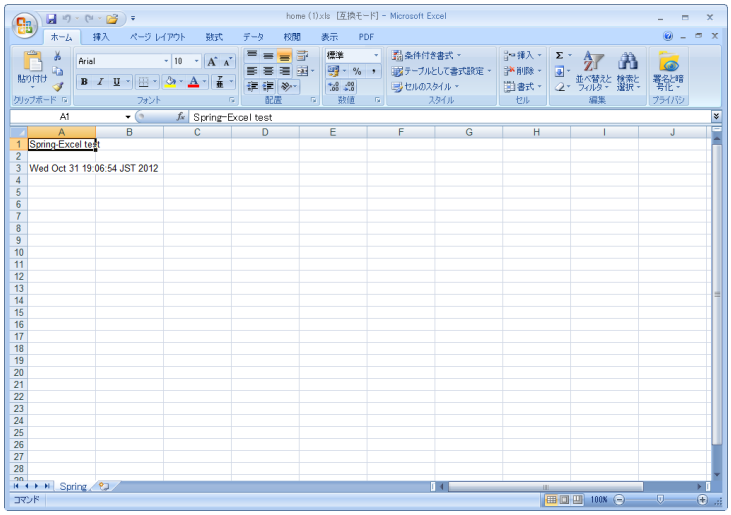


図 4.19　**Picture - FileDownload EXCEL**

ればよい。

他の形式にファイルレンダリングするために、`AbstractFileDownloadView` では、以下を実装する必要がある。

1. レスポンスボディへの書き込むための InputStream を取得する。
2. HTTP ヘッダに情報を設定する。

コントローラでファイルダウンロードを実装するための手順は、以下で説明する。

カスタム View の実装

テキストファイルをダウンロードする例を用いて、説明を行う。

AbstractFileDownloadView を継承したクラスの実装例

```
@Component  // (1)
public class TextFileDownloadView extends AbstractFileDownloadView {  // (2)

    @Override
    protected InputStream getInputStream(Map<String, Object> model,
            HttpServletRequest request) throws IOException {  // (3)
        Resource resource = new ClassPathResource("abc.txt");
        return resource.getInputStream();
    }

    @Override
    protected void addResponseHeader(Map<String, Object> model,
            HttpServletRequest request, HttpServletResponse response) {  // (4)
```

```
        response.setHeader("Content-Disposition",
                "attachment; filename=abc.txt");
        response.setContentType("text/plain");

    }
}
```

項番	説明
(1)	本例では、@Component アノテーションを使用して、component-scan の対象としている。 前述した、org.springframework.web.servlet.view.BeanNameViewResolver の対象とすることができる。
(2)	AbstractFileDownloadView を継承する。
(3)	getInputStream メソッドを実装する。 ダウンロード対象の、InputStream を返却すること。
(4)	addResponseHeader メソッドを実装する。 ダウンロードするファイルに合わせた、 Content-Disposition や、ContentType を設定する。

ViewResolver の定義

設定は、PDF ファイルをレンダリングする場合と同様である。詳しくは、*ViewResolver の定義*を参照されたい。

コントローラでの View の指定

BeanNameViewResolver により、コントローラで」textFileDownloadView」を返却することで、
Spring のコンテキストで管理された BeanID により、”textFileDownloadView”である View が使用される。

Java ソース

```
@RequestMapping(value = "download", method = RequestMethod.GET)
public String download() {
    return "textFileDownloadView"; // (1)
}
```

項番	説明
(1)	「textFileDownloadView」 をメソッドの戻り値として返却することで、 Spring のコンテキストで管理された、`TextFileDownloadView` クラスが実行される。

ちなみに: 前述してきたように、Spring は Model の情報をいろいろな View にレンダリングすることができる。Spring では、Jasper Reports のようなレンダリングエンジンをサポートし、さまざまな View を返却することも可能である。詳細は、Spring の公式ドキュメント Spring reference を参照されたい。

4.11 Tiles による画面レイアウト

4.11.1 Overview

ヘッダ、フッタ、サイドメニューといった共通的なレイアウトを持つ Web アプリケーションを開発する場合に、全ての JSP に共通部分をコーディングすると、メンテナンスが煩雑になる。
例えば、ヘッダのデザインを修正する必要がある場合、全ての JSP に修正を加えなければならない。

JSP での開発で多くの画面で同じレイアウトを使用する場合は、Apache Tiles(以下、Tiles) の利用を推奨する。
理由は、以下 3 つの通りである。

1. 設計者によるレイアウトの誤差をなくすこと
2. 冗長なコードを減らすこと
3. 大きなレイアウトの変更が容易になること

Tiles は、統一的な画面レイアウトの際に定義を行うことで、別々の JSP を組み合わせることができる。
その結果、各々の JSP ファイルに、余計なコードを記述することがなくなるため、開発者の作業を楽にできる。
例えば、下記のようなレイアウト構成が複数の画面に存在する場合、

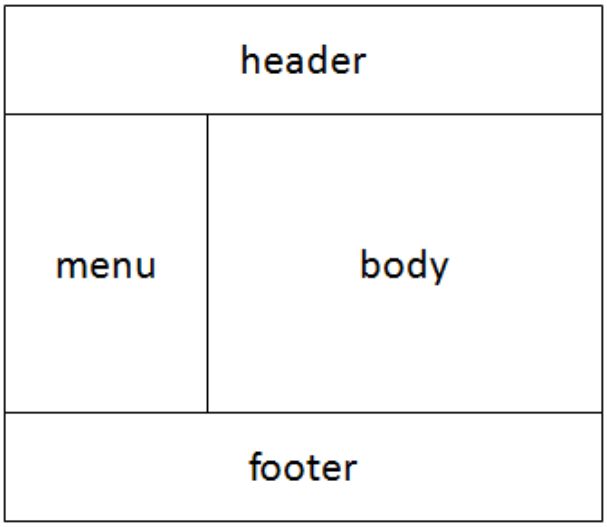


図 4.20 **Picture - Image of screen layout**

Tiles を使用することにより、同じレイアウトの全ての画面で header や menu、footer を include してサイズを指定することなく、body の作成のみに集中することができる。
実際の JSP ファイルは下記のようになる。

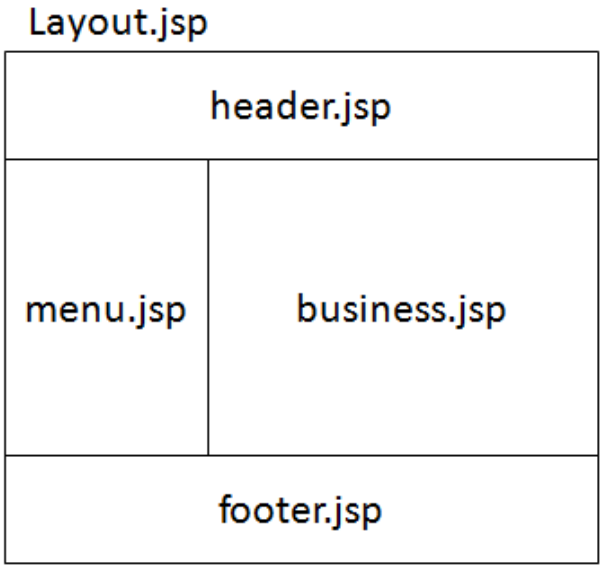


図 4.21 **Picture - Image of layout jsp**

よって、Tiles で画面レイアウトを設定した後は、業務に相当する JSP のみ (buisness.jsp) 画面毎に作成すればよい。

注釈: Tiles の適用をしない方がよい場合もある。例えば、エラー画面に Tiles を使用するのは、以下の理由により推奨しない。

- エラー画面表示中に Tiles によるエラーが発生すると解析がしにくくなるため。(二重障害発生の場合)
- web.xml の<error-pages>タグで設定する JSP では、必ずしも画面表示に Tiles によるテンプレートが適用されないため。

4.11.2 How to use

pom.xml の設定

Tiles を Maven で使用する場合、以下の dependency を pom.xml に追加する必要がある。

```
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-recommended-web-dependencies</artifactId><!-- (1) --
↪>
```

```
    <type>pom</type><!-- (2) -->
</dependency>
```

項番	説明
(1)	web に関連するライブラリ群が定義してある terasoluna-gfw-recommended-web-dependencies を dependency に追加する。
(2)	terasoluna-gfw-recommended-web-dependencies は依存関係が定義してある pom ファイルでしかないため、 `<type>pom</type>` の指定が必要である。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。

Spring MVC と Tiles の連携

Spring MVC と Tiles を連携するには `org.springframework.web.servlet.view.tiles3.TilesViewResolver` を利用すればよい。

Spring MVC の Controller の実装 (View 名の返却) を変更する必要は無い。

設定方法について、以下に示す。

Bean の定義 (ViewResolver、TilesConfigurer)

- spring-mvc.xml

```
<mvc:view-resolvers>
    <mvc:tiles /> <!-- (1) -->
    <mvc:jsp prefix="/WEB-INF/views/" /> <!-- (2) -->
</mvc:view-resolvers>
```

```
<!-- (3) -->
<mvc:tiles-configurer>
    <mvc:definitions location="/WEB-INF/tiles/tiles-definitions.xml" />
</mvc:tiles-configurer>
```

項番	説明
(1)	<mvc:tiles>要素を使用して、TilesViewResolver を定義する。 <mvc:jsp>要素より上に定義することで、最初に Tiles 定義ファイル (tiles-definitions.xml) を参照して View を解決するようにする。Controller から返却された View 名が、Tiles 定義ファイル内の definition 要素の name 属性のパターンに合致する場合、TilesViewResolver によって View が解決される。
(2)	<mvc:jsp>要素を使用して、JSP 用の InternalResourceViewResolver を定義する。 <mvc:tiles>要素より下に定義することで、TilesViewResolver で解決できなかった View 名のみ、JSP 用の InternalResourceViewResolver を使用して View を解決するようにする。View 名に対応する JSP ファイルが、/WEB-INF/views/配下に存在する場合、JSP 用の InternalResourceViewResolver によって View が解決される。
(3)	<mvc:tiles-configurer>要素を使用して、Tiles 定義ファイルを読み込む。 <mvc:definitions>要素の location 属性に、Tiles 定義ファイルを指定する。

Tiles の定義

- tiles-definitions.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
   "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
   "http://tiles.apache.org/dtds/tiles-config_3_0.dtd"> <!-- (1) -->

<tiles-definitions>
    <definition name="layouts"
        template="/WEB-INF/views/layout/template.jsp"> <!-- (2) -->
        <put-attribute name="header"
            value="/WEB-INF/views/layout/header.jsp" /> <!-- (3) -->
        <put-attribute name="footer"
            value="/WEB-INF/views/layout/footer.jsp" /> <!-- (4) -->
    </definition>

    <definition name="*/*" extends="layouts"> <!-- (5) -->
        <put-attribute name="title" value="title.{1}.{2}" /> <!-- (6) -->
        <put-attribute name="body" value="/WEB-INF/views/{1}/{2}.jsp" /> <!--␣
↪(7) -->
    </definition>
</tiles-definitions>
```

項番	説明
(1)	tiles の dtd を定義する。
(2)	レイアウト構成の親定義。 template 属性には、レイアウトを定義している jsp ファイルを指定する。
(3)	header を定義している jsp ファイルを指定する。
(4)	footer を定義している jsp ファイルを指定する。
(5)	描画の際に Controller から返却された View 名が name のパターンと同じ場合に呼ばれるレイアウト定義。 extends している layouts 定義も適用される。
(6)	タイトルを指定する。 value は spring-mvc に取り込まれている properties の中から取得する。(以下の説明では application-messages.properties に設定する。) {1},{2}は Controller から返却された View 名の」*/*」の「*」の 1 つ目、2 つ目に該当する。
(7)	body を定義している jsp ファイルの置き場所について、{1},{2}に Controller から返却された View 名の」*/*」の「*」の 1 つ目、2 つ目が一致するように設計する。これにより、Controller から返却される View 名ごとの定義を記述する手間を省くことができる。

注釈: Tiles の適用をしたくない画面 (エラー画面等) の場合、Tiles 使用対象にならないようなファイル構成にする必要がある。ブランクプロジェクトでは、エラー画面に InternalResourceViewResolver が使われるように (「*/*」形式にならないように)、 /WEB-INF/views/common/error/xxxError.jsp 形式にしている。

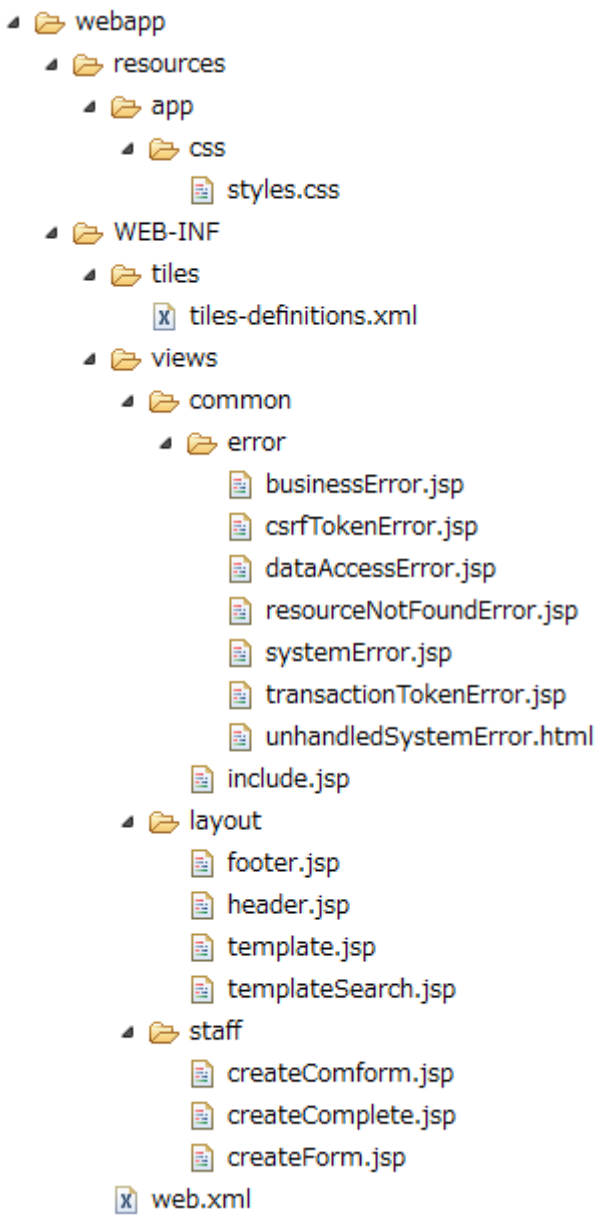
- *application-messages.properties*

```
title.staff.createForm = Create Staff Information
```

注釈: メッセージプロパティファイルの記載方法については、 メッセージ管理 を参照されたい。

Tiles を設定したときのファイル構成を以下に示す。

- tiles File Path



カスタムタグの設定

Tiles を使用するためにカスタムタグ (TLD) を設定する必要がある。

- /WEB-INF/views/common/include.jsp

```
<%@ page session="false"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec"%>
<%@ taglib uri="http://terasoluna.org/functions" prefix="f"%>
<%@ taglib uri="http://terasoluna.org/tags" prefix="t"%>
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%> <!-- (1)␣
↪-->
<%@ taglib uri="http://tiles.apache.org/tags-tiles-extras" prefix="tilesx"%>
↪<!-- (2) -->
```

項番	説明
(1)	Tiles 用のカスタムタグ (TLD) の定義を追加する。
(2)	Tiles-extras 用のカスタムタグ (TLD) の定義を追加する。

Tiles のカスタムタグの詳細は、こちらを参照されたい。

ちなみに:

version 2 系では tiles taglib は一つであったが、version 3 から tiles-extras taglib が追加された。

version 2 系では tiles taglib で利用可能であった useAttribute tag が version 3 から tiles-extras taglib へ移動されているので、利用していた場合は注意すること。

変更例) <*tiles:useAttribute*> : version 2 -> <*tilesx:useAttribute*> : version 3

- web.xml

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <el-ignored>false</el-ignored>
        <page-encoding>UTF-8</page-encoding>
        <scripting-invalid>false</scripting-invalid>
        <include-prelude>/WEB-INF/views/common/include.jsp</include-prelude>
↪<!-- (1) -->
    </jsp-property-group>
</jsp-config>
```

項番	説明
(1)	web.xml の設定で、jsp ファイル (～.jsp) を読み込む場合、事前に include.jsp を読み込ませることができる。

注釈: カスタムタグは template.jsp に設定しても問題は無いが、カスタムタグの定義はインクルード用の共通 jsp ファイルに作成することを推奨する。詳細は インクルード用の共通 *JSP* の作成 を参照されたい。

レイアウト作成

レイアウトの枠となる jsp（template）と、レイアウトに埋め込む jsp を作成する。

- template.jsp

```
<!DOCTYPE html>
<html class="no-js">
<head>
<meta charset="utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
<meta name="viewport" content="width=device-width" />
<script type="text/javascript">

</script> <!-- (1) -->
<c:set var="titleKey"> <!-- (2) -->
    <tiles:insertAttribute name="title" ignore="true" />
</c:set>
<title><spring:message code="${titleKey}" text="Create Staff Information" /></
↪title><!-- (3) -->
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/
↪css/styles.css">
</head>
<body>
    <div id="header">
        <tiles:insertAttribute name="header" /> <!-- (4) -->
    </div>
    <div id="body">
        <tiles:insertAttribute name="body" /> <!-- (5) -->
    </div>
    <div id="footer">
        <tiles:insertAttribute name="footer" /> <!-- (6) -->
    </div>
</body>
</html>
```

項番	説明
(1)	共通的に記述する必要のある内容を (1) より上に記述する。
(2)	tiles-definitions.xml の (6) で指定した `title` の値を取得し、 `titleKey` に設定する。
(3)	タイトルを設定する。 `titleKey` が取得できなかった際は、text 属性で定義したタイトルを表示する。
(4)	tiles-definitions.xml で定義した」header」を読み込む。
(5)	tiles-definitions.xml で定義した」body」を読み込む。
(6)	tiles-definitions.xml で定義した」footer」を読み込む。

- header.jsp

```
<h1>
    <a href="${pageContext.request.contextPath}">Staff Management
        System</a>
</h1>
```

- createForm.jsp(body 部分の例)

開発者は、header や footer の余分なソースを記述せずに、body 部分のみに集中して記述できる。

```
<h2>Create Staff Information</h2>
<table>
    <tr>
        <td>Staff First Name</td>
        <td><input type="text" /></td>
    </tr>
    <tr>
        <td>Staff Family Name</td>
        <td><input type="text" /></td>
    </tr>
    <tr>
```

```
        <td rowspan="5">Staff Authorities</td>
        <td><input type="checkbox" name="sa" value="01" /> Staff
            Management</td>
    </tr>
    <tr>
        <td><input type="checkbox" name="sa" value="02" /> Master
            Management</td>
    </tr>
    <tr>
        <td><input type="checkbox" name="sa" value="03" /> Stock
            Management</td>
    </tr>
    <tr>
        <td><input type="checkbox" name="sa" value="04" /> Order
            Management</td>
    </tr>
    <tr>
        <td><input type="checkbox" name="sa" value="05" /> Show Shopping
            Management</td>
    </tr>
</table>

<input type="submit" value="cancel" />
<input type="submit" value="confirm" />
```

- footer.jsp

```
<p style="text-align: center; background: #e5eCf9;">Copyright &copy; 20XX␣
↪CompanyName</p>
```

注釈: フッターに記載する著作権に関しては 画面フッターの著作権 を参照すること。

Controller 作成

Controller から View 名として」staff/createForm」を返却する。

- StaffCreateController.java

```
@RequestMapping(value = "create", method = RequestMethod.GET, params = "form")
public String createForm() {
    return "staff/createForm"; // (1)
}
```

項番	説明
(1)	staff が{1}、createForm が{2}となり、properties からタイトル名を取得し、JSP を特定する。

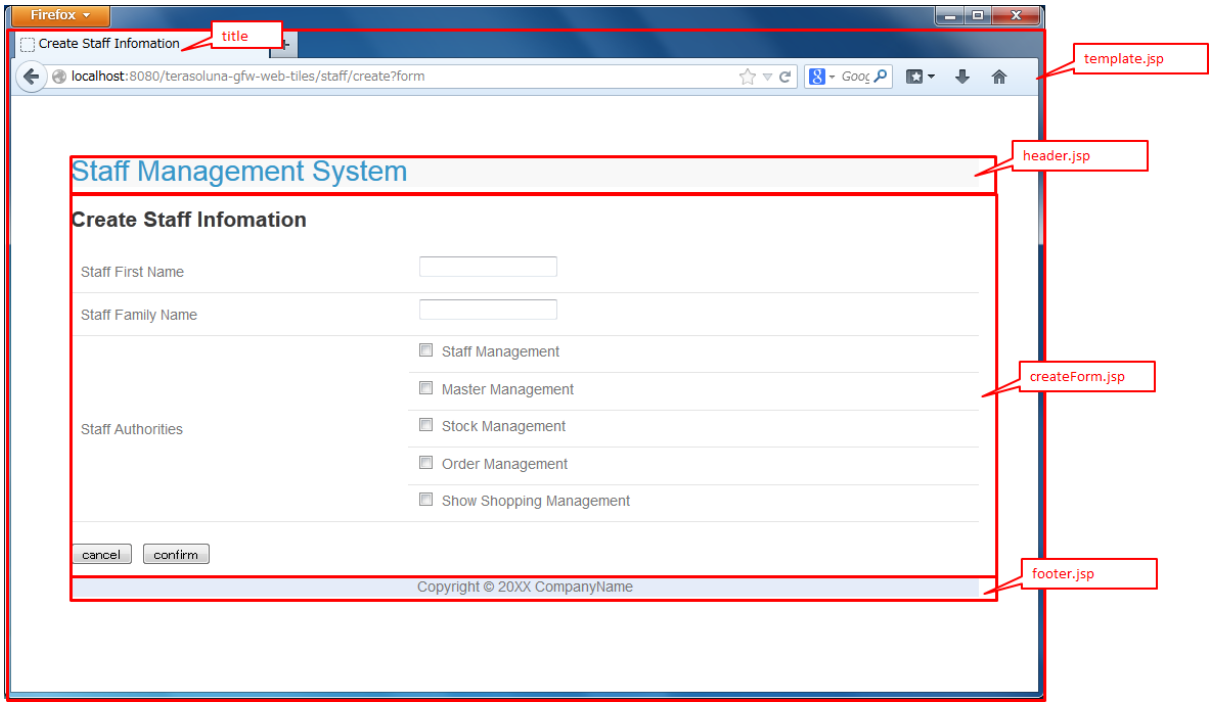
画面描画

View 名に」staff/createForm」が指定されると、以下のように Tiles がレイアウトを構築して画面描画を行う。

```
<definition name="layouts"
    template="/WEB-INF/views/layout/template.jsp"> <!-- (1) -->
    <put-attribute name="header"
        value="/WEB-INF/views/layout/header.jsp" /> <!-- (2) -->
    <put-attribute name="footer"
        value="/WEB-INF/views/layout/footer.jsp" /> <!-- (3) -->
</definition>

<definition name="*/*" extends="layouts">
  <put-attribute name="title" value="title.{1}.{2}" /> <!-- (4) -->
  <put-attribute name="body"
    value="/WEB-INF/views/{1}/{2}.jsp" /> <!-- (5) -->
</definition>
```

項番	説明
(1)	該当する View 名が指定された時、親レイアウトである layouts が呼ばれ、テンプレートが /WEB-INF/views/layout/template.jsp に設定される。
(2)	テンプレート /WEB-INF/views/layout/template.jsp 内に存在する `header` に WEB-INF/views/layout/header.jsp が設定される。
(3)	テンプレート /WEB-INF/views/layout/template.jsp 内に存在する `footer` に /WEB-INF/views/layout/footer.jsp が設定される。
(4)	staff が{1}、createForm が{2}となり、spring-mvc に取り込まれている properties から `title.staff.createForm` を key に value を取得する。
(5)	staff が{1}、createForm が{2}となり、テンプレート/WEB-INF/views/layout/template.jsp 内に存在する `body` に/WEB-INF/views/staff/createForm.jsp が設定される。

結果として上記の template.jsp に、header.jsp、createForm.jsp、footer.jsp が組み合わされた方法でブラウザに出力される。



4.11.3 How to extend

複数レイアウトを設定する場合

実際に業務アプリケーションを作成する場合、業務内容によって表示レイアウトを分けたい場合がある。

今回は、スタッフ検索機能の場合、メニューを画面の左側に出す要望があると想定する。

その設定方法について、 *How to use* をベースに以下に示す。

Tiles の定義

- tiles-definitions.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
   "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
   "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">

<tiles-definitions>
    <definition name="layoutsOfSearch"
        template="/WEB-INF/views/layout/templateSearch.jsp"> <!-- (1) -->
        <put-attribute name="header"
            value="/WEB-INF/views/layout/header.jsp" />
        <put-attribute name="menu"
            value="/WEB-INF/views/layout/menu.jsp" />
        <put-attribute name="footer"
            value="/WEB-INF/views/layout/footer.jsp" />
```

```
    </definition>

    <definition name="*/search*" extends="layoutsOfSearch"> <!-- (2) -->
        <put-attribute name="title" value="title.{1}.search{2}" /> <!-- (3) -
↪->
        <put-attribute name="body" value="/WEB-INF/views/{1}/search{2}.jsp" /
↪> <!-- (4) -->
    </definition>

    <definition name="layouts"
        template="/WEB-INF/views/layout/template.jsp">
        <put-attribute name="header"
            value="/WEB-INF/views/layout/header.jsp" />
        <put-attribute name="footer"
            value="/WEB-INF/views/layout/footer.jsp" />
    </definition>

    <definition name="*/*" extends="layouts">
        <put-attribute name="title" value="title.{1}.{2}" />
        <put-attribute name="body" value="/WEB-INF/views/{1}/{2}.jsp" />
    </definition>
</tiles-definitions>
```

項番	説明
(1)	今回追加するレイアウト構成の親定義。 別のレイアウトを使用する場合、difinition タグの name 属性について、既存のレイアウト定義」layouts」と重複しないようにする。
(2)	今回追加するレイアウトについて、描画の際に Controller から返却された View 名が name のパターンと同じ場合に呼ばれるレイアウト定義。 View 名が」*/search*」に該当する場合、このレイアウト定義が読み込まれる。 extends している レイアウト定義」layoutsOfSearch」も適用される。
(3)	今回追加するレイアウトで使用するタイトルを指定する。 value は spring-mvc に取り込まれている properties の中から取得する。(以下の説明では application-messages.properties に設定する。) {1}は Controller から返却される View 名の」*/search*」の「*」の 1 つ目、」search{2}」は View 名の」*/search*」の」search*」に該当する為、先頭が」search」で始まる必要がある。
(4)	body を定義している jsp ファイルの置き場所について、{1}に Controller から返却される View 名の」*/search*」の「*」の 1 つ目、」search{2}」は View 名の」*/search*」の」search*」であるため、先頭に」search」を含んだ JSP ファイル名が一致するように設計する。 JSP ファイルの置き場所の構成によって value 属性の値を変更する必要がある。

注釈: Controller から返却される View 名が definition タグの name 属性のパターンに複数該当する場合、上から順に確認し、1 番最初に該当するパターンが採用される。上記の場合、スタッフ検索画面の View 名が、definition タグの name 属性のパターンに複数該当するため、1 番上にレイアウト定義している。

- *application-messages.properties*

```
title.staff.createForm = Create Staff Information
title.staff.searchStaff = Search Staff Information # (1)
```

項番	説明
(1)	今回追加するメッセージ。 「staff」は Controller から返却された View 名の」*/search*」の「*」の 1 つ目。 「searchStaff」は Controller から返却された View 名の」*/search*」の」search*」に該当する為、先頭が」search」で始まる必要がある。

レイアウト作成

レイアウトの枠となる jsp（template）と、レイアウトに埋め込む jsp を作成する。

- templateSearch.jsp

```
<!DOCTYPE html>
<html class="no-js">
<head>
<meta charset="utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
<meta name="viewport" content="width=device-width" />
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/
↪css/styles.css"
    media="screen, projection">
<script type="text/javascript">

</script>
<c:set var="titleKey">
    <tiles:insertAttribute name="title" ignore="true" />
</c:set>
<title><spring:message code="${titleKey}" text="Search Staff Information" /></
↪title>
</head>
<body>
    <div id="header">
        <tiles:insertAttribute name="header" />
    </div>
    <div id="menu">
        <tiles:insertAttribute name="menu" /> <!-- (1) -->
    </div>
    <div id="body">
        <tiles:insertAttribute name="body" />
    </div>
    <div id="footer">
        <tiles:insertAttribute name="footer" />
    </div>
</body>
</html>
```

項番	説明
(1)	tiles-definitions.xml で定義した」menu」を読み込む。 それ以外は *How to use* と同じ

- styles.css

```
div#menu { /* (1) */
    float: left;
    width: 20%;
}

div#searchBody { /* (2) */
    float: right;
    width: 80%;
}

div#footer { /* (3) */
    clear: both;
}
```

項番	説明
(1)	menu 部分の style を設定する。 ここでは、float:left でメニュー画面を左側に寄せて、width:20% として横幅 2 割で表示をするようにしている。
(2)	body 部分の style を設定する。 ここでは、float:right で業務画面を右側に寄せて、width:80% として横幅 8 割で表示をするようにしている。 名前を searchBody にしているのは、既存のレイアウトと名前が重複することにより、既存のレイアウトの style に影響を与えないためである。
(3)	footer 部分の style を設定する。 上記 menu 部分と body 部分の float の効果を初期化している。これにより、menu 部分と body 部分の下に表示するようにしている。

- header.jsp

How to use と同じ

- menu.jsp

```
<table>
    <tr>
        <td><a href="${pageContext.request.contextPath}/staff/create?form">
↪Create Staff Information</a></td>
    </tr>
    <tr>
        <td><a href="${pageContext.request.contextPath}/staff/search">Search␣
↪Staff Information</a></td>
    </tr>
</table>
```

- searchStaff.jsp(body 部分の例)

```
<h2>Search Staff Information</h2>
<table>
    <tr>
        <td>Staff First Name</td>
        <td><input type="text" /></td>
    </tr>
    <tr>
        <td>Staff Family Name</td>
        <td><input type="text" /></td>
    </tr>
    <tr>
        <td rowspan="5">Staff Authorities</td>
        <td><input type="checkbox" name="sa" value="01" /> Staff
            Management</td>
    </tr>
    <tr>
        <td><input type="checkbox" name="sa" value="02" /> Master
            Management</td>
    </tr>
    <tr>
        <td><input type="checkbox" name="sa" value="03" /> Stock
            Management</td>
    </tr>
    <tr>
        <td><input type="checkbox" name="sa" value="04" /> Order
            Management</td>
    </tr>
    <tr>
        <td><input type="checkbox" name="sa" value="05" /> Show Shopping
            Management</td>
    </tr>
</table>

<input type="submit" value="Search" />
```

- footer.jsp

How to use と同じ

Controller 作成

Controller から View 名として」staff/searchStaff」を返却する。

- StaffSearchController.java

```
@RequestMapping(value = "search", method = RequestMethod.GET)
public String createForm() {
    return "staff/searchStaff"; // (1)
}
```

項番	説明
(1)	staff が{1}、searchStaff が{2}となり、properties からタイトル名を取得し、JSP を特定する。

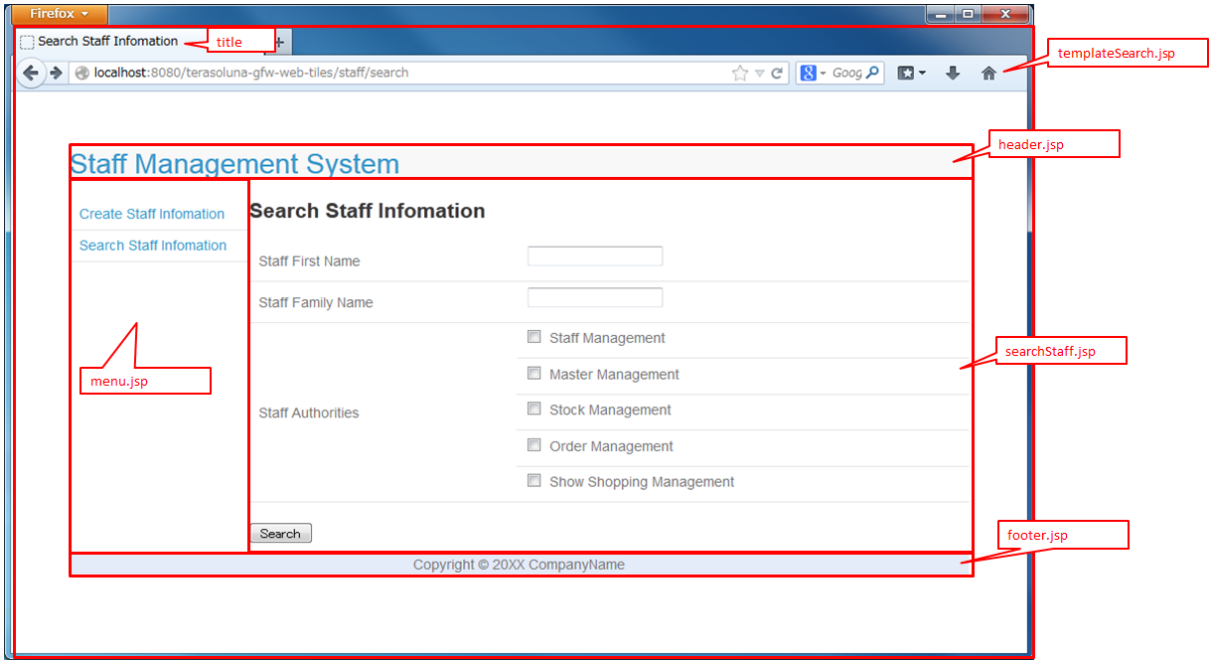
画面描画

View 名に」staff/searchStaff」が指定されると、以下のように別のレイアウトを構築して画面描画を行う。

```
<definition name="layoutsOfSearch"
    template="/WEB-INF/views/layout/templateSearch.jsp"> <!-- (1) -->
    <put-attribute name="header"
        value="/WEB-INF/views/layout/header.jsp" /> <!-- (2) -->
    <put-attribute name="menu"
        value="/WEB-INF/views/layout/menu.jsp" /> <!-- (3) -->
    <put-attribute name="footer"
        value="/WEB-INF/views/layout/footer.jsp" /> <!-- (4) -->
</definition>

<definition name="*/search*" extends="layoutsOfSearch"> <!-- (5) -->
    <put-attribute name="title" value="title.{1}.search{2}" /> <!-- (6) -->
    <put-attribute name="body" value="/WEB-INF/views/{1}/search{2}.jsp" /> <!-
↪- (7) -->
</definition>
```

項番	説明
(1)	該当する View 名が指定された時、親レイアウトである layoutsOfSearch が呼ばれ、テンプレートが /WEB-INF/views/layout/templateSearch.jsp に設定される。
(2)	テンプレート /WEB-INF/views/layout/templateSearch.jsp 内に存在する `header` に WEB-INF/views/layout/header.jsp が設定される。
(3)	テンプレート /WEB-INF/views/layout/templateSearch.jsp 内に存在する `menu` に /WEB-INF/views/layout/menu.jsp が設定される。
(4)	テンプレート /WEB-INF/views/layout/templateSearch.jsp 内に存在する `footer` に /WEB-INF/views/layout/footer.jsp が設定される。
(5)	Controller から返却された View 名が」*/search*」に該当する場合、このレイアウト定義が読み込まれる。 その時、親レイアウトである」layoutsOfSearch」も読み込まれる。
(6)	staff が{1}、searchStaff が」search{2}」となり、spring-mvc に取り込まれている properties から `title.staff.searchStaff` を key に value を取得する。
(7)	staff が{1}、searchStaff が」search{2}」となり、テンプレート/WEB-INF/views/layout/templateSearch.jsp 内に存在する `body` に/WEB-INF/views/staff/searchStaff.jsp が設定される。

結果として上記の templateSearch.jsp に、header.jsp、menu.jsp、searchStaff.jsp、footer.jsp が組み合わされた方法でブラウザに出力される。



4.12 共通ライブラリが提供する JSP Tag Library と EL Functions

4.12.1 Overview

共通ライブラリでは、JSP の実装をサポートする機能として、以下に示す JSP Tag Library と EL Functions を提供している。

JSP Tag Library

共通ライブラリから提供している JSP Tag Library を以下に示す。

項番	タグ名	概要
1.	*<t:pagination>*	ページネーションリンクを出力する。
2.	*<t:messagesPanel>*	処理結果メッセージを出力する。
3.	*<t:transaction>*	トランザクショントークンを hidden 項目として出力する。

EL Functions

共通ライブラリから提供している EL Functions を以下に示す。

XSS 対策関連

項番	関数名	概要
1.	*f:h()*	指定されたオブジェクトを文字列に変換し、変換した文字列内の HTML 特殊文字をエスケープする。
2.	*f:js()*	指定された文字列内の JavaScript 特殊文字をエスケープする。
3.	*f:hjs()*	指定された文字列内の JavaScript 特殊文字をエスケープ後、HTML 特殊文字をエスケープする。(`f:h(f:js())` のショートカット関数)

URL 関連

項番	関数名	概要
4.	*f:query()*	指定されたオブジェクトから、UTF-8 で URL エンコーディングされたクエリ文字列を生成する。
5.	*f:u()*	指定された文字列を UTF-8 で URL エンコーディングする。

DOM 関連

項番	関数名	概要
6.	*f:link()*	指定された URL にジャンプするハイパーリンク (<a>タグ) を生成する。
7.	*f:br()*	指定された文字列内の改行コードを タグに変換する。

ユーティリティ

項番	関数名	概要
8.	*f:cut()*	指定された文字列から、指定された文字数を抜き出す。

4.12.2 How to use

共通ライブラリから提供している JSP Tag Library と EL 関数の使用方法を以下に示す。なお、他の章で使用方法の説明があるものについては、該当箇所へのハイパーリンクを貼っている。

<t:pagination>

`<t:pagination>`タグは、ページ検索の結果 (`org.springframework.data.domain.Page`) に格納されている情報を参照して、ページネーションリンクを出力する JSP Tag Library である。

ページネーション機能の説明及び本タグの使用方法は、「ページネーション」の以下の節を参照されたい。

- ページネーションリンクについては、「ページネーションリンクの表示について」
- 本タグのパラメータ値については、「*JSP タブライブラリのパラメータについて*」
- 本タグを使用した JSP の基本的な実装方法については、「ページネーションリンクの表示」
- ページネーションリンクのレイアウトの変更方法については、「*JSP の実装 (レイアウト変更編)*」

<t:messagesPanel>

`<t:messagesPanel>`タグは、処理結果メッセージ (`org.terasoluna.gfw.common.message.ResultMessage` や例外が保持するメッセージなど) を出力する JSP Tag Library である。

本タグの使用方法は、「メッセージ管理」の以下の節を参照されたい。

- 本タグを使用したメッセージの表示方法については、「結果メッセージの表示」
- 本タグのパラメータ値については、「*<t:messagesPanel>タグの属性変更*」

<t:transaction>

<t:transaction>タグは、トランザクショントークンを hidden 項目 (<input type="hidden">") として出力する JSP Tag Library である。

トランザクショントークンチェック機能の説明及び本タグの使用方法は、「二重送信防止」の以下の節を参照されたい。

- トランザクショントークンチェック機能については、「トランザクショントークンチェックの適用」
- 本タグの使用方法については、「トランザクショントークンチェックの *View(JSP)* での利用方法」

注釈: 本タグは、HTML 標準の<form>タグを使用する際にトランザクショントークンをサーバに送信するために使用する。

Spring Framework 提供の<form:form>タグ (JSP Tag Library) を使用する際は、共通ライブラリから提供している org.terasoluna.gfw.web.token.transaction.TransactionTokenRequestDataValueProcessor が自動でトランザクショントークンを埋め込む仕組みになっているため、本タグを使用する必要はない。

---

f:h()

f:h() は、引数に指定されたオブジェクトを文字列に変換し、変換した文字列内の HTML 特殊文字をエスケープする EL Function である。

HTML 特殊文字とエスケープ仕様については、「*Output Escaping*」を参照されたい。

f:h() 関数仕様

引数

項番	型	説明
1.	java.lang.Object	HTML 特殊文字が含まれる可能性があるオブジェクト

注釈: 指定されたオブジェクトは、

- 配列の場合は、java.util.Arrays#toString メソッド
- 配列以外の場合は、指定されたオブジェクトの toString メソッド

を使用して文字列に変換される。

戻り値

項番	型	説明
1.	`java.lang.String`	HTML エスケープ後の文字列 引数で指定されたオブジェクトが `null` の場合は、空文字 (`""`) を返却する。

f:h() 使用方法

`f:h()` の使用方法については、「出力値を *f:h()* 関数でエスケープする例」を参照されたい。

f:js()

`f:js()` は、引数に指定された文字列内の JavaScript 特殊文字をエスケープする EL Function である。

JavaScript 特殊文字とエスケープ仕様については、「*JavaScript Escaping*」を参照されたい。

f:js() 関数仕様

引数

項番	型	説明
1.	`java.lang.String`	JavaScript 特殊文字が含まれる可能性がある文字列

戻り値

項番	型	説明
1.	`java.lang.String`	JavaScript エスケープ後の文字列 引数で指定された文字列が `null` の場合は、空文字 (`""`) を返却する。

f:js() 使用方法

`f:js()` の使用方法については、「出力値を *f:js()* 関数でエスケープする例」を参照されたい。

f:hjs()

`f:hjs()` は、引数に指定された文字列内の JavaScript 特殊文字をエスケープした後に、HTML 特殊文字をエスケープする EL Function(`f:h(f:js())` のショートカット関数) である。

- 本関数の用途については、「*Event handler Escaping*」を参照されたい。
- JavaScript 特殊文字とエスケープ仕様については、「*JavaScript Escaping*」を参照されたい。
- HTML 特殊文字とエスケープ仕様については、「*Output Escaping*」を参照されたい。

f:hjs() 関数仕様

引数

項番	型	説明
1.	`java.lang.String`	JavaScript 特殊文字又は HTML 特殊文字が含まれる可能性がある文字列

戻り値

項番	型	説明
1.	`java.lang.String`	JavaScript 及び HTML エスケープ後の文字列 引数で指定された文字列が `null` の場合は、空文字 (`""`) を返却する。

f:hjs() 使用方法

`f:hjs()` の使用方法については、「出力値を *f:hjs()* 関数でエスケープする例」を参照されたい。

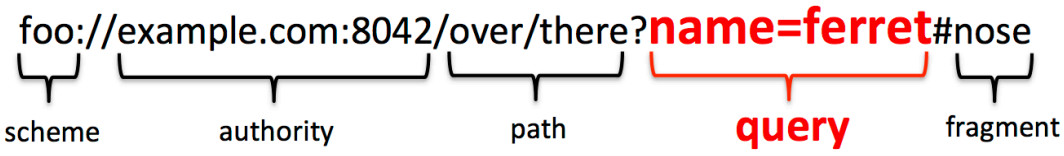
f:query()

`f:query()` は、引数に指定された JavaBean(フォームオブジェクト) 又は `java.util.Map` オブジェクトから、クエリ文字列を生成する EL Function である。クエリ文字列内のパラメータ名とパラメータ値は、UTF-8 で URL エンコーディングされる。

URL エンコーディング仕様を以下に示す。

本関数では、クエリ文字列のパラメータ名とパラメータ値に対して、RFC 3986 ベースの URL エンコーディングを行う。RFC 3986 では、クエリ文字列のパート以下のように定義している。

- query = *(pchar / 「/」 / 「?」)
- pchar = unreserved / pct-encoded / sub-delims / 「:」 / 「@」
- unreserved = ALPHA / DIGIT / 「-」 / 「.」 / 「_」 / 「~」



- sub-delims = 「!」 / 「$」 / 「&」 / 「'」 / 「(」 / 「)」 / 「*」 / 「+」 / 「,」 / 「;」 / 「=」
- pct-encoded = 「%」 HEXDIG HEXDIG

本関数では、クエリ文字列として使用できる文字のうち、

- 「=」 (パラメータ名とパラメータ値のセパレータ文字)
- 「&」 (複数のパラメータを扱う場合のセパレータ文字)
- 「+」 (HTML の form からサブミットした時に半角スペースを表す文字)

を pct-encoded 形式の文字列にエンコーディングする。

f:query() 関数仕様

引数

項番	型	説明
1.	`java.lang.Object`	クエリ文字列の生成元となるオブジェクト (JavaBean 又は `Map`) JavaBean を指定した場合はプロパティ名がリクエストパラメータ名となり、`Map` を指定した場合はキー名がリクエストパラメータとなる。 JavaBean のプロパティ及び `Map` の値としてサポートしている型は以下の通りである。 • `Iterable` インタフェースの実装クラス • 配列 • `Map` インタフェースの実装クラス • JavaBean • シンプル型 (`DefaultFormattingConversionService` を使って `String` 型へ変換可能なクラス) terasoluna-gfw-web 5.0.1.RELEASE より、ネスト構造をもつ JavaBean 及び `Map` を指定できるように改善されている。

注釈: 指定されたオブジェクトのシンプル型のプロパティ値は、`org.springframework.format.support.DefaultFormattingConversionService` の `convert` メソッドを使用して文字列に変換される。`ConversionService` については、Spring Framework Reference Documentation(Spring Type Conversion) を参照されたい。

戻り値

項番	型	説明
1.	`java.lang.String`	引数で指定されたオブジェクトを元に生成したクエリ文字列 (UTF-8 で URL エンコーディング済みの文字列) 引数で指定されたオブジェクトが、JavaBean 又は `Map` 以外の場合は、空文字 (`""`) を返却する。

注釈: クエリ文字列への変換ルール

`f:query()` は、Spring Web MVC のバインディング処理で扱うことができる形式に変換している。具体的には以下のルールでクエリ文字列に変換している。

[リクエストパラメータ名]

条件	パラメータ名の変換仕様	変換例
プロパティの型が `Iterable` の実装クラス又は配列の場合	プロパティ名 + [**要素位置**]	`status[0]=accepting`
プロパティの型が `Iterable` の実装クラス又は配列で値の要素が空の場合	プロパティ名 ([**要素位置**] は付与しない)	`status=`
プロパティの型が `Map` の実装クラスの場合	プロパティ名 + [`Map` のキー名]	`status[accepting]=AcceptOrder`
プロパティの型 (`Iterable`、配列、`Map` の要素型) が JavaBean の場合	プロパティ名を」.」(ドット) でつなげた値	`mainContract.name=xxx` `subContracts[0].name=xxx`
プロパティの型がシンプル型の場合	プロパティ名	`userId=xxx`

プロパティの値が `null` の場合	「_」 (アンダースコア) + プロパティ名	`_mainContract.name=`
		`_status[0]=`
		`_status[accepting]=`

[リクエストパラメータ値]

条件	パラメータ値の変換仕様	変換例
プロパティの値が `null` の場合	ブランク文字列	`_userId=`
プロパティの型が `Iterable` の実装クラス又は配列で値の要素が空の場合	ブランク文字列	`status=`
プロパティの値が `null` でない場合	`DefaultFormattingConversionService` を使って `String` 型へ変換した値	`issuedDate=20150801`

---

f:query() 使用方法

`f:query()` の使用方法については、「ページリンクで検索条件を引き継ぐ」を参照されたい。ここでは、ページネーションリンクを使用してページを切り替える際に、検索条件を引き継ぐ際の手段として、本関数を使用している。また、関数の仕様と注意点についても記載しているので、これについても一読されたい。

f:u()

`f:u()` は、引数に指定された文字列を UTF-8 で URL エンコーディングする EL Function である。

本関数は、クエリ文字列内のパラメータ値に設定する値を URL エンコーディングするために用意している。URL エンコーディング仕様は、「*f:query()*」を参照されたい。

f:u() 関数仕様

引数

項番	型	説明
1.	`java.lang.String`	URL エンコードが必要な文字が含まれる可能性がある文字列

戻り値

項番	型	説明
1.	`java.lang.String`	URL エンコード後の文字列 引数で指定された文字列が `null` の場合は、空文字 (`""`) を返却する。

f:u() 使用方法

```
<div id="url">
    <a href="https://search.yahoo.com/search?p=${f:u(bean.searchString)}">  <!--
↪(1) -->
        Go to Yahoo Search
    </a>
</div>
```

項番	説明
(1)	上記例では、本関数を使用して URL エンコードした値を検索サイトのリクエストパラメータに設定している。

f:link()

`f:link()` は、引数に指定された URL にジャンプするためのハイパーリンク (<a>タグ) を出力する EL Function である。

警告: 本関数では、URL エンコーディングや特殊文字のエスケープ処理は行われない点に注意すること。

f:link() 関数仕様

引数

項番	型	説明
1.	`java.lang.String`	リンク先の URL 文字列 URL 文字列は、HTTP 又は HTTPS スキーマの URL 形式である必要がある。(e.g : `http://hostname:80/terasoluna/global.ex?id=123`)

戻り値

項番	型	説明
1.	`java.lang.String`	引数に指定された文字列を元に生成したハイパーリンク(<a>タグ) 引数に指定された文字列が、 • 引数で指定された文字列が `null` の場合は、空文字 (`""`) • HTTP 又は HTTPS スキーマの URL 形式でない場合は、ハイパーリンクを生成せず入力値の文字列 を返却する。

f:link() 使用方法

実装例

```
<div id="link">
    ${f:link(bean.httpUrl)}  <!-- (1) -->
</div>
```

出力例

```
<div id="link">
    <a href="http://terasoluna.org/">http://terasoluna.org/</a>  <!-- (2) -->
</div>
```

項番	説明
(1)	引数に指定された URL 文字列からハイパーリンクを生成する。
(2)	引数で指定した URL 文字列が、<a>タグの `href` 属性と、ハイパーリンクのリンク名に設定される。

警告: URL にリクエストパラメータを付加する場合は、リクエストパラメータの値は URL エンコーディングする必要がある。リクエストパラメータを付加する場合は、`f:query()` 関数や `f:u()` 関数を使用して、リクエストパラメータの値を適切に URL エンコーディングすること。

また、戻り値の説明でも記載しているが、引数の URL 文字列の形式が適切でない場合は、ハイパーリンクを生成せず入力値の文字列を返却する仕様としている。そのため、引数に指定する URL 文字列としてユーザからの入力値を使用する場合は、文字列出力処理と同様の HTML 特殊文字のエスケープ処理 (*XSS 対策*) が必要になるケースがある。

f:br()

f:br() は、引数に指定された文字列内の改行コード（CRLF, LF, CR）を
タグに変換する EL Function である。

ちなみに: 改行コードを含む文字列をブラウザ上の表示として改行する場合は、改行コードを
タグに変換する必要がある。

例えば、入力画面のテキストエリア (<textarea>) で入力された文字列を、確認画面や完了画面などで入力された状態のまま表示する際に、本関数を使用するとよい。

---

f:br() 関数仕様

引数

項番	型	説明
1.	java.lang.String	改行コードが含まれる可能性がある文字列

戻り値

項番	型	説明
1.	java.lang.String	変換後の文字列 引数で指定された文字列が null の場合は、空文字 ("") を返却する。

f:br() 使用方法

```
<div id="text">
    ${f:br(f:h(bean.text))}">  <!-- (1) -->
</div>
```

項番	説明
(1)	引数で指定された文字列内の改行コードを タグに変換することで、ブラウザ上の表示を改行する。

注釈: 文字列を画面上に表示する際は、「*XSS 対策*」として HTML 特殊文字をエスケープする必要がある。

f:br() 関数を使用して改行コードを
タグに変換する場合は、上記例のように、HTML 特殊文字をエスケープした文字列を f:br() の引数として渡す必要がある。

f:br() を使用して改行コードを
タグに変換した文字列を、f:h() 関数の引数に渡すと、
という文字がブラウザ上に表示されてしまうため、関数を呼び出す順番に注意すること。

f:cut()

f:cut() は、引数に指定された文字列の先頭から、引数で指定された文字数までの文字列を切り出す EL Function である。

f:cut() 関数仕様

引数

項番	型	説明
1.	java.lang.String	切り出し元となる文字列
2.	int	切り出す文字数

戻り値

項番	型	説明
1.	java.lang.String	切り出した文字列 (指定された文字数を超えている部分が破棄された文字列) 引数で指定された文字列が null の場合は、空文字 ("") を返却する。

f:cut() 使用方法

```
<div id="cut">
    ${f:h(f:cut(bean.originText, 5))}  <!-- (1) -->
</div>
```

項番	説明
(1)	上記例では、引数に指定した文字列の先頭 5 文字を切り出して、画面上に表示している。

注釈: 切り出した文字列を画面上に表示する際は、「*XSS* 対策」として HTML 特殊文字をエスケープする必要がある。上記例では、`f:h()` 関数を使用してエスケープしている。

4.13 Ajax

4.13.1 Overview

本章では、Ajax を利用するアプリケーションの実装方法について説明する。

Ajax とは、以下の処理を非同期に行うための技術の総称である。

- ブラウザ上で行われる画面操作
- 画面操作をトリガーとしたサーバへの HTTP 通信、及び通信結果のユーザインタフェースへの反映

Ajax を使うことで、HTTP 通信中に画面の操作を継続できるため、ユーザビリティの向上を目的として使用されることが多い。
この技術の代表的な適用例としては、検索サイトにおける検索ワードの Suggestion 機能やリアルタイム検索などがあげられる。

4.13.2 How to use

アプリケーションの設定

Ajax 向けのアプリケーションの設定について説明する。

> **警告：StAX(Streaming API for XML) 使用時の DoS 攻撃対策について**
>
> XML 形式のデータについて StAX を使用して解析する場合は、DTD を使った DoS 攻撃を受けないように対応する必要がある。詳細は、CVE-2015-3192 - DoS Attack with XML Input を参照されたい。

Spring MVC の Ajax 関連の機能を有効化するための設定

Ajax 通信時で使用される Content-Type(`application/xml` や `application/json` など) を、Controller のハンドラメソッドでハンドリングできるようにする。

- `spring-mvc.xml`

```
<mvc:annotation-driven /> <!-- (1) -->
```

項番	説明
(1)	<mvc:annotation-driven> 要素が指定されていると、Ajax 通信時で必要となる機能が有効化されている。 そのため、Ajax 通信用に特別な設定を行う必要はない。

注釈: Ajax 通信時で必要となる機能とは、具体的には org.springframework.http.converter.HttpMessageConverter クラスで提供される機能の事をさす。

HttpMessageConverter は、以下の役割をもつ。

- リクエスト Body に格納されているデータから Java オブジェクトを生成する。
- Java オブジェクトからレスポンス Body に書き込むデータを生成する。

<mvc:annotation-driven> 指定時にデフォルトで有効化される HttpMessageConverter は以下の通りである。

項番	クラス名	対象フォーマット	説明
1.	org.springframework.http.converter.json.MappingJackson2HttpMessageConverter	JSON	リクエスト Body 又はレスポンス Body として JSON を扱うための `HttpMessageConverter`。 ブランクプロジェクトでは、Jackson を同封しているため、デフォルトの状態で使用することができる。
2.	org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter	XML	リクエスト Body 又はレスポンス Body として XML を扱うための `HttpMessageConverter`。 JavaSE6 から JAXB2.0 が標準で同封されているため、デフォルトの状態で使用することができる。

注釈: **jackson version 1.x.x から jackson version 2.x.x へ変更する場合の注意点** はこちらを参照されたい。

注釈: **XXE(XML External Entity) 対策について**

Ajax 通信で XML 形式のデータを扱う場合は、XXE(XML External Entity) 対策を行う必要がある。TERASOLUNA Server Framework for Java (5.x) では、XXE 対策が行われている Spring MVC(3.2.10.RELEASE 以上) に依存しているため、個別に対策を行う必要はない。

Controller の実装

以降で説明するサンプルコードの前提は以下の通りである。

- 応答データの形式には JSON を使用する。

- クライアント側には、JQuery を使用する。バージョンは執筆時点の 1.x 系の最新バージョン (1.10.2) を使用する。

警告: 循環参照への対策

`HttpMessageConverter` を使用して JavaBean を JSON や XML 形式にシリアライズする際に、相互参照関係のオブジェクトをプロパティに保持していると、循環参照となり `StackOverflowError` や `OutOfMemoryError` などが発生するので、注意が必要である。

循環参照を回避するためには、

- Jackson を使用して JSON 形式にシリアライズする場合は、シリアライズ対象から除外するプロパティに`@com.fasterxml.jackson.annotation.JsonIgnore` アノテーション
- JAXB を使用して XML 形式にシリアライズする場合は、シリアライズ対象から除外するプロパティに `javax.xml.bind.annotation.XmlTransient` アノテーション

を付与すればよい。

以下に Jackson を使用して JSON 形式にシリアライズする際の回避例を示す。

```
public class Order {
    private String orderId;
    private List<OrderLine> orderLines;
    // ...
}
```

```
public class OrderLine {
    @JsonIgnore
    private Order order;
    private String itemCode;
    private int quantity;
    // ...
}
```

項番	説明
(1)	シリアライズ対象から除外するプロパティに対して`@JsonIgnore` アノテーションを付与する。

データを取得する

Ajax を使ってデータを取得する方法について説明する。

下記例は、検索ワードに一致する情報を一覧として返却する Ajax 通信となっている。

- リクエストデータを受け取るための JavaBean

```
// (1)
public class SearchCriteria implements Serializable {

    // omitted

    private String freeWord; // (2)

    // omitted setter/getter

}
```

項番	説明
(1)	リクエストデータを受け取るための JavaBean を作成する。
(2)	プロパティ名は、リクエストパラメータのパラメータ名と一致させる。

- 返却するデータを格納する JavaBean

```
// (3)
public class SearchResult implements Serializable {

    // omitted

    private List<XxxEntity> list;

    // omitted setter/getter

}
```

項番	説明
(3)	返却するデータを格納するための JavaBean を作成する。

- Controller

```
@RequestMapping(value = "search", method = RequestMethod.GET) // (4)
@ResponseBody // (5)
public SearchResult search(@Validated SearchCriteria criteria) { // (6)

    SearchResult searchResult = new SearchResult(); // (7)

    // (8)
    // omitted

    return searchResult; // (9)
}
```

項番	説明
(4)	`@RequestMapping` アノテーションの method 属性に `RequestMethod.GET` を指定する。
(5)	`@org.springframework.web.bind.annotation.ResponseBody` アノテーションを付与する。 このアノテーションを付与することで、返却したオブジェクトが JSON 形式に marshal され、レスポンス Body に設定される。
(6)	リクエストデータを受け取るための JavaBean を引数に指定する。 入力チェックが必要な場合は、 `@Validated` を指定する。入力チェックのエラーハンドリングについては、「 入力エラーのハンドリング 」を参照されたい。 入力チェックの詳細については、「 入力チェック 」を参照されたい。
(7)	返却するデータを格納する JavaBean のオブジェクトを生成する。
(8)	データを検索し、(7) で生成したオブジェクトに検索結果を格納する。 上記例では、実装は省略している。
(9)	レスポンス Body に marshal するためのオブジェクトを返却する。

- HTML(JSP)

```
<!-- omitted -->

<meta name="contextPath" content="${pageContext.request.contextPath}" />
```

```
<!-- omitted -->

<!-- (10)  -->
<form id="searchForm">
  <input name="freeWord" type="text">
  <button onclick="return searchByFreeWord()">Search</button>
</form>
```

項番	説明
(10)	検索条件を入力するためのフォーム。 上記例では、検索条件を入力するためのテキストボックスと検索ボタンをもっている。

```
<!-- (11) -->
<script type="text/javascript"
    src="${pageContext.request.contextPath}/resources/vendor/jquery/jquery-1.
↪10.2.js">
</script>
```

項番	説明
(11)	JQuery の JavaScript ファイルを読み込む。 上記例では、JQuery の JavaScript ファイルを読み込むために、`/resources/vendor/jquery/jquery-1.10.2.js` というパスに対してリクエストが送信される。

注釈: JQuery の JavaScript ファイルを読み込みための設定は、以下の通り。以下はブランクプロジェクトで提供されている設定値である。

- `spring-mvc.xml`

```
<!-- (12) -->
<mvc:resources mapping="/resources/**"
    location="/resources/,classpath:META-INF/resources/"
    cache-period="#{60 * 60}" />
```

項番	説明
(12)	リソースファイル (JavaScript ファイル, Stylesheet ファイル, 画像ファイルなど) を公開するための設定。 上記設定例では、 `/resources/` から始まるパスに対してリクエストがあった場合に、war ファイル内の `/resources/` ディレクトリ又はクラスパス内の `/META-INF/resources/` ディレクトリに格納されているファイルが応答される。

上記設定の場合、JQuery の JavaScript ファイルは以下の何れかのパスに配置する必要がある。

- war ファイル内の `/resources/vendor/jquery/jquery-1.10.2.js`
  プロジェクト内のパスで表現すると、
  `src/main/webapp/resources/vendor/jquery/jquery-1.10.2.js` となる。
- クラスパス内の `/META-INF/resources/vendor/jquery/jquery-1.10.2.js`
  プロジェクト内のパスで表現すると、 `src/main/resources/META-INF/resources/vendor/jquery/jquery-1.10.2.js` と
  なる。

- JavaScript

```
var contextPath = $("meta[name='contextPath']").attr("content");

// (13)
function searchByFreeWord() {
    $.ajax(contextPath + "/ajax/search", {
        type : "GET",
        data : $("#searchForm").serialize(),
        dataType : "json", // (14)

    }).done(function(json) {
```

```
        // (15)
        // render search result
        // omitted

    }).fail(function(xhr) {
        // (16)
        // render error message
        // omitted

    });
    return false;
}
```

項番	説明
(13)	フォームに指定された検索条件をリクエストパラメータに変換し、GET メソッドで `/ajax/search` に対してリクエストを送信する Ajax 関数。 上記例では、ボタンの押下を Ajax 通信のトリガーとしているが、テキストボックスのキーダウンやキーアップをトリガーとすることでリアルタイム検索などを実現することができる。
(14)	レスポンスとして受け取るデータ形式を指定する。 上記例では `json` を指定しているため、Accept ヘッダーに `application/json` が設定される。
(15)	Ajax 通信が正常終了した時 (Http ステータスコードが `200` の時) の処理を実装する。 上記例では、実装は省略している。
(16)	Ajax 通信が正常終了しなかった時 (Http ステータスコードが `4xx` や `5xx` の時) の処理を実装する。 上記例では、実装は省略している。 エラー処理の実装例は、 *フォームデータを POST する* を参照されたい。

ちなみに: 上記例では Web アプリケーションのコンテキストパス (`${pageContext.request.contextPath}`) を HTML の `<meta>` 要素に設定しておくことで、JavaScript のコードから JSP の

コードを排除している。

上記検索フォームの「Search」ボタンを押下した際には、以下のような通信が発生する。
ポイントとなる部分にハイライトを設けている。

- リクエストデータ

```
GET /terasoluna-gfw-web-blank/ajax/search?freeWord= HTTP/1.1
Host: localhost:9999
Connection: keep-alive
Accept: application/json, text/javascript, */*; q=0.01
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like␣
↪Gecko) Chrome/30.0.1599.101 Safari/537.36
Referer: http://localhost:9999/terasoluna-gfw-web-blank/ajax/xxe
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,ja;q=0.6
Cookie: JSESSIONID=3A486604D7DEE62032BA6C073FC6BE9F
```

- レスポンスデータ

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: a8fb8fefaaf64ee2bffc2b0f77050226
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Fri, 25 Oct 2013 13:52:55 GMT

{"list":[]}
```

フォームデータを POST する

Ajax を使ってフォームのデータを POST し、処理結果を取得する方法について説明する。

下記例は、2 つの数値を受け取り、加算結果を返却する Ajax 通信となっている。

- フォームデータを受け取るための JavaBean

```
// (1)
public class CalculationParameters implements Serializable {

    // omitted

    private Integer number1;

    private Integer number2;

    // omitted setter/getter

}
```

項番	説明
(1)	フォームデータを受け取るための JavaBean を作成する。

- 処理結果を格納する JavaBean

```
// (2)
public class CalculationResult implements Serializable {

    // omitted

    private int resultNumber;

    // omitted setter/getter

}
```

項番	説明
(2)	処理結果を格納するための JavaBean を作成する。

警告: 電文から Java Bean にデシリアライズする際、プロパティにジェネリクスやインターフェイスを使用しているなどの理由で型を特定できない場合は`@com.fasterxml.jackson.annotation.JsonTypeInfo` アノテーションを付与する。`@JsonTypeInfo` アノテーションを付与したプロパティをシリアライズすると JSON に型情報が出力され、これを読み取ってデシリアライズが行われる。

ただし、`@JsonTypeInfo` アノテーションの use 属性に `Id.CLASS` や `Id.MINIMAL_CLASS` を使用すると、JSON に出力されたクラス名を元にデシリアライズが行われるため、これにより不正にリモートコードが実行される危険がある。このため、(信頼できない送信元を含み得る) 不特定多数からの電文を受け付ける前提のシステムにおいては、`Id.CLASS` や `Id.MINIMAL_CLASS` を指定してはならない。

なお、`ObjectMapper` の `defaultTyping` を利用すると、上記のようなデシリアライズ時の型判断をアプリケーション全体に適用することが可能である。こちらも合わせて注意されたい。

- Controller

```
@RequestMapping("xxx")
@Controller
public class XxxController {

    @RequestMapping(value = "plusForForm", method = RequestMethod.POST) // (3)
    @ResponseBody
    public CalculationResult plusForForm(
        @Validated CalculationParameters params) { // (4)
        CalculationResult result = new CalculationResult();
        int sum = params.getNumber1() + params.getNumber2();
        result.setResultNumber(sum); // (5)
        return result; // (6)
    }

    // omitted

}
```

項番	説明
(3)	`@RequestMapping` アノテーションの method 属性に `RequestMethod.POST` を指定する。
(4)	フォームデータを受け取るための JavaBean を引数に指定する。 入力チェックが必要な場合は、 `@Validated` を指定する。入力チェックのエラーハンドリングについては、「 入力エラーのハンドリング 」を参照されたい。 入力チェックの詳細については、「 入力チェック 」を参照されたい。
(5)	処理結果を格納するオブジェクトに処理結果を格納する。 上記例では、フォームオブジェクトから取得した 2 つの数値を加算した結果を格納している。
(6)	レスポンス Body に marshal するためのオブジェクトを返却する。

- HTML(JSP)

```
<!-- omitted -->

<meta name="contextPath" content="${pageContext.request.contextPath}" />

<sec:csrfMetaTags />

<!-- omitted -->

<!-- (7)  -->
<form id="calculationForm">
    <input name="number1" type="text">+
    <input name="number2" type="text">
    <button onclick="return plus()">=</button>
    <span id="calculationResult"></span> <!-- (8) -->
```

```
</form>
```

項番	説明
(7)	計算対象の数値を入力するためのフォーム。
(8)	計算結果を表示するための領域。 上記例では、通信成功時には計算結果が表示され、通信失敗時には計算結果がクリアされる。

- JavaScript

```
var contextPath = $("meta[name='contextPath']").attr("content");

// (9)
var csrfToken = $("meta[name='_csrf']").attr("content");
var csrfHeaderName = $("meta[name='_csrf_header']").attr("content");
$(document).ajaxSend(function(event, xhr, options) {
    xhr.setRequestHeader(csrfHeaderName, csrfToken);
});

// (10)
function plus() {
    $.ajax(contextPath + "/ajax/plusForForm", {
        type : "POST",
        data : $("#calculationForm").serialize(),
        dataType : "json"
    }).done(function(json) {
        $("#calculationResult").text(json.resultNumber);

    }).fail(function(xhr) {
        // (11)
        var messages = "";
        // (12)
        if(400 <= xhr.status && xhr.status <= 499){
            // (13)
            var contentType = xhr.getResponseHeader('Content-Type');
            if (contentType != null && contentType.indexOf("json") != -1) {
                // (14)
```

```
                json = $.parseJSON(xhr.responseText);
                $(json.errorResults).each(function(i, errorResult) {
                    messages += ("<div>" + errorResult.message + "</div>");
                });
            } else {
                // (15)
                messages = ("<div>" + xhr.statusText + "</div>");
            }
        }else{
            // (16)
            messages = ("<div>" + "System error occurred." + "</div>");
        }
        // (17)
        $("#calculationResult").html(messages);
    });

    return false;
}
```

項番	説明
(9)	POST メソッドでリクエストを行う場合、CSRF トークンを HTTP ヘッダに設定して送信する必要がある。 上記例では、`<sec:csrfMetaTags />`を利用して `<meta>` 要素に CSRF トークンヘッダー名と CSRF トークン値を設定し、JavaScript で値を取得するようにしている。 CSRF 対策の詳細については、「*CSRF 対策*」を参照されたい。
(10)	フォームに指定された数値をリクエストパラメータに変換し、POST メソッドで `/ajax/plusForForm` に対してリクエストを送信する Ajax 関数。 上記例では、ボタンの押下を Ajax 通信のトリガーとしているが、テキストボックスのロストフォーカスをトリガーとすることでリアルタイム計算を実現することができる。
(11)	エラー処理の実装例を以下に示す。 サーバ側のエラーハンドリング処理の実装例については、入力エラーのハンドリング を参照されたい。

Continued on next page

表 4.53 – continued from previous page

項番	説明
(12)	HTTP のステータスコードを判定し、どのようなエラーが発生したか判定する。 HTTP のステータスコードは、 XMLHttpRequest オブジェクトの `status` フィールドに格納されている。
(13)	レスポンスされたデータが JSON 形式か判定を行う。 上記例では、レスポンスヘッダの Content-Type に設定されている値を参照して、レスポンスされたデータの形式をチェックしている。 形式をチェックしておかないと、JSON 以外の形式で応答された際に、JSON オブジェクトにデシリアライズする処理でエラーが発生することになる。 サーバ側のエラーハンドリングを簡易的に行っていると、HTML 形式のページが返却されることがある。
(14)	レスポンスデータを JSON オブジェクトにデシリアライズする。 レスポンスデータは、 XMLHttpRequest オブジェクトの `responseText` フィールドに格納されている。 上記例では、デシリアライズした JSON オブジェクトからエラー情報を取得し、エラーメッセージを組み立てている。
(15)	レスポンスされたデータが JSON 形式以外だった場合の処理を行う。 上記例では、HTTP のステータステキストをエラーメッセージに格納している。 HTTP のステータステキストは、 XMLHttpRequest オブジェクトの `statusText` フィールドに格納されている。
(16)	サーバエラー時の処理を行う。 上記例では、システムエラーが発生したことを通知するメッセージをエラーメッセージに格納している。
(17)	エラー時の描画処理を行う。 上記例では、計算結果を表示するための領域に、エラーメッセージを表示している。

> **警告:** 上記例では、Ajax の通信処理、DOM 操作処理 (描画処理)、エラー処理を同じ function 内で行っているが、これらの処理は分離して実装することを推奨する。

ちなみに: 上記例では`<sec:csrfMetaTags />`を利用して、CSRF トークン値と CSRF トークンヘッダー名を HTML の `<meta>` 要素に設定しておくことで、JavaScript のコードから JSP のコードを排除している。*Ajax 使用時の連携*を参照されたい。

尚、CSRF トークン値と CSRF トークンヘッダー名はそれぞれ`${_csrf.token}`と`${_csrf.headerName}`を用いても取得可能である。

上記検索フォームの「=」ボタンを押下した際には、以下のような通信が発生する。
ポイントとなる部分にハイライトを設けている。

- リクエストデータ

```
POST /terasoluna-gfw-web-blank/ajax/plusForForm HTTP/1.1
Host: localhost:9999
Connection: keep-alive
Content-Length: 19
Accept: application/json, text/javascript, */*; q=0.01
Origin: http://localhost:9999
X-CSRF-TOKEN: a5dd1858-8a4f-4ecc-88bd-a326388ab5c9
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like␣
↪Gecko) Chrome/30.0.1599.101 Safari/537.36
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Referer: http://localhost:9999/terasoluna-gfw-web-blank/ajax/xxe
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,ja;q=0.6
Cookie: JSESSIONID=3A486604D7DEE62032BA6C073FC6BE9F

number1=1&number2=2
```

- レスポンスデータ

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: c2d5066d0fa946f584536775f07d1900
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Fri, 25 Oct 2013 14:27:55 GMT

{"resultNumber":3}
```

- エラー時のレスポンスデータ下記のレスポンスデータは、入力エラーが発生時のものである。

```
HTTP/1.1 400 Bad Request
Server: Apache-Coyote/1.1
X-Track: cecd7b4d746249178643b7110b0eaa74
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 04 Dec 2013 15:06:01 GMT
Connection: close

{"errorResults":[{"code":"NotNull","message":"\"number2\"maynotbenull.",
↪"itemPath":"number2"},{"code":"NotNull","message":"\"number1\"maynotbenull.
↪","itemPath":"number1"}]}
```

フォームデータを JSON として POST する

Ajax を使ってフォームのデータを JSON 形式に変換してから POST し、処理結果を取得する方法について説明する。

「フォームデータを POST する」方法との差分部分について説明する。

- Controller

```
@RequestMapping("xxx")
@Controller
public class XxxController {

    @RequestMapping(value = "plusForJson", method = RequestMethod.POST)
    @ResponseBody
    public CalculationResult plusForJson(
            @Validated @RequestBody CalculationParameters params) { // (1)
        CalculationResult result = new CalculationResult();
        int sum = params.getNumber1() + params.getNumber2();
```

```
        result.setResultNumber(sum);
        return result;
    }

    // omitted

}
```

項番	説明
(1)	フォームデータを受け取るための JavaBean の引数アノテーションとして、`@org.springframework.web.bind.annotation.RequestBody` アノテーションを付与する。 このアノテーションを付与することで、リクエスト Body に格納されている JSON 形式のデータが unmarshal され、オブジェクトに変換される。 入力チェックが必要な場合は、 `@Validated` を指定する。入力チェックのエラーハンドリングについては、「 入力エラーのハンドリング 」を参照されたい。 入力チェックの詳細については、「 入力チェック 」を参照されたい。

- JavaScript/HTML(JSP)

```
// (2)
function toJson($form) {
    var data = {};
    $($form.serializeArray()).each(function(i, v) {
        data[v.name] = v.value;
    });
    return JSON.stringify(data);
}

function plus() {

    $.ajax(contextPath + "/ajax/plusForJson", {
        type : "POST",
        contentType : "application/json;charset=utf-8", // (3)
        data : toJson($("#calculationForm")), // (2)
        dataType : "json",
        beforeSend : function(xhr) {
            xhr.setRequestHeader(csrfHeaderName, csrfToken);
```

```
        }

    }).done(function(json) {
        $("#calculationResult").text(json.resultNumber);

    }).fail(function(xhr) {
        $("#calculationResult").text("");

    });
    return false;
}
```

項番	説明
(2)	フォーム内の input 項目を JSON 形式の文字列にするための関数。
(3)	リクエスト Body に JSON を格納するので、Content-Type のメディアタイプを `application/json` にする。

上記検索フォームの「=」ボタンを押下した際には、以下のような通信が発生する。
ポイントとなる部分にハイライトを設けている。

- リクエストデータ

```
POST /terasoluna-gfw-web-blank/ajax/plusForJson HTTP/1.1
Host: localhost:9999
Connection: keep-alive
Content-Length: 31
Accept: application/json, text/javascript, */*; q=0.01
Origin: http://localhost:9999
X-CSRF-TOKEN: 9d4f1e0c-c500-43f3-9125-a7a131ff88fa
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like␣
↪Gecko) Chrome/30.0.1599.101 Safari/537.36
Content-Type: application/json;charset=UTF-8
Referer: http://localhost:9999/terasoluna-gfw-web-blank/ajax/xxe?
```

```
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,ja;q=0.6
Cookie: JSESSIONID=CECD7A6CB0431266B8D1173CCFA66B95

{"number1":"34","number2":"56"}
```

入力エラーのハンドリング

入力値に不正な値が指定された場合のエラーハンドリング方法について説明する。

入力エラーのハンドリング方法は、大きく分けて以下の２つに分類される。

- 例外ハンドリング用のメソッドを用意してエラー処理を行う。
- Controller のハンドラメソッドの引数として `org.springframework.validation.BindingResult` を受け取り、エラー処理を行う。

BindException のハンドリング

`org.springframework.validation.BindException` は、リクエストパラメータとして送信したデータを JavaBean にバインドする際に、入力値に不正な値が指定された場合に発生する例外クラスである。

GET 時のリクエストパラメータや、フォームデータを `application/x-www-form-urlencoded` の形式として受け取る場合は、`BindException` の例外ハンドリングが必要となる。

- Controller

```
@RequestMapping("xxx")
@Controller
public class XxxController {

    // omitted

    @ExceptionHandler(BindException.class) // (1)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST) // (2)
    @ResponseBody // (3)
    public ErrorResults handleBindException(BindException e, Locale locale) {
↪// (4)
        // (5)
        ErrorResults errorResults = new ErrorResults();
        for (FieldError fieldError : e.getBindingResult().getFieldErrors()) {
            errorResults.add(fieldError.getCode(),
```

```
                    messageSource.getMessage(fieldError, locale),
                        fieldError.getField());
        }
        for (ObjectError objectError : e.getBindingResult().
↪getGlobalErrors()) {
            errorResults.add(objectError.getCode(),
                    messageSource.getMessage(objectError, locale),
                        objectError.getObjectName());
        }
        return errorResults;
    }

    // omitted

}
```

項番	説明
(1)	Controller にエラーハンドリング用メソッドを定義する。 エラーハンドリング用のメソッドには、`@org.springframework.web.bind.annotation.ExceptionHandler` アノテーションを付与し、 value 属性にハンドリングする例外の型を指定する。 上記例では、 ハンドリング対象の例外として `BindException.class` を指定している。
(2)	応答する HTTP ステータス情報を指定する。 上記例では、 `400` (Bad Request) を指定している。
(3)	返却したオブジェクトをレスポンス Body に書き込むため、 `@ResponseBody` アノテーションを付与する。
(4)	エラーハンドリング用のメソッドの引数として、ハンドリング対象の例外クラスを宣言する。
(5)	エラー処理を実装する。 上記例では、エラー情報を返却するための JavaBean を生成し、返却している。

ちなみに: エラー処理としてメッセージを生成する際に国際化を意識する必要がある場合は、Localeオブジェクトを引数として受け取ることができる。

- エラー情報を保持する JavaBean

```
// (6)
public class ErrorResult implements Serializable {

    private static final long serialVersionUID = 1L;

    private String code;

    private String message;

    private String itemPath;

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String getItemPath() {
        return itemPath;
    }

    public void setItemPath(String itemPath) {
        this.itemPath = itemPath;
    }

}
```

```
// (7)
public class ErrorResults implements Serializable {

```

```
    private static final long serialVersionUID = 1L;

    private List<ErrorResult> errorResults = new ArrayList<ErrorResult>();

    public List<ErrorResult> getErrorResults() {
        return errorResults;
    }

    public void setErrorResults(List<ErrorResult> errorResults) {
        this.errorResults = errorResults;
    }

    public ErrorResults add(String code, String message) {
        ErrorResult errorResult = new ErrorResult();
        errorResult.setCode(code);
        errorResult.setMessage(message);
        errorResults.add(errorResult);
        return this;
    }

    public ErrorResults add(String code, String message, String itemPath) {
        ErrorResult errorResult = new ErrorResult();
        errorResult.setCode(code);
        errorResult.setMessage(message);
        errorResult.setItemPath(itemPath);
        errorResults.add(errorResult);
        return this;
    }

}
```

項番	説明
(6)	エラー情報を 1 件保持するための JavaBean。
(7)	エラー情報を 1 件保持する JavaBean を複数件保持するための JavaBean。 (6) の JavaBean をリストとして保持している。

MethodArgumentNotValidException のハンドリング

org.springframework.web.bind.MethodArgumentNotValidException は、@RequestBody アノテーションを使用してリクエスト Body に格納されているデータを JavaBean にバインドする際に、入力値に不正な値が指定された場合に発生する例外クラスである。

application/json や application/xml などの形式として受け取る場合は、MethodArgumentNotValidException の例外ハンドリングが必要となる。

- Controller

```
@ExceptionHandler(MethodArgumentNotValidException.class) // (1)
@ResponseStatus(value = HttpStatus.BAD_REQUEST)
@ResponseBody
public ErrorResults handleMethodArgumentNotValidException(
        MethodArgumentNotValidException e, Locale locale) { // (1)
    ErrorResults errorResults = new ErrorResults();

    // implement error handling.
    // omitted

    return errorResults;
}
```

項番	説明
(1)	エラーハンドリング対象の例外として MethodArgumentNotValidException.class を指定する。 上記以外は BindException と同様。

HttpMessageNotReadableException のハンドリング

org.springframework.http.converter.HttpMessageNotReadableException は、@RequestBody アノテーションを使用してリクエスト Body に格納されているデータを JavaBean にバインドする際に、Body に格納されているデータから JavaBean を生成できなかった場合に発生する例外クラスである。

application/json や application/xml などの形式として受け取る場合は、MethodArgumentNotValidException の例外ハンドリングが必要となる。

注釈: 具体的なエラー原因は、使用する `HttpMessageConverter` や利用するライブラリの実装によって異なる。

JSON 形式のデータについて Jackson を使用して JavaBean に変換する `MappingJackson2HttpMessageConverter` の実装では、Integer 項目に数値以外の文字列を指定すると、`HttpMessageNotReadableException` が発生する。

- Controller

```
@ExceptionHandler(HttpMessageNotReadableException.class) // (1)
@ResponseStatus(value = HttpStatus.BAD_REQUEST)
@ResponseBody
public ErrorResults handleHttpMessageNotReadableException(
        HttpMessageNotReadableException e, Locale locale) {  // (1)
    ErrorResults errorResults = new ErrorResults();

    // implement error handling.
    // omitted

    return errorResults;
}
```

項番	説明
(1)	エラーハンドリング対象の例外として `HttpMessageNotReadableException.class` を指定する。 上記以外は `BindException` と同様。

BindingResult を使用したハンドリング

正常終了時に返却する JavaBean と入力エラー時に返却する JavaBean の型が同じ場合は、`BindingResult` をハンドラメソッドの引数として受け取ることでエラーハンドリングすることができる。

この方法は、リクエストデータの形式に関係なく使用することができる。

ハンドラメソッドの引数として `BindingResult` を指定しない場合は、前述した例外をハンドリングする方法でエラー処理を実装する必要がある。

- Controller

```
@RequestMapping(value = "plus", method = RequestMethod.POST)
@ResponseBody
public CalculationResult plus(
        @Validated @RequestBody CalculationParameters params,
        BindingResult bResult) { // (1)
    CalculationResult result = new CalculationResult();
    if (bResult.hasErrors()) { // (2)

        // (3)
        // implement error handling.
        // omitted

        return result; // (4)
    }
    int sum = params.getNumber1() + params.getNumber2();
    result.setResultNumber(sum);
    return result;
}
```

項番	説明
(1)	ハンドラメソッドの引数として `BindingResult` を宣言する。 `BindingResult` は入力チェック対象の JavaBean の直後に宣言する必要がある。
(2)	入力値のエラー有無を判定する。
(3)	入力値にエラーがある場合は、入力エラー時のエラー処理を行う。 上記例ではエラー処理は省略しているが、エラーメッセージの設定などが行われる想定である。
(4)	処理結果を返却する。

注釈: 上記例では、正常時及びエラー時共にレスポンスの HTTP ステータスコードは `200` (OK) が返却される。HTTP ステータスコードを処理結果によってわける必要がある場合は、 `org.`

springframework.http.ResponseEntity を返却値とすることで実現可能である。別のアプローチとしては、ハンドラメソッドの引数として BindingResult を指定せず、前述した例外をハンドリングする方法でエラー処理を実装する方法がある。

```
@RequestMapping(value = "plus", method = RequestMethod.POST)
@ResponseBody
public ResponseEntity<CalculationResult> plus(
        @Validated @RequestBody CalculationParameters params,
        BindingResult bResult) {
    CalculationResult result = new CalculationResult();
    if (bResult.hasErrors()) {

        // implement error handling.
        // omitted

        // (1)
        return ResponseEntity.badRequest().body(result);
    }
    // omitted

    // (2)
    return ResponseEntity.ok().body(result);
}
```

項番	説明
(1)	入力エラー時の応答データと HTTP ステータスを返却する。
(2)	正常終了時の応答データと HTTP ステータスを返却する。

業務エラーのハンドリング

業務エラーのエラーハンドリング方法について説明する。

業務エラーのハンドリング方法は大きく分けて以下の 2 つに分類される。

- 業務例外ハンドリング用のメソッドを用意してエラー処理を行う。
- Controller のハンドラメソッド内で業務例外を catch してエラー処理を行う。

例外ハンドリング用のメソッドで業務例外をハンドリング

入力エラーと同様、例外ハンドリング用のメソッドを用意して業務例外をハンドリングする。
複数のハンドラメソッドに対するリクエストで同じエラー処理を実装する必要がある場合、この方法でエラーハンドリングすることを推奨する。

- Controller

```
@ExceptionHandler(BusinessException.class) // (1)
@ResponseStatus(value = HttpStatus.CONFLICT) // (2)
@ResponseBody
public ErrorResults handleHttpBusinessException(BusinessException e, // (1)
        Locale locale) {
    ErrorResults errorResults = new ErrorResults();

    // implement error handling.
    // omitted

    return errorResults;
}
```

項番	説明
(1)	エラーハンドリング対象の例外として `BusinessException.class` を指定する。 上記以外は入力エラーの `BindException` のハンドリング方法と同様。
(2)	応答する HTTP ステータス情報を指定する。 上記例では、 `409` (Conflict) を指定している。

ハンドラメソッド内で業務例外をハンドリング

業務エラーが発生する処理を try 句で囲み、業務例外を catch する。
エラー処理がリクエスト毎に異なる場合は、この方法でエラーハンドリングすることになる。

- Controller

```
@RequestMapping(value = "plus", method = RequestMethod.POST)
@ResponseBody
public ResponseEntity<CalculationResult> plusForJson(
        @Validated @RequestBody CalculationParameters params) {
    CalculationResult result = new CalculationResult();

    // omitted

    // (1)
    try {

        // call service method.
        // omitted

     // (2)
    } catch (BusinessException e) {

        // (3)
        // implement error handling.
        // omitted

        return ResponseEntity.status(HttpStatus.CONFLICT).body(result);
    }

    // omitted

    return ResponseEntity.ok().body(result);
}
```

項番	説明
(1)	業務例外が発生するメソッド呼び出しを try 句で囲む。
(2)	業務例外を catch する。
(3)	業務例外エラー時のエラー処理を行う。 上記例ではエラー処理は省略しているが、エラーメッセージの設定などが行われる想定である。

4.14 ヘルスチェック

4.14.1 Overview

本節では、ヘルスチェックについて説明する。

ロードバランサの負荷分散と縮退運転

Web システムが大量のユーザからリクエストを受けることを想定し、ロードバランサ（以下、LB と言う）を利用する。

LB は複数のサーバにリクエストを割り振ることで Web システムにかかる負荷を分散する装置であり、サーバの追加・削除により、柔軟に Web システムの処理能力を変更できるのが特徴である。

ヘルスチェックは、LB がリクエストを割り振る各サーバの稼働状況を監視する機能である。LB はヘルスチェックを利用し、異常を検知したサーバにリクエストを割り振らず、正常に稼働しているサーバに割り振る。これにより、特定のサーバで障害が発生した場合も、Web システムを停止することなく運用することが可能である。（これを縮退運転と呼ぶ）

以下の例では、LB は 3 台のサーバを管理し、リクエストを割り振っている。

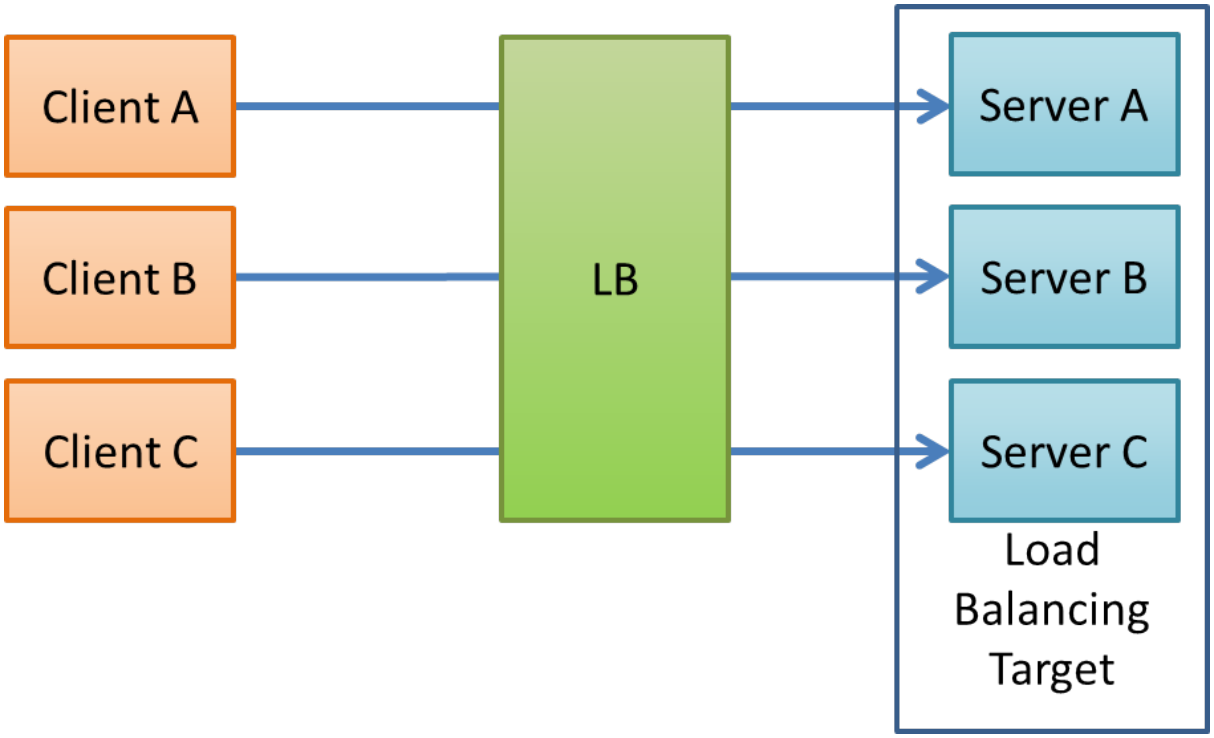


図 4.22 **Picture - About Load Balancing**

LB は定期的にサーバにリクエストを送信し、サーバから返されたステータスコードやレスポンスを確認することで、サーバの稼働状況を監視する。図のサーバ A で異常が発生した場合、LB がそれを検知し、サーバ A にリクエストを割り振らないようにする。

元々サーバ A に接続していたクライアント A は、LB によって、他のサーバ (ここではサーバ B) にリクエストを割り振られる。

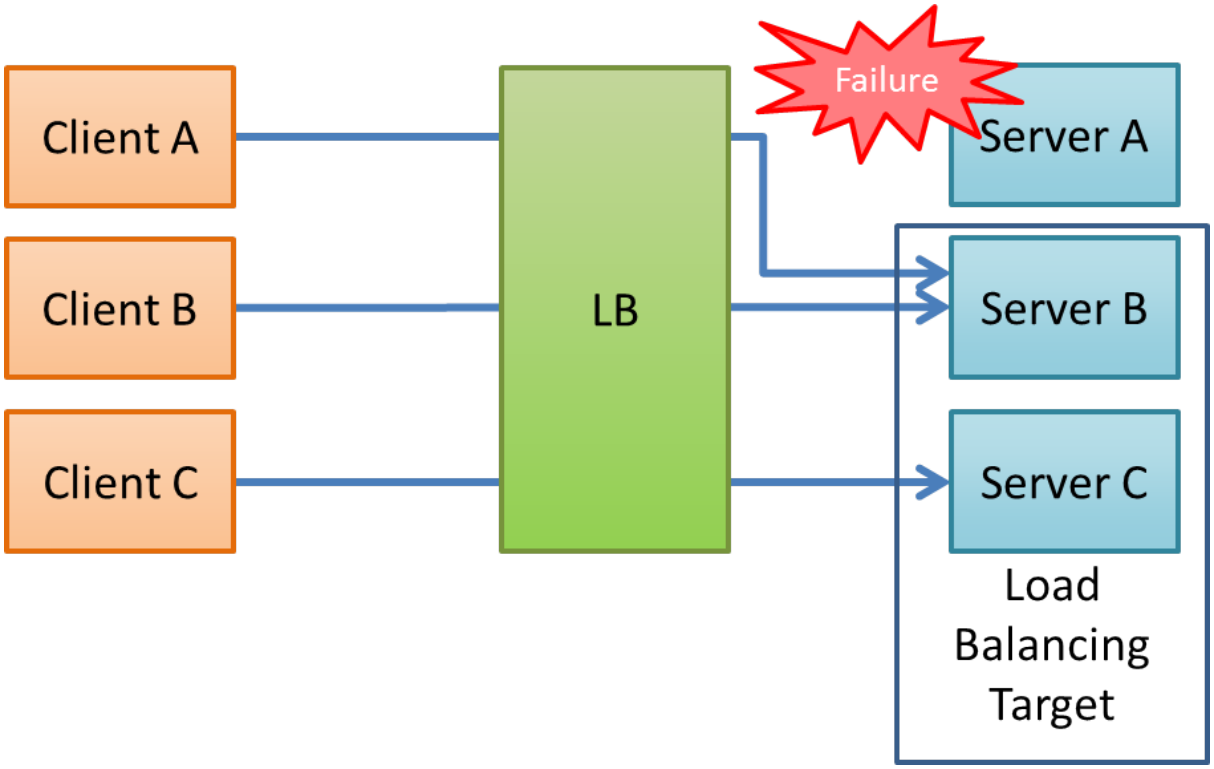


図 4.23　**Picture - About Fallback**

ヘルスチェックの種類

LB が行うヘルスチェックには、さまざまな種類がある。以下に例を示す。

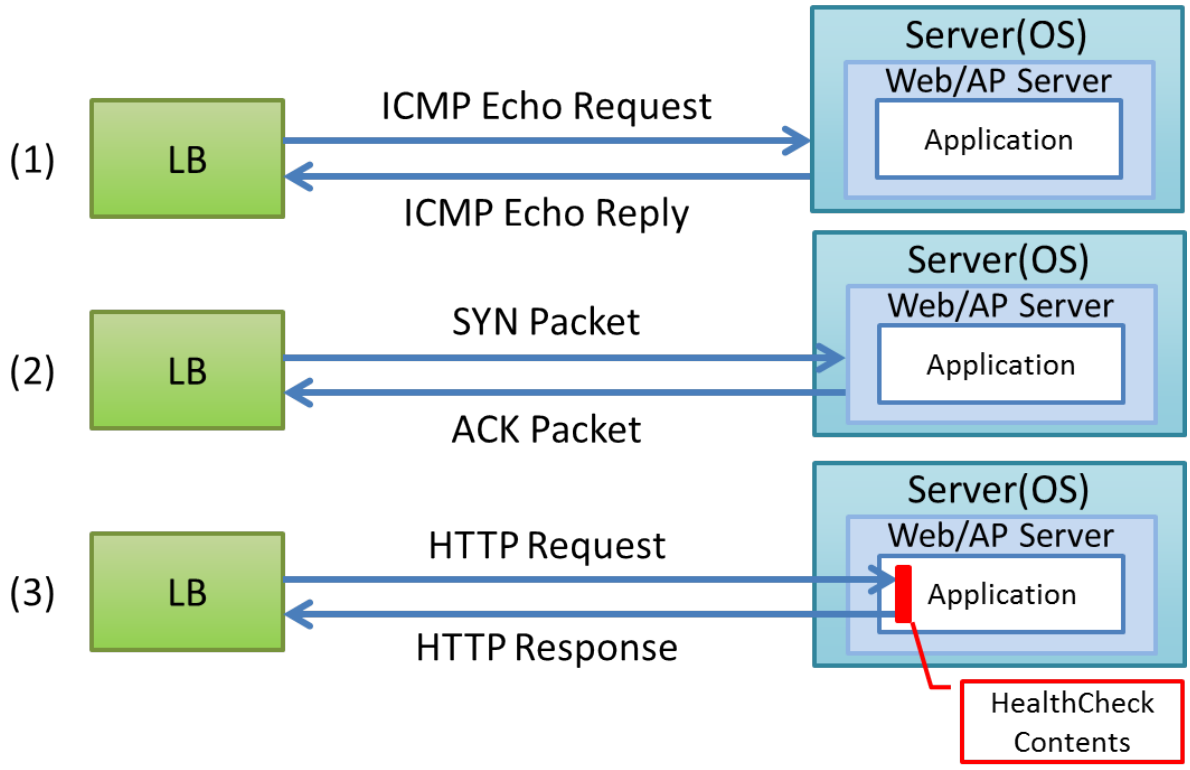


図 4.24 **Picture - HealthCheck Example**

項番	ヘルスチェックの種類	詳細
(1)	PING でのヘルスチェック	OSI 参照モデルのネットワーク層レベルで稼働状況を確認する。サーバ (OS) に対して PING を送信し、応答があれば稼働していると判断する。
(2)	TCP/UDP でのヘルスチェック	OSI 参照モデルのトランスポート層レベルで稼働状況を確認する。Web/AP サーバの TCP ポート（または UDP ポート）にリクエストを送信し、応答があれば稼働していると判断する。
(3)	アプリケーションでのヘルスチェック	OSI 参照モデルのアプリケーション層レベルで稼働状況を確認する。Web/AP サーバ上で稼働するアプリケーションに HTTP リクエストを送信し、応答が正常であれば稼働していると判断する。

PING や TCP/UDP でのヘルスチェックでは、アプリケーションの稼働状況までは確認できない。Web アプリケーションを対象とした場合は、サーバ (OS) や Web/AP サーバが稼働しているだけでは不十分であり、アプ

リケーションが稼働している必要がある。

そのため本ガイドラインでは、アプリケーションでのヘルスチェックを行うことを推奨する。

本ガイドラインで示すヘルスチェックの構成

本ガイドラインでは、アプリケーションでのヘルスチェックを行うための、アプリケーションの実装例を紹介する。

具体的には、LB からのリクエストを受け取る、以下の図のような構成のハンドラを実装する。

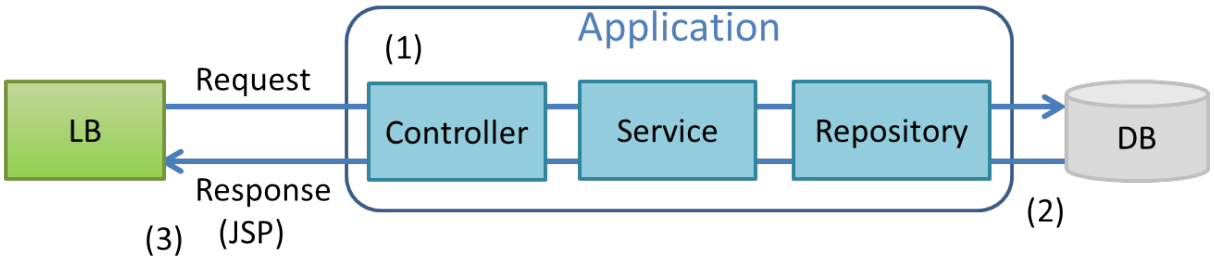


図 4.25 **Picture - HealthCheck Configuration**

項番	説明
(1)	LB からのリクエストを受け、Controller、Service、Repository を実行する。 単に稼働状況を確認する、という点では、よりシンプルにヘルスチェックを実現する方法も存在する。しかし本ガイドラインでは、ヘルスチェックによってアプリケーションが使用している仕組みやフレームワーク自体が正しく動作していることも確認するべく、対象アプリの使用技術構成にできる限り近づけるために、Controller、Service、Repository を実装する。
(2)	Repository から SQL を発行し、データベースが稼働していることを確認する。 これは、データベースアクセスを伴うアプリケーションの場合、アプリケーションが稼働していても、データベースに異常がある場合は正常に業務を行うことができないためである。
(3)	レスポンスを返す View として JSP を使用する。 本ガイドラインでは JSP を例にとって説明するが、REST や SOAP を用いる場合など、アプリケーションの特性に合わせて通信方式やレスポンス形式は適宜変更すること。詳細は、*RESTful Web Service* や、*SOAP Web Service（サーバ/クライアント）*を参照されたい。

本ガイドラインの実装例で返却されるステータスコードおよびレスポンスは以下の通りである。

ヘルスチェック処理結果	ステータスコード	レスポンス内容
成功	200(正常)	`OK.` の 3 文字
エラー発生	例外ハンドリング機能で設定されたステータスコード	例外ハンドリング機能で設定されたレスポンス

例外ハンドリングの設定を変更する場合は、例外ハンドリングを参照されたい。

4.14.2 How to use

本ガイドラインで示すヘルスチェックの構成で示した実装例について説明する。

Repository インタフェース

まず、HealthCheckRepository を作成する。HealthCheckRepository はヘルスチェック用の SQL を実行し、データベースの稼働を確認する

なお、ここでは MyBatis3 を用いてデータベースにアクセスする例を示す。他の方式を採用する場合はデータアクセスを参照されたい。

HealthCheckRepository.java

```
package com.example.domain.repository.healthcheck;

public interface HealthCheckRepository {
   void healthcheck();
}
```

ここでは、データベースへのアクセスが正しく行えていることさえ確認できればよいので、必要最小限の SQL を設定する。

本ガイドラインでは、SQL は以下の条件を満たすように設定している。

- 参照系であること
- パラメータが不要であること

以下は、PostgreSQL を使用した場合のマッピングファイルの例である。

HealthCheckRepository.xml(PostgreSQL を使用した場合)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.example.domain.repository.healthcheck.HealthCheckRepository
↪">

   <select id="healthcheck" resultType="String">
      SELECT '1'
   </select>

</mapper>
```

また、以下は、Oracle を使用した場合のマッピングファイルの例である。

HealthCheckRepository.xml(Oracle を使用した場合)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.example.domain.repository.healthcheck.HealthCheckRepository
↪">

   <select id="healthcheck" resultType="String">
      SELECT '1' FROM DUAL
   </select>

</mapper>
```

Service クラス

次に、HealthCheckService インタフェースと、HealthCheckService インタフェースを実装した HealthCheckServiceImpl クラスを作成する。

HealthCheckServiceImpl は、healthcheckRepository の healthcheck メソッドを呼び出し、データベースのヘルスチェックを行う。

HealthCheckService.java

```
package com.example.domain.service.healthcheck;

public interface HealthCheckService {
   void healthcheck();
}
```

HealthCheckServiceImpl.java

```
package com.example.domain.service.healthcheck;

import healthcheck.domain.repository.healthcheck.HealthCheckRepository;

import javax.inject.Inject;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
@Transactional(readOnly = true)
public class HealthCheckServiceImpl implements HealthCheckService {

   @Inject
   HealthCheckRepository healthcheckRepository;
```

```
    @Override
    public void healthcheck() {
        healthcheckRepository.healthcheck();
    }
}
```

Controller クラス

次に、HealthCheckController を作成する。

HealthcheckService の healthcheck メソッドを呼び出し、実行結果によって指定されたパスに遷移する。データベースの稼働が確認できた場合は、OK. を表示するためのビューを返す。

HealthCheckController.java

```
package com.example.app.healthcheck;

import healthcheck.domain.service.healthcheck.HealthCheckService;

import javax.inject.Inject;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HealthCheckController {

    @Inject
    HealthCheckService healthcheckService;

    @RequestMapping(value = "healthcheck") // (1)
    public String healthcheck(){
        healthcheckService.healthcheck();
        return "common/healthcheck/ok"; // (2)
    }
}
```

項番	説明
(1)	value 属性は、稼働状態を調べるためのヘルスチェック用の URL となる。
(2)	Apache Tiles の設定を受けないようにするため、JSP ファイルを配置するディレクトリを 1 階層深くしている。詳細は、*Apache Tiles の設定を受けないようにする*を参照されたい。

JSP ファイル

最後に、ヘルスチェック成功時に遷移する JSP ファイルを作成する。

JSP ファイル作成時、以下に示すように<%@page>ディレクティブと OK. の間に改行文字などを挟まないようにする。

これは、レスポンスのデータ量を最低限にするためである。

ok.jsp

```
<%@page contentType="text/plain; charset=utf-8" language="java" pageEncoding="utf-8
↪" %>OK.
```

レスポンスのデータ量を最低限にするにあたり、他にも注意するべき点が存在する。詳細はレスポンスのデータ量を最低限にする設定を参照されたい。

アクセス権の設定

ヘルスチェック処理を使用する際は、認証・認可機能などによりヘルスチェック用の URL がアクセス不可にならないように注意する必要がある。

例えば、どのロールでもアクセスできるようにするには、spring-security.xml の<sec:intercept-url>を設定する。

/healthcheck 配下の除外設定を行う例を以下に示す。

詳細は認可を参照されたい。

spring-security.xml

```
<sec:http>
   <sec:intercept-url pattern="/healthcheck/**" access="permitAll"/>
   <!-- omitted -->
</sec:http>
```

注釈: 認可制御を外すと、誰でもヘルスチェック用 URL にアクセスできるようになってしまうので、外部からアクセスされたくない場合は LB などで防ぐ対処が必要である。

4.14.3 Appendix

レスポンスのデータ量を最低限にする設定

*JSP ファイル*で示した通り、レスポンスのデータ量を最低限にするにあたり、主に以下の点に注意が必要である。

- *Apache Tiles* の設定を受けないようにする
- ヘッダファイルやフッタファイルを読み込まないようにする
- レスポンスから余計な改行を削除する

上記について、それぞれ実装例を以下に示す。

Apache Tiles の設定を受けないようにする

*Controller クラス*で示した通り、Apache Tiles の設定を受けないよう、tiles-definitions.xml の`<put-attribute>`タグに従わないディレクトリ配下に JSP ファイルを配置する必要がある。

ブランクプロジェクトのデフォルトの設定では、`/WEB-INF/views/{1}/{2}.jsp` に該当する JSP に Apache Tiles が適用される設定となっているため、1 階層深いディレクトリを作成し、`/WEB-INF/views/common/healthcheck/`配下に JSP ファイルを配置している。

詳細は *Tiles による画面レイアウト*を参照されたい。

ヘッダファイルやフッタファイルを読み込まないようにする

JSP ファイルは web.xml の`<jsp-config>`タグの影響を受けることに注意が必要である。

`<jsp-config>`タグ内で`<include-prelude>`タグや`<include-coda>`タグを設定した場合、JSP ファイルにヘッダファイルやフッタファイルが読み込まれてしまう。ヘッダファイルやフッタファイルを読み込む設定となっていないか注意すること。

設定の例を以下に示す。

web.xml

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>/WEB-INF/views/common/healthcheck/ok.jsp</url-pattern>
        <el-ignored>false</el-ignored>
        <page-encoding>UTF-8</page-encoding>
        <scripting-invalid>false</scripting-invalid>
        // (1)
```

```
    </jsp-property-group>
</jsp-config>
```

項番	説明
(1)	余計なヘッダファイルやフッタファイルを読み込まないために、<include-prelude>タグや<include-coda>タグは設定しない。

レスポンスから余計な改行を削除する

例えば、タグライブラリを使用するために、JSP の先頭に<%@taglib>ディレクティブを設定した場合、レスポンスの先頭に余計な改行が出力されてしまう。

そのため、<%@page>ディレクティブに trimDirectiveWhitespaces 属性を設定し、ok.jsp に余計な改行を出力しないようにする。

ok.jsp(trimDirectiveWhitespaces 属性を設定する場合)

```
<%@page contentType="text/plain; charset=utf-8" language="java" pageEncoding="utf-8
↪" trimDirectiveWhitespaces="true" %>OK.
```

注釈: ブランクプロジェクトのデフォルトの設定では、web.xml の<include-prelude>に/WEB-INF/views/common/include.jsp が設定されている。そのため、上記に示した設定をするか、もしくは、ok.jsp を含まないように<url-pattern>を修正する必要がある。そうしないと、include.jsp がすべての JSP ファイルに読み込まれ ok.jsp には改行が出力されてしまう。

警告: WebLogic を使用した場合、上記に示した trimDirectiveWhitespaces 属性を設定しても、<%@page>ディレクティブよりも前に余計な文字があった場合に改行が削除されない。そのため、別の方法で対応する必要がある。例として、web.xml の<jsp-property-group>タグに、<trim-directive-whitespaces>タグを設定する対応法を以下に示す。

web.xml(<trim-directive-whitespaces>タグを設定する場合)

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>/WEB-INF/views/common/healthcheck/ok.jsp</url-pattern>
↪ // (1)
        <el-ignored>false</el-ignored>
        <page-encoding>UTF-8</page-encoding>
        <scripting-invalid>false</scripting-invalid>
        <trim-directive-whitespaces>true</trim-directive-whitespaces> //␣
↪(2)
    </jsp-property-group>
</jsp-config>
```

項番	説明
(1)	ok.jsp のみに<trim-directive-whitespaces>タグを適用するため、 <url-pattern>タグで ok.jsp のみを指定する。
(2)	<trim-directive-whitespaces>タグに true を設定することで、対象の JSP ファイル (ok.jsp) から余計な改行を削除する。

第 5 章

Web Service

本ガイドラインで想定している Web サービスアーキテクチャについて説明する。

5.1 RESTful Web Service

5.1.1 Overview

本節では、RESTful Web Service の基本的な概念と Spring MVC を使った開発について説明する。

RESTful Web Service のアーキテクチャ、設計、実装に対する具体的な説明については、

- 「*Architecture*」

  RESTful Web Service の基本的なアーキテクチャについて説明している。

- 「*How to design*」

  RESTful Web Service の設計を行う際に考慮すべき点などを説明している。

- 「*How to use*」

  RESTful Web Service のアプリケーション構成や API の実装方法について説明している。

を参照されたい。

RESTful Web Service とは

まず REST とは、「**RE**presentational **S**tate **T**ransfer」の略であり、
クライアントとサーバ間でデータをやりとりするアプリケーションを構築するための**アーキテクチャスタイル**の一つである。
REST のアーキテクチャスタイルには、いくつかの重要な原則があり、これらの原則に従っているもの (システムなど) は **RESTful** と表現される。
つまり、「RESTful Web Service」とは、REST の原則に従って構築されている Web Service という事になる。

RESTful Web Service において最も重要なのは、「リソース」という概念である。

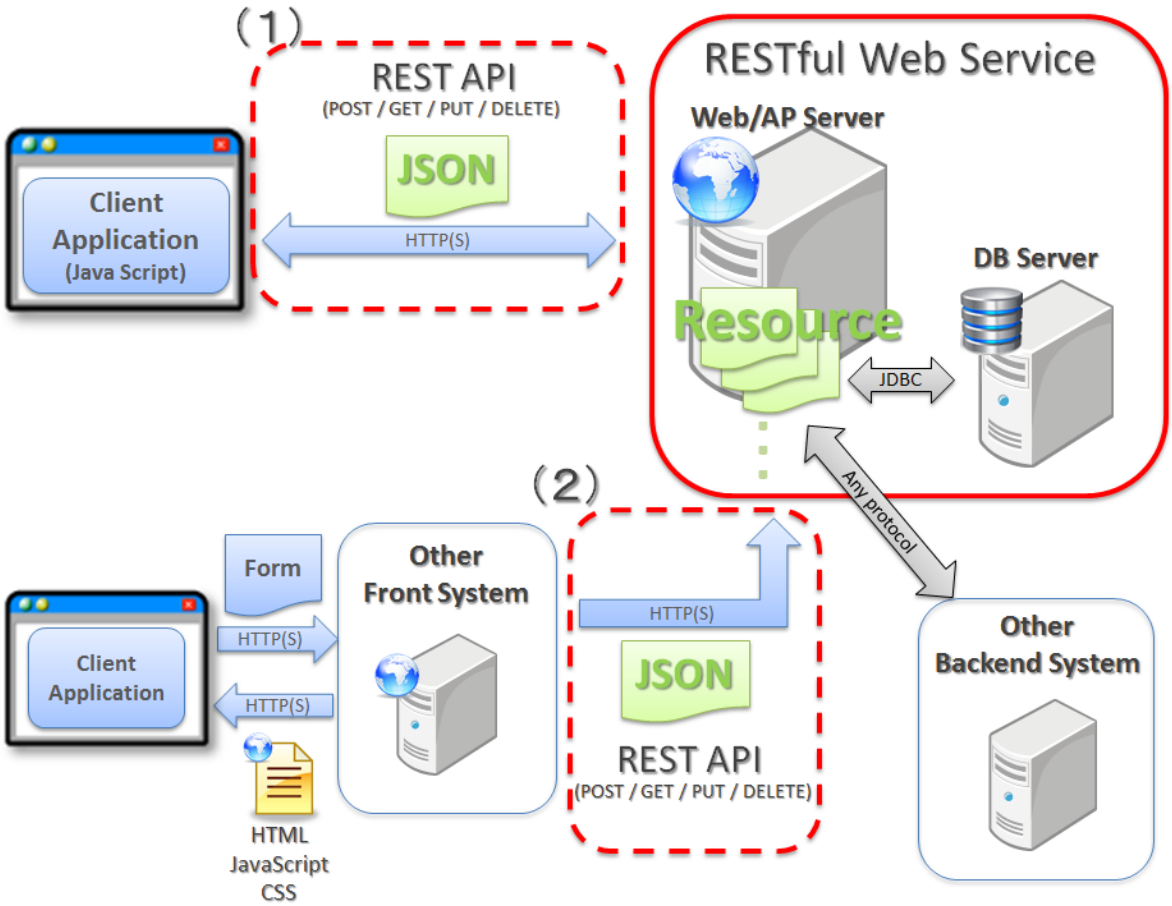
RESTful Web Service では、データベースなどで管理している情報の中からクライアントに提供すべき情報を「リソース」として抽出し、抽出した「リソース」に対する CRUD 操作を HTTP メソッド (POST/GET/PUT/DELETE) を使ってクライアントに提供することになる。

「リソース」に対する CRUD 操作は「REST API」や「RESTful API」と呼ばれる事があり、本ガイドラインでは「REST API」と呼ぶ。

また、クライアントとサーバ間でリソースをやりとりする際の電文形式には、電文の視認性、及びデータ構造の表現性が高い JSON や XML を使用するのが一般的である。

RESTful Web Service を利用するアプリケーションのシステム構成は、主に以下の 2 パターンとなる。

クライアントとサーバ間で「リソース」をやりとりするための具体的なアーキテクチャについては、「*Architecture*」で説明する。



項番	説明
(1)	ユーザインタフェースを持つクライアントアプリケーションと RESTful Web Service の間で、直接リソースのやりとりを行う。 このパターンは、要件や仕様の変更頻度が多いユーザインタフェースに依存するロジックと、より普遍的で変更頻度が少ないデータモデルに対するロジックを分離する際に採用される構成である。
(2)	ユーザインタフェースを持つクライアントアプリケーションと直接リソースをやり取りするのではなく、システム間でリソースのやりとりを行う。 このパターンは、各システムで管理しているビジネスデータを一元管理するようなシステムを構築する際に採用される構成である。

RESTful Web Service の開発について

TERASOLUNA Server Framework for Java (5.x) では、Spring MVC の機能を利用して RESTful Web Service の開発を行う。

Spring MVC では、RESTful Web Service を開発する上で必要となる共通的な機能がデフォルトで組み込まれている。
そのため、特別な設定の追加や実装を行うことなく、RESTful Web Service の開発を開始する事ができる。

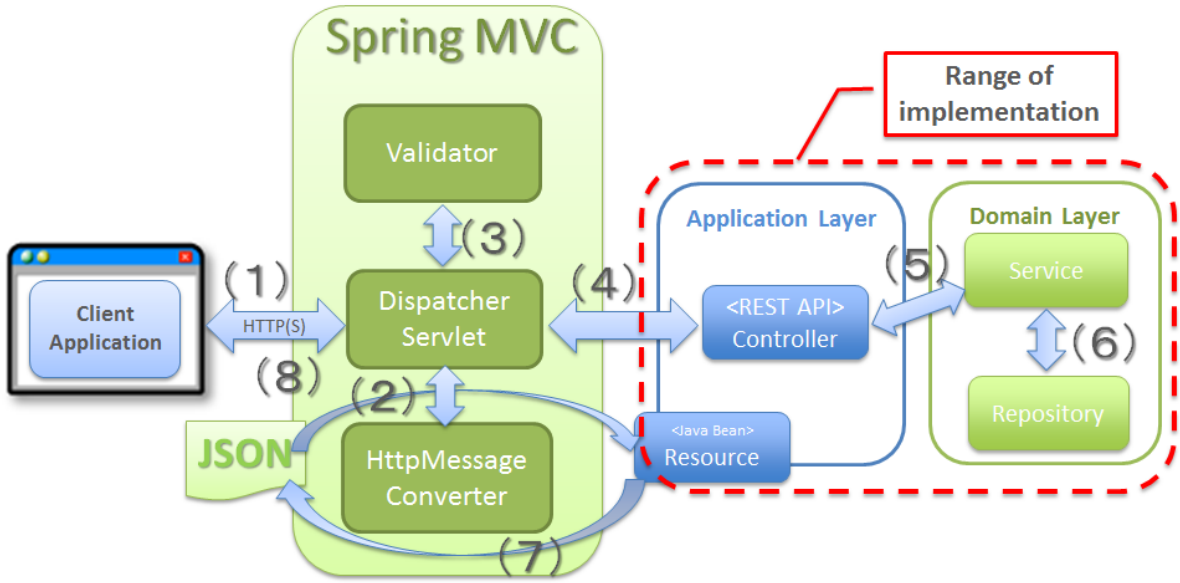
Spring MVC にデフォルトで組み込まれている主な共通機能は以下の通りである。
これらの機能は、REST API を提供する Controller のメソッドにアノテーションを指定だけで有効にする事ができる。

項番	機能概要
(1)	リクエスト BODY に設定されている JSON や XML 形式の電文を Resource オブジェクト (JavaBean) に変換し、Controller クラスのメソッド (REST API) に引き渡す機能
(2)	電文から変換された Resource オブジェクト (JavaBean) に格納された値に対して入力チェックを実行する機能
(3)	Controller クラスのメソッド (REST API) から返却した Resource オブジェクト (JavaBean) を、JSON や XML 形式に変換しレスポンス BODY に設定する機能

注釈: 例外ハンドリングについて

例外ハンドリングについては、Spring MVC から汎用的な機能の提供がないため、プロジェクト毎に実装が必要となる。例外ハンドリングの詳細については、「例外のハンドリングの実装」を参照されたい。

Spring MVC の機能を利用して RESTful Web Service を開発した場合、アプリケーションは以下のような構成となり、そのうち実装が必要なのは、赤枠の部分となる。



項番	処理レイヤ	説明
(1)	Spring MVC (Framework)	Spring MVC は、クライアントからのリクエストを受け取り、呼び出す REST API(Controller のハンドラメソッド) を決定する。
(2)		Spring MVC は、`HttpMessageConverter` を使用して、リクエスト BODY に指定されている JSON 形式の電文を Resource オブジェクトに変換する。
(3)		Spring MVC は、`Validator` を使用して、Resource オブジェクトに格納されて値に対して入力チェックを行う。
(4)		Spring MVC は、REST API を呼び出す。 その際に、JSON から変換した入力チェック済みの Resource オブジェクトが REST API に引き渡される。
(5)	REST API	REST API は、Service のメソッドを呼び出し、Entity などの DomainObject に対する処理を行う。

Continued on next page

表 5.1 – continued from previous page

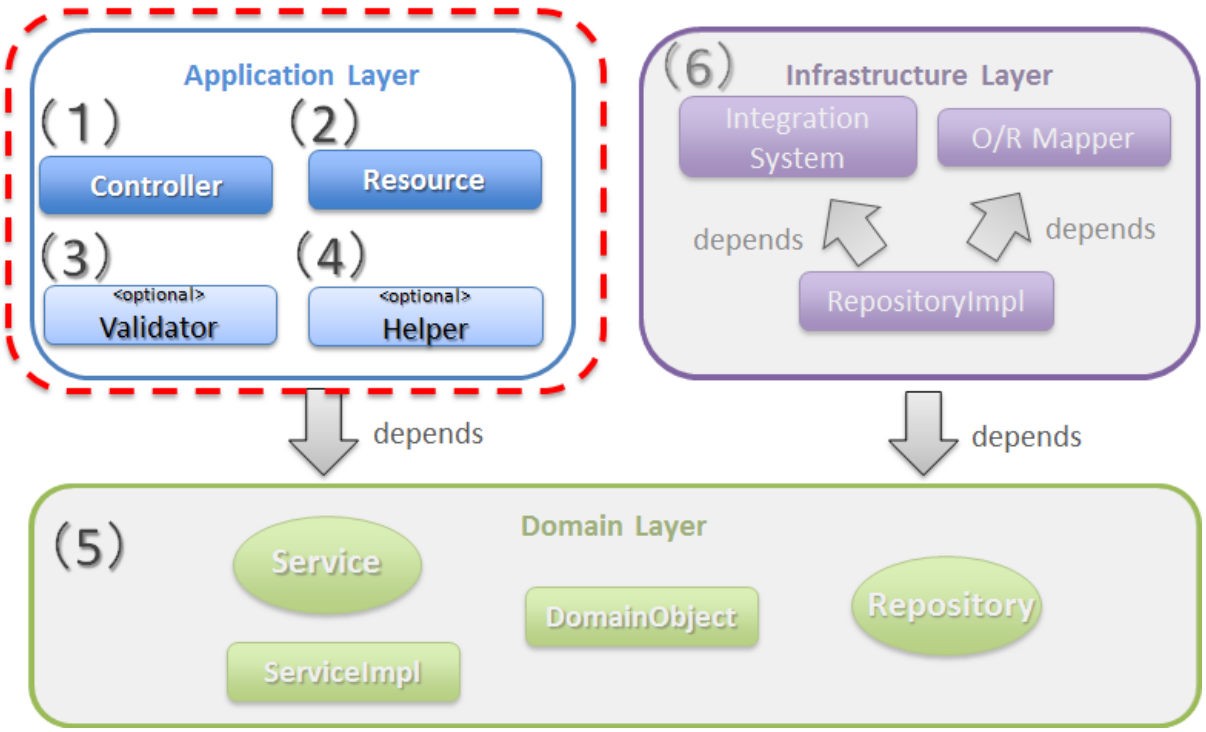
項番	処理レイヤ	説明
(6)		Service のメソッドは、Repository のメソッドを呼び出し、Entity などの DomainObject の CRUD 処理を行う。
(7)	Spring MVC (Framework)	Spring MVC は、`HttpMessageConverter` を使用して、REST API から返却された Resource オブジェクトを JSON 形式の電文に変換する。
(8)		Spring MVC は、JSON 形式の電文をレスポンス BODY に設定し、クライアントへレスポンスする。

RESTful Web Service のモジュールの構成

Spring MVC から提供されている機能を使うことにより、RESTful Web Service 固有の処理の多くを Spring MVC に任せることが出来る。

そのため、開発するモジュールの構成は、HTML を応答する従来型の Web アプリケーションの開発とほとんど同じ構成となる。

以下に、モジュールの構成要素について説明する。



- アプリケーション層のモジュール

項番	モジュール名	説明
(1)	Controller クラス	REST API を提供するクラス。 Controller クラスはリソース単位に作成し、リソース毎の REST API のエンドポイント (URI) の指定を行う。 リソースに対する CRUD 処理は、ドメイン層の Service に委譲する事で実現する。
(2)	Resource クラス	REST API の入出力となる JSON(または XML) を表現する Java Bean。 このクラスには、Bean Validation のアノテーションの指定や、JSON や XML のフォーマットを制御するためのアノテーションの指定を行う。
(3)	Validator クラス (Optional)	入力値の相関チェックを実装するクラス。 入力値の相関チェックが不要な場合は、本クラスを作成する必要はないため、オプションの扱いとしている。 入力値の相関チェックについては、「入力チェック」を参照されたい。
(4)	Helper クラス (Optional)	Controller で行う処理を補助するための処理を実装するクラス。 本クラスは、Controller の処理をシンプルに保つことを目的として作成するクラスである。 具体的には、Resource オブジェクトと DomainObject のモデル変換処理などを行うメソッドを実装する。 モデル変換が単純な値のコピーのみで済む場合は、Helper クラスは作成せずに「*Bean マッピング (Dozer)*」を使用すればよいため、オプションの扱いにしている。

- ドメイン層のモジュール

項番	説明
(5)	ドメイン層で実装するモジュールは、アプリケーションの種類に依存しないため、本節での説明は割愛する。 各モジュールの役割については「アプリケーションのレイヤ化」を、ドメイン層の開発については「ドメイン層の実装」を参照されたい。

- インフラストラクチャ層のモジュール

項番	説明
(6)	インフラストラクチャ層で実装するモジュールは、アプリケーションの種類に依存しないため、本節での説明は割愛する。 各モジュールの役割については「アプリケーションのレイヤ化」を、インフラストラクチャ層の開発については「インフラストラクチャ層の実装」を参照されたい。

REST API の実装サンプル

詳細な説明を行う前に、どのようなクラスをアプリケーション層に作成する事になるのかを知ってもらうために、Resource クラスと Controller クラスの実装サンプルを以下に示す。

下記に示す実装サンプルは、「チュートリアル *(Todo* アプリケーション *REST* 編*)*」で題材としている Todo リソースの REST API である。

注釈: 詳細な説明を読む前に、まずは「チュートリアル *(Todo* アプリケーション *REST* 編*)*」を実践する事を強く推奨する。

チュートリアルでは”習うより慣れろ”を目的としており、詳細な説明の前に実際に手を動かすことで TERASOLUNA Server Framework for Java (5.x) による RESTful Web Service の開発を体感する事が出来る。RESTful Web Service の開発を体感した後に、詳細な説明を読むことで、RESTful Web Service の開発に対する理解度がより深まる事が期待できる。

特に RESTful Web Service の開発経験がない場合は、「チュートリアルの実践」 → 「アーキテクチャ、設計、開発に関する詳細な説明 (次節以降で説明)」 → 「チュートリアルの振り返り (再実践)」というプロセスを踏むことを推奨する。

- 実装サンプルで扱うリソース

実装サンプルで扱うリソース (Todo リソース) は、以下の JSON 形式とする。

```
{
    "todoId" : "9aef3ee3-30d4-4a7c-be4a-bc184ca1d558",
    "todoTitle" : "Hello World!",
    "finished" : false,
    "createdAt" : "2014-02-25T02:21:48.493+0000"
}
```

- Resource クラスの実装サンプル

上記で示した Todo リソースを表現する JavaBean として、Resource クラスを作成する。

```
package todo.api.todo;

import java.io.Serializable;
import java.util.Date;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class TodoResource implements Serializable {

    private static final long serialVersionUID = 1L;

    private String todoId;

    @NotNull
    @Size(min = 1, max = 30)
    private String todoTitle;

    private boolean finished;

    private Date createdAt;

    public String getTodoId() {
```

```
        return todoId;
    }

    public void setTodoId(String todoId) {
        this.todoId = todoId;
    }

    public String getTodoTitle() {
        return todoTitle;
    }

    public void setTodoTitle(String todoTitle) {
        this.todoTitle = todoTitle;
    }

    public boolean isFinished() {
        return finished;
    }

    public void setFinished(boolean finished) {
        this.finished = finished;
    }

    public Date getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Date createdAt) {
        this.createdAt = createdAt;
    }
}
```

- Controller クラス (REST API) の実装サンプル

  Todo リソースに対して、以下の 5 つの REST API(Controller のハンドラメソッド) を作成する。

項番	API 名	HTTP メソッド	パス	ステータスコード	説明
(1)	GET Todos	GET	`/api/v1/todos`	200 (OK)	Todo リソースを全件取得する。
(2)	POST Todos	POST	`/api/v1/todos`	201 (Created)	Todo リソースを新規作成する。
(3)	GET Todo	GET	`/api/v1/todos/{todoId}`	200 (OK)	Todo リソースを一件取得する。
(4)	PUT Todo	PUT	`/api/v1/todos/{todoId}`	200 (OK)	Todo リソースを完了状態に更新する。
(5)	DELETE Todo	DELETE	`/api/v1/todos/{todoId}`	204 (No Content)	Todo リソースを削除する。

```
package todo.api.todo;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import javax.inject.Inject;

import org.dozer.Mapper;
import org.springframework.http.HttpStatus;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
```

```
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@RestController
@RequestMapping("todos")
public class TodoRestController {
    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    // (1)
    @RequestMapping(method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public List<TodoResource> getTodos() {
        Collection<Todo> todos = todoService.findAll();
        List<TodoResource> todoResources = new ArrayList<>();
        for (Todo todo : todos) {
            todoResources.add(beanMapper.map(todo, TodoResource.class));
        }
        return todoResources;
    }

    // (2)
    @RequestMapping(method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.CREATED)
    public TodoResource postTodos(@RequestBody @Validated TodoResource␣
↪todoResource) {
        Todo createdTodo = todoService.create(beanMapper.map(todoResource,␣
↪Todo.class));
        TodoResource createdTodoResponse = beanMapper.map(createdTodo,␣
↪TodoResource.class);
        return createdTodoResponse;
    }

    // (3)
    @RequestMapping(value="{todoId}", method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public TodoResource getTodo(@PathVariable("todoId") String todoId) {
        Todo todo = todoService.findOne(todoId);
        TodoResource todoResource = beanMapper.map(todo, TodoResource.class);
        return todoResource;
    }

    // (4)
    @RequestMapping(value="{todoId}", method = RequestMethod.PUT)
    @ResponseStatus(HttpStatus.OK)
    public TodoResource putTodo(@PathVariable("todoId") String todoId) {
        Todo finishedTodo = todoService.finish(todoId);
        TodoResource finishedTodoResource = beanMapper.map(finishedTodo,␣
↪TodoResource.class);
        return finishedTodoResource;
```

```
    }

    // (5)
    @RequestMapping(value="{todoId}", method = RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteTodo(@PathVariable("todoId") String todoId) {
        todoService.delete(todoId);
    }

}
```

5.1.2 Architecture

本節では、RESTful Web Service を構築するためのアーキテクチャについて説明する。

RESTful Web Service を構築するためのアーキテクチャとして、リソース指向アーキテクチャ (ROA) というものが存在する。

ROA は、「**R**esource **O**riented **A**rchitecture」の略であり、**REST のアーキテクチャスタイル (原則) に従った Web Service を構築するための具体的なアーキテクチャ**を定義している。

RESTful Web Service を作る際は、まず ROA のアーキテクチャの理解を深めてほしい。

本節では、ROA のアーキテクチャとして、以下の 7 つについて説明する。

これらは、RESTful Web Service を構築する上で重要なアーキテクチャ要素であるが、必ず全てを適用しなくてはいけないという事ではない。

開発するアプリケーションの特性を考慮し、必要なものを適用してほしい。

以下の 5 つのアーキテクチャは、アプリケーションの特性に関係なく適用すべきアーキテクチャである。

項番	アーキテクチャ	アーキテクチャの概要
(1)	*Web* 上のリソースとして公開	システム上で管理する情報をクライアントに提供する手段として、Web 上のリソースとして公開する。
(2)	*URI* によるリソースの識別	クライアントに公開するリソースには、Web 上のリソースとして一意に識別できる URI(Universal Resource Identifier) を割り当てる。
(3)	*HTTP* メソッドによるリソースの操作	リソースに対する操作は、HTTP メソッド (GET,POST,PUT,DELETE) を使い分けることで実現する。
(4)	適切なフォーマットの使用	リソースのフォーマットは、JSON 又は XML などのデータ構造を示すためのフォーマットを使用する。
(5)	適切な *HTTP* ステータスコードの使用	クライアントへ返却するレスポンスには、適切な HTTP ステータスコードを設定する。

以下の 2 つのアーキテクチャは、アプリケーションの特性に応じて、適用するアーキテクチャである。

項番	アーキテクチャ	アーキテクチャの概要
(6)	ステートレスなクライアント/サーバ間の通信	サーバ上でアプリケーションの状態は保持せずに、クライアントからリクエストされてきた情報のみで処理を行うようにする。
(7)	関連のあるリソースへのリンク	リソースの中には、指定されたリソースと関連をもつ他のリソースへのリンク (URI) を含める。

Web 上のリソースとして公開

システム上で管理する情報をクライアントに提供する手段として、Web 上のリソースとして公開する。

これは、HTTP プロトコルを使ってリソースにアクセスできるようにする事を意味しており、その際にリソースを識別する方法とし、URI が使用される。

例えば、ショッピングサイトを提供する Web システムであれば、以下のような情報が Web 上のリソースとして公開する事になる。

- 商品の情報
- 在庫の情報
- 注文の情報
- 会員の情報
- 会員毎の認証の情報 (ログイン ID とパスワードなど)
- 会員毎の注文履歴の情報
- 会員毎の認証履歴の情報
- and more …

URI によるリソースの識別

クライアントに公開するリソースには、Web 上のリソースとして一意に識別できる URI(Universal Resource Identifier) を割り当てる。

実際に使用されるのは、URI のサブセットである URL(Uniform Resource Locator) となる。

ROA では、URI を使用して Web 上のリソースにアクセスできる状態になっていることを「アドレス可能性」と呼んでいる。

これは同じ URI を使用すれば、どこからでも同じリソースにアクセスできる状態になっている事を意味している。

RESTful Web Service に割り当てる URI は、**「リソースの種類を表す名詞」と「リソースを一意に識別するための値 (ID など)」**の組み合わせとする。

例えば、ショッピングサイトを提供する Web システムで扱う商品情報の URI は、以下のようになる。

- *http://example.com/api/v1/items*

  「**items**」の部分が「リソースの種類を表す名詞」となり、リソースの数が複数になる場合は、複数系の名詞を使用する。

  上記例では、商品情報である事を表す名詞の複数系を指定しており、商品情報を一括で操作する際に使用する URI となる。これは、ファイルシステムに置き換えると、ディレクトリに相当する。

- *http://example.com/api/v1/items/I312-535-01216*

  「**I312-535-01216**」の部分が「リソースを識別するための値」となり、リソース毎に異なる値となる。

  上記例では、商品情報を一意に識別するための値として商品 ID を指定しており、特定の商品情報を操作する際に使用する URI となる。これは、ファイルシステムに置き換えると、ディレクトリの中に格納されているファイルに相当する。

警告: RESTful Web Service に割り当てる URI には、下記で示すような**操作を表す動詞を含んではいけない。**

- *http://example.com/api/v1/items?get&itemId=I312-535-01216*
- *http://example.com/api/v1/items?delete&itemId=I312-535-01216*

上記例では、 URI の中に **get** や **delete** という動詞を含んでいるため、RESTful Web Service に割り当てる URI として適切ではない。

RESTful Web Service では、リソースに対する操作は **HTTP** メソッド **(GET,POST,PUT,DELETE)** を使用して表現する。

HTTP メソッドによるリソースの操作

リソースに対する操作は、**HTTP** メソッド **(GET,POST,PUT,DELETE)** を使い分けることで実現する。

ROA では、HTTP メソッドの事を「統一インタフェース」と呼んでいる。

これは、HTTP メソッドが Web 上で公開される全てのリソースに対して実行する事ができ、且つリソース毎に HTTP メソッドの意味が変わらない事を意味している。

以下に、HTTP メソッドに割り当てられる、リソースに対する操作の対応付けと、それぞれの操作が保証すべき事後条件について説明する。

項番	HTTP メソッド	リソースに対する操作	操作が保証すべき事後条件
(1)	GET	リソースを取得する。	安全性、べき等性。
(2)	POST	リソースを作成する。	作成したリソースの URI の割り当てはサーバが行い、割り当てた URI はレスポンスの Location ヘッダに設定してクライアントに返却する。
(4)	PUT	リソースを作成又は更新する。	べき等性。
(5)	PATCH	リソースを差分更新する。	べき等性。
(6)	DELETE	リソースを削除する。	べき等性。
(7)	HEAD	リソースのメタ情報を取得する。 GET と同じ処理を行いヘッダのみ応答する。	安全性、べき等性。
(8)	OPTIONS	リソースに対して使用できる HTTP メソッドの一覧を応答する。	安全性、べき等性。

注釈: 安全性とべき等性の保証について

HTTP メソッドを使ってリソースの操作を行う場合、事後条件として、「安全性」と「べき等性」の保証を行う事が求められる。

【安全性とは】

ある数字に 1 を何回掛けても、数字がかわらない事 (10 に 1 を何回掛けても結果は 10 のままである事) を保証する。これは、同じ操作を何回行ってもリソースの状態が変わらない事を保証する事である。

【べき等性とは】

数字に 0 を何回掛けても 0 になる事 (10 に 0 を 1 回掛けても何回掛けても結果は共に 0 になる事) を保証する。これは、一度操作を行えば、その後で同じ操作を何回行ってもリソースの状態が変わらない事を保証する事である。ただし、別のクライアントが同じリソースの状態を変更している場合は、べき等性を保障する必要はなく、事前条件に対するエラーとして扱ってもよい。

ちなみに：クライアントがリソースに割り当てる URI を指定してリソースを作成する場合

リソースを作成する際に、クライアントによってリソースに割り当てる URI を指定する場合は、**作成するリソースに割り当てる URI に対して、PUT メソッドを呼び出すことで実現する。**

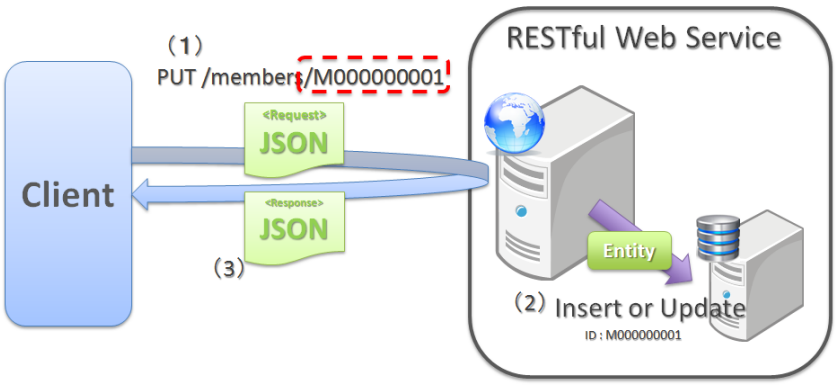
PUT メソッドを使用してリソースを作成する場合、

- 指定された URI にリソースが存在しない場合はリソースを作成
- 既にリソースが存在する場合はリソースの状態を更新

するのが一般的な動作である。

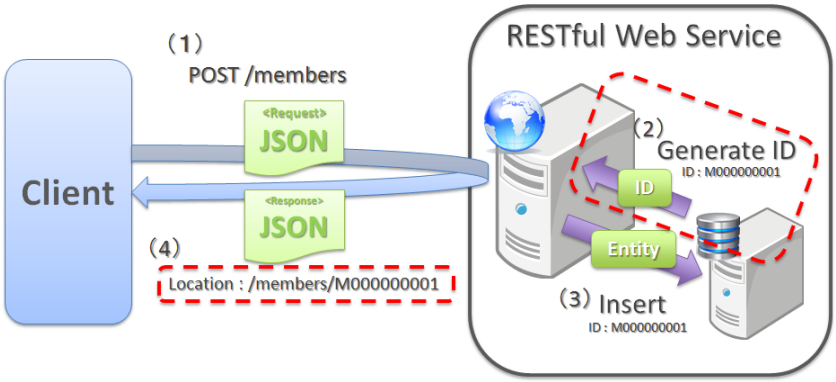
以下に、PUT と POST メソッドを使ってリソースを作成する際の処理イメージの違いについて説明する。

【PUT メソッドを使用してリソースを作成する際の処理イメージ】



項番	説明
(1)	URI に作成するリソースの URI(ID) を指定して、PUT メソッドを呼び出す。
(2)	URI で指定された ID の Entity を作成する。 既に同じ ID で作成済みの場合は、内容を更新する。
(3)	作成又は更新したリソースを応答する。

【POST メソッドを使用してリソースを作成する際の処理イメージ】



項番	説明
(1)	POST メソッドを呼び出す。
(2)	リクエストされリソースを識別するための ID を生成する。
(3)	(2) で生成した ID の Entity を作成する。
(4)	作成したリソースを応答する。 レスポンスの Location ヘッダに生成したリソースにアクセスするための URI を設定する。

適切なフォーマットの使用

リソースのフォーマットは、**JSON** 又は **XML** などのデータ構造を示すためのフォーマットを使用する。

ただし、リソースの種類によっては、JSON や XML 以外のフォーマットを使ってもよい。

例えば、統計情報に分類される様なリソースでは、折れ線グラフを画像フォーマット (バイナリデータ) としてリソースを公開する事も考えられる。

リソースのフォーマットとして、複数のフォーマットをサポートする場合は、以下のどちらかの方法で切り替えを行う。

- 拡張子によって切り替えを行う。

  レスポンスのフォーマットは、拡張子を指定する事で切り替える事ができる。

  本ガイドラインでは、拡張子による切り替えを推奨する。

  推奨する理由は、レスポンスするフォーマット指定が簡単であるという点と、レスポンスするフォーマットが URI に含まれ、直感的な URI になるという点である。

  注釈: 拡張子で切り替える場合の **URI** 例

  - *http://example.com/api/v1/items.json*
  - *http://example.com/api/v1/items.xml*
  - *http://example.com/api/v1/items/I312-535-01216.json*
  - *http://example.com/api/v1/items/I312-535-01216.xml*

- リクエストの **Accept** ヘッダの **MIME** タイプによって切り替えを行う。

  RESTful Web Service で使用される代表的な MIME タイプを以下に示す。

項番	フォーマット	MIME タイプ
(1)	JSON	application/json
(2)	XML	application/xml

適切な **HTTP** ステータスコードの使用

クライアントへ返却するレスポンスには、適切な **HTTP** ステータスコードを設定する。

HTTP ステータスコードには、クライアントから受け取ったリクエストをサーバがどのように処理したかを示す値を設定する。

これは **HTTP** の仕様であり、**HTTP** の仕様に可能な限り準拠することを推奨する。

ちなみに：**HTTP** の仕様について

RFC 7230(Hypertext Transfer Protocol – HTTP/1.1) の 3.1.2 Status Line を参照されたい。

ブラウザに HTML を返却するような伝統的な Web システムでは、処理結果に関係なく `200 OK` を応答し、処理結果はエンティティボディ (HTML) の中で表現するという事が一般的であった。

HTML を返却するような伝統的な Web アプリケーションでは、処理結果を判断するのはオペレータ (人間) のため、この仕組みでも問題が発生する事はなかった。

しかし、この仕組みで RESTful Web Service を構築した場合、以下のような問題が潜在的に存在することになるため、適切な HTTP ステータスコードを設定することを推奨する。

項番	潜在的な問題点
(1)	処理結果 (成功と失敗) のみを判断すればよい場合でも、エンティティボディを解析処理が必須になるため、無駄な処理が必要になる。
(2)	エラーハンドリングを行う際に、システム独自に定義されたエラーコードを意識する事が必須になるため、クライアント側のアーキテクチャ (設計及び実装) に悪影響を与える可能性がある。
(3)	クライアント側でエラー原因を解析する際に、システム独自に定義されたエラーコードの意味を理解しておく必要があるため、直感的なエラー解析の妨げになる可能性がある。

注釈: HTTP のメッセージ構文を規定する RFC 7230 では、HTTP ステータスコードの説明句（`reason-phrase`）の出力は必須ではなく、クライアントは無視すべきであると規定されている。例えば、RFC 7230 に準拠した実装の Tomcat 8.5 では、説明句が出力されない。

RFC 7230(Hypertext Transfer Protocol – HTTP/1.1) の 3.1.2 Status Line を参照されたい。

ステートレスなクライアント/サーバ間の通信

サーバ上でアプリケーションの状態は保持せずに、クライアントからリクエストされてきた情報のみで処理を行うようにする。

ROA では、サーバ上でアプリケーションの状態を保持しない事を「ステートレス性」と呼んでいる。

これは、アプリケーションサーバのメモリ (HTTP セッションなど) にアプリケーションの状態を保持しない事を意味し、リクエストされた情報のみでリソースに対する操作が完結できる状態にしておく事を意味している。

本ガイドラインでは、**可能な限り「ステートレス性」を保つことを推奨する。**

注釈: アプリケーションの状態とは

Web ページの遷移状態、入力値、プルダウン/チェックボックス/ラジオボタンなどの選択状態、認証状態などの事である。

注釈: CSRF 対策との関連

本ガイドラインに記載されている CSRF 対策を RESTful Web Service に対して行った場合、CSRF 対策用のトークン値が HTTP セッションに保存されるため、クライアントとサーバ間の「ステートレス性」を保つ事が出来ないという点を補足しておく。

そのため、CSRF 対策を行う場合は、システムの可用性を考慮する必要がある。

高い可用性が求められるシステムでは、

- AP サーバをクラスタ化し、セッションをレプリケーションする。
- セッションの保存先を AP サーバのメモリ以外にする。

等の対策が必要となる。ただし、上記対策は性能への影響があるため、性能要件も考慮する必要がある。

CSRF 対策については、*CSRF 対策*を参照されたい。

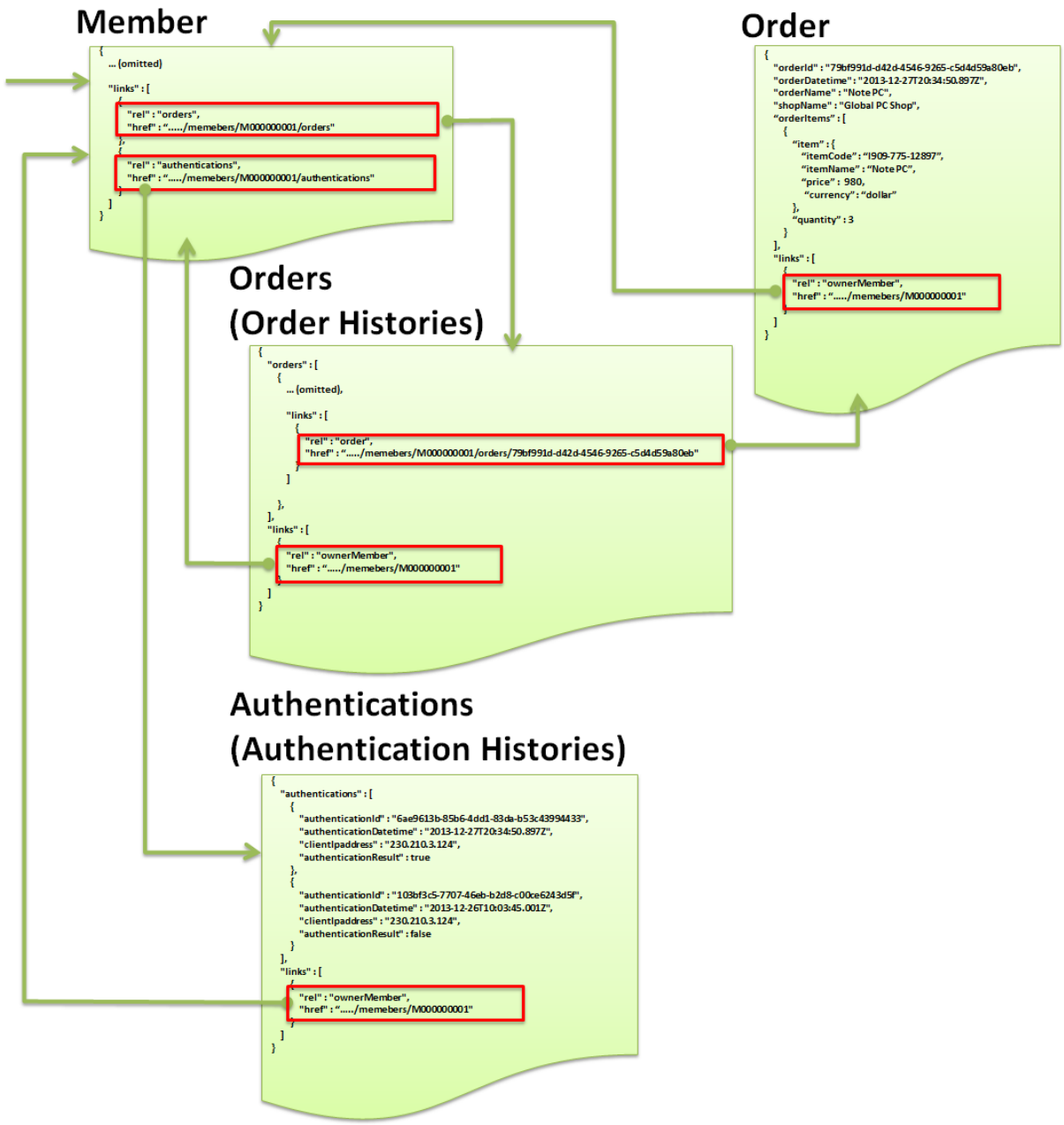
関連のあるリソースへのリンク

リソースの中には、指定されたリソースと関連をもつ他のリソースへのハイパーメディアリンク (URI) を含める。

ROA では、リソース状態の表現の中に、他のリソースへのハイパーメディアリンクを含めることを「接続性」と呼んでいる。

これは、関連をもつリソース同士が相互にリンクを保持し、そのリンクをたどる事で関連する全てのリソースにアクセスできる状態にしておく事を意味している。

下記に、ショッピングサイトの会員情報のリソースを例に、リソースの接続性について説明する。


Member
{
... (omitted)
"links" : [
{
"rel" : "orders",
"href" : "...../memebers/M000000001/orders"
},
{
"rel" : "authentications",
"href" : "...../memebers/M000000001/authentications"
}
]
}
Order
{
"orderId" : "79bf991d-d42d-4546-9265-c5d4d59a80eb",
"orderDatetime" : "2013-12-27T20:34:50.897Z",
"orderName" : "Note PC",
"shopName" : "Global PC Shop",
"orderItems" : [
{
"item" : {
"itemCode" : "I909-775-12897",
"itemName" : "Note PC",
"price" : 980,
"currency" : "dollar"
},
"quantity" : 3
}
],
"links" : [
{
"rel" : "ownerMember",
"href" : "...../memebers/M000000001"
}
]
}
Orders
(Order Histories)
{
"orders" : [
{
... (omitted),
"links" : [
{
"rel" : "order",
"href" : "...../memebers/M000000001/orders/79bf991d-d42d-4546-9265-c5d4d59a80eb"
}
]
},
],
"links" : [
{
"rel" : "ownerMember",
"href" : "...../memebers/M000000001"
}
]
}
Authentications
(Authentication Histories)
{
"authentications" : [
{
"authenticationId" : "6ae9613b-85b6-4dd1-83da-b53c43994433",
"authenticationDatetime" : "2013-12-27T20:34:50.897Z",
"clientIpaddress" : "230.210.3.124",
"authenticationResult" : true
},
{
"authenticationId" : "103bf3c5-7707-46eb-b2d8-c00ce6243d5f",
"authenticationDatetime" : "2013-12-26T10:03:45.001Z",
"clientIpaddress" : "230.210.3.124",
"authenticationResult" : false
}
],
"links" : [
{
"rel" : "ownerMember",
"href" : "...../memebers/M000000001"
}
]
}

項番	説明

(1)	会員情報のリソースを取得 (GET http://example.com/api/v1/memebers/M000000001) を行うと、以下の JSON が返却される。
	`{`
	`  "memberId" : "M000000001",`
	`  "memberName" : "John Smith",`
	`  "address" : {`
	`  "address1" : "45 West 36th Street",`
	`  "address2" : "7th Floor",`
	`  "city" : "New York",`
	`  "state" : "NY",`
	`  "zipCode" : "10018"`
	`  },`
	`  "links" : [`
	`  {`
	`  "rel" : "orders",`
	`  "href" : "http://example.com/api/v1/memebers/M000000001/orders"`
	`  },`
	`  {`
	`  "rel" : "authentications",`
	`  "href" : "http://example.com/api/v1/memebers/M000000001/authentications"`
	`  }`
	`  ]`
	`}`
	ハイライトした部分が、関連をもつ他のリソースへのハイパーメディアリンク (URI) となる。上記例では、会員毎の注文履歴と認証履歴のリソースに対して接続性を保持している。

Continued on next page

表 5.2 – continued from previous page

項番	説明

(2)	返却された JSON に設定されているハイパーメディアリンク (URI) を使用して、注文履歴のリソースを取得 (`GET http://example.com/api/v1/memebers/M000000001/orders`) を行うと、以下の JSON が返却される。
	`{`
	`  "orders" : [`
	`  {`
	`  "orderId" : "029b49d7-0efa-411b-bc5a-6570ce40ead8",`
	`  "orderDatetime" : "2013-12-27T20:34:50.897Z",`
	`  "orderName" : "Note PC",`
	`  "shopName" : "Global PC Shop",`
	`  "links" : [`
	`  {`
	`  "rel" : "order",`
	`  "href" : "http://example.com/api/v1/memebers/M000000001/orders/029b49d7-0efa-411b-bc5a-6570ce40ead8"`
	`  }`
	`  ]`
	`  },`
	`  {`
	`  "orderId" : "79bf991d-d42d-4546-9265-c5d4d59a80eb",`
	`  "orderDatetime" : "2013-12-03T19:01:44.109Z",`
	`  "orderName" : "Orange Juice 100%",`
	`  "shopName" : "Global Food Shop",`
	`  "links" : [`
	`  {`
	`  "rel" : "order",`
	`  "href" : "http://example.com/api/v1/memebers/M000000001/orders/79bf991d-d42d-4546-9265-c5d4d59a80eb"`
	`  }`
	`  ]`
	`  }`
	`  ],`
	`  "links" : [`
	`  {`
	`  "rel" : "ownerMember",`
	`  "href" : "http://example.com/api/v1/memebers/M000000001"`
	`  }`
	`  ]`
	`}`
	ハイライトした部分が、関連をもつ他のリソースへのハイパーメディアリンク (URI) となる。
	上記例では、注文履歴のオーナに関する会員情報のリソース及び注文履歴のリソースに対する接続性を保持している。

Continued on next page

表 5.2 – continued from previous page

項番	説明

(3)	注文履歴のオーナとなる会員情報のリソースを再度取得 (GET http://example.com/api/v1/memebers/M000000001) し、返却された JSON に設定されているハイパーメディアリンク (URI) を使用して、認証履歴のリソースを取得 (GET http://example.com/api/v1/memebers/M000000001/authentications/) を行うと、以下の JSON が返却される。
	`{`
	`  "authentications" : [`
	`  {`
	`  "authenticationId" : "6ae9613b-85b6-4dd1-83da-b53c43994433",`
	`  "authenticationDatetime" : "2013-12-27T20:34:50.897Z",`
	`  "clientIpaddress" : "230.210.3.124",`
	`  "authenticationResult" : true`
	`  },`
	`  {`
	`  "authenticationId" : "103bf3c5-7707-46eb-b2d8-c00ce6243d5f",`
	`  "authenticationDatetime" : "2013-12-26T10:03:45.001Z",`
	`  "clientIpaddress" : "230.210.3.124",`
	`  "authenticationResult" : false`
	`  }`
	`  ],`
	`  "links" : [`
	`  {`
	`  "rel" : "ownerMember",`
	`  "href" : "http://example.com/api/v1/memebers/M000000001"`
	`  }`
	`  ]`
	`}`
	ハイライトした部分が、関連をもつ他のリソースへのハイパーメディアリンク (URI) となる。
	上記例では、認証履歴のオーナとなる会員情報のリソースに対して接続性を保持している。

リソースの中に他のリソースへのハイパーメディアリンク (URI) を含めることは、必須ではない。

事前に全ての REST API のエンドポイント (URI) を公開している場合、リソースの中に関連リソースへのリンクを設けても、リンクが使用されない可能性が高い。

特に、システム間でリソースのやりとりを行うための REST API の場合は、事前に公開されている REST API のエンドポイントに対して直接アクセスするような実装になる事が多いため、リンクを設ける意味がない事がある。

リンクを設ける意味がない場合は、無理にリンクを設ける必要はない。

逆に、ユーザインタフェースを持つクライアントアプリケーションと RESTful Web Service の間で直接リソースのやりとりを行う場合は、リンクを設けることで、クライアントとサーバ間の疎結合性を高めることが出来る。

クライアントとサーバ間の疎結合性を高めることが出来る理由は以下の通りである。

項番	疎結合性を高めることが出来る理由
(1)	クライアントアプリケーションは、リンクの論理名のみ事前に知っていればよいため、REST API を呼び出すための具体的な URI を意識する必要がなくなる。
(2)	クライアントアプリケーションが具体的な URI を意識する必要がなくなるため、サーバ側の URI を変更する際に与える影響度を最小限に抑える事ができる。

上記にあげた点を考慮し、他のリソースへのハイパーメディアリンク (URI) を設けるか否かを判断すること。

ちなみに：HATEOAS との関係

HATEOAS は、「**H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate」の略であり、RESTful な Web アプリケーションを作成するためのアーキテクチャの一つである。

HATEOAS のアーキテクチャでは、

- サーバは、クライアントとサーバ間でやり取りするリソース (JSON や XML) の中に、アクセス可能なリソースへのハイパーメディアリンク (URI) を含める。
- クライアントは、リソース表現 (JSON や XML) の中に含まれるハイパーメディアリンクを介して、サーバから必要なリソースを取得し、アプリケーションの状態 (画面の状態など) を変化させる。

ことになるため、関連のあるリソースへのリンクを設ける事は、HATEOAS のアーキテクチャと一致する。

サーバとクライアントとの疎結合性を高めたい場合は、HATEOAS のアーキテクチャを採用する事を検討されたい。

5.1.3 How to design

本説では、RESTful Web Service の設計について説明する。

リソースの抽出

まず、Web 上に公開するリソースを抽出する。

リソースを抽出する際の注意点を以下に示す。

項番	リソース抽出時の注意点
(1)	Web 上に公開するリソースは、データベースなどで管理されている情報になるが、**安易にデータベースのデータモデルをそのままリソースとして公開してはいけない。** データベースに格納されている項目の中には、クライアントに公開すべきでない項目もあるので、精査が必要である。
(2)	**データベースの同じテーブルで管理されている情報であっても、情報の種類が異なる場合は、別のリソースとして公開する事を検討する。** 本質的には別の情報だが、データ構造が同じという理由で同じテーブルで管理されているケースがあるので、精査が必要である。
(3)	RESTful Web Service では、イベントで操作する情報をリソースとして抽出する。 **イベント自体をリソースとして抽出してはいけない。** 例えば、ワークフロー機能で発生するイベント (承認、否認、差し戻しなど) から呼び出される RESTful Web Service を作成する場合、ワークフロー自体やワークフローの状態を管理するための情報をリソースとして抽出する。

URI の割り当て

抽出したリソースに対して、リソースを識別するための URI を割り当てる。

URI は、以下の形式を推奨する。

- http(s)://{ドメイン名 (:ポート番号)}/{REST API であることを示す値}/{API バージョン}/{リソースを識別するためのパス}
- http(s)://{REST API であることを示すドメイン名 (:ポート番号)}/{API バージョン}/{リソースを識別するためのパス}

具体例は以下の通り。

- http://example.com/api/v1/members/M000000001
- http://api.example.com/v1/members/M000000001

REST API であることを示すための URI の割り当て

RESTful Web Service(REST API) 向けの URI であること明確にするために、URI 内のドメイン又はパスに api を含めることを推奨する。

具体的には、以下のような URI とする。

- http://example.com/api/...
- http://api.example.com/...

API バージョンを識別するための URI の割り当て

RESTful Web Service は、複数のバージョンで稼働が必要になる可能性があるため、クライアントに公開する URI には、API バージョンを識別するための値を含めるようにする事を推奨する。

具体的には、以下のような形式の URI とする。

- http://example.com/api/{API バージョン}/{リソースを識別するためのパス}
- http://api.example.com/{API バージョン}/{リソースを識別するためのパス}

リソースを識別するためのパスの割り当て

Web 上に公開するリソースに対して、以下の 2 つの URI を割り当てる。
下記の例では、会員情報を Web 上に公開する場合の URI 例を記載している。

項番	URI の形式	URI の具体例	説明
(1)	/{リソースのコレクションを表す名詞}	/api/v1/members	リソースを一括で操作する際に使用する URI となる。
(2)	/{リソースのコレクションを表す名詞/リソースの識別子 (ID など)}	/api/v1/members/M0001	特定のリソースを操作する際に使用する URI となる。

Web 上に公開する関連リソースへの URI は、ネストさせて表現する。
下記の例では、会員毎の注文情報を Web 上に公開する場合の URI 例を記載している。

項番	URI の形式	URI の具体例	説明
(3)	{リソースの URI}/{関連リソースのコレクションを表す名詞}	/api/v1/members/M0001/orders	関連リソースを一括で操作する際に使用する URI となる。
(4)	{リソースの URI}/{関連リソースのコレクションを表す名詞}/{関連リソースの識別子 (ID など)}	/api/v1/members/M0001/orders/O0001	特定の関連リソースを操作する際に使用する URI となる。

Web 上に公開する関連リソースの要素が 1 件の場合は、関連リソースを示す名詞は複数系ではなく単数形と

する。

下記の例では、会員毎の資格情報を Web 上に公開する場合の URI 例を記載している。

項番	URI の形式	URI の具体例	説明
(5)	{リソースの URI}/{関連リソースを表す名詞}	/api/v1/members/M0001/credential	要素が 1 件の関連リソースを操作する際に使用する URI。

HTTP メソッドの割り当て

リソース毎に割り当てた URI に対して、以下の HTTP メソッドを割り当て、リソースに対する CRUD 操作を REST API として公開する。

リソースコレクションの **URI** に対する **HTTP** メソッドの割り当て

項番	HTTP メソッド	実装する REST API の概要
(1)	GET	URI で指定されたリソースのコレクションを取得する REST API を実装する。
(2)	POST	指定されたリソースを作成しコレクションに追加する REST API を実装する。
(3)	PUT	URI で指定されたリソースの一括更新を行う REST API を実装する。
(4)	DELETE	URI で指定されたリソースの一括削除を行う REST API を実装する。
(5)	HEAD	URI で指定されたリソースコレクションのメタ情報を取得する REST API を実装する。 GET と同じ処理を行いヘッダのみ応答する。
(6)	OPTIONS	URI で指定されたリソースコレクションでサポートされている HTTP メソッド (API) のリストを応答する REST API を実装する。

特定リソースの URI に対する HTTP メソッドの割り当て

項番	HTTP メソッド	実装する REST API の概要
(1)	GET	URI で指定されたリソースを取得する REST API を実装する。
(2)	PUT	URI で指定されたリソースの作成又は更新を行う REST API を実装する。
(3)	DELETE	URI で指定されたリソースの削除を行う REST API を実装する。
(4)	HEAD	URI で指定されたリソースのメタ情報を取得する REST API を実装する。 GET と同じ処理を行いヘッダのみ応答する。
(5)	OPTIONS	URI で指定されたリソースでサポートされている HTTP メソッド (API) のリストを応答する REST API を実装する。

リソースのフォーマット

リソースを表現するフォーマットとしては、**JSON を使用する事を推奨する。**

以降の説明では、リソースを表現するフォーマットとして JSON を使用する前提で説明を記載する。

JSON のフィールド名

JSON のフィールド名は、**「lower camel case」にすることを推奨する。**

これはクライアントアプリケーションの一つとして想定される JavaScript との相性を考慮した結果である。

フィールド名を「lower camel case」にした場合の JSON のサンプルは以下の通り。

「lower camel case」は、先頭文字を小文字にし、単語の先頭文字を大文字にする。

```
{
    "memberId" : "M000000001"
}
```

NULL とブランク文字

JSON の値として、**NULL とブランク文字は区別する事を推奨する。**
アプリケーションの処理として NULL とブランク文字を同一視する事はよくあるが、JSON に設定する値としては、NULL とブランク文字は区別しておいた方がよい。

NULL とブランク文字を区別した場合の JSON のサンプルは以下の通り。

```
{
    "dateOfBirth" : null,
    "address1" : ""
}
```

日時のフォーマット

JSON の日時フィールドの形式は、**ISO-8601 の拡張形式とする事を推奨する。**
ISO-8601 の拡張形式以外でもよいが、特に理由がない場合は、ISO-8601 の拡張形式にすればよい。
ISO-8601 には基本形式と拡張形式があるが、拡張形式の方が視認性が高い表記方法である。

具体的には、以下の 3 つの形式となる。

1. yyyy-MM-dd

```
{
    "dateOfBirth" : "1977-03-12"
}
```

2. yyyy-MM-dd’T’HH:mm:ss.SSSZ

```
{
    "lastModifiedAt" : "2014-03-12T22:22:36.637+09:00"
}
```

3. yyyy-MM-dd’T’HH:mm:ss.SSS’Z』(UTC 用の形式)

```
{
    "lastModifiedAt" : "2014-03-12T13:11:27.356Z"
}
```

ハイパーメディアリンクの形式

ハイパーメディアリンクを設ける場合は、以下に示す形式とすることを推奨する。
推奨する形式のサンプルは以下の通り。

```
{
    "links" : [
        {
            "rel" : "ownerMember",
            "href" : "http://example.com/api/v1/memebers/M000000001"
        }
    ]
}
```

- `rel` と `href` という 2 つのフィールドを持った Link オブジェクトをコレクション形式で保持する。
- `rel` には、なんのリンクか識別するためのリンク名を指定する。
- `href` には、リソースにアクセスするための URI を指定する。
- Link オブジェクトをコレクション形式で保持するフィールドは、`links` とする。

エラー応答時のフォーマット

エラーを検知した場合、どのようなエラーが発生したのか保持できるフォーマットにする事を推奨する。
特に、クライアントが再操作する事でエラーが解消できる可能性がある場合は、より詳細なエラー情報を含めた方がよい。

逆に、システムの脆弱性をさらすような事象が発生した場合は、詳細なエラー情報は含めるべきではない。この場合、詳細なエラー情報はログに出力すべきである。

エラーを検知した際に応答するフォーマット例を以下に示す。

```
{
  "code" : "e.ex.fw.7001",
  "message" : "Validation error occurred on item in the request body.",
  "details" : [ {
    "code" : "ExistInCodeList",
    "message" : "\"genderCode\" must exist in code list of CL_GENDER.",
    "target" : "genderCode"
  } ]
}
```

上記のフォーマット例では、

- エラーコード (code)
- エラーメッセージ (message)
- エラー詳細リスト (details)

をエラー応答時のフォーマットとして用意している。

エラー詳細リストは、入力チェックエラー発生時に利用する事を想定しており、どのフィールドで、どのようなエラーが発生したのかを保持できるフォーマットとしている。

HTTP ステータスコード

HTTP ステータスコードは、以下の指針に則って応答する。

項番	方針
(1)	リクエストが成功した場合は、成功又は転送を示す HTTP ステータスコード (2xx 又は 3xx 系) を応答する。
(2)	リクエストが失敗した原因がクライアント側にある場合は、クライアントエラーを示す HTTP ステータスコード (4xx 系) を応答する。 リクエストが失敗した原因はクライアントにはないが、クライアントの再操作によってリクエストが成功する可能性がある場合も、クライアントエラーとする。
(3)	リクエストが失敗した原因がサーバ側にある場合は、サーバエラーを示す HTTP ステータスコード (5xx 系) を応答する。

リクエストが成功した場合の HTTP ステータスコード

リクエストが成功した場合は、状況に応じて以下の HTTP ステータスコードを応答する。

項番	HTTP ステータスコード	説明	適用条件
(1)	200 OK	リクエストが成功した事を通知する HTTP ステータスコード。	リクエストが成功した結果として、レスポンスのエンティティボディに、リクエストに対応するリソースの情報を出力する際に応答する。
(2)	201 Created	新しいリソースを作成した事を通知する HTTP ステータスコード。	POST メソッドを使用して、新しいリソースを作成した際に使用する。 レスポンスの Location ヘッダに、作成したリソースの URI を設定する。
(3)	204 No Content	リクエストが成功した事を通知する HTTP ステータスコード。	リクエストが成功した結果として、レスポンスのエンティティボディに、リクエストに対応するリソースの情報を出力しない時に応答する。

ちなみに： "`200 OK` と `204 No Content` の違いは、レスポンスボディにリソースの情報を出力する/しないの違いとなる。

リクエストが失敗した原因がクライアント側にある場合の HTTP ステータスコード

リクエストが失敗した原因がクライアント側にある場合は、状況に応じて以下の HTTP ステータスコードを応答する。

リソースを扱う個々の REST API で意識する必要があるステータスコードは以下の通り。

項番	HTTP ステータスコード	説明	適用条件
(1)	400 Bad Request	リクエストの構文やリクエストされた値が間違っている事を通知する HTTP ステータスコード。	エンティティボディに指定された JSON や XML の形式不備を検出した場合や、JSON や XML 又はリクエストパラメータに指定された入力値の不備を検出した場合に応答する。
(2)	404 Not Found	指定されたリソースが存在しない事を通知する HTTP ステータスコード。	指定された URI に対応するリソースがシステム内に存在しない場合に応答する。
(3)	409 Conflict	リクエストされた内容でリソースの状態を変更すると、リソースの状態に矛盾が発生ため処理を中止した事を通知する HTTP ステータスコード。	排他エラーが発生した場合や業務エラーを検知した場合に応答する。 エンティティボディには矛盾の内容や矛盾を解決するために必要なエラー内容を出力する必要がある。

リソースを扱う個々の REST API で意識する必要がないステータスコードは以下の通り。

以下のステータスコードは、フレームワークや共通処理として意識する必要がある。

項番	HTTP ステータスコード	説明	適用条件
(4)	405 Method Not Allowed	使用された HTTP メソッドが、指定されたリソースでサポートしていない事を通知する HTTP ステータスコード。	サポートされていない HTTP メソッドが使用された事を検知した場合に応答する。 レスポンスの Allow ヘッダに、許可されているメソッドの列挙を設定する。
(5)	406 Not Acceptable	指定された形式でリソースの状態を応答する事が出来ないため、リクエストを受理できない事を通知する HTTP ステータスコード。	レスポンス形式として、拡張子又は Accept ヘッダで指定された形式をサポートしていない場合に応答する。
(6)	415 Unsupported Media Type	エンティティボディに指定された形式をサポートしていないため、リクエストが受け取れない事を通知する HTTP ステータスコード。	リクエスト形式として、Content-Type ヘッダで指定された形式をサポートしていない場合に応答する。

リクエストが失敗した原因がサーバ側にある場合の HTTP ステータスコード

リクエストが失敗した原因がサーバ側にある場合は、状況に応じて以下の HTTP ステータスコードを応答する。

項番	HTTP ステータスコード	説明	適用条件
(1)	500 Internal Server Error	サーバ内部でエラーが発生した事を通知する HTTP ステータスコード。	サーバ内で予期しないエラーが発生した場合や、正常稼働時には発生してはいけない状態を検知した場合に応答する。

認証・認可

- OAuth2 の仕組みを使って認証・認可を行う仕組みについては、*OAuth* を参照されたい。

リソースの条件付き更新の制御

課題: TBD

HTTP ヘッダを使ったリソースの条件付き更新 (排他制御) をどのように行うか記載する。

Etag/Last-Modified などのヘッダを使って条件付き更新の仕組みについて、次版以降に記載する予定である。

リソースの条件付き取得の制御

課題: TBD

HTTP ヘッダを使ったリソースの条件付き取得 (304 Not Modified 制御) をどのように行うか記載する。

Etag/Last-Modified などのヘッダを使ったリソースの条件付き取得の仕組みについて、次版以降に記載する予定である。

リソースのキャッシュ制御

課題: TBD

HTTP ヘッダを使ったリソースのキャッシュ制御をどのように行うか記載する。

Cache-Control/Pragma/Expires などのヘッダを使ったリソースのキャッシュ制御の仕組みについて、次版以降に記載する予定である。

5.1.4 How to use

本節では、RESTful Web Service の具体的な作成方法について説明する。

Web アプリケーションの構成

RESTful Web Service を構築する場合は、以下のいずれかの構成で Web アプリケーション (war) を構築する。

特に理由がない場合は、**RESTful Web Service** 専用の **Web** アプリケーションとして構築する事を推奨する。

項番	構成	説明
(1)	RESTful Web Service 専用の Web アプリケーションとして構築する。	RESTful Web Service を利用するクライアントアプリケーション (UI 層のアプリケーション) との独立性を確保したい (する必要がある) 場合は、RESTful Web Service 専用の Web アプリケーション (war) として構築することを推奨する。 RESTful Web Service を利用するクライアントアプリケーションが複数になる場合は、この方法で RESTful Web Service を生成することになる。
(2)	RESTful Web Service 用の `DispatcherServlet` を設けて構築する。	RESTful Web Service を利用するクライアントアプリケーション (UI 層のアプリケーション) との独立性を確保する必要がない場合は、RESTful Web Service とクライアントアプリケーションを一つの Web アプリケーション (war) として構築してもよい。 ただし、RESTful Web Service 用のリクエストを受ける `DispatcherServlet` と、クライアントアプリケーション用のリクエストを受け取る `DispatcherServlet` は分割して構築することを強く推奨する。

注釈: クライアントアプリケーション (UI 層のアプリケーション) とは

ここで言うクライアントアプリケーション (UI 層のアプリケーション) とは、HTML, JavaScript などのスクリプト, CSS(Cascading Style Sheets) といったクライアント層 (UI 層) のコンポーネントを応答するアプリケーションの事をさす。JSP などのテンプレートエンジンによって生成される HTML も対象となる。

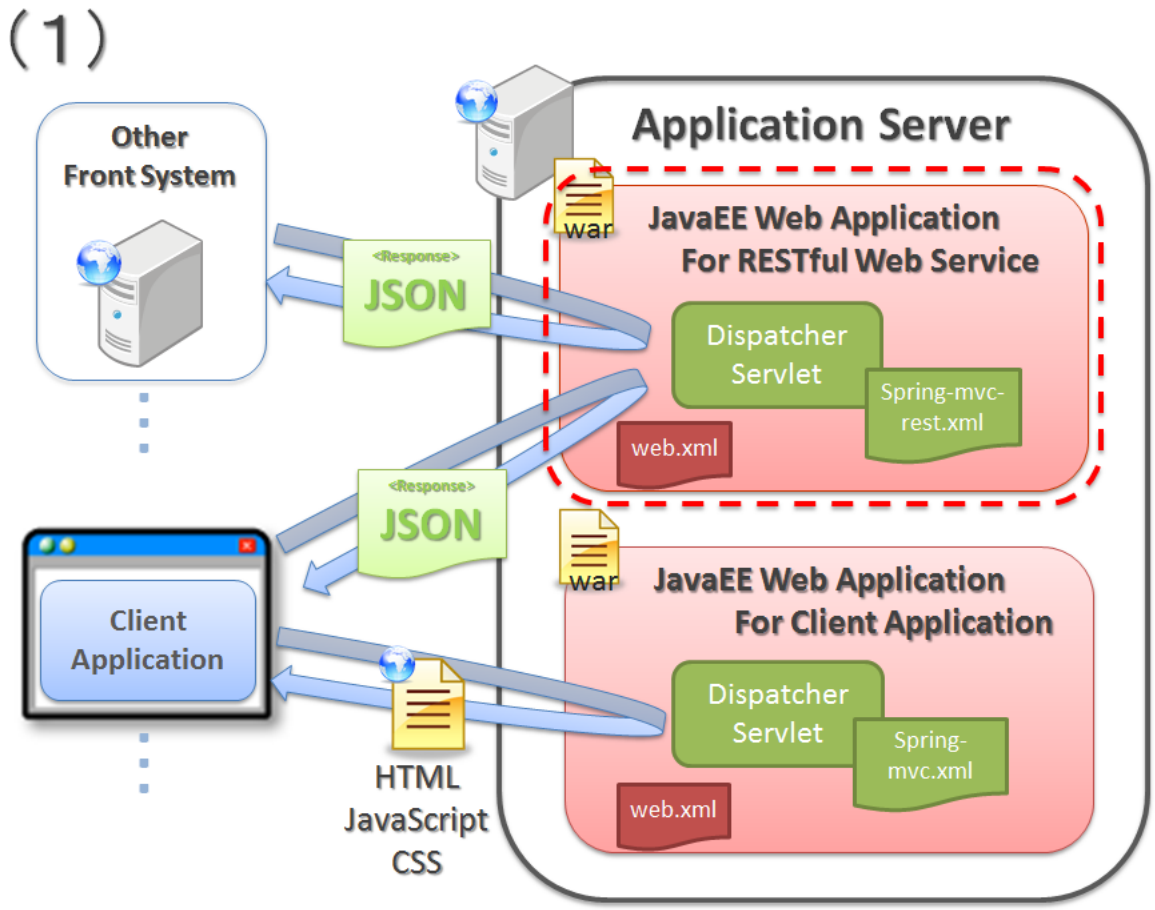
注釈: DispatcherServlet を分割する事を推奨する理由

Spring MVC では、`DispatcherServlet` 毎にアプリケーションの動作設定を定義することになる。そのため、RESTful Web Service とクライアントアプリケーション (UI 層のアプリケーション) のリクエストを同じ `DispatcherServlet` で受ける構成にしてしまうと、RESTful Web Service 又はクライアントアプリケーション固有の動作設定を定義する事ができなくなったり、設定が煩雑又は複雑になることがある。

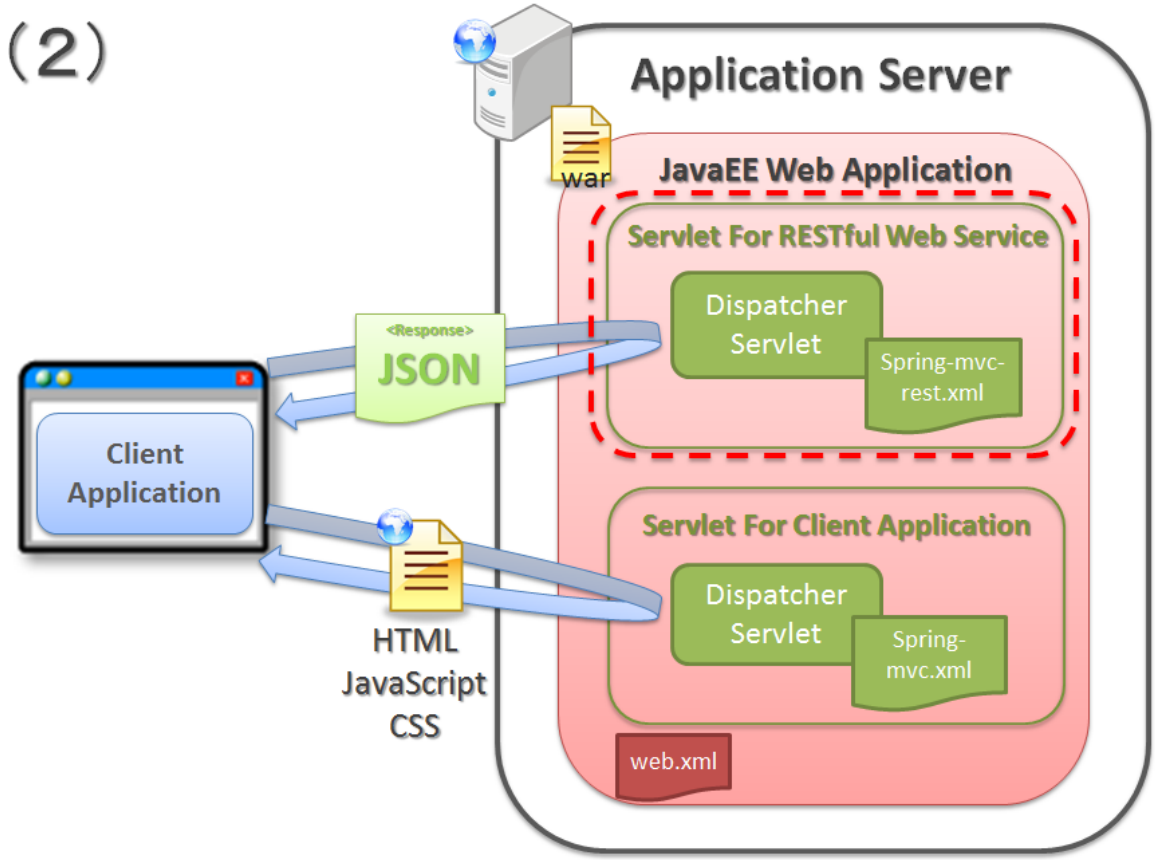
本ガイドラインでは、上記の様な問題が起こらないようにするために、RESTful Web Service につ

いてクライアントアプリケーションを同じ Web アプリケーション (war) として構築する場合は、`DispatcherServlet` を分割することを推奨している。

RESTful Web Service 専用の Web アプリケーションとして構築する際の構成イメージは以下の通り。



RESTful Web Service とクライアントアプリケーションを一つの Web アプリケーションとして構築する際の構成イメージは以下の通り。



pom.xml の設定

terasoluna-gfw-common-dependencies を使用していれば、依存関係の設定は不要である。

警告: Java SE 7 環境にて使用する場合の設定

terasoluna-gfw-common-dependencies は Java SE 8 を前提とした依存関係を設定している。Java SE 7 環境にて使用する場合は下記のように Java SE 8 依存ライブラリを exclusion すること。java SE 8 依存ライブラリについてはアーキテクチャ概要の「利用する *OSS* のバージョン」を参照

```
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-common-dependencies</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.fasterxml.jackson.datatype</groupId>
            <artifactId>jackson-datatype-jsr310</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

アプリケーションの設定

RESTful Web Service 向けのアプリケーションの設定について説明する。

> **警告：StAX(Streaming API for XML) 使用時の DoS 攻撃対策について**
>
> XML 形式のデータについて StAX を使用して解析する場合は、DTD を使った DoS 攻撃を受けないように対応する必要がある。詳細は、CVE-2015-3192 - DoS Attack with XML Input を参照されたい。

RESTful Web Service で必要となる Spring MVC のコンポーネントを有効化するための設定

RESTful Web Service 用の bean 定義ファイルを作成する。

以降の説明で示すサンプルを動かす際に必要となる定義を、以下に示す。

- spring-mvc-rest.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
        https://www.springframework.org/schema/util/spring-util.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd
">

    <!-- Load properties files for placeholder. -->
    <!-- (1) -->
    <context:property-placeholder
        location="classpath*:/META-INF/spring/*.properties" />

    <bean id="jsonMessageConverter"
        class="org.springframework.http.converter.json.
↪MappingJackson2HttpMessageConverter">
        <property name="objectMapper" ref="objectMapper" />
    </bean>

    <bean id="objectMapper" class="org.springframework.http.converter.json.
↪Jackson2ObjectMapperFactoryBean">
        <!-- (2) -->
```

```
        <property name="dateFormat">
            <bean class="com.fasterxml.jackson.databind.util.StdDateFormat" />
        </property>
    </bean>

    <!-- Register components of Spring MVC. -->
    <!-- (3) -->
     <mvc:annotation-driven>
        <mvc:message-converters register-defaults="false">
            <ref bean="jsonMessageConverter" />
        </mvc:message-converters>
        <!-- (4) -->
        <mvc:argument-resolvers>
            <bean class="org.springframework.data.web.
↪PageableHandlerMethodArgumentResolver" />
        </mvc:argument-resolvers>
    </mvc:annotation-driven>

    <!-- Register components of interceptor. -->
    <!-- (5) -->
    <mvc:interceptors>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <bean class="org.terasoluna.gfw.web.logging.
↪TraceLoggingInterceptor" />
        </mvc:interceptor>
        <!-- omitted -->
    </mvc:interceptors>

    <!-- Scan & register components of RESTful Web Service. -->
    <!-- (6) -->
    <context:component-scan base-package="com.example.project.api" />

    <!-- Register components of AOP. -->
    <!-- (7) -->
    <bean id="handlerExceptionResolverLoggingInterceptor"
        class="org.terasoluna.gfw.web.exception.
↪HandlerExceptionResolverLoggingInterceptor">
        <property name="exceptionLogger" ref="exceptionLogger" />
    </bean>
    <aop:config>
        <aop:advisor advice-ref="handlerExceptionResolverLoggingInterceptor"
            pointcut="execution(* org.springframework.web.servlet.
↪HandlerExceptionResolver.resolveException(..))" />
    </aop:config>

</beans>
```

項番	説明
(1)	アプリケーション層のコンポーネントでプロパティファイルに定義されている値を参照する必要がある場合は、`<context:property-placeholder>`要素を使用してプロパティファイルを読み込む必要がある。 プロパティファイルから値を取得する方法の詳細については、「プロパティ管理」を参照されたい。
(2)	JSON の日付フィールドの形式を ISO-8601 の拡張形式として扱うための設定を追加する。 なお、リソースを表現する JavaBean(Resource クラス) のプロパティとして Joda Time のクラスを使用する場合は、「*JSR-310 Date and Time API / Joda Time* を使う場合の設定」を行う必要がある。
(3)	RESTful Web Service を提供するために必要となる Spring MVC のフレームワークコンポーネントを bean 登録する。 本設定を行うことで、リソースのフォーマットとして JSON を使用する事ができる。 上記例では、`<mvc:message-converters>`要素の register-defaults 属性を `false` にしているので、リソースの形式は JSON に限定される。 リソースのフォーマットとして XML を使用する場合は、XXE 対策が行われている XML 用の `MessageConverter` を指定すること。指定方法は、「*XXE* 対策の有効化」を参照されたい。
(4)	ページ検索機能を有効にするための設定を追加する。 ページ検索の詳細については、「ページネーション」を参照されたい。 ページ検索が必要ない場合は、本設定は不要であるが、定義があっても問題はない。
(5)	Spring MVC のインターセプタを bean 登録する。 上記例では、共通ライブラリから提供されている `TraceLoggingInterceptor` のみを定義しているが、データアクセスとして JPA を使う場合は、別途 `OpenEntityManagerInViewInterceptor` の設定を追加する必要がある。 `OpenEntityManagerInViewInterceptor` については、「データベースアクセス（*JPA* 編）」を参照されたい。

Continued on next page

表 5.3 – continued from previous page

項番	説明
(6)	RESTful Web Service 用のアプリケーション層のコンポーネント (Controller や Helper クラスなど) をスキャンして bean 登録する。 `com.example.project.api` の部分はプロジェクト毎のパッケージ名となる。
(7)	Spring MVC のフレームワークでハンドリングされた例外を、ログ出力するための AOP 定義を指定する。 `HandlerExceptionResolverLoggingInterceptor` については、「例外ハンドリング」を参照されたい。

注釈: **ObjectMapper の Bean 定義方法について**

Jackson の `com.fasterxml.jackson.databind.ObjectMapper` の Bean 定義を行う場合は、Spring が提供している `Jackson2ObjectMapperFactoryBean` を使用するとよい。`Jackson2ObjectMapperFactoryBean` を使用すると、JSR-310 Date and Time API や Joda Time 用の拡張モジュールを自動登録することができ、さらに XML の Bean 定義ファイル上で表現が難しかった `ObjectMapper` のコンフィギュレーションも簡単に行うことができる。

なお、`ObjectMapper` を直接 Bean 定義するスタイルから `Jackson2ObjectMapperFactoryBean` を使用するスタイルに変更する場合は、以下のコンフィギュレーションに対するデフォルト値が Jackson のデフォルト値と異なる (無効化されている) 点に注意すること。

- MapperFeature#DEFAULT_VIEW_INCLUSION
- DeserializationFeature#FAIL_ON_UNKNOWN_PROPERTIES

`ObjectMapper` の動作を Jackson のデフォルト動作にあわせたい場合は、`featuresToEnable` プロパティを使用して上記のコンフィギュレーションを有効化する。

```
<bean id="objectMapper" class="org.springframework.http.converter.json.
↪Jackson2ObjectMapperFactoryBean">
    <!-- ... -->
    <property name="featuresToEnable">
        <array>
            <util:constant static-field="com.fasterxml.jackson.databind.
↪MapperFeature.DEFAULT_VIEW_INCLUSION"/>
            <util:constant static-field="com.fasterxml.jackson.databind.
↪DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES"/>
        </array>
    </property>
</bean>
```

`Jackson2ObjectMapperFactoryBean` の詳細については、Jackson2ObjectMapperFactoryBean の JavaDoc を参照されたい。

注釈: **jackson version 1.x.x から jackson version 2.x.x へ変更する場合の注意点**

- パッケージの変更

verision	package
1.x.x	*org.codehaus.jackson*
2.x.x	*com.fasterxml.jackson*

    - 注意事項として、配下のパッケージ構成も変更されている。

- Deprecated 一覧
- http://fasterxml.github.io/jackson-core/javadoc/2.8/deprecated-list.html
- http://fasterxml.github.io/jackson-databind/javadoc/2.8/deprecated-list.html
- http://fasterxml.github.io/jackson-annotations/javadoc/2.8/deprecated-list.html

RESTful Web Service 用のサーブレットの設定

下記の設定は、RESTful Web Service とクライアントアプリケーションを別の Web アプリケーションとして構築する場合の設定例となっている。

RESTful Web Service とクライアントアプリケーションを同じ Web アプリケーションとして構築する場合は、「*RESTful Web Service とクライアントアプリケーションを同じ Web アプリケーションとして動かす際の設定*」を行う必要がある。

- web.xml

```
<!-- omitted -->

<servlet>
    <!-- (1) -->
    <servlet-name>restAppServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
↪class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <!-- (2) -->
        <param-value>classpath*:META-INF/spring/spring-mvc-rest.xml</param-
↪value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<!-- (3) -->
<servlet-mapping>
    <servlet-name>restAppServlet</servlet-name>
    <url-pattern>/api/v1/*</url-pattern>
</servlet-mapping>

<!-- omitted -->
```

項番	説明
(1)	<servlet-name>要素に、RESTful Web Service 用のサーブレットであることを示す名前を指定する。 上記例では、サーブレット名として restAppServlet を指定している。
(2)	RESTful Web Service 用の DispatcherServlet を構築する際に使用する Spring MVC の bean 定義ファイルを指定する。 上記例では、Spring MVC の bean 定義ファイルとして、クラスパス上にある META-INF/spring/spring-mvc-rest.xml を指定している。
(3)	RESTful Web Service 用の DispatcherServlet へマッピングするサーブレットパスのパターンの指定を行う。 上記例では、/api/v1/配下のサーブレットパスを RESTful Web Service 用の DispatcherServlet にマッピングしている。 具体的には、 /api/v1/ /api/v1/members /api/v1/members/xxxxx といったサーブレットパスが、RESTful Web Service 用の DispatcherServlet(restAppServlet) にマッピングされる。

ちなみに: @RequestMapping アノテーションの value 属性に指定する値について

@RequestMapping アノテーションの value 属性に指定する値は、<url-pattern>要素で指定したワイルドカード (「*」) の部分の値を指定する。

例えば、@RequestMapping(value = "members") と指定した場合、/api/v1/members というパスに対する処理を行うメソッドとしてデプロイされる。そのため、@RequestMapping アノテーションの value 属性には、分割したサーブレットへマッピングするためパス (api/v1) を指定する必要はない。

@RequestMapping(value = "api/v1/members") と指定すると、/api/v1/api/v1/members というパスに対する処理を行うメソッドとしてデプロイされてしまうので、注意すること。

REST API の実装

REST API の実装方法について説明する。

以降の説明では、ショッピングサイトの会員情報 (Member リソース) に対する REST API の実装例を使用して、説明を行う。

> **注釈:** 本節では、ドメイン層の実装の説明は行わないが、「*REST API 実装時に作成したドメイン層のクラスのソースコード*」として、添付しておく。
>
> 必要に応じて、参照されたい。

まず、説明で使用する REST API の仕様を以下に示す。

リソースの形式

会員情報のリソースの形式は、以下のような JSON 形式とする。

下記の例では、全フィールドを表示しているが、全ての API のリクエストとレスポンスで使用するわけではない。

例えば、`password` はリクエストのみで使用、`createdAt` や `lastModifiedAt` はレスポンスのみ使用などの違いがある。

```
{
    "memberId" : "M000000001",
    "firstName" : "Firstname",
    "lastName" : "Lastname",
    "genderCode" : "1",
    "dateOfBirth" : "1977-03-13",
    "emailAddress" : "user1@test.com",
    "telephoneNumber" : "09012345678",
    "zipCode" : "1710051",
    "address" : "Tokyo",
    "credential" : {
        "signId" : "user1@test.com",
        "password" : "zaq12wsx",
```

```
        "passwordLastChangedAt" : "2014-03-13T04:39:14.831Z",
        "lastModifiedAt" : "2014-03-13T04:39:14.831Z"
    },
    "createdAt" : "2014-03-13T04:39:14.831Z",
    "lastModifiedAt" : "2014-03-13T04:39:14.831Z"
}
```

注釈: 本節では、関連リソースへのハイパーメディアリンクは設けない例となっている。ハイパーメディアリンクを設ける場合の実装例は、「ハイパーメディアリンクの実装」を参照されたい。

リソースの項目仕様

リソース (JSON) の項目毎の仕様は以下の通りとする。

項番	項目名	型	I/O 仕様	桁 数 (min-max)	その他の仕様
(1)	memberId	String	I/O	10-10	POST Members のリクエスト時は未指定 (NULL) であること。
(2)	firstName	String	I/O	1-128	-
(3)	lastName	String	I/O	1-128	-
(4)	genderCode	String (Code)	I/O	1-1	「0」: UNKNOWN 「1」: MEN 「2」: WOMEN
(5)	dateOfBirth	Date	I/O	-	yyyy-MM-dd 形式 (ISO-8601 拡張形式)

Continued on next page

表 5.4 – continued from previous page

項番	項目名	型	I/O 仕様	桁 数 (min-max)	その他の仕様
(6)	emailAddress	String (E-mail)	I/O	1-256	-
(7)	telephoneNumber	String	I/O	0-20	-
(8)	zipCode	String	I/O	0-20	-
(9)	address	String	I/O	0-256	-
(10)	credential	Object (MemberCredential)	I/O	-	POST Members のリクエスト時は指定されていること。
(11)	credential/signId	String (E-mail)	I/O	0-256	指定がない場合は、emailAddress の値を適用する。
(12)	credential/password	String	I	8-32	-
(13)	credential/passwordLastChangedAt	Timestamp with time-zone	O	-	yyyy-MM-dd’T’HH:mm:ss.SSS’Z’ 形式 (ISO-8601 拡張形式)
(14)	credential/lastModifiedAt	Timestamp with time-zone	O	-	yyyy-MM-dd’T’HH:mm:ss.SSS’Z’ 形式 (ISO-8601 拡張形式)

Continued on next page

表 5.4 – continued from previous page

項番	項目名	型	I/O 仕様	桁 数 (min-max)	その他の仕様
(15)	createdAt	Timestamp with time-zone	O	-	yyyy-MM-dd’T’HH:mm:ss.SSS’Z’ 形式 (ISO-8601 拡張形式)
(16)	lastModifiedAt	Timestamp with time-zone	O	-	yyyy-MM-dd’T’HH:mm:ss.SSS’Z’ 形式 (ISO-8601 拡張形式)

REST API 一覧

実装する REST API は以下の 5 つの API とする。

項番	API 名	HTTP メソッド	リソースパス	ステータスコード	API 概要
(1)	*GET Members*	GET	`/api/v1/members`	200 (OK)	条件に一致する Member リソースをページ検索する。
(2)	*POST Members*	POST	`/api/v1/members`	201 (Created)	Member リソースを一件作成する。
(3)	*GET Member*	GET	`/api/v1/members/{memberId}`	200 (OK)	Member リソースを一件取得する。
(4)	*PUT Member*	PUT	`/api/v1/members/{memberId}`	200 (OK)	Member リソースを一件更新する。
(5)	*DELETE Member*	DELETE	`/api/v1/members/{memberId}`	204 (No Content)	Member リソースを一件削除する。

注釈: Spring Framework 4.3 より HEAD と OPTIONS メソッドに対する REST API が暗黙的に用意される様になったため、開発者がこれらの REST API を明示的に実装する必要はない。

なお、暗黙的に用意される OPTIONS 用の REST API がレスポンスする Allow ヘッダの中には OPTIONS 自体が含まれないため、TERASOLUNA Server Framework for Java 5.2.x までの開発ガイドラインで紹介している実装例と異なる点に留意されたい。

REST API 用パッケージの作成

REST API 用のクラスを格納するパッケージを作成する。

REST API 用のクラスを格納するルートパッケージのパッケージ名は `api` として、配下にリソース毎のパッケージ (リソース名の小文字) を作成する事を推奨する。

説明で扱うリソース名は `Member` なので、`org.terasoluna.examples.rest.api.member` というパッケージとする。

注釈: 作成したパッケージに格納するクラスは、通常以下の 4 種類となる。作成するクラスのクラス名は、以下のネーミングルールとする事を推奨する。

- [リソース名]`Resource`
- [リソース名]`RestController`
- [リソース名]`Validator` (必要に応じて作成する)
- [リソース名]`Helper` (必要に応じて作成する)

説明で扱うリソースのリソース名は `Member` なので、

- `MemberResource`
- `MemberRestController`
- `MemberValidator`
- `MemberHelper`

となる。

関連リソースを扱う場合、関連リソース用のクラスも同じパッケージに配置すればよい。

REST API 用の共通部品を格納するパッケージは、REST API 用のクラスを格納するルートパッケージ直下に `common` という名前で作成し、サブパッケージは機能単位に作成する事を推奨する。

例えば、エラーハンドリングを行う共通部品を格納するサブパッケージの場合、`error` という名前でサブパッケージを作成する。

以降の説明で作成する例外ハンドリング用のクラスは、
`org.terasoluna.examples.rest.api.common.error` というパッケージに格納している。

注釈: 共通部品が格納されているパッケージという事がわかれば、パッケージ名は `common` 以外でもよい。

Resource クラスの作成

本ガイドラインでは、Web 上に公開するリソースを表現 (JSON や XML を表現) するクラスとして、
Resource クラスを設けることを推奨する。

注釈: Resource クラスを作成する理由

DomainObject クラス (例えば Entity クラス) があるにも関わらず、Resource クラスを作成する理由は、クライアントとの入出力で使用するユーザーインタフェース (UI) 上の情報と業務処理で扱う情報は必ずしも一致しないためである。

これらを混同して使用すると、アプリケーション層の影響がドメイン層におよび、保守性を低下させる原因となる。DomainObject と Resource クラスは別々に作成し、Dozer 等の BeanMapper を利用してデータ変換を行うことを推奨する。

Resource クラスの役割は以下の通りである。

項番	役割	説明
(1)	リソースのデータ構造の定義を行う。	Web 上に公開するリソースのデータ構造を定義する。 データベースなどの永続層で管理しているデータの構造をそのまま Web 上のリソースとして公開する事は、一般的には稀である。
(2)	フォーマットに関する定義を行う。	リソースのフォーマットに関する定義を、アノテーションを使って指定する。 使用するアノテーションは、リソースの形式 (JSON/XML など) よって異なり、JSON 形式の場合は Jackson のアノテーション、XML 形式の場合は JAXB のアノテーションを使用する事になる。
(3)	入力チェックルールの定義を行う。	項目毎の単項目の入力チェックルールを、Bean Validation のアノテーションを使って指定する。 入力チェックの詳細については、「入力チェック」を参照されたい。

警告: 循環参照への対策

Resource クラス (JavaBean) を JSON や XML 形式にシリアライズする際に、相互参照関係のオブジェクトをプロパティに保持していると、循環参照となり `StackOverflowError` や `OutOfMemoryError` などが発生するので、注意が必要である。

循環参照を回避するためには、

- Jackson を使用して JSON 形式にシリアライズする場合は、シリアライズ対象から除外するプロパティに`@com.fasterxml.jackson.annotation.JsonIgnore` アノテーション
- JAXB を使用して XML 形式にシリアライズする場合は、シリアライズ対象から除外するプロパティに `javax.xml.bind.annotation.XmlTransient` アノテーション

を付与すればよい。

以下に Jackson を使用して JSON 形式にシリアライズする際の回避例を示す。

```
public class Order {
    private String orderId;
    private List<OrderLine> orderLines;
    // ...
}
```

```
public class OrderLine {
    @JsonIgnore
    private Order order;
    private String itemCode;
    private int quantity;
    // ...
}
```

項番	説明
(1)	シリアライズ対象から除外するプロパティに対して@JsonIgnore アノテーションを付与する。

警告: 電文から Java Bean にデシリアライズする際、プロパティにジェネリクスやインターフェイスを使用しているなどの理由で型を特定できない場合は@com.fasterxml.jackson.annotation.JsonTypeInfo アノテーションを付与する。@JsonTypeInfo アノテーションを付与したプロパティをシリアライズすると JSON に型情報が出力され、これを読み取ってデシリアライズが行われる。

ただし、@JsonTypeInfo アノテーションの use 属性に Id.CLASS や Id.MINIMAL_CLASS を使用すると、JSON に出力されたクラス名を元にデシリアライズが行われるため、これにより不正にリモートコードが実行される危険がある。このため、(信頼できない送信元を含み得る) 不特定多数からの電文を受け付ける前提のシステムにおいては、Id.CLASS や Id.MINIMAL_CLASS を指定してはならない。

なお、ObjectMapper の defaultTyping を利用すると、上記のようなデシリアライズ時の型判断をアプリケーション全体に適用することが可能である。こちらも合わせて注意されたい。

以下に Resource クラスの作成例を示す。

- MemberResource.java

```
package org.terasoluna.examples.rest.api.member;

import java.io.Serializable;
```

```
import javax.validation.Valid;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Null;
import javax.validation.constraints.Past;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotEmpty;
import org.joda.time.DateTime;
import org.joda.time.LocalDate;
import org.springframework.format.annotation.DateTimeFormat;
import org.terasoluna.gfw.common.codelist.ExistInCodeList;

// (1)
public class MemberResource implements Serializable {

    private static final long serialVersionUID = 1L;

    // (2)
    interface PostMembers {
    }

    interface PutMember {
    }

    @Null(groups = PostMembers.class)
    @NotEmpty(groups = PutMember.class)
    @Size(min = 10, max = 10, groups = PutMember.class)
    private String memberId;

    @NotEmpty
    @Size(max = 128)
    private String firstName;

    @NotEmpty
    @Size(max = 128)
    private String lastName;

    @NotEmpty
    @ExistInCodeList(codeListId = "CL_GENDER")
    private String genderCode;

    @NotNull
    @Past
    private LocalDate dateOfBirth;

    @NotEmpty
    @Size(max = 256)
    @Email
    private String emailAddress;

    @Size(max = 20)
```

```
    private String telephoneNumber;

    @Size(max = 20)
    private String zipCode;

    @Size(max = 256)
    private String address;

    @NotNull(groups = PostMembers.class)
    @Null(groups = PutMember.class)
    @Valid
    // (3)
    private MemberCredentialResource credential;

    @Null
    private DateTime createdAt;

    @Null
    private DateTime lastModifiedAt;

    // omitted setter and getter

}
```

項番	説明
(1)	Member リソースを表現する JavaBean。
(2)	Bean Validation のバリデーショングループを指定するためのインタフェースを定義している。 実装例では、POST と PUT で異なる入力チェックを行うため、バリデーションをグループ化して入力チェックを行っている。 バリデーションのグループ化については、「入力チェック」を参照されたい。
(3)	関連リソースをネストした JavaBean をフィールドに定義している。 実装例では、会員の資格情報 (サイン ID とパスワード) を関連リソースとして扱っている。 これは、サイン ID の変更やパスワードの変更のみ行うという操作を考慮して、関連リソースとして切り出している。 ただし、関連リソースに対する REST API の実装例については割愛している。

- `MemberCredentialResource.java`

```
package org.terasoluna.examples.rest.api.member;

import java.io.Serializable;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Null;
import javax.validation.constraints.Size;

import com.fasterxml.jackson.annotation.JsonInclude;
import org.hibernate.validator.constraints.Email;
import org.joda.time.DateTime;

// (4)
public class MemberCredentialResource implements Serializable {

    private static final long serialVersionUID = 1L;

    @Size(max = 256)
    @Email
    private String signId;

    // (5)
    @JsonInclude(JsonInclude.Include.NON_NULL)
    @NotNull
    @Size(min = 8, max = 32)
    private String password;

    @Null
    private DateTime passwordLastChangedAt;

    @Null
    private DateTime lastModifiedAt;

    // omitted setter and getter

}
```

項番	説明
(4)	Member リソースの関連リソースとなる Credential リソースを表現する JavaBean。
(5)	値が `null` の時に、JSON にフィールド自体を出力しないようにするためのアノテーションを指定している。 これは、レスポンスする JSON の中にパスワードのフィールド出力しないようにするために行っている。 上記例では NULL の場合 (`Inclusion.NON_NULL`) に限っているが、値が空の場合 (`Inclusion.NON_EMPTY`) という指定も可能である。

- Bean のマッピング定義の追加

  これから説明する実装例において、Entity クラスと Resource クラスのコピーは「*Bean マッピング (Dozer)*」を使用する。

  上記に示した JavaBean には、Joda-Time のクラスである `org.joda.time.DateTime` と `org.joda.time.LocalDate` が含まれているが、Joda Time オブジェクトは Dozer でサポートされていないため、「カスタムコンバーターの作成」が必要となる。

Controller クラスの作成

Controller クラスはリソース毎に作成する。

全ての API の実装が完了した際のソースコードについては、*Appendix* を参照されたい。

```
package org.terasoluna.examples.rest.api.member;

// omitted
import org.springframework.web.bind.annotation.RestController;
// omitted

@RequestMapping("members") // (1)
@RestController // (2)
public class MemberRestController {
```

```
    // omitted ...

}
```

項番	説明
(1)	Controller に対して、リソースのコレクション用の URI(サーブレットパス) をマッピングする。 具体的には、`@RequestMapping` アノテーションの value 属性に、リソースのコレクションを表すサーブレットパスを指定する。 上記例では、 `/api/v1/members` というサーブレットパスをマッピングしている。
(2)	Controller に対して、`@RestController` アノテーションを付与する。 `@RestController` アノテーションを付与することで、 • クラスに `org.springframework.stereotype.Controller` アノテーションを付与 • 以降で説明する Controller のメソッドに`@org.springframework.web.bind.annotation.ResponseBody` アノテーションを付与 したのと同じ意味となる。 Controller のメソッドに`@ResponseBody` を付与することで、返却した Resource オブジェクトが JSON や XML に marshal され、レスポンス BODY に設定される。

ちなみに: `@RestController` アノテーションは、Spring Framework 4.0 から追加されたアノテーションである。

`@RestController` アノテーションの登場により、Controller の各メソッドに`@ResponseBody` アノテーションを付与する必要がなくなったため、REST API 用の Controller をよりシンプルに作成出来るようになった。`@RestController` アノテーションの詳細については、こちらを参照されたい。

従来通り`@Controller` アノテーションと`@ResponseBody` アノテーションを組み合わせて REST API 用の Controller を作成する例を以下に示す。

```
@RequestMapping("members")
@Controller
public class MemberRestController {

    @RequestMapping(method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    @ResponseBody
    public Page<MemberResource> getMembers() {
        // ...
    }

    // ...
```

```
}
```

リソースのコレクションを取得する REST API の実装

URI で指定された Member リソースのコレクションをページ検索する REST API の実装例を、以下に示す。

- 検索条件を受け取るための JavaBean の作成

  リソースのコレクションを取得する際に、検索条件が必要な場合は、検索条件を受け取るための JavaBean を作成する。

```
// (1)
public class MembersSearchQuery implements Serializable {
    private static final long serialVersionUID = 1L;

    // (2)
    @NotEmpty
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

項番	説明
(1)	検索条件を受け取るための JavaBean を作成する 検索条件が不要な場合は、JavaBean の作成は不要である。
(2)	プロパティ名は、リクエストパラメータのパラメータ名と一致させる。 上記例では、`/api/v1/members?name=John` というリクエストの場合、JavaBean の name プロパティに `John` という値が設定される。

- REST API の実装

  Member リソースのコレクションをページ検索する処理を実装する。

```
@RequestMapping("members")
@RestController
public class MemberRestController {

    // omitted

    @Inject
    MemberService memberService;

    @Inject
    Mapper beanMapper;

    // (3)
    @RequestMapping(method = RequestMethod.GET)
    // (4)
    @ResponseStatus(HttpStatus.OK)
    public Page<MemberResource> getMembers(
            // (5)
            @Validated MembersSearchQuery query,
            // (6)
            Pageable pageable) {

        // (7)
        Page<Member> page = memberService.searchMembers(query.getName(),␣
↪pageable);

        // (8)
        List<MemberResource> memberResources = new ArrayList<>();
        for (Member member : page.getContent()) {
            memberResources.add(beanMapper.map(member, MemberResource.class));
        }
        Page<MemberResource> responseResource = new PageImpl<>
↪(memberResources,
                pageable, page.getTotalElements());

        // (9)
        return responseResource;
    }

    // omitted

}
```

項番	説明

(3)	`@RequestMapping` アノテーションの method 属性に、`RequestMethod.GET` を指定する。
	
	注釈: HTTP メソッドごとの@RequestMapping アノテーション
	Spring Framework 4.3 から、HTTP メソッドごとの `@RequestMapping` 合成アノテーションが追加された。よりシンプルにマッピングを定義することができ、意図しない HTTP メソッドのマッピング防止とソースコードの可読性向上が期待できる。
	• `@GetMapping`
	• `@PostMapping`
	• `@PutMapping`
	• `@DeleteMapping`
	• `@PatchMapping`
	以下の定義は、`@RequestMapping(value = "hello", method = RequestMethod.GET)` と定義しているのと同様である。
	`@GetMapping(value = "hello")`
	`public String hello() {`
	`  // ...`
	`}`
	詳細は、Spring Framework Reference Documentation - Composed @RequestMapping Variants を参照されたい。

Continued on next page

表 5.6 – continued from previous page

項番	説明
(4)	メソッドアノテーションとして、`@org.springframework.web.bind.annotation.ResponseStatus` アノテーションを付与し、応答するステータスコードを指定する。 `@ResponseStatus` アノテーションの value 属性には、200(OK) を設定する。 --- **ちなみに：ステータスコードの指定方法について** 本例では、`@ResponseStatus` アノテーションを使って応答するステータスコードを固定で指定しているが、Controller のロジック内で指定する事もできる。 `public ResponseEntity<Page<MemberResource>> getMembers( @Validated MembersSearchQuery query, Pageable pageable) { // omitted return ResponseEntity.ok().body(responseResource); }` 応答するステータスコードを処理内容や処理結果に応じて変える必要がある場合は、上記実装例の様に、`org.springframework.http.ResponseEntity` を使用する事になる。 ---
(5)	検索条件を受け取るための JavaBean を引数に指定する。 入力チェックが必要な場合は、引数アノテーションとして、`@Validated` アノテーションを付与する。入力チェックの詳細については、「入力チェック」を参照されたい。
(6)	ページ検索が必要な場合は、`org.springframework.data.domain.Pageable` を引数に指定する。 ページ検索の詳細については、「ページネーション」を参照されたい。
(7)	ドメイン層の Service のメソッドを呼び出し、条件に一致するリソースの情報 (Entity など) を取得する。 ドメイン層の実装については、「ドメイン層の実装」を参照されたい。

Continued on next page

表 5.6 – continued from previous page

項番	説明
(8)	条件に一致したリソースの情報 (Entity など) をもとに、Web 上に公開する情報を保持する Resource オブジェクトを生成する。 ページ検索の結果を応答する際は、 `org.springframework.data.domain.PageImpl` クラスを使用することで、ページ検索時の応答として必要な項目をクライアントに返却する事ができる。 上記例では、Bean マッピングライブラリを使用して Entity から Resource オブジェクトを生成している。Bean マッピングライブラリについては、「*Bean マッピング (Dozer)*」を参照されたい。 **Resource オブジェクトを生成するためのコード量が多くなる場合は、Helper クラスに Resource オブジェクトを生成するためのメソッドを作成することを推奨する。**
(9)	(8) で生成した Resource オブジェクトを返却する。 ここで返却したオブジェクトが JSON や XML に marshal され、レスポンス BODY に設定される。

PageImpl クラスを使用した時のレスポンスは以下の様になる。

ハイライトしている部分が、ページ検索固有の項目となる。


```
{
  "content" : [ {
    "memberId" : "M000000001",
    "firstName" : "John",
    "lastName" : "Smith",
    "genderCode" : "1",
    "dateOfBirth" : "1977-03-07",
    "emailAddress" : "john.smith@test.com",
    "telephoneNumber" : "09012345678",
    "zipCode" : "1710051",
    "address" : "Tokyo",
    "credential" : {
      "signId" : "john.smit@test.com",
      "passwordLastChangedAt" : "2014-03-13T10:18:08.003Z",
      "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
    },
    "createdAt" : "2014-03-13T10:18:08.003Z",
    "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
  }, {
    "memberId" : "M000000002",
    "firstName" : "Sophia",
    "lastName" : "Smith",
    "genderCode" : "2",
    "dateOfBirth" : "1977-03-07",
    "emailAddress" : "sophia.smith@test.com",
    "telephoneNumber" : "09012345678",
    "zipCode" : "1710051",
    "address" : "Tokyo",
    "credential" : {
      "signId" : "sophia.smith@test.com",
      "passwordLastChangedAt" : "2014-03-13T10:18:08.003Z",
      "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
    },
    "createdAt" : "2014-03-13T10:18:08.003Z",
    "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
  } ],
  "last" : false,
  "totalPages" : 13,
  "totalElements" : 25,
  "size" : 2,
  "number" : 1,
  "sort" : [ {
    "direction" : "DESC",
    "property" : "lastModifiedAt",
    "ignoreCase" : false,
    "nullHandling": "NATIVE",
    "ascending" : false
  } ],
```

```
  "numberOfElements" : 2,
  "first" : false
}
```

- Bean のマッピング定義の追加

  上記実装例では、Member オブジェクトと MemberResource オブジェクトのコピーは、「*Bean マッピング (Dozer)*」を使って行っている。

  単純なフィールド値のコピーのみでよい場合は、Bean のマッピング定義の追加は不要だが、上記実装例では、Member オブジェクトの内容を MemberResource オブジェクトにコピーする際に、credential.password をコピー対象外にする必要がある。

  特定のフィールドをコピー対象外にするためには、Bean のマッピング定義の追加が必要となる。

```
<!-- (11) -->
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/
↪2001/XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">

    <mapping type="one-way">
        <class-a>org.terasoluna.examples.rest.domain.model.MemberCredential</
↪class-a>
        <class-b>org.terasoluna.examples.rest.api.member.
↪MemberCredentialResource</class-b>
        <!-- (12) -->
        <field-exclude>
            <a>password</a>
            <b>password</b>
        </field-exclude>
    </mapping>

</mappings>
```

項番	説明
(11)	Member オブジェクトと MemberResource オブジェクトのマッピングルールを定義するファイルを作成する。 Dozer のマッピング定義ファイルは、リソース毎に作成する事を推奨する。 今回の実装例では、/xxx-web/src/main/resources/META-INF/dozer/memberResource-mapping.xml に格納する。
(12)	上記例では、Member の関連エンティティである MemberCredential の内容を、MemberResource の関連リソースである MemberCredentialResource にコピーする際に、password フィールドをコピー対象外に指定している。 Bean マッピングの定義方法の詳細については、「*Bean マッピング (Dozer)*」を参照されたい。

- リクエスト例

```
GET /rest-api-web/api/v1/members?name=Smith&page=0&size=2 HTTP/1.1
Accept: text/plain, application/json, application/*+json, */*
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
```

- レスポンス例

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: fb63a6d446f849feb8ccaa4c9a794333
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 13 Mar 2014 11:10:43 GMT

{"content":[{"memberId":"M000000001","firstName":"John","lastName":"Smith",
↪"genderCode":"1","dateOfBirth":"2013-03-13","emailAddress":
↪"user1394709042120@test.com","telephoneNumber":"09012345678","zipCode":
↪"1710051","address":"Tokyo","credential":{"signId":"user1394709042120@test.
↪com","passwordLastChangedAt":"2014-03-13T11:10:43.066Z","lastModifiedAt":
↪"2014-03-13T11:10:43.066Z"},"createdAt":"2014-03-13T11:10:43.066Z",
↪"lastModifiedAt":"2014-03-13T11:10:43.066Z"},{"memberId":"M000000002",
↪"firstName":"Sophia","lastName":"Smith","genderCode":"2","dateOfBirth":
```

ちなみに: ページ検索が不要な場合は、Resource クラスのリストを直接扱えばよい。

Resource クラスのリストを直接扱う場合の Controller のメソッドは以下のような定義となる。

```
@RequestMapping(method = RequestMethod.GET)
@ResponseStatus(HttpStatus.OK)
public List<MemberResource> getMembers(
        @Validated MembersSearchQuery query) {
    // omitted
}
```

Resource クラスのリストを直接扱った場合、以下のような JSON となる。

```
[ {
    "memberId" : "M000000001",
    "firstName" : "John",
    "lastName" : "Smith",
    "genderCode" : "1",
    "dateOfBirth" : "1977-03-07",
    "emailAddress" : "john.smith@test.com",
    "telephoneNumber" : "09012345678",
    "zipCode" : "1710051",
    "address" : "Tokyo",
    "credential" : {
      "signId" : "john.smit@test.com",
      "passwordLastChangedAt" : "2014-03-13T10:18:08.003Z",
      "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
    },
    "createdAt" : "2014-03-13T10:18:08.003Z",
    "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
}, {
    "memberId" : "M000000002",
    "firstName" : "Sophia",
    "lastName" : "Smith",
    "genderCode" : "2",
    "dateOfBirth" : "1977-03-07",
    "emailAddress" : "sophia.smith@test.com",
    "telephoneNumber" : "09012345678",
    "zipCode" : "1710051",
    "address" : "Tokyo",
    "credential" : {
      "signId" : "sophia.smith@test.com",
      "passwordLastChangedAt" : "2014-03-13T10:18:08.003Z",
      "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
    },
```

```
    "createdAt" : "2014-03-13T10:18:08.003Z",
    "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
} ]
```

リソースをコレクションに追加する API REST の実装

指定された Member リソースを作成し、Member リソースをコレクションに追加する REST API の実装例を、以下に示す。

- REST API の実装

  指定された Member リソースを作成し、Member リソースをコレクションに追加する処理を実装する。

```
@RequestMapping("members")
@RestController
public class MemberRestController {

    // omitted

    // (1)
    @RequestMapping(method = RequestMethod.POST)
    // (2)
    @ResponseStatus(HttpStatus.CREATED)
    public MemberResource postMember(
            // (3)
            @RequestBody @Validated({ PostMembers.class, Default.class })
            MemberResource requestedResource) {

        // (4)
        Member inputMember = beanMapper.map(requestedResource, Member.class);
        Member createdMember = memberService.createMember(inputMember);

        MemberResource responseResource = beanMapper.map(createdMember,
                MemberResource.class);

        return responseResource;
    }

    // omitted

}
```

項番	説明
(1)	`@RequestMapping` アノテーションの method 属性に、`RequestMethod.POST` を指定する。
(2)	メソッドアノテーションとして、`@ResponseStatus` アノテーションを付与し、応答するステータスコードを指定する。 `@ResponseStatus` アノテーションの value 属性には、**201(Created)** を設定する。
(3)	新規に作成するリソースの情報を受け取るための JavaBean(Resource クラス) を引数に指定する。 引数アノテーションとして、`@org.springframework.web.bind.annotation.RequestBody` アノテーションを付与する。 `@RequestBody` アノテーションを付与することで、リクエスト Body に設定されている JSON や XML のデータが Resource オブジェクトに unmarshal される。 入力チェックを有効化するために、引数アノテーションとして、`@Validated` アノテーションを付与する。入力チェックの詳細については、「入力チェック」を参照されたい。
(4)	ドメイン層の Service のメソッドを呼び出し、新規にリソースを作成する。 ドメイン層の実装については、「ドメイン層の実装」を参照されたい。

- リクエスト例

```
POST /rest-api-web/api/v1/members HTTP/1.1
Accept: text/plain, application/json, application/*+json, */*
Content-Type: application/json;charset=UTF-8
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
Content-Length: 248

{"firstName":"John","lastName":"Smith","genderCode":"1","dateOfBirth":"2013-
↪03-13","emailAddress":"user1394708306056@test.com","telephoneNumber":
↪"09012345678","zipCode":"1710051","address":"Tokyo","credential":{"signId
↪":null,"password":"zaq12wsx"}}
```

- レスポンス例

```
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
X-Track: c7e9c8a9aa4f40ff87f3acdb77baccdf
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 13 Mar 2014 10:58:26 GMT

{"memberId":"M000000023","firstName":"John","lastName":"Smith","genderCode":"1
↪","dateOfBirth":"2013-03-13","emailAddress":"user1394708306056@test.com",
↪"telephoneNumber":"09012345678","zipCode":"1710051","address":"Tokyo",
↪"credential":{"signId":"user1394708306056@test.com","passwordLastChangedAt":
↪"2014-03-13T10:58:26.324Z","lastModifiedAt":"2014-03-13T10:58:26.324Z"},
↪"createdAt":"2014-03-13T10:58:26.324Z","lastModifiedAt":"2014-03-
↪13T10:58:26.324Z"}
```

指定されたリソースを取得する REST API の実装

URI で指定された Member リソースを取得する REST API の実装例を、以下に示す。

- REST API の実装

  URI で指定された Member リソースを取得する処理を実装する。

```
@RequestMapping("members")
@RestController
public class MemberRestController {
```

```
    // omitted

    // (1)
    @RequestMapping(value = "{memberId}", method = RequestMethod.GET)
    // (2)
    @ResponseStatus(HttpStatus.OK)
    public MemberResource getMember(
            // (3)
            @PathVariable("memberId") String memberId) {

        // (4)
        Member member = memberService.getMember(memberId);

        MemberResource responseResource = beanMapper.map(member,
                MemberResource.class);

        return responseResource;
    }

    // omitted

}
```

項番	説明
(1)	`@RequestMapping` アノテーションの value 属性にパス変数 (上記例では`{memberId}`) を、method 属性に `RequestMethod.GET` を指定する。 `{memberId}`には、リソースを一意に識別するための値が指定される。
(2)	メソッドアノテーションとして、`@ResponseStatus` アノテーションを付与し、応答するステータスコードを指定する。 `@ResponseStatus` アノテーションの value 属性には、200(OK) を設定する。
(3)	リソースを一意に識別するための値を、パス変数から取得する。 引数アノテーションとして、`@PathVariable("memberId")` を指定することで、パス変数 (`{memberId}`) に指定された値をメソッドの引数として受け取ることが出来る。 パス変数の詳細については、「*URL のパスから値を取得する*」を参照されたい。 上記例だと、URI が`/api/v1/members/M12345` の場合、引数の `memberId` に `M12345` が格納される。
(4)	ドメイン層の Service のメソッドを呼び出し、パス変数から取得した ID に一致するリソースの情報 (Entity など) を取得する。 ドメイン層の実装については、「ドメイン層の実装」を参照されたい。

- リクエスト例

```
GET /rest-api-web/api/v1/members/M000000003 HTTP/1.1
Accept: text/plain, application/json, application/*+json, */*
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
```

- レスポンス例

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: 275b4e7a61f946eea47672f272315ac2
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 13 Mar 2014 11:25:13 GMT

{"memberId":"M000000003","firstName":"John","lastName":"Smith","genderCode":"1
↪","dateOfBirth":"2013-03-13","emailAddress":"user1394709913496@test.com",
↪"telephoneNumber":"09012345678","zipCode":"1710051","address":"Tokyo",
↪"credential":{"signId":"user1394709913496@test.com","passwordLastChangedAt":
↪"2014-03-13T11:25:13.762Z","lastModifiedAt":"2014-03-13T11:25:13.762Z"},
↪"createdAt":"2014-03-13T11:25:13.762Z","lastModifiedAt":"2014-03-
↪13T11:25:13.762Z"}
```

指定されたリソースを更新する REST API の実装

URI で指定された Member リソースを更新する REST API の実装例を、以下に示す。

- REST API の実装

  URI で指定された Member リソースを更新する処理を実装する。

```
@RequestMapping("members")
@RestController
public class MemberRestController {

    // omitted

    // (1)
    @RequestMapping(value = "{memberId}", method = RequestMethod.PUT)
    // (2)
    @ResponseStatus(HttpStatus.OK)
```

```
    public MemberResource putMember(
            @PathVariable("memberId") String memberId,
            // (3)
            @RequestBody @Validated({ PutMember.class, Default.class })
            MemberResource requestedResource) {

        // (4)
        Member inputMember = beanMapper.map(
            requestedResource, Member.class);
        Member updatedMember = memberService.updateMember(
            memberId, inputMember);

        MemberResource responseResource = beanMapper.map(updatedMember,
                MemberResource.class);

        return responseResource;
    }

    // omitted

}
```

項番	説明
(1)	`@RequestMapping` アノテーションの value 属性にパス変数 (上記例では`{memberId}`) を、method 属性に `RequestMethod.PUT` を指定する。 `{memberId}`には、リソースを一意に識別するための値が指定される。
(2)	メソッドアノテーションとして、`@ResponseStatus` アノテーションを付与し、応答するステータスコードを指定する。 `@ResponseStatus` アノテーションの value 属性には、200(OK) を設定する。
(3)	リソースの更新内容を受け取るための JavaBean(Resource クラス) を引数に指定する。 引数アノテーションとして、`@RequestBody` アノテーションを付与することで、リクエスト Body に設定されている JSON や XML のデータが Resource オブジェクトに unmarshal される。 入力チェックを有効化するために、引数アノテーションとして、`@Validated` アノテーションを付与する。 入力チェックの詳細については、「入力チェック」を参照されたい。
(4)	ドメイン層の Service のメソッドを呼び出し、パス変数から取得した ID に一致するリソースの情報 (Entity など) を更新する。 ドメイン層の実装については、「ドメイン層の実装」を参照されたい。

- リクエスト例

```
PUT /rest-api-web/api/v1/members/M000000004 HTTP/1.1
Accept: text/plain, application/json, application/*+json, */*
Content-Type: application/json;charset=UTF-8
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
Content-Length: 221

{"memberId":"M000000004","firstName":"John","lastName":"Smith","genderCode":"1
↪","dateOfBirth":"2013-03-08","emailAddress":"user1394710559584@test.com",
↪"telephoneNumber":"09012345678","zipCode":"1710051","address":"Tokyo"}
```

- レスポンス例

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: 5e8fea3aae044e94bf20a293e155af57
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 13 Mar 2014 11:35:59 GMT

{"memberId":"M000000004","firstName":"John","lastName":"Smith","genderCode":"1
↪","dateOfBirth":"2013-03-08","emailAddress":"user1394710559584@test.com",
↪"telephoneNumber":"09012345678","zipCode":"1710051","address":"Tokyo",
↪"credential":{"signId":"user1394710559584@test.com","passwordLastChangedAt":
↪"2014-03-13T11:35:59.847Z","lastModifiedAt":"2014-03-13T11:35:59.847Z"},
↪"createdAt":"2014-03-13T11:35:59.847Z","lastModifiedAt":"2014-03-
↪13T11:36:00.122Z"}
```

指定されたリソースを削除する REST API の実装

URI で指定された Member リソースを削除する REST API の実装例を、以下に示す。

- REST API の実装

  URI で指定された Member リソースを削除する処理を実装する。

```
@RequestMapping("members")
@RestController
public class MemberRestController {

    // omitted

    // (1)
    @RequestMapping(value = "{memberId}", method = RequestMethod.DELETE)
    // (2)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteMember(
            @PathVariable("memberId") String memberId) {

        // (3)
        memberService.deleteMember(memberId);

    }

    // omitted

}
```

項番	説明
(1)	`@RequestMapping` アノテーションの value 属性にパス変数 (上記例では`{memberId}`) を、method 属性に `RequestMethod.DELETE` を指定する。
(2)	メソッドアノテーションとして、`@ResponseStatus` アノテーションを付与し、応答するステータスコードを指定する。 `@ResponseStatus` アノテーションの value 属性には、**204(NO_CONTENT)** を設定する。
(3)	ドメイン層の Service のメソッドを呼び出し、パス変数から取得した ID に一致するリソースの情報 (Entity など) を削除する。 ドメイン層の実装については、「ドメイン層の実装」を参照されたい。

注釈: 削除したリソースの情報をレスポンス BODY に設定する場合は、ステータスコードには 200(OK) を設定する。

- リクエスト例

```
DELETE /rest-api-web/api/v1/members/M000000005 HTTP/1.1
Accept: text/plain, application/json, application/*+json, */*
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
```

- レスポンス例

```
HTTP/1.1 204 No Content
Server: Apache-Coyote/1.1
X-Track: e06c5bd40c864a299c48d9be3f12b2c0
Date: Thu, 13 Mar 2014 11:40:05 GMT
```

例外のハンドリングの実装

RESTful Web Service で発生した例外のハンドリング方法について説明する。

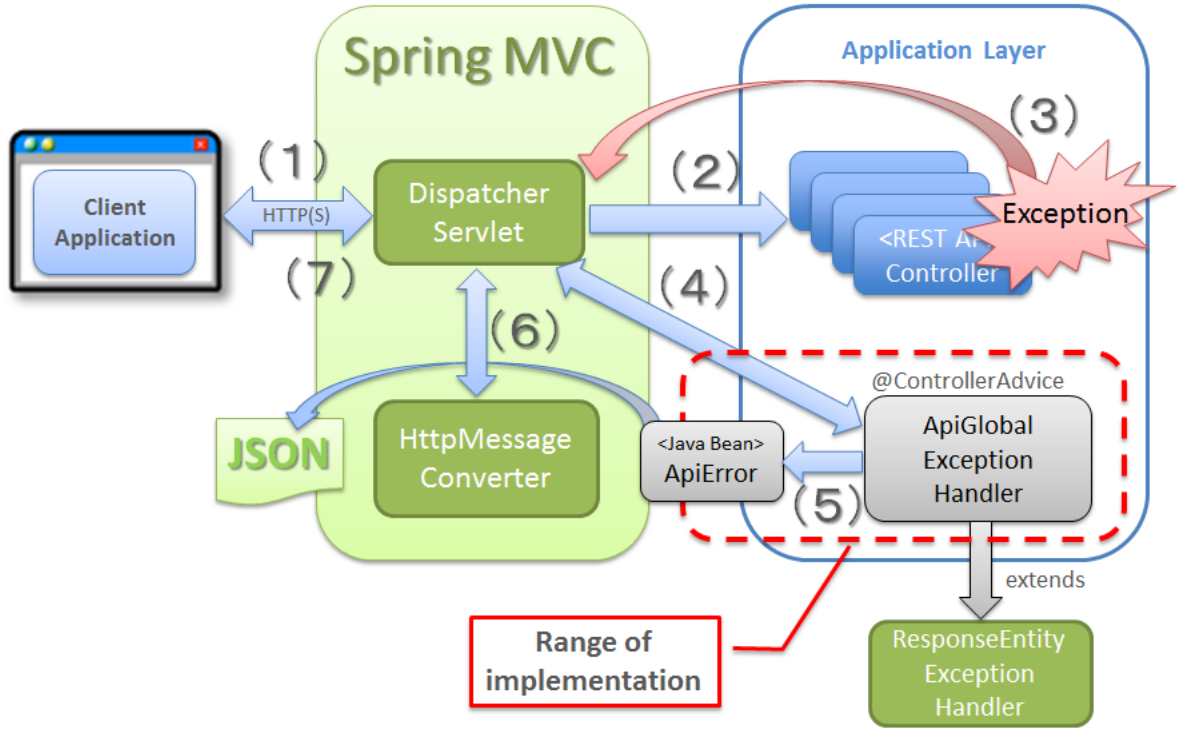
Spring MVC では、RESTful Web Service 向けの汎用的な例外ハンドリングの仕組みは用意されていない。
代わりに、RESTful Web Service 向けの例外ハンドリングの実装を補助してくれるクラスとして、(`org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler`) が提供されている。
本ガイドラインでは、Spring MVC から提供されているクラスを継承した例外ハンドリング用のクラスを作成し、作成した例外ハンドリング用のクラスに`@ControllerAdvice` アノテーションを付与する事で、例外ハンドリングを共通的に行う方法を推奨する。

`ResponseEntityExceptionHandler` では、Spring MVC のフレームワーク内で発生する例外を`@ExceptionHandler` アノテーションを使ってハンドリングするメソッドが予め実装されている。
そのため、Spring MVC のフレームワーク内で発生する例外のハンドリングを個別に実装する必要がない。
また、`ResponseEntityExceptionHandler` でハンドリングされる例外に対応する HTTP ステータスコードは、`DefaultHandlerExceptionResolver` と同様の仕様で設定される。

ハンドリングされる例外と設定される HTTP ステータスコードについては、
「*DefaultHandlerExceptionResolver で設定される HTTP レスポンスコードについて*」を参照されたい。

`ResponseEntityExceptionHandler` のデフォルトの実装ではレスポンス Body は空で返却されるが、レスポンス Body にエラー情報を出力する様に拡張する事ができる。
本ガイドラインでは、レスポンス Body に適切なエラー情報を出力する事を推奨する。

具体的な実装例を説明する前に、`ResponseEntityExceptionHandler` を継承した例外ハンドリング用のクラスを作成し、例外ハンドリングを共通的に行う際の処理フローについて説明する。
なお、個別に実装が必要になるのは、赤枠の部分となる。



項番	処理レイヤ	説明
(1) (2)	Spring MVC (Framework)	Spring MVC はクライアントからのリクエストを受け付け、REST API を呼び出す。
(3)		REST API の処理中に例外が発生する。 発生した例外は、Spring MVC によって捕捉される。
(4)		Spring MVC は、例外ハンドリング用のクラスに処理を委譲する。
(5)	Custom Exception Handler (Common Component)	例外ハンドリング用のクラスでは、エラー情報を保持するエラーオブジェクトを生成し、Spring MVC に返却する。
(6)	Spring MVC (Framework)	Spring MVC は、`HttpMessageConverter` を利用して、エラーオブジェクトを JSON 形式の電文に変換する。
(7)		Spring MVC は、JSON 形式のエラー電文をレスポンス BODY に設定し、クライアントにレスポンスする。

レスポンス Body にエラー情報を出力するための実装

- エラー情報は以下の JSON 形式とする。

```
{
  "code" : "e.ex.fw.7001",
  "message" : "Validation error occurred on item in the request body.",
  "details" : [ {
    "code" : "ExistInCodeList",
```

```
    "message" : "\"genderCode\" must exist in code list of CL_GENDER.",
    "target" : "genderCode"
  } ]
}
```

- エラー情報を保持する JavaBean を作成する。

```
package org.terasoluna.examples.rest.api.common.error;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import com.fasterxml.jackson.annotation.JsonInclude;

// (1)
public class ApiError implements Serializable {

    private static final long serialVersionUID = 1L;

    private final String code;

    private final String message;

    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private final String target; // (2)

    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private final List<ApiError> details = new ArrayList<>(); // (3)

    public ApiError(String code, String message) {
        this(code, message, null);
    }

    public ApiError(String code, String message, String target) {
        this.code = code;
        this.message = message;
        this.target = target;
    }

    public String getCode() {
        return code;
    }

    public String getMessage() {
        return message;
    }
```

```
    public String getTarget() {
        return target;
    }

    public List<ApiError> getDetails() {
        return details;
    }

    public void addDetail(ApiError detail) {
        details.add(detail);
    }

}
```

項番	説明
(1)	エラー情報を保持するためのクラスを作成する。 上記例では、エラーコード、エラーメッセージ、エラー対象、エラーの詳細情報のリストを保持するクラスとなっている。
(2)	エラーが発生した対象を識別するための値を保持するフィールド。 入力チェックでエラーが発生した場合、どの項目でエラーが発生したのかを識別できる値をクライアントに返却する事が求められるケースがある。 そのような場合は、エラーが発生した項目名を保持するフィールドが必要になる。
(3)	エラーの詳細情報のリストを保持するためのフィールド。 入力チェックでエラーが発生した場合、エラー原因が複数存在する場合があるため、すべてのエラー情報をクライアントに返却する事が求められるケースがある。 そのような場合は、エラーの詳細情報をリストで保持するフィールドが必要になる。

ちなみに: フィールドに`@JsonInclude(JsonInclude.Include.NON_EMPTY)`を指定することで、値が`null`や空の場合にJSONに項目が出力されないようにする事が出来る。項目を出力させないための条件を`null`に限定したい場合は、`@JsonInclude(JsonInclude.Include.NON_NULL)`を指定すればよい。

- エラー情報を保持する JavaBean を生成するためのクラスを作成する。

  全ての例外ハンドリングの実装が完了した際のソースコードについては、*Appendix* を参照されたい。

```
// (4)
@Component
public class ApiErrorCreator {

    @Inject
    MessageSource messageSource;

    public ApiError createApiError(WebRequest request, String errorCode,
            String defaultErrorMessage, Object... arguments) {
        // (5)
        String localizedMessage = messageSource.getMessage(errorCode,
                arguments, defaultErrorMessage, request.getLocale());
        return new ApiError(errorCode, localizedMessage);
    }

    // omitted

}
```

項番	説明
(4)	必要に応じて、エラー情報を生成するためのメソッドを提供するクラスを作成する。 このクラスの作成は必須ではないが、役割を明確に分担するために作成する事を推奨する。
(5)	エラーメッセージは、`MessageSource` より取得する。 メッセージの管理方法については、「メッセージ管理」を参照されたい。

ちなみに: 上記例では、メッセージのローカライズをサポートするために `org.springframework.web.context.request.WebRequest` を引数として受け取っている。メッセージのローカライズが必要ない場合は、`WebRequest` は不要である。

`java.util.Locale` ではなく `WebRequest` を引数にしている理由は、エラーメッセージの中に HTTP リクエストの内容を埋め込むといった要件が追加される事を考慮したためである。エラーメッセージの中に HTTP リクエストの内容を埋め込む要件がない場合は、`Locale` でもよい。

- ResponseEntityExceptionHandler のメソッドを拡張し、レスポンス Body にエラー情報を出力するための実装を行う。

  全ての例外ハンドリングの実装が完了した際のソースコードについては、*Appendix* を参照されたい。

```
@ControllerAdvice // (6)
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler
↪{

    @Inject
    ApiErrorCreator apiErrorCreator;

    @Inject
    ExceptionCodeResolver exceptionCodeResolver;

    // (7)
    @Override
    protected ResponseEntity<Object> handleExceptionInternal(Exception ex,
            Object body, HttpHeaders headers, HttpStatus status,
            WebRequest request) {
        final Object apiError;
        // (8)
        if (body == null) {
            String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
            apiError = apiErrorCreator.createApiError(request, errorCode, ex
                    .getLocalizedMessage());
        } else {
            apiError = body;
        }
        // (9)
        return ResponseEntity.status(status).headers(headers).body(apiError);
    }

    // omitted

}
```

項番	説明
(6)	Spring MVC から提供されている `ResponseEntityExceptionHandler` を継承したクラスを作成し、`@ControllerAdvice` アノテーションを付与する。
(7)	`ResponseEntityExceptionHandler` の handleExceptionInternal メソッドをオーバライドする。
(8)	レスポンス Body に出力する JavaBean の指定がない場合は、エラー情報を保持する JavaBean オブジェクトを生成する。 上記例では、共通ライブラリから提供している `ExceptionCodeResolver` を使用して、例外クラスをエラーコードへ変換している。 `ExceptionCodeResolver` の設定例については、「*ExceptionCodeResolver を使ったエラーコードとメッセージの解決*」を参照されたい。 レスポンス Body に出力する JavaBean の指定がある場合は、指定された JavaBean をそのまま使用する。 この処理は、例外毎のエラーハンドリング処理にて、個別にエラー情報が生成される事を考慮した実装となっている。
(9)	レスポンス用の HTTP Entity の Body 部分に、(8) で生成したエラー情報を設定し返却する。 返却したエラー情報は、フレームワークによって JSON に変換されレスポンスされる。 ステータスコードには、Spring MVC から提供されている `ResponseEntityExceptionHandler` によって適切な値が設定される。 設定されるステータスコードについては、「*DefaultHandlerExceptionResolver で設定される HTTP レスポンスコードについて*」を参照されたい。

ちなみに：Spring Framework 4.0 より追加された@ControllerAdvice アノテーションの属性について

`@ControllerAdvice` アノテーションの属性を指定することで、`@ControllerAdvice` が付与されたクラスで実装したメソッドを適用する Controller を柔軟に指定できるように改善されている。属性の詳細については、*@ControllerAdvice の属性*を参照されたい。

注釈: @ControllerAdvice アノテーションの属性使用時の注意点

`@ControllerAdvice` アノテーションの属性を使用することで、さまざまな粒度で例外ハンドリングを共通化することができるようになるが、アプリケーション共通の例外ハンドラクラス (上記例の `ApiGlobalExceptionHandler` クラスに相当するクラス) に対しては、`@ControllerAdvice` アノテーションの属性を指定しない方がよい。

`ApiGlobalExceptionHandler` に付与する`@ControllerAdvice` アノテーションに属性を指定した場合、Spring MVC が提供するフレームワーク処理の中で発生する一部の例外をハンドリングできないケースがある。

具体的には、リクエストに対応する REST API(Controller のハンドラメソッド) が見つからない時に発生する例外を `ApiGlobalExceptionHandler` クラスでハンドリングする事ができないため、「405 Method Not Allowed」などのエラーを正しく応答する事が出来なくなってしまう。

- レスポンス例

```
HTTP/1.1 400 Bad Request
Server: Apache-Coyote/1.1
X-Track: e60b3b6468194e22852c8bfc7618e625
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 13 Mar 2014 12:16:55 GMT
Connection: close

{"code":"e.ex.fw.7001","message":"Validation error occurred on item in the␣
↪request body.","details":[{"code":"ExistInCodeList","message":"\"genderCode\
↪" must exist in code list of CL_GENDER.","target":"genderCode"}]}
```

入力エラー例外のハンドリング実装

入力エラー（電文不正、単項目チェックエラー、相関項目チェックエラー）を応答するための実装例について説明する。

入力エラーを応答するためには、以下の 3 つの例外をハンドリングする必要がある。

項番	例外	説明
(1)	org.springframework.web.bind. MethodArgumentNotValidException	リクエスト BODY に指定された JSON や XML に対する入力チェックでエラーが発生した場合、本例外が発生する。 具体的には、リソースの POST 又は PUT 時に指定するリソースに不正な値が指定されている場合に発生する。
(2)	org.springframework.validation. BindException	リクエストパラメータ (key=value 形式のクエリ文字列) に対する入力チェックでエラーが発生した場合、本例外が発生する。 具体的には、リソースコレクションの GET 時に指定する検索条件に不正な値が指定されている場合に発生する。
(3)	org.springframework.http.converter. HttpMessageNotReadableException	JSON や XML から Resource オブジェクトを生成する際にエラーが発生した場合は、本例外が発生する。 具体的には、JSON や XML の構文不正やスキーマ定義に違反などがあった場合に発生する。

注釈: Spring Framework から提供されているアノテーションを使用してリクエストパラメータ、リクエストヘッダ、パス変数から値を取得する際に、値の型変換エラーが発生した場合、`org.springframework.beans.TypeMismatchException` が発生する。

Controller のハンドラメソッドの引数 (`String` 以外の引数) に、以下のアノテーションを指定した場合、`TypeMismatchException` が発生する可能性がある。

- `@org.springframework.web.bind.annotation.RequestParam`
- `@org.springframework.web.bind.annotation.RequestHeader`
- `@org.springframework.web.bind.annotation.Pathvariable`
- `@org.springframework.web.bind.annotation.MatrixVariable`

`TypeMismatchException` は、`ResponseEntityExceptionHandler` によって例外がハンドリングされ、400(Bad Request) となるので個別にハンドリングしなくてもよい。

エラー情報に設定するエラーコードとエラーメッセージの解決方法については、「*ExceptionCodeRe-*

solver を使ったエラーコードとメッセージの解決」を参照されたい。

- 入力チェックエラー用のエラー情報を生成するためのメソッドを作成する。

```
@Component
public class ApiErrorCreator {

    @Inject
    MessageSource messageSource;

    // omitted

    // (1)
    public ApiError createBindingResultApiError(WebRequest request,
            String errorCode, BindingResult bindingResult,
            String defaultErrorMessage) {
        ApiError apiError = createApiError(request, errorCode,
                defaultErrorMessage);
        for (FieldError fieldError : bindingResult.getFieldErrors()) {
            apiError.addDetail(createApiError(request, fieldError, fieldError
                    .getField()));
        }
        for (ObjectError objectError : bindingResult.getGlobalErrors()) {
            apiError.addDetail(createApiError(request, objectError,␣
↪objectError
                    .getObjectName()));
        }
        return apiError;
    }

    // (2)
    private ApiError createApiError(WebRequest request,
            DefaultMessageSourceResolvable messageResolvable, String target) {
        String localizedMessage = messageSource.getMessage(messageResolvable,
                request.getLocale());
        return new ApiError(messageResolvable.getCode(), localizedMessage,␣
↪target);
    }

    // omitted

}
```

項番	説明
(1)	入力チェック用のエラー情報を生成するためのメソッドを作成する。 上記例では、単項目チェックエラー (FieldError) と相関項目チェックエラー (ObjectError) を、エラーの詳細情報に追加している。 項目毎のエラー情報を出力する必要がない場合は、本メソッドを用意する必要はない。
(2)	単項目チェックエラー (FieldError) と相関項目チェックエラー (ObjectError) で同じ処理を実装する事になるので、共通メソッドとして本メソッドを作成している。

- ResponseEntityExceptionHandler のメソッドを拡張し、レスポンス Body に入力チェック用のエラー情報を出力するための実装を行う。

```
@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler
↪{

    @Inject
    ApiErrorCreator apiErrorCreator;

    @Inject
    ExceptionCodeResolver exceptionCodeResolver;

    // omitted

    // (3)
    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        return handleBindingResult(ex, ex.getBindingResult(), headers, status,
                request);
    }

    // (4)
    @Override
    protected ResponseEntity<Object> handleBindException(BindException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        return handleBindingResult(ex, ex.getBindingResult(), headers, status,
                request);
    }
```

```
    // (5)
    @Override
    protected ResponseEntity<Object> handleHttpMessageNotReadable(
            HttpMessageNotReadableException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        if (ex.getCause() instanceof Exception) {
            return handleExceptionInternal((Exception) ex.getCause(), null,
                    headers, status, request);
        } else {
            return handleExceptionInternal(ex, null, headers, status,␣
↪request);
        }
    }

    // omitted

    // (6)
    protected ResponseEntity<Object> handleBindingResult(Exception ex,
            BindingResult bindingResult, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        String code = exceptionCodeResolver.resolveExceptionCode(ex);
        String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
        ApiError apiError = apiErrorCreator.createBindingResultApiError(
                request, errorCode, bindingResult, ex.getMessage());
        return handleExceptionInternal(ex, apiError, headers, status,␣
↪request);
    }

    // omitted

}
```

項番	説明
(3)	`ResponseEntityExceptionHandler` の handleMethodArgumentNotValid メソッドをオーバライドし、`MethodArgumentNotValidException` のエラーハンドリングを拡張する。 上記例では、入力チェックエラーをハンドリングするための共通メソッド (6) に処理を委譲している。 項目毎のエラー情報を出力する必要がない場合は、オーバライドする必要はない。 ステータスコードには **400(Bad Request)** が設定され、指定されたリソースの項目値に不備がある事を通知する。
(4)	`ResponseEntityExceptionHandler` の handleBindException メソッドをオーバライドし、`BindException` のエラーハンドリングを拡張する。 上記例では、入力チェックエラーをハンドリングするための共通メソッド (6) に処理を委譲している。 項目毎のエラー情報を出力する必要がない場合は、オーバライドする必要はない。 ステータスコードには **400(Bad Request)** が設定され、指定されたリクエストパラメータに不備がある事を通知する。
(5)	`ResponseEntityExceptionHandler` の handleHttpMessageNotReadable メソッドをオーバライドし、`HttpMessageNotReadableException` のエラーハンドリングを拡張する。 上記例では、細かくエラーハンドリングを行うために、原因例外を使ってエラーハンドリングしている。 細かくエラーハンドリングをしなくてもよい場合は、オーバライドする必要はない。 ステータスコードには **400(Bad Request)** が設定され、指定されたリソースのフォーマットなどに不備がある事を通知する。
(6)	入力チェックエラーのエラー情報を保持する JavaBean オブジェクトを生成する。 上記例では、handleMethodArgumentNotValid と handleBindException で同じ処理を実装する事になるので、共通メソッドとして本メソッドを作成している。

ちなみに：JSON 使用時のエラーハンドリングについて

リソースのフォーマットとして JSON を使用する場合、以下の例外が `HttpMessageNotReadableException` の原因例外として格納される。

項番	例外クラス	説明
(1)	com.fasterxml.jackson.core. JsonParseException	JSON として不正な構文が含まれる場合に発生する。
(2)	com.fasterxml.jackson.databind.exc. UnrecognizedPropertyException	Resource オブジェクトに存在しないフィールドが JSON に指定されている場合に発生する。
(3)	com.fasterxml.jackson.databind. JsonMappingException	JSON から Resource オブジェクトへ変換する際に、値の型変換エラーが発生した場合に発生する。

- 入力チェックエラー (単項目チェック、相関項目チェックエラー) が発生した場合、以下のようなエラー応答が行われる。

```
HTTP/1.1 400 Bad Request
Server: Apache-Coyote/1.1
X-Track: 13522b3badf2432ba4cad0dc7aeaee80
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 05:08:28 GMT
Connection: close

{"code":"e.ex.fw.7002","message":"Validation error occurred on item in the␣
↪request parameters.","details":[{"code":"NotEmpty","message":"\"{0}\" may␣
↪not be empty.","target":"name"}]}
```

- JSON エラー (フォーマットエラーなど) が発生した場合、以下のようなエラー応答が行われる。

```
HTTP/1.1 400 Bad Request
Server: Apache-Coyote/1.1
X-Track: ca4c742a6bfd49e5bc01cd0b124738a1
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 13:32:24 GMT
Connection: close

{"code":"e.ex.fw.7003","message":"Request body format error occurred."}
```

リソース未検出エラー例外のハンドリング実装

リソースが存在しない場合に、リソース未検出エラーを応答するための実装例について説明する。

パス変数から取得した ID に一致するリソースが見つからない場合は、リソースが見つからない事を通知する例外を発生させる。
リソースが見つからなかった事を通知する例外として、共通ライブラリより `org.terasoluna.gfw.common.exception.ResourceNotFoundException` を用意している。
以下に実装例を示す。

- パス変数から取得した ID に一致するリソースが見つからない場合は、`ResourceNotFoundException` を発生させる。

```
public Member getMember(String memberId) {
    Member member = memberRepository.findOne(memberId);
    if (member == null) {
        throw new ResourceNotFoundException(ResultMessages.error().add(
                "e.ex.mm.5001", memberId));
    }
    return member;
}
```

- ResultMessages 用のエラー情報を生成するためのメソッドを作成する。

```
@Component
public class ApiErrorCreator {

    // omitted

    // (1)
    public ApiError createResultMessagesApiError(WebRequest request,
            String rootErrorCode, ResultMessages resultMessages,
            String defaultErrorMessage) {
        ApiError apiError;
        if (resultMessages.getList().size() == 1) {
            ResultMessage resultMessage = resultMessages.iterator().next();
            String errorCode = resultMessage.getCode();
            String errorText = resultMessage.getText();
            if (errorCode == null && errorText == null) {
                errorCode = rootErrorCode;
            }
            apiError = createApiError(request, errorCode, errorText,
                    resultMessage.getArgs());
        } else {
            apiError = createApiError(request, rootErrorCode,
                    defaultErrorMessage);
            for (ResultMessage resultMessage : resultMessages.getList()) {
                apiError.addDetail(createApiError(request, resultMessage
                        .getCode(), resultMessage.getText(), resultMessage
                        .getArgs()));
            }
        }
        return apiError;
    }

    // omitted

}
```

項番	説明
(1)	処理結果からエラー情報を生成するためのメソッドを作成する。 上記例では、ResultMessages が保持しているメッセージ情報を、エラー情報に設定している。

注釈: 上記例では、ResultMessages が複数のメッセージを保持する事ができるため、格納されているメッセージが 1 件の時と複数件の時で処理をわけている。

複数件のメッセージをサポートする必要がない場合は、先頭の 1 件をエラー情報として生成する処理にすればよい。

- エラーハンドリングを行うクラスに、リソースが見つからない事を通知する例外をハンドリングするためのメソッドを作成する。

```
@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler
↪{

    @Inject
    ApiErrorCreator apiErrorCreator;

    @Inject
    ExceptionCodeResolver exceptionCodeResolver;

    // omitted

    // (2)
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Object> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {
        return handleResultMessagesNotificationException(ex, new␣
↪HttpHeaders(),
                HttpStatus.NOT_FOUND, request);
    }

    // omitted

    // (3)
    private ResponseEntity<Object> handleResultMessagesNotificationException(
            ResultMessagesNotificationException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
        ApiError apiError = apiErrorCreator.createResultMessagesApiError(
                request, errorCode, ex.getResultMessages(), ex.getMessage());
        return handleExceptionInternal(ex, apiError, headers, status,␣
↪request);
    }

    // omitted

}
```

項番	説明
(2)	`ResourceNotFoundException` をハンドリングするためのメソッドを追加する。 メソッドアノテーションとして`@ExceptionHandler(ResourceNotFoundException.class)` を指定すると、`ResourceNotFoundException` の例外をハンドリングする事ができる。 上記例では、`ResourceNotFoundException` の親クラス(`ResultMessagesNotificationException`) の例外をハンドリングするメソッドに処理を委譲している。 ステータスコードには **404(Not Found)** を設定し、指定されたリソースがサーバに存在しない事を通知する。
(3)	リソース未検出エラー及び業務エラーのエラー情報を保持する JavaBean オブジェクトを生成する。 上記例では、以降で説明する業務エラーのハンドリング処理と同じ処理となるので、共通メソッドとして本メソッドを作成している。

- リソースが見つからない場合、以下のようなエラー応答が行われる。

```
HTTP/1.1 404 Not Found
Server: Apache-Coyote/1.1
X-Track: 5ee563877f3140fd904d0acf52eba398
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 08:46:18 GMT

{"code":"e.ex.mm.5001","message":"Specified member not found. member id :␣
↪M000000001"}
```

業務エラー例外のハンドリング実装

ビジネスルールの違反を検知した場合に、業務エラーを応答するための実装例について説明する。

ビジネスルールのチェックは Service の処理として行い、ビジネスルールの違反を検知した場合は、業務例外を発生させる。業務エラーの検知方法については、「業務エラーを通知する」を参照されたい。

- エラーハンドリングを行うクラスに、業務例外をハンドリングするためのメソッドを作成する。

```
@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler
↪{

    // omitted

    // (1)
    @ExceptionHandler(BusinessException.class)
    public ResponseEntity<Object> handleBusinessException(BusinessException␣
↪ex,
            WebRequest request) {
        return handleResultMessagesNotificationException(ex, new␣
↪HttpHeaders(),
                HttpStatus.CONFLICT, request);
    }

    // omitted

}
```

項番	説明
(1)	BusinessException をハンドリングするためのメソッドを追加する。 メソッドアノテーションとして@ExceptionHandler(BusinessException.class) を指定すると、BusinessException の例外をハンドリングする事ができる。 上記例では、BusinessException の親クラス (ResultMessagesNotificationException) の例外をハンドリングするメソッドに処理を委譲している。 ステータスコードには **409(Conflict)** を設定し、クライアントから指定されたリソース自体には不備はないが、サーバで保持しているリソースを操作するための条件が全て整っていない事を通知する。

- 業務エラーが発生した場合、以下のようなエラー応答が行われる。

```
HTTP/1.1 409 Conflict
Server: Apache-Coyote/1.1
X-Track: 37c1a899d5f74e7a9c24662292837ef7
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 09:03:26 GMT

{"code":"e.ex.mm.8001","message":"Cannot use specified sign id. sign id :␣
↪user1@test.com"}
```

排他エラー例外のハンドリング実装

排他エラーが発生した場合に、排他エラーを応答するための実装例について説明する。

排他制御を行う場合は、排他エラーのハンドリングが必要となる。

排他制御の詳細については、「排他制御」を参照されたい。

- エラーハンドリングを行うクラスに、排他エラーをハンドリングするためのメソッドを作成する。

```
@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler
↪{

    // omitted

    // (1)
    @ExceptionHandler({ OptimisticLockingFailureException.class,
            PessimisticLockingFailureException.class })
    public ResponseEntity<Object> handleLockingFailureException(Exception ex,
            WebRequest request) {
        return handleExceptionInternal(ex, null, new HttpHeaders(),
                HttpStatus.CONFLICT, request);
    }

    // omitted

}
```

項番	説明
(1)	排他エラー (OptimisticLockingFailureException と PessimisticLockingFailureException) をハンドリングするためのメソッドを追加する。 メソッドアノテーションとして@ExceptionHandler({ OptimisticLockingFailureException.class, PessimisticLockingFailureException.class }) を指定すると、排他エラー (OptimisticLockingFailureException と PessimisticLockingFailureException) の例外をハンドリングする事ができる。 ステータスコードには **409(Conflict)** を設定し、クライアントから指定されたリソース自体には不備はないが、処理が競合したためリソースを操作するための条件を満たすことが出来なかった事を通知する。

- 排他エラーが発生した場合、以下のようなエラー応答が行われる。

```
HTTP/1.1 409 Conflict
Server: Apache-Coyote/1.1
X-Track: 85200b5a51be42b29840e482ee35087f
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 16:32:45 GMT

{"code":"e.ex.fw.8002","message":"Conflict with other processing occurred."}
```

システムエラー例外のハンドリング実装

システム異常を検知した場合に、システムエラーを応答するための実装例について説明する。

システム異常の検知した場合は、システム例外を発生させる。システムエラーの検知方法については、「システムエラーを通知する」を参照されたい。

- エラーハンドリングを行うクラスに、システム例外をハンドリングするためのメソッドを作成する。

```
@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler
↪{

    // omitted

    // (1)
    @ExceptionHandler(Exception.class)
    public ResponseEntity<Object> handleSystemError(Exception ex,
            WebRequest request) {
        return handleExceptionInternal(ex, null, new HttpHeaders(),
                HttpStatus.INTERNAL_SERVER_ERROR, request);
    }

    // omitted

}
```

項番	説明
(1)	Exception をハンドリングするためのメソッドを追加する。 メソッドアノテーションとして@ExceptionHandler(Exception.class) を指定すると、Exception の例外をハンドリングする事ができる。 上記例では、使用している依存ライブラリから発生するシステム例外もハンドリング対象としている。 ステータスコードには **500(Internal Server Error)** を設定する。

- システムエラーが発生した場合、以下のようなエラー応答が行われる。

```
HTTP/1.1 500 Internal Server Error
Server: Apache-Coyote/1.1
X-Track: 3625d5a040a744e49b0a9b3763a24e9c
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 12:22:33 GMT
Connection: close

{"code":"e.ex.fw.9003","message":"System error occurred."}
```

> **警告: システムエラー時のエラーメッセージについて**
>
> システムエラーが発生した場合、クライアントへ返却するメッセージは、エラー原因が特定されないシンプルなエラーメッセージを設定することを推奨する。エラー原因が特定できるメッセージを設定してしまうと、システムの脆弱性をクライアントに公開する可能性があり、セキュリティー上問題がある。
>
> エラー原因は、エラー解析用にログに出力する。Blank プロジェクトのデフォルトの設定では、共通ライブラリから提供している ExceptionLogger によってログが出力されるようなっているため、ログを出力するための設定や実装は不要である。

ExceptionCodeResolver を使ったエラーコードとメッセージの解決

共通ライブラリより提供している ExceptionCodeResolver を使用すると、例外クラスからエラーコードを解決する事ができる。

特に、エラー原因がクライアント側にある場合は、エラー原因に応じたエラーメッセージを設定する事が求められるケースがあるため、そのような場合に便利な機能である。

- applicationContext.xml

  例外クラスとエラーコード (例外コード) のマッピングを行う。

```
<!-- omitted -->

<bean id="exceptionCodeResolver"
    class="org.terasoluna.gfw.common.exception.
↪SimpleMappingExceptionCodeResolver">
    <property name="exceptionMappings">
        <map>
            <!-- omitted -->
            <entry key="ResourceNotFoundException"              value="e.ex.
↪fw.5001" />
            <entry key="HttpRequestMethodNotSupportedException" value="e.ex.
↪fw.6001" />
            <entry key="MediaTypeNotAcceptableException"        value="e.ex.
↪fw.6002" />
            <entry key="HttpMediaTypeNotSupportedException"     value="e.ex.
↪fw.6003" />
            <entry key="MethodArgumentNotValidException"        value="e.ex.
↪fw.7001" />
            <entry key="BindException"                          value="e.ex.
↪fw.7002" />
            <entry key="JsonParseException"                     value="e.ex.
↪fw.7003" />
```

```
            <entry key="UnrecognizedPropertyException"          value="e.ex.
↪fw.7004" />
            <entry key="JsonMappingException"                   value="e.ex.
↪fw.7005" />
            <entry key="TypeMismatchException"                  value="e.ex.
↪fw.7006" />
            <entry key="BusinessException"                      value="e.ex.
↪fw.8001" />
            <entry key="OptimisticLockingFailureException"      value="e.ex.
↪fw.8002" />
            <entry key="PessimisticLockingFailureException"     value="e.ex.
↪fw.8002" />
            <entry key="DataAccessException"                    value="e.ex.
↪fw.9002" />
            <!-- omitted -->
        </map>
    </property>
    <property name="defaultExceptionCode" value="e.ex.fw.9001" />
</bean>

<!-- omitted -->
```

エラーコードに対応するメッセージの設定例を以下に示す。

メッセージの管理方法については、「メッセージ管理」を参照されたい。

- xxx-web/src/main/resources/i18n/application-messages.properties

  アプリケーション層で発生するエラーに対して、エラーコード (例外コード) に対応するメッセージの設定を行う。

```
# ---
# Application common messages
# ---
e.ex.fw.5001 = Resource not found.

e.ex.fw.6001 = Request method not supported.
e.ex.fw.6002 = Specified representation format not supported.
e.ex.fw.6003 = Specified media type in the request body not supported.

e.ex.fw.7001 = Validation error occurred on item in the request body.
e.ex.fw.7002 = Validation error occurred on item in the request parameters.
e.ex.fw.7003 = Request body format error occurred.
e.ex.fw.7004 = Unknown field exists in JSON.
e.ex.fw.7005 = Type mismatch error occurred in JSON field.
e.ex.fw.7006 = Type mismatch error occurred in request parameter or header or␣
↪path variable.
```

```
e.ex.fw.8001 = Business error occurred.
e.ex.fw.8002 = Conflict with other processing occurred.

e.ex.fw.9001 = System error occurred.
e.ex.fw.9002 = System error occurred.
e.ex.fw.9003 = System error occurred.

# omitted
```

- `xxx-web/src/main/resources/ValidationMessages.properties`

  Bean Validation を使った入力チェックで発生するエラーに対して、エラーコードに対応するメッセージの設定を行う。

  ここでは、Hibernate Validator が用意するデフォルトメッセージを利用する。

  デフォルトメッセージは、メッセージの中に項目名が含まれないため、`{0}`（フィールド名）を追加している。

```
# ---
# Bean Validation common messages
# ---

# for bean validation of standard
javax.validation.constraints.AssertFalse.message = "{0}" must be false.
javax.validation.constraints.AssertTrue.message  = "{0}" must be true.
javax.validation.constraints.DecimalMax.message  = "{0}" must be less than $
↪{inclusive == true ? 'or equal to ' : ''}{value}.
javax.validation.constraints.DecimalMin.message  = "{0}" must be greater than
↪${inclusive == true ? 'or equal to ' : ''}{value}.
javax.validation.constraints.Digits.message      = "{0}" numeric value out of␣
↪bounds (<{integer} digits>.<{fraction} digits> expected).
javax.validation.constraints.Future.message      = "{0}" must be in the␣
↪future.
javax.validation.constraints.Max.message         = "{0}" must be less than or␣
↪equal to {value}.
javax.validation.constraints.Min.message         = "{0}" must be greater than␣
↪or equal to {value}.
javax.validation.constraints.NotNull.message     = "{0}" may not be null.
javax.validation.constraints.Null.message        = "{0}" must be null.
javax.validation.constraints.Past.message        = "{0}" must be in the past.
javax.validation.constraints.Pattern.message     = "{0}" must match "{regexp}
↪".
javax.validation.constraints.Size.message        = "{0}" size must be between
↪{min} and {max}.

# for bean validation of hibernate
org.hibernate.validator.constraints.CreditCardNumber.message        = "{0}"␣
↪invalid credit card number.
org.hibernate.validator.constraints.EAN.message                     = "{0}"␣
↪invalid {type} barcode.
org.hibernate.validator.constraints.Email.message                   = "{0}"␣
↪not a well-formed email address.
```

```
org.hibernate.validator.constraints.Length.message                  = "{0}"␣
↪length must be between {min} and {max}.
org.hibernate.validator.constraints.LuhnCheck.message               = "{0}"␣
↪The check digit for ${validatedValue} is invalid, Luhn Modulo 10 checksum␣
↪failed.
org.hibernate.validator.constraints.Mod10Check.message              = "{0}"␣
↪The check digit for ${validatedValue} is invalid, Modulo 10 checksum failed.
org.hibernate.validator.constraints.Mod11Check.message              = "{0}"␣
↪The check digit for ${validatedValue} is invalid, Modulo 11 checksum failed.
org.hibernate.validator.constraints.ModCheck.message                = "{0}"
 The check digit for ${validatedValue} is invalid, ${modType} checksum failed.
org.hibernate.validator.constraints.NotBlank.message                = "{0}"␣
↪may not be empty.
org.hibernate.validator.constraints.NotEmpty.message                = "{0}"␣
↪may not be empty.
org.hibernate.validator.constraints.ParametersScriptAssert.message  = "{0}"␣
↪script expression "{script}" didn't evaluate to true.
org.hibernate.validator.constraints.Range.message                   = "{0}"␣
↪must be between {min} and {max}.
org.hibernate.validator.constraints.SafeHtml.message                = "{0}"␣
↪may have unsafe html content.
org.hibernate.validator.constraints.ScriptAssert.message            = "{0}"␣
↪script expression "{script}" didn't evaluate to true.
org.hibernate.validator.constraints.URL.message                     = "{0}"␣
↪must be a valid URL.

org.hibernate.validator.constraints.br.CNPJ.message                 = "{0}"␣
↪invalid Brazilian corporate taxpayer registry number (CNPJ).
org.hibernate.validator.constraints.br.CPF.message                  = "{0}"␣
↪invalid Brazilian individual taxpayer registry number (CPF).
org.hibernate.validator.constraints.br.TituloEleitoral.message      = "{0}"␣
↪invalid Brazilian Voter ID card number.

# for common library
org.terasoluna.gfw.common.codelist.ExistInCodeList.message   = "{0}" must␣
↪exist in code list of {codeListId}.
```

- xxx-domain/src/main/resources/i18n/domain-messages.properties

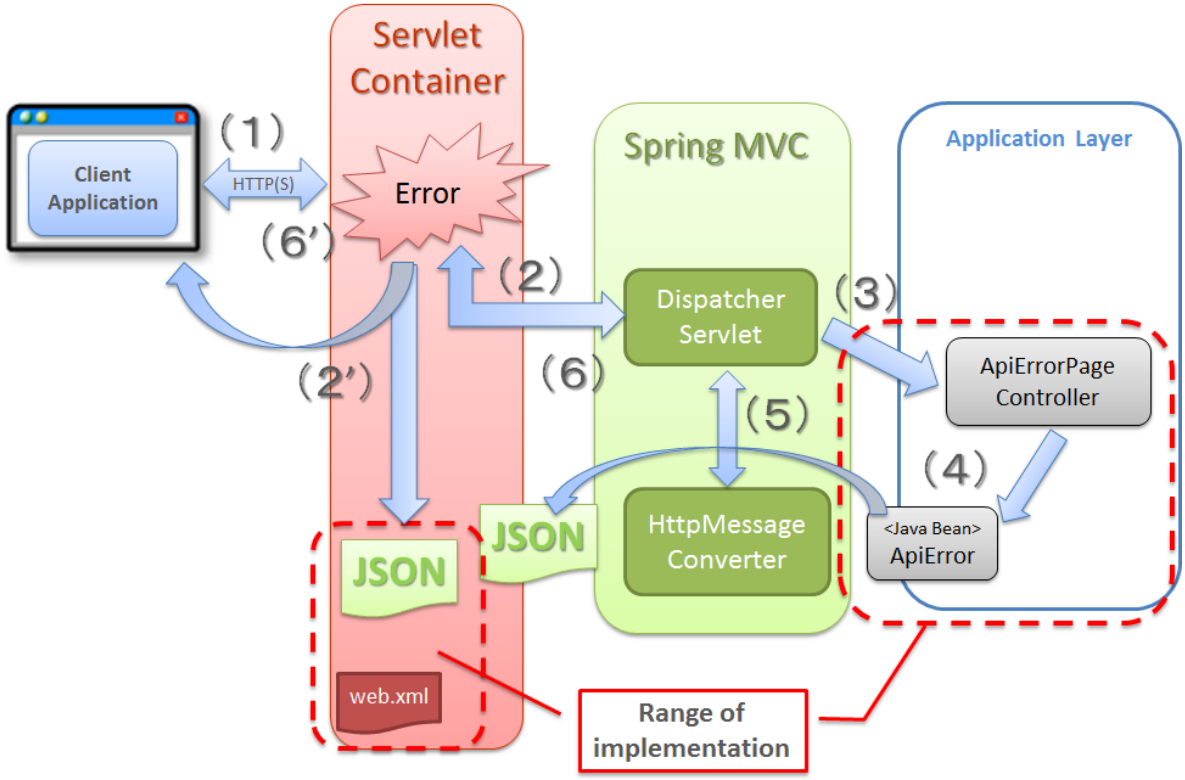
  ドメイン層で発生するエラーに対して、エラーコード (例外コード) に対応するメッセージの設定を行う。

```
# omitted

e.ex.mm.5001 = Specified member not found. member id : {0}
e.ex.mm.8001 = Cannot use specified sign id. sign id : {0}

# omitted
```

サーブレットコンテナに通知されたエラーのハンドリング実装

Filter でエラーが発生した場合や `HttpServletResponse#sendError` を使ってエラーレスポンスが行われた場合は、Spring MVC の例外ハンドリングの仕組みを使ってハンドリングできないため、これらのエラーはサーブレットコンテナに通知される。

本節では、サーブレットコンテナに通知されたエラーをハンドリングする方法について説明する。



項番	処理レイヤ	説明
(1)	Servlet Container (AP Server)	Servlet Container はクライアントからのリクエストを受け付け、処理を行う。 Servlet Container は処理中にエラーを検知する。
(2)		Servlet Container は `web.xml` の error-page の定義に従って、エラー処理を行う。 致命的なエラーでない場合は、エラーハンドリングを行う Controller を呼び出し、エラー処理を行う。
(2』)		致命的なエラーの場合は、予め用意してある静的な JSON ファイルを取得し、クライアントへ応答する。
(3)	Spring MVC (Framework)	Spring MVC は、エラーハンドリングを行う Controller を呼び出す。
(4)	Controller (Common Component)	エラーハンドリングを行う Controller では、エラー情報を保持するエラーオブジェクトを生成し、Spring MVC に返却する。
(5)	Spring MVC (Framework)	Spring MVC は、`HttpMessageConverter` を利用して、エラーオブジェクトを JSON 形式の電文に変換する。
(6)		Spring MVC は、JSON 形式のエラー電文をレスポンス BODY に設定し、クライアントにレスポンスする。

エラー応答を行うための **Controller** の実装

サーブレットコンテナに通知されたエラーのエラー応答を行う Controller を作成する。

```
package org.terasoluna.examples.rest.api.common.error;

import java.util.HashMap;
import java.util.Map;

import javax.inject.Inject;
import javax.servlet.RequestDispatcher;

import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.context.request.RequestAttributes;
import org.springframework.web.context.request.WebRequest;

// (1)
@RequestMapping("error")
@RestController
public class ApiErrorPageController {

    @Inject
    ApiErrorCreator apiErrorCreator; // (2)

    // (3)
    private final Map<HttpStatus, String> errorCodeMap = new HashMap
↪<HttpStatus, String>();

    // (4)
    public ApiErrorPageController() {
        errorCodeMap.put(HttpStatus.NOT_FOUND, "e.ex.fw.5001");
    }

    // (5)
    @RequestMapping
    public ResponseEntity<ApiError> handleErrorPage(WebRequest request) {
        // (6)
        HttpStatus httpStatus = HttpStatus.valueOf((Integer) request
                .getAttribute(RequestDispatcher.ERROR_STATUS_CODE,
                        RequestAttributes.SCOPE_REQUEST));
        // (7)
        String errorCode = errorCodeMap.get(httpStatus);
        // (8)
        ApiError apiError = apiErrorCreator.createApiError(request, errorCode,
                httpStatus.getReasonPhrase());
        // (9)
        return ResponseEntity.status(httpStatus).body(apiError);
    }
```

```
}
```

項番	説明
(1)	エラー応答を行うための Controller クラスを作成する。 上記例では、「/api/v1/error」というサーブレットパスにマッピングしている。
(2)	エラー情報を作成するクラスを Inject する。
(3)	HTTP ステータスコードとエラーコードをマッピングするための Map を作成する。
(4)	HTTP ステータスコードとエラーコードとのマッピングを登録する。
(5)	エラー応答を行うハンドラメソッドを作成する。 上記例では、レスポンスコード (<error-code>) を使ってエラーページのハンドリングを行うケースのみを考慮した実装になっている。 例外タイプ (<exception-type>) を使ってエラーページのハンドリングを行う場合は、別途考慮が必要である。
(6)	リクエストスコープに格納されているステータスコードを取得する。
(7)	取得したステータスコードに対応するエラーコードを取得する。
(8)	取得したエラーコードに対応するエラー情報を生成する。
(9)	(8) で生成したエラー情報を応答する。

致命的なエラーが発生した際に応答する静的な JSON ファイルの作成

致命的なエラーが発生した際に応答する静的な JSON ファイルを作成する。

- unhandledSystemError.json

```
{"code":"e.ex.fw.9999","message":"Unhandled system error occurred."}
```

サーブレットコンテナに通知されたエラーをハンドリングするための設定

ここでは、サーブレットコンテナに通知されたエラーをハンドリングするための設定について説明する。

- web.xml

```
<!-- omitted -->

<!-- (1) -->
<error-page>
    <error-code>404</error-code>
    <location>/api/v1/error</location>
</error-page>

<!-- (2) -->
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/WEB-INF/views/common/error/unhandledSystemError.json</location>
</error-page>

<!-- (3) -->
<mime-mapping>
    <extension>json</extension>
    <mime-type>application/json;charset=UTF-8</mime-type>
</mime-mapping>

<!-- omitted -->
```

項番	説明
(1)	必要に応じてレスポンスコードによるエラーページの定義を追加する。 上記例では、`404 Not Found` が発生した際に、「`/api/v1/error`」というリクエストにマッピングされている Controller(`ApiErrorPageController`) を呼び出してエラー応答を行っている。
(2)	致命的なエラーをハンドリングするための定義を追加する。 致命的なエラーが発生していた場合、レスポンス情報を作成する処理で二重障害が発生する可能性があるため、予め用意している静的な JSON を応答する事を推奨する。 上記例では、「`/WEB-INF/views/common/error/unhandledSystemError.json`」に定義されている固定の JSON を応答している。
(3)	json の MIME タイプを指定する。 (2) で作成する JSON ファイルの中にマルチバイト文字を含める場合は、`charset=UTF-8` を指定しないと、クライアント側で文字化けする可能性がある。 JSON ファイルにマルチバイト文字を含めない場合は、この設定は必須ではないが、設定しておいた方が無難である。

注釈: Servlet の仕様では、`<error-page>`の`<location>`にクエリパラメータを付与したパスを指定した場合の挙動について、定義していない。そのため、AP サーバによって挙動が異なる可能性がある。よって、クエリパラメータを使用してエラー時の遷移先に情報を渡すことは推奨しない。

- 存在しないパスへリクエストを送った場合、以下のようなエラー応答が行われる。

```
HTTP/1.1 404 Not Found
Server: Apache-Coyote/1.1
X-Track: 2ad50fb5ba2441699c91a5b01edef83f
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 23:24:20 GMT

{"code":"e.ex.fw.5001","message":"Resource not found."}
```

- 致命的なエラーが発生した場合、以下のようなエラー応答が行われる。

```
HTTP/1.1 500 Internal Server Error
Server: Apache-Coyote/1.1
X-Track: 69db3854a19f439781584321d9ce8336
Content-Type: application/json
Content-Length: 68
Date: Thu, 20 Feb 2014 00:13:43 GMT
Connection: close

{"code":"e.ex.fw.9999","message":"Unhandled system error occurred."}
```

セキュリティ対策

RESTful Web Service に対するセキュリティ対策の実現方法について説明する。

本ガイドラインでは、セキュリティ対策の実現方法として、Spring Security を使用する事を推奨している。

認証・認可

- OAuth2(Spring Security OAuth2) を使用して認証・認可を実現する方法について、*OAuth* を参照されたい。

CSRF 対策

- RESTful Web Service に対して CSRF 対策を行う場合の設定方法については、*CSRF 対策*を参照されたい。
- RESTful Web Service に対して CSRF 対策を行わない場合の設定方法については、*CSRF 対策の無効化*を参照されたい。

リソースの条件付き操作

課題: TBD

Etag などのヘッダを使った条件付き処理の制御の実現方法について、次版以降に記載する予定である。

リソースのキャッシュ制御

課題: TBD

Cache-Control/Expires/Pragma などのヘッダを使ったキャッシュ制御の実現方法について、次版以降に記載する予定である。

5.1.5 How to extend

@JsonView を使用したレスポンスの出力制御

`@JsonView` を使用することによって、Resource オブジェクト内のプロパティーをグループ分けすることができる。

この機能は Spring Framework が Jackson の機能をサポートすることにより実現している。

詳細は、JacksonJsonViews を参照されたい。

Controller にてグループを指定することで、指定したグループに所属するプロパティーのみ出力することができる。

1 つのプロパティーは、複数のグループに所属することも可能である。

以下は、Member リソースを「概要」と「詳細」の 2 つのフォーマットで扱う際の実装例である。

「概要フォーマット」は Member リソースの主要項目を、「詳細フォーマット」は Member リソースの全項目を出力する。

- `MemberResource.java`

```
package org.terasoluna.examples.rest.api.member;

import java.io.Serializable;
```

```
import org.joda.time.DateTime;
import org.joda.time.LocalDate;

import com.fasterxml.jackson.annotation.JsonView;

public class MemberResource implements Serializable {

    private static final long serialVersionUID = 1L;

    // (1)
    interface Summary {
    }

    // (2)
    interface Detail {
    }

    // (3)
    @JsonView({Summary.class, Detail.class})
    private String memberId;

    @JsonView({Summary.class, Detail.class})
    private String firstName;

    @JsonView({Summary.class, Detail.class})
    private String lastName;

    // (4)
    @JsonView(Detail.class)
    private String genderCode;

    @JsonView(Detail.class)
    private LocalDate dateOfBirth;

    @JsonView(Detail.class)
    private String emailAddress;

    @JsonView(Detail.class)
    private String telephoneNumber;

    @JsonView(Detail.class)
    private String zipCode;

    @JsonView(Detail.class)
    private String address;

    // (5)
    private DateTime createdAt;

    private DateTime lastModifiedAt;

    // omitted setter and getter
```

```
}
```

項番	説明
(1)	出力制御するグループを指定するためのマーカーインターフェースを定義している。 上記例では、概要出力時に指定するグループを定義している。
(2)	出力制御するグループを指定するためのマーカーインターフェースを定義している。 上記例では、詳細出力時に指定するグループを定義している。
(3)	複数のグループで出力したい項目には、引数を配列にして複数のマーカーインターフェースを渡すことで、複数のグループに所属させることができる。 上記例の場合、概要と詳細の両方のグループに所属させたい項目であるため、2 つのマーカーインターフェースを引数にしている。
(4)	単一のグループで出力したい項目には、マーカーインターフェースを引数にすることで、該当のグループに所属させることができる。 この場合は要素が 1 つため、配列にする必要はない。 上記例の場合、詳細のみのグループに所属させたい項目であるため、1 つのマーカーインターフェースを引数にしている。
(5)	グループに所属しない項目には`@JsonView` を設定しない。 グループに所属しない項目を出力するかどうかは設定によって変えることができる。 設定方法については後述する。

- `MemberRestController.java`

```
package org.terasoluna.examples.rest.api.member;

import java.util.ArrayList;
```

```
import java.util.List;

import javax.inject.Inject;

import org.dozer.Mapper;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import org.terasoluna.examples.rest.domain.model.Member;
import org.terasoluna.examples.rest.domain.service.member.MemberService;

import com.fasterxml.jackson.annotation.JsonView;

@RequestMapping("members")
@RestController
public class MemberRestController {

    @Inject
    MemberService memberService;

    @Inject
    Mapper beanMapper;

    // (1)
    @JsonView(Summary.class)
    @RequestMapping(value = "{memberId}", params = "format=summary", method =␣
↪RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public MemberResource getMemberSummary(@PathVariable("memberId") String␣
↪memberId) {

        Member member = memberService.getMember(memberId);

        MemberResource responseResource = beanMapper.map(member,
                MemberResource.class);

        return responseResource;
    }

    // (2)
    @JsonView(Detail.class)
    @RequestMapping(value = "{memberId}", params = "format=detail", method =␣
↪RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public MemberResource getMemberDetail(@PathVariable("memberId") String␣
↪memberId) {

        Member member = memberService.getMember(memberId);
```

```
        MemberResource responseResource = beanMapper.map(member,
                MemberResource.class);

        return responseResource;
    }
}
```

項番	説明
(1)	`@JsonView` を付けて、出力したいグループのマーカーインターフェースを設定する。 概要を出力するメソッドに `Summary` マーカーインターフェースを設定する。
(2)	詳細を出力するメソッドに `Detail` マーカーインターフェースを設定する。

出力されるボディは、Controller で指定したグループに所属するプロパティーのみ出力される。
出力例は以下の通りとなる。

- Summary

```
{
  "memberId" : "M000000001",
  "firstName" : "John",
  "lastName" : "Smith",
  "createdAt" : "2014-03-14T11:02:41.477Z",
  "lastModifiedAt" : "2014-03-14T11:02:41.477Z"
}
```

- Detail

```
{
  "memberId" : "M000000001",
  "firstName" : "John",
  "lastName" : "Smith",
  "genderCode" : "1",
```

```
  "dateOfBirth" : "2013-03-14",
  "emailAddress" : "user1394794959984@test.com",
  "telephoneNumber" : "09012345678",
  "zipCode" : "1710051",
  "address" : "Tokyo",
  "createdAt" : "2014-03-14T11:02:41.477Z",
  "lastModifiedAt" : "2014-03-14T11:02:41.477Z"
}
```

@JsonView を付けなかったプロパティーは、MapperFeature.DEFAULT_VIEW_INCLUSION の設定を有効にすれば出力され、無効にすれば出力されない。

上記の出力例は、MapperFeature.DEFAULT_VIEW_INCLUSION を有効にした場合の出力例である。

MapperFeature.DEFAULT_VIEW_INCLUSION を有効にする場合は、以下のように設定する。

```
<bean id="objectMapper" class="org.springframework.http.converter.json.
↪Jackson2ObjectMapperFactoryBean">
    <!-- ... -->

    <!-- (1) -->
    <property name="featuresToEnable">
        <array>
            <util:constant static-field="com.fasterxml.jackson.databind.
↪MapperFeature.DEFAULT_VIEW_INCLUSION"/>
        </array>
    </property>
</bean>
```

項番	説明
(1)	featuresToEnable 要素に MapperFeature.DEFAULT_VIEW_INCLUSION を定義することで設定が有効となる。

MapperFeature.DEFAULT_VIEW_INCLUSION を無効にする場合は、以下のように設定する。

```
<bean id="objectMapper" class="org.springframework.http.converter.json.
↪Jackson2ObjectMapperFactoryBean">
    <!-- ... -->

    <!-- (1) -->
    <property name="featuresToDisable">
        <array>
            <util:constant static-field="com.fasterxml.jackson.databind.
↪MapperFeature.DEFAULT_VIEW_INCLUSION"/>
        </array>
    </property>
</bean>
```

項番	説明
(1)	featuresToDisable 要素に MapperFeature.DEFAULT_VIEW_INCLUSION を定義することで設定が無効となる。

MapperFeature.DEFAULT_VIEW_INCLUSION が無効の場合、先ほどの出力例は、以下のように出力内容が変更される。

- Summary

```
{
  "memberId" : "M000000001",
  "firstName" : "John",
  "lastName" : "Smith"
}
```

- Detail

```
{
  "memberId" : "M000000001",
  "firstName" : "John",
  "lastName" : "Smith",
  "genderCode" : "1",
  "dateOfBirth" : "2013-03-14",
  "emailAddress" : "user1394794959984@test.com",
  "telephoneNumber" : "09012345678",
  "zipCode" : "1710051",
```

```
  "address" : "Tokyo"
}
```

警告: MapperFeature.DEFAULT_VIEW_INCLUSION を指定しない場合のデフォルト値は、ObjectMapper の設定方法によって異なるデフォルト値となるため注意が必要である。*RESTful Web Service で必要となる Spring MVC のコンポーネントを有効化するための設定*でも記述しているが、ObjectMapper の Bean 定義方法を ObjectMapper を直接 Bean 定義するスタイルにすると、デフォルト値が有効になる。Jackson2ObjectMapperFactoryBean を利用すると、デフォルト値は無効になる。設定を明示するため、どちらのスタイルで設定する場合においても、MapperFeature.DEFAULT_VIEW_INCLUSION の指定を記述することを推奨する。

注釈: @JsonView は以下の 2 つの機能を使用して作成されている。これらは、Controller 内の@RequestMapping が付けられた処理メソッドで、Object とのマッピング前後に共通的な処理を実装したい場合に、使用することができる機能である。

- org.springframework.web.servlet.mvc.method.annotation.RequestBodyAdvice
- org.springframework.web.servlet.mvc.method.annotation.ResponseBodyAdvice

@ControllerAdvice をこれらのインタフェースの実装クラスにつけることで適用することができる。@ControllerAdvice の詳細は、*@ControllerAdvice の実装*を参照されたい。

RequestBodyAdvice は下記のメソッドを実装することができる。

- org.springframework.web.servlet.mvc.method.annotation.RequestBodyAdvice

項番	メソッド名	概要
(1)	supports	この Advice が送信されたリクエストに対して適用されるかどうか決定する。true だと適用される。

Continued on next page

表 5.8 – continued from previous page

項番	メソッド名	概要
(2)	handleEmptyBody	リクエストボディの内容を Controller で使用するオブジェクトに反映する前かつ、ボディが空の場合に呼び出される。
(3)	beforeBodyRead	リクエストボディの内容を Controller で使用するオブジェクトに反映する前に呼び出される。
(4)	afterBodyRead	リクエストボディの内容を Controller で使用するオブジェクトに反映した後に呼び出される。

上記すべてのタイミングで処理を記述する必要がない場合は、上記の supports 以外のメソッドが何もしない状態で実装された `org.springframework.web.servlet.mvc.method.annotation.RequestBodyAdviceAdapter` を継承し、必要な部分だけオーバーライドすることで、簡単に実装することができる。

`ResponseBodyAdvice` は下記のメソッドを実装することができる。

- `org.springframework.web.servlet.mvc.method.annotation.ResponseBodyAdvice`

項番	メソッド名	概要
(1)	supports	この Advice が送信されたリクエストに対して適用されるかどうか決定する。`true` だと適用される。
(2)	beforeBodyWrite	Contoroller での処理終了後、レスポンスに返り値を反映する前に呼び出される。

5.1.6 Appendix

JSR-310 Date and Time API / Joda Time を使う場合の設定

リソースを表現する JavaBean(Resource クラス) のプロパティとして JSR-310 Date and Time API を使用する場合は、terasoluna-gfw-common-dependencies にて依存関係が定義されているため依存関係の追加は不要である。一方、Joda Time のクラスを使用する場合は、`pom.xml` に Jackson から提供されている拡張モジュールを依存ライブラリに追加する。

Joda Time のクラスを使用する場合

```
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-jodatime-dependencies</artifactId>
    <type>pom</type>
</dependency>
```

or

```
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-joda</artifactId>
</dependency>
```

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の jackson-datatype-joda は terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

上記以外にも、

- Java SE 7 から追加された `java.nio.file.Path`
- Java SE 8 から追加された `java.util.Optional`
- Hibernate ORM の Lazy Load 機能によって Proxy 化されたオブジェクト

などを扱うための拡張モジュール (`jackson-datatype-xxx`) が、別途 Jackson から提供されている。

RESTful Web Service とクライアントアプリケーションを同じ Web アプリケーションとして動かす際の設定

RESTful Web Service 用の `DispatcherServlet` を設ける方法

RESTful Web Service とクライアントアプリケーションを同じ Web アプリケーションとして構築する場合、RESTful Web Service 用のリクエストを受ける `DispatcherServlet` と、クライアントアプリケーション用のリクエストを受け取る `DispatcherServlet` を分割する事を推奨する。

`DispatcherServlet` を分割する方法について、以下に説明する。

- web.xml

```
<!-- omitted -->

<!-- (1) -->
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
↪class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath*:META-INF/spring/spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

<!-- (2) -->
<servlet>
    <servlet-name>restAppServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
↪class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <!-- (3) -->
        <param-value>classpath*:META-INF/spring/spring-mvc-rest.xml</param-
↪value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
<!-- (4) -->
<servlet-mapping>
    <servlet-name>restAppServlet</servlet-name>
    <url-pattern>/api/v1/*</url-pattern>
</servlet-mapping>

<!-- omitted -->
```

項番	説明
(1)	クライアントアプリケーション用のリクエストを受け取る `DispatcherServlet` とリクエストマッピング。
(2)	RESTful Web Service 用のリクエストを受ける Servlet(`DispatcherServlet`) を追加する。 `<servlet-name>`要素に、RESTful Web Service 用サーブレットであることを示す名前を指定する。 上記例では、サーブレット名として `restAppServlet` を指定している。
(3)	RESTful Web Service 用の `DispatcherServlet` を構築する際に使用する Spring MVC の bean 定義ファイルを指定する。 上記例では、Spring MVC の bean 定義ファイルとして、クラスパス上にある `META-INF/spring/spring-mvc-rest.xml` を指定している。

(4)	RESTful Web Service 用の `DispatcherServlet` へマッピングするサーブレットパスのパターンの指定を行う。
	上記例では、`/api/v1/`配下のサーブレットパスを RESTful Web Service 用の `DispatcherServlet` にマッピングしている。
	具体的には、
	`/api/v1/`
	`/api/v1/members`
	`/api/v1/members/xxxxx`
	といったサーブレットパスが、RESTful Web Service 用の `DispatcherServlet`(`restAppServlet`) にマッピングされる。

ちなみに：@RequestMapping アノテーションの value 属性に指定する値について

`@RequestMapping` アノテーションの value 属性に指定する値は、`<url-pattern>`要素で指定したワイルドカード (「`*`」) の部分の値を指定する。

例えば、`@RequestMapping(value = "members")` と指定した場合、`/api/v1/members` というパスに対する処理を行うメソッドとしてデプロイされる。そのため、`@RequestMapping` アノテーションの value 属性には、分割したサーブレットへマッピングするためパス (`api/v1`) を指定する必要

はない。

`@RequestMapping(value = "api/v1/members")` と指定すると、`/api/v1/api/v1/members` というパスに対する処理を行うメソッドとしてデプロイされてしまうので、注意すること。

ハイパーメディアリンクの実装

JSON の中に関連リソースへのハイパーメディアリンクを含める場合の実装について説明する。

共通部品の実装

- リンク情報を保持する JavaBean を作成する。

```
package org.terasoluna.examples.rest.api.common.resource;

import java.io.Serializable;

// (1)
public class Link implements Serializable {

    private static final long serialVersionUID = 1L;

    private final String rel;

    private final String href;

    public Link(String rel, String href) {
        this.rel = rel;
        this.href = href;
    }

    public String getRel() {
        return rel;
    }

    public String getHref() {
        return href;
    }

}
```

項番	説明
(1)	リンク名と URL を保持するリンク情報用の JavaBean を作成する。

- リンク情報のコレクションを保持する Resource の抽象クラスを作成する。

```
package org.terasoluna.examples.rest.api.common.resource;

import java.net.URI;
import java.util.LinkedHashSet;
import java.util.Set;

import com.fasterxml.jackson.annotation.JsonInclude;

// (2)
public abstract class AbstractLinksSupportedResource {

    // (3)
    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private final Set<Link> links = new LinkedHashSet<>();

    public Set<Link> getLinks() {
        return links;
    }

    // (4)
    public AbstractLinksSupportedResource addLink(String rel, URI href) {
        links.add(new Link(rel, href.toString()));
        return this;
    }

    // (5)
    public AbstractLinksSupportedResource addSelf(URI href) {
        return addLink("self", href);
    }

    // (5)
    public AbstractLinksSupportedResource addParent(URI href) {
        return addLink("parent", href);
    }

}
```

項番	説明
(2)	リンク情報のコレクションを保持する Resource の抽象クラス (JavaBean) を作成する。 本クラスは、ハイパーメディアリンクをサポートする Resource クラスによって、継承される事を想定したクラスである。
(3)	リンク情報を複数保持するフィールドを定義する。 上記例では、リンクの指定がない時に JSON に出力しないようにするために、`@JsonInclude(JsonInclude.Include.NON_EMPTY)` を指定している。
(4)	リンク情報を追加するためのメソッドを用意する。
(5)	必要に応じて共通的なリンク情報を追加するためのメソッドを用意する。 上記例では、自身のリソースにアクセスするためのリンク情報 (`self`) と、親のリソースにアクセスするためのリンク情報 (`parent`) を追加するためのメソッドを用意している。

リソース毎の実装

- Resource クラスにて、リンク情報のコレクションを保持する Resource の抽象クラスを継承する。

```
package org.terasoluna.examples.rest.api.member;

// (1)
public class MemberResource extends
    AbstractLinksSupportedResource implements Serializable {

    // omitted

}
```

項番	説明
(1)	リンク情報のコレクションを保持する Resource の抽象クラスを継承する。 継承することで、リンク情報のコレクションを保持するフィールド (`links`) が取り込まれ、ハイパーメディアリンクをサポートする Resource クラスとなる。

- REST API の処理で、ハイパーメディアリンクを追加する。

```
@RequestMapping("members")
@RestController
public class MemberRestController {

    // omitted

    @RequestMapping(value = "{memberId}", method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public MemberResource getMember(
            @PathVariable("memberId") String memberId
            // (2)
            UriComponentsBuilder uriBuilder) {

        Member member = memberService.getMember(memberId);

        MemberResource responseResource = beanMapper.map(member,
                MemberResource.class);

        // (3)
        responseResource.addSelf(uriBuilder.path("/members").
↪pathSegment(memberId)
                .build().toUri());

        return responseResource;
    }

    // omitted

}
```

項番	説明
(2)	リンク情報に設定する URI を組み立てるため、Spring MVC から提供されている `org.springframework.web.util.UriComponentsBuilder` クラスをメソッドの引数に指定する。 `UriComponentsBuilder` クラスを Controller のメソッドの引数に指定すると、メソッド実行時に、Spring MVC により `UriComponentsBuilder` クラスを継承した `org.springframework.web.servlet.support.ServletUriComponentsBuilder` クラスのインスタンスが渡される。
(3)	リソースにリンク情報を追加する。 上記例では、リンク情報に設定する URI を組み立てるため `UriComponentsBuilder` クラスのメソッドを呼び出し、自身のリソースにアクセスするための URI をリソースに追加している。 Controller のメソッドの引数として渡された `ServletUriComponentsBuilder` のインスタンスは、web.xml に記載の`<servlet-mapping>`要素の情報を元に初期化されており、リソースには依存しない。 そのため、Spring Framework から提供される URI Template Patterns 等を利用し、リクエスト情報をベースに URI を組み立てる事により、リソースに依存しない汎用的な組み立て処理を実装することが可能となる。 例えば、上記例において `http://example.com/api/v1/members/M000000001` に対して GET した場合、組み立てられる URI は、リクエストされた URI と同じ値（`http://example.com/api/v1/members/M000000001`）になる。 必要に応じてリンク情報に設定する URI を組み立てるためのメソッドを実装すること。

ちなみに: `ServletUriComponentsBuilder` では、URI を組み立てる際に「`X-Forwarded-Host`」ヘッダを参照することで、クライアントとアプリケーションサーバの間にロードバランサや Web サーバがある構成を考慮している。ただし、パスの構成を合わせておかないと期待通りの URI にならないので注意が必要である。

- レスポンス例

実際に動かすと、以下のようなレスポンスとなる。

```
GET /rest-api-web/api/v1/members/M000000001 HTTP/1.1
Accept: text/plain, application/json, application/*+json, */*
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
```

```
{
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/rest-api-web/api/v1/members/M000000001"
  } ],
  "memberId" : "M000000001",
  "firstName" : "John",
  "lastName" : "Smith",
  "genderCode" : "1",
  "dateOfBirth" : "2013-03-14",
  "emailAddress" : "user1394794959984@test.com",
  "telephoneNumber" : "09012345678",
  "zipCode" : "1710051",
  "address" : "Tokyo",
  "credential" : {
    "signId" : "user1394794959984@test.com",
    "passwordLastChangedAt" : "2014-03-14T11:02:41.477Z",
    "lastModifiedAt" : "2014-03-14T11:02:41.477Z"
  },
  "createdAt" : "2014-03-14T11:02:41.477Z",
  "lastModifiedAt" : "2014-03-14T11:02:41.477Z"
}
```

HTTP の仕様に準拠した RESTful Web Service の作成

本編で説明した REST API の実装では、HTTP の仕様に準拠していない箇所がある。

本節では、HTTP の仕様に準拠した RESTful Web Service にするための実装例について説明する。

POST 時の Location ヘッダの設定

HTTP の仕様に準拠する場合、POST 時のレスポンスヘッダー (「Location ヘッダ」) には、作成したリソースの URI を設定する必要がある。

POST 時のレスポンスヘッダ (「Location ヘッダ」) に、作成したリソースの URI を設定するための実装方法について説明する。

リソース毎の実装

- REST API の処理で、作成したリソースの URI を Location ヘッダに設定する。

```
@RequestMapping("members")
@RestController
public class MemberRestController {

    // omitted

    @RequestMapping(method = RequestMethod.POST)
    public ResponseEntity<MemberResource> postMembers(
            @RequestBody @Validated({ PostMembers.class, Default.class })
            MemberResource requestedResource,
            // (1)
            UriComponentsBuilder uriBuilder) {

        Member creatingMember = beanMapper.map(requestedResource, Member.
↪class);

        Member createdMember = memberService.createMember(creatingMember);

        MemberResource responseResource = beanMapper.map(createdMember,
                MemberResource.class);

        // (2)
        URI createdUri = uriBuilder.path("/members/{memberId}")
                .buildAndExpand(responseResource.getMemberId()).toUri();

        // (3)
        return ResponseEntity.created(createdUri).body(responseResource);
    }

    // omitted

}
```

項番	説明
(1)	作成したリソースの URI を組み立てるため、Spring MVC から提供されている `org.springframework.web.util.UriComponentsBuilder` クラスをメソッドの引数に指定する。 `UriComponentsBuilder` クラスを Controller のメソッドの引数に指定すると、メソッド実行時に、Spring MVC により `UriComponentsBuilder` クラスを継承した `org.springframework.web.servlet.support.ServletUriComponentsBuilder` クラスのインスタンスが渡される。

Continued on next page

表 5.10 – continued from previous page

項番	説明
(2)	作成したリソースの URI を組み立てる。 上記例では、引数として渡された `ServletUriComponentsBuilder` のインスタンスに `path` メソッドで、URI Template Patterns を用いたパスを追加し、 `buildAndExpand` メソッドを呼び出して、作成したリソースの ID をバインドすることで、作成したリソースの URI を組み立てている。 Controller のメソッドの引数として渡された `ServletUriComponentsBuilder` のインスタンスは、web.xml に記載の`<servlet-mapping>`要素の情報を元に初期化されており、リソースには依存しない。 そのため、Spring Framework から提供される URITemplatePatterns 等を利用し、 リクエスト情報をベースに URI を組み立てる事により、リソースに依存しない汎用的な組み立て処理を実装することが可能となる。 例えば、上記例において `http://example.com/api/v1/members` に対して POST した場合、組み立てられる URI は、「リクエストされた URI + 「/」 + 作成したリソースの ID」となる。 具体的には、ID に `M000000001` を指定した場合、 `http://example.com/api/v1/members/M000000001` となる。 必要に応じてリンク情報に設定する URI を組み立てるためのメソッドを実装すること。
(3)	以下のパラメータを使用して `org.springframework.http.ResponseEntity` を生成し返却する。 • ステータスコード : 201(Created) • Location ヘッダ : 作成したリソースの URI • レスポンス BODY : 作成した Resource オブジェクト

ちなみに: ServletUriComponentsBuilder では、URI を組み立てる際に「X-Forwarded-Host」ヘッダを参照することで、クライアントとアプリケーションサーバの間にロードバランサや Web サーバがある構成を考慮している。ただし、パスの構成を合わせておかないと期待通りの URI にならないので注意が必要である。

- レスポンス例

  実際に動かすと、以下のようなレスポンスヘッダとなる。

```
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
X-Track: 693e132312d64998a7d8d6cabf3d13ef
Location: http://localhost:8080/rest-api-web/api/v1/members/M000000001
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Fri, 14 Mar 2014 12:34:31 GMT
```

CSRF 対策の無効化

RESTful Web Service 向けのリクエストに対して、CSRF 対策を行わないようにするための設定方法について説明する。

ちなみに: CSRF 対策を行わない場合は、セッションを利用する必要がなくなる。

下記設定例では、Spring Security の処理でセッションが使用されなくなる様にしている。

Blank プロジェクトのデフォルトの設定では、CSRF 対策が有効化されているため、以下の設定を追加し、RESTful Web Service 向けのリクエストに対して、CSRF 対策の処理が行われないようにする。

- spring-security.xml

```
<!-- omitted -->

<!-- (1) -->
<sec:http
    pattern="/api/v1/**"
    create-session="stateless">
    <sec:http-basic/>
```

```
    <sec:csrf disabled="true"/>
</sec:http>

<sec:http>
    <sec:access-denied-handler ref="accessDeniedHandler"/>
    <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
    <sec:form-login/>
    <sec:logout/>
    <sec:session-management />
</sec:http>

<!-- omitted -->
```

項番	説明
(1)	REST API 用の Spring Security の定義を追加する。 `<sec:http>`要素の `pattern` 属性に、REST API 用のリクエストパスの URL パターンを指定している。 上記例では、`/api/v1/`で始まるリクエストパスを REST API 用のリクエストパスとして扱う。 また、`create-session` 属性を `stateless` とする事で、Spring Security の処理でセッションが使用されなくなる。 CSRF 対策を無効化するために、`<sec:csrf>`要素に `disabled="true"`を指定している。

XXE 対策の有効化

RESTful Web Service で XML 形式のデータを扱う場合は、XXE(XML External Entity) 対策を行う必要がある。

注釈: XXE(XML External Entity) 対策について

TERASOLUNA Server Framework for Java (5.x) では、XXE 対策が行われている Spring MVC(3.2.10.RELEASE 以上) に依存しているため、個別に対策を行う必要はない。

アプリケーション層のソースコード

How to use の説明で使用したアプリケーション層のソースコードのうち、断片的に貼りつけていたソースコードの完全版を添付しておく。

項番	セクション	ファイル名
(1)	*REST API の実装*	*MemberRestController.java*
(2)	例外のハンドリングの実装	*ApiErrorCreator.java*
(3)		*ApiGlobalExceptionHandler.java*

以下のファイルは、除外している。

- JavaBean
- 設定ファイル

MemberRestController.java

`java/org/terasoluna/examples/rest/api/member/MemberRestController.java`

```
package org.terasoluna.examples.rest.api.member;

import java.util.ArrayList;
import java.util.List;

import javax.inject.Inject;
import javax.validation.groups.Default;

import org.dozer.Mapper;
```

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpStatus;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import org.terasoluna.examples.rest.api.member.MemberResource.PostMembers;
import org.terasoluna.examples.rest.api.member.MemberResource.PutMember;
import org.terasoluna.examples.rest.domain.model.Member;
import org.terasoluna.examples.rest.domain.service.member.MemberService;

@RequestMapping("members")
@RestController
public class MemberRestController {

    @Inject
    MemberService memberService;

    @Inject
    Mapper beanMapper;

    @RequestMapping(method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public Page<MemberResource> getMembers(@Validated MembersSearchQuery query,
            Pageable pageable) {

        Page<Member> page = memberService.searchMembers(query.getName(), pageable);

        List<MemberResource> memberResources = new ArrayList<>();
        for (Member member : page.getContent()) {
            memberResources.add(beanMapper.map(member, MemberResource.class));
        }
        Page<MemberResource> responseResource =
            new PageImpl<>(memberResources, pageable, page.getTotalElements());

        return responseResource;
    }

    @RequestMapping(method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public List<MemberResource> getMembers() {
        List<Member> members = memberService.findAll();

        List<MemberResource> memberResources = new ArrayList<>();
        for (Member member : members) {
            memberResources.add(beanMapper.map(member, MemberResource.class));
        }
        return memberResources;
```

```
    }

    @RequestMapping(method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.CREATED)
    public MemberResource postMembers(@RequestBody @Validated({
            PostMembers.class, Default.class }) MemberResource requestedResource) {

        Member creatingMember = beanMapper.map(requestedResource, Member.class);

        Member createdMember = memberService.createMember(creatingMember);

        MemberResource responseResource = beanMapper.map(createdMember,
                MemberResource.class);

        return responseResource;
    }

    @RequestMapping(value = "{memberId}", method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public MemberResource getMember(@PathVariable("memberId") String memberId) {

        Member member = memberService.getMember(memberId);

        MemberResource responseResource = beanMapper.map(member,
                MemberResource.class);

        return responseResource;
    }

    @RequestMapping(value = "{memberId}", method = RequestMethod.PUT)
    @ResponseStatus(HttpStatus.OK)
    public MemberResource putMember(
            @PathVariable("memberId") String memberId,
            @RequestBody @Validated({
            PutMember.class, Default.class }) MemberResource requestedResource) {

        Member updatingMember = beanMapper.map(requestedResource, Member.class);

        Member updatedMember = memberService.updateMember(memberId,
                updatingMember);

        MemberResource responseResource = beanMapper.map(updatedMember,
                MemberResource.class);

        return responseResource;
    }

    @RequestMapping(value = "{memberId}", method = RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteMember(@PathVariable("memberId") String memberId) {

        memberService.deleteMember(memberId);
```

```
    }

}
```

ApiErrorCreator.java

java/org/terasoluna/examples/rest/api/common/error/ApiErrorCreator.java

```
package org.terasoluna.examples.rest.api.common.error;

import javax.inject.Inject;

import org.springframework.context.MessageSource;
import org.springframework.context.support.DefaultMessageSourceResolvable;
import org.springframework.stereotype.Component;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import org.springframework.validation.ObjectError;
import org.springframework.web.context.request.WebRequest;
import org.terasoluna.gfw.common.message.ResultMessage;
import org.terasoluna.gfw.common.message.ResultMessages;

@Component
public class ApiErrorCreator {

    @Inject
    MessageSource messageSource;

    public ApiError createApiError(WebRequest request, String errorCode,
            String defaultErrorMessage, Object... arguments) {
        String localizedMessage = messageSource.getMessage(errorCode,
                arguments, defaultErrorMessage, request.getLocale());
        return new ApiError(errorCode, localizedMessage);
    }

    public ApiError createBindingResultApiError(WebRequest request,
            String errorCode, BindingResult bindingResult,
            String defaultErrorMessage) {
        ApiError apiError = createApiError(request, errorCode,
                defaultErrorMessage);
        for (FieldError fieldError : bindingResult.getFieldErrors()) {
            apiError.addDetail(createApiError(request, fieldError, fieldError
                    .getField()));
        }
        for (ObjectError objectError : bindingResult.getGlobalErrors()) {
            apiError.addDetail(createApiError(request, objectError, objectError
                    .getObjectName()));
```

```
        }
        return apiError;
    }

    private ApiError createApiError(WebRequest request,
            DefaultMessageSourceResolvable messageResolvable, String target) {
        String localizedMessage = messageSource.getMessage(messageResolvable,
                request.getLocale());
        return new ApiError(messageResolvable.getCode(), localizedMessage, target);
    }

    public ApiError createResultMessagesApiError(WebRequest request,
            String rootErrorCode, ResultMessages resultMessages,
            String defaultErrorMessage) {
        ApiError apiError;
        if (resultMessages.getList().size() == 1) {
            ResultMessage resultMessage = resultMessages.iterator().next();
            String errorCode = resultMessage.getCode();
            String errorText = resultMessage.getText();
            if (errorCode == null && errorText == null) {
                errorCode = rootErrorCode;
            }
            apiError = createApiError(request, errorCode, errorText,
                    resultMessage.getArgs());
        } else {
            apiError = createApiError(request, rootErrorCode,
                    defaultErrorMessage);
            for (ResultMessage resultMessage : resultMessages.getList()) {
                apiError.addDetail(createApiError(request, resultMessage
                        .getCode(), resultMessage.getText(), resultMessage
                        .getArgs()));
            }
        }
        return apiError;
    }

}
```

ApiGlobalExceptionHandler.java

java/org/terasoluna/examples/rest/api/common/error/ApiGlobalExceptionHandler.java

```
package org.terasoluna.examples.rest.api.common.error;

import javax.inject.Inject;
```

```
import org.springframework.dao.OptimisticLockingFailureException;
import org.springframework.dao.PessimisticLockingFailureException;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.http.converter.HttpMessageNotReadableException;
import org.springframework.validation.BindException;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.
↪ResponseEntityExceptionHandler;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ExceptionCodeResolver;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.exception.ResultMessagesNotificationException;

@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @Inject
    ApiErrorCreator apiErrorCreator;

    @Inject
    ExceptionCodeResolver exceptionCodeResolver;

    @Override
    protected ResponseEntity<Object> handleExceptionInternal(Exception ex,
            Object body, HttpHeaders headers, HttpStatus status,
            WebRequest request) {
        final Object apiError;
        if (body == null) {
            String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
            apiError = apiErrorCreator.createApiError(request, errorCode, ex
                    .getLocalizedMessage());
        } else {
            apiError = body;
        }
        return ResponseEntity.status(status).headers(headers).body(apiError);
    }

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        return handleBindingResult(ex, ex.getBindingResult(), headers, status,
                request);
    }

    @Override
    protected ResponseEntity<Object> handleBindException(BindException ex,
```

```
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        return handleBindingResult(ex, ex.getBindingResult(), headers, status,
                request);
    }

    private ResponseEntity<Object> handleBindingResult(Exception ex,
            BindingResult bindingResult, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
        ApiError apiError = apiErrorCreator.createBindingResultApiError(
                request, errorCode, bindingResult, ex.getMessage());
        return handleExceptionInternal(ex, apiError, headers, status, request);
    }

    @Override
    protected ResponseEntity<Object> handleHttpMessageNotReadable(
            HttpMessageNotReadableException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        if (ex.getCause() instanceof Exception) {
            return handleExceptionInternal((Exception) ex.getCause(), null,
                    headers, status, request);
        } else {
            return handleExceptionInternal(ex, null, headers, status, request);
        }
    }

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Object> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {
        return handleResultMessagesNotificationException(ex, new HttpHeaders(),
                HttpStatus.NOT_FOUND, request);
    }

    @ExceptionHandler(BusinessException.class)
    public ResponseEntity<Object> handleBusinessException(BusinessException ex,
            WebRequest request) {
        return handleResultMessagesNotificationException(ex, new HttpHeaders(),
                HttpStatus.CONFLICT, request);
    }

    private ResponseEntity<Object> handleResultMessagesNotificationException(
            ResultMessagesNotificationException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
        ApiError apiError = apiErrorCreator.createResultMessagesApiError(
                request, errorCode, ex.getResultMessages(), ex.getMessage());
        return handleExceptionInternal(ex, apiError, headers, status, request);
    }

    @ExceptionHandler({ OptimisticLockingFailureException.class,
            PessimisticLockingFailureException.class })
    public ResponseEntity<Object> handleLockingFailureException(Exception ex,
            WebRequest request) {
```

```
        return handleExceptionInternal(ex, null, new HttpHeaders(),
                HttpStatus.CONFLICT, request);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<Object> handleSystemError(Exception ex,
            WebRequest request) {
        return handleExceptionInternal(ex, null, new HttpHeaders(),
                HttpStatus.INTERNAL_SERVER_ERROR, request);
    }

}
```

REST API 実装時に作成したドメイン層のクラスのソースコード

How to use で説明した REST API から呼び出しているドメイン層のクラスのソースコードを添付しておく。

なお、インフラストラクチャ層は、MyBatis3 を使って実装している。

項番	分類	ファイル名
(1)	model	*Member.java*
(2)		*MemberCredentia.java*
(3)		*Gender.java*
(4)	repository	*MemberRepository.java*
(5)	service	*MemberService.java*
(6)		*MemberServiceImpl.java*
(7)	other	*DomainMessageCodes.java*
(8)		*GenderTypeHandler.java*
(9)		*member-mapping.xml*
(10)		*mybatis-config.xml*
(11)		*MemberRepository.xml*

以下のファイルは、除外している。

- Entity 以外の JavaBean
- Dozer 以外の設定ファイル

Member.java

java/org/terasoluna/examples/rest/domain/model/Member.java

```
package org.terasoluna.examples.rest.domain.model;

import java.io.Serializable;
import org.joda.time.DateTime;
import org.joda.time.LocalDate;

public class Member implements Serializable {

    private static final long serialVersionUID = 1L;

    private String memberId;

    private String firstName;

    private String lastName;

    private Gender gender;

    private LocalDate dateOfBirth;

    private String emailAddress;

    private String telephoneNumber;

    private String zipCode;

    private String address;

    private DateTime createdAt;

    private DateTime lastModifiedAt;

    private long version;

    private MemberCredential credential;

    public String getMemberId() {
        return memberId;
```

```
    }

    public void setMemberId(String memberId) {
        this.memberId = memberId;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Gender getGender() {
        return gender;
    }

    public void setGender(Gender gender) {
        this.gender = gender;
    }

    public String getGenderCode() {
        if (gender == null) {
            return null;
        } else {
            return gender.getCode();
        }
    }

    public void setGenderCode(String genderCode) {
        this.gender = Gender.getByCode(genderCode);
    }

    public LocalDate getDateOfBirth() {
        return dateOfBirth;
    }

    public void setDateOfBirth(LocalDate dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }

    public String getEmailAddress() {
        return emailAddress;
    }
```

```
    public void setEmailAddress(String emailAddress) {
        this.emailAddress = emailAddress;
    }

    public String getTelephoneNumber() {
        return telephoneNumber;
    }

    public void setTelephoneNumber(String telephoneNumber) {
        this.telephoneNumber = telephoneNumber;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public DateTime getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(DateTime createdAt) {
        this.createdAt = createdAt;
    }

    public DateTime getLastModifiedAt() {
        return lastModifiedAt;
    }

    public void setLastModifiedAt(DateTime lastModifiedAt) {
        this.lastModifiedAt = lastModifiedAt;
    }

    public long getVersion() {
        return version;
    }

    public void setVersion(long version) {
        this.version = version;
    }
```

```
    public MemberCredential getCredential() {
        return credential;
    }

    public void setCredential(MemberCredential credential) {
        this.credential = credential;
    }

}
```

MemberCredentia.java

java/org/terasoluna/examples/rest/domain/model/MemberCredential.java

```
package org.terasoluna.examples.rest.domain.model;

import java.io.Serializable;
import org.joda.time.DateTime;

public class MemberCredential implements Serializable {

    private static final long serialVersionUID = 1L;

    private String memberId;

    private String signId;

    private String password;

    private String previousPassword;

    private DateTime passwordLastChangedAt;

    private DateTime lastModifiedAt;

    private long version;

    public String getMemberId() {
        return memberId;
    }

    public void setMemberId(String memberId) {
        this.memberId = memberId;
    }

    public String getSignId() {
        return signId;
```

```
    }

    public void setSignId(String signId) {
        this.signId = signId;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getPreviousPassword() {
        return previousPassword;
    }

    public void setPreviousPassword(String previousPassword) {
        this.previousPassword = previousPassword;
    }

    public DateTime getPasswordLastChangedAt() {
        return passwordLastChangedAt;
    }

    public void setPasswordLastChangedAt(DateTime passwordLastChangedAt) {
        this.passwordLastChangedAt = passwordLastChangedAt;
    }

    public DateTime getLastModifiedAt() {
        return lastModifiedAt;
    }

    public void setLastModifiedAt(DateTime lastModifiedAt) {
        this.lastModifiedAt = lastModifiedAt;
    }

    public long getVersion() {
        return version;
    }

    public void setVersion(long version) {
        this.version = version;
    }

}
```

Gender.java

java/org/terasoluna/examples/rest/domain/model/Gender.java

```
package org.terasoluna.examples.rest.domain.model;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import org.springframework.util.Assert;

public enum Gender {

    UNKNOWN("0"), MEN("1"), WOMEN("2");

    private static final Map<String, Gender> genderMap;

    static {
        Map<String, Gender> map = new HashMap<>();
        for (Gender gender : values()) {
            map.put(gender.code, gender);
        }
        genderMap = Collections.unmodifiableMap(map);
    }

    private final String code;

    private Gender(String code) {
        this.code = code;
    }

    public static Gender getByCode(String code) {
        Gender gender = genderMap.get(code);
        Assert.notNull(gender, "gender code is invalid. code : " + code);
        return gender;
    }

    public String getCode() {
        return code;
    }

}
```

MemberRepository.java

java/org/terasoluna/examples/rest/domain/repository/member/

MemberRepository.java

```
package org.terasoluna.examples.rest.domain.repository.member;

import java.util.List;
import org.apache.ibatis.session.RowBounds;

import org.terasoluna.examples.rest.domain.model.Member;

public interface MemberRepository {

    Member findOne(String memberId);

    List<Member> findAll();

    long countByContainsName(String name);
    List<Member> findPageByContainsName(String name, RowBounds rowBounds);

    void createMember(Member creatingMember);
    void createCredential(Member creatingMember);

    boolean updateMember(Member updatingMember);

    void deleteMember(String memberId);
    void deleteCredential(String memberId);

}
```

MemberService.java

java/org/terasoluna/examples/rest/domain/service/member/MemberService.java

```
package org.terasoluna.examples.rest.domain.service.member;

import java.util.List;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.terasoluna.examples.rest.domain.model.Member;

public interface MemberService {

    List<Member> findAll();

    Page<Member> searchMembers(String name, Pageable pageable);

    Member getMember(String memberId);
```

```
    Member createMember(Member creatingMember);

    Member updateMember(String memberId, Member updatingMember);

    void deleteMember(String memberId);

}
```

MemberServiceImpl.java

java/org/terasoluna/examples/rest/domain/service/member/MemberServiceImpl.java

```
package org.terasoluna.examples.rest.domain.service.member;

import java.util.ArrayList;
import java.util.List;
import javax.inject.Inject;
import org.apache.ibatis.session.RowBounds;
import org.dozer.Mapper;
import org.joda.time.DateTime;
import org.springframework.dao.DuplicateKeyException;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.Pageable;
import org.springframework.orm.ObjectOptimisticLockingFailureException;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.util.StringUtils;
import org.terasoluna.examples.rest.domain.message.DomainMessageCodes;
import org.terasoluna.examples.rest.domain.model.Member;
import org.terasoluna.examples.rest.domain.model.MemberCredential;
import org.terasoluna.examples.rest.domain.repository.member.MemberRepository;
import org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.message.ResultMessages;

@Transactional
@Service
public class MemberServiceImpl implements MemberService {

    @Inject
    MemberRepository memberRepository;

    @Inject
```

```
    JodaTimeDateFactory dateFactory;

    @Inject
    PasswordEncoder passwordEncoder;

    @Inject
    Mapper beanMapper;

    @Override
    @Transactional(readOnly = true)
    public List<RestMember> findAll() {
        return restMemberRepository.findAll();
    }

    @Override
    @Transactional(readOnly = true)
    public Page<Member> searchMembers(String name, Pageable pageable) {
        List<Member> members = null;
        // Count Members by search criteria
        long total = memberRepository.countByContainsName(name);
        if (0 < total) {
             RowBounds rowBounds = new RowBounds(pageable.getOffset(), pageable.
↪getPageSize());
             members = memberRepository.findPageByContainsName(name, rowBounds);
        } else {
            members = new ArrayList<Member>();
        }
        return new PageImpl<Member>(members, pageable, total);
    }

    @Override
    @Transactional(readOnly = true)
    public Member getMember(String memberId) {
        // find member
        Member member = memberRepository.findOne(memberId);
        if (member == null) {
            // If member is not exists
            throw new ResourceNotFoundException(ResultMessages.error().add(
                            DomainMessageCodes.E_EX_MM_5001, memberId));
        }
        return member;
    }

    @Override
    public Member createMember(Member creatingMember) {
        MemberCredential creatingCredential = creatingMember
                            .getCredential();

        // get processing current date time
        DateTime currentDateTime = dateFactory.newDateTime();

        creatingMember.setCreatedAt(currentDateTime);
        creatingMember.setLastModifiedAt(currentDateTime);
```

```
        // decide sign id(email-address)
        String signId = creatingCredential.getSignId();
        if (!StringUtils.hasLength(signId)) {
            signId = creatingMember.getEmailAddress();
            creatingCredential.setSignId(signId.toLowerCase());
        }

        // encrypt password
        String rawPassword = creatingCredential.getPassword();
        creatingCredential.setPassword(passwordEncoder.encode(rawPassword));
        creatingCredential.setPasswordLastChangedAt(currentDateTime);
        creatingCredential.setLastModifiedAt(currentDateTime);

        // save member & member credential
        try {

            // Registering member details
            memberRepository.createMember(creatingMember);
            // //Registering credential details
            memberRepository.createCredential(creatingMember);
            return creatingMember;
        } catch (DuplicateKeyException e) {
            // If sign id is already used
            throw new BusinessException(ResultMessages.error().add(
                            DomainMessageCodes.E_EX_MM_8001,
                            creatingCredential.getSignId()), e);
        }
    }

    @Override
    public Member updateMember(String memberId, Member updatingMember) {
        // get member
        Member member = getMember(memberId);

        // override updating member attributes
        beanMapper.map(updatingMember, member, "member.update");

        // get processing current date time
        DateTime currentDateTime = dateFactory.newDateTime();
        member.setLastModifiedAt(currentDateTime);

        // save updating member
        boolean updated = memberRepository.updateMember(member);
        if (!updated) {
                throw new ObjectOptimisticLockingFailureException(Member.class,
                                member.getMemberId());
        }
        return member;
    }

    @Override
    public void deleteMember(String memberId) {
```

```
        // First Delete from credential (Child)
        memberRepository.deleteCredential(memberId);
        // Delete member
        memberRepository.deleteMember(memberId);
    }

}
```

DomainMessageCodes.java

java/org/terasoluna/examples/rest/domain/message/DomainMessageCodes.java

```
package org.terasoluna.examples.rest.domain.message;

/**
 * Message codes of domain layer message.
 * @author DomainMessageCodesGenerator
 */
public class DomainMessageCodes {

    private DomainMessageCodes() {
        // NOP
    }

    /** e.ex.mm.5001=Specified member not found. member id : {0} */
    public static final String E_EX_MM_5001 = "e.ex.mm.5001";

    /** e.ex.mm.8001=Cannot use specified sign id. sign id : {0} */
    public static final String E_EX_MM_8001 = "e.ex.mm.8001";
}
```

GenderTypeHandler.java

Enum 型のコード値をマッピングするためのタイプハンドラー。

java/org/terasoluna/examples/infra/mybatis/typehandler/GenderTypeHandler.java

```
package org.terasoluna.examples.infra.mybatis.typehandler;

import java.sql.CallableStatement;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.terasoluna.examples.domain.model.Gender;
import org.apache.ibatis.type.JdbcType;
import org.apache.ibatis.type.BaseTypeHandler;

public class GenderTypeHandler extends BaseTypeHandler<Gender> {

    @Override
    public Gender getNullableResult(ResultSet rs, String columnName) throws␣
↪SQLException {
            return getByCode(rs.getString(columnName));
    }

    @Override
    public Gender getNullableResult(ResultSet rs, int columnIndex) throws␣
↪SQLException {
            return getByCode(rs.getString(columnIndex));
    }

    @Override
    public Gender getNullableResult(CallableStatement cs, int columnIndex)
                    throws SQLException {
            return getByCode(cs.getString(columnIndex));
    }

    @Override
    public void setNonNullParameter(PreparedStatement ps, int i,
                    Gender parameter, JdbcType jdbcType) throws SQLException {
            ps.setString(i, parameter.getCode());
    }

    private Gender getByCode(String byCode) {
            if (byCode == null) {
                return null;
            } else {
                return Gender.getByCode(byCode);
            }
    }
}
```

member-mapping.xml

実装した Service クラスでは、クライアントから指定された値を Member オブジェクトにコピーする際に、「*Bean マッピング (Dozer)*」を使って行っている。

単純なフィールド値のコピーのみでよい場合は、Bean のマッピング定義の追加は不要だが、実装例では、更新対象外の項目 (memberId、credential、createdAt、version) をコピー対象外にする必要がある。

特定のフィールドをコピー対象外にするためには、Bean のマッピング定義の追加が必要となる。

resources/META-INF/dozer/member-mapping.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">

    <mapping map-id="member.update">
        <class-a>org.terasoluna.examples.rest.domain.model.Member</class-a>
        <class-b>org.terasoluna.examples.rest.domain.model.Member</class-b>
        <field-exclude>
            <a>memberId</a>
            <b>memberId</b>
        </field-exclude>
        <field-exclude>
            <a>credential</a>
            <b>credential</b>
        </field-exclude>
        <field-exclude>
            <a>createdAt</a>
            <b>createdAt</b>
        </field-exclude>
        <field-exclude>
            <a>lastModifiedAt</a>
            <b>lastModifiedAt</b>
        </field-exclude>
        <field-exclude>
            <a>version</a>
            <b>version</b>
        </field-exclude>
    </mapping>

</mappings>
```

mybatis-config.xml

MyBatis3 の動作をカスタマイズする場合は、MyBatis 設定ファイルに設定値を追加する。MyBatis3 では、Joda-Time のクラス (org.joda.time.DateTime、org.joda.time.LocalDateTime、org.joda.time.LocalDate など) はサポートされていない。

そのため、Entity クラスのフィールドに Joda-Time のクラスを使用する場合は、Joda-Time 用の `TypeHandler` を用意する必要がある。

org.joda.time.DateTime と java.sql.Timestamp をマッピングするための `TypeHandler` の実装例、「*Joda-Time 用の TypeHandler の実装*」を使って行っている。

`resources/META-INF/mybatis/mybatis-config.xml`

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org/DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <settings>
        <setting name="jdbcTypeForNull" value="NULL" />
        <setting name="mapUnderscoreToCamelCase" value="true" />
    </settings>

    <typeAliases>
        <package name="org.terasoluna.examples.infra.mybatis.typehandler" />
    </typeAliases>

    <typeHandlers>
       <package name="org.terasoluna.examples.infra.mybatis.typehandler" />
    </typeHandlers>

</configuration>
```

MemberRepository.xml

`resources/org/terasoluna/examples/rest/domain/repository/member/MemberRepository.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper
    namespace="org.terasoluna.examples.rest.domain.repository.member.
↪MemberRepository">

```

```
    <resultMap id="MemberResultMap" type="Member">
        <id property="memberId" column="member_id" />
        <result property="firstName" column="first_name" />
        <result property="lastName" column="last_name" />
        <result property="gender" column="gender" />
        <result property="dateOfBirth" column="date_of_birth" />
        <result property="emailAddress" column="email_address" />
        <result property="telephoneNumber" column="telephone_number" />
        <result property="zipCode" column="zip_code" />
        <result property="address" column="address" />
        <result property="createdAt" column="created_at" />
        <result property="lastModifiedAt" column="last_modified_at" />
        <result property="version" column="version" />
        <result property="credential.memberId" column="member_id" />
        <result property="credential.signId" column="sign_id" />
        <result property="credential.password" column="password" />
        <result property="credential.previousPassword" column="previous_password" /
↪>
        <result property="credential.passwordLastChangedAt" column="password_last_
↪changed_at" />
        <result property="credential.lastModifiedAt" column="credential_last_
↪modified_at" />
        <result property="credential.version" column="credential_version" />
    </resultMap>

    <sql id="selectMember">
        SELECT
         member.member_id as member_id
         ,member.first_name as first_name
         ,member.last_name as last_name
         ,member.gender as gender
         ,member.date_of_birth as date_of_birth
         ,member.email_address as email_address
         ,member.telephone_number as telephone_number
         ,member.zip_code as zip_code
         ,member.address as address
         ,member.created_at as created_at
         ,member.last_modified_at as last_modified_at
         ,member.version as version
         ,credential.sign_id as sign_id
         ,credential.password as password
         ,credential.previous_password as previous_password
         ,credential.password_last_changed_at as password_last_changed_at
         ,credential.last_modified_at as credential_last_modified_at
         ,credential.version as credential_version
        FROM
         t_member member
         INNER JOIN t_member_credential credential ON credential.member_id =␣
↪member.member_id
    </sql>

    <sql id="whereMember">
        WHERE
```

```
            member.first_name LIKE #{nameContainingCondition} ESCAPE '~'
            OR member.last_name LIKE #{nameContainingCondition} ESCAPE '~'
    </sql>

    <select id="findAll" resultMap="RestMemberResultMap">
        <include refid="selectRestMember" />
        ORDER BY member_id ASC
    </select>

    <select id="findOne" parameterType="string" resultMap="MemberResultMap">
        <include refid="selectMember" />
        WHERE
        member.member_id = #{memberId}
    </select>

    <select id="countByContainsName" parameterType="string" resultType="_long">
        <bind name="nameContainingCondition"
        value="@org.terasoluna.gfw.common.query.
↪QueryEscapeUtils@toStartingWithCondition(_parameter)" />
        SELECT
        COUNT(*)
        FROM
        t_member member
        <include refid="whereMember" />
    </select>

    <select id="findPageByContainsName" parameterType="string"
        resultMap="MemberResultMap">
        <bind name="nameContainingCondition"
        value="@org.terasoluna.gfw.common.query.
↪QueryEscapeUtils@toStartingWithCondition(_parameter)" />
        <include refid="selectMember" />
        <include refid="whereMember" />
        ORDER BY member_id ASC
    </select>

    <insert id="createMember" parameterType="Member">
        <selectKey keyProperty="memberId" resultType="string" order="BEFORE">
            SELECT 'M'||TO_CHAR(NEXTVAL('s_member'),'FM000000000')
        </selectKey>
        INSERT INTO
        t_member
        (
        member_id
        ,first_name
        ,last_name
        ,gender
        ,date_of_birth
        ,email_address
        ,telephone_number
        ,zip_code
        ,address
        ,created_at
```

```
        ,last_modified_at
        ,version
        )
        VALUES
        (
        #{memberId}
        ,#{firstName}
        ,#{lastName}
        ,#{gender}
        ,#{dateOfBirth}
        ,#{emailAddress}
        ,#{telephoneNumber}
        ,#{zipCode}
        ,#{address}
        ,#{createdAt}
        ,#{lastModifiedAt}
        ,1
        )
    </insert>

    <insert id="createCredential" parameterType="Member">
        INSERT INTO
        t_member_credential
        (
        member_id
        ,sign_id
        ,password
        ,previous_password
        ,password_last_changed_at
        ,last_modified_at
        ,version
        )
        VALUES
        (
        #{memberId}
        ,#{credential.signId}
        ,#{credential.password}
        ,#{credential.previousPassword}
        ,#{credential.passwordLastChangedAt}
        ,#{credential.lastModifiedAt}
        ,1
        )
    </insert>

    <update id="updateMember" parameterType="Member">
        UPDATE
            t_member
        SET
            first_name = #{firstName}
            ,last_name = #{lastName}
            ,gender = #{gender}
            ,date_of_birth = #{dateOfBirth}
            ,email_address = #{emailAddress}
```

```
            ,telephone_number = #{telephoneNumber}
            ,zip_code = #{zipCode}
            ,address = #{address}
            ,created_at = #{createdAt}
            ,last_modified_at = #{lastModifiedAt}
            ,version = version + 1
        WHERE
            member_id = #{memberId}
            AND version = #{version}
    </update>

    <delete id="deleteCredential" parameterType="string">
        DELETE FROM t_member_credential
        WHERE
        member_id = #{memberId}
    </delete>

    <delete id="deleteMember" parameterType="string">
        DELETE FROM t_member
        WHERE
        member_id = #{memberId}
    </delete>

</mapper>
```

5.2 REST クライアント（HTTP クライアント）

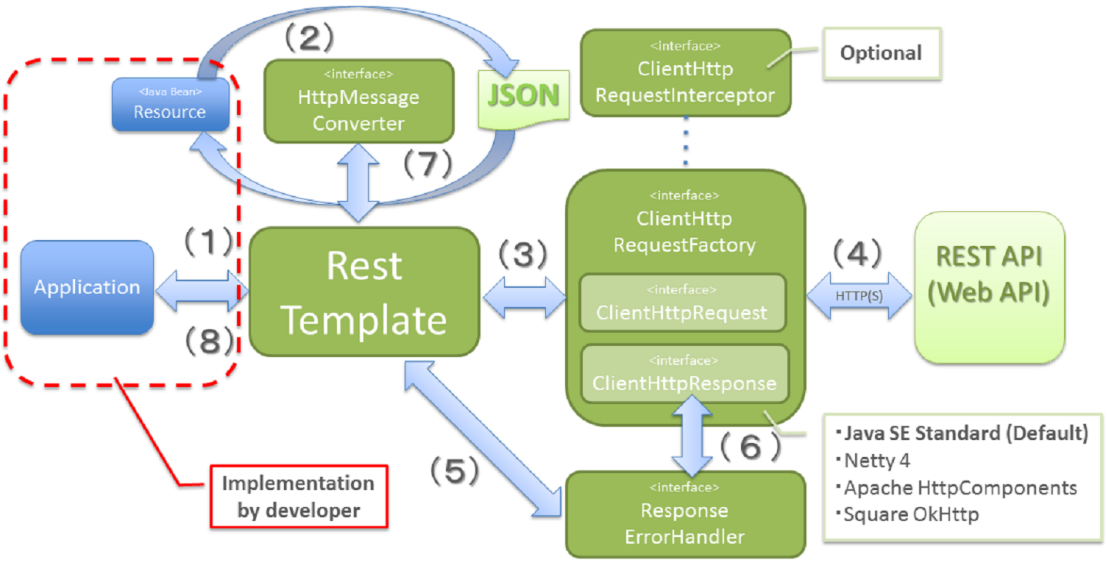
5.2.1 Overview

本節では、Spring Framework が提供する `org.springframework.web.client.RestTemplate` を使用して RESTful Web Service(REST API) を呼び出す実装方法について説明する。

`RestTemplate` とは

`RestTemplate` は、REST API(Web API) を呼び出すためのメソッドを提供するクラスであり、Spring Framework が提供する HTTP クライアントである。

具体的な実装方法の説明を行う前に、`RestTemplate` がどのように REST API(Web API) にアクセスしているかを説明する。



項番	コンポーネント	説明
(1)	アプリケーション	`RestTemplate` のメソッドを呼び出して、REST API(Web API) の呼び出し依頼を行う。
(2)	`RestTemplate`	`HttpMessageConverter` を使用して、Java オブジェクトをリクエストボディに設定する電文 (JSON 等) に変換する。
(3)		`ClientHttpRequestFactory` から `ClientHttpRequest` を取得して、電文 (JSON 等) の送信依頼を行う。
(4)	`ClientHttpRequest`	電文 (JSON 等) をリクエストボディに設定して、REST API(Web API) に HTTP 経由でリクエストを行う。
(5)	`RestTemplate`	`ResponseErrorHandler` を使用して、HTTP 通信のエラー判定及びエラー処理を行う。
(6)	`ResponseErrorHandler`	`ClientHttpResponse` からレスポンスデータを取得して、エラー判定及びエラー処理を行う。
(7)	`RestTemplate`	`HttpMessageConverter` を使用して、レスポンスボディに設定されている電文 (JSON 等) を Java オブジェクトに変換する。
(8)		REST API(Web API) の呼び出し結果 (Java オブジェクト) をアプリケーションへ返却する。

注釈: 非同期処理への対応

REST API からの応答を別スレッドで処理したい場合 (非同期で処理したい場合) は、`RestTemplate` の代わりに `org.springframework.web.client.AsyncRestTemplate` を使用すればよい。非同期処理の

実装例については、非同期リクエスト を参照されたい。

HttpMessageConverter

org.springframework.http.converter.HttpMessageConverter は、アプリケーションで扱う Java オブジェクトとサーバと通信するための電文 (JSON 等) を相互に変換するためのインタフェースである。

RestTemplate を使用した場合、デフォルトで以下の HttpMessageConverter の実装クラスが登録される。

表 5.11: デフォルトで登録される HttpMessageConverter

項番	クラス名	説明	サポート型
(1)	org.springframework.http.converter.ByteArrayHttpMessageConverter	「HTTP ボディ (テキスト or バイナリデータ) ⇔バイト配列」変換用のクラス。 デフォルトですべてのメディアタイプ (*/*) をサポートする。	byte[]
(2)	org.springframework.http.converter.StringHttpMessageConverter	「HTTP ボディ (テキスト) ⇔文字列」変換用のクラス。 デフォルトですべてのテキストメディアタイプ (text/*) をサポートする。	String
(3)	org.springframework.http.converter.ResourceHttpMessageConverter	「HTTP ボディ (バイナリデータ) ⇔ Spring のリソースオブジェクト」変換用のクラス。 デフォルトですべてのメディアタイプ (*/*) をサポートする。	Resource[*1]
(4)	org.springframework.http.converter.xml.SourceHttpMessageConverter	「HTTP ボディ (XML) ⇔ XML ソースオブジェクト」変換用のクラス。 デフォルトで XML 用のメディアタイプ (text/xml,application/xml,application/*-xml) をサポートする。	Source[*2]

Continued on next page

表 5.11 – continued from previous page

項番	クラス名	説明	サポート型
(5)	org.springframework.http.converter.support. AllEncompassingFormHttpMessageConverter	「HTTP ボディ⇔ MultiValueMap オブジェクト」変換用のクラス。 FormHttpMessageConverter の拡張クラスで、multipart のパートデータとして XML と JSON への変換がサポートされている。 デフォルトでフォームデータ用のメディアタイプ (application/x-www-form-urlencoded,multipart/form-data) をサポートする。 • メディアタイプが application/x-www-form-urlencoded の場合、MultiValueMap<String, String>として読込/書込される。 • メディアタイプが multipart/form-data の場合、MultiValueMap<String, Object>として書込され、Object は AllEncompassingFormHttpMessageConverter 内に別途設定される HttpMessageConveter で変換される。(注意： Note 参照) デフォルトで登録されるパートデータ変換用の HttpMessageConveter は、AllEncompassingFormHttpMessageConverter と FormHttpMessageConverter のソースを参照されたい。なお、任意の HttpMessageConverter を登録することもできる。	MultiValueMap[*3]

[*1] org.springframework.core.io パッケージのクラス
[*2] javax.xml.transform パッケージのクラス
[*3] org.springframework.util パッケージのクラス

注釈: **AllEncompassingFormHttpMessageConverter** のメディアタイプが **multipart/form-data** の場合について

メディアタイプが `multipart/form-data` の場合、「`MultiValueMap` オブジェクト から HTTP ボディ」への変換は可能だが、「HTTP ボディ から `MultiValueMap` オブジェクト」への変換は現状サポートされていない。よって、「HTTP ボディ から `MultiValueMap` オブジェクト」への変換を行いたい場合は、独自に実装する必要がある。

表 5.12: 依存ライブラリがクラスパス上に存在する場合に登録される HttpMessageConverter

項番	クラス名	説明	サポート型
(6)	org.springframework.http.converter.feed.AtomFeedHttpMessageConverter	「HTTP ボディ (Atom) ⇔ Atom フィードオブジェクト」変換用のクラス。 デフォルトで ATOM 用のメディアタイプ (application/atom+xml) をサポートする。 (ROME がクラスパスに存在する場合に登録される)	Feed[*4]
(7)	org.springframework.http.converter.feed.RssChannelHttpMessageConverter	「HTTP ボディ (RSS) ⇔ Rss チャネルオブジェクト」変換用のクラス。 デフォルトで RSS 用のメディアタイプ (application/rss+xml) をサポートする。 (ROME がクラスパスに存在する場合に登録される)	Channel[*5]
(8)	org.springframework.http.converter.json.MappingJackson2HttpMessageConverter	「HTTP ボディ (JSON) ⇔ JavaBean」変換用のクラス。 デフォルトで JSON 用のメディアタイプ (application/json,application/*+json) をサポートする。 (Jackson2 がクラスパスに存在する場合に登録される)	Object (JavaBean) Map
(9)	org.springframework.http.converter.xml.MappingJackson2XmlHttpMessageConverter	「HTTP ボディ (XML) ⇔ JavaBean」変換用のクラス。 デフォルトで XML 用のメディアタイプ (text/xml,application/xml,application/*-xml) をサポートする。 (Jackson-dataformat-xml がクラスパスに存在する場合に登録される)	Object (JavaBean) Map

Continued on next page

表 5.12 – continued from previous page

項番	クラス名	説明	サポート型
(10)	org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter	「HTTP ボディ (XML) ⇔ JavaBean」変換用のクラス。 デフォルトで XML 用のメディアタイプ (text/xml,application/xml,application/*+xml) をサポートする。 (JAXB がクラスパスに存在する場合に登録される)	Object (JavaBean)
(11)	org.springframework.http.converter.json.GsonHttpMessageConverter	「HTTP ボディ (JSON) ⇔ JavaBean」変換用のクラス。 デフォルトで JSON 用のメディアタイプ (application/json,application/*+json) をサポートする。 (Gson がクラスパスに存在する場合に登録される)	Object (JavaBean) Map

[*4] com.rometools.rome.feed.atom パッケージのクラス
[*5] com.rometools.rome.feed.rss パッケージのクラス

ClientHttpRequestFactory

RestTemplate は、サーバとの通信処理を以下の 3 つのインタフェースの実装クラスに委譲することで実現している。

- org.springframework.http.client.ClientHttpRequestFactory
- org.springframework.http.client.ClientHttpRequest
- org.springframework.http.client.ClientHttpResponse

この 3 つのインタフェースのうち、開発者が意識するのは ClientHttpRequestFactory である。ClientHttpRequestFactory は、サーバとの通信処理を行うクラス (ClientHttpRequest と ClientHttpResponse インタフェースの実装クラス) を解決する役割を担っている。

なお、Spring Framework が提供している主な ClientHttpRequestFactory の実装クラスは以下の通りである。

表 5.13 Spring Framework が提供している主な ClientHttpRequestFactory の実装クラス

項番	クラス名	説明
(1)	org.springframework.http.client.SimpleClientHttpRequestFactory	Java SE 標準の HttpURLConnection の API を使用して通信処理 (同期、非同期) を行うための実装クラス。(デフォルトで使用される実装クラス)
(2)	org.springframework.http.client.Netty4ClientHttpRequestFactory	Netty 4 の API を使用して通信処理 (同期、非同期) を行うための実装クラス。
(3)	org.springframework.http.client.HttpComponentsClientHttpRequestFactory	Apache HttpComponents HttpClient の API を使用して同期型の通信処理を行うための実装クラス。(HttpClient 4.3 以上が必要)
(4)	org.springframework.http.client.HttpComponentsAsyncClientHttpRequestFactory	Apache HttpComponents HttpAsyncClient の API を使用して非同期型の通信処理を行うための実装クラス。(HttpAsyncClient 4.0 以上が必要)
(5)	org.springframework.http.client.OkHttpClientHttpRequestFactory	Square OkHttp の API を使用して通信処理（同期、非同期）を行うための実装クラス。

注釈: 使用する ClientHttpRequestFactory の実装クラスについて

RestTemplate が使用するデフォルト実装は SimpleClientHttpRequestFactory であり、本ガイドラインでも SimpleClientHttpRequestFactory を使用した際の実装例となっている。Java SE の HttpURLConnection で要件が満たせない場合は、Netty、Apache Http Components などのライブラリの利用を検討されたい。

ResponseErrorHandler

RestTemplate は、サーバとの通信エラーのハンドリングを org.springframework.web.client.ResponseErrorHandler インタフェースに委譲することで実現している。

ResponseErrorHandler には、

- エラー判定を行うメソッド (hasError)
- エラー処理を行うメソッド (handleError)

が定義されており、Spring Framework はデフォルト実装として org.springframework.web.client.DefaultResponseErrorHandler を提供している。

DefaultResponseErrorHandler は、サーバから応答された HTTP ステータスコードの値によって以下のようなエラー処理を行う。

- レスポンスコードが正常系 (2xx) の場合は、エラー処理は行わない。
- レスポンスコードがクライアントエラー系 (4xx) の場合は、org.springframework.web.client.HttpClientErrorException を発生させる。
- レスポンスコードがサーバエラー系 (5xx) の場合は、org.springframework.web.client.HttpServerErrorException を発生させる。
- レスポンスコードが未定義のコード (ユーザ定義のカスタムコード) の場合は、org.springframework.web.client.UnknownHttpStatusCodeException を発生させる。

注釈: エラー時のレスポンスデータの取得方法

エラー時のレスポンスデータ (HTTP ステータスコード、レスポンスヘッダ、レスポンスボディなど) は、例外クラスの getter メソッドを呼び出すことで取得することができる。

ClientHttpRequestInterceptor

org.springframework.http.client.ClientHttpRequestInterceptor は、サーバとの通信の前後に共通的な処理を実装するためのインタフェースである。

ClientHttpRequestInterceptor を使用すると、

- サーバとの通信ログ
- 認証ヘッダの設定

といった共通的な処理を RestTemplate に適用することができる。

注釈: **ClientHttpRequestInterceptor の動作仕様**

`ClientHttpRequestInterceptor` は複数適用することができ、指定した順番でチェーン実行される。これはサーブレットフィルタの動作によく似ており、最後に実行されるチェーン先として `ClientHttpRequest` による HTTP 通信処理が登録されている。例えば、ある条件に一致した際にサーバとの通信処理をキャンセルしたいという要件があった場合は、チェーン先を呼びださなければよい。

この仕組みを活用すると、

- サーバとの通信の閉塞
- 通信処理のリトライ

といった処理を適用することもできる。

5.2.2 How to use

本節では、`RestTemplate` を使用したクライアント処理の実装方法について説明する。

注釈: **RestTemplate がサポートする HTTP メソッドについて**

本ガイドラインでは、GET メソッドと POST メソッドを使用したクライアント処理の実装例のみを紹介するが、`RestTemplate` は他の HTTP メソッド (PUT, PATCH, DELETE, HEAD, OPTIONS など) もサポートしており、同じような要領で使用することができる。詳細は RestTemplate の Javadoc を参照されたい。

`RestTemplate` のセットアップ

`RestTemplate` を使用する場合は、`RestTemplate` を DI コンテナに登録し、`RestTemplate` を利用するコンポーネントにインジェクションする。

依存ライブラリ設定

`RestTemplate` を使用するために `pom.xml` に、Spring Framework の spring-web ライブラリを追加する。マルチプロジェクト構成の場合は、domain プロジェクトの `pom.xml` に追加する。

```
<dependencies>

    <!-- (1) -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
    </dependency>

</dependencies>
```

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

項番	説明
(1)	Spring Framework の `spring-web` ライブラリを dependencies に追加する。

`RestTemplate` の bean 定義

`RestTemplate` の bean 定義を行い、DI コンテナに登録する。

bean 定義ファイル (applicationContext.xml) の定義例

```
<bean id="restTemplate" class="org.springframework.web.client.RestTemplate" /> <!--
↪ (1) -->
```

項番	説明
(1)	`RestTemplate` をデフォルト設定のまま利用する場合は、デフォルトコンストラクタを使用して bean を登録する。

注釈: **RestTemplate のカスタマイズ方法**

HTTP 通信処理をカスタマイズする場合は、以下のような bean 定義となる。

```
<bean id="clientHttpRequestFactory"
      class="org.springframework.http.client.SimpleClientHttpRequestFactory">
↪<!-- (1) -->
    <!-- Set properties for customize a http communication (omit on this␣
↪sample) -->
</bean>

<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
    <constructor-arg ref="clientHttpRequestFactory" /> <!-- (2) -->
</bean>
```

項番	説明
(1)	`ClientHttpRequestFactory` の bean 定義を行う。 本ガイドラインではタイムアウトの設定をカスタマイズする方法を紹介している。詳細は 通信タイムアウトの設定 を参照されたい。
(2)	`ClientHttpRequestFactory` を引数に指定するコンストラクタを使用して bean を登録する。

なお、`HttpMessageConverter`、`ResponseErrorHandler`、`ClientHttpRequestInterceptor` のカスタマイズ方法については、

- 任意の *HttpMessageConverter* を登録する方法
- *ResponseEntity* の返却（エラーハンドラの拡張）
- 共通処理の適用（*ClientHttpRequestInterceptor*）

を参照されたい。

`RestTemplate` の利用

`RestTemplate` を利用する場合は、DI コンテナに登録されている `RestTemplate` をインジェクションする。

RestTemplate のインジェクション例

```
@Service
public class AccountServiceImpl implements AccountService {

    @Inject
    RestTemplate restTemplate;

    // ...

}
```

GET リクエストの送信

`RestTemplate` は、GET リクエストを行うためのメソッドを複数提供している。

- 通常は `getForObject` メソッド又は `getForEntity` メソッドを使用する。
- 任意のヘッダを設定するなどリクエストに細かい設定を行いたい場合は、`org.springframework.http.RequestEntity` と `exchange` メソッドを使用する。

getForObject メソッドを使用した実装

レスポンスボディのみ取得できればよい場合は、getForObject メソッドを使用する。

getForObject メソッドの使用例

フィールド宣言部

```
@Value("${api.url:http://localhost:8080/api}")
URI uri;
```

メソッド内部

```
User user = restTemplate.getForObject(uri, User.class); // (1)
```

項番	説明
(1)	getForObject メソッドを使用した場合は、戻り値はレスポンスボディの値になる。 レスポンスボディのデータは HttpMessageConverter によって第 2 引数に指定した Java クラスへ変換された後、返却される。

getForEntity メソッドを使用した実装

HTTP ステータスコード、レスポンスヘッダ、レスポンスボディを取得する必要がある場合は、getForEntity メソッドを使用する。

getForEntity メソッドの使用例

```
ResponseEntity<User> responseEntity =
        restTemplate.getForEntity(uri, User.class); // (1)
HttpStatus statusCode = responseEntity.getStatusCode(); // (2)
HttpHeaders header = responseEntity.getHeaders(); // (3)
User user = responseEntity.getBody(); // (4)
```

項番	説明
(1)	`getForEntity` メソッドを使用した場合は、戻り値は `org.springframework.http.ResponseEntity` となる。
(2)	HTTP ステータスコードは `getStatusCode` メソッドを用いて取得する。
(3)	レスポンスヘッダは `getHeaders` メソッドを用いて取得する。
(4)	レスポンスボディは `getBody` メソッドを用いて取得する。

注釈: ResponseEntity とは

`ResponseEntity` は HTTP レスポンスを表すクラスで、HTTP ステータスコード、レスポンスヘッダ、レスポンスボディの情報を取得することができる。詳細は ResponseEntity の Javadoc を参照されたい。

---

exchange メソッドを使用した実装

リクエストヘッダを指定する必要がある場合は、`org.springframework.http.RequestEntity` を生成し `exchange` メソッドを使用する。

exchange メソッドの使用例

import 部

```
import org.springframework.http.RequestEntity;
import org.springframework.http.ResponseEntity;
```

フィールド宣言部

```
@Value("${api.url:http://localhost:8080/api}")
URI uri;
```

メソッド内部

```
RequestEntity requestEntity = RequestEntity
        .get(uri)//(1)
        .build();//(2)
```

```
ResponseEntity<User> responseEntity =
        restTemplate.exchange(requestEntity, User.class);//(3)

User user = responseEntity.getBody();//(4)
```

項番	説明
(1)	RequestEntity の get メソッドを使用し、GET リクエスト用のリクエストビルダを生成する。 パラメータに URI を設定する。
(2)	RequestEntity.HeadersBuilder の build メソッドを使用し、RequestEntity オブジェクトを作成する。
(3)	exchange メソッドを使用し、リクエストを送信する。第二引数に、レスポンスデータの型を指定する。 レスポンスは、ResponseEntity<T>になる。型パラメータに、レスポンスデータの型を設定する。
(4)	getBody メソッドを使用し、レスポンスボディのデータを取得する。

注釈: RequestEntity とは

RequestEntity は HTTP リクエストを表すクラスで、接続 URI、HTTP メソッド、リクエストヘッダ、リクエストボディを設定することができる。詳細は RequestEntity の Javadoc を参照されたい。

なお、リクエストヘッダの設定方法については、リクエストヘッダの設定 を参照されたい。

POST リクエストの送信

RestTemplate は、POST リクエストを行うためのメソッドを複数提供している。

- 通常は、postForObject、postForEntity を使用する。
- 任意のヘッダを設定するなどリクエストに細かい設定を行いたい場合は、RequestEntity と exchange メソッドを使用する。

postForObject メソッドを使用した実装

POST した結果としてレスポンスボディのみ取得できればよい場合は、postForObject メソッドを使用する。

postForObject メソッドの使用例

```
User user = new User();

//...

User user = restTemplate.postForObject(uri, user, User.class); // (1)
```

項番	説明
(1)	postForObject メソッドは、簡易に POST リクエストを実装できる。 第二引数には、HttpMessageConverter によってリクエストボディに変換される Java オブジェクトを設定する。 postForObject メソッドを使用した場合は、戻り値はレスポンスボディの値になる。

postForEntity メソッドを使用した実装

POST した結果として HTTP ステータスコード、レスポンスヘッダ、レスポンスボディを取得する必要がある場合は、postForEntity メソッドを使用する。

postForEntity メソッドの使用例

```
User user = new User();

//...

ResponseEntity<User> responseEntity =
        restTemplate.postForEntity(uri, user, User.class); // (1)
```

項番	説明
(1)	postForEntity メソッドも getForObject メソッドと同様に簡易に POST リクエストを実装できる。 postForEntity メソッドを使用した場合は、戻り値は ResponseEntity となる。 レスポンスボディの値は、ResponseEntity から取得する。

exchange メソッドを使用した実装

リクエストヘッダを指定する必要がある場合は、RequestEntity を生成し exchange メソッドを使用する。

exchange メソッドの使用例

import 部

```
import org.springframework.http.RequestEntity;
import org.springframework.http.ResponseEntity;
```

フィールド宣言部

```
@Value("${api.url:http://localhost:8080/api}")
URI uri;
```

メソッド内部

```
User user = new User();

//...

RequestEntity<User> requestEntity = RequestEntity//(1)
        .post(uri)//(2)
        .body(user);//(3)

ResponseEntity<User> responseEntity =
        restTemplate.exchange(requestEntity, User.class);//(4)
```

項番	説明
(1)	`RequestEntity` を使用して、リクエストを生成する。型パラメータに、リクエストボディに設定するデータの型を指定する。
(2)	`post` メソッドを使用し、POST リクエスト用のリクエストビルダを生成する。パラメータに URI を設定する。
(3)	`RequestEntity.BodyBuilder` の `body` メソッドを使用し、`RequestEntity` オブジェクトを作成する。 パラメータにリクエストボディに変換する Java オブジェクトを設定する。
(4)	`exchange` メソッドを使用し、リクエストを送信する。

注釈: リクエストヘッダの設定方法

リクエストヘッダの設定方法については、リクエストヘッダの設定 を参照されたい。

コレクション形式のデータ取得

サーバから応答されるレスポンスボディの電文 (JSON 等) がコレクション形式の場合は、以下のような実装となる。

コレクション形式のデータの取得例

```
ResponseEntity<List<User>> responseEntity = //(1)
    restTemplate.exchange(requestEntity, new ParameterizedTypeReference<List<User>>
↪(){}); //(2)

List<User> userList = responseEntity.getBody();//(3)
```

項番	説明
(1)	`ResponseEntity` の型パラメータに `List<レスポンスデータの型>`を指定する。
(2)	`exchange` メソッドの第二引数に `org.springframework.core.ParameterizedTypeReference` のインスタンスを指定し、型パラメータに `List<レスポンスデータの型>`を指定する。
(2)	`getBody` メソッドで、レスポンスボディのデータを取得する。

リクエストヘッダの設定

`RequestEntity` と `exchange` メソッドを使用すると、`RequestEntity` のメソッドを使用して特定のヘッダ及び任意のヘッダを設定することができる。詳細は RequestEntity の Javadoc を参照されたい。

本ガイドラインでは、

- *Content-Type* ヘッダの設定
- *Accept* ヘッダの設定
- 任意のリクエストヘッダの設定

について説明する。

Content-Type ヘッダの設定

サーバへデータを送信する場合は、通常 Content-Type ヘッダの指定が必要となる。

Content-Type ヘッダの設定例

```
User user = new User();

//...

RequestEntity<User> requestEntity = RequestEntity
        .post(uri)
        .contentType(MediaType.APPLICATION_JSON) // (1)
        .body(user);
```

項番	説明
(1)	`RequestEntity.BodyBuilder` の `contentType` メソッドを使用し、Context-Type ヘッダの値を指定する。 上記の実装例では、データ形式が JSON であることを示す「`application/json`」を設定している。

Accept ヘッダの設定

サーバから取得するデータの形式を指定する場合は、Accept ヘッダの指定が必要となる。サーバが複数のデータ形式のレスポンスをサポートしていない場合は、Accept ヘッダを明示的に指定しなくてもよいケースもある。

Accept ヘッダの設定例

```
User user = new User();

//...

RequestEntity<User> requestEntity = RequestEntity
        .post(uri)
        .accept(MediaType.APPLICATION_JSON) // (1)
        .body(user);
```

項番	説明
(1)	`RequestEntity.HeadersBuilder` の `accept` メソッドを使用して、Accept ヘッダの値を設定する。 上記の実装例では、取得可能なデータ形式が JSON であることを示す「`application/json`」を設定している。

任意のリクエストヘッダの設定

サーバへアクセスするために、リクエストヘッダの設定が必要になるケースがある。

任意ヘッダの設定例

```
User user = new User();

//...

RequestEntity<User> requestEntity = RequestEntity
        .post(uri)
```

```
        .header("Authorization", "Basic " + base64Credentials) // (1)
        .body(user);
```

項番	説明
(1)	RequestEntity.HeadersBuilder の header メソッドを使用してリクエストヘッダの名前と値を設定する。 上記の実装例では、Authorization ヘッダに Basic 認証に必要な資格情報を設定している。

エラーハンドリング

例外ハンドリング (デフォルトの動作)

RestTemplate のデフォルト実装 (DefaultResponseErrorHandler) では、

- レスポンスコードがクライアントエラー系 (4xx) の場合は、HttpClientErrorException
- レスポンスコードがサーバエラー系 (5xx) の場合は、HttpServerErrorException
- レスポンスコードが未定義のコード (ユーザ定義のカスタムコード) の場合は、UnknownHttpStatusCodeException

が発生するため、必要に応じてこれらの例外をハンドリングする必要がある。

例外ハンドリングの実装例

注釈: 以下の実装例は、サーバエラーが発生した際の例外ハンドリングの一例である。

アプリケーションの要件に応じて適切な例外ハンドリングを行うこと。

フィールド宣言部

```
@Value("${api.retry.maxCount}")
int retryMaxCount;

@Value("${api.retry.retryWaitTimeCoefficient}")
int retryWaitTimeCoefficient;
```

メソッド内部

```
int retryCount = 0;
while (true) {
    try {

        responseEntity = restTemplate.exchange(requestEntity, String.class);

        if (log.isInfoEnabled()) {
```

```
            log.info("Success({}) ", responseEntity.getStatusCode());
        }

        break;

    } catch (HttpServerErrorException e) { // (1)

        if (retryCount == retryMaxCount) {
            throw e;
        }

        retryCount++;

        if (log.isWarnEnabled()) {
            log.warn("An error ({}) occurred on the server. (The number of retries:
↪{} Times)", e.getStatusCode(),
                retryCount);
        }

        try {
            Thread.sleep(retryWaitTimeCoefficient * retryCount);
        } catch (InterruptedException ie) {
            Thread.currentThread().interrupt();
        }

        //...
    }

}
```

項番	説明
(1)	例外をキャッチしてエラー処理を行う。サーバエラー（500 系）の場合、HttpServerErrorException をキャッチする。

ResponseEntity の返却（エラーハンドラの拡張）

org.springframework.web.client.ResponseErrorHandler インタフェースの実装クラスを RestTemplate に設定することで、独自のエラー処理を行うことができる。

以下の例では、サーバエラー及びクライアントエラーが発生した場合でも ResponseEntity を返すようにエラーハンドラを拡張している。

エラーハンドラの実装クラスの作成例

```
import org.springframework.http.client.ClientHttpResponse;
import org.springframework.web.client.DefaultResponseErrorHandler;
```

```
public class CustomErrorHandler extends DefaultResponseErrorHandler { // (1)

    @Override
    public void handleError(ClientHttpResponse response) throws IOException {
        //Don't throw Exception.
    }

}
```

項番	説明
(1)	ResponseErrorHandler インタフェースの実装クラスを作成する。 上記の作成例では、デフォルトのエラーハンドラの実装クラスである DefaultResponseErrorHandler を拡張し、 サーバエラー及びクライアントエラーが発生した際に例外を発生させずに ResponseEntity が返るようにしている。

bean 定義ファイル (applicationContext.xml) の定義例

```
<bean id="customErrorHandler" class="com.example.restclient.CustomErrorHandler" />
↪<!-- (1) -->

<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
    <property name="errorHandler" ref="customErrorHandler" /><!-- (2) -->
</bean>
```

項番	説明
(1)	ResponseErrorHandler の実装クラスの bean 定義を行う。
(2)	errorHandler プロパティに ResponseErrorHandler の bean をインジェクションする。

クライアント処理の実装例

```
int retryCount = 0;
while (true) {

    responseEntity = restTemplate.exchange(requestEntity, User.class);

    if (responseEntity.getStatusCode() == HttpStatus.OK) { // (1)
```

```
        break;

    } else if (responseEntity.getStatusCode() == HttpStatus.SERVICE_UNAVAILABLE) {␣
↪// (2)

        if (retryCount == retryMaxCount) {
            break;
        }

        retryCount++;

        if (log.isWarnEnabled()) {
            log.warn("An error ({}) occurred on the server. (The number of retries:
↪{} Times)",
                    responseEntity.getStatusCode(), retryCount);
        }

        try {
            Thread.sleep(retryWaitTimeCoefficient * retryCount);
        } catch (InterruptedException ie) {
            Thread.currentThread().interrupt();
        }

        //...
    }
}
```

項番	説明
(1)	上記の実装例では、エラー時にも ResponseEntity を返すようにエラーハンドラを拡張しているので、返却された ResponseEntity から HTTP ステータスコードを取得して、処理結果が正常であったか確認する必要がある。
(2)	エラー発生時も返却された ResponseEntity から HTTP ステータスコードを取得して、その値に応じて処理を制御することができる。

通信タイムアウトの設定

サーバとの通信に対してタイムアウト時間を指定したい場合は、以下のような bean 定義を行う。

bean 定義ファイル (applicationContext.xml) の定義例

```
<bean id="clientHttpRequestFactory"
      class="org.springframework.http.client.SimpleClientHttpRequestFactory">
    <property name="connectTimeout" value="${api.connectTimeout: 2000}" /><!-- (1)␣
↪-->
```

```
    <property name="readTimeout" value="${api.readTimeout: 2000}" /><!-- (2) -->
</bean>

<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
    <constructor-arg ref="clientHttpRequestFactory" />
</bean>
```

項番	説明
(1)	connectTimeout プロパティにサーバとの接続タイムアウト時間 (ミリ秒) を設定する。 タイムアウト発生時は org.springframework.web.client.ResourceAccessException が発生する。
(2)	readTimeout プロパティにレスポンスデータの読み込みタイムアウト時間 (ミリ秒) を設定する。 タイムアウト発生時は ResourceAccessException が発生する。

注釈: タイムアウト発生時の起因例外

ResourceAccessException は起因例外をラップしており、接続タイムアウト及び読み込みタイムアウト発生時の起因例外は共に java.net.SocketTimeoutException である。デフォルト実装 (SimpleClientHttpRequestFactory) を使用した場合は、どちらのタイムアウトが発生したかを例外クラスの種類で区別できないという点を補足しておく。

なお、他の HttpRequestFactory を使用した場合の動作は未検証であるため、起因例外が上記と異なる可能性がある。他の HttpRequestFactory を使用する場合は、タイムアウト時に発生する例外を把握した上で適切な例外ハンドリングを行うこと。

SSL 自己署名証明書の使用

テスト環境などで SSL 自己署名証明書を使用する場合は、以下のように実装する。

FactoryBean の実装例

RestTemplate の Bean 定義で、コンストラクタの引数に渡す org.springframework.http.client.ClientHttpRequestFactory を作成するための org.springframework.beans.factory.FactoryBean を実装する。

```
import java.security.KeyStore;

import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManagerFactory;
```

```
import org.apache.http.client.HttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.springframework.beans.factory.FactoryBean;
import org.springframework.http.client.ClientHttpRequestFactory;
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;

public class RequestFactoryBean implements
        FactoryBean<ClientHttpRequestFactory> {

    private String keyStoreFileName;

    private char[] keyStorePassword;

    @Override
    public ClientHttpRequestFactory getObject() throws Exception {

        // (1)
        SSLContext sslContext = SSLContext.getInstance("TLS");

        KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
        ks.load(this.getClass().getClassLoader()
                .getResourceAsStream(this.keyStoreFileName),
                this.keyStorePassword);

        KeyManagerFactory kmf = KeyManagerFactory.getInstance(KeyManagerFactory
                .getDefaultAlgorithm());
        kmf.init(ks, this.keyStorePassword);

        TrustManagerFactory tmf = TrustManagerFactory
                .getInstance(TrustManagerFactory.getDefaultAlgorithm());
        tmf.init(ks);

        sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

        // (2)
        HttpClient httpClient = HttpClientBuilder.create()
                .setSSLContext(sslContext).build();

        // (3)
        ClientHttpRequestFactory factory = new␣
↪HttpComponentsClientHttpRequestFactory(
                httpClient);

        return factory;
    }

    @Override
    public Class<?> getObjectType() {
        return ClientHttpRequestFactory.class;
    }

    @Override
```

```
    public boolean isSingleton() {
        return true;
    }

    public void setKeyStoreFileName(String keyStoreFileName) {
        this.keyStoreFileName = keyStoreFileName;
    }

    public void setKeyStorePassword(char[] keyStorePassword) {
        this.keyStorePassword = keyStorePassword;
    }

}
```

項番	説明
(1)	後述の bean 定義で指定されたキーストアファイルのファイル名とパスワードを元に、SSL コンテキストを作成する。 使用する SSL 自己署名証明書のキーストアファイルは、クラスパス上に配置する。
(2)	作成した SSL コンテキストを利用する `org.apache.http.client.HttpClient` を作成する。
(3)	作成した `HttpClient` を利用する `ClientHttpRequestFactory` を作成する。

`HttpClient` および `HttpClientBuilder` を使用するためには、Apache HttpComponents HttpClient のライブラリが必要となる。以下を `pom.xml` に追加し、Apache HttpComponents HttpClient を依存ライブラリに追加する。

- `pom.xml`

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
</dependency>
```

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

bean 定義ファイル (applicationContext.xml) の定義例

SSL 自己署名証明書を使用した SSL 通信を行う RestTemplate を定義する。

```
<bean id="httpsRestTemplate" class="org.springframework.web.client.RestTemplate">
    <constructor-arg>
        <bean class="com.example.restclient.RequestFactoryBean"><!-- (1) -->
            <property name="keyStoreFileName" value="${rscl.keystore.filename}" />
            <property name="keyStorePassword" value="${rscl.keystore.password}" />
        </bean>
    </constructor-arg>
</bean>
```

項番	説明
(1)	作成した RequestFactoryBean を RestTemplate のコンストラクタに指定する。 RequestFactoryBean には、キーストアファイルのファイル名とパスワードを渡す。

RestTemplate の使用方法

RestTemplate の使い方については、SSL 自己署名証明書を使用しない場合と同じである。

Basic 認証

サーバが Basic 認証を要求する場合は、以下のように実装する。

Basic 認証の実装例

フィールド宣言部

```
@Value("${api.auth.username}")
String username;

@Value("${api.auth.password}")
String password;
```

メソッド内部

```
String plainCredentials = username + ":" + password; // (1)
String base64Credentials = Base64.getEncoder()
        .encodeToString(plainCredentials.getBytes(StandardCharsets.UTF_8)); // (2)

RequestEntity requestEntity = RequestEntity
      .get(uri)
      .header("Authorization", "Basic " + base64Credentials) // (3)
      .build();
```

項番	説明
(1)	ユーザ名とパスワードを「」:」」でつなげる。
(2)	（1）をバイト配列に変換して、Base64 エンコードする。
(3)	Authorization ヘッダを Basic 認証の資格情報を設定する。

注釈: Java SE8 以降の場合は、Java 標準の `java.util.Base64` を使用する。それ以前の場合は、Spring Security の `org.springframework.security.crypto.codec.Base64` を使用する。

ファイルアップロード (マルチパートリクエスト)

`RestTemplate` を使用してファイルアップロード (マルチパートリクエスト) を行う場合は、以下のように実装する。

ファイルアップロードの実装例

```
MultiValueMap<String, Object> multiPartBody = new LinkedMultiValueMap<>();//(1)
multiPartBody.add("file", new ClassPathResource("/uploadFiles/User.txt"));//(2)

RequestEntity<MultiValueMap<String, Object>> requestEntity = RequestEntity
        .post(uri)
        .contentType(MediaType.MULTIPART_FORM_DATA)//(3)
        .body(multiPartBody);//(4)
```

項番	説明
(1)	マルチパートリクエストとして送信するデータを格納するために `MultiValueMap` を生成する。
(2)	パラメータ名をキーに指定して、アップロードするファイルを `MultiValueMap` に追加する。 上記例では、`file` というパラメータ名を指定して、クラスパス上に配置されているファイルをアップロードファイルとして追加している。
(3)	Content-Type ヘッダのメディアタイプを `multipart/form-data` に設定する。
(4)	アップロードするファイルが格納されている `MultiValueMap` をリクエストボディに設定する。

注釈: Spring Framework が提供する Resource クラスについて

Spring Framework はリソースを表現するインタフェースとして `org.springframework.core.io.Resource` を提供しており、ファイルをアップロードする際に使用することができる。

`Resource` インタフェースの主な実装クラスは以下の通りである。

- `org.springframework.core.io.PathResource`
- `org.springframework.core.io.FileSystemResource`
- `org.springframework.core.io.ClassPathResource`
- `org.springframework.core.io.UrlResource`
- `org.springframework.core.io.InputStreamResource` (ファイル名をサーバに連携できない)
- `org.springframework.core.io.ByteArrayResource` (ファイル名をサーバに連携できない)

ファイルダウンロード

`RestTeamplate` を使用してファイルダウンロードを行う場合は、以下のように実装する。

ファイルダウンロードの実装例（ファイルサイズが小さい場合）

```
RequestEntity requestEntity = RequestEntity
        .get(uri)
```

```
        .build();

ResponseEntity<byte[]> responseEntity =
        restTemplate.exchange(requestEntity, byte[].class);//(1)

byte[] downloadContent = responseEntity.getBody();//(2)
```

項番	説明
(1)	ダウンロードファイルを指定したデータ型で扱う。ここでは、バイト配列を指定。
(2)	レスポンスボディからダウンロードしたファイルのデータを取得する。

警告：サイズの大きいファイルをダウンロードする際の注意点

サイズの大きなファイルをデフォルトで登録されている `HttpMessageConverter` を使用して `byte` 配列で取得すると、`java.lang.OutOfMemoryError` が発生する可能性がある。そのため、サイズの大きなファイルをダウンロードしたい場合は、レスポンスから `InputStream` を取得して、ダウンロードデータを少しずつファイルに書き出す必要がある。

ファイルダウンロードの実装例（ファイルサイズが大きい場合）

```
// (1)
final ResponseExtractor<ResponseEntity<File>> responseExtractor =
        new ResponseExtractor<ResponseEntity<File>>() {

    // (2)
    @Override
    public ResponseEntity<File> extractData(ClientHttpResponse response)
            throws IOException {

        File rcvFile = File.createTempFile("rcvFile", "zip");

        FileCopyUtils.copy(response.getBody(), new FileOutputStream(rcvFile));

        return ResponseEntity.status(response.getStatusCode())
                .headers(response.getHeaders()).body(rcvFile);
    }

};

// (3)
ResponseEntity<File> responseEntity = this.restTemplate.execute(targetUri,
```

```
        HttpMethod.GET, null, responseExtractor);
if (HttpStatus.OK.equals(responseEntity.getStatusCode())) {
    File getFile = responseEntity.getBody();

    .....

}
```

項番	説明
(1)	RestTemplate#execute で取得されたレスポンスから、RestTemplate#execute の戻り値を作成するための処理を作成する。
(2)	レスポンスボディ (InputStream) からデータを読込み、ファイルを作成する。 作成したファイルと HTTP ヘッダ、ステータスコードを ResponseEntity<File>に格納して返却する。
(3)	RestTemplate#execute を使用して、ファイルのダウンロードを行う。

ファイルダウンロードの実装例（ファイルサイズが大きい場合（**ResponseEntity** を使わない例））

ステータスコードの判定や HTTP ヘッダの参照等が不要な場合は、以下のように ResponseEntity ではなく File を返却すればよい。

```
final ResponseExtractor<File> responseExtractor = new ResponseExtractor<File>() {

    @Override
    public File extractData(ClientHttpResponse response)
            throws IOException {

        File rcvFile = File.createTempFile("rcvFile", "zip");

        FileCopyUtils.copy(response.getBody(), new FileOutputStream(
                rcvFile));

        return rcvFile;
    }

};

File getFile = this.restTemplate.execute(targetUri, HttpMethod.GET,
        null, responseExtractor);
.....
```

RESTful な URL(URI テンプレート) を扱う方法と実装例

RESTful な URL を扱うには、URI テンプレートを使用して実装を行えばよい。

getForObject メソッドでの使用例

フィールド宣言部

```
@Value("${api.serverUrl}/api/users/{userId}") // (1)
String uriStr;
```

メソッド内部

```
User user = restTemplate.getForObject(uriStr, User.class, "0001"); // (2)
```

項番	説明
(1)	URI テンプレートの変数{userId}は、RestTeamplate の使用時に指定の値に変換される。
(2)	URI テンプレートの変数 1 つ目が getForObject メソッドの第 3 引数に指定した値で置換され、『http://localhost:8080/api/users/0001』として処理される。

exchange メソッドでの使用例

```
@Value("${api.serverUrl}/api/users/{action}") // (1)
String uriStr;
```

メソッド内部

```
URI targetUri = UriComponentsBuilder.fromUriString(uriStr).
        buildAndExpand("create").toUri(); //(2)

User user = new User();

//...

RequestEntity<User> requestEntity = RequestEntity
        .post(targetUri)
        .body(user);

ResponseEntity<User> responseEntity = restTemplate.exchange(requestEntity, User.
↪class);
```

項番	説明
(1)	URI テンプレートの変数{action}は、RestTeamplate の使用時に指定の値に変換される。
(2)	UriComponentsBuilder を使用することで、URI テンプレートの変数 1 つ目が buildAndExpand の引数で指定した値に置換され、『http://localhost:8080/api/users/create』の URI が作成される。 詳細は UriComponentsBuilder の Javadoc を参照されたい。

5.2.3 How to extend

本節では、RestTemplate の拡張方法について説明する。

任意の HttpMessageConverter を登録する方法

デフォルトで登録されている HttpMessageConverter で電文変換の要件を満たせない場合は、任意の HttpMessageConverter を登録することができる。ただし、デフォルトで登録されていた HttpMessageConverter は削除されるので、必要な HttpMessageConverter をすべて個別に登録する必要がある。

bean 定義ファイル (applicationContext.xml) の定義例

```
<bean id="jaxb2CollectionHttpMessageConverter"
      class="org.springframework.http.converter.xml.
↪Jaxb2CollectionHttpMessageConverter" /> <!-- (1) -->

<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
    <property name="messageConverters"> <!-- (2) -->
        <list>
            <ref bean="jaxb2CollectionHttpMessageConverter" />
        </list>
    </property>
</bean>
```

項番	説明
(1)	登録する HttpMessageConverter の実装クラスを bean 定義する。
(2)	messageConverters プロパティに登録する HttpMessageConverter の bean をインジェクションする。

共通処理の適用（ClientHttpRequestInterceptor）

ClientHttpRequestInterceptor を使用することで、サーバとの通信処理の前後に任意の処理を実行させることができる。

ここでは、ロギング処理 と、*Basic* 認証用のリクエストヘッダ設定処理 を適用する方法を紹介する。

ロギング処理

サーバとの通信ログを出力したい場合は、以下のような実装を行う。

通信ログ出力の実装例

```
package com.example.restclient;

import org.springframework.http.HttpRequest;
import org.springframework.http.client.ClientHttpRequestExecution;
import org.springframework.http.client.ClientHttpRequestInterceptor;
import org.springframework.http.client.ClientHttpResponse;

public class LoggingInterceptor implements ClientHttpRequestInterceptor { //(1)

    private static final Logger logger = LoggerFactory.
↪getLogger(LoggingInterceptor.class);

    @Override
    public ClientHttpResponse intercept(HttpRequest request, byte[] body,
            ClientHttpRequestExecution execution) throws IOException {

        if (log.isInfoEnabled()) {
            String requestBody = new String(body, StandardCharsets.UTF_8);

            log.info("Request Header {}", request.getHeaders()); //(2)
            log.info("Request Body {}", requestBody);
        }

        ClientHttpResponse response = execution.execute(request, body); //(3)

        if (log.isInfoEnabled()) {
            log.info("Response Header {}", response.getHeaders()); // (4)
            log.info("Response Status Code {}", response.getStatusCode()); // (5)
        }

        return response; // (6)
    }

}
```

項番	説明
(1)	`ClientHttpRequestInterceptor` インタフェースを実装する。
(2)	リクエストする前に行いたい共通処理を実装する。 上記の実装例では、リクエストヘッダーとリクエストボディの内容をログに出力している。
(3)	`intercept` メソッドの引数として受け取った `ClientHttpRequestExecution` の `execute` メソッドを実行し、リクエストの送信を実行する。
(4)	レスポンスを受け取った後に行いたい共通処理を実装する。 上記の実装例では、レスポンスヘッダの内容をログに出力している。
(5)	(4) と同様に、ステータスコードの内容をログに出力している。
(6)	(3) で受信したレスポンスをリターンする。

bean 定義ファイル (applicationContext.xml) の定義例

```
<!-- (1) -->
<bean id="loggingInterceptor" class="com.example.restclient.LoggingInterceptor" />
```

項番	説明
(1)	`ClientHttpRequestInterceptor` の実装クラスの bean 定義を行う。

Basic 認証用のリクエストヘッダ設定処理

サーバにアクセスするために Basic 認証用のリクエストヘッダを設定する必要がある場合は、以下のような bean 定義を行う。

bean 定義ファイル (applicationContext.xml) の定義例

```
<!-- (1) -->
<bean id="basicAuthInterceptor" class="org.springframework.http.client.support.
↪BasicAuthorizationInterceptor">
    <constructor-arg index="0" value="${api.auth.username}" /><!-- (2) -->
    <constructor-arg index="1" value="${api.auth.password}" /><!-- (3) -->
</bean>
```

項番	説明
(1)	`ClientHttpRequestInterceptor` インタフェースを実装した `BasicAuthorizationInterceptor` の bean 定義を行う。
(2)	コンストラクタの第一引数にユーザ名を設定する。
(3)	コンストラクタの第二引数にパスワードを設定する。

`ClientHttpRequestInterceptor` の適用

`RestTemplate` に作成した `ClientHttpRequestInterceptor` を適用する場合は、以下のような bean 定義を行う。

bean 定義ファイル (applicationContext.xml) の定義例

```
<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
    <property name="interceptors"><!-- (1) -->
        <list>
            <ref bean="basicAuthInterceptor" />
            <ref bean="loggingInterceptor" />
        </list>
    </property>
</bean>
```

項番	説明
(1)	interceptors プロパティに ClientHttpRequestInterceptor の bean をインジェクションする。 複数の bean をインジェクションした場合は、リストの先頭から順にチェーン実行される。 上記の例だと、BasicAuthorizationInterceptor -> LoggingInterceptor -> ClientHttpRequest の順番でリクエスト前の処理が実行される。(レスポンス後の処理は順番が逆転する)

非同期リクエスト

非同期リクエストを行う場合は、org.springframework.web.client.AsyncRestTemplate を使用する。

AsyncRestTemplate の bean 定義

AsyncRestTemplate の bean 定義を行う。

bean 定義ファイル (applicationContext.xml) の定義例

```
<bean id="asyncRestTemplate" class="org.springframework.web.client.
↪AsyncRestTemplate" /> <!-- (1) -->
```

項番	説明
(1)	AsyncRestTemplate をデフォルト設定のまま利用する場合は、デフォルトコンストラクタを使用して bean を登録する。 デフォルト設定の場合、AsyncRestTemplate の org.springframework.http.client.AsyncClientHttpRequestFactory には、org.springframework.core.task.AsyncListenableTaskExecutor として org.springframework.core.task.SimpleAsyncTaskExecutor が設定された SimpleClientHttpRequestFactory が設定される。

注釈: AsyncRestTemplate のカスタマイズ方法

デフォルトで設定される SimpleAsyncTaskExecutor は、スレッドプールを使わずにスレッドを生成しており、スレッドの同時実行数に制限は無い。そのため、同時に使用するスレッド数が非常に大きい場合は OutOfMemoryError が発生する可能性がある。

AsyncRestTemplate のコンストラクタに org.springframework.core.task.AsyncListenableTaskExecutor インターフェースの Bean を設定することで、スレッドプー

ル数の上限などを指定できる。下記は org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor を設定する例である。

```
<!-- (1) -->
<bean id="asyncTaskExecutor" class="org.springframework.scheduling.concurrent.
↪ThreadPoolTaskExecutor">
    <property name="maxPoolSize" value="100" />
</bean>

<!-- (2) -->
<bean id="asyncRestTemplate" class="org.springframework.web.client.
↪AsyncRestTemplate" >
    <constructor-arg index="0" ref="asyncTaskExecutor" />
</bean>
```

項番	説明
(1)	AsyncTaskExecutor の bean 定義を行う。 ThreadPoolTaskExecutor を使うことで、スレッドプールを使ったスレッド運用が行われる。 また、maxPoolSize プロパティを設定することで、スレッド数の制御が行える。
(2)	AsyncRestTemplate の bean 定義を行う。 ThreadPoolTaskExecutor を引数に指定するコンストラクタを使用して bean を登録する。

本ガイドラインでは、タスク実行処理をカスタマイズする実装例のみを紹介するが、AsyncRestTemplate は、HTTP 通信処理もカスタマイズ出来る。詳細は AsyncRestTemplate の Javadoc を参照されたい。

また、ThreadPoolTaskExecutor についても、スレッドプールサイズ以外のカスタマイズが出来る。詳細は ThreadPoolTaskExecutor の Javadoc を参照されたい。

非同期リクエストの実装

非同期リクエストの実装例

フィールド宣言部

```
@Inject
AsyncRestTemplate asyncRestTemplate;
```

メソッド内部

```
ListenableFuture<ResponseEntity<User>> responseEntity =
        asyncRestTemplate.getForEntity(uri, User.class); // (1)

responseEntity.addCallback(new ListenableFutureCallback<ResponseEntity<User>>() { //
↪ (2)
    @Override
    public void onSuccess(ResponseEntity<User> entity) {
        //...
    }

    @Override
    public void onFailure(Throwable t) {
      //...
    }
});
```

項番	説明
(1)	AsyncRestTemplate の各メソッドを使用して、非同期リクエストを送信する。 上記の実装例では、getForEntity メソッドを使用している。 戻り値は、org.springframework.util.concurrent.ListenableFuture にラップされた、ResponseEntity となっている。 各メソッドの使用方法は、RestTemplate と似たものとなっている。
(2)	ListenableFuture に org.springframework.util.concurrent.ListenableFutureCallback を登録して、レスポンスが返ってきた際の処理を実装する。 成功のレスポンスが返ってきた場合の処理は onSuccess メソッドに、エラーが発生した場合の処理は onFailure メソッドに実装する。

非同期リクエストの共通処理の実装

org.springframework.http.client.AsyncClientHttpRequestInterceptor を使用することで、サーバとの通信処理の前後に任意の処理を実行させることができる。

ここでは、ロギング処理の実装例を紹介する。

通信ログ出力の実装例

```
package com.example.restclient;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpRequest;
import org.springframework.http.client.AsyncClientHttpRequestExecution;
import org.springframework.http.client.AsyncClientHttpRequestInterceptor;
import org.springframework.http.client.ClientHttpResponse;
import org.springframework.util.concurrent.ListenableFuture;
import org.springframework.util.concurrent.ListenableFutureCallback;

public class AsyncLoggingInterceptor implements
                                    AsyncClientHttpRequestInterceptor { // (1)
    private static final Logger logger = LoggerFactory.getLogger(
            AsyncLoggingInterceptor.class);

    @Override
    public ListenableFuture<ClientHttpResponse> intercept(HttpRequest request,
            byte[] body,
            AsyncClientHttpRequestExecution execution) throws IOException {
        // (2)
        if (log.isInfoEnabled()) {
            String requestBody = new String(body, StandardCharsets.UTF_8);

            log.info("Request Header {}", request.getHeaders());
            log.info("Request Body {}", requestBody);
        }

        // (3)
        ListenableFuture<ClientHttpResponse> future = execution.executeAsync(
                request, body);
        if (log.isInfoEnabled()) {
            // (4)
            future.addCallback(new ListenableFutureCallback<ClientHttpResponse>() {

                @Override
                public void onSuccess(ClientHttpResponse response) {
                    try {
                        log.info("Response Header {}", response
                                .getHeaders());
                        log.info("Response Status Code {}", response
                                .getStatusCode());
                    } catch (IOException e) {
                        log.warn("I/O Error", e);
                    }
                }

                @Override
                public void onFailure(Throwable e) {
                    log.info("Communication Error", e);
                }
            });
        }

        return future; // (5)
```

```
    }
}
```

項番	説明
(1)	AsyncClientHttpRequestInterceptor インタフェースを実装する。
(2)	非同期リクエストを送信する前に実行する処理を実装する。 上記の実装例では、リクエストヘッダとリクエストボディの内容をログに出力している。
(3)	intercept メソッドの引数として受け取った AsyncClientHttpRequestExecution の executeAsync メソッドを使用して、非同期リクエストを送信する。
(4)	(3) で受け取った ListenableFuture に org.springframework.util.concurrent.ListenableFutureCallback を登録して、レスポンスが返ってきた際の処理を実装する。 レスポンスが返却された場合、onSuccess メソッドが呼び出される。 また、非同期リクエスト時に例外が発生した場合、onFailure メソッドが呼び出される。以下に具体例を示す。 • 指定したホストに接続できない（ConnectException） • レスポンスデータの読み込みタイムアウトが発生した（SocketTimeoutException）
(5)	(3) で受け取った ListenableFuture をリターンする。

bean 定義ファイル (applicationContext.xml) の定義例

```
<!-- (1) -->
<bean id="asyncLoggingInterceptor" class="com.example.restclient.
↪AsyncLoggingInterceptor" />

<bean id="asyncRestTemplate" class="org.springframework.web.client.
↪AsyncRestTemplate">
    <property name="interceptors"><!-- (2) -->
        <list>
            <ref bean="asyncLoggingInterceptor" />
        </list>
    </property>
</bean>
```

項番	説明
(1)	`AsyncClientHttpRequestInterceptor` の実装クラスの bean 定義を行う。
(2)	`interceptors` プロパティに `AsyncClientHttpRequestInterceptor` の bean をインジェクションする。 複数の bean をインジェクションした場合は、`RestTemplate` と同様にリストの先頭から順にチェーン実行される。

5.2.4 Appendix

HTTP Proxy サーバの設定方法

サーバへアクセスする際に HTTP Proxy サーバを経由する必要がある場合は、システムプロパティや JVM 起動引数、または `RestTemplate` の Bean 定義にて HTTP Proxy サーバの設定が必要となる。システムプロパティや JVM 起動引数に設定した場合、アプリケーション全体に影響を与えてしまうため、`RestTemplate` 毎に HTTP Proxy サーバの設定を行う例を紹介する。

`RestTemplate` 毎の HTTP Proxy サーバの設定は、`ClientHttpRequestFactory` インタフェースのデフォルト実装である `SimpleClientHttpRequestFactory` に付与することが可能である。ただし `SimpleClientHttpRequestFactory` では資格情報を設定することはできないため、Proxy 認証を行う場合は `HttpComponentsClientHttpRequestFactory` を使用する。`HttpComponentsClientHttpRequestFactory` は、Apache HttpComponents HttpClient を用いてリクエストを生成する `ClientHttpRequestFactory` インタフェースの実装クラスである。

SimpleClientHttpRequestFactory を使用した HTTP Proxy サーバの設定方法

資格情報が不要な HTTP Proxy サーバの接続先の指定については、`RestTemplate` がデフォルトで使用する `SimpleClientHttpRequestFactory` で指定することが可能である。

Bean 定義ファイル

```
<!-- (1) -->
<bean id="inetSocketAddress" class="java.net.InetSocketAddress" >
    <constructor-arg index="0" value="${rscl.http.proxyHost}" />    <!-- (2) -->
    <constructor-arg index="1" value="${rscl.http.proxyPort}" />    <!-- (3) -->
</bean>

<!-- (4) -->
<bean id="simpleClientRestTemplate" class="org.springframework.web.client.
↪RestTemplate" >
    <constructor-arg>
        <!-- (5) -->
        <bean class="org.springframework.http.client.SimpleClientHttpRequestFactory
↪">
            <!-- (6) -->
            <property name="proxy">
                <bean class="java.net.Proxy" >
                    <!-- (7) -->
                    <constructor-arg index="0">
                        <util:constant static-field="java.net.Proxy.Type.HTTP"/>
                    </constructor-arg>
                    <constructor-arg index="1" ref="inetSocketAddress"/>
                </bean>
            </property>
        </bean>
    </constructor-arg>
</bean>
```

項番	説明
(1)	`java.net.InetSocketAddress` に HTTP Proxy サーバの設定を行う。
(2)	`InetSocketAddress` のコンストラクタの第一引数に、プロパティファイルに設定されたキー `rscl.http.proxyHost` の値を HTTP Proxy サーバのホスト名として設定する。
(3)	`InetSocketAddress` のコンストラクタの第二引数に、プロパティファイルに設定されたキー `rscl.http.proxyPort` の値を HTTP Proxy サーバのポート番号として設定する。
(4)	`RestTemplate` の Bean 定義を行う。
(5)	`RestTemplate` のコンストラクタの引数に、`SimpleClientHttpRequestFactory` を設定する。
(6)	`SimpleClientHttpRequestFactory` の `proxy` プロパティに `java.net.Proxy` を設定する。
(7)	`Proxy` のコンストラクタの引数に、`java.net.Proxy.Type.HTTP` と生成した `InetSocketAddress` を設定する。

HttpComponentsClientHttpRequestFactory を使用した HTTP Proxy サーバの設定方法

HTTP Proxy サーバの指定方法

資格情報が必要な HTTP Proxy サーバの接続先の指定は、`RestTemplate` に対して、`HttpComponentsClientHttpRequestFactory` を使用し指定する。

pom.xml

```
<!-- (1) -->
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
```

```
</dependency>
```

項番	説明
(1)	HttpComponentsClientHttpRequestFactory 内で使用する Apache HTTP Client を使用するために、Apache HttpComponents Client を pom.xml の依存ライブラリに追加する。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

Bean 定義ファイル

```
<!-- (1) -->
<bean id="proxyHttpClientBuilder" class="org.apache.http.impl.client.
↪HttpClientBuilder" factory-method="create" >
    <!-- (2) -->
    <property name="proxy">
        <bean class="org.apache.http.HttpHost" >
            <constructor-arg index="0" value="${rscl.http.proxyHost}" />    <!--␣
↪(3) -->
            <constructor-arg index="1" value="${rscl.http.proxyPort}" />    <!--␣
↪(4) -->
        </bean>
    </property>
</bean>

<!-- (5) -->
<bean id="proxyRestTemplate" class="org.springframework.web.client.RestTemplate" >
    <constructor-arg>
        <!-- (6) -->
        <bean class="org.springframework.http.client.
↪HttpComponentsClientHttpRequestFactory">
            <!-- (7) -->
            <constructor-arg>
                <bean factory-bean="proxyHttpClientBuilder" factory-method="build"␣
↪/>
            </constructor-arg>
        </bean>
    </constructor-arg>
</bean>
```

項番	説明
(1)	`org.apache.http.impl.client.HttpClientBuilder` を使用し、`org.apache.http.client.HttpClient` の設定を行う。
(2)	`HttpClientBuilder` の `proxy` プロパティに、HTTP Proxy サーバの設定を行った `org.apache.http.HttpHost` を設定する。
(3)	`HttpHost` のコンストラクタの第一引数に、プロパティファイルに設定されたキー `rscl.http.proxyHost` の値を HTTP Proxy サーバのホスト名として設定する。
(4)	`HttpHost` のコンストラクタの第二引数に、プロパティファイルに設定されたキー `rscl.http.proxyPort` の値を HTTP Proxy サーバのポート番号として設定する。
(5)	`RestTemplate` の Bean 定義を行う。
(6)	`RestTemplate` のコンストラクタの引数に、`HttpComponentsClientHttpRequestFactory` を設定する。
(7)	`HttpComponentsClientHttpRequestFactory` のコンストラクタの引数に、`HttpClientBuilder` から生成した `HttpClient` を設定する。

HTTP Proxy サーバの資格情報の指定方法

HTTP Proxy サーバにアクセスする際に資格情報 (ユーザ名とパスワード) が必要な場合は、`org.apache.http.impl.client.BasicCredentialsProvider` を使用し資格情報を設定する。

`BasicCredentialsProvider` の `setCredentials` メソッドが引数を 2 つ取るため、セッターインジェクションを利用して Bean を生成することができない。このため、`org.springframework.beans.factory.FactoryBean` を利用して Bean を生成する。

FactoryBean クラス

```
import org.apache.http.auth.AuthScope;
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.impl.client.BasicCredentialsProvider;
import org.springframework.beans.factory.FactoryBean;
import org.springframework.beans.factory.annotation.Value;

// (1)
public class BasicCredentialsProviderFactoryBean implements FactoryBean
↪<BasicCredentialsProvider> {

    // (2)
    @Value("${rscl.http.proxyHost}")
    String host;

    // (3)
    @Value("${rscl.http.proxyPort}")
    int port;

    // (4)
    @Value("${rscl.http.proxyUserName}")
    String userName;

    // (5)
    @Value("${rscl.http.proxyPassword}")
    String password;

    @Override
    public BasicCredentialsProvider getObject() throws Exception {

        // (6)
        AuthScope authScope = new AuthScope(this.host, this.port);

        // (7)
        UsernamePasswordCredentials usernamePasswordCredentials =
                new UsernamePasswordCredentials(this.userName, this.password);

        // (8)
        BasicCredentialsProvider credentialsProvider = new␣
↪BasicCredentialsProvider();
        credentialsProvider.setCredentials(authScope, usernamePasswordCredentials);

        return credentialsProvider;
    }

    @Override
    public Class<?> getObjectType() {
        return BasicCredentialsProvider.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
```

```
}
```

項番	説明
(1)	`org.springframework.beans.factory.FactoryBean` を実装した `BasicCredentialsProviderFactoryBean` クラスを定義する。 Bean の型に `BasicCredentialsProvider` を設定する。
(2)	プロパティファイルに設定されたキー `rscl.http.proxyHost` の値を HTTP Proxy サーバのホスト名として、インスタンス変数に設定する。
(3)	プロパティファイルに設定されたキー `rscl.http.proxyPort` の値を HTTP Proxy サーバのポート番号として、インスタンス変数に設定する。
(4)	プロパティファイルに設定されたキー `rscl.http.proxyUserName` の値を HTTP Proxy サーバのユーザ名として、インスタンス変数に設定する。
(5)	プロパティファイルに設定されたキー `rscl.http.proxyPassword` の値を HTTP Proxy サーバのパスワードとして、インスタンス変数に設定する。
(6)	`org.apache.http.auth.AuthScope` を作成し資格情報のスコープを設定する。この例は、HTTP Proxy サーバのホスト名とポート番号を指定したものである。その他の設定方法については、AuthScope (Apache HttpClient API) を参照されたい。
(7)	`org.apache.http.auth.UsernamePasswordCredentials` を作成し資格情報を設定する。
(8)	`org.apache.http.impl.client.BasicCredentialsProvider` を作成し、`setCredentials` メソッドを使用し、資格情報のスコープと資格情報を設定する。

Bean 定義ファイル

```
<bean id="proxyHttpClientBuilder" class="org.apache.http.impl.client.
↪HttpClientBuilder" factory-method="create">
    <!-- (1) -->
    <property name="defaultCredentialsProvider">
        <bean class="com.example.restclient.BasicCredentialsProviderFactoryBean" />
    </property>
    <property name="proxy">
        <bean id="proxyHost" class="org.apache.http.HttpHost">
            <constructor-arg index="0" value="${rscl.http.proxyHost}" />
            <constructor-arg index="1" value="${rscl.http.proxyPort}" />
        </bean>
    </property>
</bean>

<bean id="proxyRestTemplate" class="org.springframework.web.client.RestTemplate">
    <constructor-arg>
        <bean class="org.springframework.http.client.
↪HttpComponentsClientHttpRequestFactory">
            <constructor-arg>
                <bean factory-bean="proxyHttpClientBuilder" factory-method="build"␣
↪/>
            </constructor-arg>
        </bean>
    </constructor-arg>
</bean>
```

項番	説明
(1)	HttpClientBuilder の defaultCredentialsProvider プロパティに、BasicCredentialsProvider を設定する。 BasicCredentialsProvider は、FactoryBean を実装した BasicCredentialsProviderFactoryBean を使用し Bean を作成する。

JSON で JSR-310 Date and Time API を使う場合の設定

リソースを表現する JavaBean(Resource クラス) のプロパティとして JSR-310 Date and Time API を使用する場合の設定は「*JSR-310 Date and Time API / Joda Time を使う場合の設定* 」を参照されたい。

5.3 SOAP Web Service（サーバ/クライアント）

5.3.1 Overview

本節では、SOAP Web Service の基本的な概念と JAX-WS を使用した SOAP サーバ、クライアント双方の開発について説明する。

実装に対する具体的な説明については、

- 「*How to use*」

  JAX-WS を使用した SOAP Web Service のアプリケーション構成や API の実装方法について説明している。

を参照されたい。

SOAP とは

SOAP とは、XML で記述されたメッセージをコンピュータ間で送受信を行うためのプロトコルである。

もともとは「**S**imple **O**bject **A**ccess **P**rotocol」の略であった。

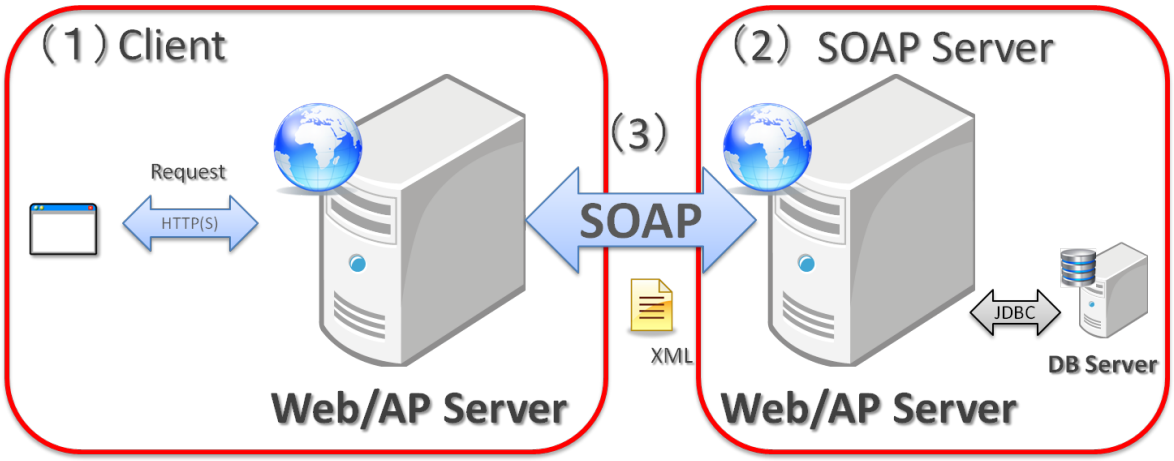
しかし現在では、「SOAP」はなにかの略ではなく、固有名詞であると W3C は宣言している。

W3C による SOAP1.1、SOAP1.2 の仕様は W3C により定義されている。

詳細は、W3C -SOAP Specifications-を参照されたい。

本ガイドラインでは、以下の図のような構成での SOAP Web Service を行う場合を想定して説明する。

ただし、下記の構成以外での SOAP Web Service の場合にも応用可能である。（例：クライアントがバッチの場合など）



項番	説明
(1)	クライアントは、別の SOAP サーバへの通信を行う Web アプリケーションを想定している。 クライアントと呼んでいるが Web アプリケーション想定なので注意が必要である。
(2)	SOAP サーバは、Web サービスを公開し、クライアントからの SOAP Web Service による XML を受信して処理を行う。データベースなどにアクセスを行い、業務処理を行うことを想定している。
(3)	SOAP Web Service では XML を使用して情報のやり取りを行う。 今回の想定では、SOAP サーバ、クライアントどちらも Java である想定としているが、他のプラットフォームでも問題なく通信可能である。

JAX-WS とは

JAX-WS とは、「**J**ava **A**PI for **X**ML-Based **W**eb **S**ervices」の略であり、SOAP などを使った Web サービスを扱うための Java 標準 API である。

JAX-WS を用いることで、Java のオブジェクトを SOAP の仕様に沿った XML に変換して送信することが可能である。

そのため、SOAP Web Service としては、XML でやり取りが行われるものの、利用者は、XML の構造をあまり意識せずデータを扱うことができる。

Oracle WebLogic Server や JBoss Enterprise Application Platform など主要な Java EE サーバは JAX-WS 実装をサーバ側で有しており、特別なライブラリを追加せずにその機能を使用して簡単に Web サービスを公開することができる。

ただし、Tomcat は、JAX-WS を実装していないため、使用する際には別途 JAX-WS 実装ライブラリを追加する必要がある。

詳細は、「*Tomcat 上での Web サービス開発*」を参照されたい。

JAX-WS を利用した Web サービスの開発について

TERASOLUNA Server Framework for Java (5.x) では、AP サーバの JAX-WS 実装と Spring の機能を利用して Web サービスの開発を行うことを推奨する。

SOAP サーバ、クライアントどちらにおいても、通常の Web アプリケーション同様に、ブランクプロジェクト内の web プロジェクトから作成した WAR ファイルを AP サーバにデプロイすることで、SOAP Web Service を実現することができる。

注釈: AP サーバの JAX-WS 実装によって、JAX-WS 仕様への対応状況や実際の Web サービスの動作やが異なる場合があり、必ずしも本ガイドラインの実装が全ての AP サーバで同様に動作するわけではない。

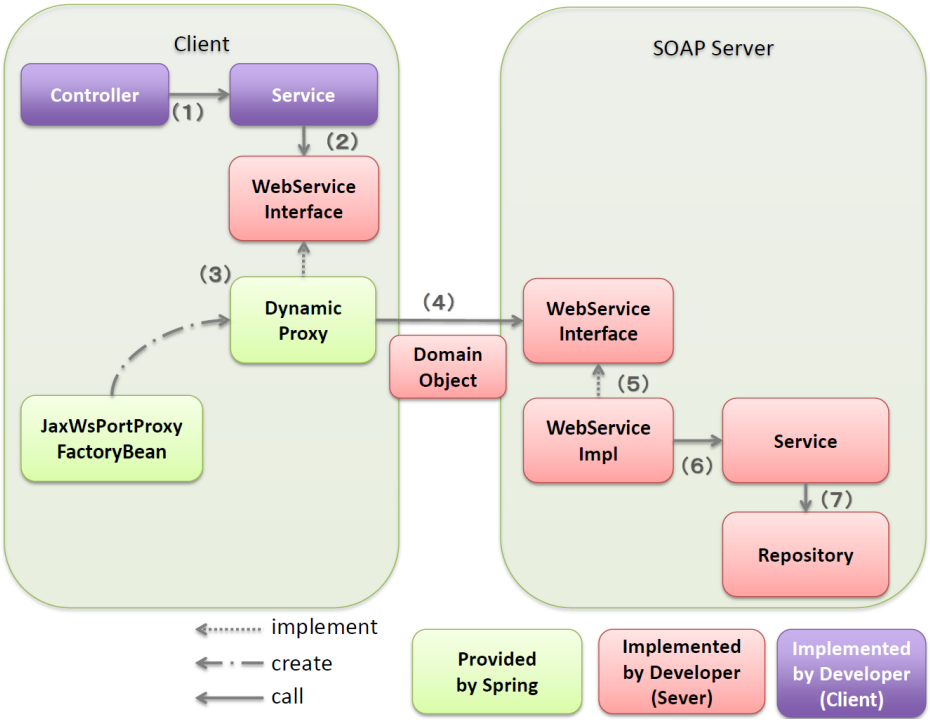
開発を始める前には必ず、「アプリケーションの設定」の Note から使用する AP サーバのマニュアルを確認されたい。

Spring Framework の JAX-WS 連携機能について

Spring Framework は JAX-WS の連携機能をサポートしており、その機能を使用することで JAX-WS を利用して作成された SOAP Web Service に接続するアプリケーションを簡単に実装することができる。

以下はその機能を用いた、推奨アクセスフローの概要である。

ここでは SOAP のクライアント (図左) である Web アプリケーションが SOAP サーバ (図右) にアクセスすることを前提としている。



項番	説明
(1)	[クライアント] Controller が Service を呼び出す。 通常の呼び出しと変更点は特にない。
(2)	[クライアント] Service が SOAP サーバ提供側で用意した WebService インターフェースを呼び出す。 この図では、Service が WebService インターフェースを呼び出しているが、要件に応じて Controller から直接 WebService インターフェースを呼び出してもよい。
(3)	[クライアント] WebService インターフェースが呼び出されると実体として「動的プロキシ (Dynamic Proxy)」(以下「プロキシ」) が呼び出される。 このプロキシは `org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean` が生成した WebService インターフェースの実装クラスである。 この実装クラスが Service にインジェクションされ、Service は WebService インターフェースのメソッドを呼び出すだけで、SOAP Web Service を利用した処理を行うことができる。
(4)	プロキシが、SOAP サーバの WebService インターフェースを呼び出す。 SOAP サーバとクライアントでの値のやり取りは Domain Object を使用して行う。 --- **注釈:** 厳密には、SOAP サーバとクライアントは XML を使用して通信を行っている。送信時、および受信時には JAXB を使用して、Domain Object と XML の相互変換が行われているが、SOAP Web Service 作成者は XML をあまり意識せず、開発を行うことができるようになっている。 ---

Continued on next page

表 5.14 – continued from previous page

項番	説明
(5)	[サーバ] WebService インターフェースが呼び出されると実体として WebService 実装クラスが呼び出される。 SOAP サーバでは、WebService インターフェースの実装クラスとして WebService 実装クラスを用意する。 この WebService 実装クラスは、`org.springframework.web.context.support.SpringBeanAutowiringSupport` を継承することで、Spring の DI コンテナ上の Bean を`@Autowired` などでインジェクションすることができる。 --- **注釈:** WebService 実装クラスは、Spring Framework が提供する DispatcherServlet 上ではなく、AP サーバの JAX-WS エンジンが実装するサーブレットとして動作する。このためガイドラインのアプリケーション層の実装に記載している実装方法とは以下のような違いがあることに注意されたい。 • WebService 実装クラスが Spring の DI コンテナ上で管理されないため、例えば Spring の AOP による横断的な処理をかけることができない。（ただし、JAX-WS 実装として Apache CXF を利用する場合には Spring の DI コンテナ上で管理される） • Spring の Controller クラスではないため、`@ControllerAdvice` や `@ExceptionHandler` などが適用されない。 また、SOAP サーバは、`@Inject` ではなく、`@Autowired` でインジェクションすることを推奨する。 `@Inject` の場合、Java EE サーバが提供する DI 機能で使用されるため、Java EE サーバの DI コンテナに存在しないとエラーになってしまう。 上記に対して、`@Autowired` であれば Spring の DI 機能のみで使用されるため、意図せず Java EE サーバの DI 機能でエラーになるのを防止することができる。 ---
(6)	[サーバ] WebService 実装クラスでは、業務処理を行う Service を呼び出す。
(7)	[サーバ] Service では、Repository などを使用して業務処理を実行する。 通常の呼び出しと変更点は特にない。

注釈: Spring では、ドキュメントドリブンで Web サービスを開発する Spring Web Services が提供されているが、ここでは扱わない。詳細は Spring Web Services を参照されたい。

注釈: Spring での JAX-WS 実装の詳細は、Spring Framework Reference Documentation -Remoting and web services using Spring(Web services)-を参照されたい。

JAX-WS を利用した Web サービスのモジュールの構成

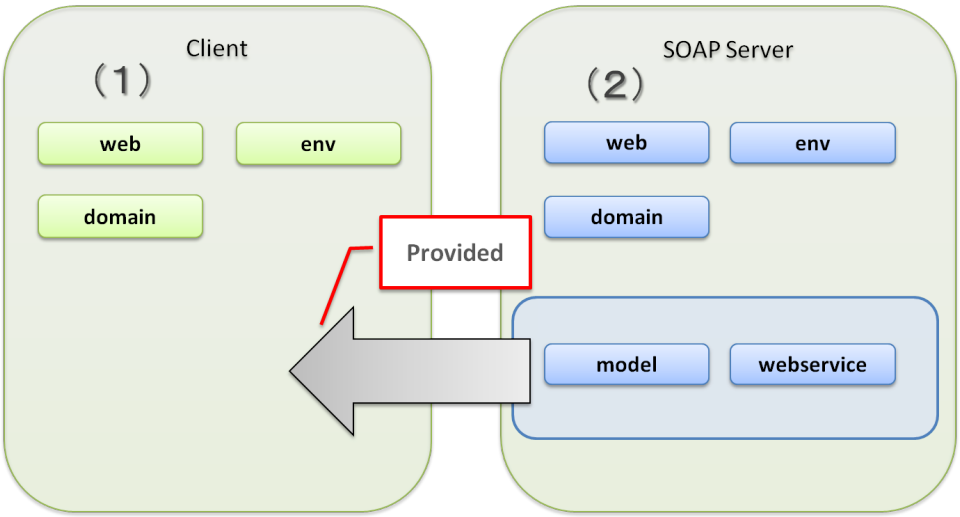
JAX-WS を利用した Web サービスを作成する場合、既存のブランクプロジェクトとは別に以下 2 つのプロジェクトを追加することを推奨する。

- model プロジェクト
- webservice プロジェクト

model プロジェクトは、Web サービスの引数や返り値に使用する Domain Object を格納する。
webservice プロジェクトは、Web サービスを呼び出すインターフェースを格納する。
この 2 つは SOAP サーバからクライアントに配布する必要があるクラスのみ格納するプロジェクトである。
配布する範囲を明確に識別するため、別プロジェクトにすることを推奨している。

本ガイドラインでは、マルチプロジェクトで以下のような構成を用いる。

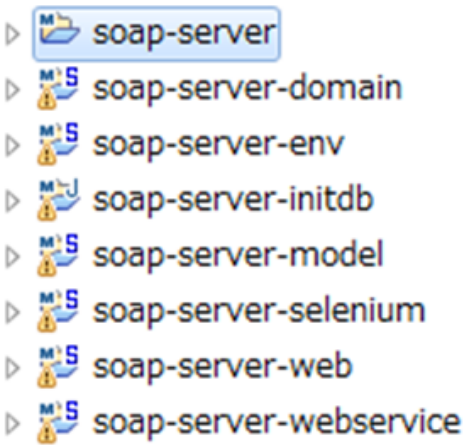
ここでもクライアントは Web アプリケーションであることを前提とするが、デスクトップアプリケーションやコマンドラインインターフェースから呼び出す場合も基本的な考え方は同じである。



項番	説明
(1)	クライアントを作成する場合、従来のマルチプロジェクトに SOAP サーバから提供される model プロジェクトと webservice プロジェクトを追加する。 ここではサーバとクライアントをともに開発することを前提としている。 これらのプロジェクトの詳細については「*SOAP サーバの作成*」で説明する。 追加方法については「*SOAP サーバ用にプロジェクトの設定を変更する*」を参照されたい。 サーバとクライアントの開発が別々で、model プロジェクトと webservice プロジェクトが提供されない場合、もしくは Java 以外で SOAP サーバが作成されている場合には、model プロジェクト内の Domain Object と webservice プロジェクト内の Web サービスインターフェースを自分で作成する必要がある。 wsimport を使用することで、WSDL から簡単に Domain Object と Web サービスインターフェースを作成することができる。 詳細については「*wsimport について*」を参照されたい。
(2)	SOAP サーバを作成する場合、従来のマルチプロジェクトに追加して model プロジェクトと webservice プロジェクトを追加する。 クライアントにこれら 2 つのプロジェクトを公開する。 クライアントへの model プロジェクト、webservice プロジェクトの公開方法は、Maven の依存関係への追加を想定している。

結果として、プロジェクトは次のような構成となる。

以下は、SOAP サーバのプロジェクト構成である。



以下は、クライアントのプロジェクト構成である。



Web サービスとして公開される URL

SOAP Web Service を作成すると WSDL（**W**eb **S**ervices **D**escription **L**anguage）という Web サービスのインターフェース定義が公開され、クライアントはこの定義をもとに SOAP Web Service を実行する。

WSDL の詳細は、W3C -Web Services Description Language (WSDL)-を参照されたい。

WSDL 内には、Web サービス実行時のアクセス URL やメソッド名、引数、戻り値などが定義される。

本ガイドラインの通りに SOAP Web Service を作成すると、以下の URL で WSDL が公開される。

クライアントではこの URL を指定する必要がある。

- *http://AAA.BBB.CCC.DDD:XXXX/*コンテキストルート*/Web* サービス名*?wsdl*

WSDL 内で定義されるエンドポイントアドレスは以下の URL である。

- *http://AAA.BBB.CCC.DDD:XXXX/*コンテキストルート*/Web* サービス名

注釈: 本ガイドラインでは、マルチプロジェクト構成の web プロジェクトを WAR ファイル化して、AP サーバにデプロイする前提である。その場合、コンテキストルートは基本的に、[server projectName]-web となる。ただし、AP サーバによって異なるので注意すること。

注釈: 本ガイドラインでは、SOAP サーバ、クライアントともに Web アプリケーションとして公開する前提であるため、クライアントでは WSDL の URL を指定している。URL ではなく、WSDL をファイルとして用意してクライアントを作成することも可能である。詳細は、*Web* サービス クライアントの実装を参照されたい。

警告: 本ガイドラインでは、AP サーバ（Tomcat の場合は使用するライブラリ）でコンテキストルートのマッピングを切り替え以下のような URL でアクセスするように設定している。

- *http://AAA.BBB.CCC.DDD:XXXX/[server projectName]-web/ws/TodoWebService?wsdl*

このコンテキストルート直下ではない URL に Web サービスをマッピングさせる方法は、AP サーバごとに異なる。詳細は以下を参照してほしい。

項番	AP サーバ名	説明
(1)	Apache Tomcat	*Tomcat* 上での *Web* サービス開発
(2)	Oracle WebLogic Server	TBD
(3)	JBoss Enterprise Application Platform	TBD

5.3.2 How to use

本節では、SOAP Web Service の具体的な作成方法について説明する。

SOAP サーバの作成

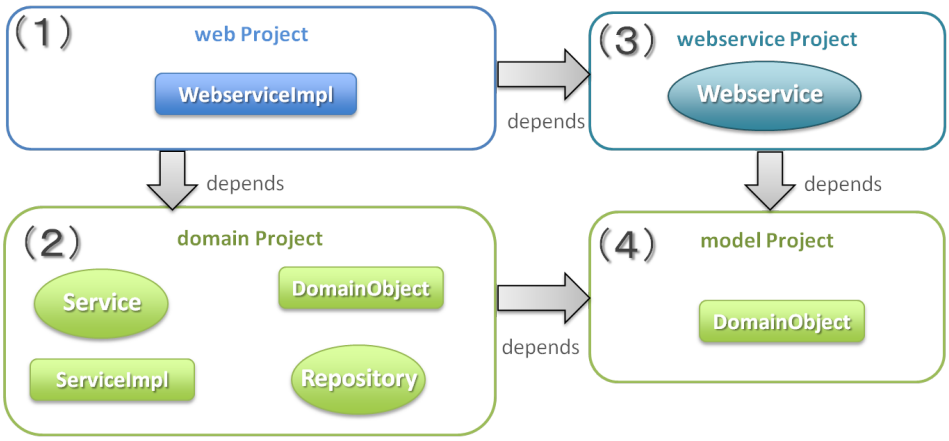
プロジェクトの構成

各プロジェクトの依存関係

「*JAX-WS を利用した Web サービスの開発について*」で述べたとおり、model プロジェクトと webservice プロジェクトを追加する。

追加方法は「*SOAP サーバ用にプロジェクトの設定を変更する*」を参照されたい。

またそれに伴い、既存のプロジェクトに依存関係を追加することが必要となる。



項番	プロジェクト名	説明
(1)	web プロジェクト	Web サービス実装クラスを配置する。
(2)	domain プロジェクト	WebService の実装クラスから呼び出される Service を配置する。 その他、Repository などは従来と同じである。
(3)	webservice プロジェクト	公開する WebService のインターフェースをここに配置する。 クライアントはこのインターフェースを使用して Web サービスを実行する。
(4)	model プロジェクト	ドメイン層に属するクラスのうち、SOAP Web Service で使用するクラスのみをここに配置する。 クライアントからの入力値や返却結果はこのプロジェクト内のクラスを使用する。

アプリケーションの設定

Web サービスを公開する際の初期設定

AP サーバとして Tomcat を使用する場合は、「*Tomcat 上での Web サービス開発*」を実施する必要がある。
その他、AP サーバによって Web サービス公開の方法は違うので、詳細は各 AP サーバのマニュアルを参照されたい。

注釈: 以下、参考資料として、AP サーバのマニュアルを記述しておく。必ず、使用するバージョンとあっているか確認してから参照すること。

Oracle WebLogic Server 12.2.1: Oracle(R) Fusion Middleware Understanding WebLogic Web Services for Oracle WebLogic Server Features and Standards Supported by WebLogic Web Services

JBoss Enterprise Application Platform 7.2: DEVELOPING JAX-WS WEB SERVICES

JBoss Enterprise Application Platform 6.4: DEVELOPMENT GUIDE JAX-WS WEB SERVICES

WebSphere Application Server 9.0: IBM Knowledge Center - Web services

パッケージのコンポーネントスキャン設定

Web サービスで使用するコンポーネントをスキャンするため、`[server projectName]-ws.xml` を作成し、コンポーネントスキャンの定義を行い、Web サービスにインジェクションできるようにする。

[server projectName]-web/src/main/resources/META-INF/spring/[server projectName]-ws.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.
↪w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         https://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         https://www.springframework.org/schema/context/spring-context.xsd">
    <!-- (1) -->
    <context:component-scan base-package="com.example.ws" />
</beans>
```

項番	説明
(1)	Web サービスで使用するコンポーネントが格納されているパッケージを指定する。

[server projectName]-web/src/main/webapp/WEB-INF/web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <!-- Root ApplicationContext -->
    <!-- (1) -->
    <param-value>
        classpath*:META-INF/spring/applicationContext.xml
```

```
        classpath*:META-INF/spring/spring-security.xml
        classpath*:META-INF/spring/[server projectName]-ws.xml
    </param-value>
</context-param>
```

項番	説明
(1)	[server projectName]-ws.xml をルート ApplicationContext 生成時の読み込み対象に加える。

入力チェックを行うための定義

入力チェックにはメソッドバリデーションを使用するため、以下の定義を追加する。

入力チェックの詳細は 入力チェックの実装を参照されたい。

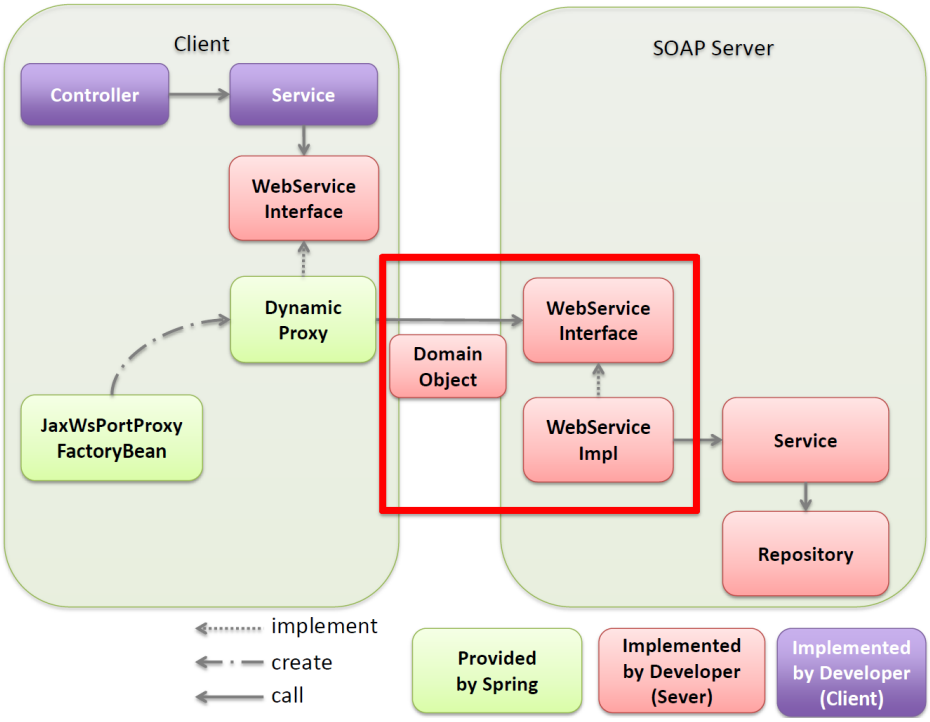
[server projectName]-web/src/main/resources/META-INF/spring/applicationContext.xml

```
<bean class="org.springframework.validation.beanvalidation.
↪MethodValidationPostProcessor">
    <property name="validator" ref="validator" />
</bean>

<bean id="validator" class="org.springframework.validation.beanvalidation.
↪LocalValidatorFactoryBean" />
```

Web サービスの実装

以下の作成を行う。

- Domain Object の作成
- WebService インターフェイスの作成
- WebService 実装クラスの作成



Domain Object の作成

model プロジェクト内に、Web サービスの引数や返り値に使用する Domain Object を作成する。

`java.io.Serializable` インターフェースを実装した一般の JavaBean と特に変わりはない。

[server projectName]-model/src/main/java/com/example/domain/model/Todo.java

```
package com.example.domain.model;

import java.io.Serializable;
import java.util.Date;

public class Todo implements Serializable {

    private String todoId;

    private String title;

    private String description;

    private boolean finished;

    private Date createdAt;
```

```
    // omitted setter and getter

}
```

WebService インターフェイスの作成

webservice プロジェクト内に Web サービスを呼び出すインターフェースを作成する。

[server projectName]-webservice/src/main/java/com/example/ws/todo/TodoWebService.java

```
package com.example.ws.todo;

import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

import com.example.domain.model.Todo;
import com.example.ws.webfault.WebFaultException;

@WebService(targetNamespace = "http://example.com/todo") // (1)
public interface TodoWebService {

    @WebMethod // (2)
    @WebResult(name = "todo") // (3)
    Todo getTodo(@WebParam(name = "todoId") /* (4) */ String todoId) throws␣
↪WebFaultException;

}
```

項番	説明
(1)	`@WebService` を付けることで、WebService インターフェースであることを宣言する。 `targetNamespace` 属性には、名前空間を定義するが、これは作成する Web サービスのパッケージ名と合わせることを推奨する。 **警告:** `targetNamespace` 属性の値は一意にする必要がある。そのため、ガイドライン上のソースを流用する場合は必ず変更すること。 **注釈:** `targetNamespace` 属性の値は WSDL 上に定義され、この Web サービスの名前空間を決定し、一意に特定するために使用される。
(2)	Web サービスのメソッドとして公開するメソッドに`@WebMethod` を付ける。 このアノテーションを付けることにより、WSDL 上にメソッドが公開され、外部から使用することが可能になる。
(3)	返り値に`@WebResult` を付け、名前を `name` 属性に指定する。返り値がない場合は不要。 このアノテーションを付けることにより、WSDL 上に返り値として公開される。
(4)	引数に`@WebParam` を付け、名前を `name` 属性に指定する。 このアノテーションを付けることにより、WSDL 上に引数が公開され、外部から呼び出すときの必要なパラメータとして定義される。 `WebFaultException` の詳細は「例外ハンドリングの実装」を参照されたい。

注釈: パッケージ名および、ネームスペースの付け方について

パッケージ名が以下のような形式になっている場合

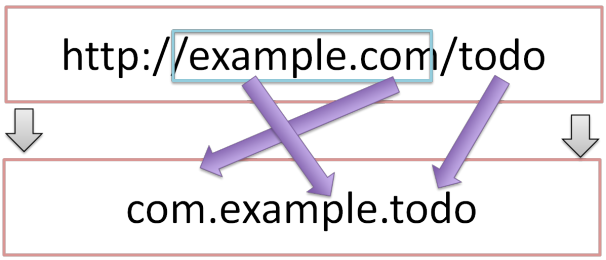
- 【ドメイン】.【アプリケーション名 (システム名)】.ws.【ユースケース名】

本ガイドラインでは、以下のようなネームスペースにすることを推奨する。

- http://【ドメイン】/【アプリケーション名 (システム名)】

注釈: ネームスペースとパッケージ名の関係

ドメインを com.example、アプリケーション名を todo とした場合、Namespace は以下のような Java のパッケージと紐づけられる。



仕様ではないが、Namespace とパッケージの命名について、XML Namespace Mapping(Red Hat JBoss Fuse) にまとまっている。

WebService 実装クラスの作成

web プロジェクト内に WebService インターフェースの実装クラスを作成する。

[server projectName]-web/src/main/java/com/example/ws/todo/TodoWebServiceImpl.java

```
package com.example.ws.todo;

import java.util.List;

import javax.jws.HandlerChain;
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.soap.SOAPBinding;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.context.support.SpringBeanAutowiringSupport;

import com.example.domain.model.Todo;
import com.example.domain.service.TodoService;
import com.example.ws.webfault.WebFaultException;
import com.example.ws.exception.WsExceptionHandler;
import com.example.ws.todo.TodoWebService;


@WebService(
        portName = "TodoWebPort",
        serviceName = "TodoWebService",
        targetNamespace = "http://example.com/todo",
```

```
        endpointInterface = "com.example.ws.todo.TodoWebService") // (1)
@BindingType(SOAPBinding.SOAP12HTTP_BINDING) // (2)
public class TodoWebServiceImpl extends SpringBeanAutowiringSupport implements␣
↪TodoWebService { // (3)

    @Autowired // (4)
    TodoService todoService;

    @Override // (5)
    public Todo getTodo(String todoId) throws WebFaultException {
        return todoService.getTodo(todoId);
    }

}
```

項番	説明
(1)	`@WebService` を付けることで、WebService の実装クラスであることを宣言する。 `portName` 属性は、WSDL 上のポート名として公開される。 `serviceName` 属性は、WSDL 上のサービス名として公開される。 `targetNamespace` 属性は、WSDL 上で使用されるネームスペース。 `endpointInterface` 属性は、このクラスが実装している Web サービスのインターフェース名を定義する。 **注釈:** `TodoWebService` インターフェースでは、`@WebService` の属性として `portName` 属性, `serviceName` 属性, `endpointInterface` 属性を設定してはいけない。これは、このインターフェースは WSDL 上の `portType` 要素に対応しており、Web サービスの内容を記述する要素ではないためである。
(2)	`@BindingType` を付けることで、バインディングの方式を設定する。 `SOAPBinding.SOAP12HTTP_BINDING` を定義すると SOAP1.2 でのバインディングとなる。 何もつけない場合は、SOAP1.1 でのバインディングとなる。 **注釈:** 使用する AP サーバの JAX-WS 実装により、バインディング方式で挙動が異なる場合があるため注意すること。 たとえば、WebSphere Application Server の特定のバージョンでは SOAP1.2 でのバインディングの場合に WSDL が自動生成されない。詳細については IBM Knowledge Center - Using annotations to create web services を参照されたい。

Continued on next page

表 5.16 – continued from previous page

項番	説明
(3)	先ほど作成した `TodoWebService` インターフェースを実装する。 `org.springframework.web.context.support.SpringBeanAutowiringSupport` を継承することで、Spring の Bean を DI できるようにする。
(4)	Service をインジェクションする。 通常の Controller で Service を呼び出す場合と変わりはない。
(5)	Service を呼び出して業務処理を実行する。 通常の Controller で Service を呼び出す場合と変わりはない。

注釈: Web サービス関連のクラスは ws パッケージ配下にまとめることを推奨する。これは、アプリケーション層のクラスは app パッケージ配下に配置することを推奨しており、それらと区別をしやすくするためである。

入力チェックの実装

SOAP Web Service により送信されたパラメータの入力チェックには、Spring から提供されているメソッドバリデーションを使用する。

メソッドバリデーションの詳細については *Method Validation 対象のメソッドにするための定義方法*を参照されたい。

以下のように、Service のインターフェースに入力チェック内容を定義する。

[server projectName]-domain/src/main/java/com/example/domain/service/todo/TodoService.java

```
package com.example.domain.service.todo;

import java.util.List;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;
import javax.validation.groups.Default;
```

```
import org.springframework.validation.annotation.Validated;

import com.example.domain.model.Todo;

@Validated // (1)
public interface TodoService {

    Todo getTodo(@NotNull String todoId); // (2)

    Todo createTodo(@Valid Todo todo); // (3)

    @Validated({ Default.class, Todo.Update.class }) // (4)
    Todo updateTodo(@Valid Todo todo);

}
```

項番	説明
(1)	`@Validated` を付けることで、このインターフェースの実装クラスが入力チェック対象であることを宣言する。
(2)	引数をチェックする場合には、引数自体にアノテーションを付ける。
(3)	JavaBean の入力チェックを行う場合も、引数に`@Valid` を付ける。
(4)	`@Validated` にグループを指定し、特定の条件を絞って入力チェックすることも可能である。 グループの詳細は次の JavaBean の説明で記述する。

[server projectName]-model/src/main/java/com/example/domain/model/Todo.java

```
package com.example.domain.model;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Null;
import java.io.Serializable;
import java.util.Date;

// (1)
```

```
public class Todo implements Serializable {

    // (2)
    public interface Create {
    }

    public interface Update {
    }

    @Null(groups = Create.class)
    @NotNull(groups = Update.class)
    private String todoId;

    @NotNull
    private String title;

    private String description;

    private boolean finished;

    @Null(groups = Create.class)
    private Date createdAt;

    // omitted setter and getter
}
```

項番	説明
(1)	Bean Validation で JavaBean の入力チェックを定義する。 詳細は「入力チェック」を参照されたい。
(2)	バリデーションのグループ化を行うために使用するインターフェースを定義する。

セキュリティ対策

認証処理

SOAP の認証・認可方式に関して、本ガイドラインでは Spring Security で Basic 認証を行う方法と Service での認可の方法のみ紹介する。

WS-Security は扱わない。

詳細な利用方法は、「認証」と「認可」を参照されたい。

以下に SOAP Web Service に対して、Basic 認証を行う Spring Security の設定例を示す。

[server projectName]-web/src/main/resources/META-INF/spring/spring-security.xml

```
<!-- (1) -->
<sec:http pattern="/ws/**"
          create-session="stateless">
   <sec:csrf disabled="true" />
   <sec:http-basic />
</sec:http>

<!-- (2) -->
<sec:authentication-manager>
   <sec:authentication-provider
       user-service-ref="sampleUserDetailsService">
       <sec:password-encoder ref="passwordEncoder" />
   </sec:authentication-provider>
</sec:authentication-manager>
```

項番	説明
(1)	sec:http-basic タグを記述すると Basic 認証を行うことができる。 pattern 属性を使用して、Web サービスを実行する部分のみ認証を行う。
(2)	authentication-provider を利用して、認証方式を定義する。 実際の認証およびユーザ情報取得は UserDetailsService を作成して実施する必要がある。 詳細は「認証」を参照されたい。

認可処理

認可は Service ごとにアノテーションを付けて行う。
詳細は「認可」のアクセス認可 (Method) を参照されたい。

[server projectName]-web/src/main/resources/META-INF/spring/spring-security.xml

```
<sec:global-method-security pre-post-annotations="enabled" /> <!-- (1) -->
```

項番	説明
(1)	<sec:global-method-security>要素の pre-post-annotations 属性を enabled に指定する。

[server projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java

```
public class TodoServiceImpl implements TodoService {

    // omitted

    // (1)
    @PreAuthorize("isAuthenticated()")
    public List<Todo> getTodos() {
        // omitted
    }

    @PreAuthorize("hasRole('ROLE_ADMIN')")
    public Todo createTodo(Todo todo) {
        // omitted
    }

}
```

項番	説明
(1)	認可処理を行うメソッドに org.springframework.security.access.prepost.PreAuthorize アノテーションを設定する。

CSRF 対策

SOAP Web Service はセッションを利用せず、ステートレスな通信にすべきである。

そのため、セッションを利用する CSRF 対策を行わないようにするための設定方法について以下に記述する。

CSRF の詳細は「*CSRF 対策*」を参照されたい。

ブランクプロジェクトのデフォルトの設定では、CSRF 対策が有効化されている。

そのため、以下の設定を追加し、SOAP Web Service のリクエストに対して、CSRF 対策の処理が行われないようにする。

[server projectName]-web/src/main/resources/META-INF/spring/spring-security.xml

```
<!-- (1) -->
<sec:http pattern="/ws/**"
    create-session="stateless">
    <sec:http-basic />
    <sec:csrf disabled="true" />
</sec:http>
```

項番	説明
(1)	SOAP Web Service 用の Spring Security の定義を追加する。 <sec:http>要素の pattern 属性に SOAP Web Service 用のリクエストパスの URL パターンを指定する。 このコード例では、/ws/で始まるリクエストパスを SOAP Web Service 用のリクエストパスとしている。 また、create-session 属性を stateless とする事で、Spring Security の処理でセッションが使用されなくなる。 CSRF 対策を無効化するために、<sec:csrf>要素の disabled 属性を true に指定する。

例外ハンドリングの実装

SOAP サーバで例外が発生した場合にクライアントへ伝えるためには専用の例外クラスをスローする必要がある。

その実装を以下に記述する。

SOAP サーバで発生する例外

SOAP サーバで発生した例外はこれから記述する例外を実装したクラス（SOAPFault）を使用することで、クライアントへの通知メッセージを決定することができる。

具体的には以下のクラスを作成する。

項番	クラス名	概要
(1)	`ErrorBean`	発生した例外のコードとメッセージなどを保持するクラス。
(2)	`WebFaultType`	例外の種類を判別するために使用する列挙型。
(3)	`WebFaultBean`	`ErrorBean` と `WebFaultType` を保持するクラス。 `ErrorBean` を `List` で保持して例外情報を複数保持できる。
(4)	`WebFaultException`	`WebFaultBean` を保持する例外クラス。

これらの例外は SOAP サーバ、クライアントで共用するため、[server projectName]-webservice に配置する。

[server projectName]-webservice/src/main/java/com/example/ws/webfault/ErrorBean.java

```
package com.example.ws.webfault;

public class ErrorBean { // (1)
    private String code;
    private String message;
    private String path;

    // omitted setter and getter
}
```

項番	説明
(1)	例外のメッセージなどを保持するクラスを作成する。

[server projectName]-webservice/src/main/java/com/example/ws/webfault/WebFaultType.java

```
package com.example.ws.webfault;

public enum WebFaultType { // (2)
    AccessDeniedFault,
    BusinessFault,
    ResourceNotFoundFault,
    ValidationFault,
}
```

項番	説明
(1)	例外の種類を判別するために使用する列挙型を定義する。

[server projectName]-webservice/src/main/java/com/example/ws/webfault/WebFaultBean.java

```
package com.example.ws.webfault;

import java.util.ArrayList;
import java.util.List;

public class WebFaultBean { // (3)

    private WebFaultType type;

    private List<ErrorBean> errors = new ArrayList<ErrorBean>();

    public WebFaultBean(WebFaultType type) {
        this.type = type;
    }
```

```
    public void addError(String code, String message) {
        addError(code, message, null);
    }

    public void addError(String code, String message, String path) {
        errors.add(new ErrorBean(code, message, path));
    }

    // omitted setter and getter
}
```

項番	説明
(1)	ErrorBean と WebFaultType を保持するクラスを作成する。

[server projectName]-webservice/src/main/java/com/example/ws/webfault/WebFaultException.java

```
package com.example.ws.webfault;

import java.util.List;

import javax.xml.ws.WebFault;

@WebFault(name = "WebFault", targetNamespace = "http://example.com/todo") // (1)
public class WebFaultException extends Exception {
    private WebFaultBean faultInfo; // (2)

    public WebFaultException() {
    }

    public WebFaultException(String message, WebFaultBean faultInfo) {
        super(message);
        this.faultInfo = faultInfo;
    }

    public WebFaultException(String message, WebFaultBean faultInfo, Throwable e) {
        super(message, e);
        this.faultInfo = faultInfo;
    }

    public List<ErrorBean> getErrors() {
        return this.faultInfo.getErrors();
```

```
    }

    public WebFaultType getType() {
        return this.faultInfo.getType();
    }
    // omitted setter and getter
}
```

項番	説明
(1)	Exception 継承クラスに`@WebFault` を付けて、SOAPFault であることを宣言する。 `name` 属性には、クライアントに送信する SOAPFault の `name` 属性を設定する。 `targetNamespace` 属性には、使用するネームスペースを設定する。Web サービスと同じにする必要がある。
(2)	faultInfo をフィールドに保持させるとともに、コード例のように以下のようなコンストラクタとメソッドを持たせる。 • メッセージ文字列と faultInfo を引数とするコンストラクタ • メッセージ文字列と faultInfo と原因例外を引数とするコンストラクタ • getFaultInfo メソッド

注釈: WebFaultException に RuntimeException ではなく、Exception を継承させている理由

`WebFaultException` の親クラスを `RuntimeException` にすれば、例外の処理をもっと簡略化することができそうに見える。しかし、親クラスを `RuntimeException` にしてはいけない。JSR 224: JavaTM API for XML-Based Web Services でも明確にしてはいけないと宣言されている。実際に試してみても、AP サーバの JAX-WS 実装次第ではあるが、クライアントで`@WebFault` を付けた例外クラス（`WebFaultException`）を取得することができず、エラーの種類やメッセージを取得することができなくなる。AOP を使用して例外処理を実施していないのも `Exception` を継承しているためである。

警告: WebFaultException のコンストラクタとフィールドについて

`WebFaultException` には、デフォルトコンストラクタと各フィールドに対応する setter が必須となる。これは、クライアントの内部処理で、`WebFaultException` を作成する際に使用するためである。そのため、各フィールドを final にすることも不可能である。

この WebFaultException を継承し、クライアントへ伝えたい種類分、子クラスを作成する。
たとえば以下のような子クラスを作成する。

- 業務エラー例外
- 入力エラー例外
- リソース未検出エラー例外
- 排他エラー例外
- 認可エラー例外
- システムエラー例外

下記は、業務エラー例外の例である。

[server projectName]-webservice/src/main/java/com/example/ws/webfault/BusinessFaultException.java

```
package com.example.ws.webfault;

import javax.xml.ws.WebFault;

@WebFault(name = "BusinessFault", targetNamespace = "http://example.com/todo") //␣
↪(1)
public class BusinessFaultException extends WebFaultException {

    public BusinessFaultException(String message, WebFaultBean faultInfo) {
        super(message, faultInfo);
    }

    public BusinessFaultException(String message, WebFaultBean faultInfo,␣
↪Throwable e) {
        super(message, faultInfo, e);
    }

}
```

項番	説明
(1)	WebFaultException を継承し、コンストラクタのみ作成する。 フィールドやその他メソッドは親クラスのメソッドを使用するため記述不要である。

発生する例外を SOAPFault でラップする例外ハンドラー

Service から発生する実行時例外を SOAPFault でラップするために例外ハンドラークラスを作成する。本ガイドラインでは WebService 実装クラスがこのハンドラーを用いて例外を変換してスローする方針とする。

Service からスローされる例外は以下を想定している。必要に応じて追加されたい。

例外名	内容
`org.springframework.security.access.AccessDeniedException`	認可エラー時の例外
`javax.validation.ConstraintViolationException`	入力チェックエラー時の例外
`org.terasoluna.gfw.common.exception.ResourceNotFoundException`	リソースが見つからない場合の例外
`org.terasoluna.gfw.common.exception.BusinessException`	業務例外

[server projectName]-web/src/main/java/com/example/ws/exception/WsExceptionHandler.java

```
package com.example.ws.exception;

import java.util.Iterator;
import java.util.Locale;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.ConstraintViolationException;
import javax.validation.Path;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.stereotype.Component;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ExceptionCodeResolver;
import org.terasoluna.gfw.common.exception.ExceptionLogger;
```

```
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.exception.SystemException;
import org.terasoluna.gfw.common.message.ResultMessage;
import org.terasoluna.gfw.common.message.ResultMessages;

import com.example.ws.webfault.WebFaultBean;
import com.example.ws.webfault.WebFaultException;
import com.example.ws.webfault.WebFaultType;

@Component  // (1)
public class WsExceptionHandler {

    @Autowired
    MessageSource messageSource; // (2)

    @Autowired
    ExceptionCodeResolver exceptionCodeResolver; // (3)

    @Autowired
    ExceptionLogger exceptionLogger; // (4)

    // (5)
    public void translateException(Exception e) throws WebFaultException {
        loggingException(e);
        WebFaultBean faultInfo = null;

        if (e instanceof AccessDeniedException) {
            faultInfo = new WebFaultBean(WebFaultType.AccessDeniedFault);
            faultInfo.addError(e.getClass().getName(), e.getMessage());
        } else if (e instanceof ConstraintViolationException) {
            faultInfo = new WebFaultBean(WebFaultType.ValidationFault);
            this.addErrors(faultInfo, ((ConstraintViolationException) e).
↪getConstraintViolations());
        } else if (e instanceof ResourceNotFoundException) {
            faultInfo = new WebFaultBean(WebFaultType.ResourceNotFoundFault);
            this.addErrors(faultInfo, ((ResourceNotFoundException) e).
↪getResultMessages());
        } else if (e instanceof BusinessException) {
            faultInfo = new WebFaultBean(WebFaultType.BusinessFault);
            this.addErrors(faultInfo, ((BusinessException) e).getResultMessages());
        } else {
            // not translate.
            throw new SystemException("e.ex.fw.9001", e);
        }

        throw new WebFaultException(e.getMessage(), faultInfo, e.getCause());
    }

    private void loggingException(Exception e) {
        exceptionLogger.log(e);
    }

    private void addErrors(WebFaultBean faultInfo, Set<ConstraintViolation<?>>␣
↪constraintViolations) {
```

```
        for (ConstraintViolation<?> v : constraintViolations) {
            Iterator<Path.Node> pathIt = v.getPropertyPath().iterator();
            pathIt.next(); // method name node (skip)
            Path.Node methodArgumentNameNode = pathIt.next();
            faultInfo.addError(
                v.getConstraintDescriptor().getAnnotation().annotationType().
↪getSimpleName(),
                v.getMessage(),
                pathIt.hasNext() ? pathIt.next().toString() :␣
↪methodArgumentNameNode.toString());
        }

    }

    private void addErrors(WebFaultBean faultInfo, ResultMessages resultMessages) {
        Locale locale = Locale.getDefault();
        for (ResultMessage message : resultMessages) {
            faultInfo.addError(
                message.getCode(),
                messageSource.getMessage(message.getCode(), message.getArgs(),␣
↪message.getText(), locale));
        }
    }

}
```

項番	説明
(1)	本クラスを DI コンテナに管理をさせるため、`@Component` を付ける。
(2)	出力するメッセージを取得するために `MessageSource` を使用する。
(3)	共通ライブラリが提供する `ExceptionCodeResolverMessageSource` を使用して例外の種類と例外コードをマッピングする。 詳細は、「例外ハンドリング」を参照されたい。
(4)	共通ライブラリが提供する `ExceptionLogger` を使用して例外情報を例外に出力する。 詳細は、「例外ハンドリング」を参照されたい。
(5)	Service から発生しうる各例外について、`SOAPFault` へのラップを行う。 例外のマッピングは冒頭の表を参考されたい。

注釈: その他の例外の扱いについて

その他の例外発生時（上記の `translateException` メソッドの else 部分）では、クライアントでは詳細な例外の内容は通知されず、`com.sun.xml.internal.ws.fault.ServerSOAPFaultException` が発生するのみとなる。他の例外同様にラップしてクライアント側に通知することも可能である。

Service で発生した例外を Web サービス内から例外ハンドラーを呼び出し、ラップする

Web サービスクラスにて、例外ハンドラーを呼び出す。以下はその例である。

[server projectName]-web/src/main/java/com/example/ws/todo/TodoWebServiceImpl.java

```
@WebService(
        portName = "TodoWebPort",
        serviceName = "TodoWebService",
```

```
        targetNamespace = "http://example.com/todo",
        endpointInterface = "com.example.ws.todo.TodoWebService")
@BindingType(SOAPBinding.SOAP12HTTP_BINDING)
public class TodoWebServiceImpl extends SpringBeanAutowiringSupport implements␣
↪TodoWebService {
    @Autowired
    TodoService todoService;
    @Autowired
    WsExceptionHandler handler; // (1)

    @Override
    public Todo getTodo(String todoId) throws WebFaultException /* (2) */ {
        try {
            return todoService.getTodo(todoId);
        } catch (RuntimeException e) {
            handler.translateException(e); // (3)
        }
    }
}
```

項番	説明
(1)	例外ハンドラーをインジェクションする。
(2)	`WebFaultException` にラップしてスローするため、throws 句を付ける。
(3)	実行時例外が発生した場合は、例外ハンドラークラスに処理を委譲する。

MTOM を利用した大容量のバイナリデータを扱う方法

SOAP では、バイナリデータを扱う場合、Byte 配列にマッピングすることで、送受信を行うことができる。

ただし、大容量のバイナリデータを扱う場合、ヒープが枯渇するなどの問題が発生することがある。

そこで、MTOM（Message Transmission Optimization Mechanism）に準拠した実装を行うことで、最適化した状態で添付ファイルとしてバイナリデータを扱うことができる。

詳細な定義は W3C -SOAP Message Transmission Optimization Mechanism-を参照されたい。

以下にその方法を記述する。

[server projectName]-webservice/src/main/java/com/example/ws/todo/TodoWebService.java

```
package com.example.ws.todo;

import java.util.List;

import javax.activation.DataHandler;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlMimeType;

import com.example.domain.model.Todo;
import com.example.ws.webfault.WebFaultException;

@WebService(targetNamespace = "http://example.com/todo")
public interface TodoWebService {

    // omitted

    @WebMethod
    void uploadFile(@XmlMimeType("application/octet-stream") /* (1) */ DataHandler
↪dataHandler) throws WebFaultException;

}
```

項番	説明
(1)	バイナリデータを処理する javax.activation.DataHandler に対して@XmlMimeType を付ける。

[server projectName]-web/src/main/java/com/example/ws/todo/TodoWebServiceImpl.java

```
package com.example.ws.todo;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;

import javax.activation.DataHandler;
import javax.jws.HandlerChain;
import javax.jws.WebService;
import javax.xml.ws.BindingType;
```

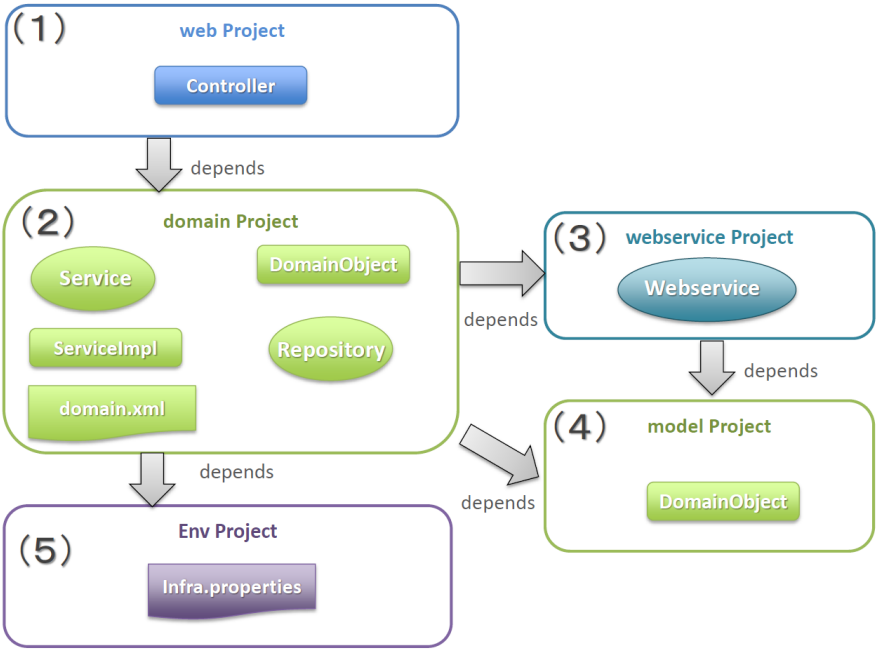
```
import javax.xml.ws.soap.MTOM;
import javax.xml.ws.soap.SOAPBinding;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.context.support.SpringBeanAutowiringSupport;
import org.terasoluna.gfw.common.exception.SystemException;

import com.example.domain.model.Todo;
import com.example.domain.service.TodoService;
import com.example.ws.webfault.WebFaultException;
import com.example.ws.exception.WsExceptionHandler;

// (1)
@MTOM
@WebService(
        portName = "TodoWebPort",
        serviceName = "TodoWebService",
        targetNamespace = "http://example.com/todo",
        endpointInterface = "com.example.ws.todo.TodoWebService")
@BindingType(SOAPBinding.SOAP12HTTP_BINDING)
public class TodoWebServiceImpl extends SpringBeanAutowiringSupport implements␣
↪TodoWebService {

    @Autowired
    TodoService todoService;

    // omitted

    @Override
    public void uploadFile(DataHandler dataHandler) throws WebFaultException {

        try (InputStream inputStream = dataHandler.getInputStream()){ // (2)
            todoService.uploadFile(inputStream);
        } catch (Exception e) {
            handler.translateException(e);
        }
    }

}
```

項番	説明
(1)	`@MTOM` を付けて、MTOM に準拠した実装を使用することを宣言する。
(2)	`javax.activation.DataHandler` から `java.io.InputStream` を取得してファイルを扱う。

クライアントの作成

プロジェクトの構成

「*JAX-WS を利用した Web サービスの開発について*」で述べたとおり、model プロジェクトと webservice プロジェクトを SOAP サーバから受領する前提である。



項番	プロジェクト名	説明
(1)	web プロジェクト	Controller を作成する。 通常の画面遷移時の Controller と特に変更点はない。
(2)	domain プロジェクト	Service クラスから webservice プロジェクトで用意された WebServe インターフェースを使用して Web サービスを呼び出す。 SOAP サーバと通信する際に使用する WebService インターフェースを実装したプロキシを定義する。

Continued on next page

表 5.20 – continued from previous page

項番	プロジェクト名	説明
(3)	webservice プロジェクト	SOAP サーバと同じ資材を配置する。 クライアントはこのインターフェースを使用して Web サービスを実行する。
(4)	model プロジェクト	SOAP サーバと同じ資材を配置する。 SOAP サーバに渡す入力値や返却結果はこのプロジェクト内のクラスを使用する。
(5)	env プロジェクト	domain プロジェクトで定義したプロキシの環境依存する値を定義する。 プロキシの定義から環境依存する値をプロパティファイルに集約し、プロパティファイルのみ env プロジェクトに配置する。

注釈: プロキシの定義ついて

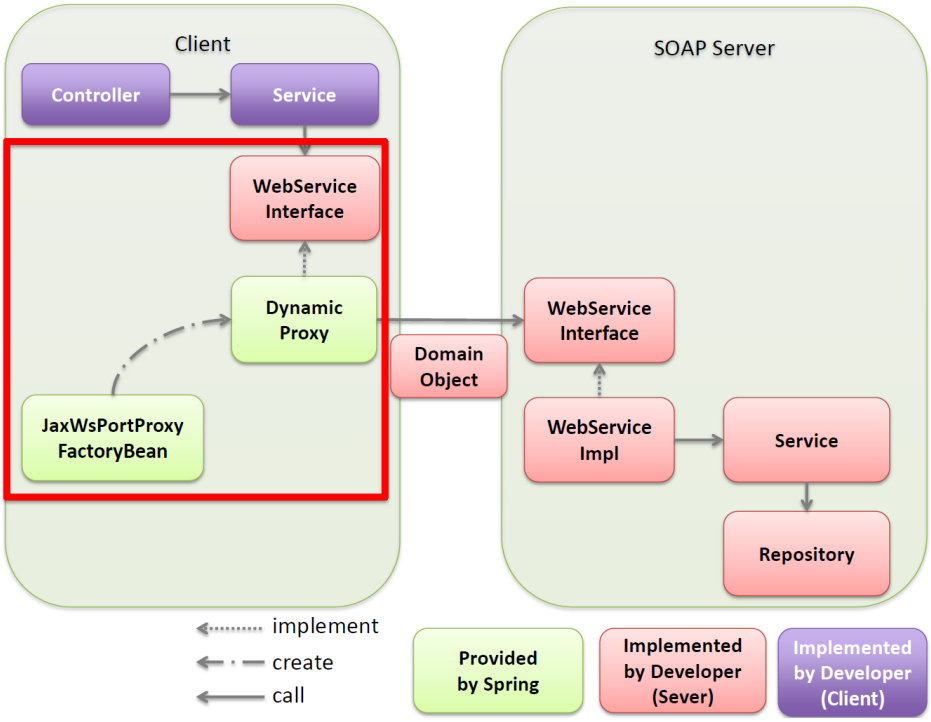
試験用 SOAP サーバ、本番用 SOAP サーバ等、複数環境向けのプロキシを定義する際に発生する重複部分を排除し、管理を容易にするために、当ガイドラインではプロキシの定義は domain プロジェクトで行い、環境依存する値はプロパティファイルに集約、プロパティファイルのみ env プロジェクトに配置することを推奨する。

ユニットテストでプロキシのスタブやモックを使用する場合は、ユニットテスト用のコンポーネントを定義するための Bean 定義ファイル (test-context.xml) に Bean を定義する。

Web サービス クライアントの実装

以下のクラスの実装を行う。

- WebService インターフェースを実装したプロキシの定義
- Service クラスから WebService インターフェース経由で Web サービスを呼び出す。



WebService インターフェースを実装したプロキシの作成

WebService インターフェースを実装したプロキシを生成する `org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean` の定義を行う。

[client projectName]-domain/src/main/resources/META-INF/spring/[client projectName]-domain.xml

```
<bean id="todoWebService"
    class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean"><!-- (1) -
↪->
    <property name="serviceInterface" value="com.example.ws.todo.TodoWebService" />
↪<!-- (2) -->
    <!-- (3) -->
    <property name="serviceName" value="TodoWebService" />
    <property name="portName" value="TodoWebPort" />
    <property name="namespaceUri" value="http://example.com/todo" />
    <property name="wsdlDocumentResource" value="${webservice.todoWebService.
↪wsdlDocumentResource}" /><!-- (4) -->
    <property name="lookupServiceOnStartup" value="false" /><!-- (5) -->
</bean>
```

[client projectName]-env/src/main/resources/META-INF/spring/[client projectName]-infra.properties

```
# (6)
webservice.todoWebService.wsdlDocumentResource=http://AAA.BBB.CCC.DDD:XXXX/[server␣
↪projectName]-web/ws/TodoWebService?wsdl
```

項番	説明
(1)	`org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean` を定義する。このクラスが生成するプロキシを経由して SOAP サーバにアクセスできる。
(2)	`serviceInterface` プロパティに本来この Web サービスが実装すべきインターフェースを定義する。
(3)	`serviceName`、`portName`、`namespaceUri` プロパティにそれぞれ SOAP サーバ側で定義している同じ内容を定義する必要がある。
(4)	`wsdlDocumentResource` プロパティに公開されている WDSL の URL を設定する。 ここでは後述するプロパティファイルに URL を記述するため、プロパティのキーを指定している。
(5)	`lookupServiceOnStartup` プロパティに Bean 生成する時、SOAP サーバから WSDL ファイルを取得するかどうかを設定する。false の場合は Bean が初めて使用されるタイミングで WSDL ファイルの取得が行われる。 SOAP サーバから WSDL ファイルの取得が不可能な場合でも Web サービス クライアントを起動させるために、`lookupServiceOnStartup` プロパティに `false` を指定することを推奨する。ただし、WSDL ファイルを Web サービス クライアントで保持している場合は設定不要である。
(6)	`[client projectName]-domain.xml` で定義したプロパティのキーの値を設定する。 WSDL の URL を記述する。 **注釈: wsdlDocumentResource への WSDL ファイルの URL 以外の指定** 上記の例では、SOAP サーバが WSDL ファイルを公開している前提である。`classpath:`や`file:`プレフィックスを使用して指定することで静的ファイルを指定することもできる。指定できる文字列は、Spring Framework Reference Documentation -Resources(The ResourceLoader)-を参照されたい。

注釈: エンドポイントアドレスの上書き指定

WSDL ファイルには、Web サービス実行時のアクセス URL（エンドポイントアドレス）が記述されているため、クライアントではアクセス URL の設定は不要である。ただし、WSDL ファイルに記述されている URL ではない URL にアクセスする場合、`org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean` の `endpointAddress` プロパティを設定することで上書きすることができる。テストなどで、環境を切り替える場合に使用するとよい。以下はその設定例である。

[client projectName]-domain/src/main/resources/META-INF/spring/[client projectName]-domain.xml

```
<bean id="todoWebService"
    class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
    <property name="serviceInterface" value="com.example.ws.todo.
↪TodoWebService" />
    <property name="serviceName" value="TodoWebService" />
    <property name="portName" value="TodoWebPort" />
    <property name="namespaceUri" value="http://example.com/todo" />
    <property name="wsdlDocumentResource" value="${webservice.todoWebService.
↪wsdlDocumentResource}" />
    <property name="endpointAddress" value="${webservice.todoWebService.
↪endpointAddress}" /><!-- (1) -->
    <property name="lookupServiceOnStartup" value="false" />
</bean>
```

[client projectName]-env/src/main/resources/META-INF/spring/[client projectName]-infra.properties

```
# (2)
webservice.todoWebService.endpointAddress=http://AAA.BBB.CCC.DDD:XXXX/[server␣
↪projectName]-web/ws/TodoWebService
```

項番	説明
(1)	エンドポイントアドレスを設定する。 ここでは後述するプロパティファイルに URL を記述するため、プロパティのキーを指定している。
(2)	`[client projectName]-domain.xml` で定義したプロパティのキーの値を設定する。 エンドポイントアドレスを記述する。

Service から Web サービスを呼び出す

上記で作成した Web サービスを Service でインジェクションして実行する。

[client projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java

```
package com.example.soap.domain.service.todo;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.example.domain.model.Todo;
import com.example.ws.webfault.WebFaultException;
import com.example.ws.todo.TodoWebService;

@Service
public class TodoServiceImpl implements TodoService {

    @Autowired
    TodoWebService todoWebService;

    @Override
    public void createTodo(Todo todo) {
        // (1)
        try {
            todoWebService.createTodo(todo);
        } catch (WebFaultException e) {
            // (2)
            // handle exception…
        }
    }
}
```

項番	説明
(1)	TodoWebService をインジェクションして、実行対象の Service を呼び出す。
(2)	サーバ側で、例外が発生した場合は、WebFaultException にラップされて送信される。 内容に応じて処理を行う。 例外処理の詳細は「例外ハンドリングの実装」を参照されたい。

注釈: レスポンスの情報取得

リトライを考慮するなど、レスポンスの情報をクライアントで取得したい場合、以下のように javax.xml.ws.BindingProvider クラスにキャストすることで取得できる。

```
BindingProvider provider = (BindingProvider) todoWebService;
int status = (int) provider.getResponseContext().get(MessageContext.HTTP_
↪RESPONSE_CODE);
```

BindingProvider の詳細については The Java API for XML-Based Web Services(JAX-WS) 2.2 -4.2 javax.xml.ws.BindingProvider-を参照されたい。

ただし、クライアントの依存関係に Apatch CXF ライブラリが含まれる場合、通信エラー時に上記の方法でレスポンスの情報を取得することができない。これは、依存関係に Apatch CXF ライブラリが含まれる場合は自動的に Apatch CXF のプロキシが使用されるため、および Apache CXF のプロキシは通信エラーが発生した場合にレスポンスの情報をレスポンスコンテキストに保持しないためである。Apache CXF のエラー処理については Apache CXF Software Architecture Guide -Fault Handling-を参照されたい。

Web サービスと別の Web サービスへのクライアントを持つ中継サービスのように、どうしてもクライアントに Apache CXF ライブラリの依存関係を含んでしまう場合は制限事項として注意されたい。

セキュリティ対策

認証処理

org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean を使用している場合で Basic 認証を使用している SOAP サーバと通信をする場合には、bean 定義にユーザ名とパスワードを追加するだけで認証を行うことができる。

[client projectName]-domain/src/main/resources/META-INF/spring/[client projectName]-domain.xml

```
<bean id="todoWebService"
    class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
    <property name="serviceInterface" value="com.example.ws.todo.TodoWebService" />
    <property name="serviceName" value="TodoWebService" />
    <property name="portName" value="TodoWebPort" />
    <property name="namespaceUri" value="http://example.com/todo" />
    <property name="wsdlDocumentResource" value="${webservice.todoWebService.
↪wsdlDocumentResource}" />
    <!-- (1) -->
    <property name="username" value="${webservice.todoWebService.username}" />
    <property name="password" value="${webservice.todoWebService.password}" />
</bean>
```

[client projectName]-env/src/main/resources/META-INF/spring/[client projectName]-infra.properties

```
# (2)
webservice.todoWebService.username=testuser
webservice.todoWebService.password=password
```

項番	説明
(1)	`org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean` の bean 定義に username と password を加えることで Basic 認証における、認証情報を送信することができる。 ユーザ名とパスワードをプロパティファイルに切り出した場合のサンプルである。
(2)	`[client projectName]-domain.xml` で定義したプロパティのキーの値を設定する。認証に使用するユーザ名とパスワードを記述する。

例外ハンドリングの実装

SOAP サーバでは、`WebFaultException` に例外をラップして、スローすることを推奨している。
クライアントは `WebFaultException` をキャッチして、その原因例外を判定してそれぞれの処理を行う。

```
@Override
public void createTodo(Todo todo) {

    try {
        // (1)
        todoWebService.createTodo(todo);
    } catch (WebFaultException e) {
        // (2)
        switch (e.getFaultInfo().getType()) {
        case ValidationFault:
            // handle exception…
            break;
        case BusinessFault:
            // handle exception…
            break;
        default:
            // handle exception…
            break;
        }
```

```
    }

}
```

項番	説明
(1)	Web サービスを呼び出す。throws がついているため、`WebFaultException` をキャッチする必要がある。
(2)	`faultInfo` の種別で例外を判定し、それぞれの処理を記述する（画面にメッセージを出す、例外をスローするなど）

タイムアウトの設定

クライアントで指定できるタイムアウトは大きく以下の 2 つがある。

- SOAP サーバとのコネクションタイムアウト
- SOAP サーバへのリクエストタイムアウト

どちらの設定も、`org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean` のカスタムプロパティに指定する必要がある。

設定の方法は以下の通りである。

[client projectName]-domain/src/main/resources/META-INF/spring/[client projectName]-domain.xml

```
<bean id="todoWebService"
    class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
    <property name="serviceInterface" value="com.example.ws.todo.TodoWebService" />
    <property name="serviceName" value="TodoWebService" />
    <property name="portName" value="TodoWebPort" />
    <property name="namespaceUri" value="http://example.com/todo" />
    <property name="wsdlDocumentResource" value="${webservice.todoWebService.
↪wsdlDocumentResource}" />
    <!-- (1) -->
    <property name="customProperties">
        <map>
```

```
            <!-- (2) -->
            <entry key="com.sun.xml.internal.ws.connect.timeout" value="$
↪{webservice.connect.timeout}"/>
            <entry key="com.sun.xml.internal.ws.request.timeout" value="$
↪{webservice.request.timeout}"/>
        </map>
    </property>
</bean>
```

[client projectName]-env/src/main/resources/META-INF/spring/[client projectName]-infra.properties

```
# (3)
webservice.request.timeout=3000
webservice.connect.timeout=3000
```

項番	説明
(1)	customProperties プロパティに Map を指定することでカスタムプロパティを定義する。

(2)	コネクションタイムアウトとリクエストタイムアウトを定義する。
	それぞれの値をプロパティファイルに切り出した場合のサンプルである。
	
	警告: タイムアウト定義に使用するキーについて
	それぞれのタイムアウトを定義するキーは JAX-WS の実装により異なる値を設定する必要がある。詳細は JAX_WS-1166 Standardize timeout settings を参照されたい。
	
	注釈: WebLogic で該当キーを指定する場合は、value-type 属性で Integer 型を指定する必要がある。
	指定をしない場合、WebLogic の JAX-WS 実装ライブラリが String 型から Integer 型へのキャストを試みて失敗し、結果的に ClassCastException が原因の org.springframework.remoting.RemoteAccessException 例外が発生する。
	設定の方法は以下のとおりである。
	`<property name="customProperties">`
	`  <map>`
	`  <entry key="com.sun.xml.internal.ws.connect.timeout" value-type="java.lang.Integer" value="${webservice.connect.timeout}"/>`
	`  <entry key="com.sun.xml.internal.ws.request.timeout" value-type="java.lang.Integer" value="${webservice.request.timeout}"/>`
	`  </map>`
	`</property>`

(3)	[client projectName]-domain.xml で定義したプロパティのキーの値を設定する。コネクションタイムアウトとリクエストタイムアウトを記述する。

5.3.3 Appendix

SOAP サーバ用にプロジェクトの設定を変更する

SOAP サーバを作成する場合、ブランクプロジェクトに model プロジェクトと webservice プロジェクトを追加することを推奨する。

以下にその方法を記述する。

ブランクプロジェクトの初期状態は以下の構成になっている。

なお、artifactId にはブランクプロジェクト作成時に指定した artifactId が設定される。

```
artifactId
├── pom.xml
├── artifactId-domain
├── artifactId-env
├── artifactId-initdb
├── artifactId-selenium
└── artifactId-web
```

以下のようなプロジェクト構成にする。

```
artifactId
├── pom.xml
├── artifactId-domain
├── artifactId-env
├── artifactId-initdb
├── artifactId-selenium
├── artifactId-web
├── artifactId-model
└── artifactId-webservice
```

既存プロジェクトの変更

ブランクプロジェクトの初期状態では、Controller など Web アプリケーションの簡易実装が含まれている。

そのままにしても SOAP Web Service は実現可能だが、不要であるため、削除することを推奨する。

削除対象は、「*Web* アプリケーション向け開発プロジェクトの作成 の マルチプロジェクトの構成」を参照されたい。

model プロジェクトの作成

model プロジェクトの構成について説明する。

```
artifactId-model
    ├─── pom.xml  ... (1)
```

項番	説明
(1)	model モジュールの構成を定義する POM(Project Object Model) ファイル。このファイルでは、以下の定義を行う。 • 依存ライブラリとビルド用プラグインの定義 • jar ファイルを作成するための定義

pom.xml は以下のようなイメージになる。必要に応じて編集する必要がある。
実際には、「artifactId」と「groupId」はブランクプロジェクト作成時に指定した値を設定する必要がある。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
↪2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0␣
↪http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <artifactId>artifactId-model</artifactId>
    <packaging>jar</packaging>
    <parent>
        <groupId>groupId</groupId>
        <artifactId>artifactId</artifactId>
        <version>1.0.0-SNAPSHOT</version>
        <relativePath>../pom.xml</relativePath>
    </parent>
    <dependencies>
        <!-- == Begin TERASOLUNA == -->
        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-common-dependencies</artifactId>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-jodatime-dependencies</artifactId>
            <type>pom</type>
        </dependency>
        <dependency>
```

```
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-security-core-dependencies</artifactId>
            <type>pom</type>
        </dependency>

        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-recommended-dependencies</artifactId>
            <type>pom</type>
        </dependency>
        <!-- == End TERASOLUNA == -->
    </dependencies>
</project>
```

webservice プロジェクトの作成

webservice プロジェクトの構成について説明する。

```
artifactId-webservice
    ├── pom.xml  ... (1)
```

項番	説明
(1)	webservice モジュールの構成を定義する POM(Project Object Model) ファイル。このファイルでは、以下の定義を行う。 • 依存ライブラリとビルド用プラグインの定義 • jar ファイルを作成するための定義

pom.xml は以下のようなイメージになる。必要に応じて編集する必要がある。

実際には、「artifactId」と「groupId」はブランクプロジェクト作成時に指定した値を設定する必要がある。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
↪2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0␣
↪http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
```

```
    <artifactId>artifactId-webservice</artifactId>
    <packaging>jar</packaging>
    <parent>
        <groupId>groupId</groupId>
        <artifactId>artifactId</artifactId>
        <version>1.0.0-SNAPSHOT</version>
        <relativePath>../pom.xml</relativePath>
    </parent>
    <dependencies>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>artifactId-model</artifactId>
        </dependency>
        <!-- == Begin TERASOLUNA == -->
        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-common-dependencies</artifactId>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-jodatime-dependencies</artifactId>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-security-core-dependencies</artifactId>
            <type>pom</type>
        </dependency>

        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-recommended-dependencies</artifactId>
            <type>pom</type>
        </dependency>
        <!-- == End TERASOLUNA == -->
    </dependencies>
</project>
```

SOAP サーバのパッケージ構成

SOAP サーバを作成するときの推奨する構成について、説明する。

ガイドラインに従いプロジェクトを追加すると以下の構成となる。

プロジェクト名	説明
[server projectName]-domain	SOAP サーバのドメイン層に関するクラス・設定ファイルを格納するプロジェクト
[server projectName]-web	SOAP サーバのアプリケーション層に関するクラス・設定ファイルを格納するプロジェクト
[server projectName]-env	SOAP サーバの環境に依存するファイル等を格納するプロジェクト
[server projectName]-model	SOAP サーバのドメイン層に関するクラスの中で、Web サービス実行時に使用し、クライアントと共有するクラスを格納するプロジェクト
[server projectName]-webservice	SOAP サーバが提供する Web サービスのインターフェースを格納するプロジェクト

[server projectName]-domain

[server projectName]-model の依存関係を追加するため、`pom.xml` に以下を追加する。

```
<dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>artifactId-model</artifactId>
</dependency>
```

その他のパッケージ構成は、通常の domain プロジェクトと変わらないため、「アプリケーションのレイヤ化の プロジェクト構成」を参照されたい。

[server projectName]-web

[server projectName]-webservice の依存関係を追加するため、pom.xml に以下を追加する。

```
<dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>artifactId-webservice</artifactId>
</dependency>
```

注釈: 依存性の解決について

[server projectName]-model の依存関係の定義は不要である。これは [server projectName]-webservice から [server projectName]-model への依存関係が定義されているため、推移的に依存関係が追加されるためである。

[server projectName]-web のプロジェクト推奨構成を、以下に示す。

```
[server projectName]-web
  └src
      └main
          ├java
          |   └com
          |       └example
          |           ├app...(1)
          |           └ws...(2)
          |             ├exception...(3)
          |             |   └WsExceptionHandler.java
          |             ├abc
          |             |   └AbcWebServiceImpl.java
          |             └def
          |                 └DefWebServiceImpl.java
          ├resources
          |   ├META-INF
          |   |   └spring
          |   |       ├applicationContext.xml...(4)
          |   |       ├application.properties...(5)
          |   |       ├spring-mvc.xml ...(6)
          |   |       ├spring-security.xml...(7)
          |   |       └[server projectName]-ws.xml...(8)
          |   └i18n
          |       └application-messages.properties...(9)
          └webapp
              ├resources...(10)
              └WEB-INF
                  ├views ...(11)
                  └web.xml...(12)
```

項番	説明
(1)	アプリケーション層の構成要素を格納するパッケージ。 Web サービスのみ作成する場合は削除してもよい。
(2)	Web サービスの関連クラスを格納するパッケージ。
(3)	Web サービスの例外ハンドラーなどを格納するパッケージ。
(4)	アプリケーション全体に関する Bean 定義を行う。
(5)	アプリケーションで使用するプロパティを定義する。
(6)	Spring MVC の設定を行う Bean 定義を行う。 Web サービスのみ作成する場合は削除してもよい。
(7)	Spring Security の設定を行う Bean 定義を行う。
(8)	Web サービスに関する Bean 定義を行う。
(9)	画面表示用のメッセージ (国際化対応) 定義を行う。
(10)	静的リソース (css、js、画像など) を格納する。 Web サービスのみ作成する場合は削除してもよい。
(11)	View(jsp) を格納する。 Web サービスのみ作成する場合は削除してもよい。
(12)	Servlet のデプロイメント定義を行う。

注釈: SOAP サーバの不要なファイル

SOAP サーバで、Web サービスのみを作成する場合、ブランクプロジェクトに存在する Spring MVC の設定ファイルなどは不要となるため、削除したほうが望ましい。

[server projectName]-env

[server projectName]-env については、通常の env プロジェクトと変わらないため、「アプリケーションのレイヤ化 の プロジェクト構成」を参照されたい。

[server projectName]-model

[server projectName]-model のプロジェクト推奨構成を、以下に示す。

```
[server projectName]-model
  └src
      └main
          └java
              └com
                  └example
                      └domain ...(1)
                          └model ...(2)
                              ├Xxx.java
                              ├Yyy.java
                              └Zzz.java
```

項番	説明
(1)	ドメイン層の構成要素を格納するパッケージ。
(2)	Domain Object の中で Web サービス実行時に使用するクラスを格納するパッケージ。

[server projectName]-webservice

[server projectName]-webservice のプロジェクト推奨構成を、以下に示す。

```
[server projectName]-webservice
  └src
      └main
          └java
              └com
                  └example
                      └ws...(1)
                        ├webfault...(2)
                        ├abc
                        |   └AbcWebService.java
                        └def
                            └DefWebService.java
```

項番	説明
(1)	Web サービスのインターフェースを格納するパッケージ。
(2)	Web サービスの webfault を格納するパッケージ。

クライアントのパッケージ構成

クライアントを作成するときの推奨する構成について、説明する。

ガイドラインに従いプロジェクトを SOAP サーバから提供されると以下の構成となる。

プロジェクト名	説明
[client projectName]-domain	クライアントのドメイン層に関するクラス・設定ファイルを格納するプロジェクト
[client projectName]-web	クライアントのアプリケーション層に関するクラス・設定ファイルを格納するプロジェクト
[client projectName]-env	クライアントの環境に依存するファイル等を格納するプロジェクト

注釈: [server projectName]-model と [server projectName]-webservice については、前述の「*SOAP サーバのパッケージ構成*」を参照されたい。

[client projectName]-domain

SOAP サーバから提供される [server projectName]-webservice の依存関係を追加するため、`pom.xml` に以下を追加する。

```
<dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>artifactId-webservice</artifactId>
</dependency>
```

注釈: 依存性の解決について

[server projectName]-web と同様に、この `pom.xml` には、[server projectName]-model の依存関係の定義は不要である。これは [server projectName]-webservice から [server projectName]-model への依存関係が定義されているため、推移的に依存関係が追加されるためである。

その他のパッケージ構成は、通常の domain プロジェクトと変わらないため、「アプリケーションのレイヤ化の プロジェクト構成」を参照されたい。

[client projectName]-web

[client projectName]-web については、通常の web プロジェクトと変わらないため、「アプリケーションのレイヤ化 の プロジェクト構成」を参照されたい。

[client projectName]-env

[client projectName]-env のプロジェクト推奨構成を、以下に示す。

```
[projectName]-env
  ├configs ...(1)
  |    └[envName] ...(2)
  |        └resources ...(3)
  └src
      └main
          └resources ...(4)
             ├META-INF
             |   └spring
             |       ├[projectName]-env.xml ...(5)
             |       └[projectName]-infra.properties ...(6)
             ├dozer.properties
             ├log4jdbc.properties
             └logback.xml ...(7)
```

項番	説明
(1)	全環境の環境依存ファイルを管理するためのディレクトリ。
(2)	環境毎の環境依存ファイルを管理するためのディレクトリ。 ディレクトリ名は、環境を識別する名前を指定する。
(3)	環境毎の設定ファイルを管理するためのディレクトリ。 サブディレクトリの構成や管理する設定ファイルは、(4) と同様。
(4)	ローカル開発環境用の設定ファイルを管理するためのディレクトリ。
(5)	ローカル開発環境用の Bean 定義を行う。
(6)	ローカル開発環境用のプロパティを定義する。 WSDL の URL など環境ごとに変更の可能性がある値を設定する。
(7)	ローカル開発環境用のログ出力定義を行う。

wsimport について

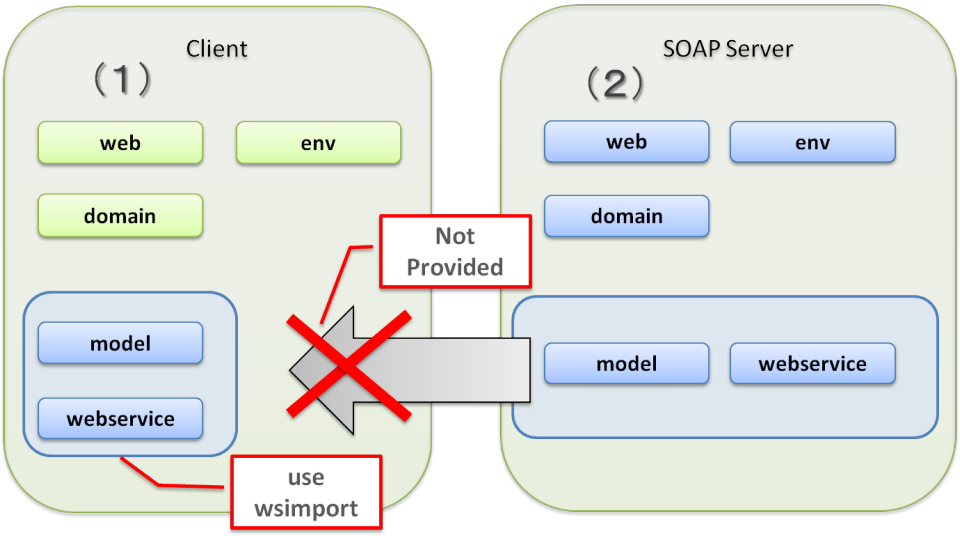
wsimport は Java SE に同梱されるコマンドライン・ツールである。

WSDL ファイルを読み取り、Web サービスを呼び出すことが可能な Java クラス（オプションによってはソースも）を出力するツールである。

wsimport の使い道

本ガイドラインでは、wsimport は以下の図のような場合に使用することを推奨している。

クライアント作成時に、SOAP サーバで使用される Domain Object や Web サービスインターフェースが使用できない場合でも、wsimport を使用することで Web サービスの実行ができるようになる。



wsimport の使い方

JDK の bin フォルダに格納されており、パスを通すだけで使用可能になる。
コマンドライン上で以下のようにコマンドを実行すると、ソースファイルがカレントディレクトリに作成される。

```
# (1)
wsimport -keep -p [出力するソースのパッケージ名] -s [出力するソースを格納する場所] [wsdl の URL]
```

項番	説明
(1)	wsimport の引数として WSDL の URL を指定する。 オプションとして以下を使用する。 • -keep ソースも出力する。 • -p 出力するソースのパッケージを指定する。 • -s 出力するソースを格納する場所を指定する。 その他オプションについては、Java Platform, Standard Edition Tools Reference -Web Services(wsimport)-を参照されたい。

注釈: wsimport はデフォルトの挙動として class ファイルのみが出力される。動かすだけなら問題はないが、デバッグなどを実行したい場合に備え keep オプションを付けてソースも保存することを推奨する。

例えば、以下のようなコマンドとなる。

```
wsimport -keep -p com.example.ws.todo -s c:/tmp http://AAA.BBB.CCC.DDD:XXXX/soap-
↪web/ws/TodoWebService?wsdl
```

作成されるソースは公開されている Web サービスに依存するが、本ガイドラインで使用している以下の Java クラスが出力される。

- Web サービスインターフェース（ソース例では `TodoWebService.java`）
- Domain Object（ソース例では `Todo.java`）

wsimport で生成したクラスを 1 つのクライアントプロジェクトのみでしか使用しない場合は、これらを domain プロジェクトへ配置すればよい。

生成したクラスはインフラストラクチャ層 (*Integration System Connector*) に所属するが、プロジェクト構成の Note で示したように通常は domain プロジェクトに含めても問題ない。

生成したクラスを複数のクライアントで使用する場合は、*SOAP サーバ用にプロジェクトの設定を変更する*をもとに、model プロジェクトと webservice プロジェクトを作成し、それぞれのクライアントから参照して使用することが望ましい。

注釈: 出力される Java クラスは上記以外にも出力される。出力されたソースのみでクライアントを作成可能なソースである。ただし、本ガイドラインではクライアントは、`org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean` を使用する方針であるため、その他の Java クラスは使用しないことを推奨する。

Tomcat 上での Web サービス開発

本ガイドラインでは、Java EE サーバ上の JAX-WS を使う前提で記述されているが、Tomcat の場合、JAX-WS 実装が存在しない。

そのため、ここでは SOAP サーバが Tomcat の場合、JAX-WS の実装プロダクトとして Apache CXF を使用する。設定を変更して `CXFServlet` を使用する必要がある。

Apache CXF を使用する場合は、WebService クラスの実装方式は以下の 2 つが存在する。

1. POJO で Web サービス実装クラスを記述する方式
2. `SpringBeanAutowiringSupport` を継承して Web サービス実装クラスを作成する方式 (これまで説明してきた方法)

1 の場合、Web サービス実装クラスが POJO になるため、単体試験などをしやすくなる。ただし、この方式は Tomcat 以外の AP サーバでは、うまく動作しないことがある。そのため、ガイドライン本体では、この方式ではなく 2 の方式での実現を記述しているが、Tomcat のみを使用する場合、この 1 の方式を使用したほうがメリットが多いのでこちらを推奨する。

2 の場合、他の AP サーバ同様に実装をすることができる。運用は Java EE サーバであるが、開発中は Tomcat を使用せざるをえないケースではこちらの方式を利用されたい。

CXFServlet を使用する場合の設定

`CXFServlet` を使用するため、`pom.xml` にライブラリの設定を記述する。

```
<!-- (1) -->
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-frontend-jaxws</artifactId>
    <version>3.1.4</version>
</dependency>
```

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http</artifactId>
    <version>3.1.4</version>
</dependency>
```

項番	説明
(1)	CXFServlet を使用するため、Apache CXF ライブラリへの依存関係を追加する。

次に web.xml に SOAP Web Service を受け付ける CXFServlet を定義する。

```
<!-- (1) -->
<servlet>
    <servlet-name>cxfServlet</servlet-name>
    <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
    <init-param>
        <param-name>config-location</param-name>
        <param-value>classpath:/META-INF/spring/cxf-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<!-- (2) -->
<servlet-mapping>
    <servlet-name>cxfServlet</servlet-name>
    <url-pattern>/ws/*</url-pattern>
</servlet-mapping>
```

項番	説明
(1)	org.apache.cxf.transport.servlet.CXFServlet のサーブレット定義を行う。 config-location には、後述する cxf-servlet.xml のパスを指定する。
(2)	定義したサーブレットへのマッピングを定義する。この場合、コンテキスト名/ws 配下に Web サービスが作成される。

POJO 方式で必要な設定

Web サービス実装クラスをエンドポイントとして設定する。

[server projectName]-web/src/main/resources/META-INF/spring/cxf-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.
↪w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jaxws="http://cxf.apache.org/jaxws" xmlns:soap="http://cxf.apache.org/
↪bindings/soap"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
         https://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         https://www.springframework.org/schema/context/spring-context.xsd
         http://cxf.apache.org/jaxws
         http://cxf.apache.org/schemas/jaxws.xsd
         http://cxf.apache.org/bindings/soap
         http://cxf.apache.org/schemas/configuration/soap.xsd">

    <!-- (1) -->
    <jaxws:endpoint id="todoWebEndpoint" implementor="#todoWebServiceImpl"
        address="/TodoWebService" />

</beans>
```

項番	説明
(1)	公開するエンドポイントを定義する。 `implementor` 属性に、DI コンテナに登録済みの Web サービスクラスの bean 名 (「#bean 名」形式) を指定する。 `address` 属性に Web サービスを公開するアドレスを指定する。 アドレスは、公開するエンドポイントのパス部分のみ記述する。 属性の詳細については Apache CXF JAX-WS Configuration を参照されたい。

`TodoWebServiceImpl` を POJO として作成する。

[server projectName]-web/src/main/java/com/example/ws/todo/TodoWebServiceImpl.java

```
package com.example.ws.todo;

import java.util.List;

import javax.jws.HandlerChain;
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.soap.SOAPBinding;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.context.support.SpringBeanAutowiringSupport;

import org.springframework.stereotype.Component;

import com.example.domain.model.Todo;
import com.example.domain.service.TodoService;
import com.example.ws.webfault.WebFaultException;
import com.example.ws.exception.WsExceptionHandler;
import com.example.ws.todo.TodoWebService;

// (1)
@Component
@WebService(
      portName = "TodoWebPort",
      serviceName = "TodoWebService",
      targetNamespace = "http://example.com/todo",
      endpointInterface = "com.example.ws.todo.TodoWebService")
@BindingType(SOAPBinding.SOAP12HTTP_BINDING)
// (2)
public class TodoWebServiceImpl implements TodoWebService {

    // omitted

}
```

項番	説明
(1)	`@Component` を付けて、DI コンテナへの登録を行う。
(2)	コンポーネントスキャンにて DI コンテナへの登録が可能であるため、POJO として作成する。つまり、`org.springframework.web.context.support.SpringBeanAutowiringSupport` を継承する必要がなくなる。

SpringBeanAutowiringSupport を継承する方式で必要な設定

CXFServlet 用の Bean 定義ファイルに、SOAP のエンドポイントとなるクラス名およびアドレスを定義する。

[server projectName]-web/src/main/resources/META-INF/spring/cxf-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.
↪w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jaxws="http://cxf.apache.org/jaxws" xmlns:soap="http://cxf.apache.org/
↪bindings/soap"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
         https://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         https://www.springframework.org/schema/context/spring-context.xsd
         http://cxf.apache.org/jaxws
         http://cxf.apache.org/schemas/jaxws.xsd
         http://cxf.apache.org/bindings/soap
         http://cxf.apache.org/schemas/configuration/soap.xsd">
    <!-- (1) -->
    <jaxws:endpoint id="todoWebEndpoint" implementor="com.example.ws.todo.
↪TodoWebServiceImpl"
        address="/TodoWebService" />

</beans>
```

項番	説明
(1)	公開するエンドポイントを定義する。 `implementor` 属性に公開する Web サービスの実装クラスを指定する。 `address` 属性に Web サービスを公開するアドレスを指定する。 アドレスは、公開するエンドポイントのパス部分のみ記述する。 属性の詳細については Apache CXF JAX-WS Configuration を参照されたい。

第 6 章

データアクセス

本ガイドラインで想定しているアーキテクチャについて説明する。

6.1 データベースアクセス（共通編）

6.1.1 Overview

本節では、RDBMS で管理されているデータにアクセスする方法について、説明する。

O/R Mapper に依存する部分については、

- データベースアクセス（*MyBatis3* 編）
- データベースアクセス（*JPA* 編）

を参照されたい。

JDBC DataSource について

RDBMS にアクセスする場合、アプリケーションからは、JDBC データソースを参照してアクセスすることになる。

JDBC データソースを使用することにより、JDBC ドライバーのロード、接続情報（接続 URL、接続ユーザ、パスワードなど）の設定を、アプリケーションから排除することができる。

そのため、アプリケーションからは、使用する RDBMS やデプロイする環境を、意識する必要がなくなる。

JDBC データソースの実装は、アプリケーションサーバ、OSS ライブラリ、Third-Party ライブラリ、Spring Framework などから提供されているので、プロジェクト要件や、デプロイ環境にあったデータソースの選定が必要になる。

以下に、代表的なデータソース 3 種類の紹介を行う。

- アプリケーションサーバ提供の *JDBC* データソース

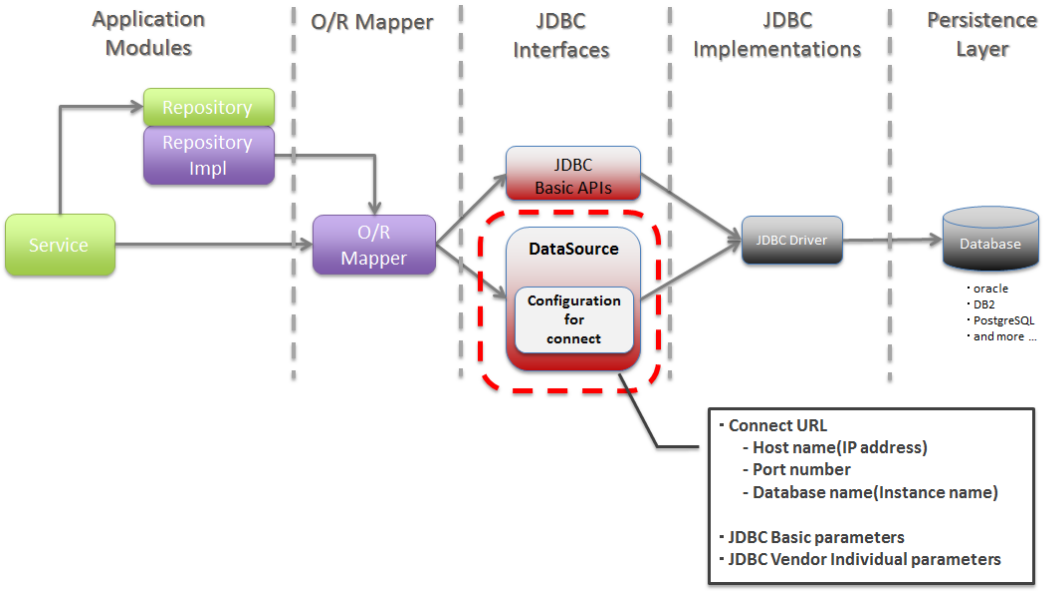


図 6.1 **Picture - About JDBC DataSource**

- *OSS/Third-Party ライブラリ提供の JDBC データソース*
- *Spring Framework 提供の JDBC データソース*

アプリケーションサーバ提供の JDBC データソース

Web アプリケーションでデータソースを使用する場合、アプリケーションサーバから提供される JDBC データソースを使うのが一般的である。

アプリケーションサーバから提供される JDBC データソースは、コネクションプーリング機能など、Web アプリケーションで使うために必要な機能が、標準で提供されている。

表 6.1　アプリケーションサーバから提供されているデータソース

項番	アプリケーションサーバ	参照ページ
1.	Apache Tomcat 9.0	Apache Tomcat 9.0 User Guide(The Tomcat JDBC Connection Pool) を参照されたい。 Apache Tomcat 9.0 User Guide(JNDI Datasource HOW-TO)(Apache Commons DBCP 2) を参照されたい。
2.	Apache Tomcat 8.5	Apache Tomcat 8.5 User Guide(The Tomcat JDBC Connection Pool) を参照されたい。 Apache Tomcat 8.5 User Guide(JNDI Datasource HOW-TO)(Apache Commons DBCP 2) を参照されたい。
3.	Apache Tomcat 7	Apache Tomcat 7 User Guide(The Tomcat JDBC Connection Pool) を参照されたい。 Apache Tomcat 7 User Guide(JNDI Datasource HOW-TO)(Apache Commons DBCP) を参照されたい。
4.	Oracle WebLogic Server 12c	Oracle WebLogic Server Product Documentation を参照されたい。
5.	IBM WebSphere Application Server Version 9.0	WebSphere Application Server Online information center を参照されたい。
6.	JBoss Enterprise Application Platform 7.2	JBoss Enterprise Application Platform 7.2 Product Documentation を参照されたい。
7.	JBoss Enterprise Application Platform 6.4	JBoss Enterprise Application Platform 6.4 Product Documentation を参照されたい。

OSS/Third-Party ライブラリ提供の JDBC データソース

アプリケーションサーバから提供される JDBC データソースを使わない場合は、OSS/Third-Party ライブラリから提供されている JDBC データソースを使用する。

本ガイドラインでは、「Apache Commons DBCP」のみ紹介するが、他のライブラリを使ってもよい。

表 6.2 OSS/Third-Party ライブラリから提供されている JDBC データソース

項番	ライブラリ名	説明
1.	Apache Commons DBCP	Apache Commons DBCP を参照されたい。

Spring Framework 提供の JDBC データソース

Spring Framework から提供されている JDBC データソースの実装クラスは、コネクションプーリング機能がないため、Web アプリケーションのデータソースとして使用する事はない。

Spring Framework では、JDBC データソースの実装クラスと、JDBC データソースのアダプタクラスを提供しているが、利用するケースが限定的なので、Appendix の *Spring Framework から提供されている JDBC データソースクラス*として紹介する。

トランザクションの管理方法について

Spring Framework の機能を使って、トランザクション管理を行う場合、プロジェクト要件や、デプロイ環境にあった PlatformTransactionManager の選定が必要になる。

詳細は、ドメイン層の実装のトランザクション管理を使うための設定についてを参照されたい。

トランザクション境界/属性の宣言について

トランザクション境界及びトランザクション属性の宣言は、Service にて、`@Transactional` アノテーションを指定することで実現する。

詳細は、ドメイン層の実装のトランザクション管理についてを参照されたい。

データの排他制御について

データを更新する場合、データの一貫性および整合性を保障するために、排他制御を行う必要がある。

データの排他制御については、排他制御を参照されたい。

例外ハンドリングについて

Spring Framework では、JDBC の例外 (`java.sql.SQLException`) や、O/R Mapper 固有の例外を、Spring Framework から提供しているデータアクセス例外 (`org.springframework.dao.DataAccessException` のサブクラス) に変換する機能がある。

Spring Framework のデータアクセス例外へ変換しているクラスについては、Appendix の *Spring Framework から提供されているデータアクセス例外へ変換するクラス*を参照されたい。

変換されたデータアクセス例外は、基本的にはアプリケーションコードでハンドリングする必要はないが、一部のエラー（一意制約違反、排他エラーなど）については、要件によっては、ハンドリングする必要がある。

データアクセス例外をハンドリングする場合、`DataAccessException` を catch するのではなく、エラー内容を通知するサブクラスの例外を catch すること。
以下に、アプリケーションコードでハンドリングする可能性がある代表的なサブクラスを紹介する。

表 6.3　ハンドリングする可能性がある **DB** アクセス例外のサブクラス

項番	クラス名	説明
1.	org.springframework.dao. DuplicateKeyException	一意制約違反が発生した場合に発生する例外。
2.	org.springframework.dao. OptimisticLockingFailureException	楽観ロックに成功しなかった場合に発生する例外。他の処理によって同一データが更新されていた場合に発生する。 本例外は、O/R Mapper として JPA を使用する場合に発生する例外である。MyBatis には楽観ロックを行う機能がないため、O/R Mapper 本体から本例外が発生することはない。
3.	org.springframework.dao. PessimisticLockingFailureException	悲観ロックに成功しなかった場合に発生する例外。他の処理で同一データがロックされており、ロック解放待ちのタイムアウト時間を超えてもロックが解放されない場合に発生する。

注釈: O/R Mapper に MyBatis を使用して楽観ロックを実現する場合は、Service や Repository の処理として楽観ロック処理を実装する必要がある。

本ガイドラインでは、楽観ロックに失敗したことを、Controller に通知する方法として、`OptimisticLockingFailureException` およびその子クラスの例外を発生させることを推奨する。

理由は、アプリケーション層の実装 (Controller の実装) を、使用する O/R Mapper に依存させないためである。

課題: JPA(Hibernate) を使用すると、現状意図しないエラーとなることが発覚している。

- 一意制約違反が発生した場合、`DuplicateKeyException` ではなく、`org.`

springframework.dao.DataIntegrityViolationException が発生する。

下記は、一意制約違反を、ビジネス例外として扱う実装例である。

```
try {
    accountRepository.saveAndFlash(account);
} catch(DuplicateKeyException e) { // (1)
    throw new BusinessException(ResultMessages.error().add("e.xx.xx.0002"),
↪e); // (2)
}
```

項番	説明
(1)	一意制約違反が発生した場合に発生する例外（DuplicateKeyException）を catch する。
(2)	データが重複している旨を伝えるビジネス例外を発生させている。 例外を catch した場合は、必ず原因例外 (「e」) をビジネス例外に指定すること。

複数データソースについて

アプリケーションによっては、複数のデータソースが必要になる場合がある。

以下に、複数のデータソースが必要になる代表的なケースを紹介する。

表 6.4　複数のデータソースが必要になる代表的なケース

項番	ケース	例	特徴
1.	データ (テーブル) の分類毎にデータベースやスキーマがわかれている場合。	顧客情報を保持するテーブル群と請求情報を保持するテーブル群が別々のデータベースやスキーマに格納されている場合など。	処理で扱うデータは決まっているので、静的に使用するデータソースを決定することができる。
2.	利用者（ログインユーザ）によって使用するデータベースやスキーマが分かれている場合。	利用者の分類毎にデータベースやスキーマがわかれている場合など（マルチテナント等）。	利用者によって使用するデータソースが異なるため、動的に使用するデータソースを決定する必要がある。

共通ライブラリから提供しているクラスについて

共通ライブラリから、以下の処理を行うクラスを提供している。

共通ライブラリの詳細については、以下を参照されたい。

- *LIKE* 検索時のエスケープについて
- *Sequencer* について

6.1.2 How to use

データソースの設定

アプリケーションサーバで定義した **DataSource** を使用する場合の設定

アプリケーションサーバで定義したデータソースを使用する場合は、Bean 定義ファイルに、JNDI 経由で取得したオブジェクトを、bean として登録するための設定を行う必要がある。

以下に、データベースは PostgreSQL、アプリケーションサーバは Tomcat8.5 を使用する際の、設定例を示す。

- `xxx-context.xml` (Tomcat の設定ファイル)

```
<!-- (1) -->
<Resource
   type="javax.sql.DataSource"
   name="jdbc/SampleDataSource"
   driverClassName="org.postgresql.Driver"
   url="jdbc:postgresql://localhost:5432/terasoluna"
   username="postgres"
   password="postgres"
   defaultAutoCommit="false"
   /> <!-- (2) -->
```

- `xxx-env.xml`

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/SampleDataSource" /> <!--␣
↪(3) -->
```

項番	属性名	説明
(1)	-	データソースを定義する。
	type	リソースの種類を指定する。`javax.sql.DataSource` を指定する。
	name	リソース名を指定する。ここで指定した名前が JNDI 名となる。
	driverClassName	JDBC ドライバクラスを指定する。例では、PostgreSQL から提供されている JDBC ドライバクラスを指定する。
	url	接続 URL を指定する。 【環境に合わせて変更が必要】
	username	接続ユーザ名を指定する。【環境に合わせて変更が必要】
	password	接続ユーザのパスワードを指定する。【環境に合わせて変更が必要】
	defaultAutoCommit	自動コミットフラグのデフォルト値を指定する。false を指定する。トランザクション管理下であれば強制的に false になる。
(2)	-	Tomcat8.5 の場合、factory 属性を省略すると tomcat-jdbc-pool が使用される。 設定項目の詳細については、Attributes of The Tomcat JDBC Connection Pool を参照されたい。
(3)	-	データソースの JNDI 名を指定する。Tomcat の場合は、データソース定義時のリソース名「(1)-name」に指定した値を指定する。

Bean 定義した DataSource を使用する場合の設定

アプリケーションサーバから提供されているデータソースを使わずに、

OSS/Third-Party ライブラリから提供されているデータソースや、Spring Framework から提供されている JDBC データソースを使用する場合は、Bean 定義ファイルに DataSource クラスの bean 定義が必要となる。

以下に、データベースは PostgreSQL、データソースは Apache Commons DBCP を使用する際の、設定例を示す。

- xxx-env.xml

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
    destroy-method="close">                                          <!--
↪(1) (8) -->
    <property name="driverClassName" value="org.postgresql.Driver" /> <!--
↪(2) -->
    <property name="url" value="jdbc:postgresql://localhost:5432/terasoluna" /
↪> <!-- (3) -->
    <property name="username" value="postgres" />                    <!--
↪(4) -->
    <property name="password" value="postgres" />                    <!--
↪(5) -->
    <property name="defaultAutoCommit" value="false"/>               <!-- (6)
↪-->
    <!-- (7) -->
</bean>
```

項番	説明
(1)	データソースの実装クラスを指定する。例では、Apache Commons DBCP から提供されているデータソースクラス (`org.apache.commons.dbcp2.BasicDataSource`) を指定する。
(2)	JDBC ドライバクラスを指定する。例では、PostgreSQL から提供されている JDBC ドライバクラスを指定する。
(3)	接続 URL を指定する。 【環境に合わせて変更が必要】
(4)	接続ユーザ名を指定する。【環境に合わせて変更が必要】
(5)	接続ユーザのパスワードを指定する。【環境に合わせて変更が必要】
(6)	自動コミットフラグのデフォルト値を指定する。false を指定する。トランザクション管理下であれば、強制的に false になる。
(7)	BasicDataSource には上記以外に、JDBC 共通の設定値の指定、JDBC ドライバー固有のプロパティ値の指定、コネクションプーリング機能の設定値の指定を行うことができる。 設定項目の詳細については、DBCP Configuration を参照されたい。
(8)	設定例では値を直接指定しているが、環境によって設定値がかわる項目については、Placeholder(${…}) を使用して、実際の設定値はプロパティファイルに指定すること。 Placeholder については、Spring Reference Document の `PropertyPlaceholderConfigurer` を参照されたい。

トランザクション管理を有効化するための設定

トランザクション管理を有効化するための基本的な設定は、ドメイン層の実装のトランザクション管理を使うための設定についてを参照されたい。

PlatformTransactionManager については、使用する O/R Mapper によって使うクラスがかわるので、詳細設定は、

- データベースアクセス（*MyBatis3* 編）

- データベースアクセス（*JPA* 編）

を参照されたい。

JDBC の Debug 用ログの設定

O/R Mapper(MyBatis, Hibernate) で出力されるログより、さらに細かい情報が必要な場合、
log4jdbc(log4jdbc-remix) を使って出力される情報が有効である。

log4jdbc の詳細については、log4jdbc project page を参照されたい。

log4jdbc-remix の詳細については、log4jdbc-remix project page を参照されたい。

> **警告：** **log4jdbc-remix** が提供している **Log4jdbcProxyDataSource** を使用していると、ログレベルを」**debug**」以外に設定しても、オーバーヘッドが少なからず発生する。そのため、本設定はデバッグ用として使用し、性能試験及び商用環境にリリースする場合は **Log4jdbcProxyDataSource** を経由せずにデータベースへ接続することを推奨する。

log4jdbc 提供のデータソースの設定

- `xxx-env.xml`

```
<jee:jndi-lookup id="realDataSource" jndi-name="jdbc/SampleDataSource" /> <!--
↪ (1) -->

<bean id="dataSource" class="net.sf.log4jdbc.Log4jdbcProxyDataSource"> <!--␣
↪(2) -->
    <constructor-arg ref="realDataSource" /> <!-- (3) -->
</bean>
```

項番	説明
(1)	データソースの実体を定義する。例では、アプリケーションサーバから JNDI 経由で取得したデータソースを使用している。
(2)	log4jdbc より提供されている `net.sf.log4jdbc.Log4jdbcProxyDataSource` を指定する。
(3)	データソースの実体となる bean を、コンストラクタに指定する。

> **警告：** 性能試験及び商用環境にリリースする場合、データソースとして **Log4jdbcProxyDataSource** は使用しないこと。

具体的には、(2) と (3) の設定を外し、realDataSource の bean 名を dataSource に変更する。

log4jdbc 用ロガーの設定

- logback.xml

```
<!-- (1) -->
<logger name="jdbc.sqltiming">
    <level value="debug" />
</logger>

<!-- (2) -->
<logger name="jdbc.sqlonly">
    <level value="warn" />
</logger>

<!-- (3) -->
<logger name="jdbc.audit">
    <level value="warn" />
</logger>

<!-- (4) -->
<logger name="jdbc.connection">
    <level value="warn" />
</logger>

<!-- (5) -->
<logger name="jdbc.resultset">
    <level value="warn" />
</logger>

<!-- (6) -->
<logger name="jdbc.resultsettable">
    <level value="debug" />
</logger>
```

項番	説明
(1)	バインド変数に値が設定された状態の SQL 文と、SQL の実行時間を出力するためのロガー。値がバインドされた形式の SQL が出力されるので、DB アクセスツールに貼りつけて実行する事ができる。
(2)	バインド変数に値が設定された状態の SQL 文を、出力するためのロガー。(1) との違いは、実行時間が出力されない。
(3)	ResultSet インタフェースを除く、JDBC インタフェースのメソッド呼び出し（引数と、返り値）を出力するためのロガー。JDBC 関連で問題が発生した時の解析に有効なログであるが、出力されるログの量が多い。
(4)	Connection の接続/切断イベントと使用中の接続数を出力するためのロガー。接続リークが発生時の解析に有効なログであるが、接続リークの問題がなければ、出力する必要はない。
(5)	ResultSet インタフェースに対するメソッド呼び出し（引数と、返り値）を出力するためのロガー。取得結果が、想定と異なった時の解析に有効なログであるが、出力されるログの量が多い。
(6)	ResultSet の中身を確認しやすい形式にフォーマットして出力するためのロガー。取得結果が、想定と異なった時の解析に有効なログであるが、出力されるログの量が多い。

警告: ロガーによっては大量にログが出力されるので、必要なロガーのみ定義、または出力対象にすること。

上記サンプルでは、開発中の非常に有効なログを出力するロガーについて、ログレベルを `debug` に設定している。その他のロガーについては、必要に応じて `debug` に設定する必要がある。

性能試験及び商用環境にリリースする場合、正常終了時に log4jdbc 用のロガーによってログが出力されないようにすること。

具体的には、ログレベルを `warn` に設定する。

log4jdbc のオプションの設定

クラスパス直下に、log4jdbc.properties というプロパティファイルを配置することで、log4jdbc のデフォルトの動作をカスタマイズすることができる。

- log4jdbc.properties

```
# (1)
log4jdbc.dump.sql.maxlinelength=0
# (2)
```

項番	説明
(1)	SQL 分の折り返し文字数を指定する。0 を指定すると、折り返しはされない。
(2)	オプションの詳細については、log4jdbc project page -Options-を参照されたい。

6.1.3 How to extend

動的にデータソースを切り替えるための設定

複数のデータソースを定義し、動的に切り替えを行うには、
org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource を継承したクラスを作成し、どのような条件でデータソースを切り替えるかを実装する必要がある。

具体的には determineCurrentLookupKey メソッドの戻り値となるキーとデータソースをマッピングさせることによって、これを実現する。キーの選択には通常、認証ユーザー情報、時間、ロケール等のコンテキスト情報を使用する。

AbstractRoutingDataSource の実装

AbstractRoutingDataSource を拡張して作成した DataSource を、通常のデータソースと同じように使用することでデータソースの動的な切り替えが実現できる。

以下に、時間によってデータソースを切り替える例を示す。

- AbstractRoutingDataSource を継承したクラスの実装例 ‘

```
package com.examples.infra.datasource;

import javax.inject.Inject;

import org.joda.time.DateTime;
import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;
import org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory;

public class RoutingDataSource extends AbstractRoutingDataSource { // (1)

    @Inject
    JodaTimeDateFactory dateFactory; // (2)

    @Override
    protected Object determineCurrentLookupKey() { // (3)

        DateTime dateTime = dateFactory.newDateTime();
        int hour = dateTime.getHourOfDay();

        if (7 <= hour && hour <= 23) { // (4)
            return "OPEN"; // (5)
        } else {
            return "CLOSE";
        }
    }
}
```

項番	説明
(1)	AbstractRoutingDataSource を継承する。
(2)	時刻を取得するため、JodaTimeDateFactory を使用する。詳細は、システム時刻を参照のこと。
(3)	determineCurrentLookupKey メソッドを実装する。このメソッドの返り値と後述する bean 定義ファイル内の targetDataSources に定義した key をマッピングすることにより使用するデータソースが決定される。
(4)	メソッド内で、コンテキスト情報（ここでは時間）を参照し、キーの切り替えを行う。ここは業務用件に合わせて実装する必要がある。このサンプルは、時刻が「7:00 から 23:59 まで」と「0:00 から 6:59 まで」で違うキーを返すように実装されている。
(5)	後述する bean 定義ファイル内の targetDataSources とマッピングさせる key を返す。

データソースの定義

作成した AbstractRoutingDataSource 拡張クラスを bean 定義ファイルに定義する。

- xxx-env.xml

```
<bean id="dataSource"
    class="com.examples.infra.datasource.RoutingDataSource">  <!-- (1) -->
    <property name="targetDataSources">  <!-- (2) -->
        <map>
            <entry key="OPEN" value-ref="dataSourceOpen" />
            <entry key="CLOSE" value-ref="dataSourceClose" />
        </map>
    </property>
    <property name="defaultTargetDataSource" ref="dataSourceDefault" />  <!--
↪(3) -->
</bean>
```

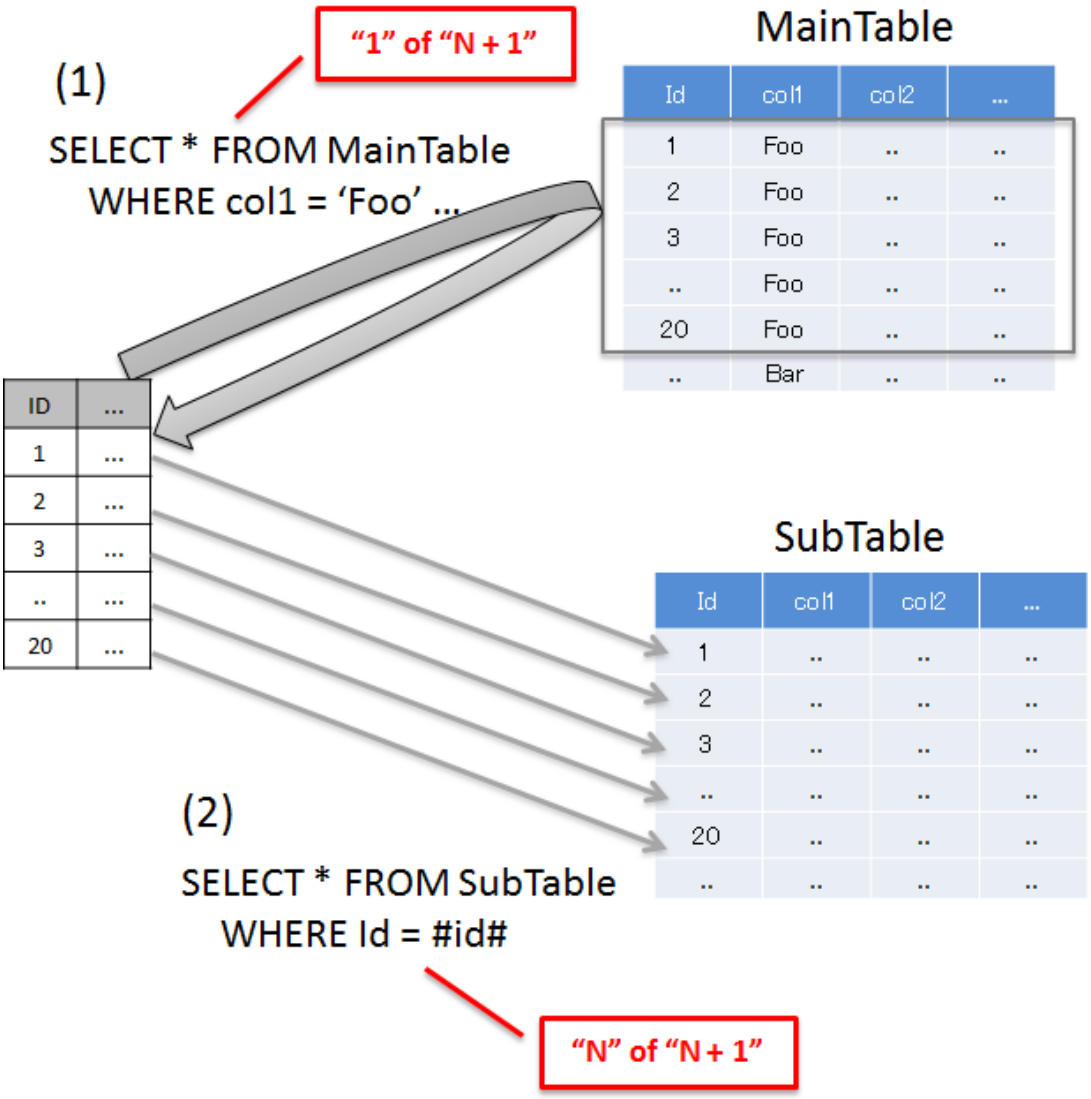
項番	説明
(1)	先ほど作成した AbstractRoutingDataSource を継承したクラスを定義する。
(2)	使用するデータソースを定義する。key は determineCurrentLookupKey メソッドで返却しうる値を定義する。value-ref には key ごとに使用するデータソースを指定する。データソースの設定をもとに切り替えるデータソースの個数分、定義を行う必要がある。
(3)	determineCurrentLookupKey メソッドで指定した key が targetDataSources に存在しない場合は、このデータソースが使用される。実装例の場合、デフォルトが使用されることはないが、今回は説明のため、defaultTargetDataSource を定義している。

6.1.4 how to solve the problem

N+1 問題の対策方法

N+1 問題とは、データベースから取得するレコード数に比例して実行される SQL の数が増えることにより、データベースへの負荷およびレスポンスタイムの劣化を引き起こす問題のことである。

以下に、具体的をあげる。



項番	説明
(1)	検索条件に一致するレコードを、メインとなるテーブルから検索する。 上記例では、 MainTable テーブルの col1 カラムが、`'Foo'` のレコードを取得しており、20 件のレコードが取得されている。
(2)	(1) で検索した各レコードに対して、関連レコードを関連テーブルから取得する。 上記例では、SubTable テーブルの id カラムが、(1) で取得したレコードの id カラムと同じレコードを取得している。 この **SQL** は、**(1)** で取得されたレコード件数分、実行される。

上記例では、合計で **21** 回の **SQL** が発行されることになる。

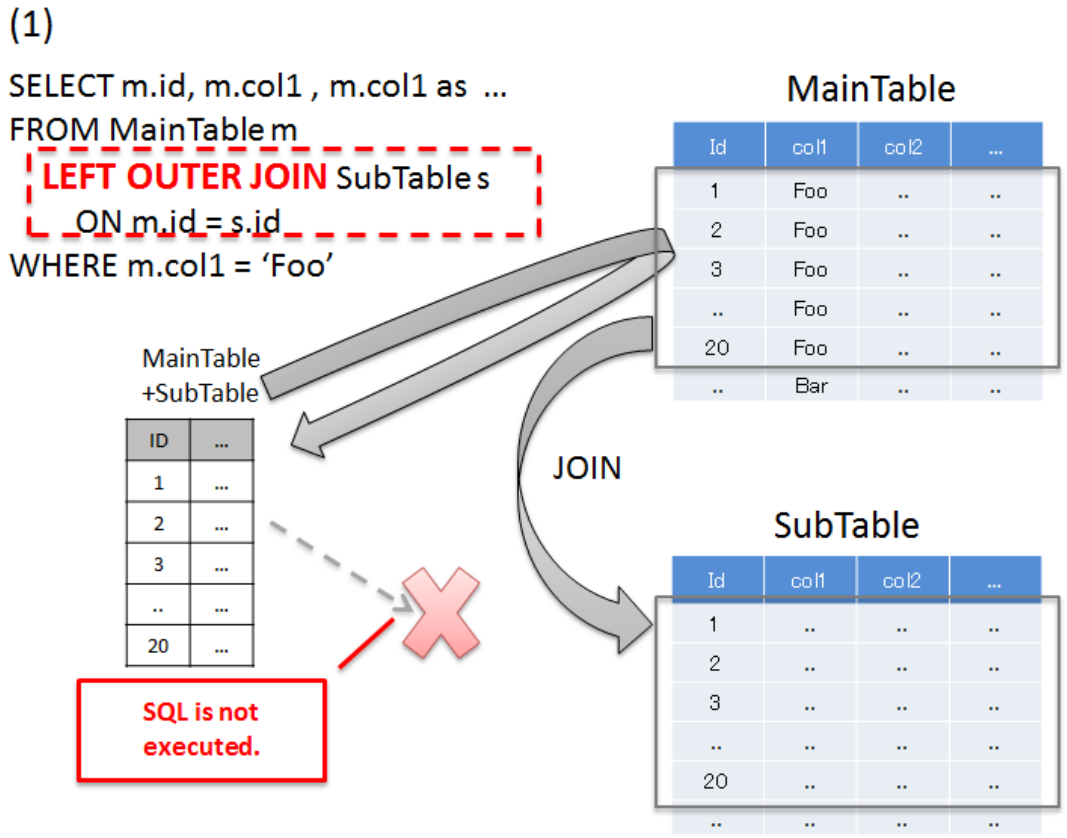
仮に関連テーブルが 3 テーブルあると、合計で **61** 回の **SQL** が発行されることになるため、対策が必要となる。

N+1 問題の解決方法の代表例を、以下に示す。

JOIN(Join Fetch) を使用して解決する

関連テーブルを JOIN することで、1 回の SQL でメインのテーブルと関連テーブルのレコードを取得する。

関連テーブルとの関係が、1:1 の場合は、この方法によって解決することを検討すること。



項番	説明
(1)	検索条件に一致するレコードを検索する際に、関連テーブルを JOIN することで、メインとなるテーブルと関連テーブルから、レコードを一括で取得する。 上記例では、 MainTable テーブルの col1 カラムが`'Foo'` のレコードと、検索条件に一致したレコードの id が一致する SubTable のレコードを一括で取得している。 カラム名が重複する場合は、別名を付与してどちらのテーブルのカラムなのか識別する必要がある。

JOIN(Join Fetch) を使用すると、**1** 回の **SQL** の発行で必要なデータを全て取得することができる。

注釈: **JPQL** で **JOIN** する場合

JPQL で JOIN する場合の実装例については、*JOIN FETCH について*を参照されたい。

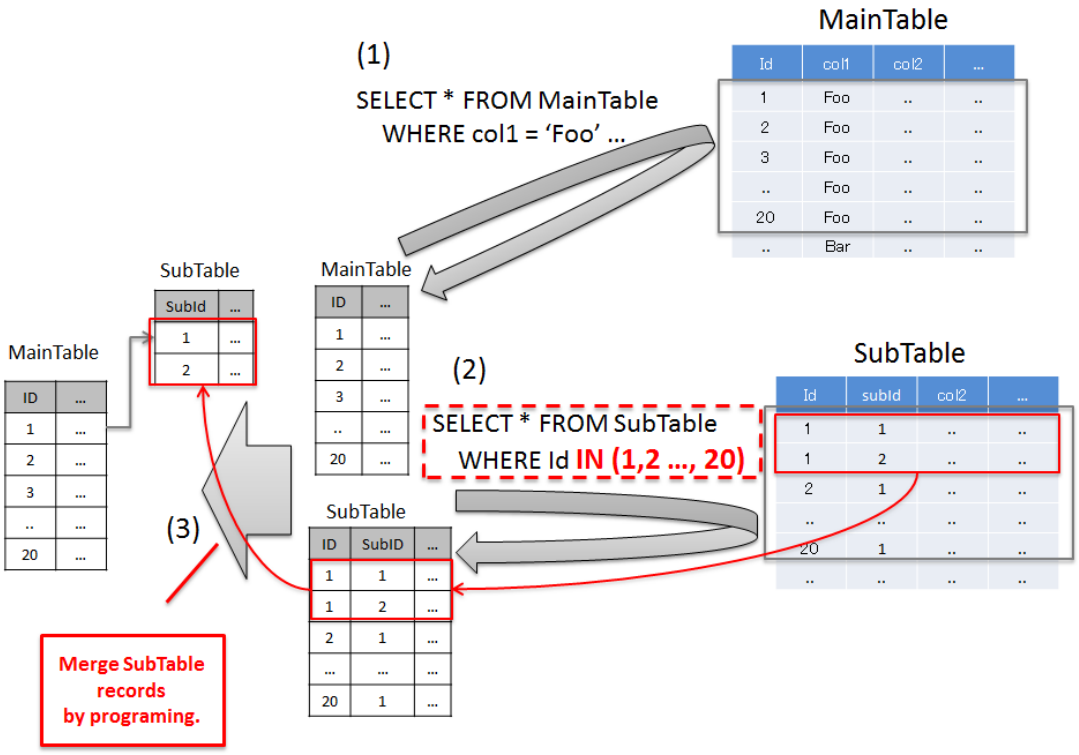
警告: 関連テーブルとの関連が、1:N の場合は、JOIN(Join Fetch) による解決も可能だが、以下の点に注意すること。

- 1:N の関連をもつレコードを JOIN する場合、関連テーブルのレコード数に比例して、無駄なデータを取得することになる。詳細については、一括取得時の注意事項を参照されたい。
- JPA(Hibernate) 使用する際に、1:N の N の部分が、複数ある場合は、N の部分を格納するコレクション型は、`java.util.List` ではなく、`java.util.Set` を使用する必要がある。

関連レコードを一括で取得する事で解決する

1:N の関係が複数あるパターンなどは、関連レコードを一括で取得し、その後プログラミングによって振り分ける方法をとった方がよいケースがある。

関連テーブルとの関係が 1:N の場合は、この方法によって解決することを検討すること。



項番	説明
(1)	検索条件に一致するレコードを、メインとなるテーブルから検索する。 上記例では、 MainTable テーブルの col1 カラムが、`'Foo'` のレコードを取得しており、20件のレコードが取得されている。
(2)	(1) で検索した各レコードに対して、関連レコードを関連テーブルから取得する。 1 レコード毎に取得するのではなく、(1) で取得した各レコードの外部キーに一致するレコードを、一括で取得する。 上記例では、SubTable テーブルの id カラムが、(1) で取得したレコードの id カラムと同じレコードを、IN 句を使用して一括取得している。
(3)	(2) で取得した SubTable のレコードを、(1) で取得したレコードに振り分けマージする。

上記例では、**合計で 2 回の SQL の発行で、必要なデータを取得することができる。**

仮に、関連テーブルが、3 テーブルあっても、**合計で 4 回の SQL の発行で済むことになる。**

注釈: この方法は、SQL の発行を最小限におさえつつ、必要なデータのみ取得することができるという特徴をもつ。関連テーブルのレコードをプログラミングによって振り分ける必要があるが、関連テーブルの数が多い場合や、1:N の N のレコード数が多い場合は、この方法で解決する方がよいケースがある。

6.1.5 Appendix

LIKE 検索時のエスケープについて

LIKE 検索を行う場合は、検索条件として使用する値を、LIKE 検索用にエスケープする必要がある。

共通ライブラリでは、LIKE 検索用のエスケープ処理を行うためのコンポーネントとして、以下のクラスを提供している。

項番	クラス	説明
1.	org.terasoluna.gfw.common.query.QueryEscapeUtils	SQL 及び JPQL のエスケープ処理を行うメソッドを提供するユーティリティクラス。 本クラスでは、 • LIKE 検索用のエスケープ処理を行うメソッドを提供している。
2.	org.terasoluna.gfw.common.query.LikeConditionEscape	LIKE 検索用のエスケープ処理を行うクラス。

注釈: `LikeConditionEscape` クラスは、「LIKE 検索用のワイルドカード文字の扱いに関するバグ」を修正するために、terasoluna-gfw-common 1.0.2.RELEASE から追加したクラスである。

`LikeConditionEscape` クラスは、データベース及びデータベースのバージョンの違いによるワイルドカード文字の違いを吸収する役割を持つ。

共通ライブラリのエスケープ仕様について

共通ライブラリから提供しているエスケープ処理の仕様は、以下の通りである。

- エスケープ文字は「「~」」。
- エスケープ対象文字は、デフォルトでは「「%」，「_」」の 2 文字。

注釈: エスケープ対象文字は、terasoluna-gfw-common 1.0.1.RELEASE までは「「%」，「_」，「％」，「＿」」の 4 文字であったが、「LIKE 検索用のワイルドカード文字の扱いに関するバグ」を修正するために、terasoluna-gfw-common 1.0.2.RELEASE より「「%」，「_」」の 2 文字に変更している。

なお、エスケープ対象文字として全角文字「」％」，「＿」」を含めてエスケープする方法も提供している。

具体的なエスケープ例を以下に示す。

[デフォルト仕様のエスケープ例]

エスケープ対象文字としてデフォルト値を使用する場合のエスケープ例を以下に示す。

項番	対象 文字列	エスケープ後 文字列	エスケープ 有無	解説
1.	「a」	「a」	無	エスケープ対象文字が含まれていないため、エスケープされない。
2.	a~	a~~	有	エスケープ文字が含まれているため、エスケープされる。
3.	a%	a~%	有	エスケープ対象文字が含まれているため、エスケープされる。
4.	a_	a~_	有	No.3 と同様。
5.	_a%	~_a~%	有	エスケープ対象文字が含まれているため、エスケープされる。エスケープ対象文字が複数存在する場合はすべてエスケープされる。
6.	a %	a %	無	No.1 と同様。 terasoluna-gfw-common 1.0.2.RELEASE より、デフォルト仕様では「」%」」はエスケープ対象外の文字として扱う。
7.	a __	a __	無	No.1 と同様。 terasoluna-gfw-common 1.0.2.RELEASE より、デフォルト仕様では「」__」」はエスケープ対象外の文字として扱う。
8.	" "	" "	無	No.1 と同様。
9.	""	""	無	No.1 と同様。
10.	null	null	無	No.1 と同様。

[全角文字を含める場合のエスケープ例]

エスケープ対象文字として全角文字を含める場合のエスケープ例を以下に示す。項番 6 と 7 以外は、デフォルト仕様のエスケープ例を参照されたい。

項番	対象 文字列	エスケープ後 文字列	エスケープ 有無	解説
6.	a％	a~％	有	エスケープ対象文字が含まれているため、エスケープされる。
7.	a＿	a~＿	有	No.6 と同様。

共通ライブラリから提供しているエスケープ用のメソッドについて

共通ライブラリから提供している `QueryEscapeUtils` クラスと `LikeConditionEscape` クラスの LIKE 検索用のエスケープメソッドの一覧を、以下に示す。

項番	メソッド名	説明
1.	toLikeCondition(String)	引数で渡された文字列を LIKE 検索用にエスケープする。 SQL や JPQL 側で一致方法 (前方一致、後方一致、部分一致) を指定する場合は、本メソッドを使用してエスケープのみ行う。
2.	toStartingWithCondition(String)	引数で渡された文字列を LIKE 検索用にエスケープした上で、エスケープ後の文字列の最後尾に 「%」 を付与する。 前方一致検索用の値に変換する場合に使用するメソッドである。
3.	toEndingWithCondition(String)	引数で渡された文字列を LIKE 検索用にエスケープした上で、エスケープ後の文字列の先頭に 「%」 を付与する。 後方一致検索用の値に変換する場合に使用するメソッドである。
4.	toContainingCondition(String)	引数で渡された文字列を LIKE 検索用にエスケープした上で、エスケープ後の文字列の先頭と最後尾に 「%」 を付与する。 部分一致検索用の値に変換する場合に使用するメソッドである。

注釈: No.2, 3, 4 については、SQL や JPQL 側で一致方法 (前方一致、後方一致、部分一致) を指定するのではなく、プログラム側で指定する時に使用するメソッドである。

共通ライブラリの使用方法

LIKE 検索時のエスケープ処理の実装例については、使用する O/R Mapper 向けのドキュメントを参照されたい。

- MyBatis3 を使用する場合は、データベースアクセス（*MyBatis3* 編）の *LIKE* 検索時のエスケープを参照されたい。
- JPA(Spring Data JPA) を使用する場合は、データベースアクセス（*JPA* 編）の *LIKE* 検索時のエスケープについてを参照されたい。

注釈: エスケープ処理を行うために使用する API は、使用するデータベースがサポートしているワイルドカード文字によって使い分ける必要がある。

[ワイルドカードとして「「%」，「_」」(半角文字) のみをサポートしているデータベースの場合]

```
String escapedWord = QueryEscapeUtils.toLikeCondition(word);
```

項番	説明
(1)	QueryEscapeUtils クラスのメソッドを直接使用して、エスケープ処理を行う。

[ワイルドカードとして「」%」，「__」」(全角文字) もサポートしているデータベースの場合]

```
String escapedWord = QueryEscapeUtils.withFullWidth()  // (2)
                         .toLikeCondition(word);       // (3)
```

項番	説明
(2)	QueryEscapeUtils メソッドの withFullWidth() メソッドを呼び出して、LikeConditionEscape クラスのインスタンスを取得する。
(3)	(2) で取得した LikeConditionEscape クラスのインスタンスのメソッドを使用して、エスケープ処理を行う。

Sequencer について

Sequencer は、シーケンス値を取得するための共通ライブラリである。

Sequencer から取得したシーケンス値は、データベースのプライマリキーカラムの設定値などとして使用する。

注釈: 共通ライブラリとして Sequencer を用意した理由

Sequencer を用意した理由は、JPA の機能として提供されている ID 採番機能において、シーケンス値を文字列としてフォーマットする仕組みがないためである。実際のアプリケーション開発では、フォーマットされた文字列をプライマリキーに設定するケースもあるため、共通ライブラリとして Sequencer を提供している。

プライマリキーに設定する値が数値の場合は、JPA の機能として提供されている ID 採番機能を使用することを推奨する。JPA の ID 採番機能については、データベースアクセス（*JPA* 編）の *Entity* の追加を参照されたい。

Sequencer を用意した主な目的は、JPA でサポートされていない機能の補完であるが、JPA と関係ない処理で、シーケンス値が必要な場合に、使用することもできる。

共通ライブラリから提供しているクラスについて

共通ライブラリから提供している Sequencer 機能のクラス一覧を以下に示す。

具体的な使用例については、How to use の共通ライブラリの利用方法を参照されたい。

項番	クラス名	説明
1.	org.terasoluna.gfw.common.sequencer. Sequencer	次のシーケンス値を取得するメソッド (getNext) とシーケンスの現在値を返却するメソッド (getCurrent) を定義しているインタフェース。
2.	org.terasoluna.gfw.common.sequencer. JdbcSequencer	`Sequencer` インタフェースの JDBC 用の実装クラス。 データベースに SQL を発行してシーケンス値を取得するためのクラスである。 データベースのシーケンスオブジェクトから値を取得することを想定したクラスではあるが、データベースに登録されているファンクションを呼び出すことで、シーケンスオブジェクト以外から値を取得することもできる。

共通ライブラリの利用方法

Sequencer を bean 定義する。

- xxx-infra.xml

```
<!-- (1) -->
<bean id="articleIdSequencer" class="org.terasoluna.gfw.common.sequencer.
↪JdbcSequencer">
    <!-- (2) -->
    <property name="dataSource" ref="dataSource" />
    <!-- (3) -->
    <property name="sequenceClass" value="java.lang.String" />
    <!-- (4) -->
    <property name="nextValueQuery"
        value="SELECT TO_CHAR(NEXTVAL('seq_article'),'AFM0000000000')" />
    <!-- (5) -->
    <property name="currentValueQuery"
        value="SELECT TO_CHAR(CURRVAL('seq_article'),'AFM0000000000')" />
</bean>
```

項番	説明
(1)	`org.terasoluna.gfw.common.sequencer.Sequencer` インタフェースを実装したクラスを、bean 定義する。 上記例では、SQL を発行してシーケンス値を取得するためのクラス (`JdbcSequencer`) を指定している。
(2)	シーケンス値を取得する SQL を、実行するデータソースを指定する。
(3)	取得するシーケンス値の型を指定する。 上記例では、SQL で文字列へ変換しているので、`java.lang.String` 型を指定している。
(4)	次のシーケンス値を取得するための SQL を指定する。 上記例では、データベース (PostgreSQL) のシーケンスオブジェクトから取得したシーケンス値を、文字列としてフォーマットしている。 データベースのから取得したシーケンス値が、」`1`」 の場合、`A0000000001` が `Sequencer#getNext()` メソッドの返り値として返却される。
(5)	現在のシーケンス値を取得するための SQL を指定する。 データベースのから取得したシーケンス値が、」`2`」 の場合、`A0000000002` が `Sequencer#getCurrent()` メソッドの返り値として返却される。

bean 定義した Sequencer からシーケンス値を取得する。

- Service

```
// omitted

// (1)
@Inject
@Named("articleIdSequencer") // (2)
Sequencer<String> articleIdSequencer;

// omitted

@Transactional
public Article createArticle(Article inputArticle) {
```

```
    String articleId = articleIdSequencer.getNext(); // (3)
    inputArticle.setArticleId(articleId);

    Article savedArticle = articleRepository.save(inputArticle);

    return savedArticle;
}
```

項番	説明
(1)	bean 定義した `Sequencer` オブジェクトを Inject する。 上記例では、シーケンス値は、フォーマットされた文字列として取得するため、 `Sequencer` のジェネリクス型には、`java.lang.String` 型を指定している。
(2)	Inject する bean の bean 名を`@javax.inject.Named` アノテーションの value 属性に指定する。 上記例では、`xxx-infra.xml` に定義した bean 名 (`articleIdSequencer`) を指定している。
(3)	`Sequencer#getNext()` メソッドを呼び出し、次のシーケンス値を取得する。 上記例では、取得したシーケンス値を、Entity の ID として使用している。 現在のシーケンス値を取得する場合は、`Sequencer#getCurrent()` メソッドを呼び出す。

ちなみに： bean 定義する `Sequencer` が一つの場合は、`@Named` アノテーションが省略できる。複数指定する場合は、`@Named` アノテーションを使用して、bean 名の指定が必要となる。

Spring Framework から提供されているデータアクセス例外へ変換するクラス

Spring Framework のデータアクセス例外へ変換する役割を持つクラスを、以下に示す。

表 6.6　Spring Framework のデータアクセス例外への変換クラス

項番	クラス名	説明
1.	org.springframework.jdbc.support.SQLErrorCodeSQLExceptionTranslator	MyBatis や、JdbcTemplate を使った場合、本クラスによって、JDBC 例外が、Spring Framework のデータアクセス例外に変換される。変換ルールは、XML ファイルに記載されており、デフォルトで使用される XML ファイルは、spring-jdbc.jar 内の org/springframework/jdbc/support/sql-error-codes.xml となる。クラスパス直下に、XML ファイル（sql-error-codes.xml）を配置することで、デフォルトの動作を変更することもできる。
2.	org.springframework.orm.jpa.vendor.HibernateJpaDialect	JPA(Hibernate の実装) を使った場合、本クラスによって、O/R Mapper 例外 (Hibernate の例外) が Spring Framework のデータアクセス例外に変換される。
3.	org.springframework.orm.jpa.EntityManagerFactoryUtils	HibernateJpaDialect で変換できない例外が発生した場合は、本クラスによって、JPA 例外が Spring Framework のデータアクセス例外に変換される。
4.	org.hibernate.dialect.Dialect のサブクラス	JPA(Hibernate の実装) を使った場合、本クラスによって、JDBC 例外と O/R Mapper 例外に変換される。

Spring Framework から提供されている JDBC データソースクラス

Spring Framework では、JDBC データソースの実装を提供しているが、非常にシンプルなクラスなので、商用環境で使われることは少ない。

主に単体試験時に使用されるクラスである。

表 6.7 Spring Framework から提供されている JDBC データソース

項番	クラス名	説明
1.	org.springframework.jdbc.datasource.DriverManagerDataSource	アプリケーションからコネクションの取得依頼があったタイミングで、java.sql.DriverManager#getConnection を呼び出し、新しいコネクションを生成するデータソースクラス。コネクションのプーリングが必要な場合は、アプリケーションサーバのデータソース、または、OSS/Third-Party ライブラリから提供されているデータソースを使用すること。
2.	org.springframework.jdbc.datasource.SingleConnectionDataSource	DriverManagerDataSource の子クラスで、一つのコネクションを使いまわす実装になっており、シングルスレッドで動くユニットテスト向けのデータソースクラスである。ユニットテストでも、マルチスレッドでデータソースにアクセスする場合は、本クラスを使用すると、期待した動作にならないことがあるので、注意が必要である。
3.	org.springframework.jdbc.datasource.SimpleDriverDataSource	アプリケーションからコネクションの取得依頼があったタイミングで、java.sql.Driver#getConnection を呼び出し、新しいコネクションを生成するデータソースクラス。コネクションのプーリングが必要な場合は、アプリケーションサーバのデータソース、または、OSS/Third-Party ライブラリから提供されているデータソースを使用すること。

Spring Framework では、JDBC データソースの動作を拡張したアダプタークラスを提供している。
以下に、代表的なアダプタークラスを紹介する。

表 6.8 Spring Framework から提供されている JDBC データソースのアダプター

項番	クラス名	説明
1.	org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy	トランザクション管理されていないデータソースを、Spring Framework のトランザクション管理対象にするためのアダプタークラス。
2.	org.springframework.jdbc.datasource.lookup.IsolationLevelDataSourceRouter	実行中のトランザクションの独立性レベルによって、使用するデータソースを切り替えるためのアダプタークラス。

6.2 データベースアクセス（MyBatis3 編）

6.2.1 Overview

本節では、MyBatis3 を使用してデータベースにアクセスする方法について説明する。

本ガイドラインでは、MyBatis3 の Mapper インタフェースを Repository インタフェースとして使用することを前提としている。Repository インタフェースについては、「*Repository の実装*」を参照されたい。

Overview では、MyBatis3 と MyBatis-Spring を使用してデータベースアクセスする際のアーキテクチャについて説明を行う。

実際の使用方法については、「*How to use*」を参照されたい。

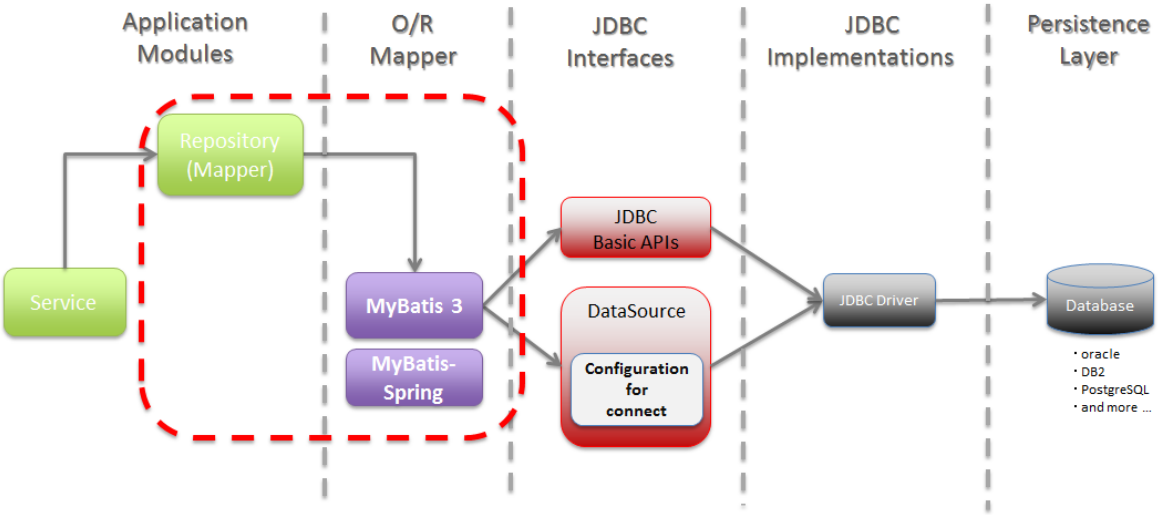


図 6.2　**Picture - Scope of description**

MyBatis3 について

MyBatis3 は、O/R Mapper の一つだが、データベースで管理されているレコードとオブジェクトをマッピングするという考え方ではなく、SQL とオブジェクトをマッピングするという考え方で開発された O/R Mapper である。

そのため、正規化されていないデータベースへアクセスする場合や、発行する SQL を O/R Mapper に任せずに、アプリケーション側で完全に制御したい場合に有効な O/R Mapper である。

本ガイドラインでは、MyBatis3 から追加された Mapper インタフェースを使用して、Entity の CRUD 操作を

行う。Mapper インタフェースの詳細については、「*Mapper インタフェースの仕組みについて*」を参照されたい。

本ガイドラインでは、MyBatis3 の全ての機能の使用方法について説明を行うわけではないため、「MyBatis 3 REFERENCE DOCUMENTATION 」も合わせて参照して頂きたい。

MyBatis3 のコンポーネント構成について

MyBatis3 の主要なコンポーネント (設定ファイル) について説明する。

MyBatis3 では、設定ファイルの定義に基づき、以下のコンポーネントが互いに連携する事によって、SQL の実行及び O/R マッピングを実現している。

項番	コンポーネント/設定ファイル	説明
1.	MyBatis 設定ファイル	MyBatis3 の動作設定を記載する XML ファイル。 データベースの接続先、マッピングファイルのパス、MyBatis の動作設定などを記載するファイルである。Spring と連携して使用する場合は、データベースの接続先やマッピングファイルのパスの設定を本設定ファイルに指定する必要がないため、MyBatis3 のデフォルトの動作を変更又は拡張する際に、設定を行う事になる。
2.	`org.apache.ibatis.session.SqlSessionFactoryBuilder`	MyBatis 設定ファイルを読込み、`SqlSessionFactory` を生成するためのコンポーネント。 Spring と連携して使用する場合は、アプリケーションのクラスから本コンポーネントを直接扱うことはない。
3.	`org.apache.ibatis.session.SqlSessionFactory`	`SqlSession` を生成するためのコンポーネント。 Spring と連携して使用する場合は、アプリケーションのクラスから本コンポーネントを直接扱うことはない。
4.	`org.apache.ibatis.session.SqlSession`	SQL の発行やトランザクション制御の API を提供するコンポーネント。 MyBatis3 を使ってデータベースにアクセスする際に、もっとも重要な役割を果たすコンポーネントである。 Spring と連携して使用する場合は、アプリケーションのクラスから本コンポーネントを直接扱うことは、基本的にはない。
5.	Mapper インタフェース	マッピングファイルに定義した SQL をタイプセーフに呼び出すためのインタフェース。 Mapper インターフェースに対する実装クラスは、MyBatis3 が自動で生成するため、開発者はインターフェースのみ作成すればよい。
6.	マッピングファイル	SQL と O/R マッピングの設定を記載する XML ファイル。

以下に、MyBatis3 の主要コンポーネントが、どのような流れでデータベースにアクセスしているのかを説明する。

データベースにアクセスするための処理は、大きく 2 つにわける事ができる。

- アプリケーションの起動時に行う処理。下記 (1)〜(3) の処理が、これに該当する。
- クライアントからのリクエスト毎に行う処理。下記 (4)〜(10) の処理が、これに該当する。

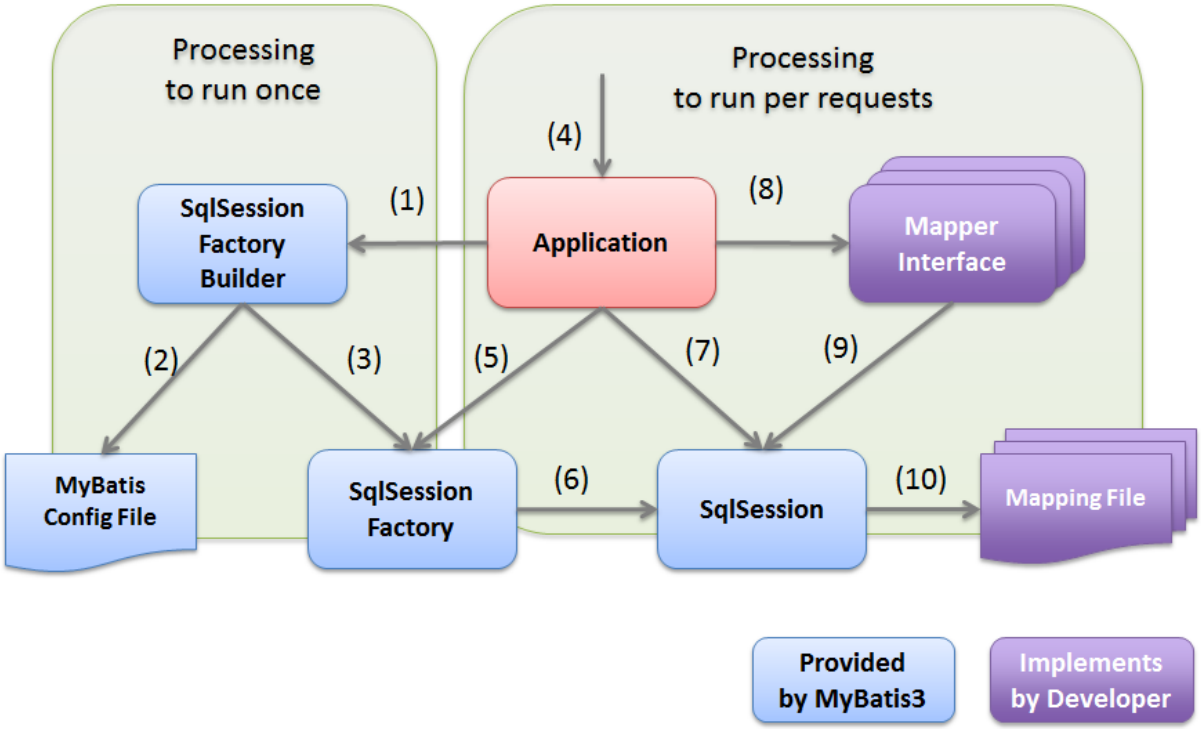


図 6.3 **Picture - Relationship of MyBatis3 components**

アプリケーションの起動時に行う処理は、以下の流れで実行する。

Spring と連携時の流れについては、「*MyBatis-Spring のコンポーネント構成について*」を参照されたい。

項番	説明
1.	アプリケーションは、`SqlSessionFactoryBuilder` に対して `SqlSessionFactory` の構築を依頼する。
2.	`SqlSessionFactoryBuilder` は、`SqlSessionFactory` を生成するために MyBatis 設定ファイルを読込む。
3.	`SqlSessionFactoryBuilder` は、MyBatis 設定ファイルの定義に基づき `SqlSessionFactory` を生成する。

クライアントからのリクエスト毎に行う処理は、以下の流れで実行する。

Spring と連携時の流れについては、「*MyBatis-Spring のコンポーネント構成について*」を参照されたい。

項番	説明
4.	クライアントは、アプリケーションに対して処理を依頼する。
5.	アプリケーションは、 `SqlSessionFactoryBuilder` によって構築された `SqlSessionFactory` から `SqlSession` を取得する。
6.	`SqlSessionFactory` は、`SqlSession` を生成しアプリケーションに返却する。
7.	アプリケーションは、 `SqlSession` から Mapper インタフェースの実装オブジェクトを取得する。
8.	アプリケーションは、Mapper インタフェースのメソッドを呼び出す。 Mapper インタフェースの仕組みについては、「*Mapper インタフェースの仕組みについて*」を参照されたい。
9.	Mapper インタフェースの実装オブジェクトは、 `SqlSession` のメソッドを呼び出して、SQL の実行を依頼する。
10.	`SqlSession` は、マッピングファイルから実行する SQL を取得し、SQL を実行する。

ちなみに：トランザクション制御について

上記フローには記載していないが、トランザクションのコミット及びロールバックは、アプリケーションのコードから `SqlSession` の API を直接呼び出して行う。

ただし、Spring と連携する場合は、Spring のトランザクション管理機能がコミット及びロールバックを行うため、アプリケーションのクラスから `SqlSession` のトランザクションを制御するための API を直接呼び出すことはない。

MyBatis3 と Spring の連携について

MyBatis3 と Spring を連携させるライブラリとして、MyBatis から MyBatis-Spring というライブラリが提供されている。

このライブラリを使用することで、MyBatis3 のコンポーネントを Spring の DI コンテナ上で管理する事ができる。

MyBatis-Spring を使用すると、

- MyBatis3 の SQL の実行を Spring が管理しているトランザクション内で行う事ができるため、MyBatis3 の API に依存したトランザクション制御を行う必要がない。
- MyBatis3 の例外は、Spring が用意している汎用的な例外 (`org.springframework.dao.DataAccessException`) へ変換されるため、MyBatis3 の API に依存しない例外処理を実装する事ができる。
- MyBatis3 を使用するための初期化処理は、すべて MyBatis-Spring の API が行ってくれるため、基本的には MyBatis3 の API を直接使用する必要がない。
- スレッドセーフな Mapper オブジェクトの生成が行えるため、シングルトンの Service クラスに Mapper オブジェクトを注入する事ができる。

等のメリットがある。本ガイドラインでは、MyBatis-Spring を使用することを前提とする。

本ガイドラインでは、MyBatis-Spring の全ての機能の使用方法について説明を行うわけではないため、「Mybatis-Spring REFERENCE DOCUMENTATION」も合わせて参照して頂きたい。

MyBatis-Spring のコンポーネント構成について

MyBatis-Spring の主要なコンポーネントについて説明する。

MyBatis-Spring では、以下のコンポーネントが連携する事によって、MyBatis3 と Spring の連携を実現している。

項番	コンポーネント/設定ファイル	説明
1.	org.mybatis.spring.SqlSessionFactoryBean	SqlSessionFactory を構築し、Spring の DI コンテナ上にオブジェクトを格納するためのコンポーネント。 MyBatis3 標準では、MyBatis 設定ファイルに定義されている情報を基に SqlSessionFactory を構築するが、SqlSessionFactoryBean を使用すると、MyBatis 設定ファイルがなくても SqlSessionFactory を構築することができる。もちろん、併用することも可能である。
2.	org.mybatis.spring.mapper.MapperFactoryBean	シングルトンの Mapper オブジェクトを構築し、Spring の DI コンテナ上にオブジェクトを格納するためのコンポーネント。 MyBatis3 標準の仕組みで生成される Mapper オブジェクトはスレッドセーフではないため、スレッド毎にインスタンスを割り当てる必要があった。MyBatis-Spring のコンポーネントで作成された Mapper オブジェクトは、スレッドセーフな Mapper オブジェクトを生成する事ができるため、Service などのシングルトンのコンポーネントに DI することが可能となる。
3.	org.mybatis.spring.SqlSessionTemplate	SqlSession インターフェースを実装したシングルトン版の SqlSession コンポーネント。 MyBatis3 標準の仕組みで生成される SqlSession オブジェクトはスレッドセーフではないため、スレッド毎にインスタンスを割り当てる必要があった。MyBatis-Spring のコンポーネントで作成された SqlSession オブジェクトは、スレッドセーフな SqlSession オブジェクトが生成されるため、Service などのシングルトンのコンポーネントに DI することが可能になる。 ただし、本ガイドラインでは、SqlSession を直接扱う事は想定していない。

以下に、MyBatis-Spring の主要コンポーネントが、どのような流れでデータベースにアクセスしているのかを説明する。データベースにアクセスするための処理は、大きく 2 つにわける事ができる。

- アプリケーションの起動時に行う処理。下記 (1)～(4) の処理が、これに該当する。
- クライアントからのリクエスト毎に行う処理。下記 (5)～(11) の処理が、これに該当する。

アプリケーションの起動時に行う処理は、以下の流れで実行される。

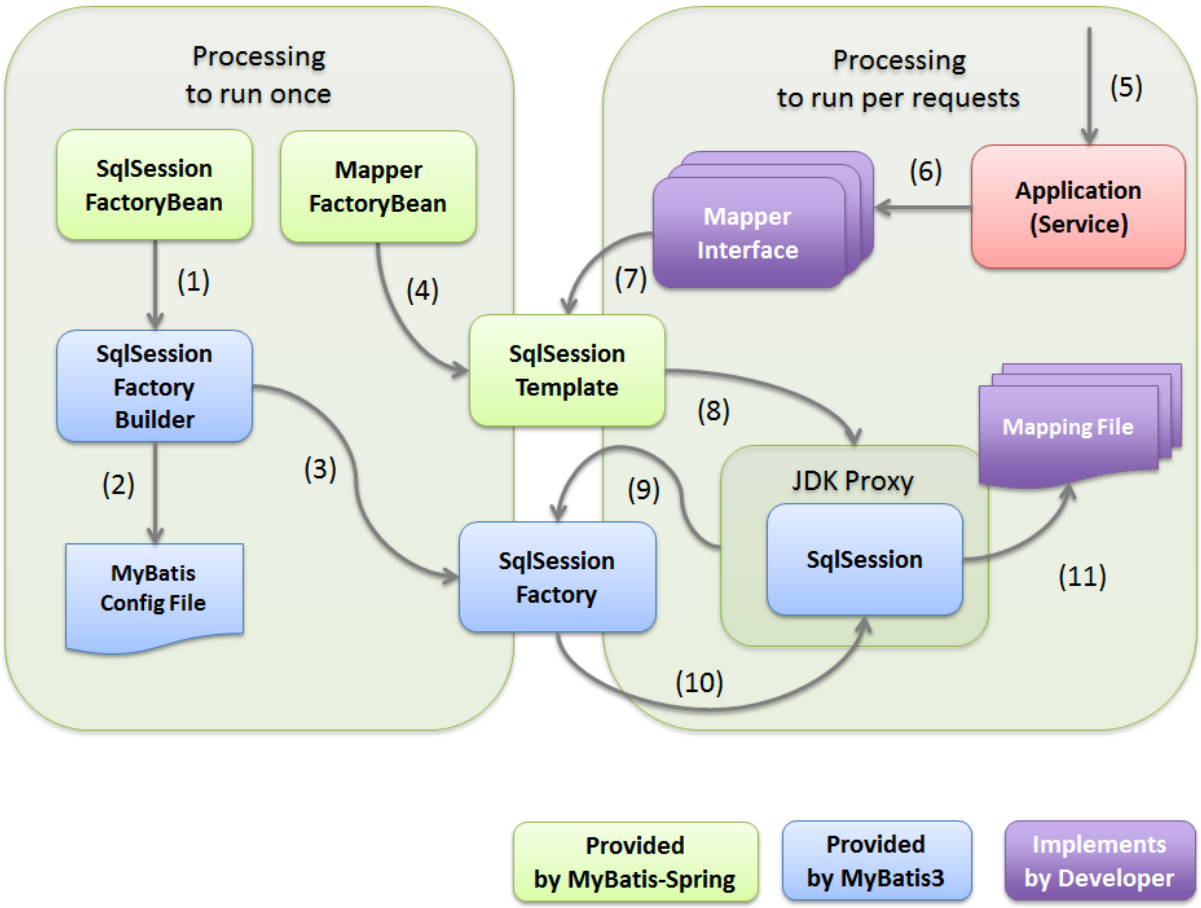


図 6.4 **Picture - Relationship of MyBatis-Spring components**

項番	説明
1.	`SqlSessionFactoryBean` は、`SqlSessionFactoryBuilder` に対して `SqlSessionFactory` の構築を依頼する。
2.	`SqlSessionFactoryBuilder` は、`SqlSessionFactory` を生成するために MyBatis 設定ファイルを読込む。
3.	`SqlSessionFactoryBuilder` は、MyBatis 設定ファイルの定義に基づき `SqlSessionFactory` を生成する。 生成された `SqlSessionFactory` は、Spring の DI コンテナによって管理される。
4.	`MapperFactoryBean` は、スレッドセーフな `SqlSession` (`SqlSessionTemplate`) と、スレッドセーフな Mapper オブジェクト (Mapper インタフェースの Proxy オブジェクト) を生成する。 生成された Mapper オブジェクトは、Spring の DI コンテナによって管理され、Service クラスなどに DI される。Mapper オブジェクトは、スレッドセーフな `SqlSession` (`SqlSessionTemplate`) を利用することで、スレッドセーフな実装を提供している。

クライアントからのリクエスト毎に行う処理は、以下の流れで実行される。

項番	説明
5.	クライアントは、アプリケーションに対して処理を依頼する。
6.	アプリケーション (Service) は、DI コンテナによって注入された Mapper オブジェクト (Mapper インターフェースを実装した Proxy オブジェクト) のメソッドを呼び出す。 Mapper インタフェースの仕組みについては、「*Mapper インタフェースの仕組みについて*」を参照されたい。
7.	Mapper オブジェクトは、呼び出されたメソッドに対応する `SqlSession` (`SqlSessionTemplate`) のメソッドを呼び出す。
8.	`SqlSession` (`SqlSessionTemplate`) は、Proxy 化されたスレッドセーフな `SqlSession` のメソッドを呼び出す。
9.	Proxy 化されたスレッドセーフな `SqlSession` は、トランザクションに割り当てられている MyBatis3 標準の `SqlSession` を使用する。 トランザクションに割り当てられている `SqlSession` が存在しない場合は、MyBatis3 標準の `SqlSession` を取得するために、`SqlSessionFactory` のメソッドを呼び出す。
10.	`SqlSessionFactory` は、MyBatis3 標準の `SqlSession` を返却する。 返却された MyBatis3 標準の `SqlSession` はトランザクションに割り当てられるため、同一トランザクション内であれば、新たに生成されることはなく、同じ `SqlSession` が使用される仕組みになっている。
11.	MyBatis3 標準の `SqlSession` は、マッピングファイルから実行する SQL を取得し、SQL を実行する。

ちなみに：トランザクション制御について

上記フローには記載していないが、トランザクションのコミット及びロールバックは、Spring のトランザクション管理機能が行う。

Spring のトランザクション管理機能を使用したトランザクション管理方法については、「トランザクション管理について」を参照されたい。

6.2.2 How to use

ここからは、実際に MyBatis3 を使用して、データベースにアクセスするための設定及び実装方法について、説明する。

以降の説明は、大きく以下に分類する事ができる。

項番	分類	説明
1.	アプリケーション全体の設定	MyBatis3 をアプリケーションで使用するための設定方法や、MyBatis3 の動作を変更するための設定方法について記載している。 ここに記載している内容は、**プロジェクト立ち上げ時にアプリケーションアーキテクトが設定を行う時に必要となる。**そのため、基本的にはアプリケーション開発者が個々に意識する必要はない部分である。 以下のセクションが、この分類に該当する。 • *pom.xml* の設定 • *MyBatis3* と *Spring* を連携するための設定 • *MyBatis3* の設定 MyBatis3 用のブランクプロジェクト からプロジェクトを生成した場合は、上記で説明している設定の多くが既に設定済みの状態となっているため、アプリケーションアーキテクトは、プロジェクト特性を判断し、必要に応じて設定の追加及び変更を行うことになる。
2.	データアクセス処理の実装方法	MyBatis3 を使った基本的なデータアクセス処理の実装方法について記載している。 ここに記載している内容は、**アプリケーション開発者が実装時に必要となる。** 以下のセクションが、この分類に該当する。 • データベースアクセス処理の実装 • 検索結果と *JavaBean* のマッピング方法 • *Entity* の検索処理 • *Entity* の登録処理 • *Entity* の更新処理 • *Entity* の削除処理 • 動的 *SQL* の実装 • *LIKE* 検索時のエスケープ • *SQL Injection* 対策

pom.xml の設定

インフラストラクチャ層に MyBatis3 を使用する場合は、pom.xml に terasoluna-gfw-mybatis3-dependencies への依存関係を追加する。

マルチプロジェクト構成の場合は、domain プロジェクトの pom.xml(projectName-domain/pom.xml) に追加する。

MyBatis3 用のブランクプロジェクト からプロジェクトを生成した場合は、terasoluna-gfw-mybatis3-dependencies への依存関係は、設定済みの状態である。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <artifactId>projectName-domain</artifactId>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.example</groupId>
        <artifactId>mybatis3-example-app</artifactId>
        <version>1.0.0-SNAPSHOT</version>
        <relativePath>../pom.xml</relativePath>
    </parent>

    <dependencies>

        <!-- omitted -->

        <!-- (1) -->
        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-mybatis3-dependencies</artifactId>
            <type>pom</type>
        </dependency>

        <!-- omitted -->

    </dependencies>

    <!-- omitted -->

</project>
```

項番	説明
1.	terasoluna-gfw-mybatis3 を dependencies に追加する。terasoluna-gfw-mybatis3 には、MyBatis3 及び MyBatis-Spring への依存関係が定義されている。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。

MyBatis3 と Spring を連携するための設定

データソースの設定

MyBatis3 と Spring を連携する場合、データソースは Spring の DI コンテナで管理しているデータソースを使用する必要がある。

MyBatis3 用のブランクプロジェクト からプロジェクトを生成した場合は、Apache Commons DBCP のデータソースが設定済みの状態であるため、プロジェクトの要件に合わせて設定を変更すること。

データソースの設定方法については、共通編の「データソースの設定 」を参照されたい。

トランザクション管理の設定

MyBatis3 と Spring を連携する場合、トランザクション管理は Spring の DI コンテナで管理している `PlatformTransactionManager` を使用する必要がある。

ローカルトランザクションを使用する場合は、JDBC の API を呼び出してトランザクション制御を行う `DataSourceTransactionManager` を使用する。

MyBatis3 用のブランクプロジェクト からプロジェクトを生成した場合は、`DataSourceTransactionManager` が設定済みの状態である。

設定例は以下の通り。

- `projectName-env/src/main/resources/META-INF/spring/projectName-env.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        https://www.springframework.org/schema/jdbc/spring-jdbc.xsd
        http://www.springframework.org/schema/jee
```

```
        https://www.springframework.org/schema/jee/spring-jee.xsd
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- omitted -->

    <!-- (1) -->
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.
↪DataSourceTransactionManager">
        <!-- (2) -->
        <property name="dataSource" ref="dataSource" />
        <!-- (3) -->
        <property name="rollbackOnCommitFailure" value="true" />
    </bean>

    <!-- omitted -->

</beans>
```

項番	説明
1.	PlatformTransactionManager として、org.springframework.jdbc.datasource.DataSourceTransactionManager を指定する。
2.	dataSource プロパティに、設定済みのデータソースの bean を指定する。 トランザクション内で SQL を実行する際は、ここで指定したデータソースからコネクションが取得される。
3.	コミット時にエラーが発生した場合にロールバック処理が呼び出される様にする。 この設定を追加することで、「未確定状態の操作を持つコネクションがコネクションプールに戻ることで発生する意図しないコミット（コネクション再利用時のコミット、コネクションクローズ時の暗黙コミットなど）」が発生するリスクを下げることができる。ただし、ロールバック処理時にエラーが発生する可能性もあるため、意図しないコミットが発生するリスクがなくなるわけではない点に留意されたい。

注釈: PlatformTransactionManager の bean ID について

id 属性には、transactionManager を指定することを推奨する。

transactionManager 以外の値を指定すると、<tx:annotation-driven> タグの transaction-manager 属性に同じ値を設定する必要がある。

アプリケーションサーバから提供されているトランザクションマネージャを使用する場合は、JTA の API を呼び出してトランザクション制御を行う `org.springframework.transaction.jta.JtaTransactionManager` を使用する。

設定例は以下の通り。

- `projectName-env/src/main/resources/META-INF/spring/projectName-env.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        https://www.springframework.org/schema/jdbc/spring-jdbc.xsd
        http://www.springframework.org/schema/jee
        https://www.springframework.org/schema/jee/spring-jee.xsd
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd">

    <!-- omitted -->

    <!-- (1) -->
    <tx:jta-transaction-manager />

    <!-- omitted -->

</beans>
```

項番	説明
1.	`<tx:jta-transaction-manager />` を指定すると、アプリケーションサーバに対して最適な `JtaTransactionManager` が bean 定義される。

MyBatis-Spring の設定

MyBatis3 と Spring を連携する場合、MyBatis-Spring のコンポーネントを使用して、

- MyBatis3 と Spring を連携するために必要となる処理がカスタマイズされた `SqlSessionFactory` の生成
- スレッドセーフな Mapper オブジェクト (Mapper インタフェースの Proxy オブジェクト) の生成

を行う必要がある。

MyBatis3 用のブランクプロジェクト からプロジェクトを生成した場合は、MyBatis3 と Spring を連携するための設定は、設定済みの状態である。

設定例は以下の通り。

- `projectName-domain/src/main/resources/META-INF/spring/projectName-infra.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://mybatis.org/schema/mybatis-spring
        http://mybatis.org/schema/mybatis-spring.xsd">

    <import resource="classpath:/META-INF/spring/projectName-env.xml" />

    <!-- (1) -->
    <bean id="sqlSessionFactory"
        class="org.mybatis.spring.SqlSessionFactoryBean">
        <!-- (2) -->
        <property name="dataSource" ref="dataSource" />
        <!-- (3) -->
        <property name="configLocation"
            value="classpath:/META-INF/mybatis/mybatis-config.xml" />
    </bean>

    <!-- (4) -->
    <mybatis:scan base-package="com.example.domain.repository" />

</beans>
```

項番	説明
1.	`SqlSessionFactory` を生成するためのコンポーネントとして、`SqlSessionFactoryBean` を bean 定義する。
2.	`dataSource` プロパティに、設定済みのデータソースの bean を指定する。 MyBatis3 の処理の中で SQL を発行する際は、ここで指定したデータソースからコネクションが取得される。
3.	`configLocation` プロパティに、MyBatis 設定ファイルのパスを指定する。 ここで指定したファイルが `SqlSessionFactory` を生成する時に読み込まれる。
4.	Mapper インタフェースをスキャンするために`<mybatis:scan>` を定義し、`base-package` 属性には、Mapper インタフェースが格納されている基底パッケージを指定する。 指定されたパッケージ配下に格納されている Mapper インタフェースがスキャンされ、スレッドセーフな Mapper オブジェクト (Mapper インタフェースの Proxy オブジェクト) が自動的に生成される。 **【指定するパッケージは、各プロジェクトで決められたパッケージにすること】**

注釈: MyBatis3 の設定方法について

`SqlSessionFactoryBean` を使用する場合、MyBatis3 の設定は、MyBatis 設定ファイルではなく bean のプロパティに直接指定することもできるが、本ガイドラインでは、MyBatis3 自体の設定は MyBatis 標準の設定ファイルに指定する方法を推奨する。

MyBatis3 の設定

MyBatis3 では、MyBatis3 の動作をカスタマイズするための仕組みが用意されている。

MyBatis3 の動作をカスタマイズする場合は、MyBatis 設定ファイルに設定値を追加する事で実現可能である。

ここでは、アプリケーションの特性に依存しない設定項目についてのみ、説明を行う。

その他の設定項目に関しては、「MyBatis 3 REFERENCE DOCUMENTATION(Configuration XML) 」を参照し、アプリケーションの特性にあった設定を行うこと。

基本的にはデフォルト値のままでも問題ないが、アプリケーションの特性を考慮し、必要に応じて設定を変更すること。

注釈: MyBatis 設定ファイルの格納場所について

本ガイドラインでは、MyBatis 設定ファイルは、projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml に格納することを推奨している。

MyBatis3 用のブランクプロジェクト からプロジェクトを生成した場合は、上記ファイルは格納済みの状態である。

fetchSize の設定

大量のデータを返すようなクエリを記述する場合は、JDBC ドライバに対して適切な fetchSize を指定する必要がある。
fetchSize は、JDBC ドライバとデータベース間の 1 回の通信で取得するデータの件数を設定するパラメータである。

fetchSize を指定しないと JDBC ドライバのデフォルト値が利用されるため、使用する JDBC ドライバによっては以下の問題を引き起こす可能性がある。

- デフォルト値が小さい JDBC ドライバの場合は「性能の劣化」
- デフォルト値が大きい又は制限がない JDBC ドライバの場合は「メモリの枯渇」

これらの問題が発生しないように制御するために、MyBatis3 は以下の 2 つの方法で fetchSize を指定することができる。

- 全てのクエリに対して適用する「デフォルトの fetchSize」の指定
- 特定のクエリに対して適用する「クエリ単位の fetchSize」の指定

注釈: 「デフォルトの fetchSize」について

「デフォルトの fetchSize」は、MyBatis 3.3.0 以降のバージョンで利用することができる。

以下に、「デフォルトの fetchSize」を指定する方法を示す。

- projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org/DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <settings>
```

```
        <!-- (1) -->
        <setting name="defaultFetchSize" value="100" />
    </settings>

</configuration>
```

項番	説明
1.	defaultFetchSize に、1 回の通信で取得するデータの件数を指定する。

注釈: 「クエリ単位の **fetchSize**」の指定方法

fetchSize をクエリ単位に指定する必要がある場合は、検索用の SQL を記述するための XML 要素 (<select>要素) の fetchSize 属性に値を指定すればよい。

なお、大量のデータを返すようなクエリを記述する場合は、「*ResultHandler の実装*」の利用も検討すること。

SQL 実行モードの設定

MyBatis3 では、SQL を実行するモードとして以下の 3 種類を用意している。

どのモードを使用するかは、各モードの特性と制約、及び性能要件を考慮して決定して頂きたい。
実行モードの設定方法などについては、「*SQL 実行モードの利用*」を参照されたい。

項番	モード	説明
1.	SIMPLE	SQL 実行毎に新しい `java.sql.PreparedStatement` を作成する。 MyBatis のデフォルトの動作であり、MyBatis3 用のブランクプロジェクト も `SIMPLE` モードとなっている。
2.	REUSE	`PreparedStatement` をキャッシュし再利用する。 同一トランザクション内で同じ SQL を複数回実行する場合は、`REUSE` モードで実行すると、`SIMPLE` モードと比較して性能向上が期待できる。 これは、SQL を解析して `PreparedStatement` を生成する処理の実行回数を減らす事ができるためである。
3.	BATCH	更新系の SQL をバッチ実行する。(`java.sql.Statement#executeBatch()` を使って SQL を実行する)。 同一トランザクション内で更新系の SQL を連続して大量に実行する場合は、`BATCH` モードで実行すると、`SIMPLE` モードや `REUSE` モードと比較して性能向上が期待できる。 これは、 • SQL を解析して `PreparedStatement` を生成する処理の実行回数 • サーバと通信する回数 を減らす事ができるためである。 ただし、`BATCH` モードを使用する場合は、MyBatis の動きが `SIMPLE` モードや `SIMPLE` モードと異なる部分がある。具体的な違いと注意点については、「*バッチモードの Repository 利用時の注意点*」を参照されたい。

TypeAlias の設定

TypeAlias を使用すると、マッピングファイルで指定する Java クラスに対して、エイリアス名 (短縮名) を割り当てる事ができる。

TypeAlias を使用しない場合、マッピングファイルで指定する `type` 属性、`parameterType` 属性、`resultType` 属性などには、Java クラスの完全修飾クラス名 (FQCN) を指定する必要があるため、マッピングファイルの記述効率の低下、記述ミスの増加などが懸念される。

本ガイドラインでは、記述効率の向上、記述ミスの削減、マッピングファイルの可読性向上などを目的として、TypeAlias を使用することを推奨する。

MyBatis3 用のブランクプロジェクト からプロジェクトを生成した場合は、Entity を格納するパッケージ (`${projectPackage}.domain.model`) 配下に格納されるクラスが TypeAlias の対象となっている。必

要に応じて、設定を追加されたい。

TypeAlias の設定方法は以下の通り。

- projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <typeAliases>
        <!-- (1) -->
        <package name="com.example.domain.model" />
    </typeAliases>
</configuration>
```

項番	説明
1.	package 要素の name 属性に、エイリアスを設定するクラスが格納されているパッケージ名を指定する。 指定したパッケージ配下に格納されているクラスは、パッケージの部分が除去された部分が、エイリアス名となる。上記例だと、com.example.domain.model.Account クラスのエイリアス名は、Account となる。 【指定するパッケージは、各プロジェクトで決められたパッケージにすること】

ちなみに: クラス単位に **Type Alias** を設定する方法について

Type Alias の設定には、クラス単位に設定する方法やエイリアス名を明示的に指定する方法が用意されている。詳細は、Appendix の「*TypeAlias の設定*」を参照されたい。

TypeAlias を使用した際の、マッピングファイルの記述例は以下の通り。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.example.domain.repository.account.AccountRepository">

    <resultMap id="accountResultMap"
        type="Account">
        <!-- omitted -->
```

```
    </resultMap>

    <select id="findOne"
        parameterType="string"
        resultMap="accountResultMap">
        <!-- omitted -->
    </select>

    <select id="findByCriteria"
        parameterType="AccountSearchCriteria"
        resultMap="accountResultMap">
        <!-- omitted -->
    </select>

</mapper>
```

ちなみに：MyBatis3 標準のエイリアス名について

プリミティブ型やプリミティブラッパ型などの一般的な Java クラスについては、予めエイリアス名が設定されている。

予め設定されるエイリアス名については、「MyBatis 3 REFERENCE DOCUMENTATION(Configuration XML-typeAliases-) 」を参照されたい。

NULL 値と JDBC 型のマッピング設定

使用しているデータベース (JDBC ドライバ) によっては、カラム値を null に設定する際に、エラーが発生する場合がある。
この事象は、JDBC ドライバが `null` 値の設定と認識できる JDBC 型を指定する事で、解決する事ができる。

`null` 値を設定した際に、以下の様なスタックトレースを伴うエラーが発生した場合は、`null` 値と JDBC 型のマッピングが必要となる。
MyBatis3 のデフォルトでは、`OTHER` と呼ばれる汎用的な JDBC 型が指定されるが、`OTHER` だとエラーとなる JDBC ドライバもある。

```
java.sql.SQLException: Invalid column type: 1111
    at oracle.jdbc.driver.OracleStatement.getInternalType(OracleStatement.
↪java:3916) ~[ojdbc6-11.2.0.2.0.jar:11.2.0.2.0]
    at oracle.jdbc.driver.OraclePreparedStatement.
↪setNullCritical(OraclePreparedStatement.java:4541) ~[ojdbc6-11.2.0.2.0.
↪jar:11.2.0.2.0]
```

```
    at oracle.jdbc.driver.OraclePreparedStatement.
↪setNull(OraclePreparedStatement.java:4523) ~[ojdbc6-11.2.0.2.0.jar:11.2.0.2.
↪0]
    ...
```

注釈: Oracle 使用時の動作について

Oracle JDBC ドライバは JDBC 型の OTHER をサポートしていないため、デフォルト設定のままだとエラーが発生することが確認されている。

Oracle では JDBC 型の NULL 型を指定すれば、null 値を正常にマッピングすることが可能となる。

以下に、MyBatis3 のデフォルトの動作を変更する方法を示す。

- projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org/DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <settings>
        <!-- (1) -->
        <setting name="jdbcTypeForNull" value="NULL" />
    </settings>

</configuration>
```

項番	説明
1.	jdbcTypeForNull に、JDBC 型を指定する。 上記例では、null 値の JDBC 型として NULL 型を指定している。

ちなみに: 項目単位で解決する方法について

別の解決方法として、null 値が設定される可能性があるプロパティのインラインパラメータに、Java 型に対応する適切な JDBC 型を個別に指定する方法もある。

ただし、インラインパラメータで個別に指定した場合、マッピングファイルの記述量及び指定ミスが発生する可能性が増えることが予想されるため、本ガイドラインとしては、全体の設定でエラーを解決す

ることを推奨している。全体の設定を変更してもエラーが解決しない場合は、エラーが発生するプロパティについてのみ、個別に設定を行えばよい。

TypeHandler の設定

TypeHandler は、Java クラスと JDBC 型をマッピングする時に使用される。

具体的には、

- SQL を発行する際に、Java クラスのオブジェクトを java.sql.PreparedStatement のバインドパラメータとして設定する
- SQL の発行結果として取得した java.sql.ResultSet から値を取得する

際に、使用される。

プリミティブ型やプリミティブラッパ型などの一般的な Java クラスについては、MyBatis3 から TypeHandler が提供されており、特別な設定を行う必要はない。

注釈: BLOB 用と CLOB 用の実装について

MyBatis 3.4 で追加された TypeHandler は、JDBC 4.0 (Java 1.6) で追加された API を使用することで、BLOB と java.io.InputStream 、CLOB と java.io.Reader の変換を実現している。JDBC 4.0 サポートの JDBC ドライバーであれば、BLOB ⇔ InputStream 、CLOB ⇔ Reader 変換用のタイプハンドラーがデフォルトで有効になるため、TypeHandler を新たに実装する必要はない。

JDBC 4.0 との互換性のない JDBC ドライバを使う場合は、利用する JDBC ドライバの互換バージョンを意識した TypeHandler を作成する必要がある。

例えば、PostgreSQL 用の JDBC ドライバ (postgresql-42.2.5.jar) では、JDBC 4.0 から追加されたメソッドの一部が、未実装の状態である。

注釈: mybatis-typehandlers-jsr310 で提供されていた JSR-310 Date and Time API 用の TypeHandler が、MyBatis 3.4.5 からコアモジュールに統合された。これにより、依存ライブラリとして別途 mybatis-typehandlers-jsr310 を追加する必要はなくなった。

ちなみに: MyBatis3 から提供されている TypeHandler については、「MyBatis 3 REFERENCE DOCUMENTATION(Configuration XML-typeHandlers-) 」を参照されたい。

ちなみに：Enum 型のマッピングについて

MyBatis3 のデフォルトの動作では、Enum 型は Enum の定数名 (文字列) とマッピングされる。

下記のような Enum 型の場合は、WAITING_FOR_ACTIVE , ACTIVE , EXPIRED , LOCKED という文字列とマッピングされてテーブルに格納される。

```
package com.example.domain.model;

public enum AccountStatus {
    WAITING_FOR_ACTIVE, ACTIVE, EXPIRED, LOCKED
}
```

MyBatis では、Enum 型を数値 (定数の定義順) とマッピングする事もできる。数値とマッピングする方法については、「MyBatis 3 REFERENCE DOCUMENTATION(Configuration XML-Handling Enums-) 」を参照されたい。

TypeHandler の作成が必要になるケースは、MyBatis3 でサポートしていない Joda-Time のクラスと JDBC 型をマッピングする場合である。

具体的には、本ガイドラインで利用を推奨している「日付操作 *(Joda Time)*」の org.joda.time.DateTime 型と、JDBC 型の TIMESTAMP 型をマッピングする場合に、TypeHandler の作成が必要となる。

Joda-Time のクラスと JDBC 型をマッピングする TypeHandler の作成例については、「*TypeHandler の実装*」を参照されたい。

ここでは、作成した TypeHandler を MyBatis に適用する方法について説明を行う。

- projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org/DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <typeHandlers>
        <!-- (1) -->
        <package name="com.example.infra.mybatis.typehandler" />
```

```
    </typeHandlers>

</configuration>
```

項番	説明
1.	MyBatis 設定ファイルに TypeHandler の設定を行う。 package 要素の name 属性に、作成した TypeHandler が格納されているパッケージ名を指定する。指定したパッケージ配下に格納されている TypeHandler が、MyBatis によって自動検出される。

ちなみに: 上記例では、指定したパッケージ配下に格納されている TypeHandler を MyBatis によって自動検出させているが、クラス単位に設定する事もできる。

クラス単位に TypeHandler を設定する場合は、typeHandler 要素を使用する。

- projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml

```
<typeHandlers>
    <typeHandler handler="xxx.yyy.zzz.CustomTypeHandler" />
    <package name="com.example.infra.mybatis.typehandler" />
</typeHandlers>
```

更に、TypeHandler の中で DI コンテナが管理している bean を使用したい場合は、bean 定義ファイル内で TypeHandler を指定すればよい。

- projectName-domain/src/main/resources/META-INF/spring/projectName-infra.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:tx="http://www.springframework.org/schema/tx" xmlns:mybatis=
↪"http://mybatis.org/schema/mybatis-spring"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx
    https://www.springframework.org/schema/tx/spring-tx.xsd
    http://mybatis.org/schema/mybatis-spring
    http://mybatis.org/schema/mybatis-spring.xsd">

    <bean id="sqlSessionFactory" class="org.mybatis.spring.
↪SqlSessionFactoryBean">
```

```
        <property name="dataSource" ref="oracleDataSource" />
        <property name="configLocation"
            value="classpath:/META-INF/mybatis/mybatis-config.xml" />
        <property name="typeHandlers">
            <list>
                <bean class="xxx.yyy.zzz.CustomTypeHandler" />
            </list>
        </property>
    </bean>

</beans>
```

TypeHandler を適用する Java クラスと JDBC 型のマッピングの指定は、

- MyBatis 設定ファイル内の typeHandler 要素の属性値として指定
- @org.apache.ibatis.type.MappedTypes アノテーションと@org.apache.ibatis.type.MappedJdbcTypes アノテーションに指定
- MyBatis3 から提供されている TypeHandler の基底クラス (org.apache.ibatis.type.BaseTypeHandler) を継承することで指定

する方法がある。

詳しくは、「MyBatis 3 REFERENCE DOCUMENTATION(Configuration XML-typeHandlers-) 」を参照されたい。

ちなみに: 上記の設定例は、いずれもアプリケーション全体に適用するための設定方法であったが、フィールド毎に個別の TypeHandler を指定する事も可能である。これは、アプリケーション全体に適用している TypeHandler を上書きする際に使用する。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://
↪mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.example.domain.repository.image.ImageRepository">
    <resultMap id="resultMapImage" type="Image">
        <id property="id" column="id" />
        <!-- (2) -->
        <result property="imageData" column="image_data" typeHandler=
↪"XxxBlobInputStreamTypeHandler" />
        <result property="createdAt" column="created_at"  />
    </resultMap>
    <select id="findOne" parameterType="string" resultMap="resultMapImage">
        SELECT
```

```
            id
            ,image_data
            ,created_at
        FROM
            t_image
        WHERE
            id = #{id}
    </select>
    <insert id="create" parameterType="Image">
        INSERT INTO
            t_image
        (
            id
            ,image_data
            ,created_at
        )
        VALUES
        (
            #{id}
            /* (3) */
            ,#{imageData,typeHandler=XxxBlobInputStreamTypeHandler}
            ,#{createdAt}
        )
    </insert>
</mapper>
```

項番	説明
2.	検索結果 (ResultSet) から値を取得する際は、id 又は result 要素の typeHandler 属性に適用する TypeHandler を指定する。
3.	PreparedStatement に値を設定する際は、インラインパラメータの typeHandler 属性に適用する TypeHandler を指定する。

TypeHandler をフィールド毎に個別に指定する場合は、TypeHandler のクラスに TypeAlias を設けることを推奨する。TypeAlias の設定方法については、「*TypeAlias の設定*」を参照されたい。

データベースアクセス処理の実装

MyBatis3 の機能を使用してデータベースにアクセスするための、具体的な実装方法について説明する。

Repository インタフェースの作成

Entity 毎に Repository インタフェースを作成する。

```
package com.example.domain.repository.todo;

// (1)
public interface TodoRepository {
}
```

項番	説明
1.	Java のインタフェースとして Repository インタフェースを作成する。 上記例では、Todo という Entity に対する Repository インタフェースを作成している。

マッピングファイルの作成

Repository インタフェース毎にマッピングファイルを作成する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!-- (1)  -->
<mapper namespace="com.example.domain.repository.todo.TodoRepository">
</mapper>
```

項番	説明
1.	mapper 要素の namespace 属性に、Repository インタフェースの完全修飾クラス名 (FQCN) を指定する。

注釈: マッピングファイルの格納先について

マッピングファイルの格納先は、

- MyBatis3 が自動的にマッピングファイルを読み込むために定めたルールに則ったディレクトリ
- 任意のディレクトリ

のどちらかを選択することができる。

本ガイドラインでは、MyBatis3 が定めたルールに則ったディレクトリに格納し、マッピングファイルを自動的に読み込む仕組みを利用することを推奨する。

マッピングファイルを自動的に読み込ませるためには、Repository インタフェースのパッケージ階層と同じ階層で、マッピングファイルをクラスパス上に格納する必要がある。

具体的には、com.example.domain.repository.todo.TodoRepository という Repository インターフェースに対するマッピングファイル (TodoRepository.xml) は、projectName-domain/src/main/resources/com/example/domain/repository/todo ディレクトリに格納すればよい。

CRUD 処理の実装

ここからは、基本的な CRUD 処理の実装方法と、SQL 実装時の考慮点について説明を行う。

基本的な CRUD 処理として、以下の処理の実装方法について説明を行う。

- 検索結果と *JavaBean* のマッピング方法
- *Entity* の検索処理
- *Entity* の登録処理
- *Entity* の更新処理
- *Entity* の削除処理
- 動的 *SQL* の実装

注釈: MyBatis3 を使用して CRUD 処理を実装する際は、検索した Entity がローカルキャッシュと呼ばれる領域にキャッシュされる仕組みになっている点を意識しておく必要がある。

MyBatis3 が提供するローカルキャッシュのデフォルトの動作は以下の通りである。

- ローカルキャッシュは、トランザクション単位で管理する。
- Entity のキャッシュは、「ステートメント ID + 組み立てられた SQL のパターン + 組み立てられた SQL にバインドするパラメータ値 + ページ位置 (取得範囲)」毎に行う。

つまり、同一トランザクション内の処理において、MyBatis が提供している検索 API を全て同じパラメータで呼び出すと、2 回目以降は SQL を発行せずに、キャッシュされている Entity のインスタンスが返却される。

ここでは、 **MyBatis の API が返却する Entity とローカルキャッシュで管理している Entity が同じインスタンス** という点を意識しておいてほしい。

ちなみに: ローカルキャッシュは、ステートメント単位で管理するように変更する事もできる。ローカルキャッシュをステートメント単位で管理する場合、MyBatis は毎回 SQL を実行して最新の Entity を

取得する。

SQL 実装時の考慮点として、以下の点について説明を行う。

- *LIKE 検索時のエスケープ*
- *SQL Injection 対策*

具体的な実装方法の説明を行う前に、以降の説明で登場するコンポーネントについて、簡単に説明しておく。

項番	コンポーネント	説明
1.	Entity	アプリケーションで扱う業務データを保持する JavaBean クラス。 Entity の詳細については、「*Entity の実装*」を参照されたい。
2.	Repository インタフェース	Entity の CRUD 操作を行うためのメソッドを定義するインタフェース。 Repository の詳細については、「*Repository の実装*」を参照されたい。
3.	Service クラス	業務ロジックを実行するためのクラス。 Service の詳細については、「*Service の実装*」を参照されたい。

注釈: 本ガイドラインでは、アーキテクチャ上の用語を統一するために、MyBatis3 の Mapper インタフェースの事を Repository インタフェースと呼んでいる。

以降の説明では、「*Entity の実装*」「*Repository の実装*」「*Service の実装*」を読んでいる前提で説明を行う。

検索結果と JavaBean のマッピング方法

Entity の検索処理の説明を行う前に、検索結果と JavaBean のマッピング方法について説明を行う。

MyBatis3 では、検索結果 (ResultSet) を JavaBean(Entity) にマッピングする方法として、自動マッピングと手動マッピングの 2 つの方法が用意されている。それぞれ特徴があるので、**プロジェクトの特性やアプリケーションで実行する SQL の特性などを考慮して、使用するマッピング方法を決めて頂きたい。**

注釈: 使用するマッピング方法について

本ガイドラインでは、

- シンプルなマッピング (単一オブジェクトへのマッピング) の場合は自動マッピングを使用し、高度なマッピング (関連オブジェクトへのマッピング) が必要な場合は手動マッピングを使用する。
- 一律手動マッピングを使用する

の、2 つの案を提示する。これは、上記 2 案のどちらかを選択する事を強制するものではなく、あくまで選択肢のひとつと考えて頂きたい。

アーキテクトは、自動マッピングと手動マッピングを使うケースの判断基準をプログラマに対して明確に示すことで、アプリケーション全体として統一されたマッピング方法が使用されるように心がけてほしい。

以下に、自動マッピングと手動マッピングに対して、それぞれの特徴と使用例を説明する。

検索結果の自動マッピング

MyBatis3 では、検索結果 (ResultSet) のカラムと JavaBean のプロパティをマッピングする方法として、カラム名とプロパティ名を一致させることで、自動的に解決する仕組みを提供している。

注釈: 自動マッピングの特徴について

自動マッピングを使用すると、マッピングファイルには実行する SQL のみ記述すればよいため、マッピングファイルの記述量を減らすことができる点が特徴である。

記述量が減ることで、単純ミスの削減や、カラム名やプロパティ名変更時の修正箇所の削減といった効果も期待できる。

ただし、自動マッピングが行えるのは、単一オブジェクトに対するマッピングのみである。ネストした関連オブジェクトに対してマッピングを行いたい場合は、手動マッピングを使用する必要がある。

ちなみに: カラム名について

ここで言うカラム名とは、テーブルの物理的なカラム名ではなく、SQL を発行して取得した検索結

果 (ResultSet) がもつカラム名の事である。そのため、AS 句を使うことで、物理的なカラム名と JavaBean のプロパティ名を一致させることは、比較的容易に行うことができる。

以下に、自動マッピングを使用して検索結果を JavaBean にマッピングする実装例を示す。

- projectName-domain/src/main/resources/com/example/domain/repository/todo/TodoRepository.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <select id="findOne" parameterType="string" resultType="Todo">
        SELECT
            todo_id AS "todoId", /* (1) */
            todo_title AS "todoTitle",
            finished, /* (2) */
            created_at AS "createdAt",
            version
        FROM
            t_todo
        WHERE
            todo_id = #{todoId}
    </select>

</mapper>
```

項番	説明
1.	テーブルの物理カラム名と JavaBean のプロパティ名が異なる場合は、AS 句を使用して一致させることで、自動マッピング対象にすることができる。
2.	テーブルの物理カラム名と JavaBean のプロパティ名が一致している場合は、AS 句を指定する必要はない。

- JavaBean

```
package com.example.domain.model;

import java.io.Serializable;
import java.util.Date;

public class Todo implements Serializable {

```

```
    private static final long serialVersionUID = 1L;

    private String todoId;

    private String todoTitle;

    private boolean finished;

    private Date createdAt;

    private long version;

    public String getTodoId() {
        return todoId;
    }

    public void setTodoId(String todoId) {
        this.todoId = todoId;
    }

    public String getTodoTitle() {
        return todoTitle;
    }

    public void setTodoTitle(String todoTitle) {
        this.todoTitle = todoTitle;
    }

    public boolean isFinished() {
        return finished;
    }

    public void setFinished(boolean finished) {
        this.finished = finished;
    }

    public Date getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Date createdAt) {
        this.createdAt = createdAt;
    }

    public long getVersion() {
        return version;
    }

    public void setVersion(long version) {
        this.version = version;
    }

}
```

ちなみに：アンダースコア区切りのカラム名とキャメルケース形式のプロパティ名のマッピング方法について

上記例では、アンダースコア区切りのカラム名とキャメルケース形式のプロパティ名の違いについてAS 句を使って吸収しているが、アンダースコア区切りのカラム名とキャメルケース形式のプロパティ名の違いを吸収するだけならば、MyBatis3 の設定を変更する事で実現可能である。

テーブルの物理カラム名をアンダースコア区切りにしている場合は、MyBatis 設定ファイル(mybatis-config.xml) に以下の設定を追加することで、キャメルケースの JavaBean のプロパティに自動マッピングする事ができる。

- projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <settings>
        <!-- (3) -->
        <setting name="mapUnderscoreToCamelCase" value="true" />
    </settings>

</configuration>
```

項番	説明
3.	mapUnderscoreToCamelCase を *true* にする設定を追加する。 設定を *true* にすると、アンダースコア区切りのカラム名がキャメルケース形式に自動変換される。具体例としては、カラム名が todo_id の場合、todoId に変換されてマッピングが行われる。

- projectName-domain/src/main/resources/com/example/domain/repository/todo/TodoRepository.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">
```

```
    <select id="findOne" parameterType="string" resultType="Todo">
        SELECT
            todo_id, /* (4) */
            todo_title,
            finished,
            created_at,
            version
        FROM
            t_todo
        WHERE
            todo_id = #{todoId}
    </select>

</mapper>
```

項番	説明
4.	アンダースコア区切りのカラム名とキャメルケース形式のプロパティ名の違いを吸収するために、AS 句の指定が不要になるため、よりシンプルな SQL となる。

検索結果の手動マッピング

MyBatis3 では、検索結果 (`ResultSet`) のカラムと JavaBean のプロパティの対応付けを、マッピングファイルに定義する事で、手動で解決する仕組みを用意している。

注釈: 手動マッピングの特徴について

手動マッピングを使用すると、検索結果 (`ResultSet`) のカラムと JavaBean のプロパティの対応付けを、マッピングファイルに 1 項目ずつ定義することになる。そのため、マッピングの柔軟性が非常に高く、より複雑なマッピングを実現する事ができる点が特徴である。

手動マッピングは、

- アプリケーションが扱うデータモデル (JavaBean) と物理テーブルのレイアウトが一致しない
- JavaBean がネスト構造になっている (別の JavaBean をネストしている)

といったケースにおいて、検索結果 (`ResultSet`) のカラムと JavaBean のプロパティをマッピングする際に力を発揮するマッピング方法である。

また、自動マッピングに比べて効率的にマッピングを行う事ができる。処理の効率性を優先するアプリケーションの場合は、自動マッピングの代わりに手動マッピングを使用した方がよい。

以下に、手動マッピングを使用して検索結果を JavaBean にマッピングする実装例を示す。
ここでは、手動マッピングの使用方法を示す事が目的なので、自動マッピングでもマッピング可能なもっともシンプルなパターンを例に、説明を行う。

実践的なマッピングの実装例については、

- 「MyBatis 3 REFERENCE DOCUMENTATION(Mapper XML Files-Advanced Result Maps-) 」
- 「関連 *Entity* を 1 回の *SQL* で取得する方法について」
- 「関連 *Entity* をネストした *SQL* を使用して取得する方法について」

を参照されたい。

- `projectName-domain/src/main/resources/com/example/domain/repository/todo/TodoRepository.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <!-- (1) -->
    <resultMap id="todoResultMap" type="Todo">
        <!-- (2) -->
        <id column="todo_id" property="todoId" />
        <!-- (3) -->
        <result column="todo_title" property="todoTitle" />
        <result column="finished" property="finished" />
        <result column="created_at" property="createdAt" />
        <result column="version" property="version" />
    </resultMap>

    <!-- (4) -->
    <select id="findOne" parameterType="string" resultMap="todoResultMap">
        SELECT
            todo_id,
            todo_title,
            finished,
            created_at,
            version
        FROM
            t_todo
        WHERE
            todo_id = #{todoId}
    </select>

</mapper>
```

項番	説明
1.	<resultMap>要素に、検索結果 (ResultSet) と JavaBean のマッピング定義を行う。 id 属性にマッピングを識別するための ID を、type 属性にマッピングする JavaBean のクラス名 (又はエイリアス名) を指定する。 <resultMap>要素の詳細は、「MyBatis 3 REFERENCE DOCUMENTATION(Mapper XML Files-resultMap-)」を参照されたい。
2.	検索結果 (ResultSet) の ID(PK) のカラムと JavaBean のプロパティのマッピングを行う。 ID(PK) のマッピングは、<id>要素を使って指定する。column 属性には検索結果 (ResultSet) のカラム名、property 属性には JavaBean のプロパティ名を指定する。 <id>要素の詳細は、「MyBatis 3 REFERENCE DOCUMENTATION(Mapper XML Files-id & result-)」を参照されたい。
3.	検索結果 (ResultSet) の ID(PK) 以外のカラムと JavaBean のプロパティのマッピングを行う。 ID(PK) 以外のマッピングは、<result>要素を使って指定する。column 属性には検索結果 (ResultSet) のカラム名、property 属性には JavaBean のプロパティ名を指定する。 <result>要素の詳細は、「MyBatis 3 REFERENCE DOCUMENTATION(Mapper XML Files-id & result-)」を参照されたい。
4.	<select>要素の resultMap 属性に、適用するマッピング定義の ID を指定する。

注釈: id 要素と result 要素の使い分けについて

<id>要素と<result>要素は、どちらも検索結果 (ResultSet) のカラムと JavaBean のプロパティをマッピングするための要素であるが、ID(PK) カラムに対してマッピングは、<id>要素を使うことを推奨する。

理由は、ID(PK) カラムに対して<id>要素を使用してマッピングを行うと、MyBatis3 が提供しているオブジェクトのキャッシュ制御の処理や、関連オブジェクトへのマッピングの処理のパフォーマンスを、全体的に向上させることが出来るためである。

Entity の検索処理

Entity の検索処理の実装方法について、目的別に説明を行う。

Entity の検索処理の実装方法に対する説明を読む前に、「検索結果と *JavaBean* のマッピング方法」を一読して頂きたい。

以降の説明では、アンダースコア区切りのカラム名をキャメルケース形式のプロパティ名に自動でマッピングする設定を有効にした場合の実装例となる。

- `projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml`

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <settings>
        <setting name="mapUnderscoreToCamelCase" value="true" />
    </settings>

</configuration>
```

単一キーの Entity の取得

PK が単一カラムで構成されるテーブルより、PK を指定して Entity を 1 件取得する際の実装例を以下に示す。

- Repository インタフェースにメソッドを定義する。

```
package com.example.domain.repository.todo;

import com.example.domain.model.Todo;

public interface TodoRepository {

    // (1)
    Todo findOne(String todoId);

}
```

項番	説明
1.	上記例では、引数に指定された `todoId`(PK) に一致する Todo オブジェクトを 1 件取得するためのメソッドとして、`findOne` メソッドを定義している。

- マッピングファイルに SQL を定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <!-- (2) -->
    <select id="findOne" parameterType="string" resultType="Todo">
        /* (3) */
        SELECT
            todo_id,
            todo_title,
            finished,
            created_at,
            version
        FROM
            t_todo
        /* (4) */
        WHERE
            todo_id = #{todoId}
    </select>

</mapper>
```

項番	属性	説明
2.	-	`select` 要素の中に、検索結果が 0〜1 件となる SQL を実装する。 上記例では、ID(PK) が一致するレコードを取得する SQL を実装している。 `select` 要素の詳細については、「MyBatis3 REFERENCE DOCUMENTATION (Mapper XML Files-select-)」を参照されたい。
	id	Repository インタフェースに定義したメソッドのメソッド名を指定する。
	parameterType	パラメータ完全修飾クラス名 (又はエイリアス名) を指定する。
	resultType	検索結果 (`ResultSet`) をマッピングする JavaBean の完全修飾クラス名 (又はエイリアス名) を指定する。 手動マッピングを使用する場合は、`resultType` 属性の代わりに `resultMap` 属性を使用して、適用するマッピング定義を指定する。 手動マッピングについては、「検索結果の手動マッピング」を参照されたい。
3.	-	取得対象のカラムを指定する。 上記例では、検索結果 (`ResultSet`) を JavaBean へマッピングする方法として、自動マッピングを使用している。自動マッピングについては、「検索結果の自動マッピング」を参照されたい。
4.	-	WHERE 句に検索条件を指定する。 検索条件にバインドする値は、`#{variableName}`形式のバインド変数として指定する。上記例では、`#{todoId}`がバインド変数となる。 Repository インタフェースの引数の型が `String` のような単純型の場合は、バインド変数名は任意の名前でよいが、引数の型が JavaBean の場合は、バインド変数名には JavaBean のプロパティ名を指定する必要がある。

注釈: 単純型のバインド変数名について

`String` のような単純型の場合は、バインド変数名に制約はないが、メソッドの引数名と同じ値にしておくことを推奨する。

- Service クラスに Repository を DI し、Repository インターフェースのメソッドを呼び出す。

```
package com.example.domain.service.todo;

import javax.inject.Inject;

import org.springframework.stereotype.Service;
```

```
import org.springframework.transaction.annotation.Transactional;

import com.example.domain.model.Todo;
import com.example.domain.repository.todo.TodoRepository;

@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    // (5)
    @Inject
    TodoRepository todoRepository;

    @Transactional(readOnly = true)
    @Override
    public Todo getTodo(String todoId) {
        // (6)
        Todo todo = todoRepository.findOne(todoId);
        if (todo == null) { // (7)
            throw new ResourceNotFoundException(ResultMessages.error().add(
                    "e.ex.td.5001", todoId));
        }
        return todo;
    }

}
```

項番	説明
5.	Service クラスに Repository インターフェースを DI する。
6.	Repository インターフェースのメソッドを呼び出し、Entity を 1 件取得する。
7.	検索結果が 0 件の場合は `null` が返却されるため、必要に応じて Entity が取得できなかった時の処理を実装する。 上記例では、Entity が取得できなかった場合は、リソース未検出エラーを発生させている。

複合キーの Entity の取得

PK が複数カラムで構成されるテーブルより、PK を指定して Entity を 1 件取得する際の実装例を以下に示す。基本的な構成は、PK が単一カラムで構成される場合と同じであるが、Repository インタフェースのメソッド引数の指定方法が異なる。

- Repository インタフェースにメソッドを定義する。

```
package com.example.domain.repository.order;

import org.apache.ibatis.annotations.Param;

import com.example.domain.model.OrderHistory;

public interface OrderHistoryRepository {

    // (1)
    OrderHistory findOne(@Param("orderId") String orderId,
            @Param("historyId") int historyId);

}
```

項番	説明
1.	PK を構成するカラムに対応する引数を、メソッドに定義する。 上記例では、受注の変更履歴を管理するテーブルの PK として、orderId と historyId を引数に定義している。

ちなみに：メソッド引数を複数指定する場合のバインド変数名について

Repository インタフェースのメソッド引数を複数指定する場合は、引数に@org.apache.ibatis.annotations.Param アノテーションを指定することを推奨する。@Param アノテーションの value 属性には、マッピングファイルから値を参照する際に指定する「バインド変数名」を指定する。

上記例だと、マッピングファイルから#{orderId}及び#{historyId}と指定することで、引数に指定された値を SQL にバインドする事ができる。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.order.
↪OrderHistoryRepository">

    <select id="findOne" resultType="OrderHistory">
        SELECT
            order_id,
            history_id,
            order_name,
            operation_type,
            created_at"
        FROM
            t_order_history
        WHERE
            order_id = #{orderId}
        AND
            history_id = #{historyId}
```

```
    </select>

</mapper>
```

@Param アノテーションの指定は必須ではないが、指定しないと以下に示すような機械的なバインド変数名を指定する必要がある。@Param アノテーションの指定しない場合のバインド変数名は、「」param」 + 引数の宣言位置 (1 から開始)」という名前になるため、ソースコードのメンテナンス性及び可読性を損なう要因となる。

```
<!-- omitted -->

WHERE
    order_id = #{param1}
AND
    history_id = #{param2}

<!-- omitted -->
```

MyBatis 3.4.1 以降では、JDK 8 から追加されたコンパイルオプション (-parameters) を使用することで、@Param アノテーションを省略する事ができる。

Entity の検索

検索結果が 0〜N 件となる SQL を発行し、Entity を複数件取得する際の実装例を以下に示す。

警告: 検索結果が大量のデータになる可能性がある場合は、「*ResultHandler の実装*」の利用を検討すること。

- Entity を複数件取得するためのメソッドを定義する。

```
package com.example.domain.repository.todo;

import java.util.List;

import com.example.domain.model.Todo;

public interface TodoRepository {
```

```
    // (1)
    List<Todo> findAllByCriteria(TodoCriteria criteria);

}
```

項番	説明
1.	上記例では、検索条件を保持する JavaBean(TodoCriteria) に一致する Todo オブジェクトをリスト形式で複数件取得するためのメソッドとして、findAllByCriteria メソッドを定義している。

ちなみに: 上記例では、メソッドの返り値に java.util.List を指定しているが、検索結果を java.util.Map として受け取る事も出来る。

Map で受け取る場合は、

- Map の key には PK の値
- Map の value には Entity オブジェクト

を格納する事になる。

検索結果を Map で受け取る場合、java.util.HashMap のインスタンスが返却されるため、Map の並び順は保証されないという点に注意すること。

以下に、実装例を示す。

```
package com.example.domain.repository.todo;

import java.util.Map;

import com.example.domain.model.Todo;
import org.apache.ibatis.annotations.MapKey;

public interface TodoRepository {

    @MapKey("todoId")
    Map<String, Todo> findAllByCriteria(TodoCriteria criteria);

}
```

検索結果を Map で受け取る場合は、@org.apache.ibatis.annotations.MapKey アノテーションをメソッドに指定する。アノテーションの value 属性には、Map の key として扱うプロパティ名を指定する。上記例では、Todo オブジェクトの PK(todoId) を指定している。

- 検索条件を保持する JavaBean を作成する。

```
package com.example.domain.repository.todo;

import java.io.Serializable;
import java.util.Date;

public class TodoCriteria implements Serializable {

    private static final long serialVersionUID = 1L;

    private String title;

    private Date createdAt;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Date getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Date createdAt) {
        this.createdAt = createdAt;
    }

}
```

注釈: 検索条件を保持するための **JavaBean** の作成について

検索条件を保持するための JavaBean の作成は必須ではないが、格納されている値の役割が明確になるため、JavaBean を作成することを推奨する。ただし、JavaBean を作成しない方法で実装してもよい。

アーキテクトは、**JavaBean** を作成するケースと作成しないケースの判断基準をプログラマに対して明確に示すことで、アプリケーション全体として統一された作りになるようにすること。

JavaBean を作成しない場合の実装例を以下に示す。

```
package com.example.domain.repository.todo;

import java.util.List;

import com.example.domain.model.Todo;

public interface TodoRepository {

    List<Todo> findAllByCriteria(@Param("title") String title,
```

```
            @Param("createdAt") Date createdAt);

}
```

JavaBean を作成しない場合は、検索条件を 1 項目ずつ引数に宣言し、`@Param` アノテーションの `value` 属性に「バインド変数名」を指定する。上記のようなメソッドを定義することで、複数の検索条件を SQL に引き渡すことができる。

- マッピングファイルに SQL を定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <!-- (2) -->
    <select id="findAllByCriteria" parameterType="TodoCriteria" resultType=
↪"Todo">
        <![CDATA[
        SELECT
            todo_id,
            todo_title,
            finished,
            created_at,
            version
        FROM
            t_todo
        WHERE
            todo_title LIKE #{title} || '%' ESCAPE '~'
        AND
            created_at < #{createdAt}
        /* (3) */
        ORDER BY
            todo_id
        ]]>
    </select>

</mapper>
```

項番	説明
2.	select 要素の中に、検索結果が 0〜N 件となる SQL を実装する。 上記例では、todo_title と created_at が指定した条件に一致する Todo レコードを取得する実装している。
3.	ソート条件を指定する。 複数件のレコードを取得する場合は、ソート条件を指定する。特に画面に表示するレコードを取得する SQL では、ソート条件の指定は必須である。

ちなみに：CDATA セクションの活用方法について

SQL 内に XML のエスケープが必要な文字 (「<」や」>」など) を指定する場合は、CDATA セクションを使用すると、SQL の可読性を保つことができる。CDATA セクションを使用しない場合は、<や>といったエンティティ参照文字を指定する必要があり、SQL の可読性を損なう要因となる。

上記例では、created_at に対する条件として」<」を使用しているため、CDATA セクションを指定している。

Entity の件数の取得

検索条件に一致する Entity の件数を取得する際の実装例を以下に示す。

- 検索条件に一致する Entity の件数を取得するためのメソッドを定義する。

```
package com.example.domain.repository.todo;

public interface TodoRepository {

    // (1)
    long countByFinished(boolean finished);

}
```

項番	説明
1.	件数を取得ためのメソッドの返り値は、数値型 (int や long など) を指定する。 上記例では、long を指定している。

- マッピングファイルに SQL を定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <!-- (2) -->
    <select id="countByFinished" parameterType="_boolean" resultType="_long">
        SELECT
            COUNT(*)
        FROM
            t_todo
        WHERE
            finished = #{finished}
    </select>

</mapper>
```

項番	説明
2.	件数を取得する SQL を実行する。 resultType 属性には、返り値の型を指定する。 上記例では、プリミティブ型の long を指定するためのエイリアス名を指定している。

ちなみに：プリミティブ型のエイリアス名について

プリミティブ型のエイリアス名は、先頭に「_」(アンダースコア) を指定する必要がある。「_」(アンダースコア) を指定しない場合は、プリミティブのラッパ型 (java.lang.Long など) として扱われる。

Entity のページネーション検索 (MyBatis3 標準方式)

MyBatis3 の取得範囲指定機能を使用して Entity を検索する際の実装例を以下に示す。

MyBatis では取得範囲を指定するクラスとして org.apache.ibatis.session.RowBounds クラスが用意されており、SQL に取得範囲の条件を記述する必要がない。

警告：検索条件に一致するデータ件数が多くなる場合の注意点について

MyBatis3 標準の方式は、検索結果 (ResultSet) のカーソルを移動することで、取得範囲外のデータをスキップする方式である。そのため、検索条件に一致するデータ件数に比例して、メモリ

枯渇やカーソル移動処理の性能劣化が発生する可能性が高くなる。

カーソルの移動処理は、JDBC の結果セット型に応じて以下の 2 種類がサポートされており、デフォルトの動作は、JDBC ドライバのデフォルトの結果セット型に依存する。

- 結果セット型が FORWARD_ONLY の場合は、ResultSet#next() を繰り返し呼び出して取得範囲外のデータをスキップする。
- 結果セット型が SCROLL_SENSITIVE 又は SCROLL_INSENSITIVE の場合は、ResultSet#absolute(int) を呼び出して取得範囲外のデータをスキップする。

ResultSet#absolute(int) を使用することで、性能劣化を最小限に抑える事ができる可能性はあるが、JDBC ドライバの実装次第であり、内部で ResultSet#next() と同等の処理が行われている場合は、メモリ枯渇や性能劣化が発生する可能性を抑える事はできない。

検索条件に一致するデータ件数が多くなる可能性がある場合は、MyBatis3 標準方式のページネーション検索ではなく、SQL 絞り込み方式の採用を検討した方がよい。

- Entity のページネーション検索を行うためのメソッドを定義する。

```
ackage com.example.domain.repository.todo;

import java.util.List;

import org.apache.ibatis.session.RowBounds;

import com.example.domain.model.Todo;

public interface TodoRepository {

    // (1)
    long countByCriteria(TodoCriteria criteria);

    // (2)
    List<Todo> findPageByCriteria(TodoCriteria criteria,
        RowBounds rowBounds);

}
```

項番	説明
1.	検索条件に一致する Entity の総件数を取得するメソッドを定義する。
2.	検索条件に一致する Entity の中から、取得範囲の Entity を抽出するメソッドを定義する。 定義したメソッドの引数として、取得範囲の情報 (offset と limit) を保持する RowBounds を指定する。

- マッピングファイルに SQL を定義する。

  検索結果から該当範囲のレコードを抽出する処理は、MyBatis3 が行うため、SQL で取得範囲のレコードを絞り込む必要がない。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <select id="countByCriteria" parameterType="TodoCriteria" resultType="_
↪long">
        <![CDATA[
        SELECT
            COUNT(*)
        FROM
            t_todo
        WHERE
            todo_title LIKE #{title} || '%' ESCAPE '~'
        AND
            created_at < #{createdAt}
        ]]>
    </select>

    <select id="findPageByCriteria" parameterType="TodoCriteria" resultType=
↪"Todo">
        <![CDATA[
        SELECT
            todo_id,
            todo_title,
            finished,
            created_at,
            version
        FROM
            t_todo
        WHERE
            todo_title LIKE #{title} || '%' ESCAPE '~'
```

```
        AND
            created_at < #{createdAt}
        ORDER BY
            todo_id
        ]]>
    </select>

</mapper>
```

注釈: **WHERE** 句の共通化について

ページネーション検索を実現する場合、「検索条件に一致する Entity の総件数を取得する SQL」と「検索条件に一致する Entity のリストを取得する SQL」で指定する WHERE 句は、MyBatis3 の include 機能を使って共通化することを推奨する。

上記 SQL の WHERE 句を共通化した場合、以下のような定義となる。詳細は、「*SQL 文の共有*」を参照されたい。

```
<sql id="findPageByCriteriaWherePhrase">
    <![CDATA[
    WHERE
        todo_title LIKE #{title} || '%' ESCAPE '~'
    AND
        created_at < #{createdAt}
    ]]>
</sql>

<select id="countByCriteria" parameterType="TodoCriteria" resultType="_long
↪">
    SELECT
        COUNT(*)
    FROM
        t_todo
    <include refid="findPageByCriteriaWherePhrase"/>
</select>

<select id="findPageByCriteria" parameterType="TodoCriteria" resultType=
↪"Todo">
    SELECT
        todo_id,
        todo_title,
        finished,
        created_at,
        version
    FROM
        t_todo
    <include refid="findPageByCriteriaWherePhrase"/>
    ORDER BY
        todo_id
</select>
```

注釈: 結果セット型を明示的に指定する方法について

結果セット型を明示的に指定する場合は、resultType 属性に結果セット型を指定する。JDBC ドライバのデフォルトの結果セット型が、FORWARD_ONLY の場合は、SCROLL_INSENSITIVE を指定することを推奨する。

```
<select id="findPageByCriteria" parameterType="TodoCriteria" resultType=
↪"Todo"
    resultSetType="SCROLL_INSENSITIVE">
    <!-- omitted -->
</select>
```

- Service クラスにページネーション検索処理を実装する。

```
// omitted

@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    @Inject
    TodoRepository todoRepository;

    // omitted

    @Transactional(readOnly = true)
    @Override
    public Page<Todo> searchTodos(TodoCriteria criteria, Pageable pageable) {
        // (3)
        long total = todoRepository.countByCriteria(criteria);
        List<Todo> todos;
        if (0 < total) {
            // (4)
            RowBounds rowBounds = new RowBounds(pageable.getOffset(),
                pageable.getPageSize());
            // (5)
            todos = todoRepository.findPageByCriteria(criteria, rowBounds);
        } else {
            // (6)
            todos = Collections.emptyList();
        }
        // (7)
        return new PageImpl<>(todos, pageable, total);
```

```
    }

    // omitted

}
```

項番	説明
3.	まず、検索条件に一致する Entity の総件数を取得する。
4.	検索条件に一致する Entity が存在する場合は、ページネーション検索の取得範囲を指定する `RowBounds` オブジェクトを生成する。 `RowBounds` の第 1 引数 (`offset`) には「スキップ件数」、第 2 引数 (`limit`) には「最大取得件数」を指定する。引数に指定する値、Spring Data Commons から提供されている `Pageable` オブジェクトの `getOffset` メソッドと `getPageSize` メソッドを呼び出して取得した値を指定すればよい。 具体的には、 • offset に」`0`」、limit に `20` を指定した場合、1～20 件目 • offset に `20`、limit に `20` を指定した場合、21～40 件目 が取得範囲となる。
5.	Repository のメソッドを呼び出し、検索条件に一致した取得範囲の Entity を取得する。
6.	検索条件に一致する Entity が存在しない場合は、空のリストを検索結果に設定する。
7.	ページ情報 (`org.springframework.data.domain.PageImpl`) を作成し返却する。

Entity のページネーション検索 (SQL 絞り込み方式)

データベースから提供されている範囲検索の仕組みを使用して Entity を検索する際の実装例を以下に示す。

SQL 絞り込み方式は、データベースから提供されている範囲検索の仕組みを使用するため、MyBatis3 標準方式に比べて効率的に取得範囲の Entity を取得することができる。

> **注釈:** 検索条件に一致するデータ件数が大量にある場合は、**SQL** 絞り込み方式を採用する事を推奨する。

- Entity のページネーション検索を行うためのメソッドを定義する。

```
package com.example.domain.repository.todo;

import java.util.List;

import org.apache.ibatis.annotations.Param;
import org.springframework.data.domain.Pageable;

import com.example.domain.model.Todo;

public interface TodoRepository {

    // (1)
    long countByCriteria(
            @Param("criteria") TodoCriteria criteria);

    // (2)
    List<Todo> findPageByCriteria(
            @Param("criteria") TodoCriteria criteria,
            @Param("pageable") Pageable pageable);
}
```

項番	説明
1.	検索条件に一致する Entity の総件数を取得するメソッドを定義する。
2.	検索条件に一致する Entity の中から、取得範囲の Entity を抽出するメソッドを定義する。 定義したメソッドの引数として、取得範囲の情報 (offset と limit) を保持する `org.springframework.data.domain.Pageable` を指定する。

注釈: 引数が 1 つのメソッドに@Param アノテーションを指定する理由について

上記例では、引数が 1 つのメソッド (`countByCriteria`) に対して`@Param` アノテーションを指定している。これは、`findPageByCriteria` メソッド呼び出し時に実行される SQL と WHERE 句を共通化するためである。

`@Param` アノテーションを使用して引数にバインド変数名を指定することで、SQL 内で指定するバインド変数名のネスト構造を合わせている。

具体的な SQL の実装例については、次に示す。

- マッピングファイルに SQL を定義する。

  SQL で取得範囲のレコードを絞り込む。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <sql id="findPageByCriteriaWherePhrase">
        <![CDATA[
        /* (3) */
        WHERE
            todo_title LIKE #{criteria.title} || '%' ESCAPE '~'
        AND
            created_at < #{criteria.createdAt}
        ]]>
    </sql>

    <select id="countByCriteria" resultType="_long">
        SELECT
            COUNT(*)
        FROM
            t_todo
        <include refid="findPageByCriteriaWherePhrase" />
    </select>

    <select id="findPageByCriteria" resultType="Todo">
        SELECT
            todo_id,
            todo_title,
            finished,
            created_at,
            version
        FROM
            t_todo
        <include refid="findPageByCriteriaWherePhrase" />
        ORDER BY
            todo_id
        LIMIT
            #{pageable.pageSize} /* (4) */
        OFFSET
            #{pageable.offset}  /* (4) */
    </select>

</mapper>
```

項番	説明
3.	countByCriteria と findPageByCriteria メソッドの引数に@Param("criteria") を指定しているため、SQL 内で指定するバインド変数名は criteria.フィールド名の形式となる。
4.	データベースから提供されている範囲検索の仕組みを使用して、必要なレコードのみ抽出する。 Pageable オブジェクトの offset には「スキップ件数」、pageSize には「最大取得件数」が格納されている。 上記例は、データベースとして H2 Database を使用した際の実装例である。

- Service クラスにページネーション検索処理を実装する。

```
// omitted

@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    @Inject
    TodoRepository todoRepository;

    // omitted

    @Transactional(readOnly = true)
    @Override
    public Page<Todo> searchTodos(TodoCriteria criteria,
            Pageable pageable) {
        long total = todoRepository.countByCriteria(criteria);
        List<Todo> todos;
        if (0 < total) {
            // (5)
            todos = todoRepository.findPageByCriteria(criteria,
                    pageable);
        } else {
            todos = Collections.emptyList();
        }
        return new PageImpl<>(todos, pageable, total);
    }

    // omitted

}
```

項番	説明
5.	Repository のメソッドを呼び出し、検索条件に一致した取得範囲の Entity を取得する。 Repository のメソッドを呼び出す際は、引数で受け取った `Pageable` オブジェクトをそのまま渡せばよい。

Entity のページネーション検索 (検索結果のソート)

`Pageable` オブジェクトの `sort` プロパティを利用して、SQL で検索結果をソートする実装例を以下に示す。

Repository および Service については、前述の *Entity のページネーション検索 (SQL 絞り込み方式)* と同様とし、実装例を省略する。

- マッピングファイルに SQL を定義する。

  SQL で検索結果に対してソートをかける。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <select id="findPageByCriteria" resultType="Todo">
        SELECT
            todo_id,
            todo_title,
            finished,
            created_at,
            version
        FROM
            t_todo
        WHERE
            todo_title LIKE #{criteria.title} || '%' ESCAPE '~'
        AND
        <![CDATA[
            created_at < #{criteria.createdAt}
        ]]>
        <choose>
            <!-- (1)  -->
            <when test="pageable.sort != null">
                ORDER BY
```

```
                <!-- (2)  -->
                <foreach item="order" collection="pageable.sort" separator=",
↪">
                    ${order.property}
                    ${order.direction}
                </foreach>
            </when>
            <!-- (3)  -->
            <otherwise>
                ORDER BY todo_id
            </otherwise>
        </choose>
        LIMIT
            #{pageable.pageSize}
        OFFSET
            #{pageable.offset}
    </select>

</mapper>
```

項番	説明
1.	Pageable オブジェクトの sort プロパティが null でない場合、ソート条件を指定する。
2.	sort プロパティに格納されているソート条件をマッピングファイルに引き渡す。order.property はソートする列、order.direction は ASC,DESC などのソート順を表す。 具体的には sort=todo_id,DESC&sort=created_at が指定された場合、ORDER BY todo_id DESC, created_at ASC が生成される。
3.	ソート条件がセットされていない場合はプライマリキー todo_id でソートを行う。

警告: ページネーションの SQL Injection 対策ついて

ソート条件は ${order.property} 、 ${order.direction} のように置換変数による埋め込みを行うため、SQL Injection が発生しないように注意する必要がある。

いずれもリクエストパラメータ sort で指定した値が格納されるが、不正な値が送信された場合の動作に以下の違いがあり、 ${order.property} で SQL Injection が発生する可能性がある。

- property には、送信されたソートする列名の値がそのまま格納される。
- direction には ASC または DESC のどちらかが格納される。それ以外の値が送信された場合は SortHandlerMethodArgumentResolver 内で例外となる。

SQL Injection 対策については、*SQL Injection 対策* を参照されたい。

Entity の登録処理

Entity の登録方法について、目的別に実装例を説明する。

Entity の 1 件登録

Entity を 1 件登録する際の実装例を以下に示す。

- Repository インタフェースにメソッドを定義する。

```
package com.example.domain.repository.todo;

import com.example.domain.model.Todo;

public interface TodoRepository {

    // (1)
    void create(Todo todo);

}
```

項番	説明
1.	上記例では、引数に指定された Todo オブジェクトを 1 件登録するためのメソッドとして、`create` メソッドを定義している。

注釈: Entity を登録するメソッドの返り値について

Entity を登録するメソッドの返り値は、基本的には `void` でよい。

ただし、SELECT した結果を INSERT するような SQL を発行する場合は、アプリケーション要件に応じて `boolean` や数値型 (`int` 又は `long`) を返り値とすること。

- 返り値として `boolean` を指定した場合は、登録件数が 0 件の際は `false`、登録件数が 1 件以上の際は `true` が返却される。
- 返り値として数値型を指定した場合は、登録件数が返却される。

- マッピングファイルに SQL を定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
```

```
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <!-- (2) -->
    <insert id="create" parameterType="Todo">
        INSERT INTO
            t_todo
        (
            todo_id,
            todo_title,
            finished,
            created_at,
            version
        )
        /* (3) */
        VALUES
        (
            #{todoId},
            #{todoTitle},
            #{finished},
            #{createdAt},
            #{version}
        )
    </insert>

</mapper>
```

項番	説明
2.	insert 要素の中に、INSERT する SQL を実装する。 id 属性には、Repository インタフェースに定義したメソッドのメソッド名を指定する。 insert 要素の詳細については、「MyBatis3 REFERENCE DOCUMENTATION (Mapper XML Files-insert, update and delete-)」を参照されたい。
3.	VALUE 句にレコード登録時の設定値を指定する。 VALUE 句にバインドする値は、#{variableName}形式のバインド変数として指定する。上記例では、Repository インタフェースの引数として JavaBean(Todo) を指定しているため、バインド変数名には JavaBean のプロパティ名を指定する。

- Service クラスに Repository を DI し、Repository インターフェースのメソッドを呼び出す。

```
package com.example.domain.service.todo;

import java.util.UUID;
```

```
import javax.inject.Inject;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory;

import com.example.domain.model.Todo;
import com.example.domain.repository.todo.TodoRepository;

@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    // (4)
    @Inject
    TodoRepository todoRepository;

    @Inject
    JodaTimeDateFactory dateFactory;

    @Override
    public Todo create(Todo todo) {
        // (5)
        todo.setTodoId(UUID.randomUUID().toString());
        todo.setCreatedAt(dateFactory.newDate());
        todo.setFinished(false);
        todo.setVersion(1);
        // (6)
        todoRepository.create(todo);
        // (7)
        return todo;
    }

}
```

項番	説明
4.	Service クラスに Repository インターフェースを DI する。
5.	引数で渡された Entity オブジェクトに対して、アプリケーション要件に応じて値を設定する。 上記例では、 • ID として「UUID」 • 登録日時として「システム日時」 • 完了フラグに「`false`: 未完了」 • バージョンに「」1」」 を設定している。
6.	Repository インターフェースのメソッドを呼び出し、Entity を 1 件登録する。
7.	登録した Entity を返却する。 Service クラスの処理で登録値を設定する場合は、登録した Entity オブジェクトを返り値として返却する事を推奨する。

キーの生成

「*Entity の 1 件登録*」では、Service クラスでキー (ID) の生成をする実装例になっているが、MyBatis3 では、マッピングファイル内でキーを生成する仕組みが用意されている。

> **注釈: MyBatis3 のキー生成機能の使用ケースについて**
>
> キーを生成するために、データベースの機能 (関数や ID 列など) を使用する場合は、MyBatis3 のキー生成機能の仕組みを使用する事を推奨する。

キーの生成方法は、2 種類用意されている。

- データベースから用意されている関数などを呼び出した結果をキーとして扱う方法
- データベースから用意されている ID 列 (IDENTITY 型、AUTO_INCREMENT 型など) + JDBC3.0 から追加された `Statement#getGeneratedKeys()` を呼び出した結果をキーとして扱う方法

まず、データベースから用意されている関数などを呼び出した結果をキーとして扱う方法について説明する。
下記例は、データベースとして H2 Database を使用している。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <insert id="create" parameterType="Todo">
        <!-- (1) -->
        <selectKey keyProperty="todoId" resultType="string" order="BEFORE">
            /* (2) */
            SELECT RANDOM_UUID()
        </selectKey>
        INSERT INTO
            t_todo
        (
            todo_id,
            todo_title,
            finished,
            created_at,
            version
        )
        VALUES
        (
            #{todoId},
            #{todoTitle},
            #{finished},
            #{createdAt},
            #{version}
        )
    </insert>

</mapper>
```

項番	属性	説明
1.	-	selectKey 要素の中に、キーを生成するための SQL を実装する。 上記例では、データベースから提供されている関数を使用して UUID を取得している。 selectKey の詳細については、「MyBatis3 REFERENCE DOCUMENTATION (Mapper XML Files-insert, update and delete-)」を参照されたい。
	keyProperty	取得したキー値を格納する Entity のプロパティ名を指定する。 上記例では、Entity の todoId プロパティに生成したキーが設定される。
	resultType	SQL を発行して取得するキー値の型を指定する。
	order	キー生成用 SQL を実行するタイミング (BEFORE 又は AFTER) を指定する。 • BEFORE を指定した場合、selectKey 要素で指定した SQL を実行した結果を Entity に反映した後に INSERT 文が実行される。 • AFTER を指定した場合、INSERT 文を実行した後に selectKey 要素で指定した SQL を実行され、取得した値が Entity に反映される。
2.	-	キーを生成するための SQL を実装する。 上記例では、H2 Database の UUID を生成する関数を呼び出して、キーを生成している。キー生成の代表的な実装としては、シーケンスオブジェクトから取得した値を文字列にフォーマットする実装があげられる。

次に、データベースから用意されている ID 列 + JDBC3.0 から追加された Statement#getGeneratedKeys() を呼び出した結果をキーとして扱う方法について説明する。下記例は、データベースとして H2 Database を使用している。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.audit.AuditLogRepository">

    <!-- (3) -->
    <insert id="create" parameterType="Todo" useGeneratedKeys="true"␣
↪keyProperty="logId">
        INSERT INTO
            t_audit_log
        (
            level,
```

```
            message,
            created_at,
        )
        VALUES
        (
            #{level},
            #{message},
            #{createdAt},
        )
    </insert>

</mapper>
```

項番	属性	説明
3.	useGeneratedKeys	true を指定すると、ID 列 +Statement#getGeneratedKeys() を呼び出してキーを取得する機能が利用可能となる。 useGeneratedKeys の詳細については、「MyBatis3 REFERENCE DOCUMENTATION (Mapper XML Files-insert, update and delete-)」を参照されたい。
	keyProperty	データベース上で自動的にインクリメントされたキー値を格納する Entity のプロパティ名を指定する。 上記例では、INSERT 文実行後に、Entity の logId プロパティに Statement#getGeneratedKeys() で取得したキー値が設定される。

Entity の一括登録

Entity を一括で登録する際の実装例を以下に示す。

Entity を一括で登録する場合は、

- 複数のレコードを同時に登録する INSERT 文を発行する
- JDBC のバッチ更新機能を使用する

方法がある。

JDBC のバッチ更新機能を使用する方法については、「バッチモードの利用」を参照されたい。

ここでは、複数のレコードを同時に登録する INSERT 文を発行する方法について説明する。下記例は、データベースとして H2 Database を使用している。

- Repository インタフェースにメソッドを定義する。

```
package com.example.domain.repository.todo;

import java.util.List;

import com.example.domain.model.Todo;

public interface TodoRepository {

    // (1)
    void createAll(List<Todo> todos);

}
```

項番	説明
1.	上記例では、引数に指定された Todo オブジェクトのリストを一括登録するためのメソッドとして、createAll メソッドを定義している。

- マッピングファイルに SQL を定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <insert id="createAll" parameterType="list">
        INSERT INTO
            t_todo
        (
            todo_id,
            todo_title,
            finished,
            created_at,
            version
        )
        /* (2) */
        VALUES
        /* (3) */
        <foreach collection="list" item="todo" separator=",">
        (
            #{todo.todoId},
            #{todo.todoTitle},
            #{todo.finished},
            #{todo.createdAt},
            #{todo.version}
        )
```

```
        </foreach>
    </insert>

</mapper>
```

項番	属性	説明
2.	-	VALUE 句にレコード登録時の設定値を指定する。
3.	-	`foreach` 要素を使用して、引数で渡された Todo オブジェクトのリストに対して繰り返し処理を行う。 `foreach` の詳細については、「MyBatis3 REFERENCE DOCUMENTATION (Dynamic SQL-foreach-)」を参照されたい。
	collection	処理対象のコレクションを指定する。 上記例では、Repository のメソッド引数のリストに対して繰り返し処理を行っている。Repository メソッドの引数に`@Param` を指定していない場合は、`list` を指定する。`@Param` を指定した場合は、`@Param` の `value` 属性に指定した値を指定する。
	item	リストの中の 1 要素を保持するローカル変数名を指定する。 `foreach` 要素内の SQL からは、#{ローカル変数名. プロパティ名}の形式で JavaBean のプロパティにアクセスする事ができる。
	separator	リスト内の要素間を区切るための文字列を指定する。 上記例では、」,」を指定することで、要素毎の VALUE 句を」,」で区切っている。

注釈: 複数のレコードを同時に登録する SQL を使用する際の注意点

複数のレコードを同時に登録する SQL を実行する場合は、前述の「キーの生成」を使用することが出来ない。

- 以下のような SQL が生成され、実行される。

```
INSERT INTO
    t_todo
(
    todo_id,
    todo_title,
    finished,
    created_at,
    version
```

```
)
VALUES
(
    '99243507-1b02-45b6-bfb6-d9b89f044e2d',
    'todo title 1',
    false,
    '09/17/2014 23:59:59.999',
    1
)
,
(
    '66b096f1-791f-412f-9a0a-ee4a3a9186c2',
    'todo title 2',
    0,
    '09/17/2014 23:59:59.999',
    1
)
```

ちなみに: 一括登録するための SQL は、データベースやバージョンによりサポート状況や文法が異なる。以下に主要なデータベースのリファレンスページへのリンクを記載しておく。

- Oracle 18c
- DB2 11.1
- PostgreSQL 10
- MySQL 8.0

Entity の更新処理

Entity の更新方法について、目的別に実装例を説明する。

Entity の 1 件更新

Entity を 1 件更新する際の実装例を以下に示す。

注釈: 以降の説明では、バージョンカラムを使用して楽観ロックを行う実装例となっているが、楽観ロックの必要がない場合は、楽観ロック関連の処理を行う必要はない。

排他制御の詳細については、「排他制御」を参照されたい。

- Repository インタフェースにメソッドを定義する。

```
package com.example.domain.repository.todo;

import com.example.domain.model.Todo;

public interface TodoRepository {

    // (1)
    boolean update(Todo todo);

}
```

項番	説明
1.	上記例では、引数に指定された Todo オブジェクトを 1 件更新するためのメソッドとして、update メソッドを定義している。

注釈: Entity を 1 件更新するメソッドの返り値について

Entity を 1 件更新するメソッドの返り値は、基本的には boolean でよい。

ただし、更新結果が複数件になった場合にデータ不整合エラーとして扱う必要がある場合は、数値型 (int 又は long) を返り値にし、更新件数が 1 件であることをチェックする必要がある。主キーが更新条件となっている場合は、更新結果が複数件になる事はないので、boolean でよい。

- 返り値として boolean を指定した場合は、更新件数が 0 件の際は false、更新件数が 1 件以上の際は true が返却される。
- 返り値として数値型を指定した場合は、更新件数が返却される。

- マッピングファイルに SQL を定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <!-- (2) -->
    <update id="update" parameterType="Todo">
        UPDATE
```

```
            t_todo
        SET
            todo_title = #{todoTitle},
            finished = #{finished},
            version = version + 1
        WHERE
            todo_id = #{todoId}
        AND
            version = #{version}
    </update>

</mapper>
```

項番	説明
2.	update 要素の中に、UPDATE する SQL を実装する。 id 属性には、Repository インタフェースに定義したメソッドのメソッド名を指定する。 update 要素の詳細については、「MyBatis3 REFERENCE DOCUMENTATION (Mapper XML Files-insert, update and delete-)」を参照されたい。 SET 句及び WHERE 句にバインドする値は、#{variableName}形式のバインド変数として指定する。上記例では、Repository インタフェースの引数として JavaBean(Todo) を指定しているため、バインド変数名には JavaBean のプロパティ名を指定する。

- Service クラスに Repository を DI し、Repository インターフェースのメソッドを呼び出す。

```
package com.example.domain.service.todo;

import javax.inject.Inject;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.example.domain.model.Todo;
import com.example.domain.repository.todo.TodoRepository;

@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    // (3)
    @Inject
    TodoRepository todoRepository;
```

```
    @Override
    public Todo update(Todo todo) {

        // (4)
        Todo currentTodo = todoRepository.findOne(todo.getTodoId());
        if (currentTodo == null || currentTodo.getVersion() != todo.
↪getVersion()) {
            throw new ObjectOptimisticLockingFailureException(Todo.class, todo
                    .getTodoId());
        }

        // (5)
        currentTodo.setTodoTitle(todo.getTodoTitle());
        currentTodo.setFinished(todo.isFinished());

        // (6)
        boolean updated = todoRepository.update(currentTodo);
        // (7)
        if (!updated) {
            throw new ObjectOptimisticLockingFailureException(Todo.class,
                    currentTodo.getTodoId());
        }
        currentTodo.setVersion(todo.getVersion() + 1);

        return currentTodo;
    }

}
```

項番	説明
3.	Service クラスに Repository インターフェースを DI する。
4.	更新対象の Entity をデータベースより取得する。 上記例では、Entity が更新されている場合 (レコードが削除されている場合又はバージョンが更新されている場合) は、Spring Framework から提供されている楽観ロック例外 (`org.springframework.orm.ObjectOptimisticLockingFailureException`) を発生させている。
5.	更新対象の Entity に対して、更新内容を反映する。 上記例では、「タイトル」「完了フラグ」を反映している。更新項目が少ない場合は上記実装例のままでもよいが、更新項目が多い場合は、「*Bean マッピング (Dozer)*」を使用することを推奨する。
6.	Repository インターフェースのメソッドを呼び出し、Entity を 1 件更新する。
7.	Entity の更新結果を判定する。 上記例では、Entity が更新されなかった場合 (レコードが削除されている場合又はバージョンが更新されている場合) は、Spring Framework から提供されている楽観ロック例外 (`org.springframework.orm.ObjectOptimisticLockingFailureException`) を発生させている。

ちなみに: 上記例では、更新処理が成功した後に、

```
currentTodo.setVersion(todo.getVersion() + 1);
```

としている。

これはデータベースに更新したバージョンと、Entity が保持するバージョンを合わせるための処理である。

呼び出し元 (Controller や JSP など) の処理でバージョンを参照する場合は、データベースの状態と Entity の状態を一致させておかないと、データ不整合が発生し、アプリケーションが期待通りの動作しない事になる。

Entity の一括更新

Entity を一括で更新する際の実装例を以下に示す。

Entity を一括で更新する場合は、

- 複数のレコードを同時に更新する UPDATE 文を発行する
- JDBC のバッチ更新機能を使用する

方法がある。

JDBC のバッチ更新機能を使用する方法については、「バッチモードの利用」を参照されたい。

ここでは、複数のレコードを同時に更新する UPDATE 文を発行する方法について説明する。

- Repository インタフェースにメソッドを定義する。

```
package com.example.domain.repository.todo;

import com.example.domain.model.Todo;
import org.apache.ibatis.annotations.Param;

import java.util.List;

public interface TodoRepository {

    // (1)
    int updateFinishedByTodIds(@Param("finished") boolean finished,
                               @Param("todoIds") List<String> todoIds);

}
```

項番	説明
1.	上記例では、引数に指定された ID のリストに該当するレコードの finished カラムを更新するためのメソッドとして、updateFinishedByTodIds メソッドを定義している。

注釈: Entity を一括更新するメソッドの返り値について

Entity を一括更新するメソッドの返り値は、数値型 (int 又は long) でよい。数値型にすると、更新されたレコード数を取得する事ができる。

- マッピングファイルに SQL を定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <update id="updateFinishedByTodIds">
        UPDATE
            t_todo
        SET
            finished = #{finished},
            /* (2) */
            version = version + 1
        WHERE
            /* (3) */
            <foreach item="todoId" collection="todoIds"
                     open="todo_id IN (" separator="," close=")">
                #{todoId}
            </foreach>
    </update>

</mapper>
```

項番	属性	説明
2.	-	バージョンカラムを使用して楽観ロックを行う場合は、バージョンカラムを更新する。 更新しないと、楽観ロック制御が正しく動作しなくなる。排他制御の詳細については、「排他制御」を参照されたい。
3.	-	WHERE 句に複数レコードを更新するための更新条件を指定する。
	-	`foreach` 要素を使用して、引数で渡された ID のリストに対して繰り返し処理を行う。 上記例では、引数で渡された ID のリストより、IN 句を生成している。 `foreach` の詳細については、「MyBatis3 REFERENCE DOCUMENTATION (Dynamic SQL-foreach-)」を参照されたい。
	collection	処理対象のコレクションを指定する。 上記例では、Repository のメソッド引数の ID のリスト (`todoIds`) に対して繰り返し処理を行っている。
	item	リストの中の 1 要素を保持するローカル変数名を指定する。
	separator	リスト内の要素間を区切るための文字列を指定する。 上記例では、IN 句の区切り文字である」,」を指定している。

Entity の削除処理

Entity の 1 件削除

Entity を 1 件削除する際の実装例を以下に示す。

注釈: 以降の説明では、バージョンカラムを使用した楽観ロックを行う実装例となっているが、楽観ロックの必要がない場合は、楽観ロック関連の処理を行う必要はない。

排他制御の詳細については、「排他制御」を参照されたい。

- Repository インタフェースにメソッドを定義する。

```
package com.example.domain.repository.todo;

import com.example.domain.model.Todo;

public interface TodoRepository {

    // (1)
    boolean delete(Todo todo);

}
```

項番	説明
1.	上記例では、引数に指定された Todo オブジェクトを 1 件削除するためのメソッドとして、`delete` メソッドを定義している。

注釈: Entity を 1 件削除するメソッドの返り値について

Entity を 1 件削除するメソッドの返り値は、基本的には `boolean` でよい。

ただし、削除結果が複数件になった場合にデータ不整合エラーとして扱う必要がある場合は、数値型 (`int` 又は `long`) を返り値にし、削除件数が 1 件であることをチェックする必要がある。主キーが削除条件となっている場合は、削除結果が複数件になる事はないので、`boolean` でよい。

- 返り値として `boolean` を指定した場合は、削除件数が 0 件の際は `false`、削除件数が 1 件以上の際は `true` が返却される。
- 返り値として数値型を指定した場合は、削除件数が返却される。

- マッピングファイルに SQL を定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <!-- (2) -->
    <delete id="delete" parameterType="Todo">
        DELETE FROM
            t_todo
        WHERE
            todo_id = #{todoId}
        AND
            version = #{version}
    </delete>

</mapper>
```

項番	説明
2.	delete 要素の中に、DELETE する SQL を実装する。 id 属性には、Repository インタフェースに定義したメソッドのメソッド名を指定する。 delete 要素の詳細については、「MyBatis3 REFERENCE DOCUMENTATION (Mapper XML Files-insert, update and delete-)」を参照されたい。 WHERE 句にバインドする値は、#{variableName}形式のバインド変数として指定する。上記例では、Repository インタフェースの引数として JavaBean(Todo) を指定しているため、バインド変数名には JavaBean のプロパティ名を指定する。

- Service クラスに Repository を DI し、Repository インターフェースのメソッドを呼び出す。

```
package com.example.domain.service.todo;

import javax.inject.Inject;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.example.domain.model.Todo;
import com.example.domain.repository.todo.TodoRepository;

@Transactional
```

```
@Service
public class TodoServiceImpl implements TodoService {

    // (3)
    @Inject
    TodoRepository todoRepository;

    @Override
    public Todo delete(String todoId, long version) {

        // (4)
        Todo currentTodo = todoRepository.findOne(todoId);
        if (currentTodo == null || currentTodo.getVersion() != version) {
            throw new ObjectOptimisticLockingFailureException(Todo.class,␣
↪todoId);
        }

        // (5)
        boolean deleted = todoRepository.delete(currentTodo);
        // (6)
        if (!deleted) {
            throw new ObjectOptimisticLockingFailureException(Todo.class,
                    currentTodo.getTodoId());
        }

        return currentTodo;
    }

}
```

項番	説明
3.	Service クラスに Repository インターフェースを DI する。
4.	削除対象の Entity をデータベースより取得する。 上記例では、Entity が更新されている場合 (レコードが削除されている場合又はバージョンが更新されている場合) は、Spring Framework から提供されている楽観ロック例外 (`org.springframework.orm.ObjectOptimisticLockingFailureException`) を発生させている。
5.	Repository インターフェースのメソッドを呼び出し、Entity を 1 件削除する。
6.	Entity の削除結果を判定する。 上記例では、Entity が削除されなかった場合 (レコードが削除されている場合又はバージョンが更新されている場合) は、Spring Framework から提供されている楽観ロック例外 (`org.springframework.orm.ObjectOptimisticLockingFailureException`) を発生させている。

Entity の一括削除

Entity を一括で削除する際の実装例を以下に示す。

Entity を一括で削除する場合は、

- 複数のレコードを同時に削除する DELETE 文を発行する
- JDBC のバッチ更新機能を使用する

方法がある。

JDBC のバッチ更新機能を使用する方法については、「バッチモードの利用」を参照されたい。

ここでは、複数のレコードを同時に削除する DELETE 文を発行する方法について説明する。

- Repository インタフェースにメソッドを定義する。

```
package com.example.domain.repository.todo;

public interface TodoRepository {

    // (1)
    int deleteOlderFinishedTodo(Date criteriaDate);

}
```

項番	説明
1.	上記例では、基準日より前に作成され完了済みのレコードを削除するためのメソッドとして、`deleteOlderFinishedTodo` メソッドを定義している。

注釈: Entity を一括削除するメソッドの返り値について

Entity を一括削除するメソッドの返り値は、数値型 (`int` 又は `long`) でよい。数値型にすると、削除されたレコード数を取得する事ができる。

- マッピングファイルに SQL を定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <delete id="deleteOlderFinishedTodo" parameterType="date">
        <![CDATA[
        DELETE FROM
            t_todo
        /* (2) */
        WHERE
            finished = TRUE
        AND
            created_at  < #{criteriaDate}
        ]]>
    </delete>

</mapper>
```

項番	説明
3.	WHERE 句に複数レコードを更新するための削除条件を指定する。 上記例では、 • 完了済み (`finished` が `TRUE`) • 基準日より前に作成された (`created_at` が基準日より前) を削除条件として指定している。

動的 SQL の実装

動的 SQL を組み立てる実装例を以下に示す。

MyBatis3 では、動的に SQL を組み立てるための XML 要素と、OGNL ベースの式（Expression 言語）を使用することで、動的 SQL を組み立てる仕組みを提供している。

動的 SQL の詳細については、「MyBatis3 REFERENCE DOCUMENTATION (Dynamic SQL)」を参照されたい。

MyBatis3 では、動的に SQL を組み立てるために、以下の XML 要素を提供している。

項番	要素名	説明
1.	`if`	条件に一致した場合のみ、SQL の組み立てを行うための要素。
2.	`choose`	複数の選択肢の中から条件に一致する 1 つを選んで、SQL の組み立てを行うための要素。
3.	`where`	組み立てた WHERE 句に対して、接頭語及び末尾の付与や除去など行うための要素。
4.	`set`	組み立てた SET 句用に対して、接頭語及び末尾の付与や除去など行うための要素。
5.	`foreach`	コレクションや配列に対して繰り返し処理を行うための要素
6.	`bind`	OGNL 式の結果を変数に格納するための要素。 `bind` 要素を使用して格納した変数は、SQL 内で参照する事ができる。

ちなみに: 一覧には記載していないが、動的 SQL を組み立てるための XML 要素として `trim` 要素が提供されている。

`trim` 要素は、`where` 要素と `set` 要素をより汎用的にした XML 要素である。

ほとんどの場合は、`where` 要素と `set` 要素で要件を充たせるため、本ガイドラインでは `trim` 要素の説明は割愛する。`trim` 要素が必要になる場合は、「MyBatis3 REFERENCE DOCUMENTATION (Dynamic SQL-trim, where, set-)」を参照されたい。

---

if 要素の実装

`if` 要素は、指定した条件に一致した場合のみ、SQL の組み立てを行うための XML 要素である。

```
<select id="findAllByCriteria" parameterType="TodoCriteria" resultType="Todo">
    SELECT
        todo_id,
        todo_title,
        finished,
        created_at,
        version
    FROM
        t_todo
    WHERE
```

```
        todo_title LIKE #{todoTitle} || '%' ESCAPE '~'
    <!-- (1) -->
    <if test="finished != null">
        AND
            finished = #{finished}
    </if>
    ORDER BY
        todo_id
</select>
```

項番	説明
1.	if 要素の test 属性に、条件を指定する。 上記例では、検索条件として finished が指定されている場合に、finished カラムに対する条件を SQL に加えている。

上記の動的 SQL で生成される SQL(WHERE 句) は、以下 2 パターンとなる。

```
-- (1) finished == null
...
WHERE
    todo_title LIKE ? || '%' ESCAPE '~'
ORDER BY
    todo_id
```

```
-- (2) finished != null
...
WHERE
    todo_title LIKE ? || '%' ESCAPE '~'
AND
    finished = ?
ORDER BY
    todo_id
```

choose 要素の実装

choose 要素は、複数の選択肢の中から条件に一致する 1 つを選んで、SQL の組み立てを行うための XML 要素である。

```
<select id="findAllByCriteria" parameterType="TodoCriteria" resultType="Todo">
    SELECT
        todo_id,
        todo_title,
        finished,
        created_at,
```

```
        version
    FROM
        t_todo
    WHERE
        todo_title LIKE #{todoTitle} || '%' ESCAPE '~'
    <!-- (1) -->
    <choose>
        <!-- (2) -->
        <when test="createdAt != null">
            AND
                created_at <![CDATA[ > ]]> #{createdAt}
        </when>
        <!-- (3) -->
        <otherwise>
            AND
                created_at <![CDATA[ > ]]> CURRENT_DATE
        </otherwise>
    </choose>
    ORDER BY
        todo_id
</select>
```

項番	説明
1.	choose 要素に中に、when 要素と otherwise 要素を指定して、SQL を組み立てる条件を指定する。
2.	when 要素の test 属性に、条件を指定する。 上記例では、検索条件として createdAt が指定されている場合に、create_at カラムの値が指定日以降のレコードを抽出するための条件を SQL に加えている。
3.	otherwise 要素に、全ての when 要素に一致しない場合時に組み立てる SQL を指定する。 上記例では、create_at カラムの値が現在日以降のレコード (当日作成されたレコード) を抽出するための条件を SQL に加えている。

上記の動的 SQL で生成される SQL(WHERE 句) は、以下 2 パターンとなる。

```
-- (1) createdAt!=null
...
WHERE
    todo_title LIKE ? || '%' ESCAPE '~'
AND
    created_at    >    ?
ORDER BY
    todo_id
```

```
-- (2) createdAt==null
...
WHERE
    todo_title LIKE ? || '%' ESCAPE '~'
```

```
AND
    created_at > CURRENT_DATE
ORDER BY
    todo_id
```

where 要素の実装

where 要素は、WHERE 句を動的に生成するための XML 要素である。

where 要素を使用すると、

- WHERE 句の付与
- AND 句、OR 句の除去

などが行われるため、シンプルに WHERE 句を組み立てる事ができる。

```
<select id="findAllByCriteria2" parameterType="TodoCriteria" resultType="Todo
↪">
    SELECT
        todo_id,
        todo_title,
        finished,
        created_at,
        version
    FROM
        t_todo
    <!-- (1) -->
    <where>
        <!-- (2) -->
        <if test="finished != null">
            AND
                finished = #{finished}
        </if>
        <!-- (3) -->
        <if test="createdAt != null">
            AND
                created_at <![CDATA[ > ]]> #{createdAt}
        </if>
    </where>
    ORDER BY
        todo_id
</select>
```

項番	説明
1.	where 要素に中で、WHERE 句を組み立てるための動的 SQL を実装する。 where 要素内で組み立てた SQL に応じて、WHERE 句の付与や、AND 句及び OR の除去などが行われる。
2.	動的 SQL を組み立てる。 上記例では、検索条件として finished が指定されている場合に、finished カラムに対する条件を SQL に加えている。
3.	動的 SQL を組み立てる。 上記例では、検索条件として createdAt が指定されている場合に、created_at カラムに対する条件を SQL に加えている。

上記の動的 SQL で生成される SQL(WHERE 句) は、以下 4 パターンとなる。

```
-- (1) finished != null && createdAt != null
...
FROM
    t_todo
WHERE
    finished = ?
AND
    created_at  >  ?
ORDER BY
    todo_id
```

```
-- (2) finished != null && createdAt == null
...
FROM
    t_todo
WHERE
    finished = ?
ORDER BY
    todo_id
```

```
-- (3) finished == null && createdAt != null
...
FROM
    t_todo
WHERE
    created_at  >  ?
ORDER BY
    todo_id
```

```
-- (4) finished == null && createdAt == null
...
FROM
    t_todo
ORDER BY
    todo_id
```

set 要素の実装例

set 要素は、SET 句を動的に生成するための XML 要素である。

set 要素を使用すると、

- SET 句の付与
- 末尾のカンマの除去

などが行われるため、シンプルに SET 句を組み立てる事ができる。

```
<update id="update" parameterType="Todo">
    UPDATE
        t_todo
    <!-- (1)  -->
    <set>
        version = version + 1,
        <!-- (2) -->
        <if test="todoTitle != null">
            todo_title = #{todoTitle}
        </if>
    </set>
    WHERE
        todo_id = #{todoId}
</update>
```

項番	説明
1.	set 要素に中で、SET 句を組み立てるための動的 SQL を実装する。 set 要素内で組み立てた SQL に応じて、SET 句の付与や、末尾のカンマの除去などが行われる。
2.	動的 SQL を組み立てる。 上記例では、更新項目として todoTitle が指定されている場合に、todo_title カラムを更新カラムとして SQL に加えている。

上記の動的 SQL で生成される SQL は、以下 2 パターンとなる。

```
-- (1) todoTitle != null
UPDATE
    t_todo
SET
    version = version + 1,
    todo_title = ?
WHERE
    todo_id = ?
```

```
-- (2) todoTitle == null
UPDATE
    t_todo
SET
   version = version + 1
WHERE
    todo_id = ?
```

foreach 要素の実装例

foreach 要素は、コレクションや配列に対して繰り返し処理を行うための XML 要素である。

```
<select id="findAllByCreatedAtList" parameterType="list" resultType="Todo">
    SELECT
        todo_id,
        todo_title,
        finished,
        created_at,
        version
    FROM
        t_todo
    <where>
        <!-- (1) -->
        <if test="list != null">
            <!-- (2) -->
            <foreach collection="list" item="date" separator="OR">
            <![CDATA[
                (created_at >= #{date} AND created_at < DATEADD('DAY', 1, #
↪{date}))
            ]]>
            </foreach>
        </if>
    </where>
    ORDER BY
        todo_id
</select>
```

項番	属性	説明
1.	-	繰り返し処理を行う対象のコレクション又は配列に対して、null チェックを行う。 null にならない事がない場合は、このチェックは実装しなくてもよい。
2.	-	foreach 要素を使用して、コレクションや配列に対して繰り返し処理を行い、動的 SQL を組み立てる。 上記例では、レコードの作成日付が、指定された日付 (日付リスト) の何れかと一致するレコードを検索するための WHERE 句を組み立てている。
	collection	collection 属性に、繰り返し処理を行うコレクションや配列を指定する。 上記例では、Repository メソッドの引数に指定されたコレクションを指定している。
	item	item 属性に、リストの中の 1 要素を保持するローカル変数名を指定する。 上記例では、collection 属性に日付リストを指定しているので、date という変数名を指定している。
	separator	separator 属性に、要素間の区切り文字列を指定する。 上記例では、OR 条件の WHERE 句を組み立てている。

ちなみに: 上記例では使用していないが、 foreach 要素には、以下の属性が存在する。

項番	属性	説明
1.	open	コレクションの先頭要素を処理する前に設定する文字列を指定する。
2.	close	コレクションの末尾要素を処理した後に設定する文字列を指定する。
3.	index	ループ番号を格納する変数名を指定する。

index 属性を使用するケースはあまりないが、open 属性と close 属性は、IN 句などを動的に生成する際に使用される。

以下に、IN 句を作成する際の foreach 要素の使用例を記載しておく。

```
<foreach collection="list" item="statusCode"
        open="AND order_status IN ("
        separator=","
        close=")">
    #{statusCode}
</foreach>
```

以下の様な SQL が組み立てられる。

```
-- list=['accepted','checking']
...
AND order_status IN (?,?)
```

上記の動的 SQL で生成される SQL(WHERE 句) は、以下 3 パターンとなる。

```
-- (1) list=null or statusCodes=[]
...
FROM
    t_todo
ORDER BY
    todo_id
```

```
-- (2) list=['2014-01-01']
...
FROM
    t_todo
WHERE
    (created_at >= ? AND created_at < DATEADD('DAY', 1, ?))
ORDER BY
    todo_id
```

```
-- (3) list=['2014-01-01','2014-01-02']
...
FROM
    t_todo
WHERE
    (created_at >= ? AND created_at < DATEADD('DAY', 1, ?))
OR
    (created_at >= ? AND created_at < DATEADD('DAY', 1, ?))
ORDER BY
    todo_id
```

bind 要素の実装例

bind 要素は、OGNL 式の結果を変数に格納するための XML 要素である。

```
<select id="findAllByCriteria" parameterType="TodoCriteria" resultType="Todo">
    <!-- (1) -->
    <bind name="escapedTodoTitle"
          value="@org.terasoluna.gfw.common.query.
↪QueryEscapeUtils@toLikeCondition(todoTitle)" />
    SELECT
        todo_id,
        todo_title,
        finished,
        created_at,
        version
    FROM
        t_todo
    WHERE
        /* (2) */
        todo_title LIKE #{escapedTodoTitle} || '%' ESCAPE '~'
    ORDER BY
        todo_id
</select>
```

項番	属性	説明
1.	-	bind 要素を使用して、OGNL 式の結果を変数に格納する 上記例では、OGNL 式を使ってメソッドを呼び出した結果を、変数に格納している。
	name	name 属性には、変数名を指定する。 ここで指定した変数名は、SQL のバインド変数として使用する事ができる。
	value	value 属性には、OGNL 式を指定する。 OGNL 式を実行した結果が、name 属性で指定した変数に格納される。 上記例では、共通ライブラリから提供しているメソッド(QueryEscapeUtils#toLikeCondition(String))を呼び出した結果を、escapedTodoTitle という変数に格納している。
2.	-	bind 要素を使用して作成した変数を、バインド変数として指定する。 上記例では、bind 要素を使用して作成した変数(escapedTodoTitle)を、バインド変数として指定している。

ちなみに: 上記例では、bind 要素を使用して作成した変数をバインド変数として指定しているが、置換変数として使用する事もできる。

バインド変数と置換変数については、「*SQL Injection 対策*」を参照されたい。

LIKE 検索時のエスケープ

LIKE 検索を行う場合は、検索条件として使用する値を LIKE 検索用にエスケープする必要がある。

LIKE 検索用のエスケープ処理は、共通ライブラリから提供している `org.terasoluna.gfw.common.query.QueryEscapeUtils` クラスのメソッドを使用することで実現する事ができる。

共通ライブラリから提供しているエスケープ処理の仕様については、「*LIKE 検索時のエスケープについて*」を参照されたい。

```
<select id="findAllByCriteria" parameterType="TodoCriteria" resultType="Todo">
    <!-- (1) -->
    <bind name="todoTitleContainingCondition"
          value="@org.terasoluna.gfw.common.query.
↪QueryEscapeUtils@toContainingCondition(todoTitle)" />
    SELECT
        todo_id,
        todo_title,
        finished,
        created_at,
        version
    FROM
        t_todo
    WHERE
        /* (2) (3) */
        todo_title LIKE #{todoTitleContainingCondition} ESCAPE '~'
    ORDER BY
        todo_id
</select>
```

項番	説明
1.	bind 要素 (OGNL 式) を使用して、共通ライブラリから提供している LIKE 検索用のエスケープ処理メソッドを呼び出す。 上記例では、部分一致用のエスケープ処理を行い todoTitleContainingCondition という変数に格納している。 QueryEscapeUtils@toContainingCondition(String) メソッドは、エスケープした文字列の前後に」%」を付与するメソッドである。
2.	部分一致用のエスケープを行った文字列を、LIKE 句のバインド変数として指定する。
3.	ESCAPE 句にエスケープ文字を指定する。 共通ライブラリから提供しているエスケープ処理では、エスケープ文字として」~」を使用しているため、ESCAPE 句に'~' を指定している。

ちなみに: 上記例では、部分一致用のエスケープ処理を行うメソッドを呼び出しているが、

- 前方一致用のエスケープ (`QueryEscapeUtils@toStartingWithCondition(String)`)
- 後方一致用のエスケープ (`QueryEscapeUtils@toEndingWithCondition(String)`)
- エスケープのみ (`QueryEscapeUtils@toLikeCondition(String)`)

を行うメソッドも用意されている。

詳細は「*LIKE* 検索時のエスケープについて」を参照されたい。

注釈: 上記例では、マッピングファイル内でエスケープ処理を行うメソッドを呼び出しているが、Repository のメソッドを呼び出す前に、Service の処理としてエスケープ処理を行う方法もある。

コンポーネントの役割としては、マッピングファイルでエスケープ処理を行う方が適切なため、本ガイドラインとしては、マッピングファイル内でエスケープ処理を行う事を推奨する。

SQL Injection 対策

SQL を組み立てる際は、SQL Injection が発生しないように注意する必要がある。

MyBatis3 では、SQL に値を埋め込む仕組みとして、以下の 2 つの方法を提供している。

項番	方法	説明
1.	バインド変数を使用して埋め込む	この方法を使用すると、 SQL 組み立て後に `java.sql.PreparedStatement` を使用して値が埋め込められるため、**安全に値を埋め込むことができる。** **ユーザからの入力値を SQL に埋め込む場合は、原則バインド変数を使用すること。**
2.	置換変数を使用して埋め込む	この方法を使用すると、SQL を組み立てるタイミングで文字列として置換されてしまうため、**安全な値の埋め込みは保証されない。**

警告: ユーザからの入力値を置換変数を使って埋め込むと、SQL Injection が発生する危険性が高くなることを意識すること。

ユーザからの入力値を置換変数を使って埋め込む必要がある場合は、SQL Injection が発生しないことを保障するために、かならず入力チェックを行うこと。

基本的には、 **ユーザからの入力値はそのまま使わないことを強く推奨する。**

バインド変数を使って埋め込む方法

バインド変数の使用例を以下に示す。

```
<insert id="create" parameterType="Todo">
    INSERT INTO
        t_todo
    (
        todo_id,
        todo_title,
        finished,
        created_at,
        version
    )
    VALUES
    (
        /* (1) */
        #{todoId},
        #{todoTitle},
        #{finished},
        #{createdAt},
        #{version}
    )
</insert>
```

項番	説明
1.	バインドする値が格納されているプロパティのプロパティ名を、#{ と」} 「で囲み、バインド変数として指定する。

ちなみに: バインド変数には、いくつかの属性を指定する事が出来る。

指定できる属性としては、

- javaType
- jdbcType
- typeHandler
- numericScale
- mode
- resultMap
- jdbcTypeName

がある。

基本的には、単純にプロパティ名を指定するだけで、MyBatis が適切な振る舞いを選択してくれる。上記属性は、MyBatis が適切な振る舞いを選択してくれない時に指定すればよい。

属性の使い方については、「MyBatis3 REFERENCE DOCUMENTATION(Mapper XML Files-Parameters-)」を参照されたい。

置換変数を使って埋め込む方法

置換変数の使用例を以下に示す。

- Repository インタフェースにメソッドを定義する。

```
public interface TodoRepository {
    List<Todo> findAllByCriteria(@Param("criteria") TodoCriteria criteria,
                                 @Param("direction") String direction);
}
```

- マッピングファイルに SQL を実装する。

```
<select id="findAllByCriteria" parameterType="TodoCriteria" resultType="Todo">
    <bind name="todoTitleContainingCondition"
          value="@org.terasoluna.gfw.common.query.
↪QueryEscapeUtils@toContainingCondition(criteria.todoTitle)" />
    SELECT
        todo_id,
        todo_title,
        finished,
        created_at,
        version
    FROM
        t_todo
    WHERE
        todo_title LIKE #{todoTitleContainingCondition} ESCAPE '~'
    ORDER BY
        /* (1) */
        todo_id ${direction}
</select>
```

項番	説明
1.	置換する値が格納されているプロパティのプロパティ名を`${`と`}`で囲み、置換変数として指定する。上記例では、`${direction}`の部分は、`DESC`または`ASC`で置換される。

警告: 置換変数による埋め込みは、必ずアプリケーションとして安全な値であることを担保した上で、テーブル名、カラム名、ソート条件などに限定して使用することを推奨する。

例えば以下のように、コード値と SQL に埋め込むための値のペアを Map に格納しておき、

```
Map<String, String> directionMap = new HashMap<String, String>();
directionMap.put("1", "ASC");
directionMap.put("2", "DESC");
```

入力値はコード値として扱い、SQL を実行する処理の中で安全な値に変換することが望ましい。

```
String direction = directionMap.get(directionCode);
todoRepository.findAllByCriteria(criteria, direction);
```

上記例では Map を使用しているが、共通ライブラリから提供している「コードリスト」を使用しても良い。「コードリスト」を使用すると、入力チェックと連動する事ができるため、より安全に値の埋め込みを行う事ができる。

- projectName-domain/src/main/resources/META-INF/spring/projectName-codelist.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="CL_DIRECTION" class="org.terasoluna.gfw.common.codelist.
↪SimpleMapCodeList">
        <property name="map">
            <map>
                <entry key="1" value="ASC" />
                <entry key="2" value="DESC" />
            </map>
        </property>
    </bean>
</beans>
```

- Service クラス

```
@Inject
@Named("CL_DIRECTION")
CodeList directionCodeList;

// ...

public List<Todo> searchTodos(TodoCriteria criteria, String␣
↪directionCode){
    String direction = directionCodeList.asMap().get(directionCode);
    List<Todo> todos = todoRepository.findAllByCriteria(criteria,␣
↪direction);
    return todos;
}
```

6.2.3 How to extend

SQL 文の共有

SQL 文を複数の SQL で共有する方法について、説明を行う。

MyBatis3 では、 `sql` 要素と `include` 要素を使用することで、SQL 文 (又は SQL 文の一部) を共有する事ができる。

> **注釈: SQL 文の共有化の使用例**
>
> ページネーション検索を実現する場合は、「検索条件に一致する Entity の総件数を取得する SQL」と「検索条件に一致する Entity のリストを取得する SQL」の WHERE 句は共有した方がよい。

マッピングファイルの実装例は以下の通り。

```
<!-- (1)  -->
<sql id="findPageByCriteriaWherePhrase">
    <![CDATA[
    WHERE
        todo_title LIKE #{title} || '%' ESCAPE '~'
    AND
        created_at < #{createdAt}
    ]]>
</sql>

<select id="countByCriteria" resultType="_long">
    SELECT
        COUNT(*)
    FROM
        t_todo
    <!-- (2)  -->
    <include refid="findPageByCriteriaWherePhrase"/>
</select>

<select id="findPageByCriteria" resultType="Todo">
    SELECT
        todo_id,
        todo_title,
        finished,
        created_at,
        version
```

```
    FROM
        t_todo
    <!-- (2)  -->
    <include refid="findPageByCriteriaWherePhrase"/>
    ORDER BY
        todo_id
</select>
```

項番	説明
1.	sql 要素の中に、複数の SQL で共有する SQL 文を実装する。 id 属性には、マッピングファイル内でユニークとなる ID を指定する。
2.	include 要素を使用して、インクルードする SQL を指定する。 refid 属性には、インクルードする SQL の ID(sql 要素の id 属性に指定した値) を指定する。

TypeHandler の実装

MyBatis3 の標準でサポートされていない Joda-Time のクラスとのマッピングが必要の場合、独自の TypeHandler の作成が必要となる。

本ガイドラインでは「*Joda-Time 用の TypeHandler の実装*」を例に、TypeHandler の実装方法について説明する。

作成した TypeHandler をアプリケーションに適用する方法については、「*TypeHandler の設定*」を参照されたい。

注釈: BLOB 用と CLOB 用の実装について

MyBatis 3.4 で追加された TypeHandler は、JDBC 4.0 (Java 1.6) で追加された API を使用することで、BLOB と java.io.InputStream 、CLOB と java.io.Reader の変換を実現している。JDBC 4.0 サポートの JDBC ドライバーであれば、BLOB ⇔ InputStream 、CLOB ⇔ Reader 変換用のタイプハンドラーがデフォルトで有効になるため、TypeHandler を新たに実装する必要はない。

JDBC 4.0 との互換性のない JDBC ドライバを使う場合は、利用する JDBC ドライバの互換バージョンを意識した TypeHandler を作成する必要がある。

例えば、PostgreSQL 用の JDBC ドライバ (postgresql-42.2.5.jar) では、JDBC 4.0 から追加されたメソッドの一部が、未実装の状態である。

Joda-Time 用の TypeHandler の実装

MyBatis3 では、Joda-Time のクラス (`org.joda.time.DateTime`、`org.joda.time.LocalDateTime`、`org.joda.time.LocalDate` など) はサポートされていない。
そのため、Entity クラスのフィールドに Joda-Time のクラスを使用する場合は、Joda-Time 用の `TypeHandler` を用意する必要がある。

`org.joda.time.DateTime` と `java.sql.Timestamp` をマッピングするための `TypeHandler` の実装例を、以下に示す。

> **注釈:** Jada-Time から提供されている他のクラス (`LocalDateTime`、`LocalDate`、`LocalTime` など) も同じ要領で実装すればよい。

```
package com.example.infra.mybatis.typehandler;

import java.sql.CallableStatement;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Timestamp;

import org.apache.ibatis.type.BaseTypeHandler;
import org.apache.ibatis.type.JdbcType;
import org.joda.time.DateTime;

// (1)
public class DateTimeTypeHandler extends BaseTypeHandler<DateTime> {

    // (2)
    @Override
    public void setNonNullParameter(PreparedStatement ps, int i,
            DateTime parameter, JdbcType jdbcType) throws SQLException {
        ps.setTimestamp(i, new Timestamp(parameter.getMillis()));
    }

    // (3)
    @Override
    public DateTime getNullableResult(ResultSet rs, String columnName)
            throws SQLException {
        return toDateTime(rs.getTimestamp(columnName));
    }

    // (3)
    @Override
    public DateTime getNullableResult(ResultSet rs, int columnIndex)
            throws SQLException {
        return toDateTime(rs.getTimestamp(columnIndex));
    }
```

```
    // (3)
    @Override
    public DateTime getNullableResult(CallableStatement cs, int columnIndex)
            throws SQLException {
        return toDateTime(cs.getTimestamp(columnIndex));
    }

    private DateTime toDateTime(Timestamp timestamp) {
        // (4)
        if (timestamp == null) {
            return null;
        } else {
            return new DateTime(timestamp.getTime());
        }
    }

}
```

項番	説明
1.	MyBatis3 から提供されている BaseTypeHandler を親クラスに指定する。 その際、BaseTypeHandler のジェネリック型には、DateTime を指定する。
2.	DateTime を Timestamp に変換し、PreparedStatement に設定する処理を実装する。
3.	ResultSet 又は CallableStatement から取得した Timestamp を DateTime に変換し、返り値として返却する。
4.	null を許可するカラムの場合、Timestamp が null になる可能性があるため、null チェックを行ってから DateTime に変換する必要がある。 上記実装例では、3 つのメソッドで同じ処理が必要になるため、private メソッドを作成している。

ResultHandler の実装

MyBatis3 では、検索結果を 1 件単位で処理する仕組みを提供している。

この仕組みを利用すると、

- DB より取得した値を Java の処理で加工する
- DB より取得した値などを Java の処理として集計する

といった処理を行う際に、同時に消費するメモリの容量を最小限に抑える事ができる。

例えば、検索結果を CSV 形式のデータとしてダウンロードするような処理を実装する場合は、検索結果を 1 件単位で処理する仕組みを使用するとよい。

注釈: 検索結果が大量になる可能性があり、且つ Java の処理で検索結果を 1 件ずつ処理する必要がある場合は、この仕組みを使用することを強く推奨する。

検索結果を 1 件単位で処理する仕組みを使用しない場合、検索結果の全データ「1 データのサイズ * 検索結果件数」をメモリ上に同時に確保することになり、全てのデータに対して処理が終了するまで GC 候補になることはない。

一方、検索結果を 1 件単位で処理する仕組みを使用した場合、基本的には「1 データのサイズ」をメモリ上に確保するだけであり、1 データの処理を終えた時点で GC 候補となる。

例えば「1 データのサイズ」が 2KB で「検索結果件数」が 10,000 件だった場合、

- まとめて処理を行う場合は、20MB のメモリ
- 1 件単位で処理を行う場合は、2KB のメモリ

が同時に消費される。シングルスレッドで動くアプリケーションであれば問題になる事はないが、Web アプリケーションの様なマルチスレッドで動くアプリケーションの場合は、問題になる事がある。

仮に 100 スレッドで同時に処理を行った場合、

- まとめて処理を行う場合は、2GB のメモリ
- 1 件単位で処理を行う場合は、200KB のメモリ

が同時に消費される。

結果として、

- まとめて処理を行う場合は、ヒープの最大サイズの指定によっては、メモリ枯渇によるシステムダウンやフル GC の頻発による性能劣化などが起こる可能性が高まる。
- 1 件単位で処理を行う場合は、メモリ枯渇やコストの高い GC 処理が発生する可能性を抑える事ができる。

上記に挙げた数字は目安であり、実際の計測値ではないという点を補足しておく。

以下に、検索結果を CSV 形式のデータとしてダウンロードする処理の実装例を示す。

- Repository インタフェースにメソッドを定義する。

```
public interface TodoRepository {

```

```
    // (1) (2)
    void collectAllByCriteria(TodoCriteria criteria, ResultHandler<Todo>␣
↪resultHandler);

}
```

項番	説明
1.	メソッドの引数として、 org.apache.ibatis.session.ResultHandler を指定する。
2.	メソッドの返り値は、void 型を指定する。 void 以外を指定すると、ResultHandler が呼び出されなくなるので、注意すること。

- マッピングファイルに SQL を定義する。

```
<!-- (3) -->
<select id="collectAllByCriteria" parameterType="TodoCriteria" resultType=
↪"Todo">
    SELECT
        todo_id,
        todo_title,
        finished,
        created_at,
        version
    FROM
        t_todo
    <where>
        <if test="title != null">
            <bind name="titleContainingCondition"
                  value="@org.terasoluna.gfw.common.query.
↪QueryEscapeUtils@toContainingCondition(title)" />
            todo_title LIKE #{titleContainingCondition} ESCAPE '~'
        </if>
        <if test="createdAt != null">
            <![CDATA[
            AND created_at < #{createdAt}
            ]]>
        </if>
    </where>
</select>
```

項番	説明
3.	マッピングファイルの実装は、通常の検索処理と同じである。

警告: fetchSize 属性の指定について

大量のデータを返すようなクエリを記述する場合には、`fetchSize` 属性に適切な値を設定すること。`fetchSize` は、JDBC ドライバとデータベース間の 1 回の通信で取得するデータの件数を設定するパラメータである。なお、MyBatis 3.3.0 以降のバージョンでは、MyBatis 設定ファイルに「デフォルトの `fetchSize`」を指定することができる。`fetchSize` の詳細は「*fetchSize の設定*」を参照されたい。

- Service クラスに Repository を DI し、Repository インターフェースのメソッドを呼び出す。

```
public class TodoServiceImpl implements TodoService {

    private static final DateTimeFormatter DATE_FORMATTER =
        DateTimeFormat.forPattern("yyyy/MM/dd");

    @Inject
    TodoRepository todoRepository;

    public void downloadTodos(TodoCriteria criteria,
        final BufferedWriter downloadWriter) {

        // (4)
        ResultHandler<Todo> handler = new ResultHandler<Todo>() {
            @Override
            public void handleResult(ResultContext<? extends Todo> context) {
                Todo todo = context.getResultObject();
                StringBuilder sb = new StringBuilder();
                try {
                    sb.append(todo.getTodoId());
                    sb.append(",");
                    sb.append(todo.getTodoTitle());
                    sb.append(",");
                    sb.append(todo.isFinished());
                    sb.append(",");
                    sb.append(DATE_FORMATTER.print(todo.getCreatedAt().
↪getTime()));
                    downloadWriter.write(sb.toString());
                    downloadWriter.newLine();
                } catch (IOException e) {
```

```
                    throw new SystemException("e.xx.fw.9001", e);
                }
            }
        };

        // (5)
        todoRepository.collectAllByCriteria(criteria, handler);

    }

}
```

項番	説明
4.	ResultHandler のインスタンスを生成する。 ResultHandler の handleResult メソッドの中に、1 件毎に行う処理を実装する。 上記例では、ResultHandler の実装クラスは作らず、無名オブジェクトとして ResultHandler の実装を行っている。実装クラスを作成してもよいが、複数の処理で共有する必要がない場合は、無理に実装クラスを作成する必要はない。
5.	Repository インタフェースのメソッドを呼び出す。 メソッドを呼び出す際に、(4) で生成した ResultHandler のインスタンスを引数に指定する。 ResultHandler を使用した場合、MyBatis は以下の処理を検索結果の件数分繰り返す。 • 検索結果からレコードを取得し、JavaBean にマッピングを行う。 • ResultHandler インスタンスの handleResult(ResultContext) メソッドを呼び出す。

警告：ResultHandler 使用時の注意点

ResultHandler を使用する場合、以下の 2 点に注意すること。

- MyBatis3 では、検索処理の性能向上させる仕組みとして、検索結果をローカルキャッシュ及びグローバルな 2 次キャッシュに保存する仕組みを提供しているが、ResultHandler を引数に取るメソッドから返されるデータはキャッシュされない。
- 手動マッピングを使用して複数行のデータを一つの Java オブジェクトにマッピングするステートメントに対して ResultHandler を使用した場合、不完全な状態 (関連 Entity のオブジェクトがマッピングされる前の状態) のオブジェクトが渡されるケースがある。

ちなみに：**ResultContext** のメソッドについて

ResultHandler#handleResult メソッドの引数である ResultContext には、以下のメソッドが用意されている。

項番	メソッド	説明
1.	getResultObject	検索結果がマッピングされたオブジェクトを取得するためのメソッド。
2.	getResultCount	ResultHandler#handleResult メソッドの呼び出し回数を取得するためのメソッド。
3.	stop	以降のレコードに対する処理を中止するように MyBatis 側に通知するためのメソッド。このメソッドは、以降のレコードを全て破棄したい場合に使用するとよい。

ResultContext には isStopped というメソッドもあるが、これは MyBatis 側が使用するメソッドなので、説明は割愛する。

SQL 実行モードの利用

MyBatis3 では、SQL を実行するモードとして以下の 3 種類を用意しており、デフォルトは SIMPLE である。

ここでは、

- 実行モードの使用方法
- バッチモードの Repository 利用時の注意点

について説明を行う。

実行モードの説明については、「*SQL 実行モードの設定*」を参照されたい。

PreparedStatement 再利用モードの利用

実行モードを SIMPLE から REUSE に変更した場合、MyBatis 内部の PreparedStatement の扱い方は変わるが、MyBatis の動作 (使い方) は変わらない。

実行モードをデフォルト (SIMPLE) から REUSE に変更する方法を、以下に示す。

- projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml に設定を追加する。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <settings>
        <!-- (1) -->
        <setting name="defaultExecutorType" value="REUSE"/>
    </settings>
</configuration>
```

項番	説明
1.	defaultExecutorType に REUSE に変更する。 上記設定を行うと、デフォルト動作が PreparedStatement 再利用モードになる。

バッチモードの利用

Mapper インタフェースの更新系メソッドの呼び出しを、全てバッチモードで実行する場合は、「*PreparedStatement 再利用モードの利用*」と同じ方法で、実行モードを BATCH モードに変更すればよい。

ただし、バッチモードはいくつかの制約事項があるため、実際のアプリケーション開発では SIMPLE 又は REUSE モードと共存して使用するケースが想定される。

例えば、

- 大量のデータ更新を伴い性能要件を充たす事が最優先される処理では、バッチモードを使用する。
- 楽観ロックの制御などデータの一貫性を保つために更新結果の判定が必要な処理では、SIMPLE 又は REUSE モードを使用する。

等の使い分けを行う場合である。

> **警告：実行モードを共存して使用する際の注意点**
>
> アプリケーション内で複数の実行モードを使用する場合は、**同一トランザクション内で実行モードを切り替える事が出来ないという点に注意すること。**
>
> 仮に同一トランザクション内で複数の実行モードを使用した場合は、MyBatis が矛盾を検知しエラーとなる。
>
> これは、同一トランザクション内の処理において、
>
> - XxxRepository のメソッド呼び出しは BATCH モードで実行する

- YyyRepository のメソッド呼び出しは REUSE モードで実行する

といった事が出来ないという事を意味する。

本ガイドラインをベースに作成するアプリケーションのトランザクション境界は、Service 又は Repository となる。そのため、**アプリケーション内で複数の実行モードを使用する場合は、Service や Repository の設計を行う際に、実行モードを意識する必要がある。**

トランザクションを分離させたい場合は、Service や Repository のメソッドアノテーションとして、`@Transactional(propagation = Propagation.REQUIRES_NEW)` を指定する事で実現する事ができる。トランザクション管理の詳細については、「トランザクション管理について」を参照されたい。

以降では、

- 複数の実行モードを共存させるための設定方法
- アプリケーションの実装例

について説明を行う。

個別にバッチモードの Repository を作成するための設定

特定の Repository に対してバッチモードの Repository を作成したい場合は、MyBatis-Spring から提供されている `org.mybatis.spring.mapper.MapperFactoryBean` を使用して、Repository の Bean 定義を行えばよい。

下記の設定例では、

- 通常使用する Repository として REUSE モードの Repository
- 特定の Repository に対して BATCH モードの Repository

を Bean 登録している。

- `projectName-domain/src/main/resources/META-INF/spring/projectName-infra.xml` に Bean 定義を追加する。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       https://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       https://www.springframework.org/schema/context/spring-context.xsd
       http://mybatis.org/schema/mybatis-spring
       http://mybatis.org/schema/mybatis-spring.xsd">

    <bean id="sqlSessionFactory"
          class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="configLocation"
                  value="classpath:META-INF/mybatis/mybatis-config.xml"/>
    </bean>

    <!-- (1) -->
    <bean id="sqlSessionTemplate"
          class="org.mybatis.spring.SqlSessionTemplate">
        <constructor-arg index="0" ref="sqlSessionFactory"/>
        <constructor-arg index="1" value="REUSE"/>
    </bean>

    <mybatis:scan base-package="com.example.domain.repository"
                  template-ref="sqlSessionTemplate"/> <!-- (2) -->

    <!-- (3) -->
    <bean id="batchSqlSessionTemplate"
          class="org.mybatis.spring.SqlSessionTemplate">
        <constructor-arg index="0" ref="sqlSessionFactory"/>
        <constructor-arg index="1" value="BATCH"/>
    </bean>

    <!-- (4) -->
    <bean id="todoBatchRepository"
          class="org.mybatis.spring.mapper.MapperFactoryBean">
        <!-- (5) -->
        <property name="mapperInterface"
                  value="com.example.domain.repository.todo.TodoRepository"/>
        <!-- (6) -->
        <property name="sqlSessionTemplate" ref="batchSqlSessionTemplate"/>
    </bean>

</beans>
```

項番	説明
1.	通常使用する Repository で利用するための `SqlSessionTemplate` を Bean 定義する。
2.	通常使用する Repository をスキャンし Bean 登録する。 `template-ref` 属性に、(1) で定義した `SqlSessionTemplate` を指定する。
3.	バッチモードの Repository で利用するための `SqlSessionTemplate` を Bean 定義する。
4.	バッチモード用の Repository を Bean 定義する。 `id` 属性には、(2) でスキャンした Repository の Bean 名と重複しない値を指定する。(2) でスキャンされた Repository の Bean 名は、インタフェース名を「lowerCamelCase」にした値にとなる。 上記例では、バッチモード用の `TodoRepository` が `todoBatchRepository` という名前の Bean で Bean 登録される。
5.	`mapperInterface` プロパティには、バッチモードを利用する Repository のインタフェース名 (FQCN) を指定する。
6.	`sqlSessionTemplate` プロパティには、(3) で定義したバッチモード用の `SqlSessionTemplate` を指定する。

一括でバッチモードの Repository を作成するための設定

一括でバッチモードの Repository を作成したい場合は、MyBatis-Spring から提供されているスキャン機能 (`mybatis:scan` 要素) を使用して、Repository の Bean 定義を行えばよい。

下記の設定例では、全ての Repository に対して、`REUSE` モードと `BATCH` モードの Repository を Bean 登録している。

- `BeanNameGenerator` を作成する。

```
package com.example.domain.repository;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.beans.factory.support.BeanNameGenerator;
import org.springframework.util.ClassUtils;

import java.beans.Introspector;

// (1)
public class BachRepositoryBeanNameGenerator implements BeanNameGenerator {
    // (2)
    @Override
```

```
    public String generateBeanName(BeanDefinition definition,␣
↪BeanDefinitionRegistry registry) {
        String defaultBeanName = Introspector.decapitalize(ClassUtils.
↪getShortName(definition
                .getBeanClassName()));
        return defaultBeanName.replaceAll("Repository", "BatchRepository");
    }
}
```

項番	説明
1.	Spring の ApplicationContext に登録する Bean 名を生成するクラスを作成する。 このクラスは、通常使用する REUSE モードの Repository の Bean 名と、BATCH モードの Bean 名が重複しないようにするために必要なクラスである。
2.	Bean 名を生成するためのメソッドを実装する。 上記例では、Bean 名の suffix を BatchRepository とする事で、通常使用される REUSE モードの Repository の Bean 名と重複しないようにしている。

- projectName-domain/src/main/resources/META-INF/spring/projectName-infra.xml に Bean 定義を追加する。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       https://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       https://www.springframework.org/schema/context/spring-context.xsd
       http://mybatis.org/schema/mybatis-spring
       http://mybatis.org/schema/mybatis-spring.xsd">

    <bean id="sqlSessionFactory"
          class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="configLocation"
                  value="classpath:META-INF/mybatis/mybatis-config.xml"/>
    </bean>

    <!-- ... -->

    <bean id="batchSqlSessionTemplate"
          class="org.mybatis.spring.SqlSessionTemplate">
        <constructor-arg index="0" ref="sqlSessionFactory"/>
```

```
        <constructor-arg index="1" value="BATCH"/>
    </bean>

    <!-- (3) -->
    <mybatis:scan base-package="com.example.domain.repository"
        template-ref="batchSqlSessionTemplate"
        name-generator="com.example.domain.repository.
↪BatchRepositoryBeanNameGenerator"/>

</beans>
```

項番	属性	説明
3.	-	mybatis:scan 要素を使用して、バッチモードの Repository を Bean 登録する。
	base-package	Repository をスキャンするベースパッケージを指定する。 指定パッケージの配下に存在する Repository インタフェースがスキャンされ、Spring の ApplicationContext に Bean 登録される。
	template-ref	バッチモード用の SqlSessionTemplate の Bean を指定する。
	name-generator	スキャンした Repository の Bean 名を生成するためのクラスを指定する。 具体的には、(1) で作成したクラスのクラス名 (FQCN) を指定する。 この指定を省略した場合、Bean 名が重複するため、バッチモードの Repository は Spring の ApplicationContext に登録されない。

バッチモードの Repository の使用例

以下に、バッチモードの Repository を使用してデータベースにアクセスするための実装例を示す。

```
@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    // (1)
    @Inject
    @Named("todoBatchRepository")
    TodoRepository todoBatchRepository;

    @Override
    public void updateTodos(List<Todo> todos) {
        for (Todo todo : todos) {
            // (2)
            todoBatchRepository.update(todo);
        }
```

```
    }

}
```

項番	説明
1.	バッチモードの Repository をインジェクションする。
2.	バッチモードの Repository のメソッドを呼び出し、Entity の更新を行う。 バッチモードの Repository の場合は、メソッドを呼び出したタイミングで SQL が実行されないため、メソッドから返却される更新結果は無視する必要がある。 Entity を更新するための SQL は、トランザクションがコミットされる直前にバッチ実行され、エラーがなければコミットされる。

注釈: バッチ実行のタイミングについて

SQL がバッチ実行されるタイミングは、基本的には以下の場合である。

- トランザクションがコミットされる直前
- クエリ (SELECT) を実行する直前

Repository のメソッドの呼び出し順番に関する注意点は、「*Repository のメソッドの呼び出し順番*」を参照されたい。

バッチモードの Repository 利用時の注意点

バッチモードの Repository を利用する場合、Service クラスの実装として、以下の点に注意する必要がある。

- 更新結果の判定
- 一意制約違反の検知方法
- *Repository のメソッドの呼び出し順番*

更新結果の判定

バッチモードの Repository を使用した場合、更新結果の妥当性をチェックする事ができない。

バッチモードを使用する場合、Mapper インタフェースのメソッドから返却される更新結果は、

- 返り値が数値 (`int` や `long`) の場合は、固定値 (`org.apache.ibatis.executor.BatchExecutor#BATCH_UPDATE_RETURN_VALUE`)

- 返り値が boolean の場合は、false

が返却される。

これは、Mapper インタフェースのメソッドを呼び出したタイミングでは SQL が発行されず、バッチ実行用にキューイング (java.sql.Statement#addBatch()) される仕組みになっているためである。

これは、以下の様な実装が出来ないことを意味している。

```
@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    @Inject
    @Named("todoBatchRepository")
    TodoRepository todoBatchRepository;

    @Override
    public void updateTodos(List<Todo> todos) {
        for (Todo todo : todos) {
            boolean updateSuccess = todoBatchRepository.update(todo);
            // (1)
            if (!updateSuccess) {
                // ...
            }
        }
    }

}
```

項番	説明
1.	上記例のように実装した場合、更新結果は常に false になるため、必ず更新失敗時の処理が実行されてしまう。

アプリケーションの要件によっては、バッチ実行した更新結果の妥当性をチェックすることが求められるケースも考えられる。そのようなケースでは、Mapper インタフェースに「バッチ実行用にキューイングされている SQL を実行するためのメソッド」を用意すればよい。

MyBatis 3.2 系では、org.apache.ibatis.session.SqlSession インタフェースの flushStatements メソッドを直接呼び出す必要があったが、MyBatis 3.3.0 以降のバージョンでは、Mapper インタフェースに@org.apache.ibatis.annotations.Flush アノテーションを付与したメソッドを作成する方法がサポートされている。

警告: バッチモード使用時の JDBC ドライバが返却する更新結果について

@Flush アノテーションを付与したメソッド (及び SqlSession インタフェースの flushStatements メソッド) を使用するとバッチ実行時の更新結果を受け取る事ができる

> と前述したが、JDBC ドライバから返却される更新結果が「処理したレコード数」になる保証はない。
>
> これは、使用する JDBC ドライバの実装に依存する部分なので、使用する JDBC ドライバの仕様を確認しておく必要がある。

以下に、`@Flush` アノテーションを付与したメソッドの作成例と呼び出し例を示す。

```
public interface TodoRepository {
    // ...
    @Flush // (1)
    List<BatchResult> flush();
}
```

```
@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    @Inject
    @Named("todoBatchRepository")
    TodoRepository todoBatchRepository;

    @Override
    public void updateTodos(List<Todo> todos) {

        for (Todo todo : todos) {
            todoBatchRepository.update(todo);
        }

        List<BatchResult> updateResults = todoBatchRepository.flush(); // (2)

        // Validate update results
        // ...

    }

}
```

項番	説明
1.	`@Flush` アノテーションを付与したメソッド（以降「`@Flush` メソッド」と呼ぶ）を作成する。 更新結果の判定が必要な場合は、返り値として `org.apache.ibatis.executor.BatchResult` のリスト型を指定する。更新結果の判定が不要な場合 (一意制約違反などのデータベースエラーのみをハンドリングしたい場合) は、返り値は `void` でよい。
2.	バッチ実行用にキューイングされている SQL を実行したいタイミングで、`@Flush` メソッドを呼び出す。`@Flush` メソッドを呼び出すと、Mapper インタフェースに紐づく `SqlSession` オブジェクトの `flushStatements` メソッドが呼び出されて、バッチ実行用にキューイングされている SQL が実行される。 更新結果の判定が必要な場合は、`@Flush` メソッドから返却される更新結果の妥当性チェックを行う。

一意制約違反の検知方法

バッチモードの Repository を使用した場合、一意制約違反などのデータベースエラーを Service の処理として検知する事が出来ないケースがある。

これは、Mapper インタフェースのメソッドを呼び出したタイミングでは SQL が発行されず、バッチ実行用にキューイング (`java.sql.Statement#addBatch()`) される仕組みになっているためであり、以下の様な実装が出来ないことを意味している。

```
@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    @Inject
    @Named("todoBatchRepository")
    TodoRepository todoBatchRepository;

    @Override
    public void storeTodos(List<Todo> todos) {
        for (Todo todo : todos) {
            try {
                todoBatchRepository.create(todo);
            // (1)
            } catch (DuplicateKeyException e) {
                // ....
            }
        }
    }

}
```

項番	説明
1.	上記例のように実装した場合、このタイミングで `org.springframework.dao.DuplicateKeyException` が発生することはないため、`DuplicateKeyException` 補足後の処理が実行される事はない。 これは、SQL がバッチ実行されるタイミングが、Service の処理が終わった後 (トランザクションがコミットされる直前) に行われるためである。

アプリケーションの要件によっては、バッチ実行時の一意制約違反を検知することが求められるケースも考えられる。そのようなケースでは、Mapper インタフェースに「バッチ実行用にキューイングされている SQL を実行するためのメソッド (`@Flush` メソッド)」を用意すればよい。`@Flush` メソッドの詳細は、前述の「更新結果の判定」を参照されたい。

Repository のメソッドの呼び出し順番

バッチモードを使用する目的は更新処理の性能向上であるが、Repository のメソッドの呼び出し順番を間違えると、性能向上につながらないケースがある。

バッチモードを使用して性能向上させるためには、以下の MyBatis の仕様を理解しておく必要がある。

- クエリ (SELECT) を実行すると、それまでキューイングされていた SQL がバッチ実行される。
- 連続して呼び出された更新処理 (Repository のメソッド) 毎に `PreparedStatement` が生成され、SQL をキューイングする。

これは、以下の様な実装をすると、バッチモードを利用するメリットがない事を意味している。

- 例 1

```
@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    @Inject
    @Named("todoBatchRepository")
    TodoRepository todoBatchRepository;

    @Override
    public void storeTodos(List<Todo> todos) {
        for (Todo todo : todos) {
            // (1)
            Todo currentTodo = todoBatchRepository.findOne(todo.getTodoId());
            if (currentTodo == null) {
                todoBatchRepository.create(todo);
            } else{
                todoBatchRepository.update(todo);
```

```
            }
        }
    }

}
```

項番	説明
1.	上記例のように実装した場合、繰り返し処理の先頭にクエリを発行しているため、1 件毎に SQL がバッチ実行される事になってしまう。これはほぼ、シンプルモード (SIMPLE) で実行しているのと同義である。 上記のような処理が必要な場合は、PreparedStatement 再利用モード (REUSE) の Repository を使用した方が効率的である。

- 例 2

```
@Transactional
@Service
public class TodoServiceImpl implements TodoService {

    @Inject
    @Named("todoBatchRepository")
    TodoRepository todoBatchRepository;

    @Override
    public void storeTodos(List<Todo> todos) {
        for (Todo todo : todos) {
            // (2)
            todoBatchRepository.create(todo);
            todoBatchRepository.createHistory(todo);
        }
    }

}
```

項番	説明
2.	上記のような処理が必要な場合は、Repository のメソッドが交互に呼び出されているため、1 件毎に PreparedStatement が生成されてしまう。これはほぼ、シンプルモード (SIMPLE) で実行しているのと同義である。 上記のような処理が必要な場合は、PreparedStatement 再利用モード (REUSE) の Repository を使用した方が効率的である。

ストアドプロシージャの実装

データベースに登録されているストアドプロシージャやファンクションを、MyBatis3 から呼び出す方法について説明を行う。

以下で説明する実装例では、PostgreSQL に登録されているファンクションを呼び出している。

- ストアドプロシージャ（ファンクション）を登録する。

```
/* (1) */
CREATE FUNCTION findTodo(pTodoId CHAR)
RETURNS TABLE(
    todo_id CHAR,
    todo_title VARCHAR,
    finished BOOLEAN,
    created_at TIMESTAMP,
    version BIGINT
) AS $$ BEGIN RETURN QUERY
SELECT
    t.todo_id,
    t.todo_title,
    t.finished,
    t.created_at,
    t.version
FROM
    t_todo t
WHERE
    t.todo_id = pTodoId;
END;
$$ LANGUAGE plpgsql;
```

項番	説明
1.	このファンクションは、指定された ID のレコードを取得するファンクションである。

- Repository インタフェースにメソッドを定義する。

```
// (2)
public interface TodoRepository extends Repository {
    Todo findOne(String todoId);
}
```

項番	説明
2.	SQL を発行する際と同じインタフェースでよい。

- マッピングファイルにストアドプロシージャの呼び出し処理を実装する。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <!-- (3) -->
    <select id="findOne" parameterType="string" resultType="Todo"
            statementType="CALLABLE">
        <!-- (4) -->
        {call findTodo(#{todoId})}
    </select>

</mapper>
```

項番	説明
3.	ストアドプロシージャを呼び出すステートメントを実装する。 ストアドプロシージャを呼び出す場合は、statementType 属性に CALLABLE を指定する。CALLABLE を指定すると、java.sql.CallableStatement を使用してストアドプロシージャが呼び出される。 OUT パラメータを JavaBean にマッピングするために、resultType 属性又は resultMap 属性を指定する。
4.	ストアドプロシージャを呼び出す。 ストアドプロシージャ（ファンクション）を呼び出す場合は、 • {call Procedure or Function 名 (IN パラメータ...)} 形式で指定する。 上記例では、findTodo という名前のファンクションに対して、IN パラメータに ID を指定して呼び出している。

6.2.4 Appendix

Mapper インタフェースの仕組みについて

Mapper インタフェースを使用する場合、開発者は Mapper インタフェースとマッピングファイルを作成するだけで、SQL を実行する事ができる。

Mapper インタフェースの実装クラスは、MyBatis3 が JDK の Proxy 機能を使用してアプリケーション実行時に生成されるため、開発者が Mapper インタフェースの実装クラスを作成する必要はない。

Mapper インタフェースは、MyBatis3 から提供されているインタフェースの継承やアノテーションなどの定義は不要であり、単に Java のインタフェースとして作成すればよい。

以下に、Mapper インタフェースとマッピングファイルの作成例、及びアプリケーション (Service) での利用例を示す。

ここでは、開発者が作成する成果物をイメージしてもらう事が目的なので、コードに対する説明はポイントとなる点に絞って行っている。

- Mapper インタフェースの作成例

  本ガイドラインでは、MyBatis3 の Mapper インタフェースを Repository インタフェースとして使用することを前提としているため、インタフェース名は、「Entity 名」 + `Repository` というネーミングにしている。

```
package com.example.domain.repository.todo;

import com.example.domain.model.Todo;

public interface TodoRepository {
    Todo findOne(String todoId);
}
```

- マッピングファイルの作成例

  マッピングファイルでは、ネームスペースとして Mapper インタフェースの FQCN(Fully Qualified Class Name) を指定し、Mapper インタフェースに定義したメソッドの呼び出し時に実行する SQL との紐づけは、各種ステートメントタグ (insert/update/delete/select タグ) の id 属性に、メソッド名を指定する事で行う事ができる。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org/DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.domain.repository.todo.TodoRepository">

    <resultMap id="todoResultMap" type="Todo">
        <result column="todo_id" property="todoId" />
        <result column="title" property="title" />
        <result column="finished" property="finished" />
    </resultMap>

    <select id="findOne" parameterType="String" resultMap="todoResultMap">
      SELECT
        todo_id,
        title,
        finished
      FROM
        t_todo
      WHERE
        todo_id = #{todoId}
    </select>
```

```
</mapper>
```

- アプリケーション (Service) での Mapper インタフェースの使用例

  アプリケーション (Service) から Mapper インタフェースのメソッドを呼び出す場合は、Spring(DI コンテナ) によって注入された Mapper オブジェクトのメソッドを呼び出す。アプリケーション (Service) は、Mapper オブジェクトのメソッドを呼び出すことで、透過的に SQL が実行され、SQL の実行結果を得ることができる。

```
package com.example.domain.service.todo;

import com.example.domain.model.Todo;
import com.example.domain.repository.todo.TodoRepository;

public class TodoServiceImpl implements TodoService {

    @Inject
    TodoRepository todoRepository;

    public Todo getTodo(String todoId){
        Todo todo = todoRepository.findOne(todoId);
        if(todo == null){
            throw new ResourceNotFoundException(
                ResultMessages.error().add("e.ex.td.5001" ,todoId));
        }
        return todo;
    }

}
```

以下に、Mapper インタフェースのメソッドを呼び出した際に、SQL が実行されるまでの処理フローについて説明を行う。

項番	説明
1.	アプリケーションは、Mapper インタフェースに定義されているメソッドを呼び出す。 Mapper インタフェースの実装クラス (Mapper インタフェースの Proxy オブジェクト) は、アプリケーション起動時に MyBatis3 のコンポーネントによって生成される。

Continued on next page

表 6.9 – continued from previous page

項番	説明
2.	Mapper インタフェースの Proxy オブジェクトは、`MapperProxy` の invoke メソッドを呼び出す。 `MapperProxy` は、Mapper インタフェースのメソッド呼び出しをハンドリングする役割をもつ。
3.	`MapperProxy` は、呼び出された Mapper インタフェースのメソッドに対応する `MapperMethod` を生成し、execute メソッドを呼び出す。 `MapperMethod` は、 呼び出された Mapper インタフェースのメソッドに対応する `SqlSession` のメソッドを呼び出す役割をもつ。
4.	`MapperMethod` は、 `SqlSession` のメソッドを呼び出す。 `SqlSession` のメソッドを呼び出す際は、実行する SQL ステートメントを特定するためのキー (以降、「ステートメント ID」と呼ぶ) を引き渡している。
5.	`SqlSession` は、指定されたステートメント ID をキーに、マッピングファイルより SQL ステートメントを取得する。
6.	`SqlSession` は、マッピングファイルより取得した SQL ステートメントに指定されているバインド変数に値を設定し、SQL を実行する。
7.	Mapper インタフェース (`SqlSession`) は、SQL の実行結果を JavaBean などに変換して、アプリケーションに返却する。 件数のカウントや、更新件数などを取得する場合は、プリミティブ型やプリミティブラッパ型などが返却値となるケースもある。

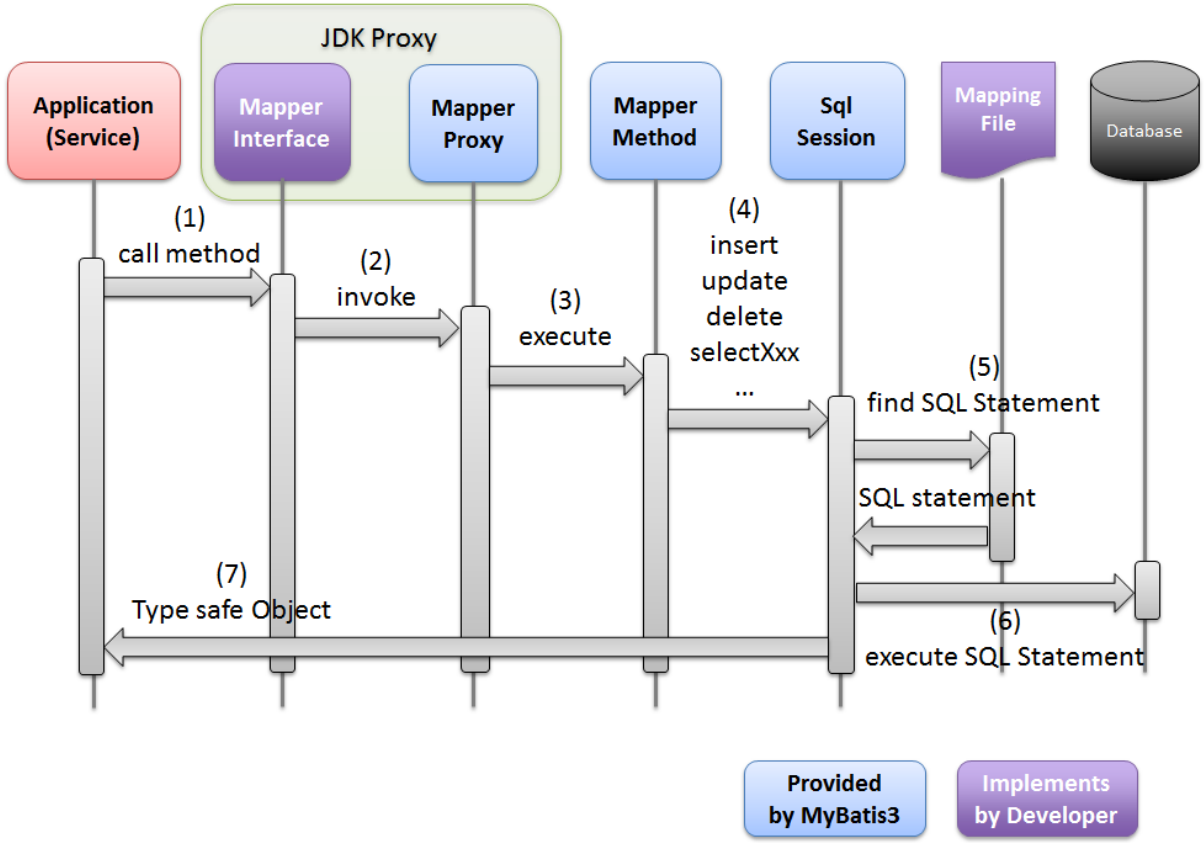


図 6.5　**Picture - Mapper mechanism**

ちなみに：ステートメント **ID** とは

ステートメント ID は、実行する SQL ステートメントを特定するためのキーであり、「**Mapper** インタフェースの **FQCN** + 「.」 + 呼び出された **Mapper** インタフェースのメソッド名」 というルールで生成される。

`MapperMethod` によって生成されたステートメント ID に対応する SQL ステートメントをマッピングファイルに定義するためには、マッピングファイルのネームスペースに「Mapper インタフェースの FQCN」、各種ステートメントタグの id 属性に「Mapper インタフェースのメソッド名」を指定する必要がある。

TypeAlias の設定

TypeAlias の設定は、基本的には `package` 要素を使用してパッケージ単位で設定すればよいが、

- クラス単位でエイリアス名を設定する方法
- デフォルトで付与されるエイリアス名を上書きする方法 (任意のエイリアス名を指定する方法)

も用意されている。

TypeAlias をクラス単位に設定

TypeAlias の設定は、クラス単位で設定する事もできる。

- projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml

```
<typeAliases>
    <!-- (1) -->
    <typeAlias
        type="com.example.domain.repository.account.AccountSearchCriteria" />
    <package name="com.example.domain.model" />
</typeAliases>
```

項番	説明
1.	typeAlias 要素の type 属性に、エイリアスを設定するクラスの完全修飾クラス名 (FQCN) を指定する。 上記例だと、com.example.domain.repository.account.AccountSearchCriteria クラスのエイリアス名は、AccountSearchCriteria (パッケージの部分が除去された部分) となる。 エイリアス名に任意の値を指定したい場合は、typeAlias 要素の alias 属性に任意のエイリアス名を指定することができる。

デフォルトで付与されるエイリアス名の上書き

package 要素を使用してエイリアスを設定した場合や、typeAlias 要素の alias 属性を省略してエイリアスを設定した場合は、TypeAlias のエイリアス名は、完全修飾クラス名 (FQCN) からパッケージの部分が除去された部分となる。

デフォルトで付与されるエイリアス名ではなく、任意のエイリアス名にしたい場合は、TypeAlias を設定したいクラスに@org.apache.ibatis.type.Alias アノテーションを指定する事で、任意のエイリアス名を指定する事ができる。

- エイリアス設定対象の Java クラス

```
package com.example.domain.model.book;

@Alias("BookAuthor") // (1)
public class Author {
    // ...
}
```

```
package com.example.domain.model.article;

@Alias("ArticleAuthor") // (1)
public class Author {
```

```
    // ...
}
```

項番	説明
1.	`@Alias` アノテーションの `value` 属性に、エイリアス名を指定する。 上記例だと、`com.example.domain.model.book.Author` クラスのエイリアス名は、`BookAuthor` となる。 異なるパッケージの中に同じクラス名のクラスが格納されている場合は、この方法を使用することで、それぞれ異なるエイリアス名を設定する事ができる。ただし、本ガイドラインでは、クラス名は重複しないように設計する事を推奨する。上記例であれば、クラス名自体を `BookAuthor` と `ArticleAuthor` にすることを検討して頂きたい。

ちなみに: TypeAlias の エイリアス名は、

- `typeAlias` 要素の `alias` 属性の指定値
- `@Alias` アノテーションの `value` 属性の指定値
- デフォルトで付与されるエイリアス名 (完全修飾クラス名からパッケージの部分が除去された部分)

の優先順で適用される。

データベースによる SQL 切り替えについて

MyBatis3 では、JDBC ドライバから接続しているデータベースのベンダー情報を取得して、使用する SQL を切り替える仕組み (`org.apache.ibatis.mapping.VendorDatabaseIdProvider`) を提供している。

この仕組みは、動作環境として複数のデータベースをサポートするようなアプリケーションを構築する際に有効である。

注釈: 本ガイドラインでは、環境依存するコンポーネントや設定ファイルについては、[projectName]-env というサブプロジェクトで管理し、ビルド時に実行環境にあったコンポーネントや設定ファイル作成を選択するスタイルを推奨している。

[projectName]-env は、

- 開発環境 (ローカルの PC 環境)
- 各種試験環境

- 商用環境

上記それぞれの差分を吸収するためのサブプロジェクトであり、複数のデータベースをサポートするアプリケーションの開発でも利用する事ができる。

基本的には、環境依存するコンポーネントや設定ファイルは、[projectName]-env というサブプロジェクトで管理する事を推奨するが、SQL のちょっとした違いを吸収したい場合は、本仕組みを使用してもよい。

アーキテクトは、データベースの違いによる **SQL** の環境依存をどのように実装するかの指針を明確に示すことで、アプリケーション全体として統一された実装となるように心がけてほしい。

- `projectName-domain/src/main/resources/META-INF/spring/projectName-infra.xml` に Bean 定義を追加する。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://mybatis.org/schema/mybatis-spring
        http://mybatis.org/schema/mybatis-spring.xsd
    ">

    <import resource="classpath:/META-INF/spring/projectName-env.xml" />

    <!-- (1) -->
    <bean id="databaseIdProvider"
          class="org.apache.ibatis.mapping.VendorDatabaseIdProvider">
        <!-- (2) -->
        <property name="properties">
            <props>
                <prop key="H2">h2</prop>
                <prop key="PostgreSQL">postgresql</prop>
            </props>
        </property>
    </bean>

    <bean id="sqlSessionFactory"
        class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <!-- (3) -->
        <property name="databaseIdProvider" ref="databaseIdProvider"/>
        <property name="configLocation"
            value="classpath:/META-INF/mybatis/mybatis-config.xml" />
```

```
    </bean>

    <mybatis:scan base-package="com.example.domain.repository" />

</beans>
```

項番	説明
1.	MyBatis3 から提供されている VendorDatabaseIdProvider を Bean 定義する。 VendorDatabaseIdProvider は、JDBC ドライバから取得したデータベースのプロダクト名 (java.sql.DatabaseMetaData#getDatabaseProductName()) をデータベース ID として扱うためのクラスである。
2.	properties プロパティには、JDBC ドライバから取得したデータベースのプロダクト名とデータベース ID のマッピングを指定する。 マッピング仕様については、「MyBatis3 REFERENCE DOCUMENTATION(Configuration-databaseIdProvider-)」を参照されたい。
3.	データベース ID を使用する SqlSessionFactoryBean の databaseIdProvider プロパティ対して、(1) で定義した DatabaseIdProvider を指定する。 この指定を行うと、マッピングファイルからデータベース ID を参照する事が可能となる。

注釈: 本ガイドラインでは、properties プロパティを指定して、データベースのプロダクト名とデータベース ID をマッピングする方式を推奨する。

理由は、JDBC ドライバから取得できるデータベースのプロダクト名は、JDBC ドライバのバージョンによって変わる可能性があるためである。properties プロパティを使用すると、使用する JDBC ドライバのバージョンによるプロダクト名の違いを、一箇所で管理する事ができる。

- マッピングファイルの実装を行う。

```
<insert id="create" parameterType="Todo">
    <!-- (1) -->
    <selectKey keyProperty="todoId" resultType="string" order="BEFORE"
               databaseId="h2">
        SELECT RANDOM_UUID()
    </selectKey>
    <selectKey keyProperty="todoId" resultType="string" order="BEFORE"
               databaseId="postgresql">
```

```
        SELECT UUID_GENERATE_V4()
    </selectKey>

    INSERT INTO
      t_todo
    (
        todo_id
        ,todo_title
        ,finished
        ,created_at
        ,version
    )
    VALUES
    (
        #{todoId}
        ,#{todoTitle}
        ,#{finished}
        ,#{createdAt}
        ,#{version}
    )
</insert>
```

項番	説明
1.	ステートメント要素 (select 要素、update 要素、sql 要素など) をデータベース毎に切り替えたい場合は、各要素の databaseId 属性にデータベース ID を指定する。 databaseId 属性を指定すると、データベース ID が一致するステートメント要素が使用される。 上記例では、データベース固有の UUID 生成関数を呼び出して、ID を生成している。

ちなみに: 上記例では、PostgreSQL の UUID 生成関数として UUID_GENERATE_V4() を呼び出しているが、この関数は、uuid-ossp と呼ばれるサブモジュールの関数である。

この関数を使用したい場合は、uuid-ossp モジュールを有効にする必要がある。

ちなみに: データベース ID は、OGNL ベースの式（Expression 言語）内でも参照する事ができる。

これは、データベース ID を動的 SQL の条件として使用できる事を意味している。以下に実装例を紹介する。

```
<select id="findAllByCreatedAtBefore" parameterType="_int" resultType="Todo
↪">
    SELECT
        todo_id,
        todo_title,
```

```
        finished,
        created_at,
        version
    FROM
        t_todo
    WHERE
        <choose>
            <!-- (2) -->
            <when test="_databaseId == 'h2'">
                <bind name="criteriaDate"
                      value="'DATEADD(\'DAY\',#{days} * -1,#{currentDate})'
↪"/>
            </when>
            <when test="_databaseId == 'postgresql'">
                <bind name="criteriaDate"
                      value="'#{currentDate}::DATE - (#{days} * INTERVAL \
↪'1 DAY\')'"/>
            </when>
        </choose>
        <![CDATA[
            created_at < ${criteriaDate}
        ]]>
</select>
```

項番	説明
2.	OGNL ベースの式（Expression 言語）内では、_databaseId という特別な変数にデータベース ID が格納されている。 上記例では、「システム日付 - 指定日」より前に作成されたレコードを抽出するための条件を、データベースの関数を利用して指定している。

関連 Entity を 1 回の SQL で取得する方法について

主 Entity と関連 Entity を 1 回の SQL でまとめて取得する方法について説明する。

主 Entity と関連 Entity をまとめて取得する仕組みを使用すると、Service クラスで Entity(JavaBean) の組み立て処理を行う必要がなくなり、Service クラスは業務ロジック (ビジネスルール) の実装に集中する事ができる。

また、この方法は、N+1 問題を回避する手段としても使用される。

N+1 問題については、「*N+1 問題の対策方法*」を参照されたい。

警告: 主 Entity と関連 Entity をまとめて取得する場合は、以下の点に注意して使用すること。

- 以下の説明では全ての関連 Entity を 1 回の SQL でまとめて取得しているが、実際のプロジェクトで使用する場合は、処理で必要となる関連 Entity のみ取得するようにした方がよいケースがある。使用しない関連 Entity を同時に取得すると、無駄なオブジェクト生成やマッピング処理が行われるため性能劣化の要因となる事がある。**特に、一覧検索を行う SQL では、必要な関連 Entity のみ取得するようにした方がよいケースが多い。**
- 使用頻度の低い関連 Entity については、まとめて取得せず必要なときに個別に取得する方法を採用した方がよいケースがある。使用頻度の低い関連 Entity を同時に取得すると、無駄なオブジェクト生成やマッピング処理が行われるため性能劣化の要因となる事がある。
- 1:N の関係となる関連 Entity が複数含まれる場合、主 Entity と関連 Entity を別々に取得する方法を採用した方がよいケースがある。1:N の関係となる関連 Entity が複数ある場合、無駄なデータを DB から取得する必要があるため、性能劣化の要因となる事がある。主 Entity と関連 Entity を別々に取得する方法の一例については、「*N+1 問題の対策方法*」を参照されたい。

ちなみに: 使用頻度の低い関連 Entity を必要になった時に個別に取得する方法としては、

- Service クラスの処理で関連 Entity を取得するメソッド (SQL) を呼び出して取得する。
- 関連 Entity を」Lazy Load」対象にし、Getter メソッドが呼び出された際に SQL を透過的に実行して取得する。

方法がある。

「Lazy Load」の仕組みを使用すると、Service クラスで Entity(JavaBean) の組み立て処理を行う必要がなくなり、Service クラスは業務ロジック (ビジネスルール) の実装に集中する事ができる。

一覧検索を行う SQL で」Lazy Load」を使用すると N+1 問題を引き起こすので、使用する際は注意すること。

「Lazy Load」の使用方法については、「*関連 Entity を Lazy Load するための設定*」を参照されたい。

ここからは、ショッピングサイトで扱う注文データを、1 回の SQL でまとめて取得し、主 Entity 及び関連 Entity にマッピングする実装例について説明を行う。

ここで説明する実装方法は、あくまで一例である。MyBatis3 では、本節で説明していない機能も多く提供しており、より高度なマッピングを行う事も可能である。

MyBatis3 のマッピング機能の詳細については、「MyBatis3 REFERENCE DOCUMENTATION(Mapper XML Files-Result Maps-)」を参照されたい。

テーブルレイアウトとデータ

説明で使用するテーブルは、以下の通り。

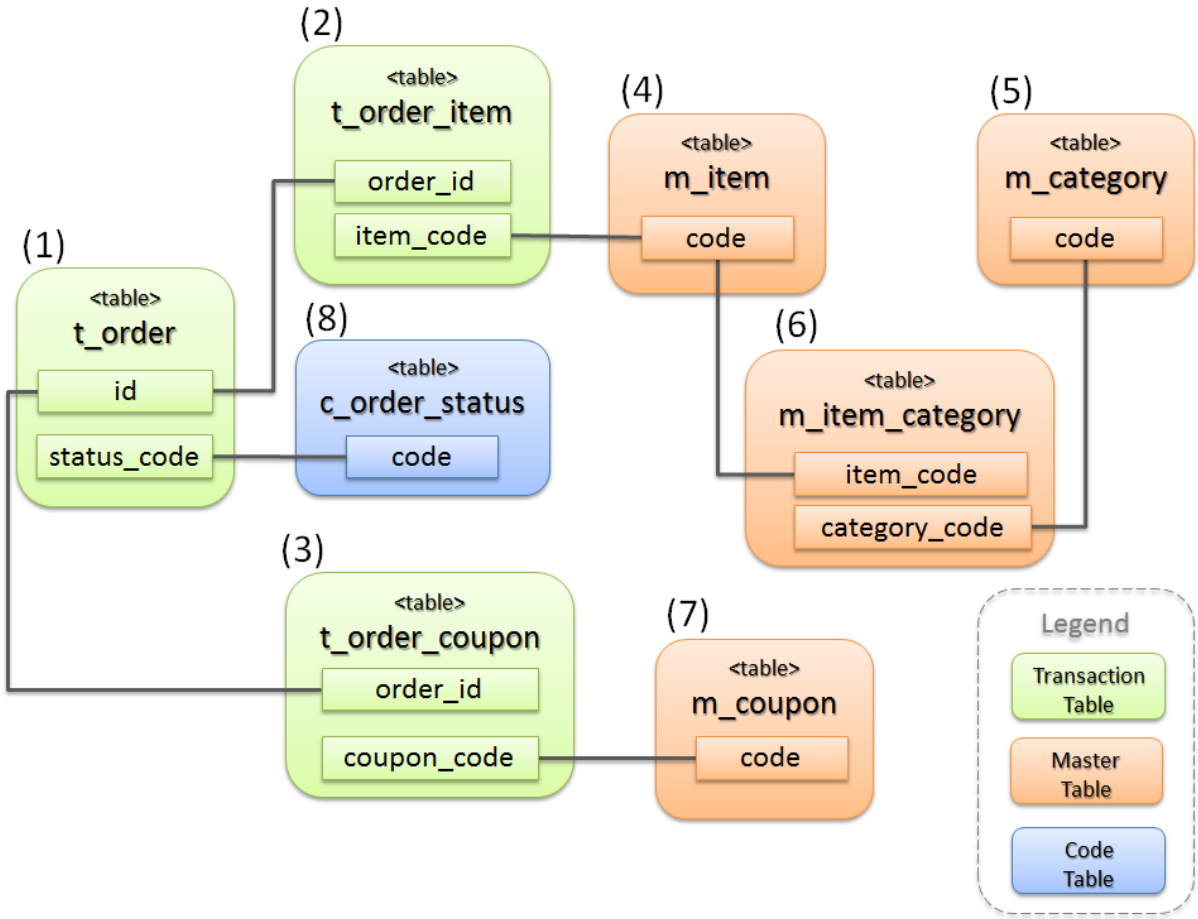


図 6.6 **Picture - ER diagram**

項番	カテゴリ	テーブル名	説明
1.	トランザクション系	t_order	注文データを保持するテーブル。 1 つの注文に対して、1 レコードが格納される。
2.		t_order_item	1 つの注文で購入された商品データを保持するテーブル。 1 つの注文で複数の商品が購入された場合は、商品数分レコードが格納される。
3.		t_order_coupon	1 つの注文で使用されたクーポンのデータを保持するテーブル。 1 つの注文で、複数のクーポンが使用された場合は、クーポン数分レコードが格納される。クーポンを使用しなかった場合は、レコードは格納されない。
4.	マスタ系	m_item	商品を定義するマスタテーブル。
5.		m_category	商品のカテゴリを定義するマスタテーブル。
6.		m_item_category	商品が所属するカテゴリを定義するマスタテーブル。 商品とカテゴリのマッピングを保持している。1 つの商品は、複数のカテゴリに属すことができるモデルとなっている。
7.		m_coupon	クーポンを定義するマスタテーブル。
8.	コード系	c_order_status	注文ステータスを定義するコードテーブル。

説明で使用するテーブルレイアウトと格納データを作成するための SQL(DDL と DML) を以下に示す。(SQL は H2 Database 用である)

- マスタ系テーブル作成用の DDL

```
CREATE TABLE m_item (
    code CHAR(10),
    name NVARCHAR(256),
    price INTEGER,
    CONSTRAINT m_item_pk PRIMARY KEY(code)
);
```

```
CREATE TABLE m_category (
    code CHAR(10),
    name NVARCHAR(256),
    CONSTRAINT m_category_pk PRIMARY KEY(code)
);

CREATE TABLE m_item_category (
    item_code CHAR(10),
    category_code CHAR(10),
    CONSTRAINT m_item_category_pk PRIMARY KEY(item_code, category_code),
    CONSTRAINT m_item_category_fk1 FOREIGN KEY(item_code) REFERENCES m_
↪item(code),
    CONSTRAINT m_item_category_fk2 FOREIGN KEY(category_code) REFERENCES m_
↪category(code)
);

CREATE TABLE m_coupon (
    code CHAR(10),
    name NVARCHAR(256),
    price INTEGER,
    CONSTRAINT m_coupon_pk PRIMARY KEY(code)
);
```

- コード系テーブル作成用の DDL

```
CREATE TABLE c_order_status (
    code VARCHAR(10),
    name NVARCHAR(256),
    CONSTRAINT c_order_status_pk PRIMARY KEY(code)
);
```

- トランザクション系テーブル作成用の DDL

```
CREATE TABLE t_order (
    id INTEGER,
    status_code VARCHAR(10),
    CONSTRAINT t_order_pk PRIMARY KEY(id),
    CONSTRAINT t_order_fk FOREIGN KEY(status_code) REFERENCES c_order_
↪status(code)
);

CREATE TABLE t_order_item (
    order_id INTEGER,
    item_code CHAR(10),
    quantity INTEGER,
    CONSTRAINT t_order_item_pk PRIMARY KEY(order_id, item_code),
    CONSTRAINT t_order_item_fk1 FOREIGN KEY(order_id) REFERENCES t_order(id),
    CONSTRAINT t_order_item_fk2 FOREIGN KEY(item_code) REFERENCES m_item(code)
);

CREATE TABLE t_order_coupon (
    order_id INTEGER,
```

```
    coupon_code CHAR(10),
    CONSTRAINT t_order_coupon_pk PRIMARY KEY(order_id, coupon_code),
    CONSTRAINT t_order_coupon_fk1 FOREIGN KEY(order_id) REFERENCES t_
↪order(id),
    CONSTRAINT t_order_coupon_fk2 FOREIGN KEY(coupon_code) REFERENCES m_
↪coupon(code)
);
```

- データ投入用の DML

```
-- Setup master tables
INSERT INTO m_item VALUES ('ITM0000001','Orange juice',100);
INSERT INTO m_item VALUES ('ITM0000002','NotePC',100000);

INSERT INTO m_category VALUES ('CTG0000001','Drink');
INSERT INTO m_category VALUES ('CTG0000002','PC');
INSERT INTO m_category VALUES ('CTG0000003','Hot selling');

INSERT INTO m_item_category VALUES ('ITM0000001','CTG0000001');
INSERT INTO m_item_category VALUES ('ITM0000002','CTG0000002');
INSERT INTO m_item_category VALUES ('ITM0000002','CTG0000003');

INSERT INTO m_coupon VALUES ('CPN0000001','Join coupon',3000);
INSERT INTO m_coupon VALUES ('CPN0000002','PC coupon',30000);

-- Setup code tables
INSERT  INTO  c_order_status VALUES ('accepted','Order accepted');
INSERT  INTO  c_order_status VALUES ('checking','Stock checking');
INSERT  INTO  c_order_status VALUES ('shipped','Item Shipped');

-- Setup transaction tables
INSERT INTO t_order VALUES (1,'accepted');
INSERT INTO t_order VALUES (2,'checking');

INSERT INTO t_order_item VALUES (1,'ITM0000001',1);
INSERT INTO t_order_item VALUES (1,'ITM0000002',2);
INSERT INTO t_order_item VALUES (2,'ITM0000001',3);
INSERT INTO t_order_item VALUES (2,'ITM0000002',4);

INSERT INTO t_order_coupon VALUES (1,'CPN0000001');
INSERT INTO t_order_coupon VALUES (1,'CPN0000002');

COMMIT;
```

Entity のクラス図

実装例では、上記テーブルに格納されているレコードを、以下の Entity(JavaBean) にマッピングする。

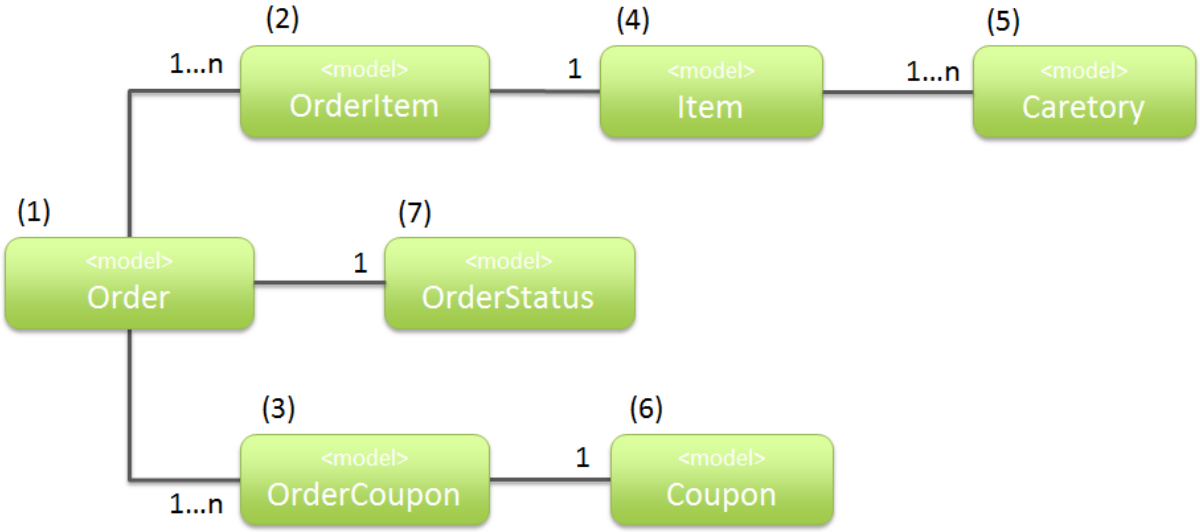


図 6.7 **Picture - Class(JavaBean) diagram**

項番	クラス名	説明
1.	Order	t_order テーブルの 1 レコードを表現する JavaBean。 関連 Entity として、`OrderStatus` を 1 件、`OrderItem` および `OrderCoupon` を複数保持する。 `public class Order implements Serializable { private static final long serialVersionUID = 1L; private int id; private OrderStatus orderStatus; List<OrderItem> orderItems; List<OrderCoupon> orderCoupons; // ... }`
2.	OrderItem	t_order_item テーブルの 1 レコードを表現する JavaBean。 関連 Entity として、`Item` を保持する。 `public class OrderItem implements Serializable { private static final long serialVersionUID = 1L; private int orderId; private Item item; private int quantity; // ... }`

Continued on next page

表 6.10 – continued from previous page

項番	クラス名	説明
3.	OrderCoupon	t_order_coupon テーブルの 1 コードを表現する JavaBean。 関連 Entity として、`Coupon` を保持する。 `public class OrderCoupon implements Serializable { private static final long serialVersionUID = 1L; private int orderId; private Coupon coupon; // ... }`
4.	Item	m_item テーブルの 1 コードを表現する JavaBean。 関連オブジェクトとして、所属している `Category` を複数保持する。`Category` との紐づけは、m_item_category テーブルによって行われる。 `public class Item implements Serializable { private static final long serialVersionUID = 1L; private String code; private String name; private int price; private List<Category> categories; // ... }`
5.	Category	m_category テーブルの 1 レコードを表現する JavaBean。 `public class Category implements Serializable { private static final long serialVersionUID = 1L; private String code; private String name; // ... }`
6.	Coupon	m_coupon テーブルの 1 レコードを表現する JavaBean。 `public class Coupon implements Serializable { private static final long serialVersionUID = 1L; private String code; private String name; private int price; // ... }`

Continued on next page

表 6.10 – continued from previous page

項番	クラス名	説明

7.	OrderStatus	c_order_status テーブルの 1 レコードを表現する JavaBean。
		`public class OrderStatus implements Serializable {`
		`    private static final long serialVersionUID = 1L;`
		`    private String code;`
		`    private String name;`
		`    // ...`
		`}`

Repository インタフェースの実装

実装例では、

- Order オブジェクトを 1 件取得するメソッド (`findOne`)
- 該当ページの Order オブジェクトを取得するメソッド (`findPage`)

を実装する。

```
package com.example.domain.repository.order;

import com.example.domain.model.Order;

import java.util.List;

public interface OrderRepository {

    Order findOne(int id);

    List<Order> findPage(@Param("pageable") Pageable pageable);

}
```

SQL の実装

関連 Entity を 1 回の SQL でまとめて取得する場合は、取得対象のテーブルを JOIN してマッピングに必要な全てのレコードを取得する。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.example.domain.repository.order.OrderRepository">

    <!-- (1) -->
    <sql id="selectFromJoin">
        SELECT
            /* (2) */
            o.id,
            /* (3) */
            o.status_code,
            os.name AS status_name,
            /* (4) */
            oi.quantity,
```

```
            i.code AS item_code,
            i.name AS item_name,
            i.price AS item_price,
            /* (5) */
            ct.code AS category_code,
            ct.name AS category_name,
            /* (6) */
            cp.code AS coupon_code,
            cp.name AS coupon_name,
            cp.price AS coupon_price
        FROM
            ${orderTable} o
        /* (7) */
        INNER JOIN c_order_status os ON os.code = o.status_code
        INNER JOIN t_order_item oi ON oi.order_id = o.id
        INNER JOIN m_item i ON i.code = oi.item_code
        INNER JOIN m_item_category ic ON ic.item_code = i.code
        INNER JOIN m_category ct ON ct.code = ic.category_code
        /* (8) */
        LEFT JOIN t_order_coupon oc ON oc.order_id = o.id
        LEFT JOIN m_coupon cp ON cp.code = oc.coupon_code
    </sql>

    <!-- (9) -->
    <select id="findOne" parameterType="_int" resultMap="orderResultMap">
        <bind name="orderTable" value="'t_order'" />
        <include refid="selectFromJoin"/>
        WHERE
            o.id = #{id}
        ORDER BY
            item_code ASC,
            category_code ASC,
            coupon_code ASC
    </select>

    <!-- (10) -->
    <select id="findPage" resultMap="orderResultMap">
        <bind name="orderTable" value="
            '(
              SELECT
                  *
              FROM
                  t_order
              ORDER BY
                  id DESC
              LIMIT #{pageable.pageSize}
              OFFSET #{pageable.offset}
              )'" />
        <include refid="selectFromJoin"/>
        ORDER BY
            id DESC,
            item_code ASC,
            category_code ASC,
```

```
            coupon_code ASC
    </select>

    <!-- ... -->

</mapper>
```

項番	説明
1.	`findOne` メソッドと `findPage` メソッド用の SELECT 句、FROM 句、JOIN 句を実装する。 上記例では、`findOne` メソッドと `findPage` メソッドの共通箇所を共通化している。
2.	Order オブジェクトを生成するために必要なデータを取得する。
3.	OrderStatus オブジェクトを生成するために必要なデータを取得する。 取得するカラム名は重複しないようにする必要がある。上記例では、`name` カラムが重複するため、AS 句を使用して別名 (`status_`プレフィックス) を指定している。
4.	OrderItem オブジェクトと Item オブジェクトを生成するために必要なデータを取得する。 取得するカラム名は重複しないようにする必要がある。上記例では、`code`,`name`, `price` が重複するため、AS 句を使用して別名 (`item_`プレフィックス) を指定している。
5.	Category オブジェクトを生成するために必要なデータを取得する。 取得するカラム名は重複しないようにする必要がある。上記例では、`code`,`name` が重複するため、AS 句を使用して別名 (`category_`プレフィックス) を指定している。
6.	OrderCoupon オブジェクトと Coupon オブジェクトを生成するために必要なデータを取得する。 取得するカラム名は重複しないようにする必要がある。上記例では、`code`,`name`, `price` が重複するため、AS 句を使用して別名 (`coupon_`プレフィックス) を指定している。
7.	関連オブジェクトを生成するために必要なデータが格納されているテーブルを結合する。
8.	レコードが格納されない可能性のあるテーブルについては、外部結合とする。クーポンを使用しない場合、t_order_coupon にレコードが格納されないので外部結合にする必要がある。t_order_coupon と結合する t_coupon も同様である。
9.	`findOne` メソッド用の SQL を実装する。 ORDER BY 句には、1:N の関連をもつ Entity の並び順を指定する。上記例では、PK の昇順で並べ替えている。
10.	`findPage` メソッド用の SQL を実装する。ORDER BY 句には、Order と 1:N の関連をもつ Entity の並び順を指定する。上記例では、Order は PK の降順 (新しい順)、関連 Entity は PK の昇順で並べ替えている。

ちなみに： 1:N の関連を持つ関連 Entity を 1 回の SQL でまとめて取得する際にページネーション検索が必要な場合は、MyBatis3 から提供されている `RowBounds` を使用することが出来ない。

代替案としては、

- まず主 Entity のみを検索するメソッドを呼び出し、関連 Entity は別途のメソッドを呼び出して取得する
- SQL でページ範囲内の主 Entity のみ格納されている仮想テーブルを作成し、仮想テーブルのレコードと JOIN する事で、マッピングに必要な全てのレコードを取得する (上記例の `findPage` は、このパターンで実装している)

等の方法が考えられる。

上記 SQL(findPage) を実行すると以下のレコードが取得される。注文レコードとしては 2 件だが、レコードが複数件格納される関連テーブルと結合しているため、合計で 9 レコードが取得される。

内訳は、

- 1〜3 行目は、注文 ID が」`2`」の `Order` オブジェクトを生成するためのレコード
- 4〜9 行目は、注文 ID が」`1`」の `Order` オブジェクトを生成するためレコード

となる。

以降の説明では、注文 ID が」`1`」のレコードを例に、どのように検索結果 (`ResultSet`) を JavaBean にマッピングするかを説明していく。

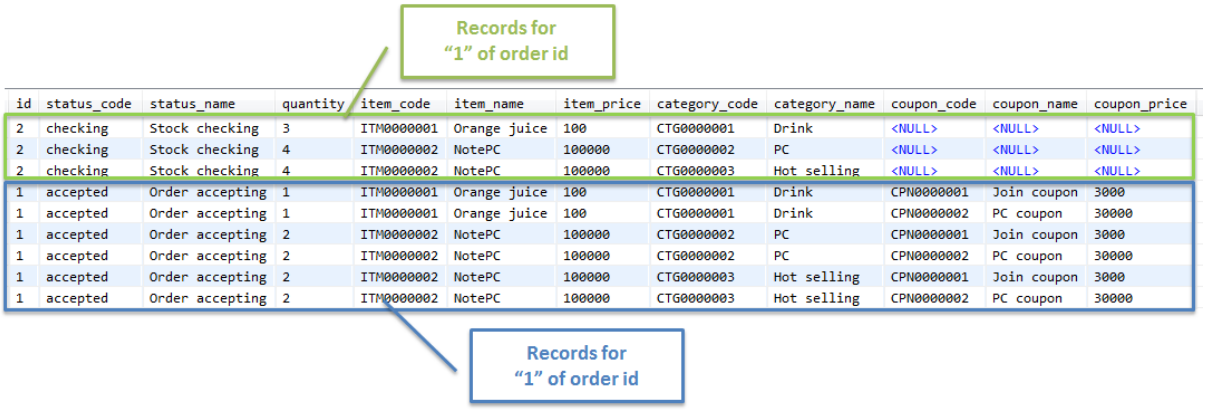


図 6.8 **Picture - Result Set of findPage**

マッピングの実装

上記レコードを、Order オブジェクトと関連 Entity にマッピングするための定義を以下に示す。

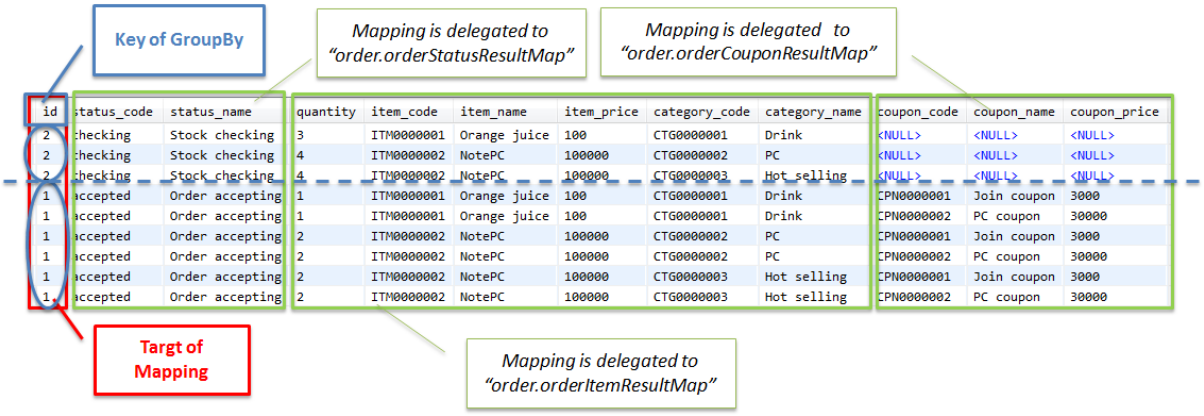
```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.example.domain.repository.order.OrderRepository">

    <!-- ... -->

    <!-- (1) -->
    <resultMap id="orderResultMap" type="Order">
        <id property="id" column="id"/>
        <!-- (2) -->
        <result property="orderStatus.code" column="status_code" />
        <result property="orderStatus.name" column="status_name" />
        <!-- (3) -->
        <collection property="orderItems" ofType="OrderItem">
            <id property="orderId" column="id"/>
            <id property="item.code" column="item_code"/>
            <result property="quantity" column="quantity"/>
            <association property="item" resultMap="itemResultMap"/>
        </collection>
        <!-- (4) -->
        <collection property="orderCoupons" ofType="OrderCoupon"
                    notNullColumn="coupon_code">
            <id property="orderId" column="id"/>
            <!-- (5) -->
            <id property="coupon.code" column="coupon_code"/>
            <result property="coupon.name" column="coupon_name"/>
            <result property="coupon.price" column="coupon_price"/>
        </collection>
    </resultMap>

    <!-- (6) -->
    <resultMap id="itemResultMap" type="Item">
        <id property="code" column="item_code"/>
        <result property="name" column="item_name"/>
        <result property="price" column="item_price"/>
        <!-- (7) -->
        <collection property="categories" ofType="Category">
            <id property="code" column="category_code"/>
            <result property="name" column="category_name"/>
        </collection>
    </resultMap>

</mapper>
```

項番	説明
1.	取得したレコードを Order オブジェクトにマッピングするための定義。関連 Entity(OrderStatus, OrderItem,OrderCoupon) のマッピングを行う。
2.	取得したレコードを OrderStatus オブジェクトにマッピングするための定義。
3.	取得したレコードを OrderItem オブジェクトにマッピングするための定義。関連 Entity(Item) へのマッピングは、別の resultMap(6) に委譲している。
4.	取得したレコードを OrderCoupon オブジェクトにマッピングするための定義。
5.	取得したレコードを Coupon オブジェクトにマッピングするための定義。
6.	取得したレコードを Item オブジェクトにマッピングするための定義。
7.	取得したレコードを Category オブジェクトにマッピングするための定義。

Order オブジェクトへのマッピングの実装

Order オブジェクトへのマッピングを行う。

```
<!-- (1) -->
<resultMap id="orderResultMap" type="Order">
    <!-- (2) -->
    <id property="id" column="id"/>
    <result property="orderStatus.code" column="status_code" />
    <result property="orderStatus.name" column="status_name" />
    <collection property="orderItems" ofType="OrderItem">
        <id property="orderId" column="id"/>
        <id property="item.code" column="item_code"/>
        <result property="quantity" column="quantity"/>
        <association property="item" resultMap="itemResultMap"/>
    </collection>
    <collection property="orderCoupons" ofType="OrderCoupon"
                notNullColumn="coupon_code">
        <id property="orderId" column="id"/>
        <id property="coupon.code" column="coupon_code"/>
        <result property="coupon.name" column="coupon_name"/>
        <result property="coupon.price" column="coupon_price"/>
    </collection>
</resultMap>
```



id	status_code	status_name	quantity	item_code	item_name	item_price	category_code	category_name	coupon_code	coupon_name	coupon_price
2	checking	Stock checking	3	ITM0000001	Orange juice	100	CTG0000001	Drink	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000002	PC	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000003	Hot selling	<NULL>	<NULL>	<NULL>
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000001	Join coupon	3000
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000002	PC coupon	30000

図 6.9 **Picture - ResultMap for Order**

項番	説明
1.	検索結果を `Order` オブジェクトにマッピングする。 `type` 属性にマッピングするクラスを指定する。
2.	取得したレコードの `id` カラムの値を、`Order#id` プロパティに設定する。 `id` カラムは PK なので、`id` 要素を使用してマッピングを行う。`id` 要素を使用すると、指定したプロパティの値でレコードがグループ化される。具体的には、`id=1` と `id=2` の 2 つにグループ化され、2 つの `Order` オブジェクトが生成される。

OrderStatus オブジェクトへのマッピングの実装

`OrderStatus` オブジェクトへのマッピングを行う。

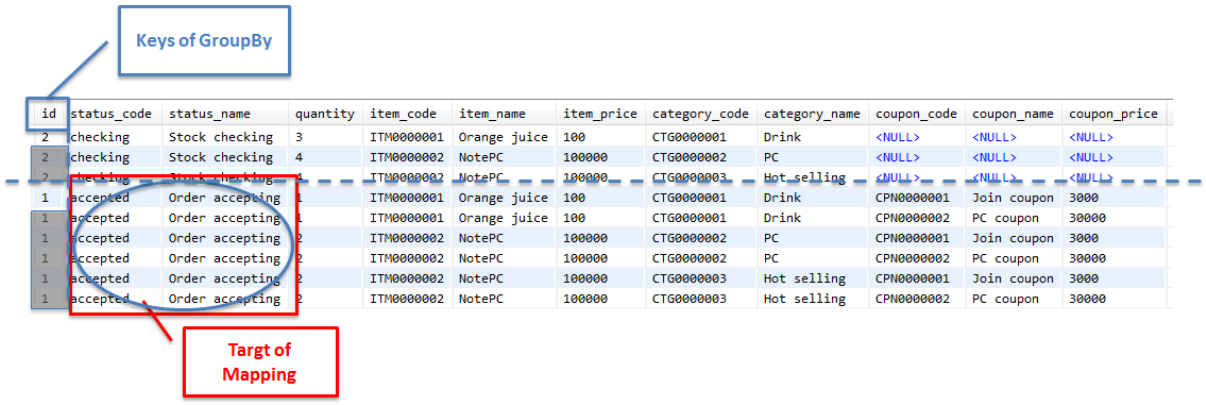
注釈: 「*Entity の実装*」の Entity クラスの作成方針では、「コード系テーブルは、Entity として扱うのではなく、java.lang.String などの基本型で扱う。」としている。これは、コード系テーブルで保持しているデータは、「コードリスト」などの別の仕組みを使用するケースが多いためである。

本節では、関連 Entity(JavaBean) へのマッピング方法を説明する事が目的なので、コード系テーブルも Entity として扱っている点を補足しておく。

実際のプロジェクトでは、Entity クラスの作成方針を参考に Entity を作成することを推奨する。

```
<resultMap id="orderResultMap" type="Order">
    <id property="id" column="id"/>
    <!-- (1) -->
    <result property="orderStatus.code" column="status_code" />
    <!-- (2) -->
```

```
    <result property="orderStatus.name" column="status_name" />
    <collection property="orderItems" ofType="OrderItem">
        <id property="orderId" column="id"/>
        <id property="item.code" column="item_code"/>
        <result property="quantity" column="quantity"/>
        <association property="item" resultMap="itemResultMap"/>
    </collection>
    <collection property="orderCoupons" ofType="OrderCoupon"
                notNullColumn="coupon_code">
        <id property="orderId" column="id"/>
        <id property="coupon.code" column="coupon_code"/>
        <result property="coupon.name" column="coupon_name"/>
        <result property="coupon.price" column="coupon_price"/>
    </collection>
</resultMap>
```



id	status_code	status_name	quantity	item_code	item_name	item_price	category_code	category_name	coupon_code	coupon_name	coupon_price
2	checking	Stock checking	3	ITM0000001	Orange juice	100	CTG0000001	Drink	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000002	PC	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	1	ITM0000002	NotePC	100000	CTG0000003	Hot selling	<NULL>	<NULL>	<NULL>
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000001	Join coupon	3000
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000002	PC coupon	30000

図 6.10 **Picture - ResultMap for OrderStatus**

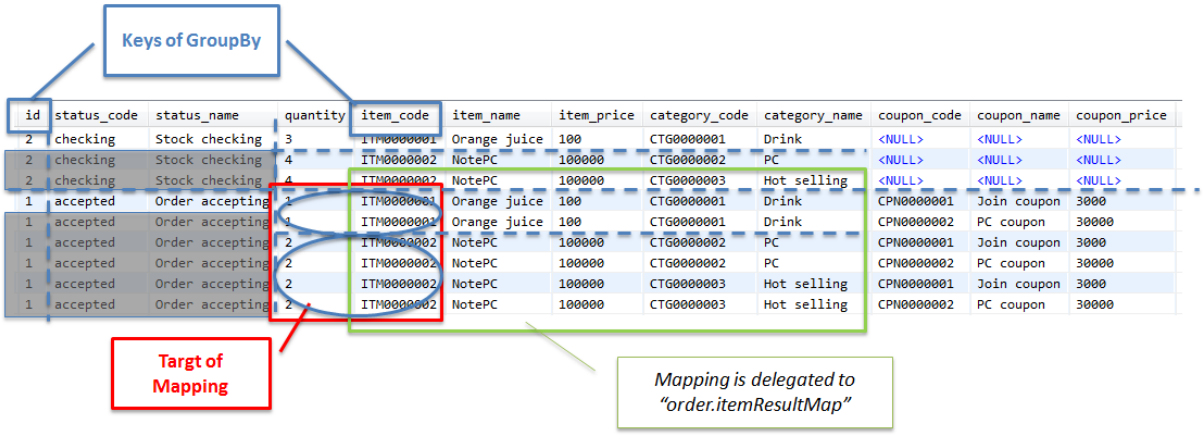
項番	説明
1.	取得したレコードの `status_code` カラムの値を、`OrderStatus#code` プロパティに設定する。
2.	取得したレコードの `status_name` カラムの値を、`OrderStatus#name` プロパティに設定する。

注釈: `OrderStatus` オブジェクトには、`id` カラムでグループ化されたレコードの値が設定される。

OrderItem オブジェクトへのマッピングの実装

`OrderItem` オブジェクトへのマッピングを行う。

```
<resultMap id="orderResultMap" type="Order">
    <id property="id" column="id"/>
    <result property="orderStatus.code" column="status_code" />
    <result property="orderStatus.name" column="status_name" />
    <!-- (1) -->
    <collection property="orderItems" ofType="OrderItem">
        <!-- (2) -->
        <id property="orderId" column="id"/>
        <!-- (3) -->
        <id property="item.code" column="item_code"/>
        <!-- (4) -->
        <result property="quantity" column="quantity"/>
        <!-- (5) -->
        <association property="item" resultMap="itemResultMap"/>
    </collection>
    <collection property="orderCoupons" ofType="OrderCoupon"
                notNullColumn="coupon_code">
        <id property="orderId" column="id"/>
        <id property="coupon.code" column="coupon_code"/>
        <result property="coupon.name" column="coupon_name"/>
        <result property="coupon.price" column="coupon_price"/>
    </collection>
</resultMap>
```



id	status_code	status_name	quantity	item_code	item_name	item_price	category_code	category_name	coupon_code	coupon_name	coupon_price
2	checking	Stock checking	3	ITM0000001	Orange juice	100	CTG0000001	Drink	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000002	PC	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000003	Hot selling	<NULL>	<NULL>	<NULL>
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000001	Join coupon	3000
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000002	PC coupon	30000

図 6.11 **Picture - ResultMap for OrderItem**

項番	説明
1.	検索結果を OrderItem オブジェクトにマッピングし、Order#orderItems プロパティに追加する。 1:N の関係の関連 Entity にマッピングする場合は、collection 要素を使用する。collection 要素の詳細は、「MyBatis3 REFERENCE DOCUMENTATION(Mapper XML Files-collection-)」を参照されたい。
2.	取得したレコードの id カラムの値を、OrderItem#orderId プロパティに設定する。 id カラムは PK なので、id 要素を使用してマッピングを行う。
3.	取得したレコードの item_code カラムの値を、Item#code プロパティに設定する。 item_code カラムは PK なので、id 要素を使用してマッピングを行う。id 要素を使用すると、指定したプロパティの値でレコードがグループ化される。具体的には、Item#code=ITM0000001 と Item#code=ITM0000002 の 2 つにグループ化され、2 つの OrderItem オブジェクトが生成される。
4.	取得したレコードの quantity カラムの値を、OrderItem#quantity プロパティに設定する。
5.	Item オブジェクトの生成を、別の resultMap に委譲し、生成されたオブジェクトを OrderItem#item プロパティに設定する。実際のマッピングは、「*Item オブジェクトへのマッピングの実装*」を参照されたい。 1:1 の関係の関連 Entity にマッピングする場合は、association 要素を使用する。association 要素の詳細は、「MyBatis3 REFERENCE DOCUMENTATION(Mapper XML Files-association-)」を参照されたい。

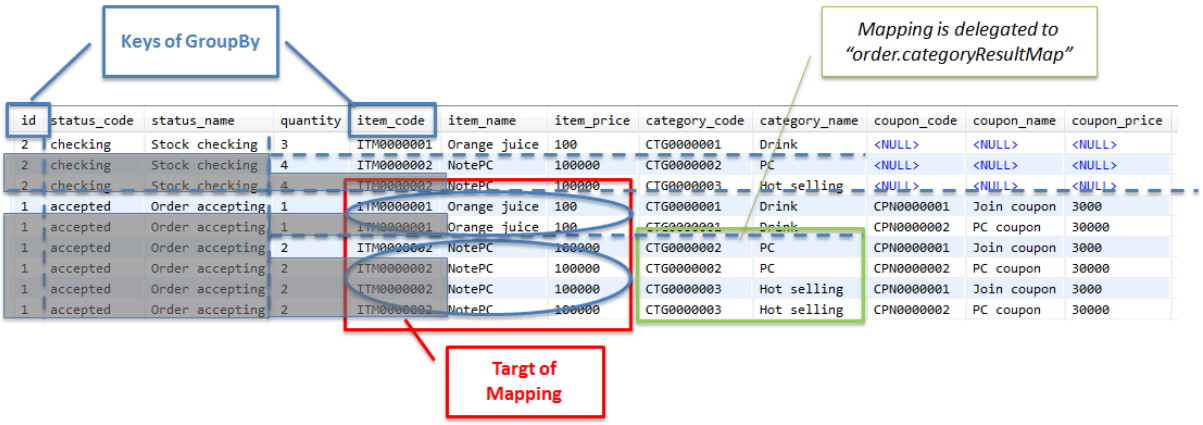
注釈: OrderItem オブジェクトには、id カラムと item_code カラムでグループ化されたレコードの値が設定される。

Item オブジェクトへのマッピングの実装

Item オブジェクトへのマッピングを行う。

```
<!-- (1) -->
<resultMap id="itemResultMap" type="Item">
    <!-- (2) -->
    <id property="code" column="item_code"/>
    <!-- (3) -->
    <result property="name" column="item_name"/>
    <!-- (4) -->
    <result property="price" column="item_price"/>
```

```
    <collection property="categories" ofType="Category">
        <id property="code" column="category_code"/>
        <result property="name" column="category_name"/>
    </collection>
</resultMap>
```



id	status_code	status_name	quantity	item_code	item_name	item_price	category_code	category_name	coupon_code	coupon_name	coupon_price
2	checking	Stock checking	3	ITM0000001	Orange juice	100	CTG0000001	Drink	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000002	PC	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000003	Hot selling	<NULL>	<NULL>	<NULL>
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000001	Join coupon	3000
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000002	PC coupon	30000

図 6.12　**Picture - ResultMap for Item**

項番	説明
1.	検索結果を `Item` オブジェクトにマッピングする。 `type` 属性にマッピングするクラスを指定する。
2.	取得したレコードの `item_code` カラムの値を、`Item#code` に設定する。 `item_code` カラムは PK なので、`id` 要素を使用してマッピングを行う。
3.	取得したレコードの `item_name` カラムの値を、`Item#name` に設定する。
4.	取得したレコードの `item_price` カラムの値を、`Item#price` に設定する。

注釈: `Item` オブジェクトには、`id` カラムと `item_code` カラムでグループ化されたレコードの値が設定される。

Category オブジェクトへのマッピングの実装

`Category` オブジェクトへのマッピングを行う。

```
<resultMap id="itemResultMap" type="Item">
    <id property="code" column="item_code"/>
    <result property="name" column="item_name"/>
    <result property="price" column="item_price"/>
    <!-- (1) -->
    <collection property="categories" ofType="Category">
        <!-- (2) -->
        <id property="code" column="category_code"/>
        <!-- (3) -->
        <result property="name" column="category_name"/>
    </collection>
</resultMap>
```

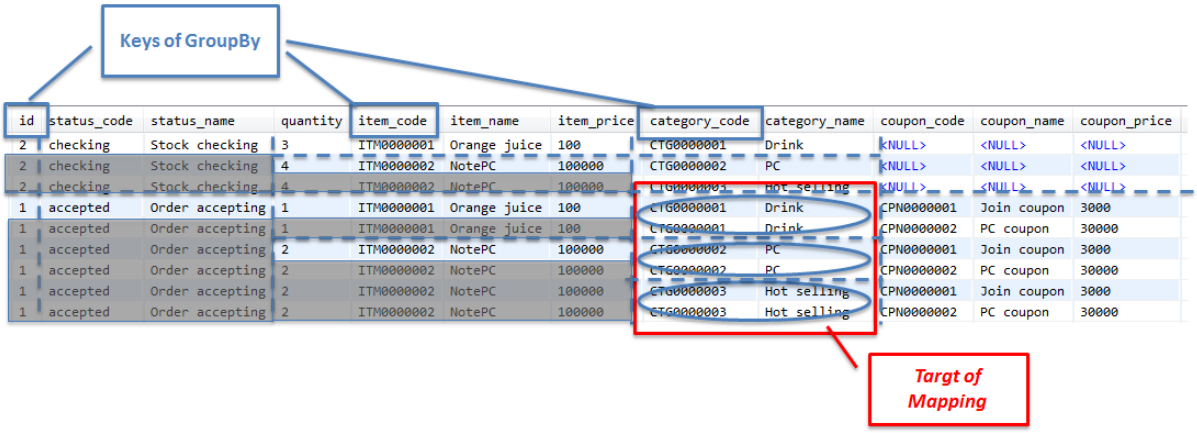


図 6.13 **Picture - ResultMap for Category**

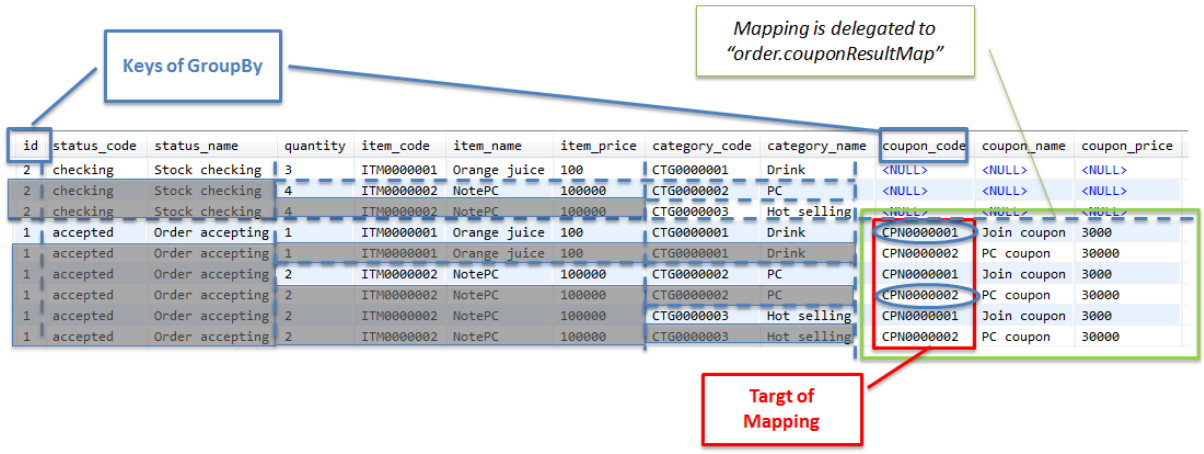
項番	説明
1.	検索結果を Category オブジェクトにマッピングし、Item#categories プロパティに追加する。 1:N の関係の関連 Entity にマッピングする場合は、collection 要素を使用する。collection 要素の詳細は、「MyBatis3 REFERENCE DOCUMENTATION(Mapper XML Files-collection-)」を参照されたい。
2.	取得したレコードの category_code カラムの値を、Category#code に設定する。 category_code カラムは PK なので、id 要素を使用してマッピングを行う。 id 要素を使用すると、指定したプロパティの値でレコードがグループ化される。 具体的には、 • Item#code=ITM0000001 のカテゴリとして、Category#code=CTG0000001 の Category オブジェクト • Item#code=ITM0000002 のカテゴリとして、Category#code=CTG0000002 と Category#code=CTG0000003 の 2 つの Category オブジェクト が生成される。
3.	取得したレコードの category_name カラムの値を、Category#name に設定する。

注釈: Category オブジェクトには、id カラムと item_code カラムと category_code カラムでグループ化されたレコードの値が設定される。

OrderCoupon オブジェクトへのマッピングの実装

OrderCoupon オブジェクトへのマッピングを行う。

```
<resultMap id="orderResultMap" type="Order">
    <id property="id" column="id"/>
    <result property="orderStatus.code" column="status_code" />
    <result property="orderStatus.name" column="status_name" />
    <collection property="orderItems" ofType="OrderItem">
        <id property="orderId" column="id"/>
        <id property="item.code" column="item_code"/>
        <result property="quantity" column="quantity"/>
        <association property="item" resultMap="itemResultMap"/>
    </collection>
    <!-- (1) -->
    <collection property="orderCoupons" ofType="OrderCoupon" notNullColumn=
↪"coupon_code">
        <!-- (2) -->
        <id property="orderId" column="id"/>
        <!-- (3) -->
        <id property="coupon.code" column="coupon_code"/>
        <result property="coupon.name" column="coupon_name"/>
        <result property="coupon.price" column="coupon_price"/>
    </collection>
</resultMap>
```



id	status_code	status_name	quantity	item_code	item_name	item_price	category_code	category_name	coupon_code	coupon_name	coupon_price
2	checking	Stock checking	3	ITM0000001	Orange juice	100	CTG0000001	Drink	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000002	PC	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000003	Hot selling	<NULL>	<NULL>	<NULL>
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000001	Join coupon	3000
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000002	PC coupon	30000

図 6.14 **Picture - ResultMap for OrderCoupon**

項番	説明
1.	検索結果を OrderCoupon オブジェクトにマッピングし、Order#orderCoupons プロパティに追加する。 1:N の関係の関連 Entity にマッピングする場合は、collection 要素を使用する。collection 要素の詳細は、「MyBatis3 REFERENCE DOCUMENTATION(Mapper XML Files-collection-)」を参照されたい。 上記例にて、notNullColumn 属性を指定している点に注目してほしい。 これは t_coupon テーブルにレコードが存在しない時に、OrderCoupon オブジェクトを生成させないための設定である。本実装例では、id カラムが」2」のデータには t_coupon テーブルのレコードを格納していないため、検索結果をみると、coupon_code と coupon_name と coupon_price の値が null になっているのがわかる。 OrderCoupon オブジェクトにマッピングするカラムがこの 3 つだけであれば、notNullColumn 属性を指定する必要はないが、実装例では id カラムの値を OrderCoupon#orderId プロパティにマッピングする設定を行っているため、notNullColumn 属性の指定が必要となる。これは、マッピング対象のカラムの中に null でない値がセットされていた場合に、MyBatis がオブジェクトを生成するためである。 上記例のように、notNullColumn 属性に coupon_code カラムを指定しておくと、coupon_code カラムが null でない場合 (つまり、レコードが存在する場合) にのみ、オブジェクトが生成される。notNullColumn 属性には、複数のカラムを指定する事もできる。
2.	取得したレコードの id カラムの値を、OrderCoupon#orderId プロパティに設定する。 orderId は PK なので、id 要素を使用する。
3.	取得したレコードの coupon_code カラムの値を Coupon#code に設定する。 coupon_code カラムは PK なので、id 要素を使用してマッピングを行う。id 要素を使用すると、指定したプロパティの値でレコードがグループ化される。 具体的には、Coupon#code=CPN0000001 と Coupon#code=CPN0000002 の 2 つにグループ化され、2 つの OrderCoupon オブジェクトが生成される。

Coupon オブジェクトへのマッピングの実装

Coupon オブジェクトへのマッピングを行う。

```
<resultMap id="orderResultMap" type="Order">
    <id property="id" column="id"/>
    <result property="orderStatus.code" column="status_code" />
    <result property="orderStatus.name" column="status_name" />
    <collection property="orderItems" ofType="OrderItem">
        <id property="orderId" column="id"/>
```

```
        <id property="item.code" column="item_code"/>
        <result property="quantity" column="quantity"/>
        <association property="item" resultMap="itemResultMap"/>
    </collection>
    <collection property="orderCoupons" ofType="OrderCoupon" notNullColumn=
↪"coupon_code">
        <id property="orderId" column="id"/>
        <!-- (1) -->
        <id property="coupon.code" column="coupon_code"/>
        <!-- (2) -->
        <result property="coupon.name" column="coupon_name"/>
        <!-- (3) -->
        <result property="coupon.price" column="coupon_price"/>
    </collection>
</resultMap>
```

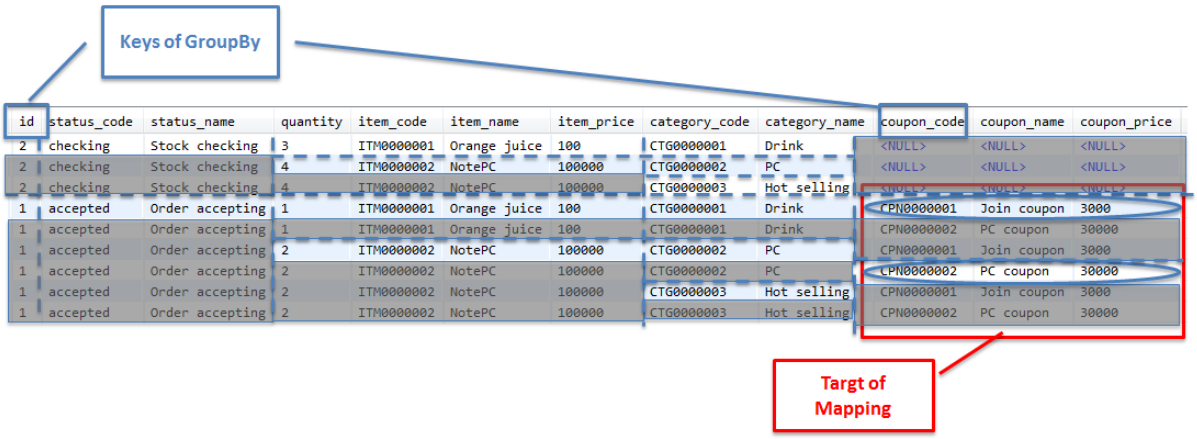


図 6.15 **Picture - ResultMap for Coupon**

項番	説明
1.	取得したレコードの coupon_code カラムの値を、Coupon#code に設定する。
2.	取得したレコードの coupon_name カラムの値を、Coupon#name に設定する。
3.	取得したレコードの coupon_price カラムの値を、Coupon#price に設定する。

注釈: Coupon オブジェクトには、id カラムと coupon_code カラムでグループ化されたレコードの値が設定される。

マッピング後のオブジェクト図

実際にマッピングされた Order オブジェクトおよび関連 Entity の状態は、以下の通りである。

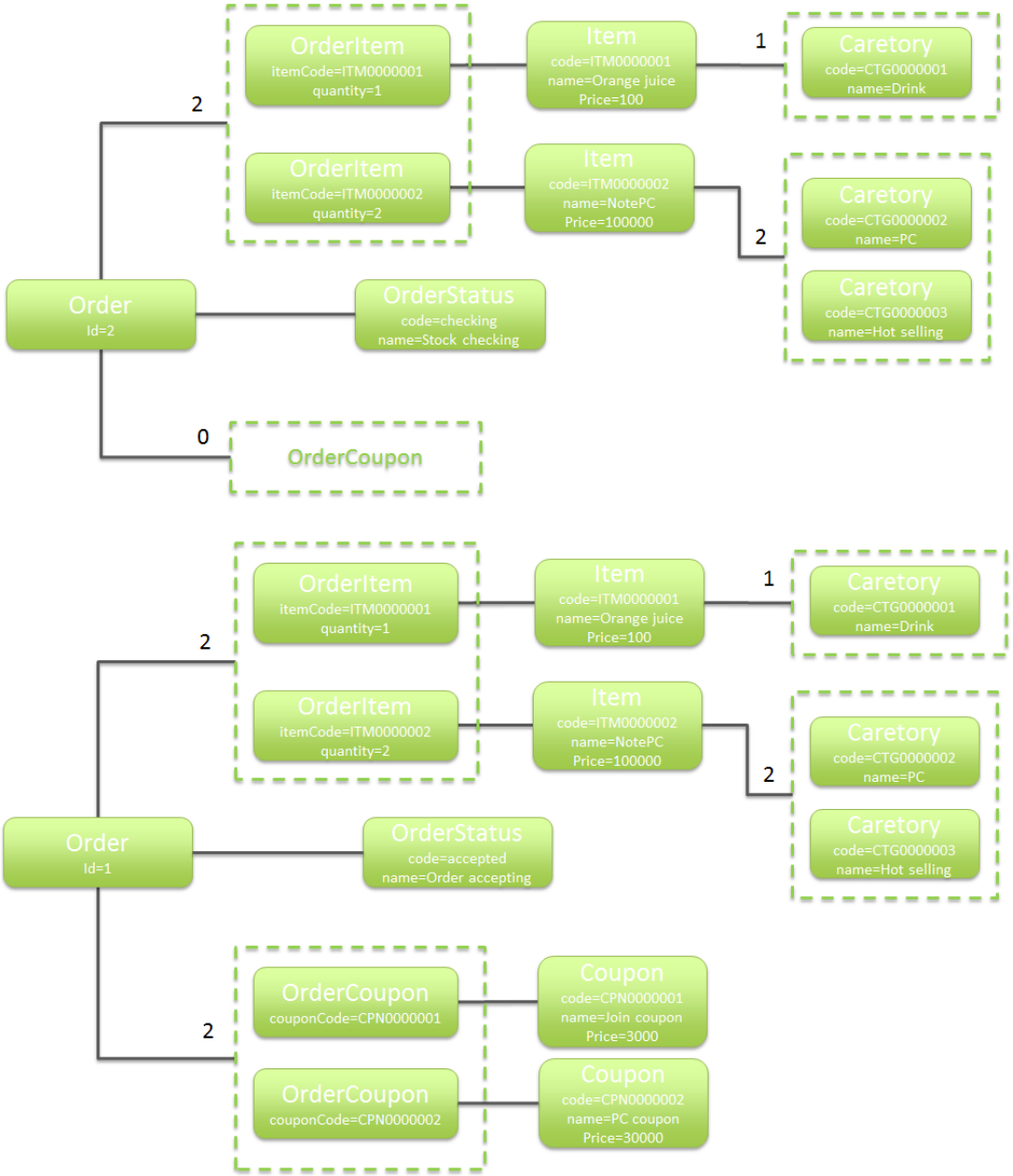
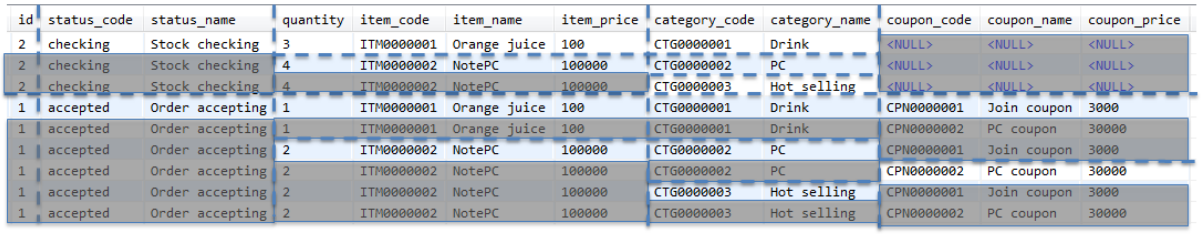


図 6.16　**Picture - Mapped object diagram**

Order オブジェクトにマッピングされたレコードとカラムは、以下の通りである。

グレーアウトしている部分は、グループ化によって、グレーアウトされていない部分にマージされる。



id	status_code	status_name	quantity	item_code	item_name	item_price	category_code	category_name	coupon_code	coupon_name	coupon_price
2	checking	Stock checking	3	ITM0000001	Orange juice	100	CTG0000001	Drink	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000002	PC	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000003	Hot selling	<NULL>	<NULL>	<NULL>
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000001	Join coupon	3000
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000002	PC coupon	30000

図 6.17　**Picture - Valid Result Set**

> **警告:** 1:N の関連をもつレコードを JOIN してマッピングする場合、グレーアウトされている部分のデータの取得が無駄になる点を、意識しておくこと。
>
> N の部分のデータを使用しない処理で、同じ SQL を使用した場合、さらに無駄なデータの取得となってしまうので、N の部分を取得する SQL と、取得しない SQL を、別々に用意しておくなどの工夫を行うこと。

関連 Entity をネストした SQL を使用して取得する方法について

MyBatis3 では、マッピング時に別の SQL(ネストした SQL) を使用して関連 Entity を取得する方法を提供している。

ネストした SQL を使用して関連 Entity を取得する仕組みを使用すると、

- 個々の SQL 定義
- `resultMap` 要素のマッピング定義

をシンプルにする事ができる。

> **警告:** 各種定義がシンプルになる一方で、ネストした SQL を多用すると、N+1 問題を引き起こす要因になるという事を意識する必要がある。
>
> ネストした SQL を使用する場合の MyBatis のデフォルトの動作は、」Eager Load」となる。これは、関連 Entity の使用有無に関係なく SQL が発行される事を意味しており、
>
> - 無駄な SQL の実行とデータの取得
> - N+1 問題
>
> などが発生する危険性が高まる。

ちなみに: MyBatis3 では、ネストした SQL を使用して関連 Entity を取得する際の動作を、「Lazy

Load」に変更するためのオプションを提供している。

「Lazy Load」の使用方法については、「関連 *Entity* を *Lazy Load* するための設定」を参照されたい。

関連 Entity をネストした SQL を使用して取得する実装例

ネストした SQL を使用して関連 Entity を取得する際の実装例を以下に示す。

```
<resultMap id="itemResultMap" type="Item">
    <id property="code" column="item_code"/>
    <result property="name" column="item_name"/>
    <result property="price" column="item_price"/>
    <!-- (1) -->
    <collection property="categories" column="item_code"
        select="findAllCategoryByItemCode" />
</resultMap>

<select id="findAllCategoryByItemCode"
    parameterType="string" resultType="Category">
    SELECT
        ct.code,
        ct.name
    FROM
        m_item_category ic
    INNER JOIN m_category ct ON ct.code = ic.category_code
    WHERE
        ic.item_code = #{itemCode}
    ORDER BY
        code
</select>
```

項番	説明
1.	association 要素又は collection 要素の select 属性に、呼び出す SQL のステートメント ID を指定する。 column 属性には、SQL に渡すパラメータ値が格納されているカラム名を指定する。上記例では、findAllCategoryByItemCode のパラメータとして item_code カラムの値を渡している。 指定可能な属性の詳細は、「MyBatis3 REFERENCE DOCUMENTATION(Mapper XML Files-Nested Select for Association-)」を参照されたい。

注釈: 上記例では、fetchType 属性を指定していないため、「Lazy Load」と」Eager Load」のどちらで実行されるかは、アプリケーション全体の設定に依存する。

アプリケーション全体の設定については、「*Lazy Load を使用するための MyBatis の設定*」を参照されたい。

関連 Entity を Lazy Load するための設定

ネストした SQL を使用して関連 Entity を取得する際の MyBatis3 のデフォルト動作は、「Eager Load」であるが、」Lazy Load」を使用する事も可能である。

以下に、」Lazy Load」を使用するために最低限必要な設定及び使用方法について説明を行う。

説明していない設定値については、「MyBatis3 REFERENCE DOCUMENTATION(Mapper XML Files-settings-)」を参照されたい。

バイトコード操作ライブラリの追加

「Lazy Load」を使用する場合は、」Lazy Load」を実現するための Proxy オブジェクトを生成するために、

- JAVASSIST
- CGLIB

のいずれか一方のライブラリが必要となる。

MyBatis 3.2 系までは CGLIB がデフォルトで使用されるライブラリであったが、MyBatis 3.3.0 以降のバージョンでは JAVASSIST がデフォルトで使用される。さらに、MyBatis 3.3.0 から JAVASSIST が MyBatis 本体に内包されているため、ライブラリを追加しなくても」Lazy Load」を使用する事ができる。

注釈: MyBatis 3.3.0 以降のバージョンで CGLIB を使用する場合は、

- `pom.xml` に CGLIB のアーティファクトを追加
- MyBatis 設定ファイル (`projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml`) に「`proxyFactory=CGLIB`」を追加

すればよい。

CGLIB のアーティファクト情報については、「MyBatis3 PROJECT DOCUMENTATION(Project Dependencies-compile-)」を参照されたい。

Lazy Load を使用するための MyBatis の設定

MyBatis3 では、」Lazy Load」の使用有無を、

- アプリケーションの全体設定 (MyBatis 設定ファイル)
- 個別設定 (マッピングファイル)

の 2 箇所で指定する事ができる。

- アプリケーションの全体設定は、MyBatis 設定ファイル (`projectName-domain/src/main/resources/META-INF/mybatis/mybatis-config.xml`) に指定する。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <settings>
        <!-- (1) -->
        <setting name="lazyLoadingEnabled" value="true"/>
    </settings>
</configuration>
```

項番	説明
1.	アプリケーションのデフォルト動作を `lazyLoadingEnabled` に指定する。 • `true`: 「Lazy Load」 • `false`: 「Eager Load」 (デフォルト) `association` 要素と `collection` 要素の `fetchType` 属性を指定した場合は、`fetchType` 属性の指定値が優先される。

警告: 「`false`:「Eager Load」」の状態で `association` 要素又は `collection` 要素の `select` 属性を使用すると、マッピング時に SQL が実行されるので、注意が必要である。

特に理由がない場合は、`lazyLoadingEnabled` は `true` にする事を推奨する。

- 個別設定は、マッピングファイルの `association` 要素と `collection` 要素の `fetchType` 属性で指定する。

```
<resultMap id="itemResultMap" type="Item">
    <id property="code" column="item_code"/>
    <result property="name" column="item_name"/>
    <result property="price" column="item_price"/>
    <!-- (2) -->
    <collection property="categories" column="item_code"
        fetchType="lazy"
        select="findAllCategoryByItemCode" />
</resultMap>

<select id="findAllCategoryByItemCode"
    parameterType="string" resultType="Category">
    SELECT
        ct.code,
        ct.name
    FROM
        m_item_category ic
    INNER JOIN m_category ct ON ct.code = ic.category_code
    WHERE
        ic.item_code = #{itemCode}
    ORDER BY
        code
</select>
```

項番	説明
2.	association 要素又は collection 要素の fetchType 属性に、lazy 又は eager を指定する。 fetchType 属性を指定すると、アプリケーション全体の設定を上書きする事ができる。

Lazy Load の実行タイミングを制御するための設定

MyBatis3 では、」Lazy Load」を実行するタイミングを制御するためのオプション (aggressiveLazyLoading) を提供している[*1]。

このオプションのデフォルト値は Mybatis 3.4.2 以降から false であり、」Lazy Load」対象となっているプロパティの getter メソッドが呼び出されたタイミングで実行する。

> **警告:** aggressiveLazyLoading が「true」の場合、」Lazy Load」対象となっているプロパティを保持するオブジェクトの getter メソッドが呼び出されたタイミングで」Lazy Load」が実行される。このため、実際にはデータの取得が必要ないにもかかわらず SQL が実行されてしまう可

[*1] 設定方法は、MyBatis のリファレンス を参照されたい。

能性があることに注意が必要である。

具体的には、以下のようなマッピングを行い、「Lazy Load」対象になっていないプロパティだけにアクセスするケースである。「true」の場合、」Lazy Load」対象のプロパティに対して直接アクセスしなくても、「Lazy Load」が実行されてしまう。

特に理由がない場合は、aggressiveLazyLoading は「false」(デフォルト) のまま変更しないことを推奨する。

- Entity

```
public class Item implements Serializable {
    private static final long serialVersionUID = 1L;
    private String code;
    private String name;
    private int price;
    private List<Category> categories;
    // ...
}
```

- マッピングファイル

```
<resultMap id="itemResultMap" type="Item">
    <id property="code" column="item_code"/>
    <result property="name" column="item_name"/>
    <result property="price" column="item_price"/>
    <collection property="categories" column="item_code"
        fetchType="lazy" select="findByItemCode" />
</resultMap>
```

- アプリケーションコード (Service)

```
    Item item = itemRepository.findOne(itemCode);
    // (1)
    String code = item.getCode();
    String name = item.getName();
    String price = item.getPrice();
    // ...
}
```

項番	説明
1.	上記例では、」Lazy Load」対象のプロパティである categories プロパティにアクセスしていないが、Item#code プロパティにアクセスした際に、」Lazy Load」が実行される。 「false」(デフォルト) の場合、上記のケースでは」Lazy Load」は実行されない。

6.3 データベースアクセス（JPA 編）

課題：TBD

本章では以下の内容について、現在精査中である。

- `persistence.xml` の設定項目について。
- QueryDSL を使用した動的 Query の実装方法について。
- 関連エンティティの fetch 方法を指定する設定値をデフォルト値から変更した方がよいケースの具体例について。
- 複数の PersistenceUnit を使用する方法について。
- Native クエリの使用方法について。

6.3.1 Overview

本節では、JPA を使ってデータベースにアクセスする方法について、説明する。

本ガイドラインでは、JPA プロバイダとして Hibernate を、JPA のラッパーとして Spring Data JPA を使用することを前提としている。

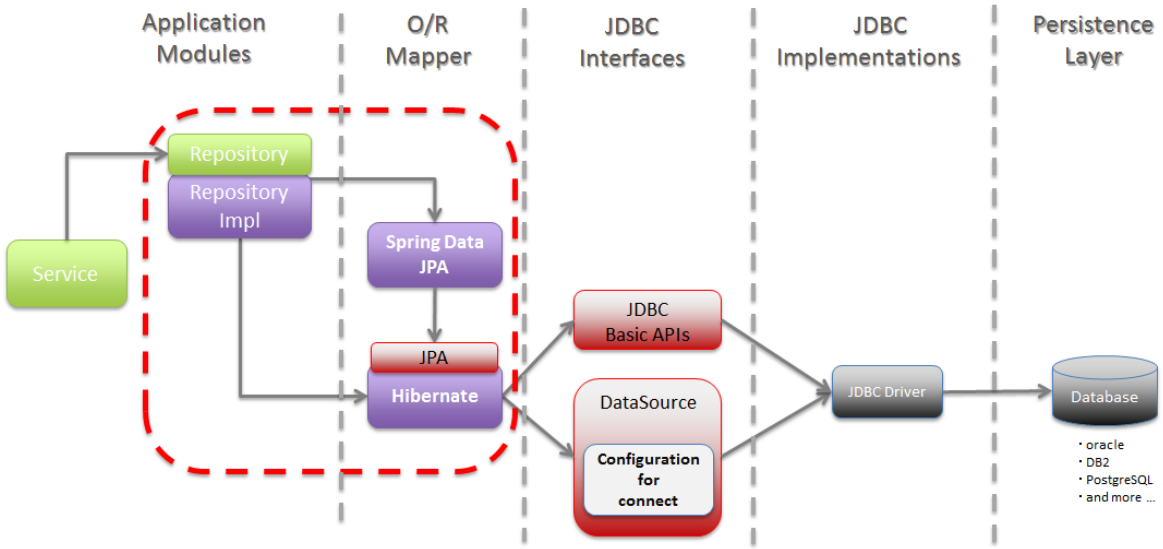


図 6.18　**Picture - Target of description**

警告： 本節で説明した内容が EclipseLink など、Hibernate 以外の JPA プロバイダでも動作することは保証しない。

JPA について

JPA(Java Persistence API) は、リレーショナルデータベースで管理されているレコードを、Java オブジェクトにマッピングする方法と、マッピングされた Java オブジェクトに対して行われた操作を、リレーショナルデータベースのレコードに反映するための仕組みを Java の API 仕様として定義したものである。

JPA は、仕様の定義をしているだけで、実装は提供していない。

JPA の実装は、Hibernate のような O/R Mapper を開発しているベンダーによって、参照実装として提供されている。

このように、O/R Mapper を開発しているベンダーによって実装された参照実装のことを、JPA プロバイダと呼ぶ。

JPA の O/R Mapping

JPA を使用した際の、リレーショナルデータベースで管理されているレコードと Java オブジェクトは、以下のようなイメージでマッピングされる。

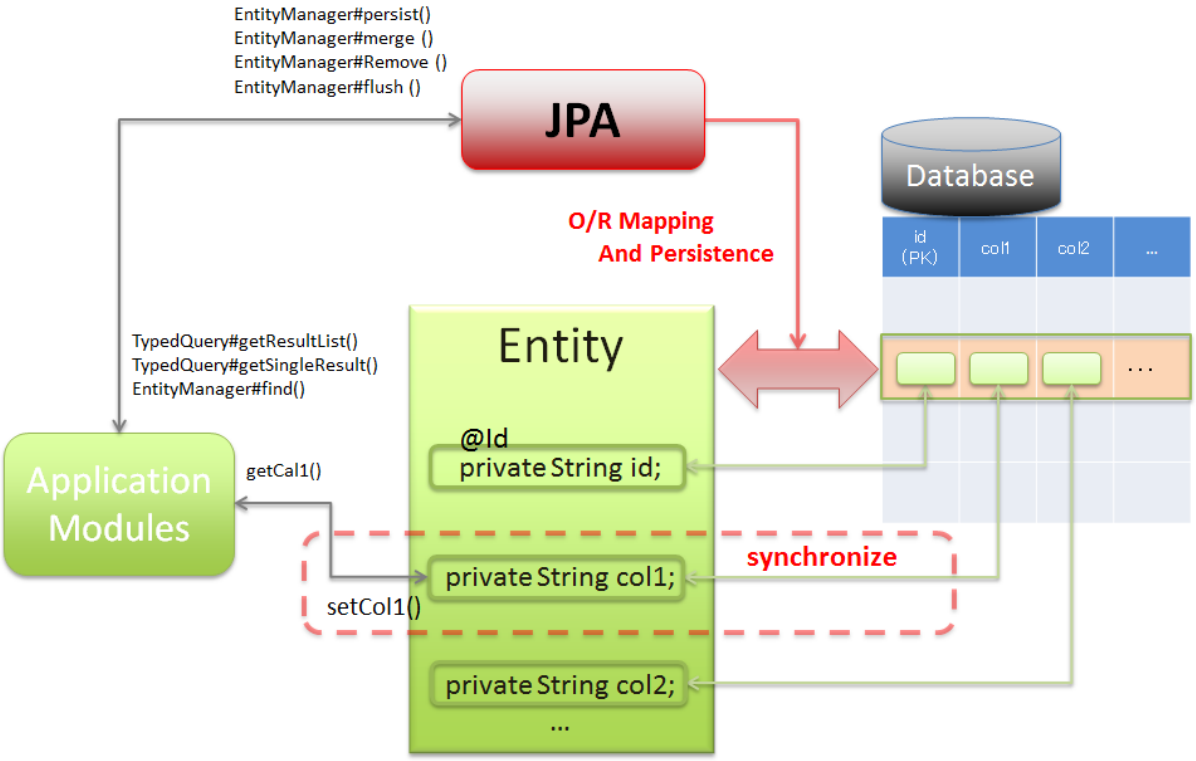


図 6.19 **Picture - Image of O/R Mapping**

JPA では、「管理状態」と呼ばれる状態の Entity が保持している値を、変更 (setter メソッドを呼び出して値を変更) した場合、変更内容が、リレーショナルデータベースに反映される仕組みになっている。

これは、編集機能をもつテーブルビューアなどのクライアントソフトウェアに、よく似ている仕組みである。

テーブルビューアなどのクライアントソフトウェアでは、ビューア上の値を変更すると、データベースに反映されるが、JPA では、Entity と呼ばれる Java オブジェクト (JavaBean) の値を変更すると、データベースに反映されることになる。

JPA の基本用語

以下に、JPA を使う上で、最低限知っていてほしい用語について、簡単に説明する。

項番	用語	説明
1.	Entity クラス	リレーショナルデータベースで管理されているレコードを表現する Java クラス。 `@javax.persistence.Entity` アノテーションを付与されたクラスが、Entity クラスとなる。
2.	EntityManager	Entity のライフサイクルを管理するために、必要な API を提供するインタフェース。 アプリケーションは、`javax.persistence.EntityManager` のメソッドを使用して、リレーショナルデータベースで管理されているレコードを、Java オブジェクトとして操作する。 Spring Data JPA を使う場合は、直接、使用することはないが、Spring Data JPA の仕組みでは表現できないような、Query を発行する必要がある場合は、このインタフェースを経由して Entity を取得することになる。
3.	TypedQuery	Entity を検索するための API を提供するインタフェース。 アプリケーションは、`javax.persistence.TypedQuery` のメソッドを使用して、ID 以外の条件に一致する Entity を検索する。 Spring Data JPA を使う場合は、直接使用することはないが、Spring Data JPA の仕組みでは、表現できないような Query を発行する必要がある場合は、このインタフェースを使用して Entity を検索することになる。 条件に一致する永続層 (DB) の Entity を直接操作 (更新または削除) するためのメソッドもこのインタフェースに用意されている。

Continued on next page

表 6.11 – continued from previous page

項番	用語	説明
4.	PersistenceContext	Entity を管理するための領域。 `EntityManager` を経由して取得または作成された Entity は、この領域に格納されてライフサイクル管理される。この領域で管理されている Entity の事を「管理状態の Entity」と呼ぶ。 この領域は、アプリケーションから直接アクセスすることはできない。 Entity の状態は「管理状態」以外に、「作成状態」「削除状態」「分離状態」が存在する。
5.	find メソッド	管理状態の Entity を取得するためのメソッド。 ID に対応する Entity が PersistenceContext に存在しない場合は、リレーショナルデータベースに格納されているレコードを取得 (SELECT) し、管理状態の Entity を生成する。
6.	persist メソッド	アプリケーションで作成した作成状態の Entity を、管理状態の Entity にするためのメソッド。 `EntityManager` のメソッドとして提供されており、リレーショナルデータベースにレコードの INSERT を実行するための操作が PersistenceContext に蓄積される。
7.	merge メソッド	PersistenceContext で管理されていない分離状態の Entity を、管理状態の Entity にするためのメソッド。 `EntityManager` のメソッドとして提供されており、基本的にはリレーショナルデータベースに格納されているレコードに対して、UPDATE を実行するための操作が PersistenceContext に蓄積される。 ただし、リレーショナルデータベースに ID に一致するレコードが存在しない場合は、UPDATE ではなく INSERT が実行される。
8.	remove メソッド	管理状態の Entity を、削除状態の Entity にするためのメソッド。 `EntityManager` のメソッドとして提供されており、リレーショナルデータベースに格納されているレコードに対して、DELETE を実行するための操作が PersistenceContext に蓄積される。

Continued on next page

表 6.11 – continued from previous page

項番	用語	説明
9.	flush メソッド	PersistenceContext で管理されている Entity に対して行われた操作を、リレーショナルデータベースに強制的に反映するためのメソッド。 `EntityManager` のメソッドとして提供されており、蓄積された未反映の操作をリレーショナルデータベースに対して実行する。 通常、リレーショナルデータベースへの反映は、トランザクションコミット時に行われるが、コミットより前に反映する必要がある場合は、メソッドを使用する。

Entity のライフサイクル管理

Entity のライフサイクル管理イメージは、以下の通りである。

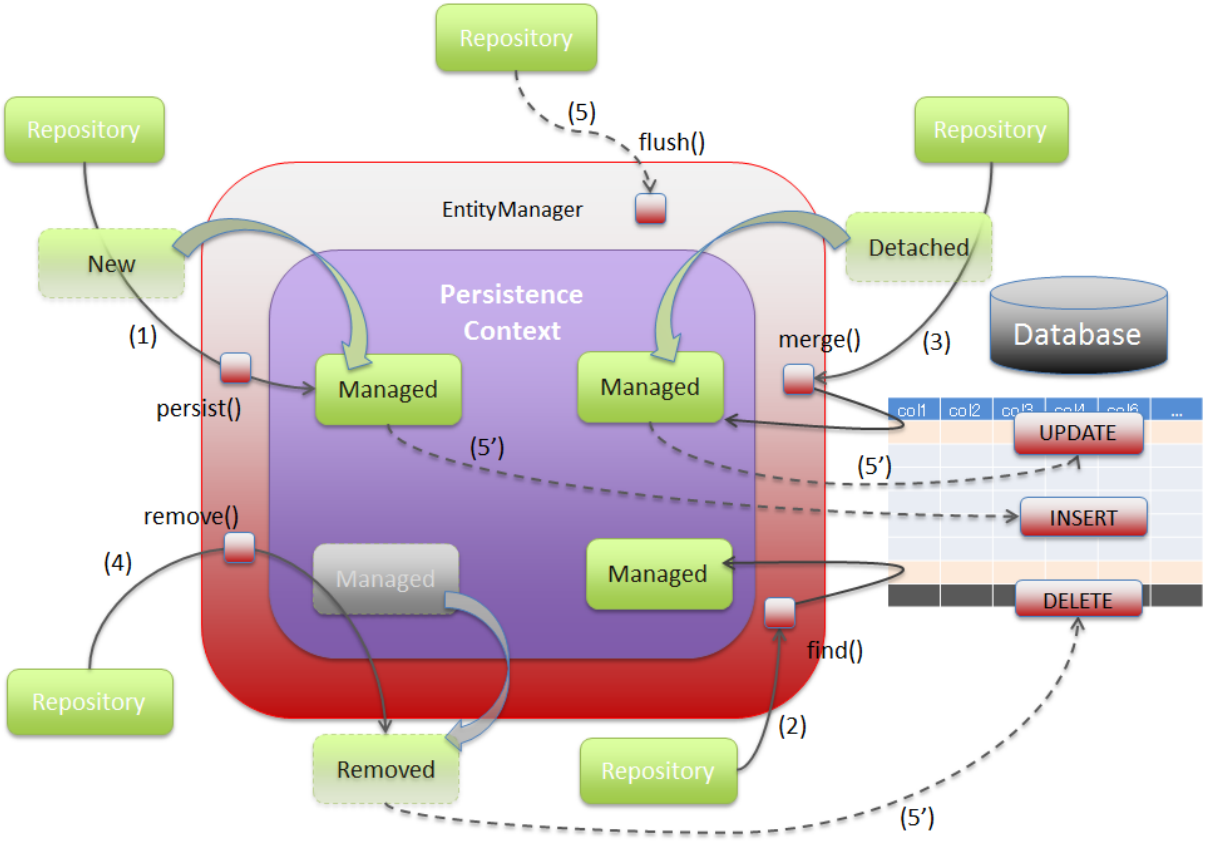


図 6.20　**Picture - Life cycle of entity**

項番	説明
(1)	`EntityManager` の persist メソッドを呼び出すと、引数に渡した Entity(作成状態の Entity) が、管理状態の Entity として、PersistenceContext に格納される。
(2)	`EntityManager` の find メソッドを呼び出すと、引数に渡した ID をもつ管理状態の、Entity が返却される。 PersistenceContext に存在しない場合は、Query を発行して、リレーショナルデータベースよりマッピング対象のレコードを取得し、管理状態の Entity として格納する。

Continued on next page

表 6.12 – continued from previous page

項番	説明
(3)	`EntityManager` の merge メソッドを呼び出すと、引数に渡した Entity(分離状態の Entity) の状態が、管理状態の Entity にマージされる。 PersistenceContext に存在しない場合は、Query を発行してリレーショナルデータベースよりマッピング対象のレコードを取得し、管理状態の Entity を格納した後に、引数に渡された Entity の状態がマージされる。 このメソッドを呼び出した場合、persist メソッドとは異なり、引数に渡した Entity が管理状態の Entity として格納されるわけではないという点に注意すること。
(4)	`EntityManager` の remove メソッドを呼び出すと、引数に渡した管理状態の Entity が削除状態の Entity となる。 このメソッドを呼び出した場合、削除状態となった Entity を取得することはできなくなる。
(5)	`EntityManager` の flush メソッドを呼び出すと、persist、merge、remove メソッドによって、蓄積された Entity への操作が、リレーショナルデータベースに反映される。 このメソッドを呼び出すことで、Entity に対して行った変更内容が、リレーショナルデータベースのレコードに同期される。 ただし、リレーショナルデータベース側のレコードに対してのみ行われた変更は、Entity には同期されない `EntityManager` の find メソッドを使わずに Query を発行して Entity を検索すると、検索処理を行う前に、`EntityManager` 内部の処理で flush メソッドと同等の処理が実行され、蓄積されていた Entity への操作がリレーショナルデータベースに反映される仕組みになっている。 Spring Data JPA を使用した際の永続操作の反映タイミングについては、 永続操作の反映タイミングについて *(その 1)* 永続操作の反映タイミングについて *(その 2)* を参照されたい。

注釈: 他のライフサイクル管理用のメソッドについて

`EntityManager` には、Entity のライフサイクルを管理するためのメソッドとして、detach メソッド、refresh メソッド、clear メソッドなどがあるが、Spring Data JPA を使用する場合、デフォルトの機能で、これらのメソッドを呼び出す仕組みが存在しないため、メソッドの役割のみ説明しておく。

- detach メソッドは、管理状態の Entity を分離状態の Entity にするためのメソッド。
- refresh メソッドは、管理状態の Entity をリレーショナルデータベースの状態で最新化するためのメソッド。
- clear メソッドは、PersistenceContext で、管理されている Entity および蓄積された操作をメモリ上から破棄するためのメソッド。

clear メソッドについては、Spring Data JPA より提供されている`@Modifying` アノテーションの clearAutomatically 属性を、`true` に設定することで、呼び出すことができる。詳細については、永続層の *Entity* を直接操作するを参照されたい。

注釈: 作成状態と分離状態の **Entity** への操作について

作成状態の Entity や、分離状態の Entity に行った操作は、persist メソッドまたは merge メソッドを呼び出さないと、リレーショナルデータベースには反映されない。

Spring Data JPA について

Spring Data JPA は、JPA を使って Repository を作成するための、ライブラリを提供している。

Spring Data JPA を使用すると、Query メソッドと呼ばれるメソッドを、Repository インタフェースに定義するだけで、
指定した条件に一致する Entity を取得することが出来るため、Entity の操作を行うための実装を減らすことができる。
ただし、Query メソッドで定義できるのはアノテーションで表現できる静的な Query のみなので、
動的 Query などのアノテーションで表現できない Query については、カスタム Repository クラスの実装が必要となる。

Spring Data JPA を使ってデータベースにアクセスする際の基本フローを以下に示す。

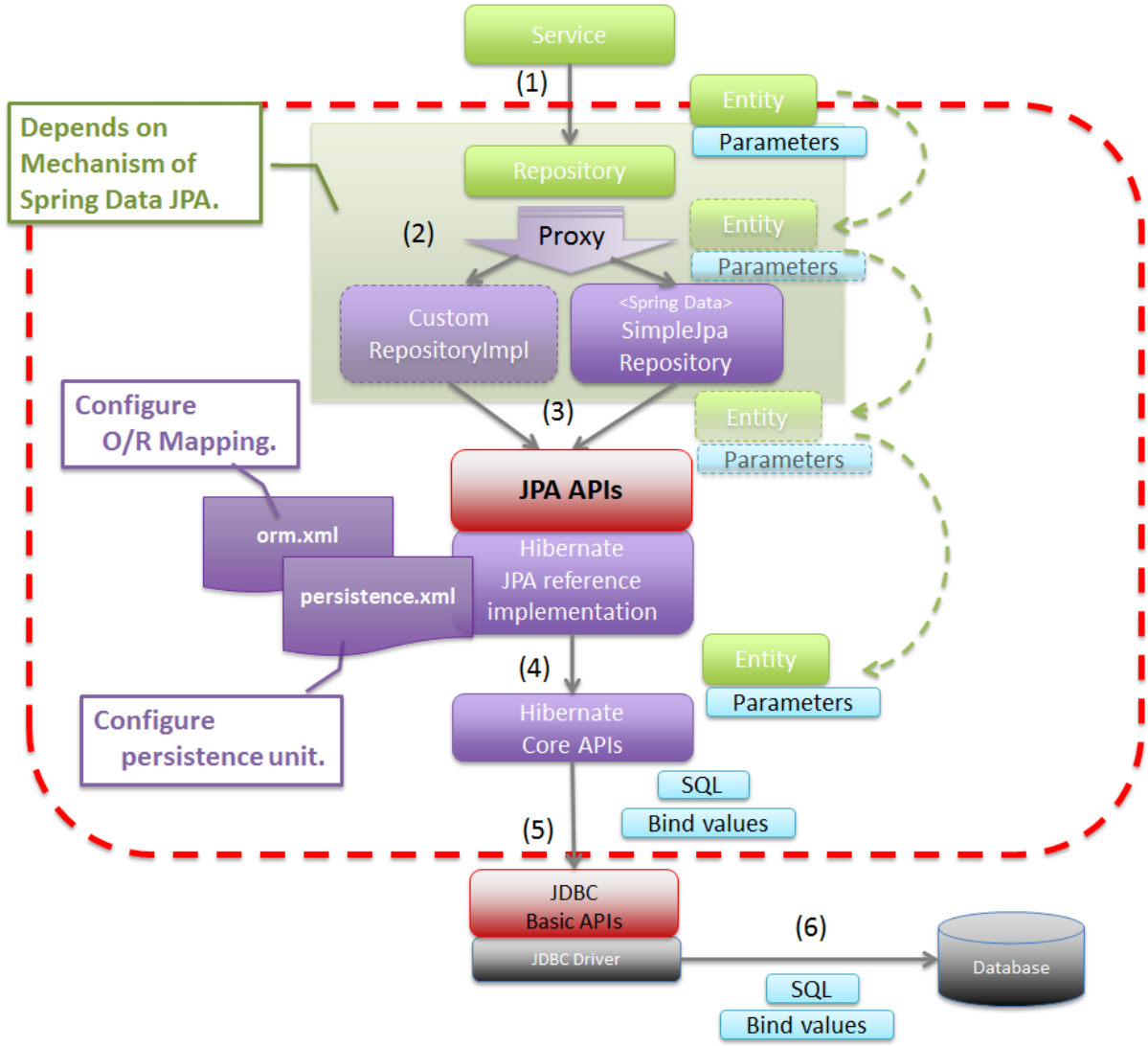


図 6.21 **Picture - Basic flow of Spring Data JPA**

項番	説明
(1)	Service から、Repository インタフェースのメソッドを呼び出す。 メソッドの呼び出しパラメータとして、Entity オブジェクト、Entity の ID などが渡される。 上記例では Entity を渡しているが、プリミティブな値となることもある。
(2)	Repository インタフェースを動的に実装した Proxy クラスは、`org.springframework.data.jpa.repository.support.SimpleJpaRepository` や、 カスタム Repository クラスに処理を委譲する。 Service から指定されたパラメータが渡される。

Continued on next page

表 6.13 – continued from previous page

項番	説明
(3)	Repository の実装クラスは、JPA の API を呼び出す。 Service から指定されたパラメータや、Repository の実装クラスで生成したパラメータなどが渡される。
(4)	Hibernate の JPA 参照実装は、 Hibernate のコア API の処理を呼び出す。 Repository の実装クラスから指定されたパラメータや Hibernate の JPA 参照実装のクラスで生成したパラメータなどが渡される。
(5)	Hibernate のコア API は、指定されたパラメータから SQL とバインド値を生成し JDBC ドライバに渡す。 (実際の値のバインドは、java.sql.PreparedStatement の API が使われている)
(6)	JDBC ドライバは、渡された SQL とバインド値をデータベースに送信することで、SQL を実行する。

Spring Data JPA を使用して Repository を作成する場合、JPA の API を直接呼び出す必要はないが、Spring Data JPA の Repository インタフェースのメソッドが、

JPA のどのメソッドを呼び出しているのかは、意識しておいた方がよい。

以下に、Spring Data JPA の、Repository インタフェースの代表的なメソッドが、JPA のどのメソッドを呼び出しているのかを示す。

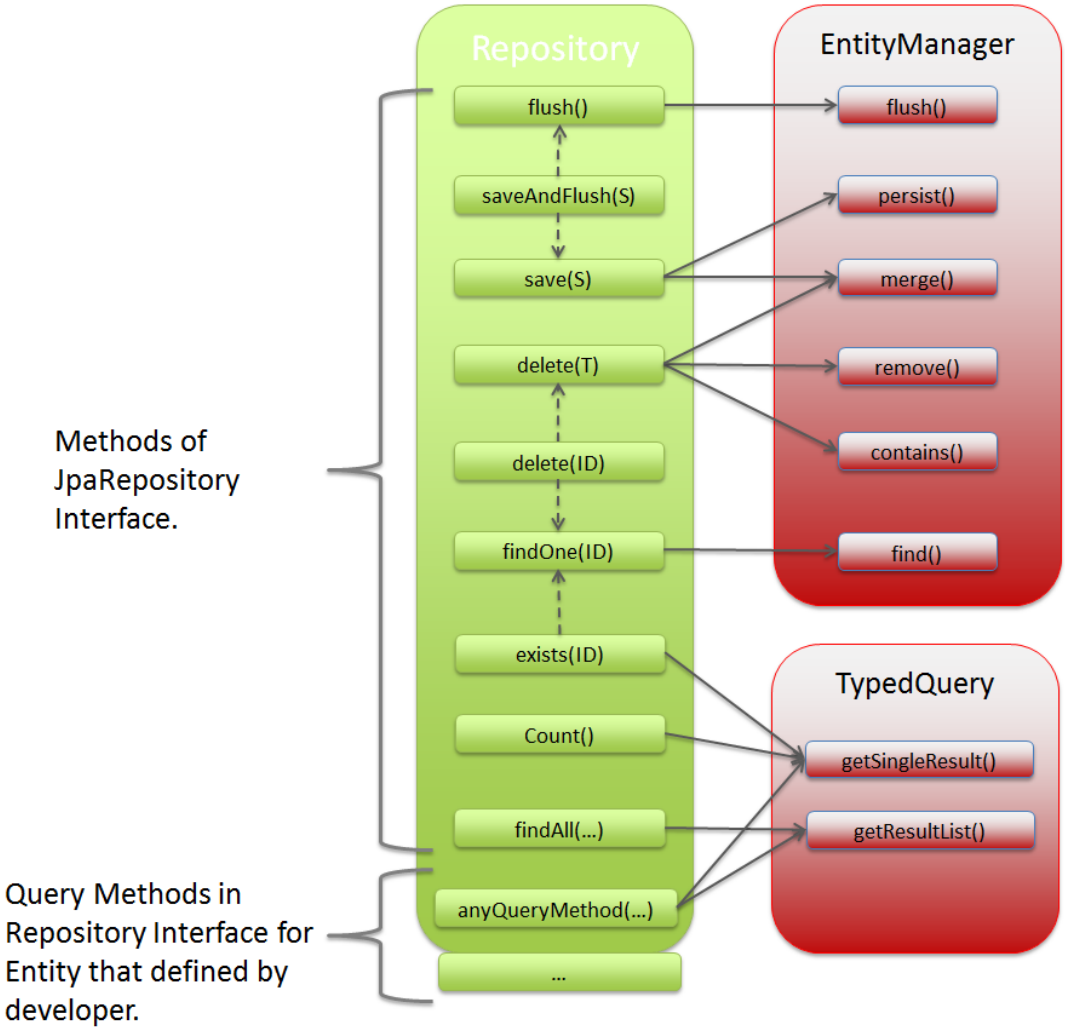


図 6.22 **Picture - API Mapping of Spring Data JPA and JPA**

6.3.2 How to use

pom.xml の設定

インフラストラクチャ層に JPA(Spring Data JPA) を使用する場合、以下の dependency を、pom.xml に追加する。

```
<!-- (1) -->
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-jpa-dependencies</artifactId>
    <type>pom</type>
</dependency>
```

項番	説明
(1)	JPA に関連するライブラリ群が定義してある `terasoluna-gfw-jpa-dependencies` を、dependency に追加する。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。

アプリケーションの設定

データソースの設定

データベースの接続情報をデータソースに設定する。

データソースの設定については、共通編のデータソースの設定を参照されたい。

EntityManager の設定

`EntityManager` を使用するための設定を行う。

- xxx-infra.xml

```
<!-- (1) -->
<bean id="jpaVendorAdapter"
    class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <!-- (2) -->
    <property name="showSql" value="false" />
    <!-- (3) -->
    <property name="database" value="POSTGRESQL" />
</bean>

<!-- (4) -->
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean
↪">
```

```
    <!-- (5) -->
    <property name="packagesToScan" value="xxxxxx.yyyyyy.zzzzzz.domain.model"␣
↪/>
    <!-- (6) -->
    <property name="dataSource" ref="dataSource" />
    <!-- (7) -->
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
    <!-- (8) -->
    <property name="jpaPropertyMap">
        <util:map>
            <entry key="hibernate.hbm2ddl.auto" value="" />
            <entry key="hibernate.ejb.naming_strategy"
                value="org.hibernate.cfg.ImprovedNamingStrategy" />
            <entry key="hibernate.connection.charSet" value="UTF-8" />
            <entry key="hibernate.show_sql" value="false" />
            <entry key="hibernate.format_sql" value="false" />
            <entry key="hibernate.use_sql_comments" value="true" />
            <entry key="hibernate.jdbc.batch_size" value="30" />
            <entry key="hibernate.jdbc.fetch_size" value="100" />
        </util:map>
    </property>
</bean>
```

項番	説明
(1)	JPA プロバイダが提供する実装クラスとのアダプタクラスを指定する。 JPA プロバイダとして Hibernate を使用するので、`org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter` を指定する。
(2)	SQL の出力有無を指定する。設定例では、「false:出力しない 」を指定している。
(3)	使用する RDBMS に対応する値を設定する。 `org.springframework.orm.jpa.vendor.Database` 列挙型に定義されている値を、指定することができる。 設定例では「PostgreSQL」を指定している。 **【プロジェクトで使用するデータベースに対応する値に変更が必要】** 環境によって使用するデータベースがかわる場合は、プロパティファイルに値を定義すること。

Continued on next page

表 6.14 – continued from previous page

項番	説明
(4)	`javax.persistence.EntityManagerFactory` のインスタンスを作成する FactoryBean のクラスを指定する。 `org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean` を指定する。
(5)	Entity クラスが格納されているパッケージを指定する。 指定したパッケージに格納されている Entity クラスが、`javax.persistence.EntityManager` で管理することができる Entity クラスとなる。 **【プロジェクトのパッケージに変更が必要】**
(6)	永続層 (DB) にアクセスする際に使用するデータソースを指定する。 設定済みのデータソースの bean を指定する。
(7)	`JpaVendorAdapter` の bean を指定する。 (1) で設定済みの bean を指定する。
(8)	Hibernate から提供されている `EntityManager` の動作設定を指定する。 詳細については「Hibernate Reference Documentation」を参照されたい。

ちなみに: データベースに Oracle を使う際に、テーブル結合を行う SQL に ANSI 標準の JOIN を使用したい場合は、(8) の jpaPropertyMap に、以下の設定を指定することで実現できる。

```
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.
↪LocalContainerEntityManagerFactoryBean">
    <!-- omitted -->
    <property name="jpaPropertyMap">
        <util:map>
            <!-- omitted -->
            <entry key="hibernate.dialect"
                   value="org.hibernate.dialect.Oracle12cDialect" />  <!--␣
↪(9) -->
        </util:map>
    </property>
</bean>
```

項番	説明
(9)	hibernate.dialect に org.hibernate.dialect.Oracle12cDialect を指定する。 Oracle12cDialect を指定することで、テーブル結合を行う SQL に ANSI 標準の JOIN 句が使用される。

アプリケーションサーバから提供されているトランザクションマネージャ (JTA) を使用する場合は、以下の設定を行う。

JTA を使用しない場合との差分について、説明する。

特に説明がない箇所については、JTA を使用しない場合と同じ設定でよい。

- xxx-infra.xml

```
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean
↪">

    <!-- omitted -->

    <!-- (10) -->
    <property name="jtaDataSource" ref="dataSource" />

    <!-- omitted -->
```

```
    <property name="jpaPropertyMap">
        <util:map>

            <!-- omitted -->

            <!-- (11)  -->
            <entry key="hibernate.transaction.jta.platform"
                value="org.hibernate.service.jta.platform.internal.
↪WeblogicJtaPlatform" />

        </util:map>
    </property>
</bean>
```

項番	説明
(10)	永続層 (DB) にアクセスする際に使用するデータソースを指定する。 JTA を使用する場合は、`dataSource` プロパティではなく、`jtaDataSource` プロパティに、アプリケーションサーバで定義した DataSource を指定する。 アプリケーションサーバで定義した DataSource の取得方法については、共通編のデータソースの設定を参照されたい。
(11)	`jpaPropertyMap` プロパティに、JTA のプラットフォームの指定を追加する。 上記は、Weblogic の JTA を使用する場合の設定例となる。 設定可能な値 (プラットフォーム) は、`org.hibernate.service.jta.platform.spi.JtaPlatform` の実装クラスの FQCN となる。 主なアプリケーションサーバ向けの実装クラスについては、Hibernate から提供されている。

注釈: 環境によって使用するトランザクションマネージャを切り替える必要がある場合は、`entityManagerFactory` の bean 定義は `xxx-infra.xml` ではなく、 `xxx-env.xml` に行うことを推奨する。

トランザクションマネージャを環境によって切り替える必要がある具体例としては、ローカルの開発環境では Tomcat など JTA の機能を持たないアプリケーションサーバを使用し、本番および各試験環境では、Weblogic などの JTA の機能をもつアプリケーションサーバを使用するといったケースがあげられる。

PlatformTransactionManager の設定

ローカルトランザクションを使用する場合は、以下の設定を行う。

- xxx-env.xml

```
<bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager"> <!-- (1) -->
    <property name="entityManagerFactory" ref="entityManagerFactory" /> <!--
↪(2) -->
</bean>
```

項番	説明
(1)	org.springframework.orm.jpa.JpaTransactionManager を指定する。 このクラスは、JPA の API を呼び出してトランザクション制御を行う。
(2)	トランザクション内で使用する EntityManager の、Factory を指定する。 設定済みの EntityManagerFactory の bean を指定する。

アプリケーションサーバから提供されているトランザクションマネージャ (JTA) を使用する場合は、以下の設定を行う。

- xxx-env.xml

```
<tx:jta-transaction-manager /> <!-- (1) -->
```

項番	説明
(1)	アプリケーションがデプロイされているアプリケーションサーバに最適な org.springframework.transaction.jta.JtaTransactionManager が、」transactionManager」という id で、bean 定義される。bean 定義されたクラスは、JTA の API を呼び出して、トランザクション制御を行う。

persistence.xml の設定

LocalContainerEntityManagerFactoryBean を使用する場合は、 persistence.xml に必ず設定しなくてはいけない設定項目はない。

課題：TBD

現時点では、persistence.xml に必ず設定しなくてはいけない設定項目はないが、今後増える可能性はある。

また、Java EE のアプリケーションサーバー上の EntityManagerFactory を使用する場合は、persistence.xml に、設定が必要となると思われるので、Java EE のアプリケーションサーバー上の EntityManagerFactory を使用する場合の設定については、今後整備する予定である。

Spring Data JPA を有効化するための設定

- xxx-infra.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:jpa="http://www.springframework.org/schema/data/jpa"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=".....
    http://www.springframework.org/schema/data/jpa
    https://www.springframework.org/schema/data/jpa/spring-jpa.xsd"> <!-- (1)␣
↪-->

    <!-- ... -->

</beans>
```

```
<jpa:repositories base-package="xxxxxx.yyyyyy.zzzzzz.domain.repository" /> <!-
↪- (2) -->
```

項番	説明
(1)	Spring Data JPA のコンフィギュレーション用のスキーマ定義を取り込み、ネームスペースとして (jpa) を付与する。
(2)	Repository インタフェースおよびカスタム Repository クラスが格納されているベースパッケージを指定する。 org.springframework.data.repository.Repository を継承しているインタフェースと、org.springframework.data.repository.RepositoryDefinition アノテーションが付与されているインタフェースが、 Spring Data JPA の Repository クラスとして自動的に bean 定義される。

- <jpa:repositories>要素の属性について

  属性として、entity-manager-factory-ref、transaction-manager-ref、named-queries-location、query-lookup-strategy、factory-class、repository-impl-postfix が存在する。

項番	要素	説明
1.	entity-manager-factory-ref	Repository で使用する `EntityManager` を生成するための Factory を指定する。 通常指定する必要はないが、 `EntityManager` の Factory を複数用意する場合は、使用する bean を指定する必要がある。
2.	transaction-manager-ref	Repository のメソッドが呼び出された際に使用する `PlatformTransactionManager` を指定する。 デフォルトは `transactionManager` という bean 名で登録されている bean が使用される。 使用する `PlatformTransactionManager` の bean 名が `transactionManager` でない場合は指定が必要である。
3.	named-queries-location	Named Query が指定されている Spring Data JPA のプロパティファイルのロケーションを指定する。 デフォルトは「classpath:META-INF/jpa-named-queries.properties」が使用される。
4.	query-lookup-strategy	Query メソッドが呼び出された特に実行する Query を Lookup する方法を指定する。 デフォルトは `CREATE_IF_NOT_FOUND` となっている。詳細は、Spring Data Commons - Reference Documentation の 「Query lookup strategies」を参照されたい。 特に理由がない場合は、デフォルトのままでよい。
5.	factory-class	Repository インタフェースのメソッドが呼び出された際の処理を実装するクラスを生成するための Factory を指定する。 デフォルトでは、 `org.springframework.data.jpa.repository.support.JpaRepositoryFactory` が使用される。 Spring Data JPA のデフォルト実装を変更する場合や、新しいメソッドを追加する場合に作成した Factory を指定する。 新しいメソッドを追加する方法については、*すべての Repository インタフェースに一括で追加する*を参照されたい。

Continued on next page

表 6.16 – continued from previous page

項番	要素	説明
6.	repository-impl-postfix	カスタム Repository の実装クラスであることを表す接尾辞を指定する。 デフォルトは `Impl` となっている。例えば、Repository インタフェースの名前が `OrderRepository` の場合は、 `OrderRepositoryImpl` がカスタム Repository の実装クラスとなる。特に理由がない場合は、デフォルトのままでよい。 カスタム Repository については、「*Entity 毎の Repository インタフェースに個別に追加する*」を参照されたい。

JPA のアノテーションを使用するための設定

JPA から提供されているアノテーション（`javax.persistence.PersistenceContext` と、`javax.persistence.PersistenceUnit`）を使用して、`javax.persistence.EntityManagerFactory` と `javax.persistence.EntityManager` を Inject するためには、`org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor` を bean 定義する必要がある。

`<jpa:repositories>` 要素を指定した場合、デフォルトで bean が定義されるため、bean 定義の必要はない。

JPA の例外を DataAccessException に変換するための設定

JPA の例外を Spring Framework から提供されている `DataAccessException` に変換するためには、`org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor` を bean 定義する必要がある。

`<jpa:repositories>` 要素を指定した場合、デフォルトで bean が定義されるため、bean 定義の必要はない。

OpenEntityManagerInViewInterceptor の設定

Controller や JSP 等のアプリケーション層で Entity の Lazy Fetch を行うためには、`org.springframework.orm.jpa.support.OpenEntityManagerInViewInterceptor` を使用して、`EntityManager` の生存期間をアプリケーション層まで伸ばす必要がある。

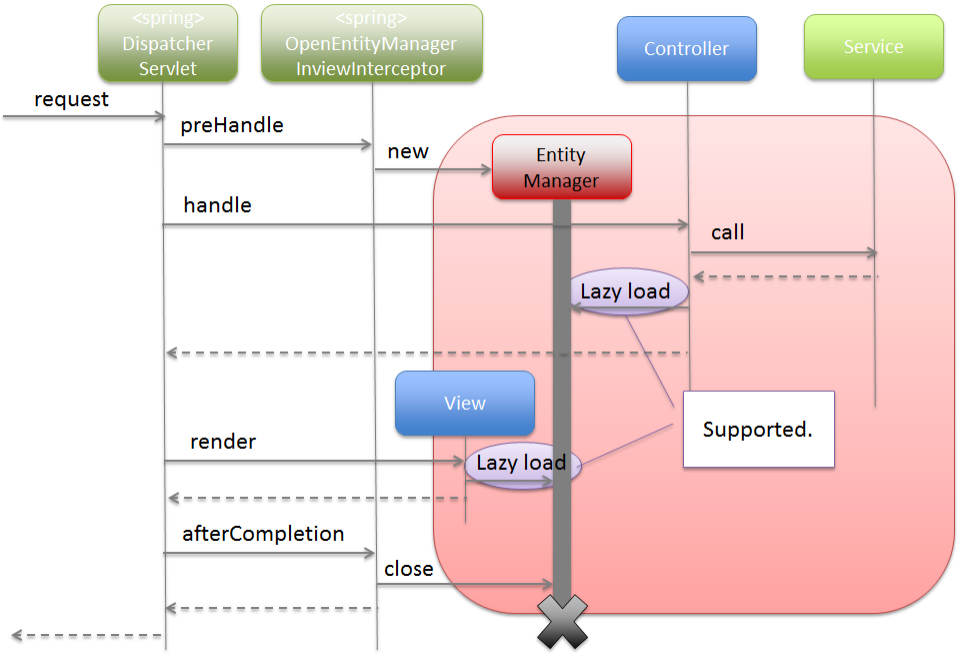


図 6.23 **Picture - Lifetime of EntityManager on OpenEntityManagerInViewInterceptor**

`OpenEntityManagerInViewInterceptor` を使用しない場合は、 `EntityManager` の生存期間はトランザクションと同じになるため、アプリケーション層で必要となるデータを Service クラスの処理として Fetch するか、Lazy Fetch を使わずに Eager Fetch を使用する必要がある。

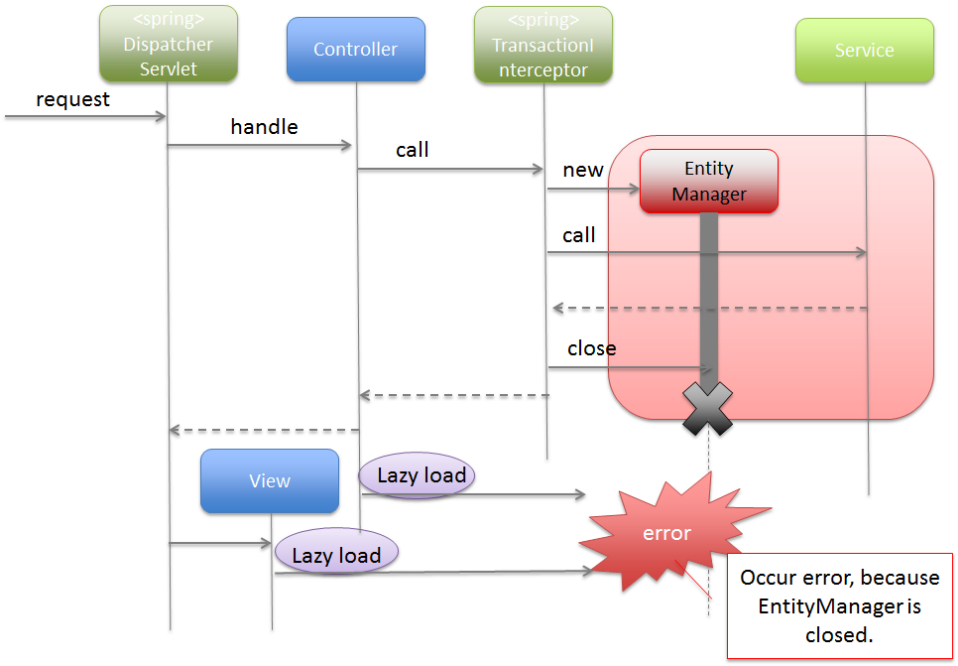


図 6.24 **Picture - Default Life time of EntityManager**

下記の点から、基本的には Fetch 方法は Lazy Fetch として、`OpenEntityManagerInViewInterceptor` を使用することを推奨する。

- Service クラスの処理として Fetch した場合、getter メソッドを呼び出すだけの処理や getter メソッドへアクセスしたコレクションへのアクセスなど、一見意味のない処理を実装することになってしまう。
- Eager Fetch にした場合、アプリケーション層で使用しないデータへの Fetch も行われる可能性があるため、性能に影響を与える可能性がある。

以下に `OpenEntityManagerInViewInterceptor` の設定例を示す。

- spring-mvc.xml

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**" /> <!-- (1) -->
        <mvc:exclude-mapping path="/resources/**" /> <!-- (1) -->
        <!-- (2) -->
        <bean
            class="org.springframework.orm.jpa.support.
↪OpenEntityManagerInViewInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>
```

項番	説明
(1)	Interceptor を適用するパスと、除外パスを指定する。 例では、リソースファイル (js、css、image など) のパス以外のリクエストに対して Interceptor を適用している。
(2)	`org.springframework.orm.jpa.support.OpenEntityManagerInViewInterceptor` を指定する。

注釈: 静的リソースへのパスを適用対象外とする

静的リソース (js、css、image、html など) のパスについては、データアクセスが発生しないため、Interceptor の適用外とすることを推奨する。適用対象にしてしまうと、 `EntityManager` に対して無駄な処理（インスタンス生成とクローズ処理）が実行されることになる。

Servlet Filter で Lazy Fetch が必要な場合は、`org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter` を使用して、EntityManager の生存期間を Servlet Filter 層まで伸ばす必要がある。

例えば、 SpringSecurity の `org.springframework.security.core.userdetails.UserDetailsService` を拡張実装し、拡張した処理の中で Entity オブジェクトにアクセスする場合、このケースにあてはまる。

ただし、Lazy Fetch の必要がないのであれば、 EntityManager の生存期間を Servlet Filter 層まで伸ばす必要はない。

注釈: **Servlet Filter 層での Lazy Fetch について**

Servlet Filter 層で Lazy Fetch が発生しないように設計および実装することを推奨する。 OpenEntityManagerInViewInterceptor を使用した方が、 適用パターンと除外パターンを柔軟に指定できるため、 EntityManager の生存期間をアプリケーション層まで伸ばす対象のパスを指定しやすくなる。Servlet Filter で必要となるデータへのアクセスについては、Service クラスの処理として事前に Fetch しておくか、または Eager Fetch を使用して事前にロードしておくことで、Lazy Fetch が発生しないようにする。

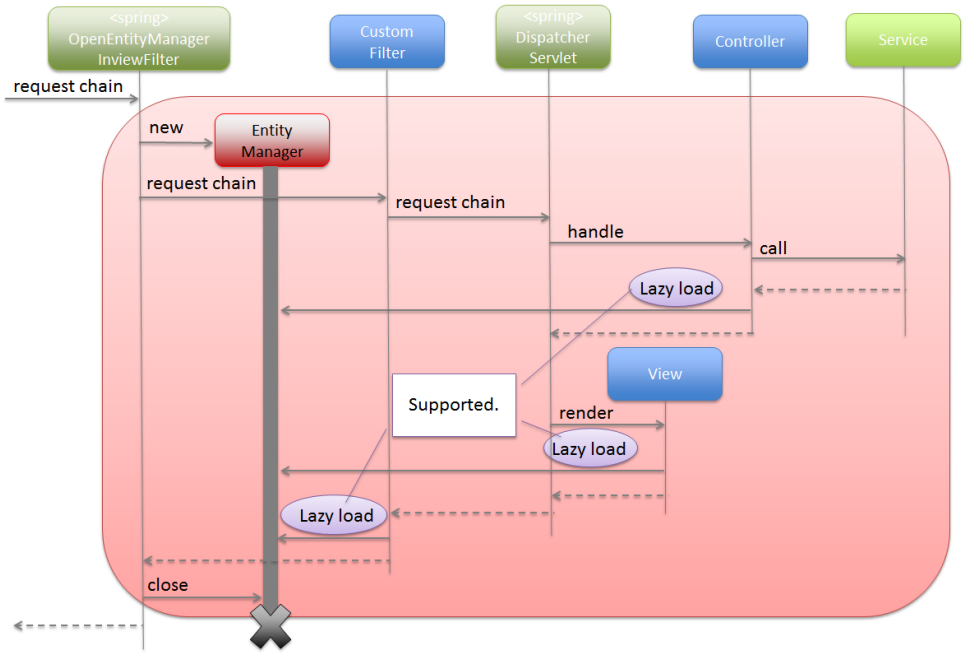


図 6.25 **Picture - Lifetime of EntityManager on OpenEntityManagerInViewFilter**

以下に OpenEntityManagerInViewFilter の設定例を示す。

- web.xml

```
<!-- (1) -->
<filter>
    <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
    <filter-class>org.springframework.orm.jpa.support.
↪OpenEntityManagerInViewFilter</filter-class>
```

```
</filter>
<!-- (2) -->
<filter-mapping>
    <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

項番	説明
(1)	`org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter` を指定する。 この Servlet Filter は、 **Lazy Fetch が発生する Servlet Filter より前に定義する必要がある。**
(2)	フィルタを適用する URL のパターンを指定する。可能な限り必要なパスのみに適用することを推奨するが、設定が煩雑になってしまうのであれば、「/*」（すべてのリクエスト）としてもよい。

注釈: `OpenEntityManagerInViewFilter` を適用する URL のパターンに「/*」（すべてのリクエスト）を指定した場合は、 `OpenEntityManagerInViewInterceptor` の設定は不要となる。

警告: 意図せぬ LazyInitializationException の発生を防止するために

`OpenEntityManagerInViewInterceptor` や `OpenEntityManagerInViewFilter` を利用するかどうかに関わらず、Lazy Fetch が行われないままの Entity を EntityManager の生存期間外まで引き継いでしまうと、意図せぬところで `org.hibernate.LazyInitializationException` が発生する可能性がある。例として、Session や Flash scope に登録後、別リクエストやリダイレクト先で取得する場合が挙げられる。上記のような場合には、あらかじめ EntityManager の生存期間内で Fetch を行い、取得したデータを別オブジェクトに入れ替えて引き継ぐように実装することを推奨する。

Repository インタフェースの作成

Spring Data では Entity 毎の Repository インタフェースを作成する方法として、以下 3 つの方法を提供している。

項番	作成方法	説明
1.	*Spring Data* 提供のインタフェースを継承する	Spring Data から提供されているインタフェースを継承することで、Entity 毎の Repository インタフェースを作成する。**特に理由がない場合は、この方法で、Entity 毎の Repository インタフェースを作成することを推奨する。**
2.	必要なメソッドのみ定義したインタフェースを継承する	Spring Data から提供されている Repository インタフェースのメソッドの中から、必要なメソッドのみ定義したプロジェクト用の共通インタフェースを作成し、作成した共通インタフェースを継承することで Entity 毎の Repository インタフェースを作成する。
3.	インタフェースの継承は行わない	Spring Data から提供されているインタフェースやプロジェクト用の共通インタフェースの継承は行わずに、Entity 毎に Repository インタフェースを作成する。

Spring Data 提供のインタフェースを継承する

Spring Data から提供されているインタフェースを継承して Entity 毎の Repository インタフェースを作成する方法について説明する。

継承することができるインタフェースは以下の通り。

項番	インタフェース	説明
1.	org.springframework.data.repository CrudRepository	汎用的な CRUD 操作を行うメソッドを提供している Repository インタフェース。
2.	org.springframework.data.repository PagingAndSortingRepository	`CrudRepository` の findAll メソッドにページネーション機能とソート機能を追加した Repository インタフェース。
3.	org.springframework.data.jpa.repository JpaRepository	JPA の仕様に依存するメソッドを提供している Repository インタフェース。 `PagingAndSortingRepository` を継承しているため、`PagingAndSortingRepository` および `CrudRepository` のメソッドも使用する事ができる。 **特に理由がない場合は、本インタフェースを継承して Entity 毎の Repository インタフェースを作成することを推奨する。**

注釈: Spring Data 提供の Repository インタフェースのデフォルト実装について

上記インタフェースで定義されているメソッドの実装は、Spring Data JPA より提供されている `org.springframework.data.jpa.repository.support.SimpleJpaRepository` で行われている。

以下に、作成例を示す。

```
public interface OrderRepository extends JpaRepository<Order, Integer> { //␣
↪(1)

}
```

項番	説明
(1)	JpaRepository を継承し、ジェネリック型 <T> に Entity の型、ジェネリック型 <ID extends Serializable> に Entity の ID の型を指定する。 上記例では、Entity に Order 型、Entity の ID に Integer 型を指定している。

JpaRepository を継承して Entity 毎の Repository インタフェースを作成すると、以下のメソッドに対する実装を得ることが出来る。

項番	メソッド	説明
1.	<S extends T> S save(S entity)	指定された Entity に対する永続操作 (INSERT/UPDATE) を javax.persistence.EntityManger に蓄積するためのメソッド。 ID プロパティ (@javax.persistence.Id アノテーションまたは @javax.persistence.EmbeddedId アノテーションが付与されているプロパティ) に値が設定されていない場合は EntityManager の persist メソッドが呼ばれ、値が設定されている場合は merge メソッドが呼び出される。 merge メソッドが呼び出された場合、返却される Entity オブジェクトは、引数で渡された Entity とは別のオブジェクトとなるので注意すること。
2.	<S extends T> List<S> save(Iterable<S> entities)	指定された複数の Entity に対する永続操作を EntityManger に蓄積するためのメソッド。 <S extends T> S save(S entity) メソッドを繰り返し呼び出す事で実現している。

Continued on next page

表 6.17 – continued from previous page

項番	メソッド	説明
3.	T saveAndFlush(T entity)	指定された Entity に対する永続操作を EntityManger に蓄積した後に、蓄積されている永続操作 (INSERT/UPDATE/DELETE) を永続層 (DB) に反映するためのメソッド。
4.	void flush()	EntityManager に蓄積された Entity への永続操作 (INSERT/UPDATE/DELETE) を永続層 (DB) に実行するためのメソッド。
5.	void delete(ID id)	指定された ID の Entity に対する削除操作を EntityManger に蓄積するためのメソッド。 このメソッドは T findOne(ID) メソッドを呼び出して Entity オブジェクトを EntityManger の管理下にしてから削除している。 T findOne(ID) メソッドの呼び出し時に Entity が存在しない場合は、 org.springframework.dao.EmptyResultDataAccessException が発生する。
6.	void delete(T entity)	指定された Entity に対する削除操作を EntityManger に蓄積するためのメソッド。
7.	void delete(Iterable<? extends T> entities)	指定された複数の Entity に対する削除操作を EntityManger に蓄積するためのメソッド。 void delete(T entity) メソッドを繰り返し呼び出す事で実現している。削除対象の Entity が大量になる場合は、 void deleteInBatch(Iterable<T> entities) メソッドを使用した方が効率的に削除することができる。

Continued on next page

表 6.17 – continued from previous page

項番	メソッド	説明
8.	void deleteAll()	すべての Entity に対する削除操作を `EntityManger` に蓄積するためのメソッド。 `List<T> findAll()` メソッドで取得した Entity に対して `void delete(T entity)` メソッドを繰り返し呼び出す事で実現している。 削除対象の Entity が大量になる場合は、 `void deleteAllInBatch()` メソッドを使用すること。このメソッドは削除する Entity をすべてアプリケーション上に読み込むので、メモリ枯渇の原因になる。
9.	void deleteInBatch(Iterable<T> entities)	指定された複数の Entity を直接永続層 (DB) から削除するためのメソッド。 このメソッドを使って削除した場合、 `EntityManager` 上で管理 (キャッシュ) されている Entity は削除されないため、 本メソッドで削除した後に `T findOne(ID id)` メソッドを呼び出すと `EntityManager` 上で管理されている Entity が返却されるので注意が必要。 後続処理で削除した Entity に対して `T findOne(ID id)` メソッドなどの `EntityManager` 上で管理 (キャッシュ) されている Entity オブジェクトを返却するメソッドを呼び出す可能性がある場合は、 `void delete(Iterable<? extends T> entities)` メソッドを使用して削除すること。
10.	void deleteAllInBatch()	すべての Entity を直接永続層 (DB) から削除するためのメソッド。 `void deleteInBatch(Iterable<T> entities)` メソッドと同様、`EntityManager` 上で管理 (キャッシュ) されている Entity は削除されない点に注意すること。

Continued on next page

表 6.17 – continued from previous page

項番	メソッド	説明
11.	T findOne(ID id)	指定された ID の Entity を永続層 (DB) から取得するためのメソッド。 永続層から取得された Entity は `EntityManager` によって管理 (キャッシュ) されるため、2 回目以降のアクセスでは永続層へのアクセスは発生せず、キャッシュされている Entity が返却される。
12.	List<T> findAll()	すべての Entity を永続層 (DB) から取得するためのメソッド。 永続層から取得された Entity は `EntityManager` によって管理 (キャッシュ) される。
13.	Iterable<T> findAll(Iterable<ID> ids)	指定された複数の ID の Entity を永続層 (DB) から取得するためのメソッド。 永続層から取得された Entity は `EntityManager` によって管理 (キャッシュ) される。 指定された ID は IN 句を使って検索されるため、Oracle など IN 句に指定できる値の数に制限がある DB を使う場合は注意すること。
14.	List<T> findAll(Sort sort)	指定された並び順ですべての Entity を永続層 (DB) から取得するためのメソッド。 永続層から取得された Entity は `EntityManager` によって管理 (キャッシュ) される。
15.	Page<T> findAll(Pageable pageable)	指定されたページ (並び順、ページ数、ページ内に表示する件数) に一致する Entity を永続層 (DB) から取得するためのメソッド。 永続層から取得された Entity は `EntityManager` によって管理 (キャッシュ) される。
16.	boolean exists(ID id)	指定された ID の Entity が存在するかチェックするためのメソッド。

Continued on next page

表 6.17 – continued from previous page

項番	メソッド	説明
17.	long count()	永続化対象の Entity の件数を取得するためのメソッド。

警告: JPA の楽観ロック (@javax.persistence.Version) 使用時の動作について

JPA の楽観ロック (`@Version`) 使用時に更新対象の Entity が更新または削除された場合は、`org.springframework.dao.OptimisticLockingFailureException` が発生する。`OptimisticLockingFailureException` が発生する可能性があるメソッドは、以下の通りである。

- <S extends T> S save(S entity)
- <S extends T> List<S> save(Iterable<S> entities)
- T saveAndFlush(T entity)
- void delete(ID id)
- void delete(T entity)
- void delete(Iterable<? extends T> entities)
- void deleteAll()
- void flush()

JPA の楽観ロックの詳細については 排他制御 を参照されたい。

注釈: 永続操作の反映タイミングについて (その 1)

`EntityManager` に蓄積された Entity への永続操作は、トランザクションをコミットする直前に実行され永続層 (DB) に反映される。そのため、一意制約違反などのエラーをトランザクション管理内の処理 (Service の処理) でハンドリングしたい場合は、 `saveAndFlush` メソッドまたは `flush` メソッドを呼び出して `EntityManager` 内に蓄積されている Entity への永続操作を強制的に実行する必要がある。単にエラーをクライアントに通知するだけでよければ、Controller で例外ハンドリングを行い適切なメッセージを設定すればよい。

`saveAndFlush` メソッドおよび `flush` メソッドは JPA 依存のメソッドなので、意図なく使用しないように注意すること。

- 通常のフロー

- flush 時のフロー

注釈: 永続操作の反映タイミングについて (その 2)

以下のメソッドを呼び出した場合、`EntityManager` と、永続層 (DB) で管理しているデータの不整

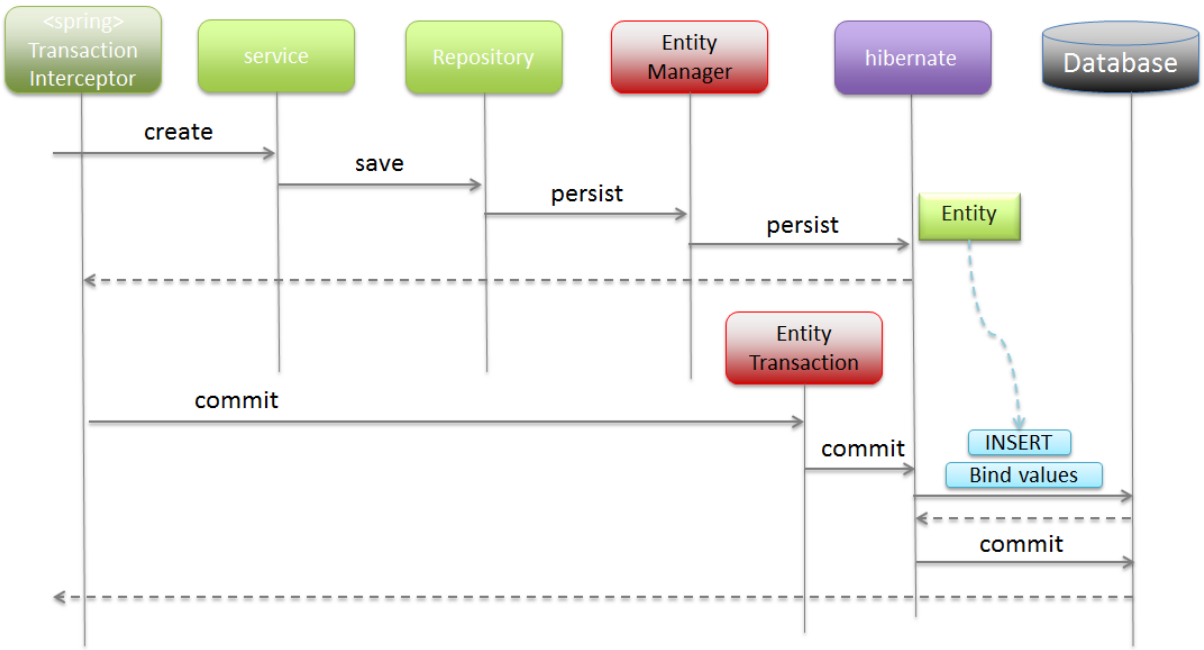


図 6.26 **Picture - Normal sequence of persistence processing**

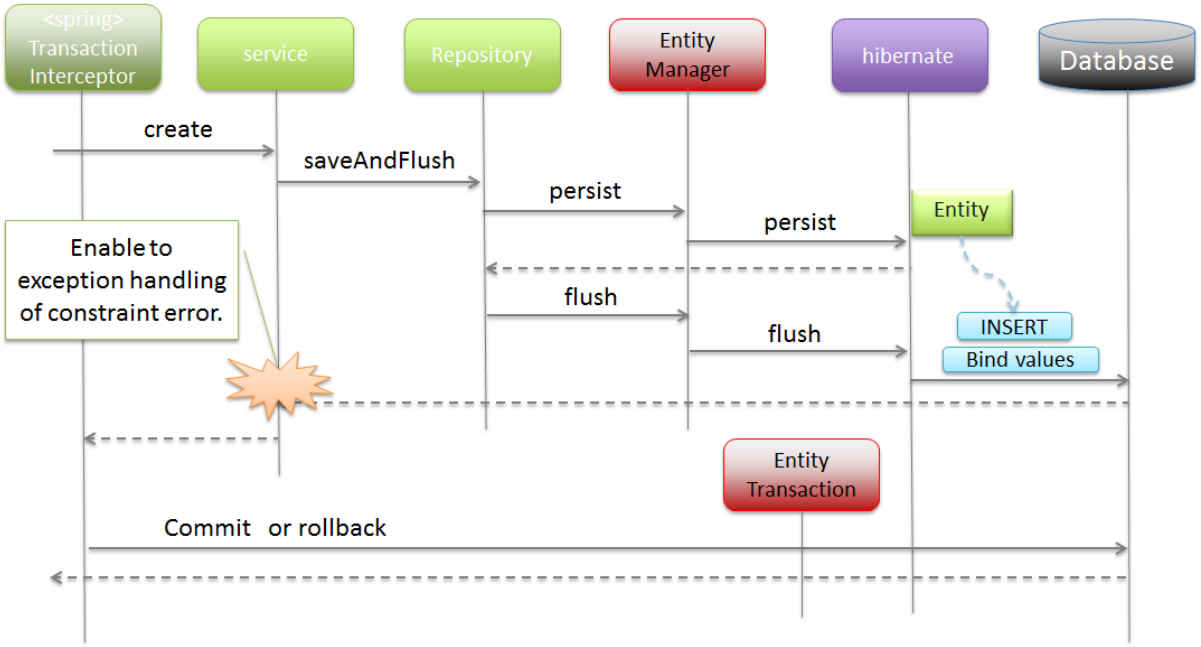


図 6.27 **Picture - Sequence of persistence processing when using flush method**

合が発生しないようにするために、メインの処理が行われる前に `EntityManager` に蓄積されている Entity への永続操作が永続層 (DB) に反映される。

- `List<T> findAll` 系メソッド
- `boolean exists(ID id)`
- `long count()`

上記のメソッドは、永続層 (DB) に直接 Query を発行するため、 メインの処理が行われる前に永続層

(DB) に反映されないとデータの不整合が発生することになる。後述する Query メソッドを呼び出した際も、 `EntityManager` に蓄積されていた Entity への永続操作が、永続層 (DB) に反映されるトリガーとなる。

- Query 発行時のフロー

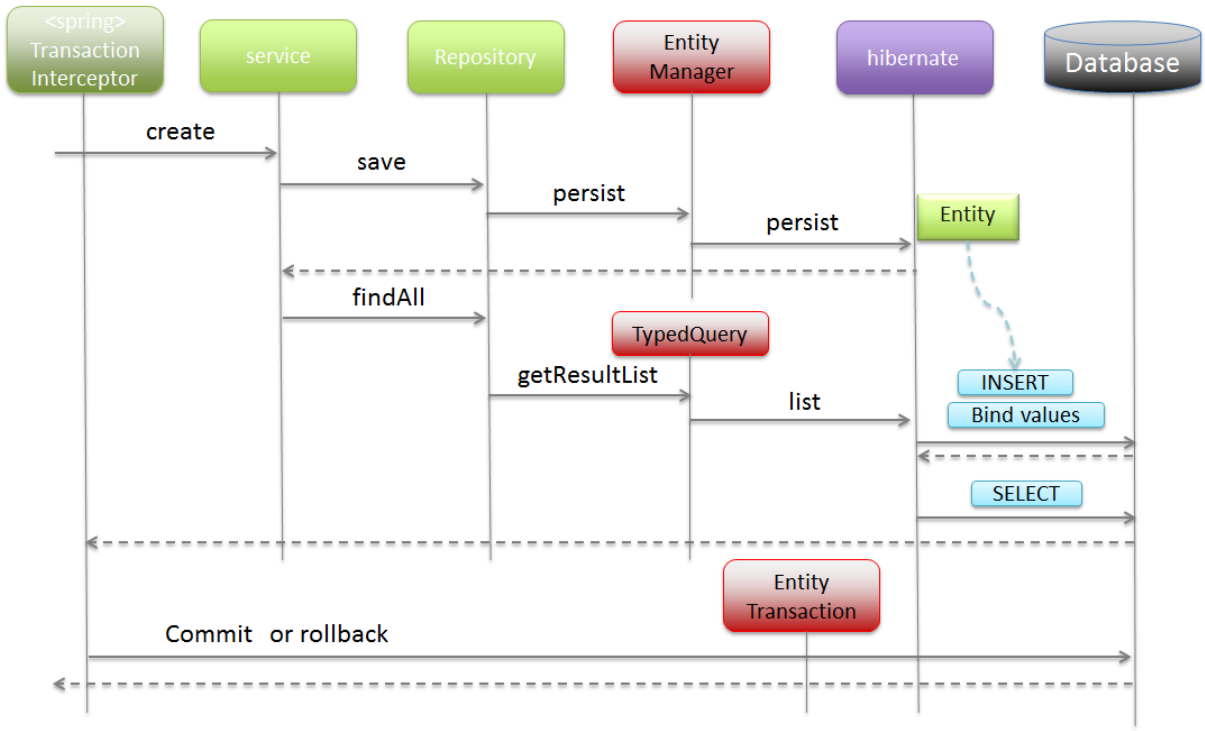


図 6.28 **Picture - Sequence of persistence processing when using query method**

必要なメソッドのみ定義したインタフェースを継承する

Spring Data から提供されているインタフェースに定義されているメソッドの中から、必要なメソッドのみ定義した共通インタフェースを作成し継承する事で Entity 毎の Repository インタフェースを作成する方法について説明する。

メソッドのシグネチャを Spring Data から提供されている Repository インタフェースのメソッドと一致させる必要があるが、Spring Data 提供の Repository インタフェースを継承して作成した際と同様、メソッドの実装は不要である。

注釈: 想定される適用ケース

Spring Data から提供されている Repository インタフェースのメソッドの中には、実際のアプリケーションでは使用しないまたは使用しない方がよいメソッドもある。そのようなメソッドを Repository インタフェースから削除したい場合は、この方法で作成する。インタフェースに定義したメソッドの実装は、Spring Data JPA より提供されている `org.springframework.data.jpa.repository.`

support.SimpleJpaRepository で行われている。

以下に、作成例を示す。

```
@NoRepositoryBean // (1)
public interface MyProjectRepository<T, ID extends Serializable> extends
        Repository<T, ID> { // (2)

    T findOne(ID id); // (3)

    T save(T entity); // (3)

    // ...

}
```

```
public interface OrderRepository extends MyProjectRepository<Order, Integer>
↪{ // (4)

}
```

項番	説明
(1)	@NoRepositoryBean を指定し、Spring Data による Repository インターフェースのインスタンス化対象から外す。
(2)	org.springframework.data.repository.Repository を継承しプロジェクト用の汎用インタフェースを作成する。 汎用インタフェースなので、ジェネリック型を使用する。
(3)	Spring Data から提供されている Repository インタフェースのメソッドの中から必要なメソッドを選んで定義する。
(4)	プロジェクト用の汎用インタフェースを継承し、ジェネリック型 <T> に Entity の型、ジェネリック型 <ID extends Serializable> に Entity の ID の型を指定する。例では、Entity に Order 型、Entity の ID に Integer 型を指定している。

インタフェースの継承は行わない

Spring Data より提供されているインタフェースや共通インタフェースを継承しないで、Entity 毎の Repository インタフェースを作成する方法について、説明する。

クラスアノテーションとして`@org.springframework.data.repository.RepositoryDefinition`アノテーションを指定し、 domainClass 属性に Entity の型を、idClass 属性に Entity の ID の型を指定する。

Spring Data から提供されている Repository インタフェースに定義されているメソッドと同じシグネチャのメソッドについては、Spring Data 提供の Repository インタフェースを継承して作成した際と同様、メソッドの実装は不要である。

注釈: 想定される適用ケース

共通的な Entity の操作が必要ない場合は、この方法で作成してもよい。Spring Data から提供されている Repository インタフェースに定義されているメソッドと同じシグネチャのメソッドの実装は、Spring Data JPA より提供されている `org.springframework.data.jpa.repository.support.SimpleJpaRepository` で行われている。

以下に、作成例を示す。

```
@RepositoryDefinition(domainClass = Order.class, idClass = Integer.class) //(1)
public interface OrderRepository { //(2)

    Order findOne(Integer id); // (3)

    Order save(Order entity); // (3)

    // ...
}
```

項番	説明
(1)	`@RepositoryDefinition` アノテーションを指定する。 例では、domainClass 属性 (Entity の型) に `Order` 型、idClass 属性 (Entity の ID の型) に `Integer` 型を指定している。
(2)	Spring Data から提供されているインタフェース (`org.springframework.data.repository.Repository`) の継承は不要である。
(3)	Entity 毎に必要なメソッドを定義する。

Query メソッドの追加

Spring Data より提供されている汎用的な CRUD 操作を行うためのインタフェースだけでは、実際のアプリケーションを構築する事は難しい。

そのため Spring Data では、Entity 毎の Repository インタフェースに対して任意の永続操作 (SELECT/UPDATE/DELETE) を行うための Query メソッドを追加できる仕組みを提供している。

追加した Query メソッドでは、Query 言語 (JPQL または Native な SQL) を使用して Entity の操作を行う。

注釈: JPQL とは

JPQL とは」Java Persistence Query Language」の略で、永続層 (DB) のレコードに対応する Entity を操作 (SELECT/UPDATE/DELETE) するための Query 言語である。文法は SQL に似ているが、永続層 (DB) のレコードを直接操作するのではなく、永続層のレコードにマッピングされている Entity を操作することになる。Entity に対して行った操作の永続層 (DB) への反映は、JPA プロバイダ (Hibernate) によって行われる。

JPQL の詳細については、JSR 338: Java Persistence API, Version 2.1 の Specification(PDF)「Chapter 4 Query Language」を参照されたい。

Query メソッドを定義する

Query メソッドは、Entity 毎の Repository インタフェースのメソッドとして定義する。

```
public interface OrderRepository extends JpaRepository<Order, Integer> {
    List<Order> findByStatusCode(String statusCode);
}
```

実行する Query を指定する

Query メソッド呼び出し時に実行する Query を指定する必要がある。

指定方法は以下の通り。詳細は、*Query メソッドの Query 指定*を参照されたい。

項番	Query の指定方法	説明
1.	*@Query アノテーション* (Spring Data の機能)	Entity 毎の Repository インタフェースに追加したメソッドに `@org.springframework.data.jpa.repository.Query` アノテーションを指定し、実行する Query を指定する。 **特に理由がない場合は、この方法で指定することを推奨する。**
2.	命名規約ベースのメソッド名 (Spring Data の機能)	Spring Data が定めた命名規約に則りメソッド名を付与することで実行する Query を指定する。 Spring Data JPA の機能によってメソッド名から実行する Query(JPQL) が生成される。生成できる JPQL は SELECT のみとなっている。 **条件が少なくシンプルな Query の場合は、 @Query アノテーションを使わずにこの方法を使ってもよい。** ただし、条件が多く複雑な Query の場合は、メソッド名は振る舞いを表すシンプルな名前にして `@Query` アノテーションで Query を指定すること。
3.	*プロパティファイルの Named query* (Spring Data の機能)	Spring Data JPA から提供されているプロパティファイルに実行する Query を指定する。 **メソッド定義と Query 指定を行う箇所が分離してしまうので、基本的にはこの方法での指定は推奨しない。** **ただし、Query として Native な SQL を使用する場合は、データベースに依存する SQL をプロパティファイルに定義する必要があるか確認すること。** 使用するデータベースを任意に選択できるアプリケーションや、実行環境によって用意できるデータベースが変わる (変わる可能性がある) 場合には、この方法で Query を指定し、プロパティファイルを環境依存資材として管理する必要がある。

注釈: Query 指定方法の併用について

複数の Query の指定方法を併用することに対して、特に制限は設けない。プロジェクトで使用する指定方法や併用の制限については、プロジェクト毎に判断すること。

注釈: **Query の Lookup 方法について**

Spring Data デフォルトの動作は、 `CREATE_IF_NOT_FOUND` に設定されているため、以下の動作となる。

1. `@Query` アノテーションに指定されている Query を取得し、指定があればその Query を使用する。
2. Named query から対応する Query を取得し、対応する Query が見つかった場合はその Query を使用する。
3. メソッド名から Query(JPQL) を作成して使用する。
4. メソッド名から Query(JPQL) が作成できない場合は、エラーとなる。

Query の Lookup 方法の詳細については、 Spring Data Commons - Reference Documentation「Defining query methods」の「Query lookup strategies」を参照されたい。

Entity のロックを取得する

Entity のロックを取得する必要がある場合は、Query メソッドに `@org.springframework.data.jpa.repository.Lock` アノテーションを追加し、ロックモードを指定する。

詳細については、 排他制御 を参照されたい。

```
@Query(value = "SELECT o FROM Order o WHERE o.status.code = :statusCode ORDER
↪BY o.id DESC")
@Lock(LockModeType.PESSIMISTIC_WRITE) // (1)
List<Order> findByStatusCode(@Param("statusCode") String statusCode);
```

```
-- (2) statusCode='accepted'
SELECT
        order0_.id AS id1_5_
        ,order0_.status_code AS status2_5_
    FROM
        t_order order0_
    WHERE
        order0_.status_code = 'accepted'
    ORDER BY
        order0_.id DESC
    FOR UPDATE
```

項番	説明
(1)	`@Lock` アノテーションの value 属性にロックモードを指定する。 指定可能なロックモードについては、Java Platform, Enterprise Edition API Specification を参照されたい。
(2)	JPQL から変換された Native な SQL。(使用 DB は PostgreSQL) 例では、`LockModeType.PESSIMISTIC_WRITE` を指定しているので、SQL に」FOR UPDATE」句が追加される。

永続層の Entity を直接操作する

Entity の更新および削除の操作は、原則 `EntityManager` 上で管理されている Entity オブジェクトに対して行うことを推奨する。

ただし、Entity を一括で更新または削除する必要がある場合は、Query メソッドを使って永続層 (DB) の Entity を直接操作することを検討すること。

注釈: 性能劣化の要因軽減

永続層の Entity を直接操作することで、Entity の操作を行うための SQL の発行回数を減らすことができる。そのため、高い性能要件があるアプリケーションの場合は、この方法で Entity の一括操作を行うことで、性能劣化の要因を減らす事ができる。減らすことが出来る SQL は以下の通り。

- Entity オブジェクトを `EntityManager` 上に読み込むための SQL。発行が不要となる。
- Entity を更新および削除するための SQL。n 回の発行が必要だったものが 1 回の発行で済む。

注釈: 永続層の Entity を直接操作するかの判断基準について

永続層の Entity を直接操作する場合、機能的な注意点がいくつかあるため、 **性能要件が高くないアプリケーションの場合は、一括操作についても EntityManager 上で管理されている Entity オブジェクトに対して行うことを推奨する。**具体的な注意点については、実装例を参照されたい。

以下に、Query メソッドを使って、永続層の Entity を直接操作する実装例を示す。

```
@Modifying // (1)
@Query("UPDATE OrderItem oi SET oi.logicalDelete = true WHERE oi.id.orderId =␣
↪:orderId ") // (2)
int updateToLogicalDelete(@Param("orderId") Integer orderId); // (3)
```

項番	説明
(1)	更新系の Query メソッドであることを示す `@org.springframework.data.jpa.repository.Modifying` アノテーションを指定する。 指定しないと実行時にエラーとなる。
(2)	更新系 (UPDATE または DELETE) 用の Query を指定する。
(3)	更新件数や削除件数が必要な場合は、`int` または `java.lang.Integer` を戻り値の型として指定し、件数が必要ない場合は、 `void` を指定する。

警告: EntityManager 上で管理している Entity との整合性について

Query メソッドを使って永続層の Entity を直接操作した場合、Spring Data JPA のデフォルト動作では EntityManager 上で管理されている Entity に反映されない。そのため、直後に `JpaRepository#findOne(ID)` メソッドを呼び出して取得される Entity オブジェクトは、操作前の状態である点に注意すること。

この動作を回避する方法として、 `@Modifying` アノテーションの clearAutomatically 属性を `true` に指定する方法がある。clearAutomatically 属性に `true` を指定した場合、永続層の Entity を直接操作した後に、`EntityManager` の `clear()` メソッドが呼び出され、`EntityManager` 上で管理されていた Entity オブジェクトと蓄積されていた永続操作が `EntityManager` 上から破棄される。そのため、直後に `JpaRepository#findOne(ID)` メソッドを呼び出した場合、永続層から最新状態の Entity が取得され、永続層と `EntityManager` の状態が同期される仕組みになっている。

警告: @Modifying(clearAutomatically = true) 使用時の注意点

`@Modifying(clearAutomatically = true)` とすることで、蓄積されていた永続操作 (INSERT/UPDATE/DELETE) も `EntityManager` 上から破棄されてしまうという点に注意が必要となる。これは、必要な永続操作が永続層に反映されない可能性がある事を意味するため、バグを引き起こす要因となりうる。

この問題を回避するためには、永続層の Entity を直接操作する前に `JpaRepository#saveAndFlush(T entity)` または `JpaRepository#flush()` メ

ソッドを呼び出し、蓄積されている永続操作を永続層に反映しておく必要がある。

Query ヒントを設定する

Query にヒントを設定する必要がある場合は、Query メソッドに `@org.springframework.data.jpa.repository.QueryHints` アノテーションを追加し、value 属性に Query ヒント (`@javax.persistence.QueryHint`) を指定する。

```
@Query(value = "SELECT o FROM Order o WHERE o.status.code = :statusCode ORDER␣
↪BY o.id DESC")
@Lock(LockModeType.PESSIMISTIC_WRITE)
@QueryHints(value = { @QueryHint(name = "javax.persistence.lock.timeout",␣
↪value = "0") }) // (1)
List<Order> findByStatusCode(@Param("statusCode") String statusCode);
```

項番	説明
(1)	`@QueryHint` アノテーションの name 属性にヒント名、value 属性にヒント値を設定する。 指定できるヒントは、JPA の仕様で決められているものに加え、プロバイダ固有のものがある。 上記例では、ロックタイムアウトを 「0」 に設定している (使用 DB は Oracle)。SQL に」FOR UPDATE NOWAIT」句が追加される。

注釈: Hibernate で指定できる Query ヒントについて

JPA 仕様で決められている Query ヒントは以下の通り。詳細は、JSR 338: Java Persistence API, Version 2.1 の Specification(PDF) を参照されたい。

- `javax.persistence.query.timeout`
- `javax.persistence.lock.timeout`
- `javax.persistence.cache.retrieveMode`
- `javax.persistence.cache.storeMode`

Hibernate 固有の Query ヒントについては、Hibernate EntityManager Reference guide の「3.4.1.7. Query hints」を参照されたい。

Query メソッドの Query 指定

Query メソッド呼び出し時に実行する Query の指定方法について説明する。

- *@Query アノテーションで指定する*

- 命名規約ベースのメソッド名で指定する
- プロパティファイルに *Named query* として指定する

@Query アノテーションで指定する

`@Query` アノテーションの value 属性に実行する Query(JPQL) を指定する。

```
@Query(value = "SELECT o FROM Order o WHERE o.status.code = :statusCode ORDER␣
↪BY o.id DESC") // (1)
List<Order> findByStatusCode(@Param("statusCode") String statusCode);
```

```
-- (2) statusCode='accepted'
SELECT
        order0_.id AS id1_5_
        ,order0_.status_code AS status2_5_
    FROM
        t_order order0_
    WHERE
        order0_.status_code = 'accepted'
    ORDER BY
        order0_.id DESC
```

項番	説明
(1)	`@Query` アノテーションの value 属性に実行する Query(JPQL) を指定する。 上記例では、`Order` オブジェクトで保持している `status` プロパティ (`OrderStatus` 型) の `code` プロパティ (`String` 型) の値が指定したパラメータ値 (`statusCode`) と一致する `Order` オブジェクトを `id` プロパティの降順に並べて取得するための Query を指定している。
(2)	JPQL から変換された Native な SQL。`@Query` アノテーションの value 属性に指定した Query(JPQL) は、使用するデータベースの Native な SQL に変換され実行される。

注釈: JPQL ではなく Native な SQL を直接指定する方法

Query として JPQL ではなく Native な SQL を直接指定したい場合は、 nativeQuery 属性を `true` に設定することで指定可能となる。**基本的には JPQL を使用する事を推奨するが、JPQL で表現できない Query を発行する必要がある場合は Native な SQL を直接指定してもよい。**データベースに依存する SQL を指定する場合は、SQL をプロパティファイルに定義することを検討すること。

SQL をプロパティファイルに定義する方法については、「プロパティファイルに *Named query* として指定する」を参照されたい。

注釈: **Named Parameters** について

Query にバインドするパラメータに対して名前を付与し、Query 内からはパラメータ名を指定することで値をバインドすることができる。Named Parameter を使用する場合は、 `@org.springframework.data.repository.query.Param` アノテーションを対象とする引数に追加し、value 属性にパラメータ名を指定する。Query では、バインドしたい位置に「:パラメータ名」の形式で指定する。

特に理由がない場合は、 **Query** のメンテナンス性と可読性を考慮し、**Named Parameters** を使用することを推奨する。

警告: **Query(JPQL)** 解釈のバグについて

Hibernate 5.2.10 以前においては、Query(JPQL) に余分な閉じ括弧があった場合、それ以降が無視された不完全な Query(JPQL) が実行されてしまうバグが存在する。TERASOLUNA Server Framework for Java 5.4.2 では Hibernate 5.0.12 を使用しているため、バグの影響を受け不完全な SQL が実行される恐れがある。不完全な SQL が実行されたことをログ等から確認することが困難であるため、コーディング時に十分注意されたい。

LIKE 検索の一致方法 (前方一致、後方一致、部分一致) が固定の場合は、JPQL 内に 「`%`」 を指定することが出来る。

ただし、これは JPQL の標準形式ではなく Spring Data JPA の拡張形式になるので、`@Query` アノテーションで指定する JPQL でのみ指定することが出来る。

Named query として指定する JPQL 内に 「`%`」 を指定するとエラーになるので注意すること。

項番	一致方法	形式	具体例
1.	前方一致	:parameterName% or ?n%	SELECT a FROM Account WHERE a.firstName LIKE :firstName% SELECT a FROM Account WHERE a.firstName LIKE ?1%
2.	後方一致	%:parameterName or %?n	SELECT a FROM Account WHERE a.firstName LIKE %:firstName SELECT a FROM Account WHERE a.firstName LIKE %?1
3.	部分一致	%:parameterName% or %?n%	SELECT a FROM Account WHERE a.firstName LIKE %:firstName% SELECT a FROM Account WHERE a.firstName LIKE %?1%

注釈: LIKE 検索時のエスケープについて

LIKE 検索を行う場合は、検索条件となる値を LIKE 検索用にエスケープする必要がある。

`org.terasoluna.gfw.common.query.QueryEscapeUtils` クラスにエスケープするためのメソッドが用意されているため、要件を充たせる場合は使用を検討すること。`QueryEscapeUtils` クラスの詳細については、「データベースアクセス（共通編）」の「*LIKE* 検索時のエスケープについて」を参照されたい。

注釈: 一致方法を動的に変化させる必要がある場合

一致方法 (前方一致、後方一致、部分一致) を動的に変化させる必要がある場合は、JPQL 内に 「`%`」 を指定するのではなく、従来通りバインドするパラメータ値の前後に 「`%`」 を追加すること。

`org.terasoluna.gfw.common.query.QueryEscapeUtils` クラスに一致方法に対応する検索条件値に変換するメソッドが用意されているため、要件を充たせる場合は、使用を検討すること。`QueryEscapeUtils` クラスの詳細については、「データベースアクセス（共通編）」の「*LIKE* 検索時のエスケープについて」を参照されたい。

ソート条件は、Query 内に直接指定することができる。

以下に、実装例を示す。

```
// (1)
@Query(value = "SELECT o FROM Order o WHERE o.status.code = :statusCode ORDER␣
↪BY o.id DESC")
Page<Order> findByStatusCode(@Param("statusCode") String statusCode, Pageable␣
↪pageable);
```

項番	説明
(1)	Query に ORDER BY を指定する。降順にする場合は DESC を、昇順にする場合は ASC を指定する。DESC/ASC を省略した場合は、 ASC が適用される。

ソート条件は Query 内に直接指定する以外に、 Pageable オブジェクト内に保持している org.springframework.data.domain.Sort オブジェクトに指定することが出来る。

この方法でソート条件を指定する場合は、countQuery 属性の指定は不要。

以下に、Pageable オブジェクト内に保持している Sort オブジェクトを使用してソートする実装例を示す。

- Controller

```
@RequestMapping("list")
public String list(@PageableDefault(
                        size=5,
                        sort = "id", // (1)
                        direction = Direction.DESC // (1)
                        ) Pageable pageable,
                          Model model) {
    Page<Order> orderPage = orderService.getOrders(pageable); // (2)
    model.addAttribute("orderPage", orderPage);
    return "order/list";
}
```

項番	説明
(1)	ソート条件を指定する。 `Pageable#getSort()` メソッドで取得できる `Sort` オブジェクトにソート条件が設定される。 上記例では、 id フィールドの降順をソート条件として指定している。
(2)	`Pageable` オブジェクトを指定して Service のメソッドを呼び出す。

- Service (Caller)

```
public String getOrders(Pageable pageable){
    return orderRepository.findByStatusCode("accepted", pageable); // (3)
}
```

項番	説明
(3)	Controller から渡された `Pageable` オブジェクトを指定して Repository のメソッドを呼び出す。

- Repository インタフェース

```
@Query(value = "SELECT o FROM Order o WHERE o.status.code = :statusCode") //
↪(4)
Page<Order> findByStatusCode(@Param("statusCode") String statusCode, Pageable
↪pageable);
```

```
-- (5) statusCode='accepted'
SELECT
        COUNT(order0_.id) AS col_0_0_
    FROM
        t_order order0_
    WHERE
        order0_.status_code = 'accepted'
```

```
-- (6) statusCode='accepted'
SELECT
        order0_.id AS id1_5_
        ,order0_.status_code AS status2_5_
    FROM
        t_order order0_
    WHERE
        order0_.status_code = 'accepted'
    ORDER BY
        order0_.id DESC
    LIMIT 5
```

項番	説明
(4)	Query に」ORDER BY」句の指定は行わない。countQuery 属性の指定も不要。
(5)	JPQL から変換された件数カウント用の Native な SQL。
(6)	JPQL から変換された「指定されたページ位置の Entity を取得する」ための Native な SQL。 Query に指定はしていないが、`Pageable` オブジェクト内に保持している `Sort` オブジェクトに指定した条件で」ORDER BY」句が追加される。例では、PostgreSQL 用の SQL になっている。

命名規約ベースのメソッド名で指定する

Spring Data が定めた命名規約に則ったメソッド名にすることで実行する Query(JPQL) を指定する。

Spring Data JPA の機能によってメソッド名から JPQL が生成される。

ただし、メソッド名から JPQL を作成できるのは SELECT のみで、UPDATE および DELETE の JPQL は生成できない。

メソッド名から JPQL を生成するための命名規約などのルールについては、以下のページを参照されたい。

項番	参照ページ	説明
1.	Spring Data Commons - Reference Documentation「Defining query methods」の「Query creation」	Distinct、ORDER BY、Case insensitive の指定方法などが記載されている。
2.	Spring Data Commons - Reference Documentation「Defining query methods」の「Property expressions」	ネストされた Entity のプロパティを条件に指定する方法などが記載されている。
3.	Spring Data Commons - Reference Documentation「Defining query methods」の「Special parameter handling」	特別なメソッド引数 (`Pageable`、`Sort`) についての説明が記載されている。
4.	Spring Data JPA - Reference Documentation「Query methods」の「Query creation」	JPQL を組み立てるための命名規約 (キーワード) に関する説明が記載されている。
5.	Spring Data Commons - Reference Documentation「Appendix C. Repository query keywords」	JPQL を組み立てるための命名規約 (キーワード) に関する説明が記載されている。

以下に、実装例を示す。

- OrderRepositry.java

```
Page<Order> findByStatusCode(String statusCode, Pageable pageable); // (1)
```

項番	説明
(1)	メソッド名が `^(find|read|get).*By(.+)` のパターンに一致する場合、メソッド名から JPQL を生成する対象のメソッドとなる。 `(.+)` の部分に条件となる Entity のプロパティや操作を示すキーワードを指定する。 例では、`Order` オブジェクトで保持している `status` プロパティ (`OrderStatus` 型) の `code` プロパティ (`String` 型) の値が指定したパラメータ値 (`statusCode`) と一致する `Order` オブジェクトをページ形式で取得している。

- 件数カウント用 Query

```
-- (2) JPQL
SELECT
        COUNT(*)
    FROM
        ORDER AS generatedAlias0
            LEFT JOIN generatedAlias0.status AS generatedAlias1
        WHERE
            generatedAlias1.code = ?1

-- (3) SQL statusCode='accepted'
```

```
SELECT
        COUNT(*) AS col_0_0_
    FROM
        t_order order0_
            LEFT OUTER JOIN c_order_status orderstatu1_
                ON order0_.status_code = orderstatu1_.code
    WHERE
        orderstatu1_.code = 'accepted'
```

項番	説明
(2)	メソッド名から生成された件数カウント用の JPQL の Query。
(3)	(2) の JPQL から変換された件数カウント用の Native な SQL。

- Entity 取得用 Query

```
-- (4) JPQL
SELECT
        generatedAlias0
    FROM
        ORDER AS generatedAlias0
            LEFT JOIN generatedAlias0.status AS generatedAlias1
        WHERE
            generatedAlias1.code = ?1
        ORDER BY
            generatedAlias0.id DESC;

-- (5) statusCode='accepted'
SELECT
        order0_.id AS id1_5_
        ,order0_.status_code AS status2_5_
    FROM
        t_order order0_
            LEFT OUTER JOIN c_order_status orderstatu1_
                ON order0_.status_code = orderstatu1_.code
    WHERE
        orderstatu1_.code = 'accepted'
    ORDER BY
        order0_.id DESC
    LIMIT 5
```

項番	説明
(4)	メソッド名から生成された Entity 取得用の JPQL の Query。
(5)	(4) の JPQL から変換された Entity 取得用の Native な SQL。

プロパティファイルに Named query として指定する

Spring Data JPA から提供されているプロパティファイル (classpath:META-INF/jpa-named-queries.properties) に、実行する Query を指定する。

この方法は、**Query** として **NativeQuery** を使用する際に、データベース固有の **SQL** を記載する必要が出た場合に使用するか検討すること。
データベース固有の **SQL** であっても、実行環境に依存しないのであれば **@Query** アノテーションに直接指定する方法を推奨する。

- OrderRepositry.java

```
@Query(nativeQuery = true)
List<Order> findAllByStatusCode(@Param("statusCode") String statusCode); //␣
↪(1)
```

項番	説明
(1)	Named query の Lookup 名は、Entity のクラス名とメソッド名を 「.」 (dot) で連結したものが使用される。上記例だと、 `Order.findAllByStatusCode` が Lookup 名となる。

ちなみに: Named query の Lookup 名を指定する方法

デフォルトの動作では、Entity のクラス名とメソッド名を 「.」 (dot) で連結したものが Lookup 名として使用されるが、任意のクエリ名を指定することも出来る。

- Entity 取得用の Lookup 名に任意のクエリ名を指定したい場合は、 `@Query` アノテーションの name 属性にクエリ名を指定する。
- ページ検索時の件数カウント用の Lookup 名に任意のクエリ名を指定したい場合は、 `@Query` アノテーションの countName 属性にクエリ名を指定する。

```
@Query(name = "OrderRepository.findAllByStatusCode", nativeQuery = true) //␣
↪(2)
List<Order> findAllByStatusCode(@Param("statusCode") String statusCode);
```

項番	説明
(2)	上記例では、OrderRepository.findAllByStatusCode を Lookup 用のクエリ名として指定している。

- jpa-named-queries.properties

```
# (3)
Order.findAllByStatusCode=SELECT * FROM order WHERE status_code = :statusCode
```

項番	説明
(3)	クエリー名をキーとして、実行する SQL を指定する。 上記例では、Order.findAllByStatusCode をキーに、実行する SQL を指定している。

ちなみに: Spring Data JPA から提供されているプロパティファイルではなく、任意のプロパティファイルに Named Query を指定する方法を以下に紹介する。

- xxx-infra.xml

```
<!-- (4) -->
<jpa:repositories base-package="xxxxxx.yyyyyy.zzzzzz.domain.repository"
    named-queries-location="classpath:META-INF/jpa/jpa-named-queries.
↪properties" />
```

項番	説明
(4)	<jpa:repositories>要素の named-queries-location 属性に、任意のプロパティファイルを指定する。 上記例では、クラスパス上にある `META-INF/jpa/jpa-named-queries.properties` が使用される。

Entity の検索処理の実装

Entity の検索方法について、目的別に説明する。

条件に一致する Entity を全件検索

条件に一致する Entity を全件取得する Query メソッドを呼び出す。

- Repositroy インタフェース

```
public interface AccountRepository extends JpaRepository<Account, String> {

    // (1)
    @Query("SELECT a FROM Account a WHERE :createdDateFrom <= a.createdDate
↪AND a.createdDate < :createdDateTo ORDER BY a.createdDate DESC")
    List<Account> findByCreatedDate(
            @Param("createdDateFrom") Date createdDateFrom,
            @Param("createdDateTo") Date createdDateTo);

}
```

項番	説明
(1)	`java.util.List` インタフェースを返却する Query メソッドを定義する。

- Service

```
public List<Account> getAccounts(Date targetDate) {
    LocalDate targetLocalDate = new LocalDate(targetDate);
    Date fromDate = targetLocalDate.toDate();
    Date toDate = targetLocalDate.dayOfYear().addToCopy(1).toDate();

    // (2)
    List<Account> accounts = accountRepository.findByCreatedDate(fromDate,
            toDate);
    if (accounts.isEmpty()) { // (3)
        // ...
    }
```

```
    return accounts;
}
```

項番	説明
(2)	Repositry インタフェースに実装した Query メソッドを呼び出す。
(3)	検索結果が 0 件の場合は、空のリストが返却される。null は返却されないので null チェックは不要。 必要に応じて、検索結果が 0 件の場合の処理を実装する。

条件に一致する Entity のページ検索

条件に一致する Entity の中から指定ページに該当する Entity を取得する Query メソッドを呼び出す。

- Repositroy インタフェース

```
public interface AccountRepository extends JpaRepository<Account, String> {

    // (1)
    @Query("SELECT a FROM Account a WHERE :createdDateFrom <= a.createdDate␣
↪AND a.createdDate < :createdDateTo")
    Page<Account> findByCreatedDate(
            @Param("createdDateFrom") Date createdDateFrom,
            @Param("createdDateTo") Date createdDateTo, Pageable pageable);

}
```

項番	説明
(1)	引数として `org.springframework.data.domain.Pageable` インタフェースを受け取り、 `org.springframework.data.domain.Page` インタフェースを返却する Query メソッドを定義する。

- Controller

```
@RequestMapping("list")
public String list(@RequestParam("targetDate") Date targetDate,
                   @PageableDefault(
                       page = 0,
                       value = 5,
                       sort = { "createdDate" },
                       direction = Direction.DESC)
                       Pageable pageable, // (2)
                   Model model) {
    Page<Order> accountPage = accountService.getAccounts(targetDate,␣
↪pageable);
    model.addAttribute("accountPage", accountPage);
    return "account/list";
}
```

項番	説明
(2)	Spring Data より提供されているページング検索用のオブジェクト（ org.springframework.data.domain.Pageable ）を生成する。 詳細は「ページネーション」を参照されたい。

- Service

```
public Page<Account> getAccounts(Date targetDate ,Pageable pageable) {

    LocalDate targetLocalDate = new LocalDate(targetDate);
    Date fromDate = targetLocalDate.toDate();
    Date toDate = targetLocalDate.dayOfYear().addToCopy(1).toDate();

    // (3)
    Page<Account> page = accountRepository.findByCreatedDate(fromDate,
            toDate, pageable);
    if (!page.hasContent()) { // (4)
        // ...
    }
    return page;
}
```

項番	説明
(3)	Repositry インタフェースに実装した Query メソッドを呼び出す。
(4)	検索結果が 0 件の場合は、`Page` オブジェクトに空のリストが設定され、 `Page#hasContent()` メソッドの返り値が `false` になる。 必要に応じて、検索結果が 0 件の場合の処理を実装する。

Entity の動的条件による検索処理の実装

動的条件による Entity の検索を行う Query メソッドを Repository メソッドに追加する場合は、 Entity 毎の Repository インタフェースに対して、カスタム Repository インタフェースとカスタム Repository インタフェースの実装クラスを用意する方法で実装する。カスタム Repository インタフェースとカスタム Repository クラスの作成方法については、「*Entity 毎の Repository インタフェースに個別に追加する*」を参照されたい。

以降では、動的条件を適用して Entity を検索する方法について、目的別に説明する。

課題: TBD

今後、以下の内容を追加する予定である。

- QueryDSL を使用した動的 Query の実装例。

動的条件に一致する Entity を全件検索

動的条件に一致する Entity を全件取得する Query メソッドを実装し、呼び出す。

以下の実装例を示す。

実装例では、動的条件として、

- 注文 ID
- 商品名

- 注文状態 (複数指定可能)

を指定可能とし、指定された条件に一致する注文を AND 条件で絞り込む検索とする。なお、条件の指定がない場合は、検索は行わず空のリストを返却する。

- Criteria (JavaBean)

```
public class OrderCriteria implements Serializable { // (1)

    private Integer id;

    private String itemName;

    private List<String> statusCodes;

    // ...

}
```

項番	説明
(1)	検索条件を保持する Criteria オブジェクト (JavaBean) を作成する。

- カスタム Repository インタフェース

```
public interface OrderRepositoryCustom {

    Page<Order> findAllByCriteria(OrderCriteria criteria); // (2)

}
```

項番	説明
(2)	カスタム Repository インタフェースに、Criteria オブジェクトを引数にとり、List を返却するメソッドを定義する。

- カスタム Repository クラス

```
public class OrderRepositoryImpl implements OrderRepositoryCustom { // (3)

    @PersistenceContext
    EntityManager entityManager; // (4)

    public List<Order> findAllByCriteria(OrderCriteria criteria) { // (5)

        // Collect dynamic conditions.
```

```
        // (6)
        final List<String> andConditions = new ArrayList<String>();
        final List<String> joinConditions = new ArrayList<String>();
        final Map<String, Object> bindParameters = new HashMap<String, Object>
↪();

        // (7)
        if (criteria.getId() != null) {
            andConditions.add("o.id = :id");
            bindParameters.put("id", criteria.getId());
        }
        if (!CollectionUtils.isEmpty(criteria.getStatusCodes())) {
            andConditions.add("o.status.code IN :statusCodes");
            bindParameters.put("statusCodes", criteria.getStatusCodes());
        }
        if (StringUtils.hasLength(criteria.getItemName())) {
            joinConditions.add("o.orderItems oi");
            joinConditions.add("oi.item i");
            andConditions.add("i.name LIKE :itemName ESCAPE '~'");
            bindParameters.put("itemName", QueryEscapeUtils
                    .toLikeCondition(criteria.getItemName()));
        }

        // (8)
        if (andConditions.isEmpty()) {
            return Collections.emptyList();
        }

        // (9)
        // Create dynamic query.
        final StringBuilder queryString = new StringBuilder();

        // (10)
        queryString.append("SELECT o FROM Order o");

        // (11)
        // add join conditions.
        for (String joinCondition : joinConditions) {
            queryString.append(" LEFT JOIN ").append(joinCondition);
        }
        // add conditions.
        Iterator<String> andConditionsIt = andConditions.iterator();
        if (andConditionsIt.hasNext()) {
            queryString.append(" WHERE ").append(andConditionsIt.next());
        }
        while (andConditionsIt.hasNext()) {
            queryString.append(" AND ").append(andConditionsIt.next());
        }

        // (12)
        // add order by condition.
        queryString.append(" ORDER BY o.id");
```

```
        // (13)
        // Create typed query.
        final TypedQuery<Order> findQuery = entityManager.createQuery(
                queryString.toString(), Order.class);
        // Bind parameters.
        for (Map.Entry<String, Object> bindParameter : bindParameters
                .entrySet()) {
            findQuery.setParameter(bindParameter.getKey(), bindParameter
                    .getValue());
        }

        // (14)
        // Execute query.
        return findQuery.getResultList();

    }

}
```

項番	説明
(3)	カスタム Repository インタフェースの実装クラスを作成する。
(4)	`EntityManager` を Inject する。 `@javax.persistence.PersistenceContext` アノテーションを使用して Inject すること。
(5)	動的条件に一致する Entity を全件取得する Query メソッドを実装する。 上記実装例では説明のためにメソッド分割はしていないが、必要に応じてメソッド分割すること。
(6)	動的クエリを組み立てるための変数（AND 条件用リスト、結合条件用リスト、バインドパラメータ用マップ）を定義している。 OrderCriteria オブジェクトに条件の指定があるものについて、これらの変数に必要な情報を設定する。

Continued on next page

表 6.19 – continued from previous page

項番	説明
(7)	OrderCriteria オブジェクトに条件の指定があるか判定し、動的クエリを組み立てるために必要な情報を設定していく。 上記例では、`id` は指定した値と完全一致するもの、 `statusCodes` は指定したリストに含まれるもの、 `itemName` は指定した値と前方一致するものを取得対象とするための情報を設定している。 `itemName` については、比較対象の値を保持する関連 Entity が複雑なネスト関係になっているため、関連 Entity を JOIN する必要がある。
(8)	上記実装例では条件が指定されていない場合は、検索する必要がないため、空のリストを返却する。
(9)	条件が指定されている場合は、Entity を検索するための Query を組み立てる。 上記例では実行する Query を `java.lang.StringBuilder` クラスを使って組み立てる実装例となっている。
(10)	静的な Query 要素を組み立てる。 上記例では、SELECT 句と FROM 句は静的な Query の構成要素として組み立てている。
(11)	動的な Query 要素を組み立てる。 上記例では、結合条件用リスト (JOIN 句) と AND 条件用リスト (WHERE 句) に設定する条件を動的な Query 要素として組み立てている。
(12)	静的な Query 要素を組み立てる。 上記例では、ORDER BY 句は静的な Query 要素として組み立てている。
(13)	動的に組み立てた Query 文字列を `javax.persistence.TypedQuery` に変換し、Query 実行に必要なバインド用のパラメータを設定する。
(14)	動的に組み立てた Query を実行し、条件に一致する Entity を全件取得する。

- Entity 毎の Repository インタフェース

```
public interface OrderRepository extends JpaRepository<Order, Integer>,
                                 OrderRepositoryCustom { // (15)
    // ...
}
```

項番	説明
(15)	Entity 毎の Repository インタフェースに、カスタム Repository インタフェースを継承する。

- Service (Caller)

```
// condition values for sample.
Integer conditionValueOfId = 4;
List<String> conditionValueOfStatusCodes = Arrays.asList("accepted");
String conditionValueOfItemName = "Wat";

// implementation of sample.
// (16)
OrderCriteria criteria = new OrderCriteria();
criteria.setId(conditionValueOfId);
criteria.setStatusCodes(conditionValueOfStatusCodes);
criteria.setItemName(conditionValueOfItemName);
List<Order> orders = orderRepository.findAllByCriteria(criteria); // (17)
if (orders.isEmpty()) { // (18)
    // ...
}
```

項番	説明
(16)	OrderCriteria オブジェクトに検索条件を指定する。
(17)	OrderCriteria オブジェクトを引数として、動的条件に一致する Entity を全件取得する Query メソッドを呼び出す。
(18)	必要に応じて検索結果を判定し、0 件の場合の処理を行う。

- 実行される JPQL(SQL)

```
-- (19)
--   conditionValueOfId=4
--   conditionValueOfStatusCodes = ["accepted"]
--   conditionValueOfItemName = "Wat"

-- JPQL
SELECT
        o
    FROM
        ORDER o
            JOIN o.orderItems oi
                JOIN oi.item i
            WHERE
                o.id = :id
                AND o.status.code IN :statusCodes
                AND i.name LIKE :itemName ESCAPE '~'
            ORDER BY
                o.id

-- SQL
SELECT
        order0_.id AS id1_6_
        ,order0_.created_by AS created2_6_
        ,order0_.created_date AS created3_6_
        ,order0_.last_modified_by AS last4_6_
        ,order0_.last_modified_date AS last5_6_
        ,order0_.status_code AS status6_6_
    FROM
        t_order order0_ INNER JOIN t_order_item orderitems1_
            ON order0_.id = orderitems1_.order_id INNER JOIN m_item item2_
            ON orderitems1_.item_code = item2_.code
WHERE
    order0_.id = 4
    AND (
        order0_.status_code IN ('accepted')
    )
    AND (
        item2_.name LIKE 'Wat%' ESCAPE '~'
    )
ORDER BY
    order0_.id
```

項番	説明
(19)	すべての条件を指定した場合に発行される JPQL と SQL の例。

```
-- (20)
--   conditionValueOfId=4
```

```
--   conditionValueOfStatusCodes = ["accepted"]
--   conditionValueOfItemName = ""
-- JPQL
SELECT
        o
    FROM
        ORDER o
    WHERE
        o.id = :id
        AND o.status.code IN :statusCodes
    ORDER BY
        o.id

-- SQL
SELECT
        order0_.id AS id1_6_
        ,order0_.created_by AS created2_6_
        ,order0_.created_date AS created3_6_
        ,order0_.last_modified_by AS last4_6_
        ,order0_.last_modified_date AS last5_6_
        ,order0_.status_code AS status6_6_
    FROM
        t_order order0_
    WHERE
        order0_.id = 4
        AND (
            order0_.status_code IN ('accepted')
        )
    ORDER BY
        order0_.id;
```

項番	説明
(20)	iteName 以外の条件を指定した場合に発行される JPQL と SQL の例。

```
-- (21)
--   conditionValueOfId=4
--   conditionValueOfStatusCodes = []
--   conditionValueOfItemName = ""
-- JPQL
SELECT
        o
    FROM
        ORDER o
    WHERE
        o.id = :id
    ORDER BY
        o.id
```

```
-- SQL
SELECT
        order0_.id AS id1_6_
        ,order0_.created_by AS created2_6_
        ,order0_.created_date AS created3_6_
        ,order0_.last_modified_by AS last4_6_
        ,order0_.last_modified_date AS last5_6_
        ,order0_.status_code AS status6_6_
    FROM
        t_order order0_
    WHERE
        order0_.id = 4
    ORDER BY
        order0_.id;
```

項番	説明
(21)	id のみを指定した場合に発行される JPQL と SQL の例。

動的条件に一致する Entity をページ検索

動的条件に一致する Entity の中から指定ページに該当する Entity を取得する Query メソッドを実装し、呼び出す。

以下に実装例を示すが、該当ページを取得する箇所以外は、全件取得する場合と同じ仕様とする。なお、全件取得で説明した箇所の説明は省略する。

- カスタム Repository インタフェース

```
public interface OrderRepositoryCustom {

    Page<Order> findPageByCriteria(OrderCriteria criteria, Pageable pageable);
↪ // (1)

}
```

項番	説明
(1)	動的条件に一致する Entity の中から指定ページに該当する Entity を取得する Query メソッドを定義する。

- カスタム Repository クラス

```
public class OrderRepositoryCustomImpl implements OrderRepositoryCustom {

    @PersistenceContext
    EntityManager entityManager;

    public Page<Order> findPageByCriteria(OrderCriteria criteria,
            Pageable pageable) { // (2)

        // collect dynamic conditions.
        final List<String> andConditions = new ArrayList<String>();
        final List<String> joinConditions = new ArrayList<String>();
        final Map<String, Object> bindParameters = new HashMap<String, Object>
↪();

        if (criteria.getId() != null) {
            andConditions.add("o.id = :id");
            bindParameters.put("id", criteria.getId());
        }
        if (!CollectionUtils.isEmpty(criteria.getStatusCodes())) {
            andConditions.add("o.status.code IN :statusCodes");
            bindParameters.put("statusCodes", criteria.getStatusCodes());
        }
        if (StringUtils.hasLength(criteria.getItemName())) {
            joinConditions.add("o.orderItems oi");
            joinConditions.add("oi.item i");
            andConditions.add("i.name LIKE :itemName ESCAPE '~'");
            bindParameters.put("itemName", QueryEscapeUtils.
↪toLikeCondition(criteria
                    .getItemName()));
        }

        if (andConditions.isEmpty()) {
            List<Order> orders = Collections.emptyList();
            return new PageImpl<Order>(orders, pageable, 0); // (3)
        }

        // create dynamic query.
        final StringBuilder queryString = new StringBuilder();
        final StringBuilder countQueryString = new StringBuilder(); // (4)
        final StringBuilder conditionsString = new StringBuilder(); // (4)

        queryString.append("SELECT o FROM Order o");
        countQueryString.append("SELECT COUNT(o) FROM Order o"); // (5)
```

```
        // add join conditions.
        for (String joinCondition : joinConditions) {
            conditionsString.append(" JOIN ").append(joinCondition);
        }

        // add conditions.
        Iterator<String> andConditionsIt = andConditions.iterator();
        if (andConditionsIt.hasNext()) {
            conditionsString.append(" WHERE ").append(andConditionsIt.next());
        }
        while (andConditionsIt.hasNext()) {
            conditionsString.append(" AND ").append(andConditionsIt.next());
        }
        queryString.append(conditionsString); // (6)
        countQueryString.append(conditionsString); // (6)

        // add order by condition.
        // (7)
        String orderByString = QueryUtils.applySorting("", pageable.getSort(),
↪ "o");
        queryString.append(orderByString);

        // create typed query.
        final TypedQuery<Long> countQuery = entityManager.createQuery(
                countQueryString.toString(), Long.class); // (8)

        final TypedQuery<Order> findQuery = entityManager.createQuery(
                queryString.toString(), Order.class);

        // bind parameters.
        for (Map.Entry<String, Object> bindParameter : bindParameters
                .entrySet()) {
            countQuery.setParameter(bindParameter.getKey(), bindParameter
                    .getValue()); // (8)
            findQuery.setParameter(bindParameter.getKey(), bindParameter
                    .getValue());
        }

        long total = countQuery.getSingleResult().longValue(); // (9)
        List<Order> orders = null;
        if (total != 0) { // (10)
            findQuery.setFirstResult(pageable.getOffset());
            findQuery.setMaxResults(pageable.getPageSize());
            // execute query.
            orders = findQuery.getResultList();
        } else { // (11)
            orders = Collections.emptyList();
        }

        return new PageImpl<Order>(orders, pageable, total); // (12)
    }
```

```
}
```

項番	説明
(2)	動的条件に一致する Entity の中から指定ページに該当する Entity を取得する Query メソッドを実装する。 上記実装例では説明のためにメソッド分割はしていないが、必要に応じてメソッド分割すること。
(3)	上記実装例では条件が指定されていない場合は、検索する必要がないため、空のページ情報を返却する。
(4)	件数取得用の Query を組み立てるための変数と、条件 (結合条件と AND 条件) を組み立てるための変数を用意する。 Entity 取得用の Query と件数取得用の Query で同じ条件を使用する必要があるため、条件 (結合条件と AND 条件) を組み立てるための変数を用意している。
(5)	件数取得用 Query の静的な Query 要素を組み立てる。 上記例では、SELECT 句と FROM 句は静的な Query の構成要素として組み立てている。
(6)	Entity 取得用 Query と件数取得用 Query の動的な Query 要素を組み立てる。
(7)	Entity 取得用 Query に対してソート条件 (ORDER BY 句) を動的な Query 要素として組み立てている。 ORDER BY 句の組み立ては、 Spring Data JPA から提供されているユーティリティ部品 (`org.springframework.data.jpa.repository.query.QueryUtils`) を使用している。
(8)	動的に組み立てた件数取得用の Query 文字列を `javax.persistence.TypedQuery` に変換し、Query 実行に必要なバインド用のパラメータを設定する。
(9)	件数取得用の Query を実行し、条件に一致する合計件数を取得する。

Continued on next page

表 6.20 – continued from previous page

項番	説明
(10)	条件に一致する Entity が存在する場合は、Entity 取得用 Query を実行し、該当ページの情報を取得する。 Entity 取得用 Query を実行する際は、取得開始位置 (`TypedQuery#setFirstResult`) と取得件数 (`TypedQuery#setMaxResults`) を指定する。
(11)	条件に一致する Entity が存在しない場合は、空のリストを取得結果とする。
(12)	該当ページの Entity のリスト、ページ情報、条件に条件した合計件数を引数に指定して、`Page` オブジェクトを生成し返却する。

- Service (Caller)

```
// condition values for sample.
Integer conditionValueOfId = 4;
List<String> conditionValueOfStatusCodes = Arrays.asList("accepted");
String conditionValueOfItemName = "Wat";

// implementation of sample.
OrderCriteria criteria = new OrderCriteria();
criteria.setId(conditionValueOfId);
criteria.setStatusCodes(conditionValueOfStatusCodes);
criteria.setItemName(conditionValueOfItemName);
Page<Order> orderPage = orderRepository.findPageByCriteria(criteria,
        pageable); // (13)
if (!orderPage.hasContent()) {
    // ...
}
```

項番	説明
(13)	動的条件に一致する Entity の中から指定ページに該当する Entity を取得する Query メソッドを呼び出す。

Entity の取得処理の実装

Entity の取得方法について、目的別に説明する。

ID を指定して Entity を 1 件取得

ID(Primary Key) がわかっている場合は、Repositry インタフェースの findOne メソッドを呼び出して Entity オブジェクトを取得する。

```
public Account getAccount(String accountUuid) {
    Account account = accountRepository.findOne(accountUuid); // (1)
    if (account == null) { // (2)
        // ...
    }
    return account;
}
```

項番	説明
(1)	Entity の ID(Primary Key) を指定して、Repositry インタフェースの findOne(ID) メソッドを呼び出す。
(2)	指定した ID の Entity が存在しない場合は、返却値が null になるので、null 判定が必要となる。 必要に応じて、指定した ID の Entity が存在しない場合の処理を実装する。

注釈: 返却される Entity オブジェクトについて

指定した ID の Entity オブジェクトが、既に `EntityManager` 上で管理されている場合は、永続層 (DB) へのアクセスは行われずに、`EntityManager` 上で管理されている Entity オブジェクトが返却される。そのため、findOne メソッドを使用すると、永続層への無駄なアクセスを抑えることが出来る。

注釈: 関連 Entity のロードタイミングについて

Query 実行時の関連 Entity のロードは、Entity の関連付けアノテーション (`@javax.persistence.OneToOne` 、 `@javax.persistence.OneToMany` 、 `@javax.persistence.ManyToOne` 、 `@javax.persistence.ManyToMany`) の fetch 属性に指定されている値によって決定される。

- `javax.persistence.FetchType#LAZY` の場合は、JOIN FETCH 対象からはずれるため、関連 Entity のロードは初回アクセス時に行われる。
- `javax.persistence.FetchType#EAGER` の場合は、 JOIN FETCH されるため、関連 Entity がロードされる。

fetch 属性のデフォルトはアノテーションによって異なる。デフォルト値は以下の通り。

- `@OneToOne` アノテーション : `EAGER`
- `@ManyToOne` アノテーション : `EAGER`
- `@OneToMany` アノテーション : `LAZY`
- `@ManyToMany` アノテーション : `LAZY`

注釈: 1:N(N:M) の関連をもつ関連 Entity の並び順について

関連 Entity のプロパティのアノテーションに、 `@javax.persistence.OrderBy` アノテーション

を指定して制御する。

例を以下に示す。

```
@OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval =␣
↪true)
@OrderBy // (1)
private Set<OrderItem> orderItems;
```

項番	説明
(1)	value 属性を指定しない場合は、 ID の昇順となる。 詳細は、 JSR 338: Java Persistence API, Version 2.1 の Specification(PDF)「11.1.42 OrderBy Annotation」

課題：TBD

今後、以下の内容を追加する予定である。

- fetch 属性をデフォルト値から変更した方がよいケースの例。

ID 以外の条件を指定して Entity を 1 件取得

ID がわからない場合は、ID 以外の条件で Entity を検索する Query メソッドを呼び出す。

- Repositroy インタフェース

```
public interface AccountRepository extends JpaRepository<Account, String> {

    // (1)
    @Query("SELECT a FROM Account a WHERE a.accountId = :accountId")
    Account findByAccountId(@Param("accountId") String accountId);

}
```

項番	説明
(1)	ID 以外の項目を条件として、Entity を 1 件検索する Query メソッドを定義する。

- Service

```
public Account getAccount(String accountId) {
    Account account = accountRepository.findByAccountId(accountId); // (2)
    if (account == null) { // (3)
        // ...
    }
```

項番	説明
(2)	Repositry インタフェースに実装した Query メソッドを呼び出す。
(3)	Repositry インタフェースの findOne メソッドと同様に、条件に一致する Entity が存在しない場合は、返却値が null になるので、null 判定が必要となる。 必要に応じて、指定した条件に一致する Entity が存在しない場合の処理を実装する。 条件に一致する Entity が複数存在した場合は、`org.springframework.dao.IncorrectResultSizeDataAccessException` が発生する。

注釈: 返却される Entity オブジェクトについて

Query メソッドを呼び出した場合、必ず永続層 (DB) に対して Query が実行される。ただし、Query を実行して取得された Entity が既に `EntityManager` 上で管理されている場合は、 Query を実行して取得された Entity オブジェクトは破棄され、 `EntityManager` 上で管理されている Entity オブジェクトが返却される。

注釈: ID + α を条件とする Query メソッドについて

ID + α を条件とする Query メソッドは、原則作成しないようにすることを推奨する。このケースは、findOne メソッドを呼び出して取得した Entity オブジェクトのプロパティ値を比較するロジックを組むことで、同じことが実現できる。

原則作成しないことを推奨する理由は、Query を実行して取得された Entity オブジェクトが破棄される可能性があり、無駄な Query の実行となってしまうためである。ID がわかっているのであれば、 無駄

な Query の実行が行われない findOne メソッドを使用した方がよい。特に、性能要件が高いアプリケーションの場合は、意識して実装すること。

ただし、以下の条件に当てはまる場合は、Query メソッドを使用した方が Query の実行回数が少なくなる事もあるので、その場合は Query メソッドを使用してもよい。

- + α の条件の部分に関連オブジェクトのプロパティ (関連テーブルのカラム) が含まれる。
- 条件となる関連 Entity への FetchType に `LAZY` が含まれる。

注釈: 関連 Entity のロードタイミングについて

JOIN FETCH に指定された関連 Entity は、Query 実行直後にロードされる。

JOIN FETCH に指定がない関連 Entity は、関連付けアノテーション (`@OneToOne` 、 `@OneToMany` 、 `@ManyToOne` 、 `@ManyToMany`) の fetch 属性に指定されている値によって以下の動作になる。

- `javax.persistence.FetchType#LAZY` の場合は、Lazy Load 対象となり、関連 Entity のロードは初回アクセス時に行われる。
- `javax.persistence.FetchType#EAGER` の場合は、 関連 Entity をロードするための Query が実行され、関連 Entity のオブジェクトがロードされる。

注釈: 1:N(N:M) の関連をもつ関連 Entity の並び順について

- JOIN FETCH に指定された関連 Entity の並び順は、JPQL に」ORDER BY」句を指定して制御する。
- Query 実行後にロードされる関連 Entity の並び順は、 関連 Entity のプロパティのアノテーションに、 `@javax.persistence.OrderBy` アノテーションを指定して制御する。

Entity の追加処理の実装

Entity の追加方法について、目的別に実装例を説明する。

Entity の追加

Entity を追加したい場合は、Entity オブジェクトを生成し、Repositry インタフェースの save メソッドを呼び出す。

- Service

```
Order order = new Order("accepted"); // (1)
order = orderRepository.save(order); // (2)
```

項番	説明
(1)	Entity オブジェクトのインスタンスを生成し、必要なプロパティに値を設定する。 上記例では、ID の設定は JPA の ID 採番機能を使用している。JPA の ID 採番機能を使用する場合は、アプリケーションコードで ID の設定は行ってはいけない。 アプリケーションコードで ID を設定してしまうと、 `EntityManager` の merge メソッドが呼び出されるため、不要な処理が実行されてしまう。
(2)	Repository インタフェースの save メソッドを呼び出し、(1) で生成した Entity オブジェクトを `EntityManager` の管理対象にする。 `EntityManager` によって管理される Entity オブジェクトは、save メソッドの引数に渡した Entity オブジェクトではなく、save メソッドから返却された Entity オブジェクトになるので注意すること。 ID は、このタイミングで JPA の ID 採番機能によって設定される。

注釈: merge メソッドが呼び出されるデメリット

`EntityManager` の merge メソッドは、Entity を `EntityManager` の管理対象にする際に、永続層 (DB) から同じ ID をもつ Entity を取得する仕組みになっている。Entity の追加処理としては、Entity の取得処理は無駄な処理となる。高い性能要件があるアプリケーションの場合は、ID の採番タイミングを意識すること。

注釈: 制約エラーのハンドリング

save メソッドを呼び出したタイミングでは、永続層 (DB) に Query(INSERT) は実行されない。そのため、一意制約違反などの制約系のエラーをハンドリングする必要がある場合は、Repository インタフェースの save メソッドではなく、saveAndFlush メソッドまたは flush メソッドを呼び出す必要がある。

- Entity

```
@Entity // (3)
@Table(name = "t_order") // (4)
public class Order implements Serializable {
```

```
    // (5)
    @SequenceGenerator(name = "GEN_ORDER_ID", sequenceName = "s_order_id",
                       allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "GEN_ORDER_ID")
    @Id // (6)
    private int id;

    // (7)
    @ManyToOne
    @JoinColumn(name = "status_code")
    private OrderStatus status;

    // ...

    public Order(String statusCode) {
        this.status = new OrderStatus(statusCode);
    }

    // ...

}
```

項番	説明
(3)	Entity クラスには `@javax.persistence.Entity` アノテーションを付与する。
(4)	Entity クラスとマッピングするテーブル名は `@javax.persistence.Table` アノテーションの name 属性に指定する。 エンティティ名からテーブル名が解決できる場合は指定しなくてもよいが、解決できない場合は指定すること。
(5)	JPA の ID 採番機能を使用するために必要なアノテーションを指定している。 JPA の ID 採番機能を使用する場合は、 `@javax.persistence.GeneratedValue` アノテーションを指定する。 シーケンスオブジェクトを使用する場合は、`@javax.persistence.SequenceGenerator` アノテーション、採番テーブルを使用する場合は、 `@javax.persistence.TableGenerator` アノテーションの指定も必要となる。 上記例では、 `s_order_id` という名前のシーケンスオブジェクトを使って ID を採番している。
(6)	主キーを保持するプロパティに、`@javax.persistence.Id` アノテーションを付与する。 複合キーの場合は、 `@javax.persistence.EmbeddedId` アノテーションを付与する。
(7)	別の Entity との関連を保持するプロパティに、 関連付けアノテーション (`@OneToOne` 、`@OneToMany` 、 `@ManyToOne` 、 `@ManyToMany`) を付与する。

注釈: ID 採番用のアノテーションについて

各アノテーションの詳細は、JSR 338: Java Persistence API, Version 2.1 の Specification(PDF) を参照されたい。

- `@GeneratedValue` : 11.1.20 GeneratedValue Annotation
- `@SequenceGenerator` : 11.1.48 SequenceGenerator Annotation
- `@TableGenerator` : 11.1.50 TableGenerator Annotation

注釈: ID の採番方法について

採番方法は、 `@GeneratedValue` アノテーションの strategy 属性に `javax.persistence.GenerationType` 列挙の値を指定する。指定可能な値は以下の通り。

- `TABLE` : 永続層 (DB) のテーブルを使用して ID を採番する。
- `SEQUENCE` : 永続層 (DB) のシーケンスオブジェクトを使用して ID を採番する。
- `IDENTITY` : 永続層 (DB) の identity 列を使用して ID を採番する。
- `AUTO` : 永続層 (DB) に最も適切な採番方法を選択して ID を採番する。

原則 `AUTO` は使用せず、明示的に使用するタイプを指定することを推奨する。

Entity と関連 Entity の追加

Entity と関連 Entity を一緒に追加したい場合は、Repositry インタフェースの save メソッドを呼び出して Entity オブジェクトを `EntityManager` の管理対象にした後に、関連 Entity のオブジェクトを生成し、Entity オブジェクトに関連づける。

この方法を使用するためには、関連 Entity の Cascade 対象の操作に、`persist` が含まれている必要がある。

注釈: Cascade 対象となった場合の挙動について

関連 Entity が Cascade 対象に指定されている場合、Entity に対する JPA の操作 (`persist` , `merge` , `remove` , `refresh` , `detach`) が関連 Entity に対して、連鎖して行われる。

Spring Data JPA の Repository インタフェースの操作に対するマッピングは以下の通り。

- save メソッド : `persist` or `merge`
- delete メソッド : `remove`

`refresh` , `detach` は Spring Data JPA のデフォルト実装で実行されることはない。

- Service

```
String itemCode = "ITM0000001";
int itemQuantity = 10;
String wayToPay = "card";
```

```
Order order = new Order("accepted");
order = orderRepository.save(order); // (1)

OrderItem orderItem = new OrderItem(order.getId(), itemCode,
        itemQuantity);
order.setOrderItems(Collections.singleton(orderItem));     // (2)
order.setOrderPay(new OrderPay(order.getId(), wayToPay)); // (2)
```

項番	説明
(1)	Repository インタフェースの save メソッドを呼び出し、まず Entity オブジェクトを EntityManager の管理対象にする。 上記例では、Entity オブジェクトに対して、関連 Entity のオブジェクトを設定する前に、save メソッドを呼び出している。これは、関連 Entity の ID の一部として使用される Entity の ID(orderId) を採番する必要があるためである。
(2)	EntityManager の管理対象となっている Entity オブジェクトに対して、関連 Entity のオブジェクトを設定する。 トランザクションのコミット時に、Entity オブジェクトへの persist 操作 (INSERT) が、関連 Entity のオブジェクトに対して連鎖される。

- Entity

```
@Entity
@Table(name = "t_order")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @SequenceGenerator(name = "GEN_ORDER_ID", sequenceName = "s_order_id",
                       allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "GEN_ORDER_ID")
    private int id;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL,  // (3)
               orphanRemoval = true)
    @OrderBy
    private Set<OrderItem> orderItems;

    @OneToOne(mappedBy = "order", cascade = CascadeType.ALL,
              orphanRemoval = true)
    private OrderPay orderPay;
```

```
    @ManyToOne
    @JoinColumn(name = "status_code")
    private OrderStatus status;

    // ...

    public Order(String statusCode) {
        this.status = new OrderStatus(statusCode);
    }

    // ...

}
```

項番	説明
(3)	関連アノテーションの cascade 属性に、Cascade 対象にする操作のタイプ (`javax.persistence.CascadeType`) を指定する。 上記例では、すべての操作を関連 Entity の Cascade 対象としている。 特に理由がない場合は、すべての操作を Cascade 対象にしておくことを推奨する。

警告: cascade 属性を指定してはいけない関連 Entity について

トランザクション系の Entity からコード系およびマスタ系の Entity に関連をもつ場合は、cascade 属性は指定してはいけない。上記例だと、`OrderStatus` はコード系の Entity になるので `Order` の `status` プロパティの `@ManyToOne` アノテーションに cascade 属性は指定してはいけない。

- 関連 Entity

```
@Entity
@Table(name = "t_order_item")
public class OrderItem implements Serializable {

    @EmbeddedId
    private OrderItemPK id;

    private int quantity;

    @ManyToOne
    @JoinColumn(name = "order_id", insertable = false, updatable = false)
    private Order order;

    // ...

    public OrderItem(Integer orderId, String itemCode, int quantity) {
```

```
        this.id = new OrderItemPK(orderId, itemCode);
        this.quantity = quantity;
    }

    // ...

}

@Entity
@Table(name = "t_order_pay")
public class OrderPay implements Serializable {

    @Id
    @Column(name = "order_id")
    private Integer orderId;

    @Column(name = "way_to_pay")
    private String wayToPay;

    @OneToOne
    @JoinColumn(name = "order_id")
    private Order order;

    // ...

    public OrderPay(int orderId, String wayToPay) {
        this.orderId = orderId;
        this.wayToPay = wayToPay;
    }

    // ...

}
```

関連 Entity の追加

関連 Entity を追加したい場合は、Repository インタフェース経由で取得した Entity オブジェクトに対して、生成した関連 Entity のオブジェクトを関連付ける。

この方法を使用するためには、関連 Entity の Cascade 対象の操作に、`persist` と `merge` が含まれている必要がある。

```
String itemCode = "ITM0000003";
int quantity = 30;

Order order = orderRepository.findOne(orderId); // (1)
```

```
OrderItem orderItem = new OrderItem(order.getId(), itemCode, quantity);
order.getOrderItems().add(orderItem); // (2)

OrderPay orderPay = order.getOrderPay();
if (orderPay == null) {
    order.setOrderPay(new OrderPay(order.getId(), "cash")); // (3)
} else {
    orderPay.setWayToPay("cash");
}
```

項番	説明
(1)	Repository インタフェースのメソッドを使用して、Entity オブジェクトを取得する。
(2)	1:N の関連 Entity の場合、生成した関連 Entity のオブジェクトを、Entity オブジェクトから取得したコレクションに追加する。
(3)	1:1 の関連 Entity の場合、生成した関連 Entity のオブジェクトを Entity オブジェクトに設定する。

関連 Entity の直接追加

関連 Entity を、親の Entity オブジェクトに関連付けせずに直接追加したい場合は、関連 Entity 用の Repository インタフェースを使用して保存する。

```
String itemCode = "ITM0000003";
int quantity = 40;

OrderItem orderItem = new OrderItem(orderId, itemCode, quantity); // (1)

orderItemRepository.save(orderItem); // (2)
```

項番	説明
(1)	関連 Entity のオブジェクトを生成する。
(2)	関連 Entity 用の Repository インタフェースの save メソッドを呼び出す。

注釈: 関連 Entity を直接保存するメリット

生成されるオブジェクトの数が少なくなる。 親の Entity オブジェクトを取得する場合、処理に不要な関連 Entity のオブジェクトが生成される可能性がある。

注釈: 関連 Entity を直接保存するデメリット

ID の一部に親 Entity の ID を使用している場合、save メソッドを呼び出す前に ID を設定しておく必要がある。ID を設定してしまうと Spring Data JPA のデフォルト実装は `EntityManager` の merge メソッドを呼び出す。そのため、必ず永続層 (DB) から ID が同じ Entity を取得する処理が実行されてしまう。

警告: 追加後に親の Entity オブジェクトを利用する際の注意点

関連 Entity 用 Repository の save メソッドを使って関連 Entity 追加した場合は、親にあたる Entity オブジェクトには関連付けられていないため、親の Entity オブジェクトを経由して取得することができない。

回避方法は、

1. 関連 Entity のオブジェクトを直接追加するのではなく、親の Entity オブジェクトに関連付けて追加するようにする。
2. saveAndFlush メソッドを使用することで、永続層 (DB) と同期を親の Entity を取得する前に行う。

このケースの場合、特に理由がないのであれば、前者の方法で関連 Entity のオブジェクトを追加すること。後者は、親にあたる Entity オブジェクトが既に管理状態の Entity になっていた場合に、問題を回避することができない。

Entity の更新処理の実装

Entity の更新方法について、目的別に実装例を説明する。

Entity の更新

Entity を更新したい場合は、Repository インタフェースのメソッドを使用して取得した Entity オブジェクトに対して、変更したい値を設定する。

```
Order order = orderRepository.findOne(orderId); // (1)
order.setStatus(new OrderStatus("checking")); // (2)
```

項番	説明
(1)	Repository インタフェースのメソッドを使用して、Entity オブジェクトを取得する。
(2)	Entity オブジェクトの状態を、setter メソッドを呼び出して更新する。

注釈: Repository の save メソッドの呼び出しについて

Repository インタフェースのメソッドを使用して取得した Entity オブジェクトは、`EntityManager` 上で管理されている。`EntityManager` 上で管理されている Entity オブジェクトは、setter メソッドを使ってオブジェクトの状態を変更するだけで、トランザクションコミット時に変更内容が永続層 (DB) に反映される仕組みになっている。そのため、Repository インタフェースの save メソッドを明示的に呼び出す必要はない。

ただし、Entity オブジェクトが `EntityManager` 上で管理されていない場合は、save メソッドを呼び出す必要がある。例えば、画面から送信されたリクエストパラメータをもとに Entity オブジェクトを生成した場合などが、このケースに当てはまる。

関連 Entity の更新

関連 Entity を更新したい場合は、Repository インタフェースのメソッドを使用して取得した Entity オブジェクトから取得した関連 Entity のオブジェクトに対して、変更したい値を設定する。

この方法を使用するためには、関連 Entity の Cascade 対象の操作に、`merge` が含まれている必要がある。

```
Order order = orderRepository.findOne(orderId); // (1)

for (OrderItem orderItem : order.getOrderItems()) {
    int newQuantity = quantityMap.get(orderItem.getId().getItemCode());
    orderItem.setQuantity(newQuantity); // (2)
}

order.getOrderPay().setWayToPay("cash"); // (3)
```

項番	説明
(1)	Repository インタフェースのメソッドを使用して、Entity オブジェクトを取得する。
(2)	1:N の関連 Entity の場合、Entity オブジェクトから取得したコレクションに格納されている関連 Entity のオブジェクトの状態を、setter メソッドを呼び出して更新する。
(3)	1:1 の関連 Entity の場合、Entity オブジェクトから取得した関連 Entity のオブジェクトの状態を、setter メソッドを呼び出して更新する。

関連 Entity の直接更新

関連 Entity を、親の Entity を使用せず直接更新したい場合は、関連 Entity 用の Repository インタフェースのメソッドを使用して取得した Entity オブジェクトに対して、変更したい値を設定する。

```
int quantity = 43;

OrderItem orderItem = orderItemRepository.findOne(new OrderItemPK(
        orderId, itemCode)); // (1)

orderItem.setQuantity(quantity); // (2)
```

項番	説明
(1)	関連 Entity の Repository インタフェースの findOne メソッドを呼び出して、関連 Entity のオブジェクトを取得する。
(2)	関連 Entity のオブジェクトの状態を、setter メソッドを使って更新する。

注釈: 更新後に親の Entity を利用した場合の動作について

関連 Entity 用 Repository の save メソッドを使って関連 Entity を更新した場合は、追加時とは異なり、親の Entity オブジェクトで保持している関連 Entity にも反映される。これは、`EntityManager` 上で管理されている同じインスタンスの参照を保持しているためである。

Query メソッドを使用して更新

永続層 (DB) の Entity を直接更新したい場合は、Query メソッドを使用して更新する。

詳細は、「永続層の *Entity* を直接操作する」を参照されたい。

Entity の削除処理の実装

Entity と関連 Entity の削除

Entity と関連 Entity を一緒に削除したい場合は、Repositry インタフェースの delete メソッドを呼び出す。

この方法を使用するためには、関連 Entity の Cascade 対象の操作に `remove` が含まれているか、または関連 Entity を削除するための設定を有効 (関連付けアノテーションの orphanRemoval 属性を `true`) にする必要がある。

- Service

```
orderRepository.delete(orderId); // (1)
```

項番	説明
(1)	ID または Entity オブジェクトを指定して Repositry インタフェースの delete メソッドを呼び出す。

- Entity

```
@Entity
@Table(name = "t_order")
public class Order implements Serializable {

    // ...

    @OneToMany(mappedBy = "order",
               cascade = CascadeType.ALL, orphanRemoval = true) // (2)
    @OrderBy
    private Set<OrderItem> orderItems;

    // ...

}
```

項番	説明
(2)	Cascade 対象の操作に `remove` を含め、 関連 Entity を削除するための設定を有効 (関連付けアノテーションの orphanRemoval 属性を `true`) にする。

注釈: 関連 Entity の削除について

関連 Entity のオブジェクトを一緒に削除したくない場合は、 Cascade 対象の操作に `remove` が含まれないようにしておく必要がある。また、関連付けアノテーションの orphanRemoval 属性も `false` に指定しておくこと。

関連 Entity の削除

関連 Entity を削除したい場合は、Repository インタフェース経由で取得した Entity オブジェクトから、関連 Entity のオブジェクトを削除する。

この方法を使用するためには、関連 Entity を削除するための設定を有効 (関連付けアノテーションの orphanRemoval 属性を `true`) にする必要がある。

注釈: 関連 Entity を削除する設定を有効にした場合の挙動について

関連 Entity を削除する設定を有効にした場合、以下の動作になる。

- 1:N の関連 Entity の場合は、関連 Entity のオブジェクトをコレクションの中から削除すると、トランザクションコミット時に永続層 (DB) からも削除される。
- 1:1 の関連 Entity の場合は、関連 Entity を `null` に設定すると、トランザクションコミット時に永続層 (DB) からも削除される。

orphanRemoval 属性のデフォルト値となっている `false` が指定されている場合は、メモリ上からは消えるが、削除した関連 Entity のオブジェクトに対する永続処理 (UPDATE/DELETE) は行われない。

- Service

```
Order order = orderRepository.findOne(orderId); // (1)

// (2)
Set<OrderItem> orderItemsOfRemoveTarget = new LinkedHashSet<OrderItem>(); //␣
↪(3)
for (OrderItem orderItem : order.getOrderItems()) {
    String itemCode = orderItem.getId().getItemCode();
    if (quantityMap.containsKey(itemCode)) {
        int newQuantity = quantityMap.get(itemCode);
        orderItem.setQuantity(newQuantity);
    } else {
        orderItemsOfRemoveTarget.add(orderItem); // (4)
    }
}
order.getOrderItems().removeAll(orderItemsOfRemoveTarget); // (5)

order.setOrderPay(null); // (6)
```

項番	説明
(1)	Repository インタフェースのメソッドを使用して、Entity オブジェクトを取得する。
(2)	1:N の関連 Entity を削除する実装例について説明する。
(3)	削除する関連 Entity のオブジェクトを格納しておくコレクションを用意する。
(4)	削除対象の関連 Entity のオブジェクトを、(3) のコレクションに追加する。
(5)	削除対象の関連 Entity のオブジェクトから取得したコレクションの中から削除する。
(6)	1:1 の関連 Entity の場合、削除したい関連 Entity のオブジェクトを保持するプロパティを `null` に設定する。

- Entity

```
@Entity
@Table(name = "t_order")
public class Order implements Serializable {

    @Id
    @SequenceGenerator(name = "GEN_ORDER_ID", sequenceName = "s_order_id",␣
↪allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "GEN_
↪ORDER_ID")
    private int id;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL,
               orphanRemoval = true) // (7)
    @OrderBy
    private Set<OrderItem> orderItems;

    @OneToOne(mappedBy = "order", cascade = CascadeType.ALL,
              orphanRemoval = true) // (7)
    private OrderPay orderPay;

    @ManyToOne
```

```
    @JoinColumn(name = "status_code")
    private OrderStatus status;

    // ...

}
```

項番	説明
(7)	関連 Entity を削除するための設定を有効 (関連付けアノテーションの orphanRemoval 属性を `true`) にする。

注釈: **orphanRemoval 属性が指定できる関連付けアノテーションについて**

orphanRemoval 属性が指定できる関連付けアノテーションは、 `@OneToOne` と `@OneToMany` の 2 つとなっている。

関連 Entity の直接削除

関連 Entity を、親の Entity を使用せず直接削除したい場合は、関連 Entity の Repository インタフェースの delete メソッドを呼び出す。

```
int quantity = 43;

orderItemRepository.delete(new OrderItemPK(orderId, itemCode)); // (1)
```

項番	説明
(1)	ID または Entity オブジェクトを指定して関連 Entity 用 Repositry インタフェースの delete メソッドを呼び出す。

警告: **直接削除の注意点**

関連 Entity 用 Repository の delete メソッドを呼び出した場合、永続層 (DB) から削除されないケースがあるので注意すること。

削除されないケースは以下の通り。

- delete メソッド内の中で findOne メソッドを呼び出しているため、親の Entity との関連が `@OneToOne` の場合は、親の Entity が `EntityManager` 上で管理されてしまう。親の Entity が管理対象になった際に、delete メソッドで削除しようとしていた関連 Entity が親の Entity オブジェクト内にロードされる可能性がある。親の Entity オブジェクト内にロードされてしまうと、永続層 (DB) から削除されない。
- delete メソッド呼び出し後に、親の Entity オブジェクトを取得した場合、delete メソッドで削除した関連 Entity が親の Entity オブジェクト内にロードされる可能性がある。親の Entity オブジェクト内にロードされてしまうと、永続層 (DB) から削除されない。

回避方法は、

1. 関連 Entity のオブジェクトを直接削除するのではなく、親の Entity との関連付けを削除するようにする。

関連付けアノテーションの見直しを行うことで回避出来る可能性はあるが、確実な回避方法は、直接削除を行わないようにする方法である。

Query メソッドを使用して削除

永続層 (DB) から Entity を直接削除したい場合は、Query メソッドを使用して削除する。
詳細は、「永続層の *Entity* を直接操作する」を参照されたい。

警告: 関連オブジェクトの扱いについて

Query メソッドを使用して永続層 (DB) から直接 Entity を削除した場合、関連付けアノテーションの指定に関係なく、関連 Entity のオブジェクトは永続層から削除されない。

Repository インタフェースの `void deleteInBatch(Iterable<T> entities)` と `void deleteAllInBatch()` も同様の動作となる。

LIKE 検索時のエスケープについて

LIKE 検索を行う場合は、検索条件として使用する値を LIKE 検索用にエスケープする必要がある。

LIKE 検索用のエスケープ処理は、共通ライブラリから提供している `org.terasoluna.gfw.common.query.QueryEscapeUtils` クラスのメソッドを使用する事で実現することができる。

共通ライブラリから提供しているエスケープ処理の仕様については、「データベースアクセス（共通編）」の「*LIKE 検索時のエスケープについて*」を参照されたい。

以下に、共通ライブラリから提供しているエスケープ処理の使用方法について説明する。

一致方法を Query 側で指定する場合の使用方法

一致方法 (前方一致、後方一致、部分一致) の指定を JPQL として指定する場合は、エスケープのみ行うメソッドを使用する。

- Repository

```
// (1) (2)
@Query("SELECT a FROM Article a WHERE"
        + " (a.title LIKE %:word% ESCAPE '~' OR a.overview LIKE %:word%␣
↪ESCAPE '~')")
Page<Article> findPageBy(@Param("word") String word, Pageable pageable);
```

項番	説明
(1)	`@Query` アノテーションに指定する JPQL 内に LIKE 検索用のワイルドカード (「`%`」 または 「`_`」) を指定する。 上記例では、引数 `word` の前後にワイルドカード (「`%`」) を指定することで、一致方法を部分一致にしている。
(2)	共通ライブラリから提供しているエスケープ処理は、エスケープ文字として 「~」 を使用しているため、 LIKE 句の後ろに `ESCAPE '~'` を指定する。

注釈: ワイルドカード文字 「_」 について

ワイルドカード文字「`%`」は、「*@Query アノテーションで指定する*」で説明しているように、`@Query`

アノテーションを利用した場合に限り直接 JPQL 内で使用することができる。ワイルドカード文字「_」は直接 JPQL 内の LIKE 検索に利用することはできないため、下記 2 方法で利用すること。

1. ワイルドカード文字 「_」 をバインド変数内に含める。
2. ワイルドカード文字 「_」 を CONCAT などのデータベース製品が持つ文字列結合機能を利用してバインド変数に結合する。

- Service

```
@Transactional(readOnly = true)
public Page<Article> searchArticle(ArticleSearchCriteria criteria,
        Pageable pageable) {

    String escapedWord = QueryEscapeUtils.toLikeCondition(criteria.getWord());
↪ // (3)

    Page<Article> page = articleRepository.findPageBy(escapedWord, pageable);␣
↪// (4)

    return page;
}
```

項番	説明
(3)	LIKE 検索の一致方法を Query 側で指定する場合は、`QueryEscapeUtils#toLikeCondition(String)` メソッドを呼び出し、LIKE 検索用のエスケープのみ行う。
(4)	LIKE 検索用にエスケープされた値を Query メソッドの引数に渡す。

一致方法をロジック側で指定する場合の使用方法

一致方法 (前方一致、後方一致、部分一致) をロジック側で判定する場合は、エスケープされた値にワイルドカードを付与するメソッドを使用する。

- Repository

```
// (1)
@Query("SELECT a FROM Article a WHERE"
        + " (a.title LIKE :word ESCAPE '~' OR a.overview LIKE :word ESCAPE '~
↪')")
Page<Article> findPageBy(@Param("word") String word, Pageable pageable);
```

項番	説明
(1)	@Query アノテーションに指定する JPQL 内に LIKE 検索用のワイルドカードは指定しない。

- Service

```
@Transactional(readOnly = true)
public Page<Article> searchArticle(ArticleSearchCriteria criteria,
        Pageable pageable) {

    String word = QueryEscapeUtils
            .toContainingCondition(criteria.getWord()); // (2)

    Page<Article> page = articleRepository.findPageBy(word, pageable); // (3)

    return page;
}
```

項番	説明
(2)	ロジック側で一致方法を指定する場合は、 以下の何れかのメソッドを呼び出し、LIKE 検索用のエスケープと LIKE 検索用のワイルドカードを付与する。 QueryEscapeUtils#toStartingWithCondition(String) QueryEscapeUtils#toEndingWithCondition(String) QueryEscapeUtils#toContainingCondition(String)
(3)	LIKE 検索用にエスケープ＋ワイルドカードが付与された値を Query メソッドの引数に渡す。

JOIN FETCH について

JOIN FETCH は、関連する Entity を JOIN して一括で取得することで、Entity を取得するために発行する Query の数を減らすための仕組みである。

任意の Query を実行して Entity を取得する場合、関連付けアノテーションの fetch 属性が `EAGER` となっているプロパティは必ず JOIN FETCH すること。

JOIN FETCH しないと、関連 Entity を取得するための Query が別途発行されるため、性能に影響を与える可能性がある。

関連付けアノテーションの fetch 属性が `LAZY` となっているプロパティについては、使用頻度を考慮して JOIN FETCH するか判断すること。

必ず参照される関連 Entity の場合は JOIN FETCH すべきであるが、参照されるケースが少ないのであれば JOIN FETCH せず、初回アクセス時に Query を発行して取得する方がよい。

- Repository

```
@Query("SELECT a FROM Article a"
        + " INNER JOIN FETCH a.articleClass"   // (1)
        + " WHERE a.publishedDate = :publishedDate"
        + " ORDER BY a.articleId DESC")
List<Article> findAllByPublishedDate(
        @Param("publishedDate") Date publishedDate);
```

項番	説明
(1)	`[LEFT [OUTER] | INNER] JOIN FETCH Join` **対象のプロパティ** の形式で指定する。 具体的には以下のパターンとなる。 1. `LEFT JOIN FETCH` 関連 Entity が存在しない場合でも Entity は取得される。 SQL としては、`left outer join RelatedTable r on m.fkColumn = r.fkColumn` となる。 2. `LEFT OUTER JOIN FETCH` 1 と同様。 3. `INNER JOIN FETCH` 関連 Entity が存在しない場合は、Entity は取得されない。 SQL としては、`inner join RelatedTable r on m.fkColumn = r.fkColumn` となる。 4. `JOIN FETCH` 3 と同様。 上記例では、Article クラスの articleClass プロパティを JOIN FETCH 対象として指定している。

注釈: JOIN FETCH は N+1 問題の解決策の一つとして使用される。N+1 問題については、「*N+1 問題の対策方法*」を参照されたい。

6.3.3 How to extend

カスタムメソッドの追加方法

Spring Data では、Repository インタフェースに対して、任意のカスタムメソッドを追加する仕組みを提供している。

項番	追加方法	使用ケース
1.	*Entity* 毎の *Repository* インタフェースに個別に追加する	Spring Data より提供されている Query メソッドの仕組みで表現できない Query を実装する必要がある場合に使用する。 動的 Query を実装する場合は、この方法を使用してメソッドを追加する。
2.	すべての *Repository* インタフェースに一括で追加する	すべての Repository インタフェースで使用できるような汎用的な Query を実装する必要がある場合に使用する。 プロジェクト固有の汎用 Query を実装する場合は、この方法を使用してメソッドを追加する。 Spring Data JPA から提供されているデフォルト実装 (`SimpleJpaRepository`) の振る舞いを変更する必要がある場合、この仕組みを使用してメソッドをオーバーライドすることで対応することが出来る。

Entity 毎の Repository インタフェースに個別に追加する

Entity 毎の Repository インタフェースに個別にカスタムメソッドを追加する方法について説明する。

- Entity 毎のカスタム Repository インタフェース

```
// (1)
public interface OrderRepositoryCustom {

    // (2)
    Page<Order> findByCriteria(OrderCriteria criteria, Pageable pageable);

}
```

項番	説明
(1)	カスタムメソッドを定義するカスタムインタフェースを作成する。 インタフェース名に特に制約はないが、Entity 毎の Repository インタフェース名 + `Custom` とすることを推奨する。
(2)	カスタムメソッドを定義する。

- Entity 毎のカスタム Repository クラス

```
// (3)
public class OrderRepositoryImpl implements OrderRepositoryCustom {

    @PersistenceContext
    EntityManager entityManager; // (4)

    // (5)
    public Page<Order> findByCriteria(OrderCriteria criteria, Pageable␣
↪pageable) {
        // ...
        return new PageImpl<Order>(orders, pageable, totalCount);
    }

}
```

項番	説明
(3)	カスタムインタフェースの実装クラスを作成する。 クラス名は、Entity 毎の Repository インタフェース名 + `Impl` とすること。
(4)	Query を実行するために必要となる `javax.persistence.EntityManager` は `@javax.persistence.PersistenceContext` アノテーションを使ってインジェクションする。
(5)	カスタムインタフェースに定義したメソッドを実装する。

- Entity 毎の Repository インタフェース

```
public interface OrderRepository extends JpaRepository<Order, Integer>,
        OrderRepositoryCustom { // (6)
    // ...
}
```

項番	説明
(6)	Entity 毎の Repository インタフェースでカスタムインタフェースを継承する。 Repository インタフェースを継承することで、実行時にカスタムインタフェースの実装クラスのメソッドが呼び出される。

- Service(Caller)

```
public Page<Order> search(OrderCriteria criteria, Pageable pageable) {
    return orderRepository.findByCriteria(criteria, pageable); // (7)
}
```

項番	説明
(7)	呼び出し側は、他のメソッドと同様に Entity 毎の Repository インタフェースのメソッドを呼び出せばよい。 上記例では、`OrderRepository#findByCriteria(OrderCriteria, Pageable)` を呼び出すと `OrderRepositoryImpl#findByCriteria(OrderCriteria, Pageable)` が実行される。

すべての Repository インタフェースに一括で追加する

すべての Repository インタフェースに一括でカスタムメソッドを追加する方法について説明する。

下記の実装例では、取得した Entity とのバージョンチェックを行い、バージョンが異なる場合に楽観排他エラーとするメソッドを追加している。

- 共通の Repository インタフェース

```
// (1)
@NoRepositoryBean // (2)
public interface MyProjectRepository<T, ID extends Serializable> extends
```

```
        JpaRepository<T, ID> {

    // (3)
    T findOneWithValidVersion(ID id, Integer version);

}
```

項番	説明
(1)	追加するメソッドを定義するための共通 Repository インタフェースを作成する。 上記例では、Spring Data から提供されている `JpaRepository` を継承して共通 Repository インタフェースを作成している。
(2)	共通 Repository インタフェースの実装クラス (例だと `MyProjectRepositoryImpl`) が、Repository の Bean として自動登録されないようにするためのアノテーション。 共通 Repository インタフェースの実装クラスを<jpa:repositories>要素の base-package 属性で指定したパッケージ配下に格納する場合は、このアノテーションを指定しないと起動時にエラーとなる。
(3)	追加するメソッドを定義する。例では、取得した Entity のバージョンチェックを行うメソッドを追加している。

- 共通 Repository インタフェースの実装クラス

```
// (6)
public class MyProjectRepositoryImpl<T, ID extends Serializable>
        extends SimpleJpaRepository<T, ID>
        implements MyProjectRepository<T, ID> {

    private JpaEntityInformation<T, ID> entityInformation;
    private EntityManager entityManager;
    Method versionMethod;

    // (7)
    public MyProjectRepositoryImpl(
            JpaEntityInformation<T, ID> entityInformation,
            EntityManager entityManager) {
        super(entityInformation, entityManager);

        this.entityInformation = entityInformation; // (8)
        this.entityManager = entityManager; // (8)
```

```
        try {
            versionMethod = entityInformation.getJavaType().getMethod(
↪"getVersion");
        } catch (NoSuchMethodException | SecurityException e) { }

    }

    // (9)
    public T findOneWithValidVersion(ID id, Integer version) {

        if (versionMethod == null) {
            throw new UnsupportedOperationException(
                    String.format(
                            "Does not found version field in entity class.␣
↪class is '%s'.",
                            entityInformation.getJavaType().getName()));
        }

        T entity = findOne(id);

        if (entity != null && version != null) {
            Integer currentVersion;
            try {
                currentVersion = (Integer) versionMethod.invoke(entity);
            } catch (IllegalAccessException | IllegalArgumentException
                    | InvocationTargetException e) {
                throw new IllegalStateException(e);
            }
            if (!version.equals(currentVersion)) {
                throw new ObjectOptimisticLockingFailureException(
                        entityInformation.getJavaType().getName(), id);
            }
        }

        return entity;
    }

}
```

項番	説明
(6)	共通 Repository インタフェース (例だと、 `MyProjectRepository`) の実装クラスを作成する。 上記例では、Spring Data から提供されている `SimpleJpaRepository` を継承して実装クラスを作成している。
(7)	`org.springframework.data.jpa.repository.support.JpaEntityInformation` と `javax.persistence.EntityManager` を引数にとるコンストラクタを作成する。
(8)	追加するメソッドの処理で必要な場合は、 `JpaEntityInformation` および `EntityManager` をフィールドに保持しておく。
(9)	共通 Repository インタフェースに定義したメソッドの実装を行う。

注釈: デフォルト実装の変更

デフォルト実装 (`SimpleJpaRepository`) の振る舞いを変更する場合は、このクラスで振る舞いを変更したいメソッドをオーバーライドすればよい。

- Entity 毎の Repository インタフェース

```
public interface OrderRepository extends MyProjectRepository<Order, Integer>
↪{ // (10)
    // ...
}
```

項番	説明
(10)	Entity 毎の Repository インタフェースでは、作成した共通インタフェース (例だと `MyProjectRepository`) を継承先として指定する。

- `xxx-infra.xml`

```
<jpa:repositories base-package="x.y.z.domain.repository"
    base-class="x.y.z.domain.repository.MyProjectRepositoryImpl" /> <!-- (11)␣
↪-->
```

項番	説明
(11)	<jpa:repositories>要素の base-class 属性に、作成した共通 Repository インタフェースの実装クラス (例だと `MyProjectRepositoryImpl`) のクラス名を指定する。

- Service(Caller)

```
public Order updateOrder(Order chngedOrder, Integer version) {

    Order order = orderRepository.findOneWithValidVersion(chngedOrder.getId(),
↪ version); // (12)

    // ....

    return order;
}
```

項番	説明
(12)	呼び出し側は、他のメソッドと同様に Entity 毎の Repository インタフェースのメソッドを呼び出せばよい。 上記例では、`OrderRepository#findOneWithValidVersion(Integer, Integer)` を呼び出すと `MyProjectRepositoryImpl#findOneWithValidVersion(Integer, Integer)` が実行される。

Entity 以外のオブジェクトに Query の取得結果を格納する方法

Query の取得結果を Entity にマッピングするのではなく、別のオブジェクトにマッピングすることができる。これは、永続層 (DB) に格納されているレコードを Entity 以外のオブジェクト (JavaBean) として扱いたい時に使用する。

注釈: 想定される適用ケース

1. Query 内で集計関数を使用して集計済みの情報を取得したい場合は、集計結果を Entity にマッピングすることはできたいため、別のオブジェクトにマッピングする必要がある。
2. 巨大な Entity の中の一部の情報のみ参照したい場合や、ネストが深い関連 Entity の一部の情報のみ参照したい場合は、必要なプロパティのみ定義した JavaBean にマッピングして取得する事を検討した方がよい場合がある。Entity として取得した場合、アプリケーションの処理に不要な項目に対するマッピング処理が行われる点や、処理に不要な情報を取得することでメモリを無駄に使用する点などにより、処理性能に影響を与える場合があるためである。処理性能に大きな影響を与えない場合は、Entity として取得してよい。

以下の実装例を示す。

- JavaBean

```
// (1)
public class OrderSummary implements Serializable {

    private Integer id;
    private Long totalPrice;

    // ...

    public OrderSummary(Integer id, Long totalPrice) { // (2)
        super();
        this.id = id;
        this.totalPrice = totalPrice;
    }

    // ...

}
```

項番	説明
(1)	Query 結果を格納する JavaBean を作成する。
(2)	Query を実行した結果でオブジェクト生成するためのコンストラクタを用意する。

- Repository インタフェース

```
// (3)
@Query("SELECT NEW x.y.z.domain.model.OrderSummary(o.id, SUM(i.price*oi.
↪quantity))"
        + " FROM Order o LEFT JOIN o.orderItems oi LEFT JOIN oi.item i"
        + " GROUP BY o.id ORDER BY o.id DESC")
List<OrderSummary> findOrderSummaries();
```

項番	説明
(3)	(2) で作成したコンストラクタを指定する。 「NEW」キーワードの後に FQCN でコンストラクタを指定する。

Audit 用プロパティの設定方法

Spring Data JPA より提供されている、永続層の Audit 用プロパティ (作成者、作成日時、最終更新者、最終更新日時) に値を設定する仕組みと適用方法について説明する。

Spring Data JPA では、新たに作成された Entity と更新された Entity に対して、Audit 用プロパティに値を設定する仕組みを提供している。この仕組みを使用すると、Service などのアプリケーションのコードから Audit 用プロパティに値を設定する処理を切り離すことができる。

注釈: アプリケーションのコードから Audit 用プロパティに値を設定する処理を切り離す理由

1. Audit 用プロパティへの値の設定は、一般的にアプリケーション要件ではなく、データの監査要件によって必要になる処理である。本質的には Service などのアプリケーション内のコードで意識すべき処理ではないため、アプリケーションのコードから切り離した方がよい。
2. JPA では、永続層から取得した Entity に対して値の変更があった場合のみ、永続層へ反映 (SQL を発行) する仕様になっており、永続層へ無駄なアクセスが行われないように制御されている。Service などのアプリケーションのコードで Audit 用プロパティに無条件に値を設定してしまうと、Audit 用プロパティ以外に変更がない場合でも、永続層への反映が行われることになるため、JPA の有効な機能を無駄にしてしまう。値の変更があった場合のみ Audit 用プロパティに値を設定するように制御すればこの問題は回避できるが、アプリケーションのコードが煩雑になるため推奨できない。

注釈: 値の変更有無に関係なく永続層の Audit 用カラムを更新する必要がある場合

値に変更がなくても永続層の Audit 用カラム (最終更新者、最終更新日時) を更新するのが、アプリケー

ションの仕様となっている場合は、Service などのアプリケーションのコードで Audit 用プロパティを設定する必要がある。

ただし、このケースではデータモデリングまたはアプリケーション仕様が間違っている可能性があるため、データモデリングおよびアプリケーション仕様の見直しを行った方がよい。

以下に、適用時の実装例を示す。

- Entity クラス

```
public class XxxEntity implements Serializable {
    private static final long serialVersionUID = 1L;

    // ...

    @Column(name = "created_by")
    @CreatedBy // (1)
    private String createdBy;

    @Column(name = "created_date")
    @CreatedDate // (2)
    @Type(type = "org.jadira.usertype.dateandtime.joda.PersistentDateTime") // (3)
    private DateTime createdDate; // (4)

    @Column(name = "last_modified_by")
    @LastModifiedBy // (5)
    private String lastModifiedBy;

    @Column(name = "last_modified_date")
    @LastModifiedDate // (6)
    @Type(type = "org.jadira.usertype.dateandtime.joda.PersistentDateTime") // (3)
    private DateTime lastModifiedDate; // (4)

    // ...

}
```

項番	説明
(1)	作成者を保持するフィールドに `@org.springframework.data.annotation.CreatedBy` アノテーションを付与する。
(2)	作成日時を保持するフィールドに `@org.springframework.data.annotation.CreatedDate` アノテーションを付与する。
(3)	`org.joda.time.DateTime` 型を使用する場合は、Hibernate で扱えるようにするために、 フィールドに `@org.hibernate.annotations.Type` アノテーションを付与する。 type 属性は、 `org.jadira.usertype.dateandtime.joda.PersistentDateTime` 固定。最終更新日時のフィールドも同様。
(4)	作成日時を保持するフィールドの型は、 `org.joda.time.DateTime` 、 `java.util.Date` 、`java.util.Calendar` 、 `java.lang.Long` 、 `long` 型 、Java 8 から追加された Date and Time API などをサポートしている。 最終更新日時のフィールドも同様。
(5)	最終更新者を保持するフィールドの型に `@org.springframework.data.annotation.LastModifiedBy` アノテーションを付与する。
(6)	最終更新日時を保持するフィールドに `@org.springframework.data.annotation.LastModifiedDate` アノテーションを付与する。

警告: `@Type` アノテーションは、JPA の標準アノテーションではなく、Hibernate 独自のアノテーションである。

- AuditorAware インタフェースの実装クラス

```
// (7)
@Component // (8)
public class SpringSecurityAuditorAware implements AuditorAware<String> {

    // (9)
    public String getCurrentAuditor() {
        Authentication authentication = SecurityContextHolder.getContext()
                .getAuthentication();
        if (authentication == null || !authentication.isAuthenticated()) {
            return null;
        }
        return ((UserDetails) authentication.getPrincipal()).getUsername();
    }

}
```

項番	説明
(7)	`org.springframework.data.domain.AuditorAware` インタフェースの実装クラスを作成する。 `AuditorAware` インタフェースは、Entity の操作者 (作成者または最終更新者) を解決するためのインタフェースとなっている。 このクラスはプロジェクト毎に作成する必要がある。
(8)	`@Component` アノテーションを付与することで、component-scan 対象になるようにしている。 `@Component` アノテーションを付けずに、Bean 定義ファイルに bean 定義してもよい。
(9)	Entity の操作者 (作成者または最終更新者) のプロパティに設定するオブジェクトを返却する。 上記例では、SpringSecurity によって認証されたログインユーザのユーザ名 (文字列) を Entity の操作者として返却している。

- Object/relational mapping file(`orm.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
```

```
    http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
    version="2.0">

    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <entity-listeners>
                <entity-listener
                    class="org.springframework.data.jpa.domain.support.
↪AuditingEntityListener" /> <!-- (10) -->
            </entity-listeners>
        </persistence-unit-defaults>
    </persistence-unit-metadata>

</entity-mappings>
```

項番	説明
(10)	Entity Listener として、 org.springframework.data.jpa.domain.support.AuditingEntityListener クラスを指定 する。 このクラスで実装しているメソッドにて、Audit 用プロパティに値を設定している。

- infra.xml

```
<jpa:auditing auditor-aware-ref="springSecurityAuditorAware" /> <!-- (11) -->
```

項番	説明
(11)	<jpa:auditing>要素の auditor-aware-ref 属性に、(7) で作成した Entity の操作者を解決するためのクラスの bean を指定する。 上記例では、 SpringSecurityAuditorAware という実装クラスを component-scan しているので、 springSecurityAuditorAware という bean 名を指定している。

@CreatedDate および @LastModifiedDate アノテーションが付与されたフィールドに設定される値は、デフォルト実装だと org.springframework.data.auditing.CurrentDateTimeProvider の getNow() メソッドから返却される java.util.Calendar のインスタンスの値が使用される。

以下に、使用する値の生成方法を変更する場合の拡張例を示す。

```
// (1)
@Component // (2)
public class AuditDateTimeProvider implements DateTimeProvider {

    @Inject
    JodaTimeDateFactory dateFactory;

    // (3)
    @Override
    public Calendar getNow() {
        DateTime currentDateTime = dateFactory.newDateTime();
        return currentDateTime.toGregorianCalendar();
    }

}
```

項番	説明
(1)	org.springframework.data.auditing.DateTimeProvider インタフェースの実装クラスを作成する。
(2)	@Component アノテーションを付与することで、component-scan 対象になるようにしている。 @Component アノテーションを付けずに、Bean 定義ファイルに bean 定義してもよい。
(3)	Entity の操作日時 (作成日時、最終更新日時) のプロパティに設定するインスタンスを返却する。 上記例では、共通ライブラリから提供している org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory から取得したインスタンスを操作日時として返却している。 JodaTimeDateFactory の詳細については、「システム時刻」を参照されたい。

- infra.xml

```
<jpa:auditing
    auditor-aware-ref="springSecurityAuditorAware"
    date-time-provider-ref="auditDateTimeProvider" /> <!-- (4) -->
```

項番	説明
(4)	<jpa:auditing>要素の date-time-provider-ref 属性に、(1) で作成した Entity の操作日時に設定する値を返却するクラスの bean を指定する。 上記例では、 `AuditDateTimeProvider` という実装クラスを component-scan しているので、 `auditDateTimeProvider` という bean 名を指定している。

永続層から Entity を取得する JPQL に共通条件を加える方法

永続層 (DB) から Entity を取得するための JPQL に共通条件を加える仕組みを紹介する。これは JPA の標準仕様ではなく、Hibernate が拡張した仕組み機能である。

注釈: 想定される適用ケース

データの監査やデータの保存期間などの要件により、Entity を削除する際に、レコードの物理削除 (DELETE) ではなく、論理削除 (論理削除フラグの UPDATE) を行うようなアプリケーションが有効である。このケースでは、アプリケーションとしては論理削除されたレコードは一律検索対象から除外する必要があるため、Query ひとつひとつに除外するための条件を加えるより、共通条件として指定した方がよい。

警告: 適用範囲について

この仕組みを適用した Entity を操作するすべての Query に対して、指定した条件が追加される点に注意すること。条件を加えたくない Query が 1 つでもある場合は、この仕組みは使用できない。

Entity を取得する JPQL に共通条件を加える

Repository インタフェースのメソッド呼び出し時に実行される JPQL に対して、共通の条件を加える方法を以下に示す。

- Entity

```
@Entity
@Table(name = "t_order")
@Where(clause = "is_logical_delete = false") // (1)
public class Order implements Serializable {
    // ...
    @Id
    private Integer id;
    // ...
}
```

- Repository インタフェースの findOne メソッドを呼び出した時に発行される SQL

```
SELECT
        -- ....
    FROM
        t_order order0_
   WHERE
        order0_.id = 1
        AND (
            order0_.is_logical_delete = false -- (2)
        );
```

項番	説明
(1)	Entity のクラスアノテーションとして、 `@org.hibernate.annotations.Where` アノテーションを付与し、 clause 属性に共通の条件を指定する。 ここで指定する WHERE 句の書式は、JPQL ではなく、SQL で指定する必要がある。つまり Java オブジェクトのプロパティ名ではなくカラム名を指定する必要がある。
(2)	`@Where` アノテーションで指定した条件が追加されている。

注釈: Dialect 拡張について

`@Where` アノテーション内で SQL 固有のキーワードを指定する場合、Hibernate が SQL 固有のキーワードを一般的な文字列として認識されてしまい、期待した SQL へ変換されないケースがある。SQL 固有のキーワードを利用する場合に、 `Dialect` を拡張して使用する必要がある。

- 標準的なキーワード `true` 、 `false` 、 `unknown` などを登録するための `Dialect` を拡張する

```
package com.example.infra.hibernate;

public class ExtendedPostgreSQL9Dialect extends PostgreSQL9Dialect { // (1)
```

```
    public ExtendedPostgreSQL9Dialect() {
        super();
        // (2)
        registerKeyword("true");
        registerKeyword("false");
        registerKeyword("unknown");
    }
}
```

項番	説明
(1)	Hibernate 4.3 のデフォルトの状態では、SQL のキーワードを正しく認識できない場合がある。例えば、PostgreSQL 向けのキーワードを管理する `org.hibernate.dialect.PostgreSQL9Dialect` にはキーワードとして BOOLEAN 型の `true` 、`false` 、`unknown` などが登録されていないため、一般的な文字列として認識されてしまい、正しい SQL へ変換されない。 そのため、必要に応じて `org.hibernate.dialect.Dialect` を拡張し、キーワードを登録する必要がある。
(2)	`@Where` で利用する可能性のある SQL キーワードを登録する。

- 拡張した Dialect を設定する

```
<bean id="jpaVendorAdapter"
    class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <property name="databasePlatform" value="com.example.infra.hibernate.
↪ExtendedPostgreSQL9Dialect"/> // (3)
    // ...
</bean>
```

項番	説明
(3)	拡張した `Dialect` を `EntityManager` である `JpaVendorAdapter` の `databasePlatform` プロパティの値に設定する。

注釈: 指定可能なクラスについて

`@Where` アノテーションは、`@Entity` が付与されているクラスでのみ有効である。つまり、`@javax.persistence.MappedSuperclass` を付与した基底 Entity クラスに付与しても SQL に付与され

ない。

> **警告:** `@Where` アノテーションは、JPA の標準アノテーションではなく、Hibernate 独自のアノテーションである。

関連 Entity を取得する JPQL に共通条件を加える

Repository インタフェースのメソッド呼び出して取得した Entity の関連 Entity を取得するための JPQL に対して、共通の条件を加える方法を以下に示す。

- Entity

```
@Entity
@Table(name = "t_order")
@Where(clause = "is_logical_delete = false")
public class Order implements Serializable {
    // ...
    @Id
    private Integer id;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval =␣
↪true)
    @OrderBy
    @Where(clause="is_logical_delete = false") // (1)
    private Set<OrderItem> orderItems;
    // ...

}
```

- 関連 Entity にアクセスした際に発行される SQL(Lazy Load)

```
SELECT
        -- ...
    FROM
        t_order_item orderitems0_
    WHERE
        (orderitems0.is_logical_delete = false) -- (2)
        AND orderitems0_.order_id = 1
    ORDER BY
        orderitems0_.item_code ASC
        ,orderitems0_.order_id ASC;
```

- Entity と関連 Entity を同時に取得する際に発行される SQL(Eager Load / Join Fetch)

```
SELECT
    -- ...
    FROM
        t_order order0_
            LEFT OUTER JOIN t_order_item orderitems1_
                ON order0_.id = orderitems1_.order_id
                AND (orderitems1_.is_logical_delete = false) -- (2)
    WHERE
        order0_.id = 1
        AND (
            order0_.is_logical_delete = false
        )
    ORDER BY
        orderitems1_.item_code ASC
        ,orderitems1_.order_id ASC;
```

項番	説明
(1)	関連 Entity のフィールドアノテーションとして、`@org.hibernate.annotations.Where` アノテーションを付与し、 clause 属性に共通の条件を指定する。 ここで指定する WHERE 句の書式は、JPQL ではなく、SQL で指定する必要がある。つまり Java オブジェクトのプロパティ名ではなくカラム名を指定する必要がある。
(2)	`@Where` アノテーションで指定した条件が追加されている。

注釈: 指定可能な関連の種類について

関連 Entity を取得する際の SQL に共通の条件を加えることができるのは、 `@OneToMany` および `@ManyToMany` の関連をもつ Entity に対してのみとなる。

警告: `@Where` アノテーションは、JPA の標準アノテーションではなく、Hibernate 独自のアノテーションである。

複数の **PersistenceUnit** を使用する方法

課題: TBD

今後、以下の内容を追加する予定である。

- 複数の PersistenceUnit の使用例。

Native クエリの使用方法

課題: TBD

今後、以下の内容を追加する予定である。

- Native クエリを使用した Query の実装例。

6.4 排他制御

6.4.1 Overview

排他制御とは、複数のトランザクションから同じデータに対して、同時に更新処理が行われる際に、データの整合性を保つために行う処理のことである。

複数のトランザクションから同じデータに対して、同時に更新処理が行われる可能性がある場合は、基本的に排他制御を行う必要がある。ここで言うトランザクションとは、かならずしもデータベースとのトランザクションとは限らず、ロングトランザクションも含まれる。

注釈: ロングトランザクションとは

データの取得とデータの更新を、別々のデータベーストランザクションとして行う際に発生するトランザクションのことである。

具体例としては、取得したデータを編集画面に表示し、画面で編集した値をデータベースに更新するようなアプリケーションで発生する。

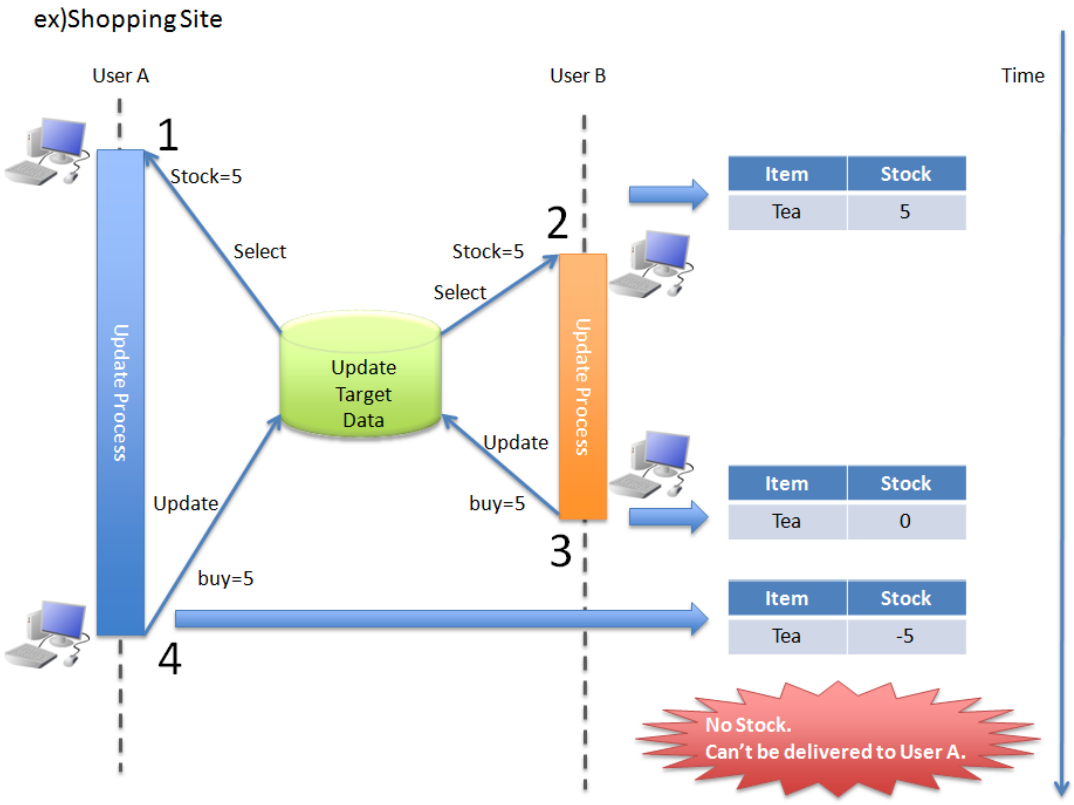
本節では、データベース上で管理されているデータに対する排他制御について、説明する。
しかし、データベース以外で管理されているデータ (例えば、メモリ、ファイルなど) についても、同様に排他制御を行う必要があることに留意すること。

排他制御の必要性

まず、排他制御の必要性を理解してもらうために、排他制御を行わなかった際に発生する問題について、具体例を 3 つ挙げて説明する。

Problem1

ここでは、ショッピングサイトにて、ユーザから Tea の注文を受け付ける場合の例を示す。



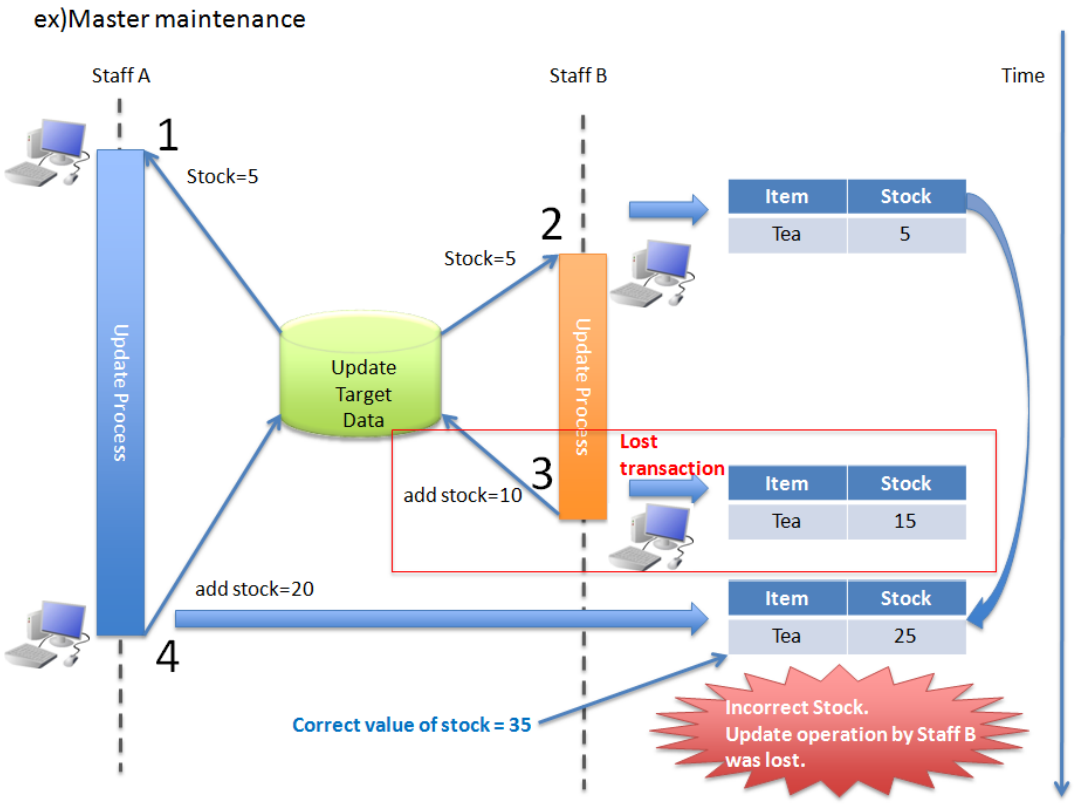
項番	UserA	UserB	説明
1.	○	-	User A が、商品画面にて Tea の在庫が 5 個あることを確認する。
2.	-	○	User B が、商品画面にて Tea の在庫が 5 個あることを確認する。
3.	-	○	User B が Tea を 5 個注文する。DB 上の Tea の在庫を-5 し、Tea の在庫は 0 になる。
4.	○	-	User A が Tea を 5 個注文する。DB 上の Tea の在庫を-5 し、Tea の在庫は-5 となる。

User A の注文は受け付けられたが、実際の在庫が無いため、謝りの連絡を入れることになる。

テーブルで管理している **Tea** の在庫数についても、実際の **Tea** の在庫数と異なる値 (マイナス値) になってしまう。

Problem2

ここでは、ショッピングサイトで Tea の在庫数を管理するスタッフが、Tea の在庫数を表示し、仕入れた Tea の数をクライアントで計算して、Tea の在庫数を更新する場合の例を示す。

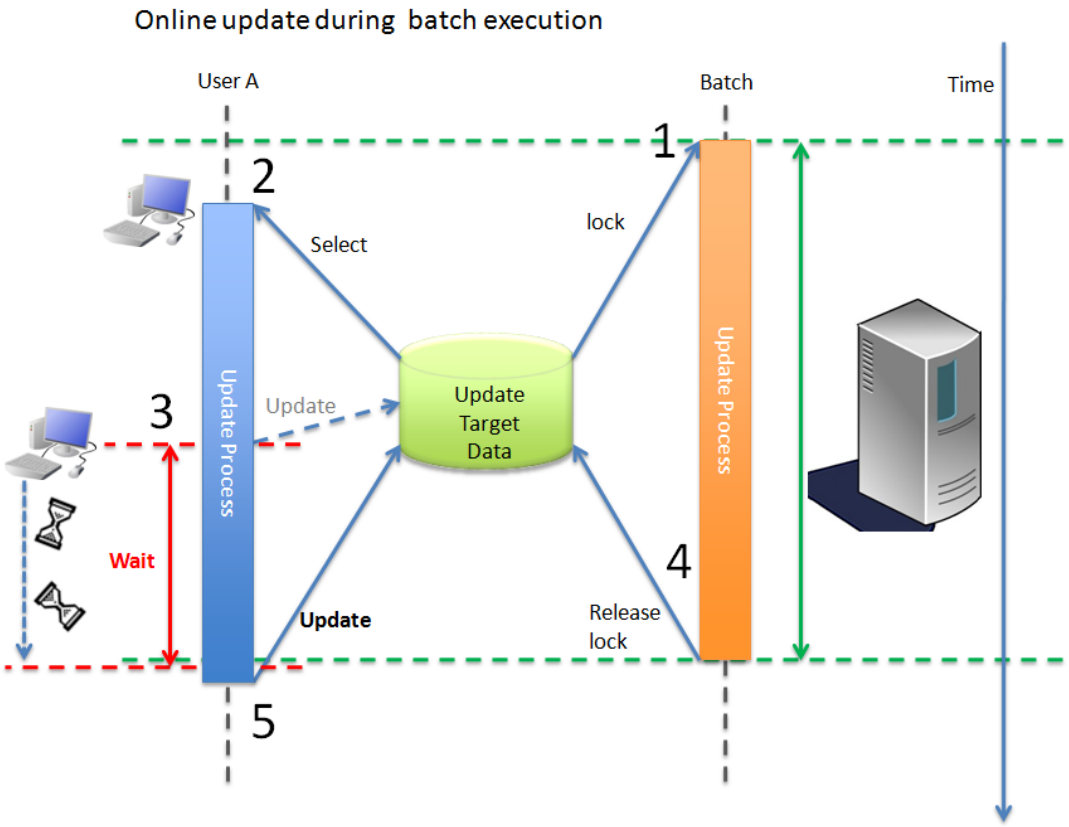


項番	UserA	UserB	説明
1.	○	-	Staff A が Tea の在庫が 5 個あることを確認する。
2.	-	○	Staff B が Tea の在庫が 5 個あることを確認する。
3.	-	○	Staff B が Tea を 10 個仕入れ、在庫数をクライアントで 5 ＋ 10=15 個と計算して更新する。
4.	○	-	Staff A が Tea を 20 個仕入れ、在庫数をクライアントで 5 ＋ 20=25 個と計算して更新する。

3 の処理で追加した **10** 個の仕入れが無くなってしまい、実際の在庫数 **(35** 個**)** と合わなくなってしまう。

Problem3

ここでは、バッチ処理によってロックされているデータに対して、オンライン処理で更新する例を示す。



項番	UserA	Batch	説明
1.	-	○	Batch がテーブルの更新対象の該当行（ここでは仮に全ての行とする。）をロックし、他の処理で更新できないようにする。
2.	○	-	User A が更新情報を検索する。この時点で Batch はコミットされていないため、Batch 更新前の情報が取得できる。
3.	○	-	User A が更新要求をするが、Batch にロックされているため、更新が待たされる。
4.	-	○	Batch が処理を終えてロックを解放する。
5.	○	-	User A の待たされていた更新処理が、実行可能となり更新処理を実行する。

User A は **Batch** 終了を待たされた後に、更新処理を実行する。しかし、**User A** の取得した元のデータは、**Batch** の更新前のデータであり、**Batch** で更新した情報を上書く可能性がある。

また、**Batch** 時間はオンライン処理と比べると長いものが多く、ユーザが待たされる時間が長くなる。

トランザクションの分離レベルによる排他制御

排他制御の必要性 で挙げた 3 つの問題をすべて解決するための最も簡単な方法は、データベースへの処理を一つひとつ順番に（シリアルに）実行されるようにすることである。

このようにシリアルに処理させることで、トランザクションが互いに影響を及ぼし合わなくなる。

しかしながら、シリアルに処理させる場合、単位時間内に実行可能なトランザクション数が減少するため、パフォーマンスが低下することになる。

ANSI/ISO SQL 標準では、トランザクションの分離レベル（各トランザクションがそれぞれどの程度互いに影響を及ぼし合うか）を表す指標を定義している。以下に、トランザクションの分離レベルを 4 つ示す。併せて、各分離レベルで起こりうる現象について説明する。

項番	分離レベル	ダーティ・リード DRITY READ	再読込不可能読取 NON-REPEATABLE READ	ファントム・リード PHANTOM READ
1.	未コミット読込 READ UNCOMMITTED	有	有	有
2.	コミット済読込 READ COMMITTED	無	有	有
3.	再読込可能読取 REPEATABLE READ	無	無	有
4.	直列化 SERIALIZABLE	無	無	無

ちなみに：ダーティ・リード（DRITY READ）

まだコミットされていないトランザクションが書き込んだデータを、別のトランザクションが読み込む現象の

ことである。

ちなみに：再読込不可能読取（NON-REPEATABLE READ）

同一トランザクション内で同じレコードを 2 度読み込むような場合、1 度目と 2 度目の読み込みの間に他トランザクションがコミットすると、1 度目に読み込んだ内容と 2 度目に読み込んだ内容が異なる可能性がある。複数回の読み込みの結果が、他のトランザクションのコミットのタイミングによって変わることである。

ちなみに：ファントム・リード（PHANTOM READ）

同一トランザクション内で、同じレコードを 2 回読み込む間に、他のトランザクションがレコードを追加、または削除することにより、2 回目の読み込みで 1 回目と取得レコード数（内容）が異なることである。

上記の表に定義されている分離レベルは、下にいくほどトランザクションの分離レベルが高くなる。
分離レベルが高ければ、データは安全に守られるが、ロック待ちが多くなり、パフォーマンスが低下する。
SERIALIZABLE は、アクセス頻度がかなり低い場合を除き、選択すべきでない。
その理由は、SELECT を含め、すべてのデータアクセスが、一つずつ順番に行われるためである。

トランザクション間の分離性と同時実行性は、トレードオフの関係である。
すなわち、分離レベルを高くすれば同時実効性が下がり、分離レベルを下げると、同時実効性が上がる。
そのため、アプリケーションの要件に合わせて、トランザクションの分離性と同時実行性のバランスをとる必要がある。

使用するデータベースにより、サポートされている分離レベルは違うため、使用するデータベースの特性を理解する必要がある。
以下に、データベース毎でサポートされている分離レベルと、デフォルト値を示す。

項番	データベース	READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SERIALIZABLE
1.	Oracle	×	○ (default)	×	○
2.	PostgreSQL	×	○ (default)	×	○
3.	DB2	○	○ (default)	○	○
4.	MySQL InnoDB	○	○	○ (default)	○

データの整合性を保ちつつ、分離性と同時実行性のバランスをとる場合、データベースのロック機能を使用して排他制御を行う必要がある。以下に、データベースのロック機能について説明する。

データベースのロック機能による排他制御

更新対象のデータを適切な方法でロックする必要がある。その理由は、下記 2 点の通りである。

- データベース上で管理されているデータの整合性を保つため
- 更新処理が競合しないようにするため

データベース上で管理されているデータをロックする方法は、下記の通り 3 種類ある。

アーキテクトは、これらのロックの特徴を十分に理解した上で、アプリケーションの特性にあったロックの方法を採用すること。

表 6.21　ロックの種類

項番	ロック種類	適用ケース	特徴
1.	RDBMS による自動的なロック	• データの更新条件として、データの整合性を保証するために必要な条件を指定できる場合。 • 同一データに対する同時実行数が少なく、更新処理も短い時間でおわる場合。	• チェックと更新処理を一つの SQL で実行するため、効率的である。 • 楽観ロックに比べ、データの整合性を保証するための条件を個別に検討する必要がある。
2.	楽観ロック	• 事前に取得したデータが他のトランザクションによって更新されていた場合に、更新内容を確認させる必要がある場合。 • 同一データに対する同時実行数が少なく、更新処理も短い時間でおわる場合。	• 取得したデータに対して、他のトランザクションからの更新が行われていないことが保証される。 • テーブルに Version を管理するためのカラムを定義する必要がある。
3.	悲観ロック	• 長い時間ロックされる可能性があるデータに対して更新する場合。 • 楽観ロックが使用できない (Version を管理するためのカラムが定義できない) ため、処理としてデータの整合性チェックを行う必要がある場合。 • 同一データに対する同時実行数が多く、更新処理も長い時間実行される可能性がある場合。	• 他のトランザクションの処理結果によって処理が失敗する可能性がなくなる。 • 悲観ロックを取得するための select 文を発行する必要があるので、その分コストがかかる。

注釈: ロックの種類の採用基準について

どの手法を採用するかについて、アーキテクトが、機能要件および性能要件を考慮して決定すること。

- 画面にデータを戻し、画面上でデータを変更するような、データベースとのトランザクションが切れて、次のトランザクションでデータが変わっていないことを保証するためには、楽観ロックが必要となる。
- 1 トランザクション内でロックをかける必要がある場合は、悲観ロックと楽観ロックの両方で実現できるが、悲観ロックを使用した場合、データベース内のロック制御処理が行われるため、データベース内の処理コストが高くなる可能性がある。特に問題がない場合は、楽観ロックの方がよい。
- 更新頻度が高い処理で、1 トランザクション内で多くのテーブルを更新する場合は、楽観ロックを使用すると、ロックを取得するための待ち時間は最小限に抑えることが出できるが、途中で排他エラーとなる可能性があるため、エラーが発生するポイントが増える。悲観ロックを使用すると、ロックを取得す

るまでの待ち時間が長くなる可能性はあるが、ロックを取得した後の処理で排他エラーが発生することはないため、エラーが発生するポイントが減る。

ちなみに：業務トランザクションについて

実際のアプリケーション開発では、業務フローレベルのトランザクションに対して、排他制御が必要になる場合もある。業務フローレベルのトランザクションとなる代表例としては、旅行代理店のカウンタで、お客様と話をしながら予約作業を進めていく際に使用するアプリケーションがあげられる。

旅行予約を行う場合、鉄道、宿泊施設、さらに追加プランなどを話しながら決めていくことになる。その際に、予約することに決めた宿泊施設や追加プランが、他の利用者に予約されないようにする仕組みが必要になる。このような場合は、テーブルにステータスを持たせ、仮予約 -> 予約 のように更新し、仮予約中の場合も、他の利用者から更新されないようにする必要がある。

業務トランザクションに対する排他制御については、業務設計や機能設計として検討・設計すべき箇所になるので、本節の説明範囲からは省いている。

データベースの行ロック機能による排他制御

ほとんどのデータベースでは、レコードを更新（UPDATE,DELETE）した場合、コミットまたはロールバックされるまで、他のトランザクションからの更新を待機させるための行ロックが取得される。
そのため、更新件数が想定通りであれば、データの整合性を保証することができる。

この特性を活かし、更新時の WHERE 句に対して、データの整合性を担保するための条件を指定することで、排他制御を行うことができる。
以下に、データベース毎の、更新時の行ロックのサポート状況を示す。

項番	データベース	確認 Version	デフォルト設定時のロック	備考
1.	Oracle	11	行ロック	ロック分メモリ使用量が増大する。
2.	PostgreSQL	9	行ロック	メモリ上に変更された行の情報を記憶しないので、同時にロックできる行数に、上限はない。ただし、テーブルに書き込むため、定期的に VACUUM しなければならない。
3.	DB2	9	行ロック	ロック分メモリ使用量が増大する。
4.	MySQL InnoDB	5	行ロック	ロック分メモリ使用量が増大する。

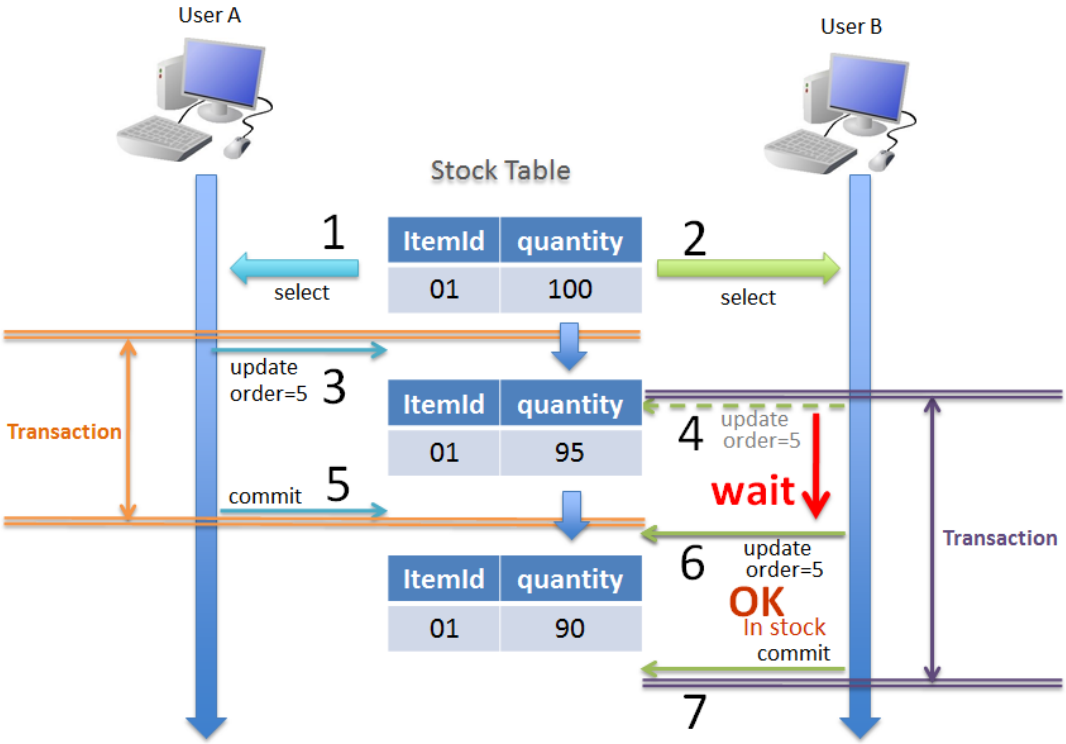
データベースの行ロック機能による排他制御は、他のトランザクションによって更新した内容を確認する必要がない場合に使用することができる。

例えば、ショッピングサイトの購入処理にて、購入した商品の個数を、商品の在庫数を管理するレコードからマイナスするような処理が挙げられる。

ステータス管理を管理する処理などでは、前のステータスが重要になるので、この方法で排他制御を実現することを推奨しない。

以下に、具体例を示す。シナリオは、以下の通りである。

- ショッピングサイトで User A,User B ともに同じ商品の購入画面を同時に表示する。その際に、Stock Table から取得した在庫数も表示されている。
- 買いたい商品を 5 個ずつ同時に購入したが、少し UserA の方が早く購入ボタンを押下したため、User A が先に購入し、User B が次に購入する。

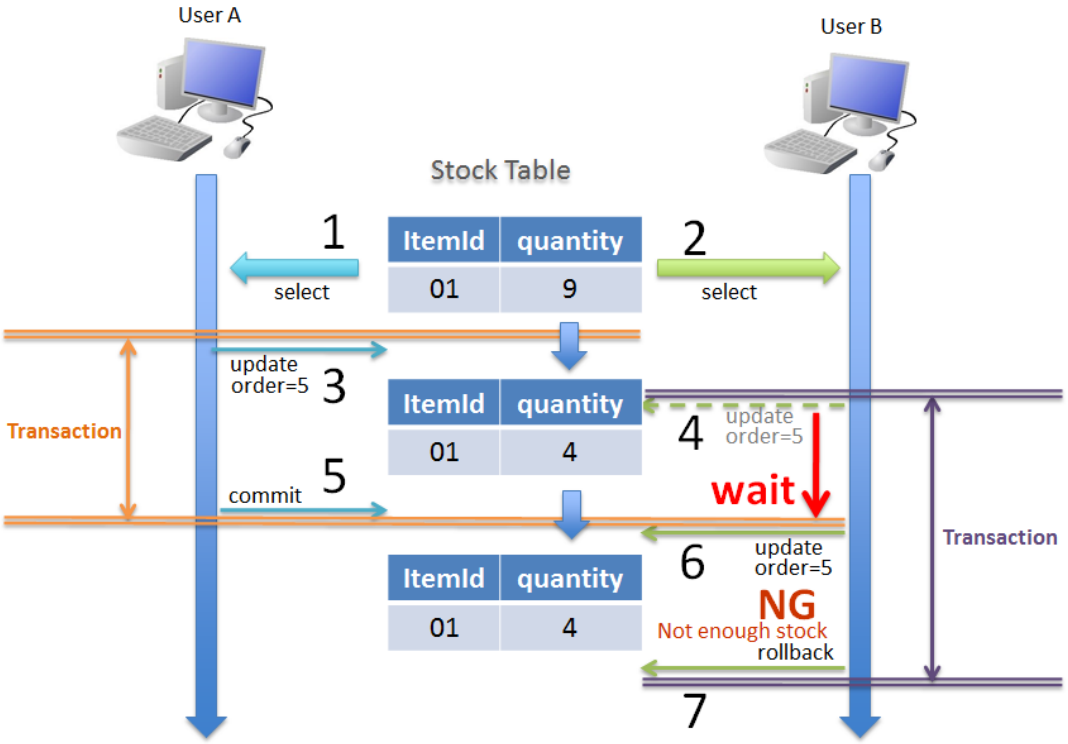


項番	UserA	UserB	説明
1.	○	-	User A が、商品の購入画面を表示する。在庫数が 100 個で画面に表示されている。 `select quantity from Stock where ItemId = '01'`
2.	-	○	User B が、商品の購入画面を表示する。在庫数が 100 個で画面に表示されている。 `select quantity from Stock where ItemId = '01'`
3.	○	-	User A が、ItemId=01 の商品を 5 個購入する。Stock Table から個数を-5 する。 `Update from Stock set quantity = quantity - 5 where ItemId='01' and quantity >= 5`
4.	-	○	User B が、ItemId=01 の商品を 5 個購入する。Stock Table から個数を-5 しようとするが、User A のトランザクションが終了していないので、User B の購入処理が待たされる。
5.	○	-	User A のトランザクションをコミットする。
6.	-	○	User A のトランザクションがコミットされたため、4 で待たされていた UserB の購入処理が再開する。 この時、在庫は画面で見ると、個数は 100 ではなく、95 になっているが、購入数 (上記例では、5 個) 以上の在庫が残っているため、Stock Table から個数を-5 する。 `Update from Stock set quantity = quantity - 5 where ItemId='01' and quantity >= 5`
7.	-	○	User B のトランザクションをコミットする。

注釈: ポイント

SQL 内で減算 (`quantity - 5`) と、更新条件 (`and quantity >= 5`) の指定を行うことが、ポイントとなる。

上と同じシナリオで、商品の購入画面を表示した際の在庫数が、9 個だった場合、User B の更新処理が再開した時点の在庫数が、4 個のため、`quantity >= 5` を満たさないので、更新件数が 0 件となる。
アプリケーションでは、更新件数が 0 件の場合、購入処理をロールバックし、User B に再度実行を促す。



注釈: ポイント

アプリケーションで更新件数をチェックし、想定件数と異なる場合にエラーを発生させ、トランザクションをロールバックすることが、ポイントとなる。

この方法でロックする場合、参照した情報が変わっていても条件次第で処理を進めることができ、かつ、データベースの機能によってデータの整合性を保証することができる。

楽観ロックによる排他制御

楽観ロックとは、データそのものに対してロックは行わずに、更新対象のデータが、データ取得時と同じ状態であることを確認してから更新することで、データの整合性を保証する手法である。

楽観ロックを使用する場合は、更新対象のデータが、データ取得時と同じ状態であることを判断するために、Version を管理するためのカラム (Version カラム) を用意する。

更新時の条件として、データ取得時の Version と、データ更新時の Version を同じとすることで、データの整合性を保証することができる。

注釈: Version カラムとは

レコードの更新回数を管理するためのカラムで、レコード挿入時に 0 を設定し、更新成功時にインクリメントしていく楽観ロック用のカラムである。Version カラムは、数値以外に最終更新タイムスタンプで代用することもできる。しかし、タイムスタンプを用いると、同時に処理が実行された際の、一意性が保証されない。そこで、確実な一意性を求める場合、Version カラムは、数値を使用する必要がある。

楽観ロックによる排他制御は、他のトランザクションによって更新されていた場合に、更新内容を確認させる必要がある場合に使用する。

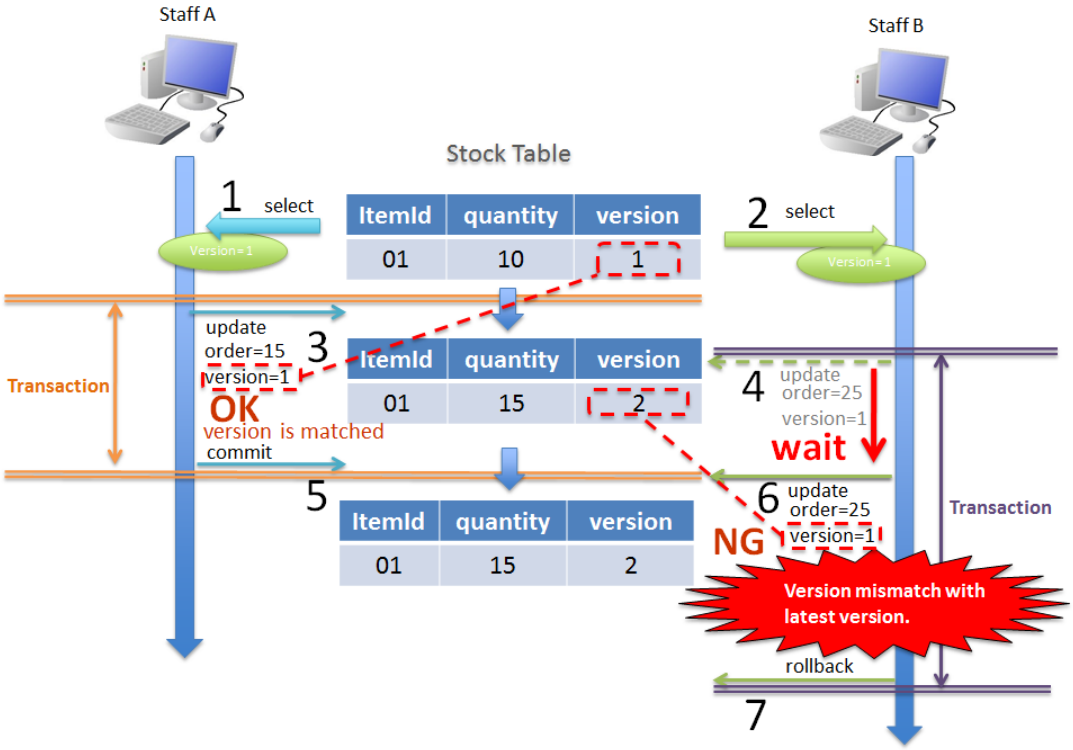
例えば、ワークフローアプリケーションにおいて、申請者と承認者が同時に操作（引き戻しと承認）を行った場合を想像してほしい。

この時、楽観ロックによる排他制御を行うことで、操作の前後で状態が変わっているため、操作が完了しなかったことを、申請者と承認者に通知することができる。

警告: 楽観ロックを行う場合、ID と Version 以外の条件を加えて更新・削除するのは適切でない。なぜなら更新できなかった場合に、Version が一致しないことが理由なのか、別の条件に一致しないのが理由なのか、判断できないためである。更新条件として別の条件がある場合は、事前の処理として条件を満たしているか、チェックを行う必要がある。

具体例を、以下に示す。シナリオは、以下の通りである。

- ショッピングサイトの在庫数を管理するスタッフ (Staff A, Staff B) が、それぞれ商品を仕入れる。Staff A が 5 個、Staff B が 15 個仕入れたものとする。
- 仕入れた商品を、在庫管理システムに反映するために、在庫管理画面を表示する。その際、在庫管理システムで管理されている在庫数が表示される。
- それぞれ表示された在庫数に対して、仕入れた数を加算した値を更新フォームに入力し、更新を行う。



項番	Staff A	Staff B	説明
1.	○	-	Staff A が、商品の在庫管理画面を表示する。在庫数は 10 個と画面に表示されている。参照したデータの Version は 「1」 である。
2.	-	○	Staff B が、商品の在庫管理画面を表示する。在庫数は 10 個と画面に表示されている。参照したデータの Version は 「1」 である。
3.	○	-	Staff A が、画面に表示されていた在庫数 10 に対して、仕入れた 5 個を加算し、変更後の在庫数を 15 個で更新する。更新条件として、参照したデータの Version を含める。 `UPDATE Stock SET quantity = 15, version = version + 1 WHERE itemId = '01' and version = 1`
4.	-	○	Staff B が、画面に表示されていた在庫数 10 に対して仕入れた 15 個を加算し、変更後の在庫数を 25 個で更新しようとするが、Staff A のトランザクションが終了していないので待たされる。更新条件として、参照したデータの Version を含める。
5.	○	-	Staff A のトランザクションをコミットする。 この時点で、**Version は 2 になる。**
6.	○	-	Staff A のトランザクションがコミットされたため、4 で待たされていた Staff B の更新処理が再開する。この時、Stock Table のデータの Version が 「2」 になっているため、更新結果が 0 件となる。更新結果が 0 件の場合は排他エラーとする。 `UPDATE Stock SET quantity = 25, version = version + 1 WHERE itemId = '01' and version = 1`
7.	○	-	Staff B のトランザクションをロールバックする。

注釈: ポイント

SQL 内で Version のインクリメント (`version + 1`) と、更新条件 (`and version = 1`) の指定を行うことが、ポイントとなる。

悲観ロックによる排他制御

悲観ロックとは、更新対象のデータを取得する際にロックをかけることで、他のトランザクションから更新されないようにする手法である。

悲観ロックを使用する場合は、トランザクション開始直後に更新対象となるレコードのロックを取得する。

ロックされたレコードは、トランザクションが、コミットまたはロールバックされるまで、他のトランザクションから更新されないため、データの整合性を保証することができる。

表 6.23　RDBMS 別の悲観ロック取得方法

項番	データベース	悲観ロック方法
1.	Oracle	FOR UPDATE
2.	PostgreSQL	FOR UPDATE
3.	DB2	FOR UPDATE WITH
4.	MySQL	FOR UPDATE

注釈: 悲観ロックのタイムアウトについて

悲観ロックには、悲観ロック取得時に他のトランザクションによってロックが取得されていた場合に、どのような動作にするかをオプションとして指定することがある。Oracle の場合は、

- デフォルトでは、`select for update [wait]` となり、ロックが解除されるまで待つ。
- `select for update nowait` とすると、他にロックされている場合は、即時にリソースビジーのエラーとなる。
- `select for update wait 5` とすると 5 秒待ち、5 秒間ロックが解除されない場合は、リソースビジーのエラーが返却される。

DB により機能に差はあるが、悲観ロックを使用する際は、どの手法を採用するか検討が必要である。

注釈: **JPA(Hibernate)** を使用する場合

悲観ロックの取得方法はデータベースによって異なるが、その差分は JPA(Hibernate) によって吸収される。Hibernate のサポートしている RDBMS については、 Hibernate Developer Guide を参照されたい。

悲観ロックによる排他制御は、以下 3 ケースのいずれかに当てはまる場合に使用する。

1. 更新対象のデータが複数のテーブルに分かれて管理されている。

   更新対象のテーブルが複数のテーブルに分かれている場合、各テーブルに対して更新が終わるまでの間に、他のトランザクションから更新がされないことを保証するために、必要となる。

2. 更新処理を行う前に取得したデータの状態をチェックする必要がある。

   チェック処理が終わった後に、他のトランザクションから更新がされていないことを保証するために、必要となる。

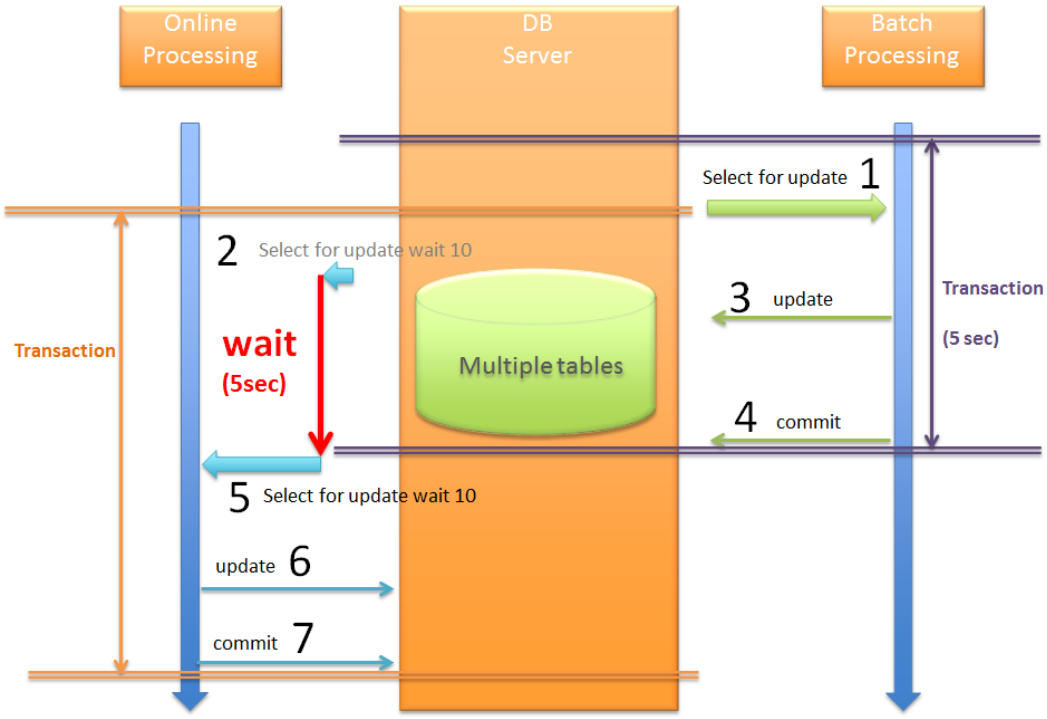
3. バッチ実行中にオンラインの処理が実行されることがある。

   バッチ処理では、実行途中に排他エラーが発生しないようにするために、更新対象となるデータのロックを一括で取得することがある。

   一括で取得されたロックが取得された場合、オンラインの処理が待たされる時間が長くなる可能性がある。その場合、タイムアウト時間を指定して、悲観ロックを使用するのが妥当である。

具体例を以下に示す。シナリオは、以下の通りである。

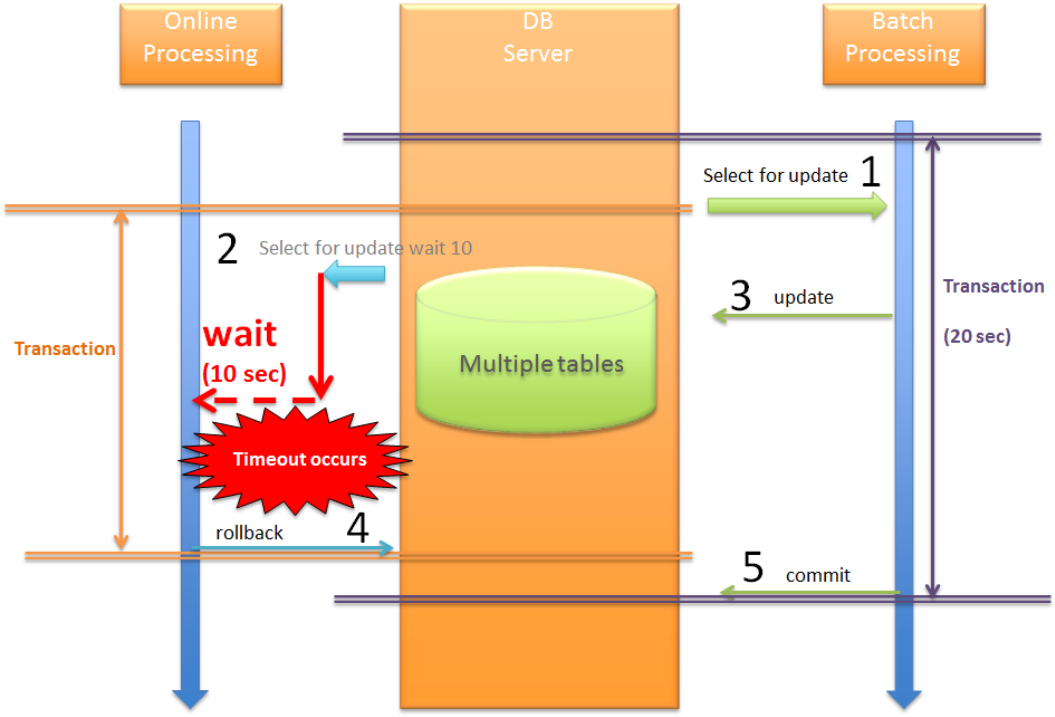
- バッチ処理が既に実行済みで、オンラインで更新するデータを悲観ロックしている。
- オンライン処理は 10 秒のタイムアウト時間を指定して、更新対象のデータのロックを取得する。
- バッチ処理は 5 秒後 (タイムアウト前) に終了する。



項番	Online	Batch	説明
1.	-	○	バッチ処理が、オンライン処理で更新するデータの悲観ロックを取得する。
2.	○	-	オンライン処理が、更新対象のデータの悲観ロックを行うが、バッチ処理のトランザクションによって悲観ロックされているので待たされる。 `SELECT * FROM Stock WHERE quantity < 5 FOR UPDATE WAIT 10`
3.	-	○	バッチ処理が、データを更新する。
4.	-	○	バッチ処理のトランザクションをコミットする。
5.	○	-	バッチ処理のトランザクションがコミットされたため、オンライン処理の処理が再開する。取得されるデータはバッチ処理の更新結果が反映されているので、データ不整合が発生することはない。
6.	○	-	オンライン処理が、データを更新する。
7.	○	-	オンライン処理のトランザクションをコミットする。

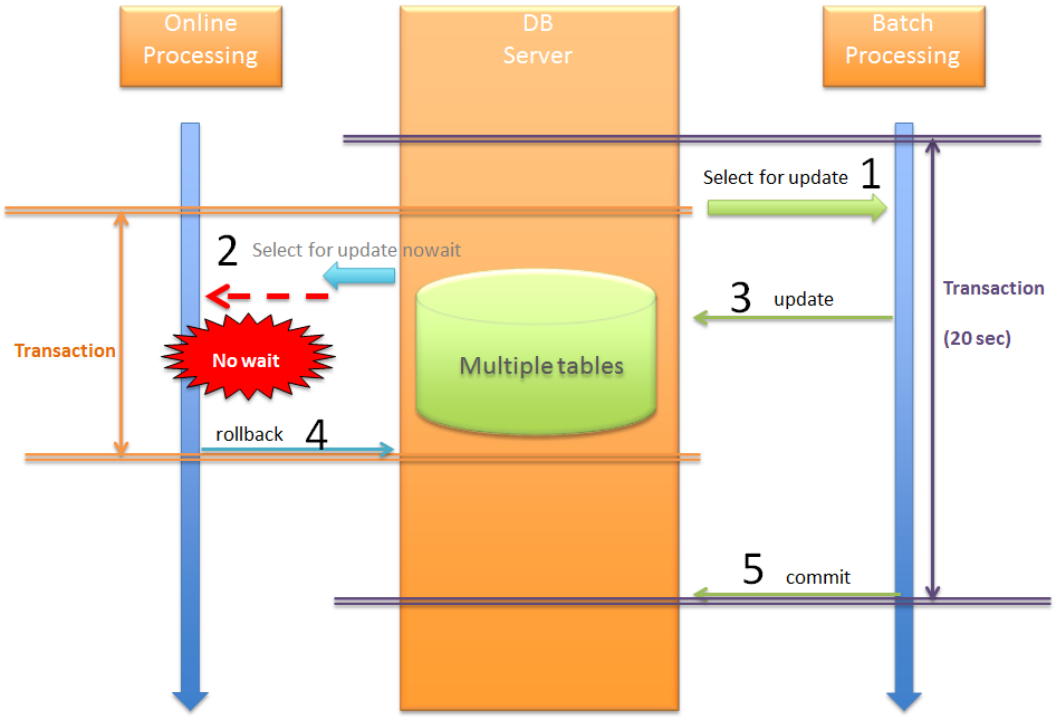
以下は、タイムアウトとなった場合の流れとなる。

バッチ処理の終了まで待たずに排他エラーとなる。



以下は、悲観ロックの取得待ちを行わない設定のとき、他のトランザクションによって、悲観ロックが取得されていた場合の流れとなる。

悲観ロックの解放を待つことなく、すぐに排他エラーとなる。



バッチ処理とオンライン処理が競合する可能性があり、かつバッチ処理の処理時間が長くなる場合は、悲観排他のタイムアウト時間を指定することを推奨する。タイムアウト時間については、オンライン処理の処理要件に応じて決めること。

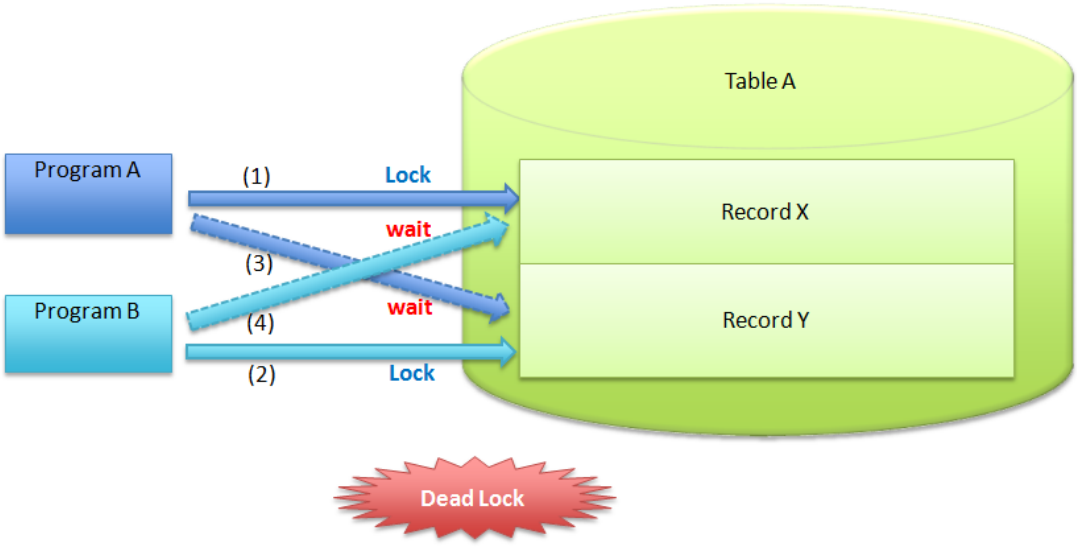
デッドロックの予防

データベースのロック機能を使用する場合、同一トランザクション内で複数のレコードを更新すると、以下2通りの、デッドロックが発生する可能性があるため、注意する必要がある。

- テーブル内でのデッドロック
- テーブル間でのデッドロック

テーブル内でのデッドロック

以下 (1)～(5) の流れで、複数のトランザクションから、同一テーブルのレコードに対してロックを行うと、デッドロックとなる。



項番	Program A	Program B	説明
(1)	○	-	Program A は、Record X に対するロックを取得する。
(2)	○	-	Program B は、Record Y に対するロックを取得する。
(3)	○	-	Program A は、Program B のトランザクションによってロックされている Record Y に対してロックの取得を試みるが、(2) のロック状態が解放されていないので、解放待ちの状態となる。
(4)	-	○	Program B は、Program A のトランザクションによってロックされている Record X に対してロックの取得を試みるが、(1) のロック状態が解放されていないので、解放待ちの状態となる。
(5)	-	-	Program A と Program B が、お互いに保持しているロックに対して解放待ちの状態となるため、デッドロックとなる。デッドロックが発生した場合、データベースによって検知されエラーとなる。

注釈: デッドロックの解決方法について

タイムアウトやリトライ実施での解消する方法もあるが、同一テーブル上でのレコードの更新順序にルールを決めることが重要である。1 行ずつ更新する場合は、PK(PRIMARY KEY) 順の若い順に更新するなどのルールを定めること。

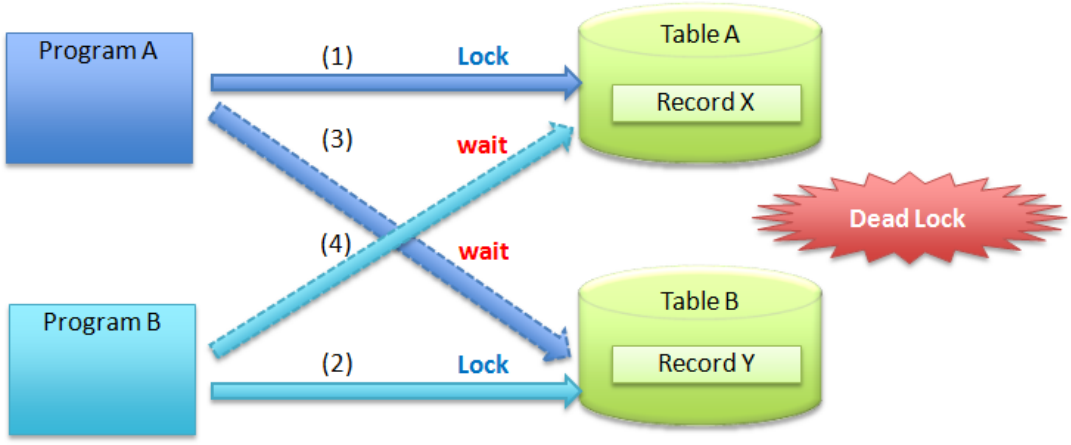
仮に Program A も Program B も Record X から更新するというルールに準じていれば、上記テーブル内

でのデッドロックの図のようなデッドロックは発生しなくなる。

テーブル間でのデッドロック

以下 (1)〜(5) の流れで、複数のトランザクションから、別テーブルのレコードに対してロックを行うと、デッドロックとなる。

基本的な考え方は、 テーブル内でのデッドロック と同じである。



項番	Program A	Program B	説明
(1)	○	-	Program A は、Table A の Record X に対するロックを取得する。
(2)	○	-	Program B は、Table B の Record Y に対するロックを取得する。
(3)	○	-	Program A は、Program B のトランザクションによってロックされている Table B の Record Y に対してロックの取得を試みるが、(2) のロック状態が解放されていないので、解放待ちの状態となる。
(4)	-	○	Program B は、Program A のトランザクションによってロックされている Table A の Record X に対してロックの取得を試みるが、(1) のロック状態が解放されていないので、解放待ちの状態となる。
(5)	-	-	Program A と Program B が、お互いに保持しているロックに対して解放待ちの状態となるため、デッドロックとなる。デッドロックが発生した場合、データベースによって検知されエラーとなる。

注釈: デッドロックの解決方法について

タイムアウトやリトライ実施での解消する方法もあるが、テーブルを跨った際も、更新順序をルール化しておくことが重要である。

仮に Program A も Program B も Table A から更新するというルールに準じていれば、上記テーブル間でのデッドロックの図のような、デッドロックは発生しなくなる。

警告: 注意としては、どの方法を採用したとしても、レコードをロックする順序により、デッドロックが発生する可能性がある。テーブル、レコードのロック順序については、ルールを決めること。

6.4.2 How to use

ここからは、O/R Mapper を使用した排他制御の実現方法について説明を行う。

使用する O/R Mapper の実装方法を確認されたい。

- *MyBatis3* 使用時の実装方法
- *JPA(Spring Data JPA)* 使用時の実装方法

また、排他エラーのハンドリング方法については、

- 排他エラーのハンドリング方法

を参照されたい。

MyBatis3 使用時の実装方法

RDBMS の行ロック機能

RDBMS の行ロック機能を使って排他制御を行う場合は、SQL の中で、

- SET 句に指定する更新内容
- WHERE 句に指定する更新条件

を意識する必要がある。

- Repository インタフェースにメソッドを定義する。

```
public interface StockRepository {
    // (1)
    boolean decrementQuantity(@Param("itemCode") String itemCode,
                              @Param("quantity") int quantity);
}
```

項番	説明
(1)	Repository インタフェースに、RDBMS の行ロック機能を使ってデータを更新するメソッドを定義する。 上記例では、在庫数を減らすためのメソッドを定義している。在庫数の減らす事ができた場合は、`true` が返却される。

- RDBMS の行ロック機能を使った排他制御が有効となる SQL を定義する。

```
<!-- (2) -->
<update id="decrementQuantity">
<![CDATA[
    UPDATE
        m_stock
    SET
        /* (3) */
        quantity = quantity - #{quantity}
    WHERE
        item_code = #{itemCode}
    AND
        /* (4) */
        quantity >= #{quantity}
]]>
</update>
```

項番	説明
(2)	RDBMS の行ロック機能を使ってデータを更新するためのステートメント (SQL) を定義する。 上記例では、在庫数を減らすための SQL を定義している。 RDBMS の行ロック機能を使う場合は、 • 他のトランザクションが同一データに対してロックを取得している場合は、ロックが解放 (コミット or ロールバック) された後に SQL が実行される。 • 在庫数を減らすことに成功した場合は、RDBMS の行ロックが取得され、他のトランザクションからの更新がロックされる。 という動作になるため、データを安全に更新する事ができる。
(3)	在庫数の減算処理 (`quantity = quantity - #{quantity}`) は、SQL の中で行う。
(4)	更新条件として、「在庫数が注文数以上ある事 (`quantity >= #{quantity}`)」を加える。

- Repository のメソッドを呼び出し、RDBMS の行ロック機能を使用してデータを安全に更新する。

```
// (5)
boolean updated = stockRepository.decrementQuantity(itemCode,␣
↪quantityOfOrder);
// (6)
if (!updated) {
    // (7)
    ResultMessages messages = ResultMessages.error().add(ResultMessage
            .fromText("Not enough stock. Please, change quantity."));
    throw new BusinessException(messages);
}
```

項番	説明
(5)	Repository のメソッドを呼び出し、更新処理を行う。
(6)	Repository のメソッドの呼び出し結果を判定する。 `false` の場合、更新条件を充たしていないため、在庫数が不足していることになる。
(7)	業務エラーを発生させる。 上記例では、ビジネスルールのチェック (在庫数チェック) を排他制御しながら行っているだけなので、更新条件を充たさない場合は、排他エラーではなく業務エラーとしている。 発生させた業務エラーは、Controller で適切にハンドリングすること。

楽観ロック

MyBatis3 では、ライブラリとして楽観ロックを行う仕組みは提供していない。
そのため、楽観ロックを行う場合は、SQL の中でバージョンを意識する必要がある。

- Entity にバージョン管理用のプロパティを定義する。

```
public class Stock implements Serializable {
    private static final long serialVersionUID = 1L;

    private String itemCode;
    private int quantity;
    // (1)
    private long version;

    // ...

}
```

項番	説明
(1)	Entity にバージョン管理用のプロパティを用意する。

- Repository インタフェースにメソッドを定義する。

```
public interface StockRepository {
    // (2)
    Stock findOne(String itemCode);
    // (3)
    boolean update(Stock stock);
}
```

項番	説明
(2)	Repository インタフェースに、Entity を取得するためにメソッドを定義する。
(3)	Repository インタフェースに、楽観ロック機能を使ってデータを更新するメソッドを定義する。 上記例では、指定された Entity の内容でレコードを更新するためのメソッドを定義している。 更新できた場合は、`true` が返却される。

- マッピングファイルに SQL を定義する。

```
<!-- (4) -->
<select id="findOne" parameterType="string" resultType="Stock">
    SELECT
        item_code,
        quantity,
        version
    FROM
        m_stock
    WHERE
        item_code = #{itemCode}
</select>

<!-- (5) -->
<update id="update" parameterType="Stock">
    UPDATE
        m_stock
    SET
        quantity = #{quantity},
        /* (6) */
        version = version + 1
    WHERE
        item_code = #{itemCode}
    AND
        /* (7) */
        version = #{version}
```

```
</update>
```

項番	説明
(4)	Entity を取得するためのステートメント (SQL) を定義する。 楽観ロックを使用する場合は、Entity 取得時にバージョンを取得しておく必要がある。
(5)	楽観ロック機能を使ってデータを更新するためのステートメント (SQL) を定義する。 上記例では、指定された Entity の内容でレコードを更新する SQL を定義している。
(6)	バージョンの更新 (`version = version + 1`) は、SQL の中で行う。
(4)	更新条件として、「バージョンが変わっていない事 (`version = #{version}`)」を加える。

- Repository のメソッドを呼び出し、楽観ロック機能を使用してデータを安全に更新する。

```
// (5)
Stock stock = stockRepository.findOne(itemCode);
if (stock == null) {
    ResultMessages messages = ResultMessages.error().add(ResultMessage
            .fromText("Stock not found. itemCode : " + itemCode));
    throw new ResourceNotFoundException(messages);
}

// (6)
stock.setQuantity(stock.getQuantity() + addedQuantity);

// (7)
boolean updated = stockRepository.update(stock);
if(!updated) {
    // (8)
    throw new ObjectOptimisticLockingFailureException(Stock.class, itemCode);
}
```

項番	説明
(5)	Repository インタフェースの findOne メソッドを呼び出し、Entity を取得する。
(6)	(5) で取得した Entity に対して、更新する値を指定する。 上記例では、仕入れた在庫数を加算している。
(7)	Repository インタフェースの update メソッドを呼び出し、(5) の処理で更新した Entity を永続層 (DB) に反映する。
(8)	更新結果を判定し、更新結果が `false` の場合は、他のトランザクションによって Entity が更新されたことになるので、楽観ロックエラー (`org.springframework.orm.ObjectOptimisticLockingFailureException`) を発生させる。

ロングトランザクションに対して楽観ロックを行う場合は、以下の点に注意すること。

> **警告:** ロングトランザクションに対して楽観ロックを行う場合は、更新時のチェックとは別に、データ取得時にもバージョンのチェックを行うこと。

以下に、実装例を示す。

- データ取得時にもバージョンのチェックを行う。

```
Stock stock = stockRepository.findOne(itemCode);
if (stock == null || stock.getVersion() != version) {
    // (9)
    throw new ObjectOptimisticLockingFailureException(Stock.class, itemCode);
}

stock.setQuantity(stock.getQuantity() + addedQuantity);
boolean updated = stockRepository.update(stock);
// ...
```

項番	説明
(9)	別のデータベーストランザクションで取得した Entity のバージョンと、(5) で取得した Entity のバージョンを比較する。 バージョンが異なる場合は、他のトランザクションによってデータが更新されているので、楽観ロックエラー (`org.springframework.dao.ObjectOptimisticLockingFailureException`) を発生させる。 データが存在しない (`stock == null`) 時の考慮も必要であり、アプリケーションの仕様に対応した実装を行う必要がある。上記例では、楽観ロックエラーとしている。

RDBMS の行ロック機能と楽観ロック機能を併用するアプリケーション場合は、以下の点に注意すること。

> **警告:** RDBMS の行ロック機能を利用して排他制御を行う処理と、楽観ロック機能を利用して排他制御を行う処理が共存するアプリケーションの場合は、RDBMS の行ロック機能を使う SQL の中で、**バージョンの更新 (インクリメント) が必要となる。**
>
> 仮に RDBMS の行ロック機能を使って排他制御を行う SQL の中でバージョンを更新しなかった場合、楽観ロック機能を利用して排他制御を行っている SQL でデータを上書きしてしまう可能性がある。

以下に、実装例を示す。

- SQL 内でバージョンを更新する。

```
<update id="decrementQuantity">
<![CDATA[
    UPDATE
        m_stock
    SET
        quantity = quantity - #{quantity},
        /* (10) */
        version = version + 1
    WHERE
        item_code = #{itemCode}
    AND
        quantity >= #{quantity}
]]>
</update>
```

項番	説明
(10)	バージョンの更新 (インクリメント) を行う。

悲観ロック

MyBatis3 では、ライブラリとして悲観ロックを行う仕組みは提供していない。
そのため、悲観ロックを行う場合は、SQL の中でロックを取得するためのキーワードを指定する必要がある。

- SQL の中でロックを取得するためのキーワードを指定する

```
<select id="findOneForUpdate" parameterType="string" resultType="Stock">
    SELECT
        item_code,
        quantity,
        version
    FROM
        m_stock
    WHERE
        item_code = #{itemCode}
    /* (1) */
    FOR UPDATE
</select>
```

項番	説明
(1)	悲観ロックの取得が必要な SQL に対して、悲観ロックを取得するためのキーワードを指定する。 キーワードやキーワードの指定位置は、データベースによって異なる。

JPA(Spring Data JPA) 使用時の実装方法

RDBMS の行ロック機能

RDBMS の行ロック機能を使って排他制御を行う場合は、Repository インタフェースに Query メソッドを追加して実現する。

Query メソッドについては、*Query メソッドの追加*と、*永続層の Entity を直接操作する*を参照されたい。

- Repository インタフェース

```
public interface StockRepository extends JpaRepository<Stock, String> {

    @Modifying
    @Query("UPDATE Stock s"
            + " SET s.quantity = s.quantity - :quantity"
            + " WHERE s.itemCode = :itemCode"
            + " AND :quantity <= s.quantity")  // (1)
    public int decrementQuantity(@Param("itemCode") String itemCode,
            @Param("quantity") int quantity);

}
```

項番	説明
(1)	Query メソッドに、在庫数が注文数以上ある場合に、在庫数を減らす JPQL を指定する。 更新件数をチェックする必要があるので、Query メソッドの返り値は、`int` を指定する。

- Service

```
String itemCodeOfOrder = "ITM0000001";
int quantityOfOrder = 31;

int updateCount = stockRepository.decrementQuantity(itemCodeOfOrder,␣
↪quantityOfOrder); // (2)
if (updateCount == 0) { // (3)
    ResultMessages message = ResultMessages.error();
    message.add(ResultMessage
            .fromText("Not enough stock. Please, change quantity."));
    throw new BusinessException(message); // (4)
}
```

```
update m_stock set quantity=quantity-31
               where item_code='ITM0000001' and 31<=quantity -- (5)
```

項番	説明
(2)	Query メソッドを呼び出す。
(3)	Query メソッドの呼び出し結果を判定する。」0」 の場合、更新条件を満たしていないので、在庫数が不足していることになる。
(4)	在庫がない、または不足している旨のメッセージを格納し、業務エラーを発生させる。 発生させたエラーは、Controller で要件に応じて適切にハンドリングすること。 上記例では、ビジネスルールのチェックを排他制御しながら行っているだけなので、更新条件を満たさない場合は、排他エラーではなく業務エラーとしている。 エラーのハンドリング方法については、コーディングポイント（*Controller* 編）を参照されたい。
(5)	Query メソッド呼び出し時に実行される SQL。

楽観ロック

JPA では、バージョン管理用のプロパティに、`@javax.persistence.Version` アノテーションを指定することで、楽観ロックを行うことができる。

- Entity

```
@Entity
@Table(name = "m_stock")
public class Stock implements Serializable {

    @Id
    @Column(name = "item_code")
    private String itemCode;

    private int quantity;

    @Version // (1)
    private long version;

    // ...

}
```

項番	説明
(1)	バージョン管理用のプロパティに、@Version アノテーションを指定する。

- Service

```
String itemCode = "ITM0000001";
int newQuantity = 30;

Stock stock = stockRepository.findOne(itemCode); // (2)
if (stock == null) {
    ResultMessages messages = ResultMessages.error().add(ResultMessage
            .fromText("Stock not found. itemCode : " + itemCode));
    throw new ResourceNotFoundException(messages);
}

stock.setQuantity(newQuantity); // (3)

stockRepository.flush(); // (4)
```

```
update m_stock set quantity=30, version=7
               where item_code='ITM0000001' and version=6 -- ( 5)
```

項番	説明
(2)	Repository インタフェースの findOne メソッドを呼び出し、Entity を取得する。
(3)	(2) で取得した Entity に対して、更新する値を指定する。
(4)	(3) の変更内容を永続層 (DB) に反映する。この処理は説明のために行っている処理のため、通常は不要である。 通常は、トランザクションコミット時に自動で反映される。 上記例だと、(2) で取得した Entity がもつバージョンと永続層 (DB) で保持しているバージョンが一致しない場合に、楽観ロックエラー (org.springframework.dao.OptimisticLockingFailureException) が発生する。
(5)	(4) の永続層 (DB) に反映する際に実行される SQL。

ロングトランザクションに対する楽観ロックを行う場合は、以下の点に注意すること。

> **警告:** ロングトランザクションに対する楽観ロックについては、`@Version` アノテーションを付与するだけでは不十分である。ロングトランザクションに対して楽観ロックを行う場合は、JPA の機能で行われる更新時のチェックに加えて、更新対象のデータを取得する際にも、バージョンのチェックを行うこと。

以下に、実装例を示す。

- Service

```
long version = 12;
String itemCode = "ITM0000001";
int newQuantity = 30;

Stock stock = stockRepository.findOne(itemCode); // (1)
if (stock == null || stock.getVersion() != version) { // (2)
    throw new ObjectOptimisticLockingFailureException(Stock.class, itemCode);
↪// (3)
}

stock.setQuantity(newQuantity);

stockRepository.flush();
```

項番	説明
(1)	永続層 (DB) から Entity を取得する。
(2)	事前に別のデータベーストランザクションで取得された Entity のバージョンと、(1) で取得した永続層 (DB) の最新のバージョンを比較する。 バージョンが一致する場合は、以降の処理で`@Version` アノテーションを使った楽観ロックの仕組みが有効となる。
(3)	バージョンが異なる場合は、楽観ロックエラー (`org.springframework.dao.ObjectOptimisticLockingFailureException`) を発生させる。

> **警告: Version 管理用のプロパティに対する値の設定について**
>
> Repository インタフェースを使って取得した Entity は、「管理状態の Entity」と呼ばれる。

「管理状態の **Entity**」に対して、処理で **Version** 管理用のプロパティの値を設定することはできないので、注意すること。

以下のような処理をしても、「管理状態の Entity」に設定したバージョンの値は反映されないため、楽観ロックを取得する際に使用されることはない。楽観ロックで使用されるのは、findOne メソッドで取得した時点のバージョンとなる。

```
long version = 12;
String itemCode = "ITM0000001";
int newQuantity = 30;

Stock stock = stockRepository.findOne(itemCode);
if (stock == null) {
    ResultMessages messages = ResultMessages.error().add(ResultMessage
            .fromText("Stock not found. itemCode : " + itemCode));
    throw new ResourceNotFoundException(messages);
}
stock.setVersion(version); // ★ Invalid Processing
stock.setQuantity(newQuantity);

stockRepository.flush();
```

例えば、画面から送られてきたバージョンの値を上書きしても、Entity には反映されないため、排他制御が正しく行われなくなってしまう。

注釈: ロングトランザクションに対する楽観ロック処理の共通化について

複数の処理でロングトランザクションに対して楽観ロックが必要になる場合は、上記の (1)〜(3) の処理を共通的なメソッドにすることを検討した方がよい。共通化の方法については、カスタムメソッドの追加方法を参照されたい。

RDBMS の行ロック機能と、楽観ロック機能を両方使用する場合は、以下の点に注意すること。

警告: 同じデータに対して、RDBMS の行ロック機能を利用して排他制御を行う処理と、楽観ロック機能を利用して排他制御を行う処理が共存するアプリケーションの場合は、RDBMS の行ロック機能を使う Query メソッドにて、**Version** の更新を必ず行う必要がある。

RDBMS の行ロック機能を使って、排他制御を行う Query メソッドで Version を更新しない場合、Query メソッドで更新した内容が、別のトランザクションの処理で上書きされる可能性があるため、正しく排他制御が行われない。

以下に、実装例を示す。

- Repository インタフェース

```
public interface StockRepository extends JpaRepository<Stock, String> {

    @Modifying
    @Query("UPDATE Stock s SET s.quantity = s.quantity - :quantity"
            + ", s.version = s.version + 1" // (1)
            + " WHERE s.itemCode = :itemCode"
            + " AND :quantity <= s.quantity")
    public int decrementQuantity(@Param("itemCode") String itemCode,
            @Param("quantity") int quantity);

}
```

項番	説明
(1)	Version の更新 (`s.version = s.version + 1`) を行う必要がある。

悲観ロック

Spring Data JPA では、`@org.springframework.data.jpa.repository.Lock` アノテーションを指定することで、悲観ロックを行うことができる。

- Repository インタフェース

```
public interface StockRepository extends JpaRepository<Stock, String> {

    @Lock(LockModeType.PESSIMISTIC_WRITE) // (1)
    @Query("SELECT s FROM Stock s WHERE s.itemCode = :itemCode")
    Stock findOneForUpdate(@Param("itemCode") String itemCode);

}
```

```
-- (2)
SELECT
        stock0_.item_code AS item1_5_
        ,stock0_.quantity AS quantity2_5_
        ,stock0_.version AS version3_5_
    FROM
        m_stock stock0_
    WHERE
        stock0_.item_code = 'ITM0000001'
    FOR UPDATE;
```

項番	説明
(1)	Query メソッドに、`@Lock` アノテーションを指定する。
(2)	実行される SQL。上記例では PostgreSQL を使用した場合に実行される SQL となる。

`@Lock` アノテーションで指定することができる悲観ロックの種類は、以下の通りである。

項番	LockModeType	説明	発行される SQL
1.	PESSIMISTIC_READ	参照用の悲観ロックが取得される。データベースによっては、排他ロックではなく共有ロックとなる。 コミットまたはロールバック時に、ロック解放される	select … for update / select … for share
2.	PESSIMISTIC_WRITE	更新用の悲観ロックが取得され、排他ロックがかかる。 排他ロックの場合、既にロックがかかっていた場合には、ロックが解放されるまで待機してからエンティティが取得される。 コミットまたはロールバック時に、ロック解放される	select … for update
3.	PESSIMISTIC_FORCE_INCREMENT	エンティティを取得した時点から、対象データに対して排他ロックがかかる。取得直後に強制的にバージョンの更新も行われる。 コミットまたはロールバック時に、ロック解放される	select … for update + update

注釈: ロックタイムアウト時間について

JPA(`EntityManager`) の設定または Query ヒントとして、`javax.persistence.lock.`

timeout を指定することで、タイムアウト時間を指定することができる。

ロックのタイムアウト時間の指定は、全体に適用する方法と、Query 毎に適用する 2 つの方法が用意されている。

全体に適用する方法は、以下の通りである。

- xxx-infra.xml

```
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean
↪">
    <property name="packagesToScan" value="xxxxxx.yyyyyy.zzzzzz.domain.model"␣
↪/>
    <property name="dataSource" ref="dataSource" />
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
    <property name="jpaPropertyMap">
        <util:map>
            <!-- ... -->
            <entry key="javax.persistence.lock.timeout" value="1000" /> <!--␣
↪(1) -->
        </util:map>
    </property>
</bean>
```

項番	説明
(1)	タイムアウトをミリ秒で指定する。1000 を指定すると、1 秒となる。

注釈: **nowait** のサポート

Oracle と PostgreSQL については、」0」 を指定した場合、nowait が付加され、他のトランザクションによってロックされていた場合に、ロックの解放待ちを行わずに排他エラーとなる。

> 警告: **PostgreSQL** 使用時に」**nowait**」句が付加されない不具合について
>
> TERASOLUNA Server Framework for Java 5.4.2.RELEASE 版の依存ライブラリである Hibernate 5.0.X 系の不具合 (HHH-10797) のため、PostgreSQL を使用した場合、タイムアウト時間を 「0」 に設定しても SQL に」nowait」句が付加されない。
>
> そのため、以下の様な対策を検討する必要がある。
>
> - Hibernate 提供の Dialect クラスにパッチを適用して使う。
> - Hibernate 5.2.1 以上へバージョンアップする。

警告：PostgreSQL の制約

PostgreSQL では nowait の指定はできるが、wait 時間の指定ができない。そのため、Query のタイムアウトを別途設けておくなどの対策を行う必要がある。

Query 毎に適応する方法は、以下の通りである。

- Repository インタフェース

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
@QueryHints(@QueryHint(name = "javax.persistence.lock.timeout", value = "2000
↪")) // (1)
@Query("SELECT s FROM Stock s WHERE s.itemCode = :itemCode")
Stock findOneForUpdate(@Param("itemCode") String itemCode);
```

項番	説明
(1)	タイムアウトをミリ秒で指定する。2000 を指定すると、2 秒となる。 全体に指定した値は、上書きされる。

排他エラーのハンドリング方法

楽観ロックの失敗時のエラーハンドリング

楽観ロックの失敗時には、org.springframework.dao.OptimisticLockingFailureException が発生するため、Controller で適切にハンドリングする必要がある。

ハンドリング方法は、楽観ロックエラーが発生した時のアプリケーションの動作仕様によって異なる。

リクエスト単位に動作を変える必要がない場合は、@ExceptionHandler アノテーションを使用してハンドリングする。

```
@ExceptionHandler(OptimisticLockingFailureException.class) // (1)
public ModelAndView handleOptimisticLockingFailureException(
        OptimisticLockingFailureException e) {
    // (2)
```

```
    ExtendedModelMap modelMap = new ExtendedModelMap();
    ResultMessages resultMessages = ResultMessages.warning();
    resultMessages.add(ResultMessage.fromText("Other user updated!!"));
    modelMap.addAttribute(setUpForm());
    modelMap.addAttribute(resultMessages);
    String viewName = top(modelMap);
    return new ModelAndView(viewName, modelMap);
}
```

項番	説明
(1)	@ExceptionHandler アノテーションの value 属性に、OptimisticLockingFailureException.class を指定する。
(2)	エラーハンドリングの処理を実装する。エラーを通知するためのメッセージ、画面表示に必要な情報（フォームやその他のモデル）を生成し、遷移先を指定した ModelAndView を返却する。 エラーハンドリングの詳細については、ユースケース単位で例外をハンドリングする方法を参照されたい。

リクエスト単位に動作を変える必要がある場合は、Controller のハンドラメソッドの中で、try - catch を使用してハンドリングする。

```
@RequestMapping(value = "{itemId}/update", method = RequestMethod.POST)
public String update(StockForm form, Model model, RedirectAttributes␣
↪attributes){

    // ...

    try {
        stockService.update(...);
    } catch (OptimisticLockingFailureException e) { // (1)
        // (2)
        ResultMessages resultMessages = ResultMessages.warn();
        resultMessages.add(ResultMessage.fromText("Other user updated!!"));
        model.addAttribute(resultMessages);
        return updateRedo(modelMap);
    }

    // ...

}
```

項番	説明
(1)	OptimisticLockingFailureException を catch する。
(2)	エラーハンドリングの処理を実装する。エラーを通知するためのメッセージ、画面表示に必要な情報（フォームやその他のモデル）を生成し、遷移先の view 名を返却する。 エラーハンドリングの詳細については、リクエスト単位で例外をハンドリングする方法を参照されたい。

悲観ロックの失敗時のエラーハンドリング

悲観ロックの失敗時には、org.springframework.dao.PessimisticLockingFailureException が発生するため、 Controller で適切にハンドリングする必要がある。

ハンドリング方法は、悲観ロックエラーが発生した時のアプリケーションの動作仕様によって異なる。

リクエスト単位に動作を変える必要がない場合は、@ExceptionHandler アノテーションを使用してハンドリングする。

```
@ExceptionHandler(PessimisticLockingFailureException.class) // (1)
public ModelAndView handlePessimisticLockingFailureException(
        PessimisticLockingFailureException e) {
    // (2)
    ExtendedModelMap modelMap = new ExtendedModelMap();
    ResultMessages resultMessages = ResultMessages.warning();
    resultMessages.add(ResultMessage.fromText("Other user updated!!"));
    modelMap.addAttribute(setUpForm());
    modelMap.addAttribute(resultMessages);
    String viewName = top(modelMap);
    return new ModelAndView(viewName, modelMap);
}
```

項番	説明
(1)	`@ExceptionHandler` アノテーションの value 属性に、`PessimisticLockingFailureException.class` を指定する。
(2)	エラーハンドリングの処理を実装する。エラーを通知するためのメッセージ、画面表示に必要な情報（フォームやその他のモデル）を生成し、遷移先を指定した `ModelAndView` を返却する。 エラーハンドリングの詳細については、ユースケース単位で例外をハンドリングする方法を参照されたい。

リクエスト単位に動作を変える必要がある場合は、Controller のハンドラメソッドの中で、`try - catch` を使用してハンドリングする。

```
@RequestMapping(value = "{itemId}/update", method = RequestMethod.POST)
public String update(StockForm form, Model model, RedirectAttributes␣
↪attributes){

    // ...

    try {
        stockService.update(...);
    } catch (PessimisticLockingFailureException e) { // (1)
        // (2)
        ResultMessages resultMessages = ResultMessages.warn();
        resultMessages.add(ResultMessage.fromText("Other user updated!!"));
        model.addAttribute(resultMessages);
        return updateRedo(modelMap);
    }

    // ...

}
```

項番	説明
(1)	`PessimisticLockingFailureException` を catch する。
(2)	エラーハンドリングの処理を実装する。エラーを通知するためのメッセージ、画面表示に必要な情報（フォームやその他のモデル）を生成し、遷移先の view 名を返却する。 エラーハンドリングの詳細については、リクエスト単位で例外をハンドリングする方法を参照されたい。

第 7 章

アプリケーション形態に依存しない汎用機能

7.1 ロギング

注釈: 本ガイドラインで説明する内容はあくまで指針のため、業務要件に合わせて柔軟に対応すること。

7.1.1 Overview

システムを運用する上、業務使用量の調査、システムダウンや、
業務エラー等でその原因を究明するための情報源として、ログおよびメッセージを出力する。

デバッグ時にログ出力を取り入れることで、解析の作業効率が格段に向上するため、ログを出力しておくことは重要である。

動きを確認するだけであれば、IDE のデバッグ実行や、`System.out.println` のような簡易的な出力で行える。
しかし、出力結果を手動で保存しておかないと、後に結果の確認ができず、解析の作業効率が格段に下がる。
ロギングライブラリを導入してログをとることは、出力するコードを書くのみで、
その後、好きなタイミングでログを確認することができる。
作業の時間、証跡、解析を考えると、ロギングライブラリを導入することを推奨する。

Java では、ログ出力の方法は複数あり、多くの方法が選べるが、コーディングの簡易性、変更の容易性、性能を判断して、
本ガイドラインでは、ロギングライブラリに、SLF4J (インタフェース) + Logback (実装) を推奨している。

ログの種類

アプリケーション開発時における代表的なログを、以下に示す。

ログレベル	カテゴリ	出力目的	出力内容
TRACE	性能ログ	リクエストの処理時間の測定 (本番環境運用時は出力対象としない)	処理開始終了時間、処理経過時間 (ms)、 実行処理を判別できる情報 (実行コントローラ + メソッド、リクエスト URL など) 等
DEBUG	デバッグログ	開発時のデバッグ (本番環境運用時には出力対象としない)	任意 (実行したクエリ、入力パラメータ、戻り値など)
INFO	アクセスログ	業務量の把握	アクセス日時、利用ユーザを判別できる情報 (IP アドレス、認証情報)、 実行処理を判別できる情報 (リクエスト URL) 等、証跡を残すための情報
INFO	外部通信ログ	外部システムとの通信におけるエラー解析	送信受信時間、送受信データなど
WARN	業務エラーログ	業務エラーの記録	業務エラー発生時間、業務エラーに対応するメッセージ ID とメッセージ 入力情報、例外メッセージなど解析に必要な情報
ERROR	システムエラーログ	システム運用の継続が困難な事象の記録	システムエラー発生時間、システムエラーに対応するメッセージ ID とメッセージ 入力情報、例外メッセージなど解析に必要な情報 基本的には、フレームワークが出力し、ビジネスロジックは出力しない。

Continued on next page

表 7.1 – continued from previous page

ログレベル	カテゴリ	出力目的	出力内容
ERROR	監視ログ	例外発生の監視	例外発生時間、システムエラーに対応するメッセージ ID ツールを用いて監視することを考慮し、出力内容は最低限とすること

デバッグログ、アクセスログ、外部通信ログ、業務エラーログ、システムエラーログは、同一のファイルに出力する。
本ガイドラインでは、上記を出力するログファイルを、アプリケーションログと呼ぶこととする。

注釈: SLF4J や Logback のログレベルの順番は、 TRACE < DEBUG < INFO < WARN < ERROR である。commons-loggins や、Log4J で用意されていた FATAL レベルは、存在しない。

ログの出力内容

ログの出力内容として考慮すべき点を、以下に示す。

1. ログに出力する ID について

   ログを運用で監視する場合は、運用監視で使用するログに、メッセージ ID を含めることを推奨する。

   また、アクセスログを用いて業務量を把握する場合は、集計を容易にするため、メッセージ管理で示しているように、業務ごとに切り分けられる ID をあわせて出力すること。

   注釈: ログに ID を含めることにより、ログの可読性が高まるため、システム運用時は、故障解析の一次切り分けの短時間化につながる。ログ ID の体系は、メッセージ管理を参考にすると良い。ただし、すべてのログに ID を付与する必要はなく、debug 時には、ID は不要である。運用時に、素早く切り分け可能になることを推奨する。

   障害発生時に、ログ ID(またはメッセージ ID) を、エラー画面に表示して、システム利用者に通知し、利用者に対して、その ID をコールセンターに通知してもらうような運用にすると、障害解析が容易になる。

   ただし、障害の内容までエラーが画面に表示してしまうと、システムの脆弱性を晒してしまう可能性があるため、注意すること。

   例外が発生した際に、ログや画面にメッセージ ID(例外コード) を含めるための仕組み (コンポーネント) を共通ライブラリから提供している。詳細については、「例外ハンドリング」を参照されたい。

   ---

2. トレーサビリティ

   トレーサビリティ向上のために、各ログにリクエスト単位で、一意となるような Track ID(以降 X-Track と呼ぶ) を出力させることを推奨する。

   X-Track を含めたログの例を、以下に示す。

```
date:2013-09-06 19:36:31    X-Track:85a437108e9f4a959fd227f07f72ca20
↪message:[START CONTROLLER] (omitted)
date:2013-09-06 19:36:31    X-Track:85a437108e9f4a959fd227f07f72ca20
↪message:[END CONTROLLER  ] (omitted)
date:2013-09-06 19:36:31    X-Track:85a437108e9f4a959fd227f07f72ca20
↪message:[HANDLING TIME   ] (omitted)
date:2013-09-06 19:36:33    X-Track:948c8b9fd04944b78ad8aa9e24d9f263
↪message:[START CONTROLLER] (omitted)
date:2013-09-06 19:36:33    X-Track:142ff9674efd486cbd1e293e5aa53a78
↪message:[START CONTROLLER] (omitted)
date:2013-09-06 19:36:33    X-Track:142ff9674efd486cbd1e293e5aa53a78
↪message:[END CONTROLLER  ] (omitted)
date:2013-09-06 19:36:33    X-Track:142ff9674efd486cbd1e293e5aa53a78
↪message:[HANDLING TIME   ] (omitted)
date:2013-09-06 19:36:33    X-Track:948c8b9fd04944b78ad8aa9e24d9f263
↪message:[END CONTROLLER  ] (omitted)
date:2013-09-06 19:36:33    X-Track:948c8b9fd04944b78ad8aa9e24d9f263
↪message:[HANDLING TIME   ] (omitted)
```

   Track ID を出力させることで、不規則に出力された場合でも、ログを結びつけることができる。

   上記の例だと、4 行目と 8,9 行目が、同じリクエストに関するログであることがわかる。

   共通ライブラリでは、リクエスト毎のユニークキーを生成し、MDC に追加する `org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter` を提供している。

   `XTrackMDCPutFilter` は、HTTP レスポンスヘッダの」X-Track」にも Track ID を設定する。ログ中では、Track ID のラベルとして、X-Track を使用している。

   使用方法については、*MDC について*を参照されたい。

3. ログのマスクについて

   個人情報、クレジットカード番号など、

   ログファイルにそのまま出力すると、セキュリティ上問題のある情報は、必要に応じてマスクすること。

ログの出力ポイント

カテゴリ	出力ポイント
性能ログ	業務処理の処理時間を計測し、業務処理実行後に出力したり、リクエストの処理時間を計測し、レスポンスを返す際に、ログを出力する。 通常は、AOP やサーブレットフィルタ等で実装する。 共通ライブラリでは、Spring MVC の Controller のハンドラメソッドの処理時間を、Controller のハンドラメソッド実行後に、TRACE ログで出力する、`org.terasoluna.gfw.web.logging.TraceLoggingInterceptor` を提供している。
デバッグログ	開発時にデバッグ情報を出力する必要がある場合、ソースコード中に、適宜ログ出力処理を実装する。 共通ライブラリでは、HTTP セッションの生成・破棄・属性追加のタイミングで、DEBUG ログを出力するリスナー `org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener` を提供している。
アクセスログ	リクエストの受付時、レスポンス返却時に、INFO ログを出力する。 通常は、AOP やサーブレットフィルタで実装する。
外部通信ログ	外部のシステムと連携前後で、INFO ログを出力する。
業務エラーログ	業務例外がスローされたタイミング等で、WARN ログを出力する。 通常は、AOP で実装する。 共通ライブラリでは、業務処理実行時に *org.terasoluna.gfw.common.exception.BusinessException* がスローされた場合に、WARN ログを出力する `org.terasoluna.gfw.common.exception.ResultMessagesLoggingInterceptor` を提供している。 詳細は 例外ハンドリング を参照。

Continued on next page

表 7.2 – continued from previous page

カテゴリ	出力ポイント
システムエラーログ	システム例外や、予期せぬ例外が発生した際に、ERROR ログを出力する。 通常は、AOP やサーブレットフィルタ等で実装する。 共通ライブラリでは、`org.terasoluna.gfw.web.exception.HandlerExceptionResolverLoggingInterceptor` や、`org.terasoluna.gfw.web.exception.ExceptionLoggingFilter` を提供している。 詳細は、例外ハンドリング を参照されたい。
監視ログ	業務エラーログ、システムエラーログの出力タイミングと同様である。

注釈: ログを出力する際は、どこで出力されたかわかりやすくなるように、他のログと、全く同じ内容を出力にならないように注意すること。

7.1.2 How to use

SLF4J + Logback でログを出力するには、

1. Logback の設定
2. SLF4J の API 呼び出し

が必要である。

Logback の設定

Logback の設定は、クラスパス直下の logback.xml に記述する。以下に、設定例を示す。

logback.xml の詳細な設定方法については、Logback の公式マニュアル -Logback configuration-を参照されたい。

注釈: Logback の設定は、以下のルールによる自動で読み込まれる。

1. クラスパス上の logback.groovy
2.「1」のファイルが見つからない場合、クラスパス上の logback-test.xml
3.「2」のファイルが見つからない場合、クラスパス上の logback.xml
4.「3」のファイルが見つからない場合、`com.qos.logback.classic.spi.Configurator` インタフェースの実装クラスの設定内容 (ServiceLoader の仕組みを使用して実装クラスを指定)
5. `Configurator` インタフェースの実装クラスが見つからない場合、BasicConfigurator クラスの設定内容 (コンソール出力)

本ガイドラインでは、logback.xml をクラスパス上に配置することを推奨する。このほか、自動読み込み以外にも、API によってプログラマティックに読み込んだり、システムプロパティで設定ファイルを指定することができる。

---

logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
```

```
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender"> <!-- (1) -
↪->
        <encoder>
            <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tthread:%thread\tX-
↪Track:%X{X-Track}\tlevel:%-5level\tlogger:%-48logger{48}\tmessage:%msg%n]]></
↪pattern> <!-- (2) -->
        </encoder>
    </appender>

    <appender name="APPLICATION_LOG_FILE" class="ch.qos.logback.core.rolling.
↪RollingFileAppender"> <!-- (3) -->
        <file>${app.log.dir:-log}/projectName-application.log</file> <!-- (4) -->
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${app.log.dir:-log}/projectName-application-%d
↪{yyyyMMddHH}.log</fileNamePattern> <!-- (5) -->
            <maxHistory>7</maxHistory> <!-- (6) -->
        </rollingPolicy>
        <encoder>
            <charset>UTF-8</charset> <!-- (7) -->
            <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tthread:%thread\tX-
↪Track:%X{X-Track}\tlevel:%-5level\tlogger:%-48logger{48}\tmessage:%msg%n]]></
↪pattern>
        </encoder>
    </appender>

    <appender name="MONITORING_LOG_FILE" class="ch.qos.logback.core.rolling.
↪RollingFileAppender"> <!-- (8) -->
        <file>${app.log.dir:-log}/projectName-monitoring.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${app.log.dir:-log}/projectName-monitoring-%d
↪{yyyyMMdd}.log</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder>
            <charset>UTF-8</charset>
            <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tX-Track:%X{X-Track}
↪\tlevel:%-5level\tmessage:%msg%n]]></pattern>
        </encoder>
    </appender>

    <!-- Application Loggers -->
    <logger name="com.example.sample"> <!-- (9) -->
        <level value="debug" />
    </logger>

    <!-- TERASOLUNA -->
    <logger name="org.terasoluna.gfw">
        <level value="info" />
    </logger>
    <logger name="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor">
        <level value="trace" />
    </logger>
```

```
    <logger name="org.terasoluna.gfw.common.exception.ExceptionLogger">
        <level value="info" />
    </logger>
    <logger name="org.terasoluna.gfw.common.exception.ExceptionLogger.Monitoring"␣
↪additivity="false"><!-- (10) -->
        <level value="error" />
        <appender-ref ref="MONITORING_LOG_FILE" />
    </logger>

    <!-- 3rdparty Loggers -->
    <logger name="org.springframework">
        <level value="warn" />
    </logger>

    <logger name="org.springframework.web.servlet">
        <level value="info" />
    </logger>

    <!--  REMOVE THIS LINE IF YOU USE JPA
    <logger name="org.hibernate.engine.transaction">
        <level value="debug" />
    </logger>
          REMOVE THIS LINE IF YOU USE JPA  -->
    <!--  REMOVE THIS LINE IF YOU USE MyBatis3
    <logger name="org.springframework.jdbc.datasource.DataSourceTransactionManager
↪">
        <level value="debug" />
    </logger>
          REMOVE THIS LINE IF YOU USE MyBatis3  -->

    <logger name="jdbc.sqltiming">
        <level value="debug" />
    </logger>

    <!-- only for development -->
    <logger name="jdbc.resultsettable">
        <level value="debug" />
    </logger>

    <root level="warn"> <!-- (11) -->
        <appender-ref ref="STDOUT" /> <!-- (12) -->
        <appender-ref ref="APPLICATION_LOG_FILE" />
    </root>

</configuration>
```

項番	説明
(1)	コンソールにログを出力するための、アペンダ定義を指定する。 出力先を標準出力にするか、標準エラーにするか選べるが、指定しない場合は、標準出力となる。
(2)	ログの出力形式を指定する。何も記述しなければ、メッセージだけが出力される。 時刻やメッセージレベルなど、業務要件に合わせて出力させる。 ここでは」ラベル:値<TAB>ラベル:値<TAB>…」形式の LTSV(Labeled Tab Separated Value) フォーマットを設定している。
(3)	アプリケーションログを出力するための、アペンダ定義を指定する。 どのアペンダを使用するかは、<logger>に指定することもできるが、ここではアプリケーションログはデフォルトで使用するため、root（11）に参照させている。 アプリケーションログを出力する際によく使用されるのは、RollingFileAppender であるが、ログのローテーションを logrotate など別機能で実施する場合、FileAppender を使用することもある。
(4)	カレントファイル名 (出力中のログのファイル名) を指定する。固定のファイル名としたい場合は指定すること。 <file>ログファイル名</file>を指定しないと、(5) のパターンの名称で出力される。
(5)	ローテーション後のファイル名を指定する。通常は、日付か時間の形式が、多く採用される。 誤って HH を hh と設定してしまうと、24 時間表記されないため注意すること。
(6)	ローテーションしたファイルをいくつ残すかを指定する。
(7)	ログファイルの文字コードを指定する。
(8)	デフォルトでアプリケーションログが出力されるように設定する。
(9)	ロガー名は、com.example.sample 以下のロガーが、debug レベル以上のログを出力するように設定する。

Continued on next page

表 7.3 – continued from previous page

項番	説明
(10)	監視ログの設定を行う。例外ハンドリングの共通設定を参照されたい。 **警告: additivity の設定値について** `false` を指定すること。`true`(デフォルト値) を指定すると、上位のロガー (例えば、root) によって、同じログが出力されてしまう。具体的には、監視ログは 3 つのアペンダー (`MONITORING_LOG_FILE`、`STDOUT`、`APPLICATION_LOG_FILE`) によって出力される。
(11)	<logger>の指定が無いロガーが、warn レベル以上のログを出力するように設定する。
(12)	デフォルトで ConsoleAppender, RollingFileAppender(アプリケーションログ) が使用されるように設定する。

ちなみに：**LTSV(Labeled Tab Separated Value)** について

LTSV は、テキストデータのフォーマットの一つであり、主にログのフォーマットとして使用される。

LTSV は、

- フィールドの区切り文字としてタブを使用することで、他の区切り文字に比べてフィールドを分割しやすい。
- フィールドにラベル (名前) を設けることで、フィールド定義の変更 (定義位置の変更、フィールドの追加、フィールドの削除) を行ってもパース処理には影響を与えない。

また、エクセルに貼り付けるだけで最低限のフォーマットが行える点も特徴の一つである。

logback.xml で設定するものは、次の 3 つになる。

種類	概要
appender	「どの場所に」「どんなレイアウト」で出力するのか
root	デフォルトでは、「どのログレベル」以上で「どの appender」に出力するのか
logger	「どのロガー (パッケージやクラス等)」は、「どのログレベル」以上で出力するのか

<appender>要素には、「どの場所に」「どんなレイアウト」で出力するのかを定義する。appender を定義しただけではログ出力の際に使用されず、<logger>要素や<root>要素に参照されると、初めて使用される。属性は、name と class の 2 つで、共に必須である。

属性	概要
name	appender の名前。appender-ref で指定される。好きな名前をつけてよい。
class	appender 実装クラスの FQCN。

提供されている主な appender を、以下に示す

Appender	概要
ConsoleAppender	コンソール出力
FileAppender	ファイル出力
RollingFileAppender	ファイル出力 (ローリング可能)
AsyncAppender	非同期出力。性能を求められる処理中のロギングに使用する。(出力先は、他の Appender で設定する必要がある。)

Appender の詳細な種類は、Logback の公式マニュアル -Appenders-を参照されたい。

SLF4J の API 呼び出しによる基本的なログ出力

SLF4J のロガー (`org.slf4j.Logger`) の各ログレベルに応じたメソッドを呼び出してログを出力する。

```
package com.example.sample.app.welcome;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class HomeController {

    private static final Logger logger = LoggerFactory
            .getLogger(HomeController.class);   // (1)

    @RequestMapping(value = "/", method = { RequestMethod.GET,
            RequestMethod.POST })
    public String home(Model model) {
        logger.trace("This log is trace log."); // (2)
        logger.debug("This log is debug log."); // (3)
        logger.info("This log is info log.");   // (4)
        logger.warn("This log is warn log.");   // (5)
        logger.error("This log is error log."); // (6)
        return "welcome/home";
    }

}
```

項番	説明
(1)	`org.slf4j.LoggerFactory` から `Logger` を生成する。`getLogger` の引数に Class オブジェクトを 設定した場合は、ロガー名は、そのクラスの FQCN になる。 この例では、」com.example.sample.app.welcome.HomeController」が、ロガー名になる。
(2)	TRACE レベルのログを出力する。
(3)	DEBUG レベルのログを出力する。
(4)	INFO レベルのログを出力する。
(5)	WARN レベルのログを出力する。
(6)	ERROR レベルのログを出力する。

ログの出力結果を、以下に示す。この com.example.sample のログレベルは、DEBUG なので、TRACE ログは出力されない。

```
date:2013-11-06 20:13:05    thread:tomcat-http--3 X-
↪Track:5844f073b7434b67a875cb85b131e686    level:DEBUG logger:com.example.sample.
↪app.welcome.HomeController    message:This log is debug log.
date:2013-11-06 20:13:05    thread:tomcat-http--3 X-
↪Track:5844f073b7434b67a875cb85b131e686    level:INFO  logger:com.example.sample.
↪app.welcome.HomeController    message:This log is info log.
date:2013-11-06 20:13:05    thread:tomcat-http--3 X-
↪Track:5844f073b7434b67a875cb85b131e686    level:WARN  logger:com.example.sample.
↪app.welcome.HomeController    message:This log is warn log.
date:2013-11-06 20:13:05    thread:tomcat-http--3 X-
↪Track:5844f073b7434b67a875cb85b131e686    level:ERROR logger:com.example.sample.
↪app.welcome.HomeController    message:This log is error log.
```

ログメッセージのプレースホルダに引数を埋め込む場合は、次のように記述すればよい。

```
int a = 1;
logger.debug("a={}", a);
String b = "bbb";
logger.debug("a={}, b={}", a, b);
```

以下のようなログが出力される。

```
date:2013-11-06 20:32:45    thread:tomcat-http--3   X-
↪Track:853aa701a401404a87342a574c69efbc    level:DEBUG logger:com.example.sample.
↪app.welcome.HomeController    message:a=1
date:2013-11-06 20:32:45    thread:tomcat-http--3   X-
↪Track:853aa701a401404a87342a574c69efbc    level:DEBUG logger:com.example.sample.
↪app.welcome.HomeController    message:a=1, b=bbb
```

警告: `logger.debug("a=" + a + " , b=" + b);` というように、文字列連結を行わないように注意すること。

例外をキャッチする際は、以下のように ERROR ログ (場合によっては WARN ログ) を出力し、ログメソッドにエラーメッセージと発生した例外を渡す。

```
public String home(Model model) {
    // omitted

    try {
        throwException();
    } catch (Exception e) {
        logger.error("Exception happend!", e);
        // omitted
    }
    // omitted
}

public void throwException() throws Exception {
    throw new Exception("Test Exception!");
}
```

これにより、起因例外のスタックトレースが出力され、エラーの原因を解析しやすくなる。

```
date:2013-11-06 20:38:04    thread:tomcat-http--5   X-
↪Track:11d7dbdf64e44782822c5aea4fc4bb4f    level:ERROR logger:com.example.sample.
↪app.welcome.HomeController    message:Exception happend!
java.lang.Exception: Test Exception!
    at com.example.sample.app.welcome.HomeController.throwException(HomeController.
↪java:40) ~[HomeController.class:na]
    at com.example.sample.app.welcome.HomeController.home(HomeController.java:31) ~
↪[HomeController.class:na]
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.7.0_40]
    (omitted)
```

ただし、以下のようにキャッチした例外を別の例外にラップして、上位に再スローする場合はログを出力しなくてもよい。通常は上位でエラーログが出力されるためである。

```
try {
    throwException();
} catch (Exception e) {
    throw new SystemException("e.ex.fw.9001", e);
    // no need to log
}
```

注釈: 起因例外をログメソッドに渡す場合は、プレースホルダーを使用できない。この場合に限り、メッセージの引数を文字列で連結してもよい。

```
try {
    throwException();
} catch (Exception e) {
    // NG => logger.error("Exception happend! [a={} , b={}]", e, a, b);
    logger.error("Exception happend! [a=" + a + " , b=" + b + "]", e);
    // omitted
}
```

ログ出力の記述の注意点

SLF4J の Logger は、内部でログレベルのチェックを行い、必要なレベルの場合にのみ実際にログを出力する。

したがって、次のようなログレベルのチェックは、基本的に不要である。

```
if (logger.isDebugEnabled()) {
    logger.debug("This log is Debug.");
}

if (logger.isDebugEnabled()) {
    logger.debug("a={}", a);
}
```

ただし、次の場合は性能劣化を防ぐために、ログレベルのチェックを行うこと。

1. 引数が 3 個以上の場合

   ログメッセージの引数が 3 以上の場合、SLF4J の API では引数の配列を渡す必要がある。配列生成のコストを避けるため、ログレベルのチェックを行い、必要なときのみ、配列が生成されるようにすること。

   ```
   if (logger.isDebugEnabled()) {
       logger.debug("a={}, b={}, c={}", new Object[] { a, b, c });
   }
   ```

2. 引数の生成にメソッド呼び出しが必要な場合

ログメッセージの引数を生成する際にメソッド呼び出しが必要な場合、メソッド実行コストを避けるため、ログレベルのチェックを行い、必要なときのみメソッドが実行されるようにすること。

```
if (logger.isDebugEnabled()) {
    logger.debug("xxx={}", foo.getXxx());
}
```

7.1.3 How to extend

ログ出力仕様は監視製品や要件等で独自の規定があるケースが多く、個別に実装するケースが想定される。ここでは、以下の 2 例を説明する。

1. ログメッセージの一元管理
2. ログメッセージの出力フォーマットの統一

ログメッセージの一元管理

ログメッセージの一元管理によるメンテナンス性向上等を目的とした実装例を紹介する。

ログメッセージの一元管理は、ログメッセージをプロパティファイル等の別ファイルにまとめ、ログ出力時にメッセージ解決を行うことで実現できる。

ここでは実装例として、ログ出力メソッドの引数にログ ID を設定できるようにし、プロパティファイルの中のログ ID に対応するメッセージを出力する方法を説明する。

注釈: ログ ID とログメッセージの管理方法は、Java の enum を用いてまとめる方法も存在するが、本ガイドラインでは一般的なプロパティファイルを用いた方法を紹介する。

本実装例では

1. Logger ラッパークラス
2. プロパティファイル

を作成することで実現する。

ここでは Logger ラッパークラスを `LogIdBasedLogger`、プロパティファイルを `log-messages.properties` とする。

- *LogIdBasedLogger* (Logger ラッパークラス)

```
package com.example.sample.common.logger;

import java.text.MessageFormat;
import java.util.Arrays;
import java.util.Locale;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.NoSuchMessageException;
import org.springframework.context.support.ResourceBundleMessageSource;

public class LogIdBasedLogger {

    private static final String UNDEFINED_MESSAGE_FORMAT = "UNDEFINED-MESSAGE id:
↪{0} arg:{1}";    // (1)

    private static ResourceBundleMessageSource messageSource = new␣
↪ResourceBundleMessageSource();// (2)

    static {    // (3)
        messageSource.setDefaultEncoding("UTF-8");          // (4)
        messageSource.setBasenames("i18n/log-messages");    // (5)
    }

    private final Logger logger;

    private LogIdBasedLogger(Class<?> clazz) {
        logger = LoggerFactory.getLogger(clazz);            // (6)
    }

    public static LogIdBasedLogger getLogger(Class<?> clazz) {
        return new LogIdBasedLogger(clazz);
    }

    public boolean isDebugEnabled() {                       // (7)
        return logger.isDebugEnabled();
    }

    public void debug(String format, Object... args) {
        logger.debug(format, args);                         // (8)
    }

    public void info(String id, Object... args) {
        if (logger.isInfoEnabled()) {
            logger.info(createLogMessage(id, args));        // (9)
        }
    }

    public void warn(String id, Object... args) {
        if (logger.isWarnEnabled()) {
            logger.warn(createLogMessage(id, args));        // (9)
        }
    }

    public void error(String id, Object... args) {
        if (logger.isErrorEnabled()) {
            logger.error(createLogMessage(id, args));       // (9)
        }
    }
```

```
    public void trace(String id, Object... args) {
        if (logger.isTraceEnabled()) {
            logger.trace(createLogMessage(id, args));       // (9)
        }
    }

    public void warn(String id, Throwable t, Object... args) {
        if (logger.isWarnEnabled()) {
            logger.warn(createLogMessage(id, args), t);     // (9)
        }
    }

    public void error(String id, Throwable t, Object... args) {
        if (logger.isErrorEnabled()) {
            logger.error(createLogMessage(id, args), t);    // (9)
        }
    }

    private String createLogMessage(String id, Object... args) {
        return getMessage(id, args);
    }

    private String getMessage(String id, Object... args) {
        String message;
        try {
            message = messageSource.getMessage(id, args, Locale
                    .getDefault());
        } catch (NoSuchMessageException e) {                // (10)
            message = MessageFormat.format(UNDEFINED_MESSAGE_FORMAT, id, Arrays
                    .toString(args));
        }
        return message;
    }
}
```

項番	説明
(1)	ログ ID 未定義時のログメッセージ。ここでは例として `org.terasoluna.gfw.common.exception.ExceptionLogger` と同じメッセージを使用する。

Continued on next page

表 7.4 – continued from previous page

項番	説明
(2)	MessageSource でログメッセージを取得する実装例。 メッセージデータを管理する MessageSource は、汎用性を高めるため static 領域に格納している。 このような実装をすることで DI コンテナへのアクセス可否に依存しなくなるため、Logger ラッパークラスをいつでも使用することができるようになる。
(3)	static イニシャライザにて MessageSource を生成する。 本実装では i18n に配置した log-messages.properties を読み込む。
(4)	プロパティファイルをパースする際に使用する文字コードを設定する。 本実装ではプロパティファイルは UTF-8 エンコードとしたので UTF-8 を指定する。 詳細は、メッセージ管理のプロパティに設定したメッセージの表示を参照されたい。
(5)	国際化を考慮し setBasenames メソッドを使用してプロパティファイルを指定する。 setBasenames の詳細は ReloadableResourceBundleMessageSource クラスの setBasenames の JavaDoc を参照されたい。
(6)	Logger ラッパークラスにおいても、SLF4J を使用する。ロギングライブラリの実装を直接使用しない。
(7)	DEBUG レベルのログ出力を許可しているか、判定する。 使用時の注意点については、ログ出力の記述の注意点を参照されたい。
(8)	本実装例では DEBUG レベルのログにはログ ID を使わない。引数のログメッセージをそのまま、ログ出力する。
(9)	TRACE/INFO/WARN/ERROR レベルのログはログ ID に該当するログメッセージをプロパティファイルから取得して、ログ出力する。

Continued on next page

表 7.4 – continued from previous page

項番	説明
(10)	getMessage を呼び出す際にプロパティファイルにログ ID が記載されていないと例外:`NoSuchMessageException` が発生する。 そのため `NoSuchMessageException` を catch し、ログ ID がプロパティファイルに定義されていない旨のログメッセージを出力する。

- *log-messages.properties* (プロパティファイル)

```
i.ab.cd.1001 = This message is Info-Level. {0}
w.ab.cd.2001 = This message is Warn-Level. {0}
e.ab.cd.3001 = This message is Error-Level. {0}
t.ab.cd.4001 = This message is Trace-Level. {0}
```

注釈: 本ガイドラインでは、画面出力用メッセージとログ出力用メッセージを別々に管理するため、新たにプロパティファイルを作成しているが 1 ファイルにしてもかまわない。

アプリケーションの性質やメッセージの管理方法に合わせてファイルの単位を決めること。

実行結果は、以下のようになる。

- 呼び出しサンプル

```
package com.example.sample.app.welcome;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.example.sample.common.logger.LogIdBasedLogger;

@Controller
public class HomeController {

    private static final LogIdBasedLogger logger = LogIdBasedLogger
            .getLogger(HomeController.class);

    @RequestMapping(value = "/", method = { RequestMethod.GET,
            RequestMethod.POST })
    public String home(Model model) {
        logger.debug("debug log");
        logger.info("i.ab.cd.1001","replace_value_1");
        logger.warn("w.ab.cd.2001","replace_value_2");
        logger.error("e.ab.cd.3001","replace_value_3");
        logger.trace("t.ab.cd.4001","replace_value_4");
        logger.info("i.ab.cd.1002","replace_value_5");
        return "welcome/home";
    }
}
```

- ログ出力例

```
date:2016-05-30 17:34:18.590  thread:http-bio-8080-exec-3  X-
↪Track:e2a65cd9160b48d6aaeb63fe6e751c6b  level:DEBUG  logger:com.example.sample.
↪app.welcome.HomeController   message:debug log
date:2016-05-30 17:34:18.590  thread:http-bio-8080-exec-3  X-
↪Track:e2a65cd9160b48d6aaeb63fe6e751c6b  level:INFO   logger:com.example.sample.
↪app.welcome.HomeController   message:This message is Info-Level. replace_value_1
date:2016-05-30 17:34:18.590  thread:http-bio-8080-exec-3  X-
↪Track:e2a65cd9160b48d6aaeb63fe6e751c6b  level:WARN   logger:com.example.sample.
↪app.welcome.HomeController   message:This message is Warn-Level. replace_value_2
date:2016-05-30 17:34:18.590  thread:http-bio-8080-exec-3  X-
↪Track:e2a65cd9160b48d6aaeb63fe6e751c6b  level:ERROR  logger:com.example.sample.
↪app.welcome.HomeController   message:This message is Error-Level. replace_value_3
date:2016-05-30 17:34:18.590  thread:http-bio-8080-exec-3  X-
↪Track:e2a65cd9160b48d6aaeb63fe6e751c6b  level:TRACE  logger:com.example.sample.
↪app.welcome.HomeController   message:This message is Trace-Level. replace_value_4
date:2016-05-30 17:34:18.590  thread:http-bio-8080-exec-3  X-
↪Track:e2a65cd9160b48d6aaeb63fe6e751c6b  level:INFO   logger:com.example.sample.
↪app.welcome.HomeController   message:UNDEFINED-MESSAGE id:i.ab.cd.1002␣
↪arg:[replace_value_5]
```

ログメッセージの出力フォーマットの統一

ログメッセージの出力フォーマットは、下表のとおりログ出力の方式ごとで異なる。

そのため出力ログフォーマットの統一には、ログ出力フォーマットをもう一方のフォーマットに合わせる、または、両方とも独自のフォーマットに統一する必要がある。

本ガイドラインでは、業務ロジックで出力するログにフォーマットを定める例と、両方とも独自のフォーマット（[{例外コード (メッセージ ID) またはログ ID}], {メッセージまたはログメッセージ}）に統一する例を説明する。

項番	ログ出力方式	該当ログ	デフォルトフォーマット
(1)	業務ロジックで明示的にログを出力	アクセスログ・外部通信ログなど	なし
(2)	フレームワークが例外を検知して暗黙的にログを出力	業務エラーログ・システムエラーログなど	[{例外コード (メッセージ ID)}] {メッセージ}

注釈: 共通ライブラリ の例外ハンドリングの仕組みにより、例外発生時に出力される「業務エラーログ」および「システムエラーログ」は上記の表のデフォルトフォーマットで出力される。

フレームワークが例外を検知して出力するログのフォーマットに統一

業務ロジックで出力するログをフレームワークが例外を検知して出力するログのフォーマットに合わせるための実装例を示す。

本ガイドラインでは Logger ラッパークラス (`LogIdBasedLogger`) に、フォーマットを行う処理を追加して実現する。

```
package com.example.sample.common.logger;

import java.text.MessageFormat; // (1)

// omitted

public class LogIdBasedLogger {

    private static final String LOG_MESSAGE_FORMAT = "[{0}] {1}"; // (2)

    // omitted
```

```
    private String createLogMessage(String id, String... args) {
        return MessageFormat.format(LOG_MESSAGE_FORMAT, id, getMessage(id,
                args)); // (1)
    }

    // omitted

}
```

項番	説明
(1)	ログメッセージフォーマットを元にログメッセージを作成する処理を追加する
(2)	フォーマットを定義する。 {0}はログ ID、{1}はログメッセージがリプレースされる。

実行結果は、以下のようになる。

```
date:2016-05-30 16:32:33.239  thread:http-bio-8080-exec-4  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:DEBUG  logger:com.example.sample.
↪app.welcome.HomeController   message:debug log
date:2016-05-30 16:32:33.239  thread:http-bio-8080-exec-4  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:INFO   logger:com.example.sample.
↪app.welcome.HomeController   message:[i.ab.cd.1001] This message is Info-Level.␣
↪replace_value_1
date:2016-05-30 16:32:33.239  thread:http-bio-8080-exec-4  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:WARN   logger:com.example.sample.
↪app.welcome.HomeController   message:[w.ab.cd.2001] This message is Warn-Level.␣
↪replace_value_2
date:2016-05-30 16:32:33.239  thread:http-bio-8080-exec-4  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:ERROR  logger:com.example.sample.
↪app.welcome.HomeController   message:[e.ab.cd.3001] This message is Error-Level.␣
↪replace_value_3
date:2016-05-30 17:34:18.590  thread:http-bio-8080-exec-3  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:TRACE  logger:com.example.sample.
↪app.welcome.HomeController   message:[t.ab.cd.4001] This message is Trace-Level.␣
↪replace_value_4
date:2016-05-30 16:32:33.239  thread:http-bio-8080-exec-4  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:INFO   logger:com.example.sample.
↪app.welcome.HomeController   message:[i.ab.cd.1002] UNDEFINED-MESSAGE id:i.ab.cd.
↪1002 arg:[replace_value_5]
```

独自のフォーマットに統一

業務ロジックとフレームワークが出力するログを独自のフォーマット（[{例外コード (メッセージ ID) またはログ ID}], {メッセージまたはログメッセージ}）に統一する実装例を示す。

業務ロジックで出力するログにフォーマットを定義

業務ロジックで出力するログを前述のフォーマットで出力する例を示す。

本ガイドラインでは Logger ラッパークラス (`LogIdBasedLogger`) に、フォーマットを行う処理を追加して実現する。

```
package com.example.sample.common.logger;

import java.text.MessageFormat; // (1)

// omitted

public class LogIdBasedLogger {

    private static final String LOG_MESSAGE_FORMAT = "[{0}], {1}"; // (2)

    // omitted

    private String createLogMessage(String id, String... args) {
        return MessageFormat.format(LOG_MESSAGE_FORMAT, id, getMessage(id,
                args)); // (1)
    }

    // omitted

}
```

項番	説明
(1)	ログメッセージフォーマットを元にログメッセージを作成する処理を追加する
(2)	フォーマットを定義する。 `{0}`はログ ID、`{1}`はログメッセージがリプレースされる。

実行結果は、以下のようになる。

```
date:2016-05-30 16:32:33.239  thread:http-bio-8080-exec-4  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:DEBUG  logger:com.example.sample.
↪app.welcome.HomeController   message:debug log
date:2016-05-30 16:32:33.239  thread:http-bio-8080-exec-4  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:INFO   logger:com.example.sample.
↪app.welcome.HomeController   message:[i.ab.cd.1001], This message is Info-Level.␣
↪replace_value_1
date:2016-05-30 16:32:33.239  thread:http-bio-8080-exec-4  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:WARN   logger:com.example.sample.
↪app.welcome.HomeController   message:[w.ab.cd.2001], This message is Warn-Level.␣
↪replace_value_2
```

```
date:2016-05-30 16:32:33.239  thread:http-bio-8080-exec-4  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:ERROR  logger:com.example.sample.
↪app.welcome.HomeController   message:[e.ab.cd.3001], This message is Error-Level.
 replace_value_3
date:2016-05-30 17:34:18.590  thread:http-bio-8080-exec-3  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:TRACE  logger:com.example.sample.
↪app.welcome.HomeController   message:[t.ab.cd.4001], This message is Trace-Level.
 replace_value_4
date:2016-05-30 16:32:33.239  thread:http-bio-8080-exec-4  X-
↪Track:4f61314a51524ab3a41832b0ceae7119  level:INFO   logger:com.example.sample.
↪app.welcome.HomeController   message:[i.ab.cd.1002], UNDEFINED-MESSAGE␣
↪arg:[replace_value_5]
```

フレームワークが出力するログのフォーマットを変更

フレームワークが出力するログを前述のフォーマットで出力する例を示す。

業務エラーログやシステムエラーログのフォーマットを変更するには、applicationContext.xml の ExceptionLogger の bean 定義を変更する。

以下に、ExceptionLogger の定義の例を挙げる。

- **applicationContext.xml**

```
<!-- Exception Logger. -->
<bean id="exceptionLogger"
    class="org.terasoluna.gfw.common.exception.ExceptionLogger">
    <property name="exceptionCodeResolver" ref="exceptionCodeResolver" />
    <property name="logMessageFormat" value="[{0}], {1}" />    <!-- (1) -->
</bean>
```

項番	説明
(1)	logMessageFormat にフォーマットを定義する。 {0}は例外コード (メッセージ ID)、{1}はメッセージがリプレースされる。

実行結果は、以下のようになる。

```
date:2013-09-19 21:03:06   thread:tomcat-http--3   X-
↪Track:c19eec546b054d54a13658f94292b24f    level:ERROR logger:o.t.gfw.common.
↪exception.ExceptionLogger         message:[e.ad.od.9012], not found item entity.␣
↪item code [10-123456].
...
// stackTarace omitted
```

7.1.4 Appendix

MDC の使用

MDC(Mapped Diagnostic Context) を利用することで、横断的なログ出力が可能となる。
1 リクエスト中に出力されるログに、同じ情報 (ユーザー名やリクエストで一意な ID) を
埋め込んで出力することにより、ログのトレーサビリティが向上する。

MDC は、スレッドローカルな Map を内部にもち、キーに対して値を put する。remove されるまで、ログに put した値を出力することができる。
Filter などでリクエストの先頭で put し、処理終了時に remove すればよい。

基本的な使用方法

次に、MDC を用いた例を挙げる。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

public class Main {

    private static final Logger logger = LoggerFactory.getLogger(Main.class);

    public static void main(String[] args) {
        String key = "MDC_SAMPLE";
        MDC.put(key, "sample"); // (1)
        try {
            logger.debug("debug log");
            logger.info("info log");
            logger.warn("warn log");
            logger.error("error log");
        } finally {
            MDC.remove(key); // (2)
        }
        logger.debug("mdc removed!");
    }
}
```

logback.xml の<pattern>に %X{キー名}形式で出力フォーマットを定義することで、MDC に追加した値をログに出力できる。

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
        <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tthread:%thread\tmdcSample:
↪%X{MDC_SAMPLE}\tlevel:%-5level\t\tmessage:%msg%n]]></pattern>
    </encoder>
</appender>
```

実行結果は、以下のようになる。

```
date:2013-11-08 17:45:48    thread:main mdcSample:sample    level:DEBUG    ␣
↪message:debug log
date:2013-11-08 17:45:48    thread:main mdcSample:sample    level:INFO     ␣
↪message:info log
date:2013-11-08 17:45:48    thread:main mdcSample:sample    level:WARN     ␣
↪message:warn log
date:2013-11-08 17:45:48    thread:main mdcSample:sample    level:ERROR    ␣
↪message:error log
date:2013-11-08 17:45:48    thread:main mdcSample:  level:DEBUG     message:mdc␣
↪removed!
```

> **注釈:** `MDC.clear()` を実行すると、追加したすべての値が削除される。

Filter で MDC に値を Put する

共通ライブラリからは Filter で MDC へ値の追加・削除するためのベースクラスとして、
`org.terasoluna.gfw.web.logging.mdc.AbstractMDCPutFilter`
を提供している。またその実装クラスとして、

- リクエスト毎にユニークな ID を MDC に設定する `org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter`
- Spring Security の認証ユーザ名を MDC に設定する `org.terasoluna.gfw.security.web.logging.UserIdMDCPutFilter`

を提供している。

Filter で独自の値を MDC に追加したい場合は
`org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter` の実装を参考に
`AbstractMDCPutFilter` を実装すればよい。

MDCFilter の使用方法

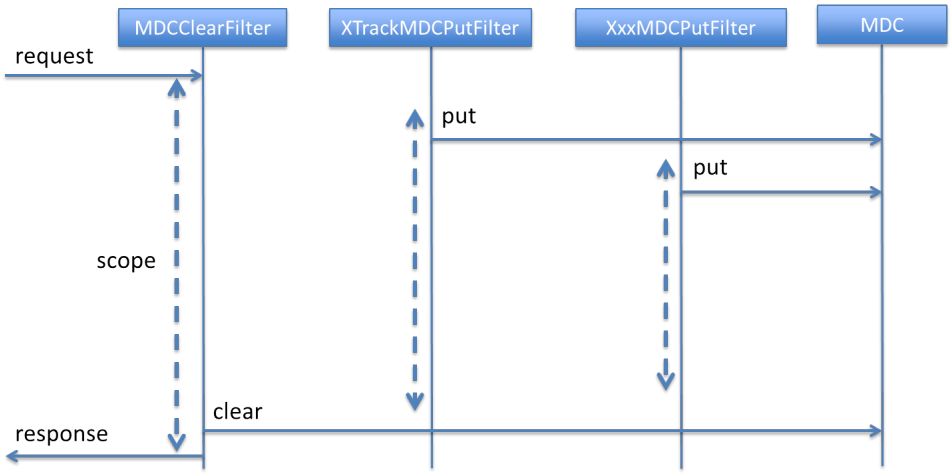
web.xml の filter 定義に MDCFilter の定義を追加する。

```
<!-- omitted -->

<!-- (1) -->
<filter>
```

```
    <filter-name>MDCClearFilter</filter-name>
    <filter-class>org.terasoluna.gfw.web.logging.mdc.MDCClearFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>MDCClearFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- (2) -->
<filter>
    <filter-name>XTrackMDCPutFilter</filter-name>
    <filter-class>org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter</filter-
↪class>
</filter>
<filter-mapping>
    <filter-name>XTrackMDCPutFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- (3) -->
<filter>
    <filter-name>UserIdMDCPutFilter</filter-name>
    <filter-class>org.terasoluna.gfw.security.web.logging.UserIdMDCPutFilter</
↪filter-class>
</filter>
<filter-mapping>
    <filter-name>UserIdMDCPutFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- omitted -->
```

項番	説明
(1)	MDC の内容をクリアする MDCClearFilter を設定する。 各種 MDCPutFilter が追加した MDC への値を、この Filter が消去する。
(2)	XTrackMDCPutFilter を設定する。XTrackMDCPutFilter はキー」X-Track」にリクエスト ID を put する。
(3)	UserIdMDCPutFilter を設定する。UserIdMDCPutFilter はキー」USER」にユーザー ID を put する。

MDCClearFilter は以下のシーケンス図のように、後処理として MDC の内容をクリアするため、各種 MDCPutFilter よりも、先に定義すること。



logback.xml の<pattern>に %X{X-Track}および、%X{USER}を追加することで、リクエスト ID とユーザー ID をログに出力することができる。

```
<!-- omitted -->
<appender name="APPLICATION_LOG_FILE" class="ch.qos.logback.core.rolling.
↪RollingFileAppender">
    <file>${app.log.dir:-log}/projectName-application.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>${app.log.dir:-log}/projectName-application-%d{yyyyMMdd}.
↪log</fileNamePattern>
        <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder>
```

```
        <charset>UTF-8</charset>
        <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tthread:%thread\tUSER:%X
↪{USER}\tX-Track:%X{X-Track}\tlevel:%-5level\tlogger:%-48logger{48}\tmessage:%msg
↪%n]]></pattern>
    </encoder>
</appender>
<!-- omitted -->
```

ログの出力例

```
date:2013-09-06 23:05:22  thread:tomcat-http--3   USER:   X-
↪Track:97988cc077f94f9d9d435f6f76027428    level:DEBUG logger:o.t.g.w.logging.
↪HttpSessionEventLoggingListener  message:SESSIONID
↪#D7AD1D42D3E77D61DB64E7C8C65CB488 sessionCreated : org.apache.catalina.session.
↪StandardSessionFacade@e51960
date:2013-09-06 23:05:22  thread:tomcat-http--3   USER:anonymousUser  X-
↪Track:97988cc077f94f9d9d435f6f76027428    logger:o.t.gfw.web.logging.
↪TraceLoggingInterceptor      message:[START CONTROLLER] HomeController.
↪home(Locale,Model)
date:2013-09-06 23:05:22  thread:tomcat-http--3   USER:anonymousUser  X-
↪Track:97988cc077f94f9d9d435f6f76027428    level:INFO  logger:c.terasoluna.
↪logging.app.welcome.HomeController  message:Welcome home! The client locale is␣
↪ja.
date:2013-09-06 23:05:22  thread:tomcat-http--3   USER:anonymousUser  X-
↪Track:97988cc077f94f9d9d435f6f76027428    logger:o.t.gfw.web.logging.
↪TraceLoggingInterceptor      message:[END CONTROLLER  ] HomeController.
↪home(Locale,Model)-> view=home, model={serverTime=2013/09/06 23:05:22 JST}
date:2013-09-06 23:05:22  thread:tomcat-http--3   USER:anonymousUser  X-
↪Track:97988cc077f94f9d9d435f6f76027428    logger:o.t.gfw.web.logging.
↪TraceLoggingInterceptor      message:[HANDLING TIME   ] HomeController.
↪home(Locale,Model)-> 36,508,860 ns
```

注釈: `UserIdMDCPutFilter` が MDC に put するユーザー情報は Spring Security の Filter により作成される。前述のように `UserIdMDCPutFilter` を web.xml に定義した場合、ユーザー ID がログに出力されるのは Spring Security の一連の処理が終わった後になる。ユーザー情報が生成された後、すぐにログに出力したい場合は、web.xml の定義は削除して、以下のように Spring Security の Filter に組み込む必要がある。

spring-security.xml には以下のような定義を追加する。

```
<sec:http>
    <!-- omitted -->
    <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
↪<!-- (1) -->
    <!-- omitted -->
</sec:http>

<!-- (2) -->
```

```
<bean id="userIdMDCPutFilter" class="org.terasoluna.gfw.security.web.
↪logging.UserIdMDCPutFilter">
</bean>
```

項番	説明
(1)	Bean 定義した `UserIdMDCPutFilter` を」ANONYMOUS_FILTER」の後に追加する。
(2)	`UserIdMDCPutFilter` を定義する。

blank プロジェクトでは `UserIdMDCPutFilter` を spring-security.xml に定義している。

共通ライブラリが提供するログ出力関連機能

HttpSessionEventLoggingListener

`org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener` は、セッションの生成・破棄・活性・非活性、セッションへの属性の追加・削除のタイミングで debug ログを出力するためのリスナークラスである。

web.xml に、以下を追加すればよい。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/
↪2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/
↪ns/javaee/web-app_3_0.xsd"
    version="3.0">
    <listener>
        <listener-class>org.terasoluna.gfw.web.logging.
↪HttpSessionEventLoggingListener</listener-class>
    </listener>

    <!-- omitted -->
</web-app>
```

logback.xml には、以下のように `org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener` を、debug レベルで設定する。

```
<logger
    name="org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener"> <!--␣
↪(1) -->
    <level value="debug" />
```

```
</logger>
```

以下のようなデバッグログが出力される。

```
date:2013-09-06 16:41:33    thread:tomcat-http--3   USER:   X-
↪Track:c004ddb56a3642d5bc5f6b5d884e5db2        level:DEBUG     logger:o.t.g.w.
↪logging.HttpSessionEventLoggingListener  message:SESSIONID
↪#EDC3C240A7A1CCE87146A6BA1321AD0F sessionCreated : org.apache.catalina.session.
↪StandardSessionFacade@f00e0f
```

`@SessionAttributes` など、Session を使用してオブジェクトのライフサイクルを管理している場合、本リスナーを利用して、セッションへ追加した属性が、想定通りに削除されているか確認することを、強く推奨する。

TraceLoggingInterceptor

`org.terasoluna.gfw.web.logging.TraceLoggingInterceptor` は、Controller の処理開始、終了をログ出力する `HandlerInterceptor` である。終了時には Controller が返却した View 名と Model に追加された属性、および Controller の処理に要した時間も出力する。

spring-mvc.xml の`<mvc:interceptors>`内に以下のように `TraceLoggingInterceptor` を追加する。

```
<mvc:interceptors>
    <!-- omitted -->
    <mvc:interceptor>
        <mvc:mapping path="/**" />
        <mvc:exclude-mapping path="/resources/**" />
        <bean
            class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor">
        </bean>
    </mvc:interceptor>
    <!-- omitted -->
</mvc:interceptors>
```

デフォルトでは、Controller の処理に 3 秒以上かかった場合に WARN ログを出力する。

この閾値を変える場合は、`warnHandlingNanos` プロパティにナノ秒単位で指定する。

閾値を 10 秒 (10 * 1000 * 1000 * 1000 ナノ秒) に変更したい場合は以下のように設定すればよい。

このとき、10 秒（10000000000 ナノ秒）のように int 型の範囲を超える閾値を設定する場合は、long 型で値を設定する点に留意されたい。

```
<mvc:interceptors>
    <!-- omitted -->
    <mvc:interceptor>
        <mvc:mapping path="/**" />
```

```
        <mvc:exclude-mapping path="/resources/**" />
        <bean
            class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor">
            <property name="warnHandlingNanos" value="#{10L * 1000L * 1000L *␣
↪1000L}" />
        </bean>
    </mvc:interceptor>
    <!-- omitted -->
</mvc:interceptors>
```

logback.xml には、以下のように、org.terasoluna.gfw.web.logging.TraceLoggingInterceptor を trace レベルで設定する。

```
<logger name="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor"> <!-- (1) -->
    <level value="trace" />
</logger>
```

ExceptionLogger

例外発生時のロガーとして、org.terasoluna.gfw.common.exception.ExceptionLogger が提供されている。

使用方法は、」例外ハンドリング」の」*How to use*」を参照されたい。

7.2 プロパティ管理

7.2.1 Overview

本節では、プロパティの管理方法について説明する。

プロパティとして管理が必要となる値は、以下の 2 つに分類することができる。

項番	分類	説明	例
1.	環境依存設定値	アプリケーションが動作する環境に応じて指定する値を変える必要がある設定値。 システム構成などの非機能要件に依存する。	• データベースの接続情報 (接続 URL、接続ユーザ、パスワードなど) • ファイルの保存先 (ディレクトリのパスなど) • more …
2.	アプリケーション設定値	アプリケーションの動作をカスタマイズできる様にするための設定値。 アプリケーションの機能要件に依存する。	• パスワード有効日数 • 予約期間日数 • more …

注釈: 本ガイドラインでは、これらの設定値については、プロパティとして管理 (プロパティファイルに定義) することを推奨している。

これらの設定値をプロパティから取得する仕組みにしておくと、設定値を変更する際に、アプリケーション (war ファイルや jar ファイル) を再ビルドする必要がないため、テスト済みのアプリケーションをプロダクト環境にリリースする事が可能になる。

ちなみに: プロパティとして管理している値は、JVM のシステムプロパティ (-D オプション) や OS の環境変数から取得することができる。アクセス順番については、「*How to use*」を参照されたい。

プロパティとして管理されている値は、以下の 2 箇所で利用することができる。

- bean 定義ファイル
- DI コンテナで管理する Java クラス

7.2.2 How to use

プロパティファイル定義方法について

Bean 定義ファイルに <context:property-placeholder/> タグを定義することで、Java クラスや Bean 定義ファイル内でプロパティファイル中の値にアクセスできるようになる。

<context:property-placeholder/> タグは、指定されたプロパティファイル群を読み込み、

@Value アノテーションや、 Bean 定義ファイル中で、${xxx}形式で指定されたプロパティファイルのキー xxx に対する値を取得できる。

> **注釈:** ${xxx:defaultValue}形式で指定すると、プロパティファイルにキー xxx の設定が存在しない場合に defaultValue を使用する。

以下に、プロパティファイルの定義方法について説明する。

bean 定義ファイル

- applicationContext.xml
- spring-mvc.xml

```
<context:property-placeholder location="classpath*:META-INF/spring/*.
↪properties"/>  <!-- (1) -->
```

項番	説明
(1)	location に設定する値は、リソースのロケーションパスを設定すること。 location 属性には、カンマ区切りで複数のパスを指定することができる。 上記設定により、クラスパス中の META-INF/spring ディレクトリ配下の properties ファイルを読み込む。 一度設定すれば、あとは META-INF/spring 以下に properties ファイルを追加するだけで良い。 location の設定値の詳細は、リファレンスを参照されたい。

注釈: <context:property-placeholder>の定義は、applicationContext.xml と spring-mvc.xml の両方に定義が必要である。

デフォルトでは、以下の順番でプロパティにアクセスする。

1. 実行中の JVM のシステムプロパティ
2. 環境変数
3. アプリケーション定義のプロパティファイル

デフォルトでは、すべての環境関連のプロパティ (JVM のシステムプロパティと環境変数) を読み込んだ後に、アプリケーションに定義されたプロパティファイルが検索され、読み込まれる。

読み込み順番を変更するには、<context:property-placeholder/> タグの local-override 属性を true に設定する。

このように設定することで、アプリケーションに定義されたプロパティが、優先的に有効になる。

bean 定義ファイル

```
<context:property-placeholder
    location="classpath*:META-INF/spring/*.properties"
    local-override="true" /> <!-- (1) -->
```

項番	説明
(1)	local-override 属性を true に設定すると、以下の順番でプロパティにアクセスする。 1. アプリケーション定義のプロパティ 2. 実行中の JVM のシステムプロパティ 3. 環境変数

注釈: 通常は上記の設定で十分である。複数の <context:property-placeholder/> タグを指定する場合、order 属性の値を設定することで、読み込みの順位付けをすることができる。

bean 定義ファイル

```
<context:property-placeholder
    location="classpath:/META-INF/property/extendPropertySources.
↪properties"
    order="1" ignore-unresolvable="true" /> <!-- (1) -->
<context:property-placeholder
    location="classpath*:/META-INF/spring/*.properties"
    order="2" ignore-unresolvable="true" /> <!-- (2) -->
```

項番	説明
(1)	order 属性を (2) より低い値を設定することにより、(2) より先に location 属性に該当するプロパティファイルが読み込まれる。 (2) で読み込んだプロパティファイル内のキーと重複するキーが存在する場合、(1) で取得した値が優先される。 ignore-unresolvable 属性を true にすることで、(2) のプロパティファイルのみにキーが存在する場合にエラーが発生するのを防ぐ。
(2)	order 属性を (1) より高い値を設定することにより、(1) の次に location 属性に該当するプロパティファイルが読み込まれる。 (1) で読み込んだプロパティファイル内のキーと重複するキーが存在する場合、(1) で取得した値が設定される。 ignore-unresolvable 属性を true にすることで、(1) のプロパティファイルのみにキーが存在する場合にエラーが発生するのを防ぐ。

bean 定義ファイル内でプロパティを使用する

データソースの設定ファイルを例に説明を行う。

以下の例では、プロパティファイル定義 (<context:property-placeholder/>) が指定されている前提で行う。

基本的には、bean 定義ファイルに、プロパティファイルのキーを ${} プレースホルダで設定することで、プロパティ値を設定することができる。

プロパティファイル

```
database.url=jdbc:postgresql://localhost:5432/shopping
database.password=postgres
database.username=postgres
database.driverClassName=org.postgresql.Driver
```

bean 定義ファイル

```
<bean id="dataSource"
    destroy-method="close"
    class="org.apache.commons.dbcp2.BasicDataSource">
    <property name="driverClassName"
              value="${database.driverClassName}"/>  <!-- (1) -->
    <property name="url" value="${database.url}"/>  <!-- (2) -->
    <property name="username" value="${database.username}"/>  <!-- (3) -->
    <property name="password" value="${database.password}"/>  <!-- (4) -->
    <!-- omitted -->
</bean>
```

項番	説明
(1)	`${database.driverClassName}` を設定することで、読み込まれたプロパティファイルのキー `database.driverClassName` に対する値が代入される。
(2)	`${database.url}` を設定することで、読み込まれたプロパティファイルのキー `database.url` に対する値が代入される。
(3)	`${database.username}` を設定することで、読み込まれたプロパティファイルのキー `database.username` に対する値が代入される。
(4)	`${database.password}` を設定することで、読み込まれたプロパティファイルのキー `database.password` に対する値が代入される。

properties ファイルのキーが読み込まれた結果、以下のように置換される。

```
<bean id="dataSource"
    destroy-method="close"
    class="org.apache.commons.dbcp2.BasicDataSource">
    <property name="driverClassName" value="org.postgresql.Driver"/>
    <property name="url"
              value="jdbc:postgresql://localhost:5432/shopping"/>
    <property name="username" value="postgres"/>
    <property name="password" value="postgres"/>
    <!-- omitted -->
</bean>
```

Java クラス内でプロパティを使用する

Java クラスでプロパティを利用する場合、プロパティの値を格納したいフィールドに `@Value` アノテーションを指定することで実現できる。

`@Value` アノテーションを使用するためには、そのオブジェクトは Spring の DI コンテナに管理されている必要がある。

以下の例では、プロパティファイル定義 (`<context:property-placeholder/>`) が指定されている前提で行う。

基本的に、変数に `@Value` アノテーションを付与し、value に propety ファイルのキーを `${}` プレースホルダで設定することで外部参照することができる。

プロパティファイル

```
item.upload.title=list of update file
item.upload.dir=file:/tmp/upload
item.upload.maxUpdateFileNum=10
```

Java クラス

```
@Value("${item.upload.title}")  // (1)
private String uploadTitle;

@Value("${item.upload.dir}")  // (2)
private Resource uploadDir;

@Value("${item.upload.maxUpdateFileNum}")  // (3)
private int maxUpdateFileNum;

// Getters and setters omitted
```

項番	説明
(1)	`@Value` アノテーションの value に `${item.upload.title}` を設定することで、読み込まれたプロパティファイルのキー `item.upload.title` に対する値が代入される。 `uploadTitle` には String クラスに」list of update file」が代入される。
(2)	`@Value` アノテーションの value に `${item.upload.dir}` を設定することで、読み込まれたプロパティファイルのキー `item.upload.dir` に対する値が代入される。 `uploadDir` には初期値」/tmp/upload」でオブジェクト生成された `org.springframework.core.io.Resource` オブジェクトが格納される。
(3)	`@Value` アノテーションの value に `${item.upload.maxUpdateFileNum}` を設定することで、読み込まれたプロパティファイルのキー `item.upload.maxUpdateFileNum` に対する値が代入される。 `maxUpdateFileNum` には整数型に 10 が代入される。

警告: Utility クラスなどの static メソッドからプロパティ値を利用したい場合も考えられるが、Bean 定義されないクラスでは `@Value` アノテーションによるプロパティ値の取得は行えない。このような場合には、 `@Component` アノテーションを付けた Helper クラスを作成し、`@Value` アノテーションでプロパティ値を取得することを推奨する。(当然、該当クラスは component-scan の対象にする必要がある。) プロパティ値を利用したいクラスは、Utility クラスにすべきでない。

7.2.3 How to extend

プロパティ値の取得方法の拡張について説明する。プロパティ値の取得方法の拡張は `org.springframework.context.support.PropertySourcesPlaceholderConfigurer` クラスを拡張することで実現できる。

拡張例として、暗号化したプロパティファイルを使用するケースを挙げる。

暗号化したプロパティ値を復号して使用する

セキュリティを強化するため、プロパティファイルを暗号化しておきたい場合がある。

例として、プロパティ値が暗号化されている場合に復号を行う実装を示す。(具体的な暗号化、復号方法は省略する。)

Bean 定義ファイル

- applicationContext.xml
- spring-mvc.xml

```
<!-- (1) -->
<bean class="com.example.common.property.
↪EncryptedPropertySourcesPlaceholderConfigurer">
    <!-- (2) -->
    <property name="locations"
              value="classpath*:/META-INF/spring/*.properties" />
</bean>
```

項番	説明
(1)	<context:property-placeholder/>の代わりに拡張した PropertySourcesPlaceholderConfigurer を定義する。 <context:property-placeholder/>タグを削除しておくこと。
(2)	property タグの name 属性に」locations」を設定し、value 属性に読み込むプロパティファイルパスを指定する。 読み込むプロパティファイルパスの指定方法は プロパティファイル定義方法について と同じ。

Java クラス

- 拡張した PropertySourcesPlaceholderConfigurer

```
public class EncryptedPropertySourcesPlaceholderConfigurer extends
    PropertySourcesPlaceholderConfigurer { // (1)
    @Override
    protected void doProcessProperties(
            ConfigurableListableBeanFactory beanFactoryToProcess,
            StringValueResolver valueResolver) { // (2)
        super.doProcessProperties(beanFactoryToProcess,
            new EncryptedValueResolver(valueResolver)); // (3)
```

```
    }
}
```

項番	説明
(1)	拡張した PropertySourcesPlaceholderConfigurer は `org.springframework.context.support.PropertySourcesPlaceholderConfigurer` を extend する。
(2)	`org.springframework.context.support.PropertySourcesPlaceholderConfigurer` クラスの `doProcessProperties` メソッドを override する。
(3)	親クラスの `doProcessProperties` を呼び出すが、 `valueResolver` は独自実装した valueResolver(`EncryptedValueResolver`) を使用する。 `EncryptedValueResolver` クラス内で、プロパティファイルの暗号化された value を取得した場合に復号する。

- EncryptedValueResolver.java

```
public class EncryptedValueResolver implements
                                    StringValueResolver { // (1)

    private final StringValueResolver valueResolver;

    EncryptedValueResolver(StringValueResolver stringValueResolver) { // (2)
        this.valueResolver = stringValueResolver;
    }

    @Override
    public String resolveStringValue(String strVal) { // (3)

        // Values obtained from the property file to the naming
        // as seen with the encryption target
        String value = valueResolver.resolveStringValue(strVal); // (4)

        // Target messages only, implement coding
        if (value.startsWith("Encrypted:")) { // (5)
```

```
            value =  value.substring(10); // (6)
            // omitted decryption
        }
        return value;
    }
}
```

項番	説明
(1)	拡張した `EncryptedValueResolver` は、`org.springframework.util.StringValueResolver` を実装する。
(2)	コンストラクタで `EncryptedValueResolver` クラスを生成したときに、`EncryptedPropertySourcesPlaceholderConfigurer` から引き継いできた `StringValueResolver` を設定する。
(3)	`org.springframework.util.StringValueResolver` の `resolveStringValue` メソッドを ovewride する。 `resolveStringValue` メソッド内にて、プロパティファイルの暗号化された value を取得した場合に復号する。 以降、(5)〜(6) は一例の処理になるため、実装によって処理が異なる。
(4)	コンストラクタで設定した `StringValueResolver` の `resolveStringValue` メソッドの引数にキーを指定して値を取得している。この値は実際にプロパティファイルに定義されている値である。
(5)	プロパティファイルの値が暗号化された値かどうかをチェックする。判定方法については実装によって異なる。 ここでは値が」Encrypted:」から始まるかどうかで、暗号化されているかどうかを判断する。 暗号化されている場合、(6) で復号を実施し、暗号化されていない場合、そのままの値を返却する。
(6)	プロパティファイルの暗号化された value の復号を行っている。(具体的な復号処理については省略する。) 復号の方法については実装によって異なる。

- プロパティを取得する Helper

```
@Value("${encrypted.property.string}") // (1)
private String testString;
```

```
@Value("${encrypted.property.int}") // (2)
private int testInt;

@Value("${encrypted.property.integer}") // (3)
private Integer testInteger;

@Value("${encrypted.property.file}") // (4)
private File testFile;

// Getters and setters omitted
```

項番	説明
(1)	`@Value` アノテーションの value に `${encrypted.property.string}` を設定することで、読み込まれたプロパティファイルのキー `encrypted.property.string` に対する値が復号されて代入される。 `testString` には String クラスに復号された値が代入される。
(2)	`@Value` アノテーションの value に `${encrypted.property.int}` を設定することで、読み込まれたプロパティファイルのキー `encrypted.property.int` に対する値が復号されて代入される。 `testInt` には整数型に復号された値が代入される。
(3)	`@Value` アノテーションの value に `${encrypted.property.integer}` を設定することで、読み込まれたプロパティファイルのキー `encrypted.property.integer` に対する値が復号されて代入される。 `testInteger` には Integer クラスに復号された値が代入される。
(4)	`@Value` アノテーションの value に `${encrypted.property.file}` を設定することで、読み込まれたプロパティファイルのキー `encrypted.property.file` に対する値が復号されて代入される。 `testFile` には初期値に復号された値でオブジェクト生成された File オブジェクトが格納される。(自動変換)

プロパティファイル

プロパティ値として、暗号化した値の prefix に、暗号化されていることを示す」Encrypted:」を付加している。
暗号化されているため、プロパティファイルの中身を見ても理解できない状態になっている。

```
encrypted.property.
↪string=Encrypted:ZlpbQRJRWlNAU1FGV0ASRVteXhJQVxJXXFFAS0JGV1Yc
encrypted.property.int=Encrypted:AwI=
encrypted.property.integer=Encrypted:AwICAgI=
encrypted.property.file=Encrypted:YkBdQldARkt/U1xTVVdfV1xGHFpGX14=
```

7.3 日付操作 (JSR-310 Date and Time API)

7.3.1 Overview

本ガイドラインでは、`java.util.Date`, `java.util.Calendar` に比べて、
様々な日時計算が提供されている JSR-310 Date and Time API の使用を推奨する。

> **注釈:** JSR-310 Date and Time API は Java8 から導入されたため、Java8 未満の環境は、Joda Time の利用を推奨している。Joda Time の利用方法は、 日付操作 *(Joda Time)* を参照されたい。

7.3.2 How to use

Date and Time API では、日付のみ扱うクラス、時刻のみ扱うクラスなど、用途に応じた様々なクラスが提供されている。

本ガイドラインでは、`java.time.LocalDate`, `java.time.LocalTime`, `java.time.LocalDateTime` を中心に説明を進めるが、主要な日時操作については、各クラスで提供されるメソッドの接頭辞が同一であるため、適時クラス名を置き換えて解釈されたい。

主に使われるクラスとメソッドを示す。

日時を扱う主なクラス

クラス名	説明	主なファクトリメソッド
java.time.LocalDate java.time.LocalTime java.time.LocalDateTime	タイムゾーン・時差の情報を持たない日付・時刻の操作を行うクラス	now 現在日時で生成 of 任意日時で生成 parse 日時文字列から生成 from 日時情報を持つ他オブジェクトから生成
java.time.OffsetTime java.time.OffsetDateTime java.time.ZonedDateTime	タイムゾーン・時差を考慮した日付・時刻の操作を行うクラス	同上
java.time.chrono.JapaneseDate	和暦の操作を行うクラス	同上

期間の情報を扱う主なクラス

クラス名	説明	主なファクトリメソッド
java.time.Period java.time.Duration	日時ベース、時間ベースの期間を扱うクラス	between 日時情報を持つ 2 つのオブジェクトの差から生成 from 時間量を持つ他オブジェクトから生成 of 任意期間で生成

フォーマットを扱うクラス

クラス名	説明	主なファクトリメソッド
java.time.format.DateTimeFormatter	日時のフォーマットに関する操作を行うクラス	ofPattern 指定されたパターンでフォーマッタを生成

各クラス・メソッドの具体的な利用方法を、以下で説明する。

注釈: 本ガイドラインで触れなかった内容を含め、詳細は Javadoc を参照されたい。

注釈: Date and Time API のクラスは、immutable である (日時計算等の結果は、新規オブジェクトであり、計算元オブジェクトに変化は起きない)。

日時取得

現在日時で取得

利用用途に合わせて `java.time.LocalTime` , `java.time.LocalDate` , `java.time.LocalDateTime` を使い分けること。以下に例を示す。

1. 時刻のみ取得したい場合は `java.time.LocalTime` を使用する。

```
LocalTime localTime =  LocalTime.now();
```

2. 日付のみ取得したい場合は `java.time.LocalDate` を使用する。

```
LocalDate localDate =  LocalDate.now();
```

3. 日付・時刻を取得したい場合は `java.time.LocalDateTime` を使用する。

```
LocalDateTime localDateTime = LocalDateTime.now();
```

年月日時分秒を指定して取得

of メソッドを使うことで特定の日時を指定することができる。以下に例を示す。

1. 時刻を指定して java.time.LocalTime を取得する。

```
// 23:30:59
LocalTime localTime =  LocalTime.of(23, 30, 59);
```

2. 日付を指定して java.time.LocalDate を取得する。

```
// 2015/12/25
LocalDate localDate =  LocalDate.of(2015, 12, 25);
```

3. 日付・時刻）を指定して java.time.LocalDateTime を取得する。

```
// 2015/12/25 23:30:59
LocalDateTime localDateTime = LocalDateTime.of(2015, 12, 25, 23, 30, 59);
```

また、java.time.temporal.TemporalAdjusters を使うことで様々な日時を取得することができる。

```
// LeapYear(2012/2)
LocalDate localDate1 = LocalDate.of(2012, 2, 1);

// Last day of month(2012/2/29)
LocalDate localDate2 = localDate1.with(TemporalAdjusters.lastDayOfMonth());

// Next monday (2012/2/6)
LocalDate localDate3 = localDate1.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
```

注釈: java.util.Calendar の仕様とは異なり、Month は 1 始まりである。

タイムゾーンを指定する場合の日時取得

国際的なアプリケーションを作成する場合、タイムゾーンを意識した設計を行う場合がある。
Date and Time API では、利用用途に合わせて、 java.time.OffsetTime , java.time.OffsetDateTime , java.time.ZonedDateTime を使い分けること。
以下に例を示す。

1. 時刻＋ UTC との時差を取得したい場合は、 java.time.OffsetTime を使用する。

```
// Ex, 12:30:11.567+09:00
OffsetTime offsetTime =  OffsetTime.now();
```

2. 日付・時刻＋ UTC との時差を取得したい場合は java.time.OffsetDateTime を使用する。

```
// Ex, 2015-12-25T12:30:11.567+09:00
OffsetDateTime offsetDateTime =  OffsetDateTime.now();
```

3. 日付・時刻＋ UTC との時差・地域を取得したい場合は java.time.ZonedDateTime を使用する。

```
// Ex, 2015-12-25T12:30:11.567+09:00[Asia/Tokyo]
ZonedDateTime zonedDateTime = ZonedDateTime.now();
```

また、これらのメソッドでは全て、タイムゾーンを表す java.time.ZoneId を引数に設定することで、タイムゾーンを考慮した現在日時が取得できる。

以下に java.time.ZoneId の例を示す。

```
ZoneId zoneIdTokyo = ZoneId.of("Asia/Tokyo");
OffsetTime offsetTime =  OffsetTime.now(zoneIdTokyo);
```

なお、 java.time.ZoneId は地域名/地名形式で定義する方法や、UTC からの時差で定義する方法がある。

```
ZoneId.of("Asia/Tokyo");
ZoneId.of("UTC+01:00");
```

java.time.OffsetDateTime , java.time.ZonedDateTime の 2 クラスは用途が似ているが、具体的には以下のような違いがある。

作成するシステムの特性に応じて適切なクラスを選択されたい。

クラス名	説明
java.time.OffsetDateTime	定量値（時差のみ）を持つため、各地域の時間の概念に変化がある場合も、システムに変化が起こらない。
java.time.ZonedDateTime	時差に加えて地域の概念があるため、各地域の時間の概念に変化があった場合、システムに変化が起こる。（政策としてサマータイム導入される場合など）

期間

期間の取得

日付ベースの期間を扱う場合は、 java.time.Period 、時間ベースの期間を扱う場合は、 java.time.Duration を使用する。

java.time.Duration で表される 1 日は厳密に 24 時間であるため、サマータイムの変化が解釈されずに想定通りの結果にならない可能性がある。

対して、 java.time.Period はサマータイムなどの概念を考慮した 1 日を表すため、サマータイムを扱うシステムであっても誤差は生じない。

以下に例を示す。

```
LocalDate date1 = LocalDate.of(2010, 01, 15);
LocalDate date2 = LocalDate.of(2011, 03, 18);
LocalTime time1 = LocalTime.of(11, 50, 50);
LocalTime time2 = LocalTime.of(12, 52, 53);

// One year, two months and three days.
Period pd = Period.between(date1, date2);

// One hour, two minutes and three seconds.
Duration dn = Duration.between(time1, time2);
```

> **注釈:** of メソッドを利用して、期間を指定して生成する方法もある。詳細は Period, Duration の Javadoc を参照されたい。

型変換

Date and Time API の各クラスの相互運用性

java.time.LocalTime , java.time.LocalDate , java.time.LocalDateTime はそれぞれ容易

に変換が可能である。以下に例を示す。

1. java.time.LocalTime から java.time.LocalDateTime への変換。

```
// Ex. 12:10:30
LocalTime localTime =  LocalTime.now();

// 2015-12-25 12:10:30
LocalDateTime localDateTime = localTime.atDate(LocalDate.of(2015, 12, 25));
```

2. java.time.LocalDate から java.time.LocalDateTime への変換。

```
// Ex. 2012-12-25
LocalDate localDate =  LocalDate.now();

// 2015-12-25 12:10:30
LocalDateTime localDateTime = localDate.atTime(LocalTime.of(12, 10, 30));
```

3. java.time.LocalDateTime から java.time.LocalTime , java.time.LocalDate への変換。

```
// Ex. 2015-12-25 12:10:30
LocalDateTime localDateTime =  LocalDateTime.now();

// 12:10:30
LocalTime localTime =  localDateTime.toLocalTime();

// 2012-12-25
LocalDate localDate =  localDateTime.toLocalDate();
```

同様に、java.time.OffsetTime , java.time.OffsetDateTime , java.time.ZonedDateTime もそれぞれ容易に変換が可能である。以下に例を示す。

1. java.time.OffsetTime から、 java.time.OffsetDateTime への変換。

```
// Ex, 12:30:11.567+09:00
OffsetTime offsetTime =  OffsetTime.now();

// 2015-12-25T12:30:11.567+09:00
OffsetDateTime offsetDateTime = offsetTime.atDate(LocalDate.of(2015, 12, 25));
```

2. java.time.OffsetDateTime から java.time.ZonedDateTime への変換。

```
// Ex, 2015-12-25T12:30:11.567+09:00
OffsetDateTime offsetDateTime =  OffsetDateTime.now();

```

```
// 2015-12-25T12:30:11.567+09:00[Asia/Tokyo]
ZonedDateTime zonedDateTime = offsetDateTime.atZoneSameInstant(ZoneId.of("Asia/
↪Tokyo"));
```

3. java.time.ZonedDateTime から java.time.OffsetDateTime , java.time.OffsetTime への変換。

```
// Ex, 2015-12-25T12:30:11.567+09:00[Asia/Tokyo]
ZonedDateTime zonedDateTime =  ZonedDateTime.now();

// 2015-12-25T12:30:11.567+09:00
OffsetDateTime offsetDateTime =  zonedDateTime.toOffsetDateTime();

// 12:30:11.567+09:00
OffsetTime offsetTime =  zonedDateTime.toOffsetDateTime().toOffsetTime();
```

また、時差情報を加えることで、java.time.LocalTime を java.time.OffsetTime に変換することも可能である。

```
// Ex, 12:30:11.567
LocalTime localTime =  LocalTime.now();

// 12:30:11.567+09:00
OffsetTime offsetTime = localTime.atOffset(ZoneOffset.ofHours(9));
```

この他についても、不足している情報（ LocalTime から LocalDateTime の変換であれば日付の情報が不足している の要領）を加えることで別のクラスへ変換が可能である。

変換メソッドは接頭辞が at または to で始まる。詳細は 各クラスの Javadoc を参照されたい。

java.util.Date との相互運用性

java.time.LocalDate 等のクラスは、java.time.Instant に変換したうえで java.util.Date に変換することが可能である。

以下に例を示す。

1. java.time.LocalDateTime から、 java.util.Date への変換。

```
LocalDateTime localDateTime = LocalDateTime.now();
Instant instant = localDateTime.toInstant(ZoneOffset.ofHours(9));
Date date = Date.from(instant);
```

2. java.util.Date から java.time.LocalDateTime への変換。

```
Date date = new Date();
Instant instant = date.toInstant();
LocalDateTime localDateTime = LocalDateTime.ofInstant(instant, ZoneId.
↪systemDefault());
```

> **注釈:** java.time.LocalTime , java.time.LocalDate は Instant 値を持たないため、一度 java.time.LocalDateTime に変換する必要がある。

java.sql パッケージとの相互運用性

Java8 より java.sql パッケージに改修が入り、 java.time パッケージとの相互変換メソッドが定義された。
以下に例を示す。

1. java.sql.Date から java.time.LocalDate への変換。

```
java.sql.Date date =  new java.sql.Date(System.currentTimeMillis());
LocalDate localDate = date.toLocalDate();
```

2. java.time.LocalDate から java.sql.Date への変換。

```
LocalDate localDate = LocalDate.now();
java.sql.Date date =  java.sql.Date.valueOf(localDate);
```

3. java.sql.Time から java.time.LocalTime への変換。

```
java.sql.Time time =  new java.sql.Time(System.currentTimeMillis());
LocalTime localTime = time.toLocalTime();
```

4. java.time.LocalTime から java.sql.Time への変換。

```
LocalTime localTime = LocalTime.now();
java.sql.Time time =  java.sql.Time.valueOf(localTime);
```

5. java.sql.Timestamp から java.time.LocalDateTime への変換。

```
java.sql.Timestamp timestamp =  new java.sql.Timestamp(System.currentTimeMillis());
LocalDateTime localDateTime = timestamp.toLocalDateTime();
```

6. java.time.LocalDateTime から java.sql.Timestamp への変換。

```
LocalDateTime localDateTime = LocalDateTime.now();
java.sql.Timestamp timestamp =  java.sql.Timestamp.valueOf(localDateTime);
```

org.terasoluna.gfw.common.date パッケージの利用方法

現在、Date and Time API 用の Date Factory は共通ライブラリから提供されていない。（参照： システム時刻 ）
ただし、暫定対処として、 org.terasoluna.gfw.common.date.ClassicDateFactory と java.sql.Date を利用することで、 java.time.LocalDate を生成できる。
java.time.LocalTime , java.time.LocalDateTime クラスに関しても、
java.time.LocalDate から変換することで生成できる。
以下に例を示す。

bean 定義ファイル ([projectname]-env.xml)

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.
↪DefaultClassicDateFactory" />
```

Java クラス

```
@Inject
ClassicDateFactory dateFactory;

public DateFactorySample getSystemDate() {

 java.sql.Date date = dateFactory.newSqlDate();
 LocalDate localDate = date.toLocalDate();

 // omitted
}
```

> **注釈:** Date and Time API に対応した Date Factory は今後追加予定である。

文字列へのフォーマット

日時情報を持つオブジェクトを文字列に変換するには、toString メソッドを使用する方法と、
java.time.fomat.DateTimeFormatter を使用する方法がある。

任意の日時文字列で出力したい場合は、java.time.fomat.DateTimeFormatter を使用し様々な日時文字列へ変換することが出来る。

java.time.fomat.DateTimeFormatter は、事前定義された ISO パターンのフォーマッタを利用する方法と、任意のパターンのフォーマットを定義して利用する方法がある。

```
DateTimeFormatter formatter1 = DateTimeFormatter.BASIC_ISO_DATE;

DateTimeFormatter formatter2 = DateTimeFormatter.ofPattern("G uuuu/MM/dd E")
                                                .withLocale(Locale.JAPANESE)
                                                .withResolverStyle(ResolverStyle.STRICT);
```

その際、文字列の形式の他に Locale と ResolverStyle （厳密性）を定義できる。

Locale のデフォルト値はシステムによって変化するため、初期化時に設定することが望ましい。

また、 ResolverStyle （厳密性）は ofPattern メソッドを使う場合、デフォルトで ResolverStyle.SMART が設定されるが、本ガイドラインでは予期せぬ挙動が起こらないよう、厳密に日付を解釈する ResolverStyle.STRICT の設定を推奨している。（ISO パターンのフォーマッタを利用する場合は ResolverStyle.STRICT が設定されている)

なお、Date and Time API では書式 yyyy は暦に対する年を表すため、暦によって解釈が異なる (西暦なら 2015 と解釈されるが、和暦なら 0027 と解釈される)。

西暦を表したい場合は、 yyyy 形式に変えて uuuu 形式を利用することを推奨する。定義されている書式一覧は DateTimeFormatter を参照されたい。

以下に例を示す。

```
DateTimeFormatter formatter1 = DateTimeFormatter.BASIC_ISO_DATE;

DateTimeFormatter formatter2 = DateTimeFormatter.ofPattern("G uuuu/MM/dd E")
                                                .withLocale(Locale.JAPANESE)
                                                .withResolverStyle(ResolverStyle.STRICT);

LocalDate localDate1 = LocalDate.of(2015, 12, 25);

// "2015-12-25"
System.out.println(localDate1.toString());
// "20151225"
System.out.println(formatter1.format(localDate1));
// "西暦 2015/12/25 金"
System.out.println(formatter2.format(localDate1));
```

また、これらの文字列を画面上に表示したい場合、

Date and Time API では、Joda Time と異なり、専用の JSP タグは存在していない。

JSTL の `fmt:formatDate` タグは、`java.util.Date` と、 `java.util.TimeZone` オブジェクトのみを扱うため、

JSP 上で Date and Time API のオブジェクトが持つ日時情報を表示する場合は、フォーマット済みの文字列を渡して表示する。

Controller クラス

```
@Controller
public class HomeController {

    @RequestMapping(value = "/", method = {RequestMethod.GET, RequestMethod.POST})
    public String home(Model model, Locale locale) {

        DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("uuuu/MM/dd")
                                            .withLocale(locale)
                                            .withResolverStyle(ResolverStyle.
↪STRICT);

        LocalDate localDate1 = LocalDate.now();

        model.addAttribute("currentDate", localDate1.toString());
        model.addAttribute("formattedCurrentDateString", dateFormatter.
↪format(localDate1));

    // omitted

    }
}
```

jsp ファイル

```
<p>currentDate =  ${f:h(currentDate)}.</p>
<p>formattedCurrentDateString =  ${f:h(formattedCurrentDateString)}.</p>
```

出力結果例 (html)

```
<p>currentDate =  2015-12-25.</p>
<p>formattedCurrentDateString =  2015/12/25.</p>
```

文字列からのパース

文字列への変換と同様に、`java.time.fomat.DateTimeFormatter` を用いることで、様々な日付文字列を Date and Time API のクラスへ変換することが出来る。

以下に例を示す。

```
DateTimeFormatter formatter1 = DateTimeFormatter.ofPattern("uuuu/MM/dd")
                                                .withLocale(Locale.JAPANESE)
                                                .withResolverStyle(ResolverStyle.
↪STRICT);

DateTimeFormatter formatter2 = DateTimeFormatter.ofPattern("HH:mm:ss")
                                                .withLocale(Locale.JAPANESE)
                                                .withResolverStyle(ResolverStyle.
↪STRICT);

LocalDate localDate = LocalDate.parse("2015/12/25", formatter1);
LocalTime localTime = LocalTime.parse("14:09:20", formatter2);
```

日付操作

Date and Time API では、日時の計算や比較などを容易に行うことが出来る。

以下に例を示す。

日時の計算

日時の計算をするために `plus` メソッドと `minus` メソッドが提供されている。

1. 時間の計算を行う場合の例。

```
LocalTime localTime =  LocalTime.of(20, 30, 50);
LocalTime plusHoursTime = localTime.plusHours(2);
LocalTime plusMinutesTime = localTime.plusMinutes(10);
LocalTime minusSecondsTime = localTime.minusSeconds(15);
```

2. 日付の計算を行う場合の例。

```
LocalDate localDate =  LocalDate.of(2015, 12, 25);
LocalDate plusYearsDate = localDate.plusYears(10);
LocalDate minusMonthsTime = localDate.minusMonths(1);
LocalDate plusDaysTime = localDate.plusDays(3);
```

注釈: `plus` メソッドに負数を渡すと、 `minus` メソッドを利用した場合と同様の結果が得られる。

minus メソッドも同様。

日時の比較

Date and Time API では、過去・未来・同時などの時系列の比較が行える。
以下に例を示す。

1. 時間の比較を行う場合の例。

```
LocalTime morning =  LocalTime.of(7, 30, 00);
LocalTime daytime =  LocalTime.of(12, 00, 00);
LocalTime evening =  LocalTime.of(17, 30, 00);

daytime.isBefore(morning); // false
morning.isAfter(evening); // true
evening.equals(LocalTime.of(17, 30, 00)); // true

daytime.isBefore(daytime); // false
morning.isAfter(morning); // false
```

2. 日付の比較を行う場合の例。

```
LocalDate may =  LocalDate.of(2015, 6, 1);
LocalDate june =  LocalDate.of(2015, 7, 1);
LocalDate july =  LocalDate.of(2015, 8, 1);

may.isBefore(june); // true
june.isAfter(july); // false
july.equals(may); // false

may.isBefore(may); // false
june.isAfter(june); // false
```

なお、Date and Time API では現在、Joda Time の `Interval` に当たるクラスは存在しない。

日時の判定

以下に日時の判定の例を示す。

1. 妥当な日時文字列かを判定したい場合、 `java.time.format.DateTimeParseException` の発生有無で判定できる。

```
String strDateTime = "aabbcc";
DateTimeFormatter timeFormatter  = DateTimeFormatter.ofPattern("HHmmss")
```

```
                                .withLocale(Locale.JAPANESE)
                                .withResolverStyle(ResolverStyle.STRICT);;

DateTimeFormatter dateFormatter  = DateTimeFormatter.ofPattern("uuMMdd")
                                .withLocale(Locale.JAPANESE)
                                .withResolverStyle(ResolverStyle.STRICT);;

try {
    // DateTimeParseException
    LocalTime localTime = LocalTime.parse(strDateTime, timeFormatter);
}
catch (DateTimeParseException e) {
    System.out.println("Invalid time string !!");
}

try {
    // DateTimeParseException
    LocalDate localDate = LocalDate.parse(strDateTime, dateFormatter);
}
catch (DateTimeParseException e) {
    System.out.println("Invalid date string !!");
}
```

2. うるう年かを判定したい場合、java.time.LocalDate の isLeapYear メソッドで判定できる。

```
LocalDate date1 = LocalDate.of(2012, 1, 1);
LocalDate date2 = LocalDate.of(2015, 1, 1);

date1.isLeapYear(); // true
date2.isLeapYear(); // false
```

年月日時分秒の取得

年月日時分秒をそれぞれ取得したい場合は、 get メソッドを利用する。

以下に日付に関する情報を取得する例を示す。

```
LocalDate localDate = LocalDate.of(2015, 2, 1);

// 2015
int year = localDate.getYear();

// 2
int month = localDate.getMonthValue();

// 1
int dayOfMonth = localDate.getDayOfMonth();

// 32 ( day of year )
int dayOfYear = localDate.getDayOfYear();
```

和暦（JapaneseDate）

Date and Time API では java.time.chrono.JapaneseDate という、和暦を扱うクラスが提供されている。

> **注釈:** java.time.chrono.JapaneseDate は、グレゴリオ暦が導入された明治 6 年 (西暦 1873 年) より前は利用できない。

和暦の取得

java.time.LocalDate と同様に、 now メソッド, of メソッドで取得できる。

また、java.time.chrono.JapaneseEra クラスを使うことで、和暦を指定した取得も行うことが出来る。

以下に例を示す。

```
JapaneseDate japaneseDate1 = JapaneseDate.now();
JapaneseDate japaneseDate2 = JapaneseDate.of(2015, 12, 25);
JapaneseDate japaneseDate3 = JapaneseDate.of(JapaneseEra.HEISEI, 27, 12, 25);
```

明治 6 年より前を引数に指定すると例外が発生する。

```
// DateTimeException
JapaneseDate japaneseDate = JapaneseDate.of(1500, 1, 1);
```

文字列へのフォーマット

java.time.fomat.DateTimeFormatter を用いることで、和暦日付へ変換することが出来る。利用の際には、 DateTimeFormatter#withChronology メソッドで暦を java.time.chrono.JapaneseChronology に設定する。

和暦日付でも様々なフォーマットを扱うことが出来るため、0 埋めや空白埋めなど要件に応じた出力が行える。

以下に空白埋めで和暦を表示する例を示す。

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("Gppy 年 ppM 月 ppd 日")
                                .withLocale(Locale.JAPANESE)
                                .withResolverStyle(ResolverStyle.STRICT)
                                .withChronology(JapaneseChronology.INSTANCE);

JapaneseDate japaneseDate = JapaneseDate.of(1992, 1, 1);
```

```
// "平成 4年 1月 1日"
System.out.println(formatter.format(japaneseDate));
```

文字列からのパース

java.time.fomat.DateTimeFormatter を用いることで、和暦文字列から java.time.chrono.JapaneseDate へ変換することが出来る。

以下に例を示す。

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("Gy 年 MM 月 dd 日")
                                .withLocale(Locale.JAPANESE)
                                .withResolverStyle(ResolverStyle.STRICT)
                                .withChronology(JapaneseChronology.INSTANCE);

JapaneseDate japaneseDate1 = JapaneseDate.from(formatter.parse("平成 27 年 12 月 25 日
↪"));
JapaneseDate japaneseDate2 = JapaneseDate.from(formatter.parse("明治 6 年 01 月 01 日"));
```

西暦・和暦の変換

from メソッドを使うことで java.time.LocalDate からの変換を容易に行える。

```
LocalDate localDate = LocalDate.of(2015, 12, 25);
JapaneseDate jpDate = JapaneseDate.from(localDate);
```

7.4 日付操作 (Joda Time)

7.4.1 Overview

java.util.Date 、 java.util.Calendar クラスの API は、非常に貧弱であるため、複雑な日付計算ができない。

本ガイドラインでは、日付計算が強力な Joda Time の使用を推奨している。

Joda Time では、 java.util.Date の代わりに、 org.joda.time.DateTime 、 org.joda.time.LocalDate や org.joda.time.LocalTime オブジェクトを用いて日付を表現する。

なお、 org.joda.time.DateTime 、 org.joda.time.LocalDate や org.joda.time.LocalTime オブジェクトは、immutable である (日付計算等の結果は、新規オブジェクトである)。

7.4.2 How to use

Joda Time, Joda Time JSP tags の利用方法を、以下で説明する。

日付取得

現在時刻を取得

利用用途に併せて、 `org.joda.time.DateTime`,`org.joda.time.LocalDate`, `org.joda.time.LocalTime` を使い分けること。以下に、使用方法を示す。

1. ミリ秒まで取得したい場合は、 `org.joda.time.DateTime` を使用する。

```
DateTime dateTime = new DateTime();
```

2. TimeZone と、時間を除いた日付だけが必要な場合は、 `org.joda.time.LocalDate` を使用する。

```
LocalDate localDate = new LocalDate();
```

3. TimeZone と、日付を除いた時間だけが必要な場合は、 `org.joda.time.LocalTime` を使用する。

```
LocalTime localTime = new LocalTime();
```

4. 日付開始時刻と現在日付を取得したい場合は、 `org.joda.time.DateTime.withTimeAtStartOfDay()` を使用する。

```
DateTime dateTimeAtStartOfDay = new DateTime().withTimeAtStartOfDay();
```

注釈: LocalDate と LocalTime は、TimeZone 情報を持たない。

注釈: 実際 Service や Controller で現在時刻を取得するときの DateTime, LocalDate や、LocalTime のインスタンス取得には、`org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory` を利用することを推奨する。

```
DateTime dateTime = dataFactory.newDateTime();
```

DateFactory の利用方法は、 システム時刻 を参照されたい。

LocalDate や LocalTime の生成は

```
LocalDate localDate = dataFactory.newDateTime().toLocalDate();
LocalTime localTime = dataFactory.newDateTime().toLocalTime();
```

とすればよい。

タイムゾーンを指定して現在時刻を取得

`org.joda.time.DateTimeZone` は、timezone を表すクラスである。
Timezone を指定して取得したい場合に使用する。以下に、使用方法を示す。

```
DateTime dateTime = new DateTime(DateTimeZone.forID("Asia/Tokyo"));
```

`org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory` を利用する場合は、以下のようになる。

```
// Fetching current system date using default TimeZone
DateTime dateTime = dataFactory.newDateTime();

// Changing to TimeZone of Tokyo
DateTime dateTimeTokyo = dateTime.withZone(DateTimeZone.forID("Asia/Tokyo"));
```

他の使用可能な Timezone ID 文字列の一覧は、 Available Time Zones を参照されたい。

タイムゾーンを指定せず現在時刻を取得

タイムゾーンを指定せず現在時刻を取得したい場合に使用する。以下に、使用方法を示す。

```
LocalDateTime localDateTime = new LocalDateTime();
```

`org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory` を利用する場合は、以下のようになる。

```
// Fetching current system date using default TimeZone
LocalDateTime localDateTime = dateFactory.newDateTime().toLocalDateTime();
```

> **注釈:** TimeZone を意識する必要がない場合は、DateTime ではなく LocalDateTime を利用することを推奨する。

年月日時分秒を指定して取得

コンストラクタで、特定の時間を指定することができる。以下に例を示す。

- ミリ秒まで指定して、DateTime を取得したい場合

```
DateTime dateTime = new DateTime(year, month, day, hour, minite, second,␣
↪millisecond);
```

- 年月日を指定して、LocalDate を取得したい場合

```
LocalDate localDate = new LocalDate(year, month, day);
```

- 時分秒を指定して、LocalDate 取得したい場合

```
LocalTime localTime = new LocalTime(hour, minutes, seconds, milliseconds);
```

年月日等の個別取得

DateTime では、年、月などを取得するメソッドを用意している。以下に、利用例を示す。

```
DateTime dateTime = new DateTime(2013, 1, 10, 2, 30, 22, 123);

int year = dateTime.getYear();  // (1)
int month = dateTime.getMonthOfYear();  // (2)
```

```
int day = dateTime.getDayOfMonth();  // (3)
int week = dateTime.getDayOfWeek();  // (4)
int hour = dateTime.getHourOfDay();  // (5)
int min = dateTime.getMinuteOfHour();  // (6)
int sec = dateTime.getSecondOfMinute();  // (7)
int millis = dateTime.getMillisOfSecond();  // (8)
```

項番	説明
(1)	年を取得する。本例では、2013 が返却される。
(2)	月を取得する。本例では、」1」 が返却される。
(3)	日を取得する。本例では、10 が返却される。
(4)	曜日を取得する。本例では、」4」 が返却される。 返却される値と曜日の対応は、[1:月曜、2:火曜、3:水曜、4:木曜、5:金曜、6:土曜、7:日曜] となる。
(5)	時を取得する。本例では、」2」 が返却される。
(6)	分を取得する。本例では、30 が返却される。
(7)	秒を取得する。本例では、22 が返却される。
(8)	ミリ秒を取得する。本例では、123 が返却される。

注釈: java.util.Calendar の仕様とは異なり、getDayOfMonth() は、1 始まりである。

型変換

java.util.Date との相互運用性

DateTime では、 java.util.Date との型変換を、容易に行える。

```
Date date = new Date();

DateTime dateTime = new DateTime(date);  // (1)

Date convertDate = dateTime.toDate();  // (2)
```

項番	説明
(1)	DateTime のコンストラクタの引数に、 java.util.Date を引数に渡すことで、 java.util.Date -> DateTime への変換を行う。
(2)	DateTime#toDate メソッドで、DateTime -> java.util.Date への変換を行う。

文字列へのフォーマット

```
DateTime dateTime = new DateTime();

dateTime.toString("yyyy-MM-dd HH:mm:ss");  // (1)
```

項番	説明
(1)	「yyyy-MM-dd HH:mm:ss」 形式で変換された、文字列が取得される。 toString の引数として指定可能な値については、 Input and Output を参照されたい。

文字列からのパース

```
LocalDate localDate = DateTimeFormat.forPattern("yyyy-MM-dd").parseLocalDate("2012-
↪08-09");  // (1)
```

項番	説明
(1)	「yyyy-MM-dd」 形式の文字列を、LocalDate 型に変換する。 DateTimeFormat#forPattern の引数として指定可能な値は、 Formatters を参照されたい。

日付操作

日付の計算

LocalDate には、日付の加減算を行うメソッドが用意されている。以下に、利用例を示す。

```
LocalDate localDate = new LocalDate(); // localDate is 2013-01-10
LocalDate yesterday = localDate.minusDays(1);  // (1)
LocalDate tomorrow = localDate.plusDays(1);  // (2)
LocalDate afterThreeMonth = localDate.plusMonths(3);  // (3)
LocalDate nextYear = localDate.plusYears(1);  // (4)
```

項番	説明
(1)	LocalDate#minusDays 引数に、指定した値分の日付が減算される。本例では `2013-01-09` となる。
(2)	LocalDate#plusDays 引数に、指定した値分の日付が加算される。本例では `2013-01-11` となる。
(3)	LocalDate#plusMonths 引数に、指定した値分の月数が加算される。本例では `2013-04-10` となる。
(4)	LocalDate#plusYears 引数に、指定した値分の年数が加算される。本例では `2014-01-10` となる。

上記で示したメソッド以外は、 LocalDate JavaDoc を参照されたい。

月末月初の取得

現在日時を基準日とした、月末日と月初日の取得方法を、以下に示す。

```
LocalDate localDate = new LocalDate(); // dateTime is 2013-01-10
Property dayOfMonth = localDate.dayOfMonth(); // (1)
LocalDate firstDayOfMonth = dayOfMonth.withMinimumValue(); // (2)
LocalDate lastDayOfMonth = dayOfMonth.withMaximumValue(); // (3)
```

項番	説明
(1)	現在月の日付に関する属性値を保持する Property オブジェクトを取得する。
(2)	Property オブジェクトから最小値を取得する事で、月初日を取得する事ができる。本例では `2013-01-01` となる。
(3)	Property オブジェクトから最大値を取得する事で、月末日を取得する事ができる。本例では `2013-01-31` となる。

週末週初の取得

現在日時を基準日とした、週末日と週初日の取得方法を、以下に示す。

```
LocalDate localDate = new LocalDate(); // dateTime is 2013-01-10
Property dayOfWeek = localDate.dayOfWeek(); // (1)
LocalDate firstDayOfWeek = dayOfWeek.withMinimumValue(); // (2)
LocalDate lastDayOfWeek = dayOfWeek.withMaximumValue(); // (3)
```

項番	説明
(1)	現在週の日付に関する属性値を保持する Property オブジェクトを取得する。
(2)	Property オブジェクトから最小値を取得する事で、週初日 (月曜日) を取得する事ができる。本例では `2013-01-07` となる。
(3)	Property オブジェクトから最大値を取得する事で、週末日 (日曜日) を取得する事ができる。本例では `2013-01-13` となる。

日時の比較

日時を比較して過去か未来を判定できる。

```
DateTime dt1 = new DateTime();
DateTime dt2 = dt1.plusHours(1);
DateTime dt3 = dt1.minusHours(1);


System.out.println(dt1.isAfter(dt1)); // false
System.out.println(dt1.isAfter(dt2)); // false
System.out.println(dt1.isAfter(dt3)); // true

System.out.println(dt1.isBefore(dt1)); // false
System.out.println(dt1.isBefore(dt2)); // true
System.out.println(dt1.isBefore(dt3)); // false

System.out.println(dt1.isEqual(dt1)); // true
System.out.println(dt1.isEqual(dt2)); // false
System.out.println(dt1.isEqual(dt3)); // false
```

項番	説明
(1)	`isAfter` メソッドは対象の日時が引数の日時より未来の場合に `true` を返す。
(2)	`isBefore` メソッドは対象の日時が引数の日時より過去の場合に `true` を返す。
(3)	`isEqual` メソッドは対象の日時が引数の日時と同じ場合に `true` を返す。

期間の取得

Joda-Time では、期間に関して、いくつかのクラスが提供されている。ここでは以下の 2 クラスについて説明する。

- `org.joda.time.Interval`
- `org.joda.time.Period`

Interval

2 つのインスタンス（DateTime）の期間を表すクラス。

Interval で調べられることは、以下 4 つである。

- 期間内に指定の日付や期間が含まれるかのチェック

- 2 つの期間が連続するかのチェック
- 2 つの期間の差を期間で取得
- 2 つの期間の重なった期間を取得

実装例は、以下を参照されたい。

```
DateTime start1 = new DateTime(2013,8,14,0,0,0);
DateTime end1 = new DateTime(2013,8,16,0,0,0);

DateTime start2 = new DateTime(2013,8,16,0,0,0);
DateTime end2 = new DateTime(2013,8,18,0,0,0);

DateTime anyDate = new DateTime(2013, 8, 15, 0, 0, 0);

Interval interval1 = new Interval(start1, end1);
Interval interval2 = new Interval(start2, end2);

interval1.contains(anyDate);  // (1)

interval1.abuts(interval2);  // (2)

DateTime start3 = new DateTime(2013,8,18,0,0,0);
DateTime end3 = new DateTime(2013,8,20,0,0,0);
Interval interval3 = new Interval(start3, end3);

interval1.gap(interval3);  // (3)

DateTime start4 = new DateTime(2013,8,15,0,0,0);
DateTime end4 = new DateTime(2013,8,17,0,0,0);
Interval interval4 = new Interval(start4, end4);

interval1.overlap(interval4);  // (4)
```

項番	説明
(1)	Interval#contains メソッドで、期間内に指定の日付や期間が含まれるかのチェックを行う。 期間内に含まれる場合、」true」、含まれない場合、」false」を返却する。
(2)	Interval#abuts メソッドで、2 つの期間が連続するかのチェックを行う。 2 つの期間が連続する場合は」true」、連続しない場合は」false」を返却する。
(3)	Interval#gap メソッドで、2 つの期間の差を期間 (Interval) で取得する。 本例では、」2013-08-16～2013-08-18」 の期間が取得される。 期間の差が存在しない場合、null が戻り値となる。
(4)	Interval#overlap メソッドで、2 つの期間の重なった期間 (Interval) を取得する。 本例では、」2013-08-15～2013-08-16」 の期間が取得される。 重なった期間が存在しない場合、null が戻り値となる。

Interval 同士を比較したい場合は、Period に変換して行う。

- 月、日、などより抽象的な観点で比較をしたい場合は、Period に変換すること。

```
// Convert to Period
interval1.toPeriod();
```

Period

Period は、期間を、年、月、週などの単位で表すクラスである。

たとえば、「3 月 1 日」を表す Instant（DateTime）に「1 ヶ月」に相当する Period を追加した場合、DateTime は「4 月 1 日」になる。
「3 月 1 日」と「4 月 1 日」に対して、「1 か月」に相当する Period を追加した時の結果を以下に示す。

- 「3 月 1 日」に「1 ヶ月」という Period を追加したときの日数は「31 日」
- 「4 月 1 日」に「1 ヶ月」という Period を追加したときの日数は「30 日」

「1 ヶ月」に相当する Period の追加は、対象の DateTime によって、違う意味を持つ。

Period は、さらに 2 種類の実装が用意されている。

- Single field Period (例：「1 日」や「1 ヶ月」など一つの単位の値しか持たないタイプ)
- Any field Period (例：「1 ヶ月 2 日 4 時間」など、複数の単位の値を持てて期間を表すタイプ)

詳細は、 Period を参照されたい。

JSP Tag Library

JSTL の fmt:formatDate タグは、java.util.Date と、java.util.TimeZone オブジェクトを扱う。

Joda-time の DateTime, LocalDateTime, LocalDate, LocalTime と、DateTimeZone オブジェクトを扱うためには、Joda のタグライブラリを使う。

機能面で JSTL とほぼ同じであるため、JSTL の知識がある場合は、Joda の JSP タグライブラリを容易に使える。

設定方法

タブライブラリを利用するには、以下の taglib 定義が必要である。

```
<%@ taglib uri="http://www.joda.org/joda/time/tags" prefix="joda"%>
```

joda:format タグ

joda:format タグとは、DateTime, LocalDateTime, LocalDate, LocalTime オブジェクトをフォーマットするタグである。

```
<% pageContext.setAttribute("now", new org.joda.time.DateTime()); %>

<span>Using pattern="yyyyMMdd" to format the current system date</span><br/>
<joda:format value="${now}" pattern="yyyyMMdd" />
<br/>
```

```
<span>Using style="SM" to format the current system date</span><br/>
<joda:format value="${now}" style="SM" />
```

出力結果

Using pattern="yyyyMMdd" to format the current system date
20131025
Using style="SM" to format the current system date
10/25/13 1:02:32 PM

joda:format タグの属性一覧は、以下の通りである。

表 7.6　属性情報

No.	Attributes	Description
1.	value	ReadableInstant か ReadablePartial のインスタンスを設定する。
2.	var	時刻情報を持つ変数名
3.	scope	時刻情報を持つ変数名のスコープ
4.	locale	ロケール情報
5.	style	フォーマットするためのスタイル情報（2 桁。日付部分と時刻部分それぞれのスタイルを設定する。入力可能な値は S=Short, M=Medium, L=Long, F=Full, -=None）
6.	pattern	フォーマットするためのパターン（yyyyMMdd など）。入力可能なパターンは、 Input and Output を参照されたい。
7.	dateTimeZone	タイムゾーン

Joda-Time のほかのタグは、 Joda Time JSP tags User guide を参照されたい。

注釈: style 属性を指定して日付と時刻部分を表示する場合、ブラウザの locale によって表示内容が異なる。上記 style 属性で表示した形式の locale は」en」である。

応用例 (カレンダーの表示)

Spring MVC を使って、月単位のカレンダーを表示するサンプルを示す。

処理名	URL	ハンドラメソッド
今月のカレンダー表示	/calendar	today
指定月のカレンダー表示	/calendar/month?year=yyyy&month=m	month

コントローラの実装は、以下のようになる。

```
@Controller
@RequestMapping("calendar")
public class CalendarController {

    @RequestMapping
    public String today(Model model) {
        LocalDate today = new LocalDate();
        int year = today.getYear();
        int month = today.getMonthOfYear();
        return month(year, month, model);
    }

    @RequestMapping(value = "month")
    public String month(@RequestParam("year") int year,
            @RequestParam("month") int month, Model model) {
        LocalDate firstDayOfMonth = new LocalDate(year, month, 1);
        LocalDate lastDayOfMonth = firstDayOfMonth.dayOfMonth()
                .withMaximumValue();

        LocalDate firstDayOfCalendar = firstDayOfMonth.dayOfWeek()
                .withMinimumValue();
        LocalDate lastDayOfCalendar = lastDayOfMonth.dayOfWeek()
                .withMaximumValue();

        List<List<LocalDate>> calendar = new ArrayList<List<LocalDate>>();
        List<LocalDate> weekList = null;
        for (int i = 0; i < 100; i++) {
            LocalDate d = firstDayOfCalendar.plusDays(i);
            if (d.isAfter(lastDayOfCalendar)) {
                break;
            }
```

```
            if (weekList == null) {
                weekList = new ArrayList<LocalDate>();
                calendar.add(weekList);
            }

            if (d.isBefore(firstDayOfMonth) || d.isAfter(lastDayOfMonth)) {
                // skip if the day is not in this month
                weekList.add(null);
            } else {
                weekList.add(d);
            }

            int week = d.getDayOfWeek();
            if (week == DateTimeConstants.SUNDAY) {
                weekList = null;
            }
        }

        LocalDate nextMonth = firstDayOfMonth.plusMonths(1);
        LocalDate prevMonth = firstDayOfMonth.minusMonths(1);
        CalendarOutput output = new CalendarOutput();
        output.setCalendar(calendar);
        output.setFirstDayOfMonth(firstDayOfMonth);
        output.setYearOfNextMonth(nextMonth.getYear());
        output.setMonthOfNextMonth(nextMonth.getMonthOfYear());
        output.setYearOfPrevMonth(prevMonth.getYear());
        output.setMonthOfPrevMonth(prevMonth.getMonthOfYear());

        model.addAttribute("output", output);

        return "calendar";
    }
}
```

以下の CalendarOutput クラスは、画面に出力する情報をまとめた JavaBean である。

```
public class CalendarOutput {
    private List<List<LocalDate>> calendar;

    private LocalDate firstDayOfMonth;

    private int yearOfNextMonth;

    private int monthOfNextMonth;

    private int yearOfPrevMonth;

    private int monthOfPrevMonth;

    // omitted getter/setter
}
```

> **警告:** このサンプルコードは単純なため Controller のハンドラメソッドに全ての処理を記述しているが、メンテナンス性向上のため本来この処理は、Helper クラスに記述すべきである。

JSP(calendar.jsp) で、次のように出力する。

```
<p>
    <a
        href="${pageContext.request.contextPath}/calendar/month?year=$
↪{f:h(output.yearOfPrevMonth)}&month=${f:h(output.monthOfPrevMonth)}">&larr;
        Prev</a> <a
        href="${pageContext.request.contextPath}/calendar/month?year=$
↪{f:h(output.yearOfNextMonth)}&month=${f:h(output.monthOfNextMonth)}">Next
        &rarr;</a> <br>
    <joda:format value="${output.firstDayOfMonth}"
        pattern="yyyy-M" />
</p>
<table>
    <tr>
        <th>Mon.</th>
        <th>Tue.</th>
        <th>Wed.</th>
        <th>Thu.</th>
        <th>Fri.</th>
        <th>Sat.</th>
        <th>Sun.</th>
    </tr>
    <c:forEach var="week" items="${output.calendar}">
        <tr>
            <c:forEach var="day" items="${week}">
                <td><c:choose>
                        <c:when test="${day != null}">
                            <joda:format value="${day}"
                                pattern="d" />
                        </c:when>
                        <c:otherwise> </c:otherwise>
                    </c:choose></td>
            </c:forEach>
        </tr>
    </c:forEach>
</table>
```

{contextPath}/calendar にアクセスすると、以下のカレンダーが表示される（2012 年 11 月時点での結果で

ある)。



{contextPath}/calendar/month?year=2012&month=12 にアクセスすると、以下のカレンダーが表示される。



7.4.3 Appendix

Java8 未満の和暦操作

Java8 では java.time.chrono.JapaneseDate という和暦操作クラスが提供されているが、Java8 未満の環境では java.util.Calendar クラスで和暦を扱うことが出来る。

具体的には、 java.util.Calendar クラス、 java.text.DateFormat クラスに以下の java.util.Locale を指定する必要がある。

```
Locale locale = new Locale("ja", "JP", "JP");
```

以下に、Calendar クラスを利用した和暦表示の例を示す。

```
Locale locale = new Locale("ja", "JP", "JP");
Calendar cal = Calendar.getInstance(locale); // Ex, 2015-06-05
String format1 = "Gy.MM.dd";
String format2 = "GGGGyy/MM/dd";

DateFormat df1 = new SimpleDateFormat(format1, locale);
DateFormat df2 = new SimpleDateFormat(format2, locale);

df1.format(cal.getTime()); // "H27.06.05"
df2.format(cal.getTime()); // "平成 27/06/05"
```

また、同様に文字列からのパースも行うことが出来る。

```
Locale locale = new Locale("ja", "JP", "JP");
String format1 = "Gy.MM.dd";
String format2 = "GGGGyy/MM/dd";

DateFormat df1 = new SimpleDateFormat(format1, locale);
DateFormat df2 = new SimpleDateFormat(format2, locale);

Calendar cal1 = Calendar.getInstance(locale);
Calendar cal2 = Calendar.getInstance(locale);

cal1.setTime(df1.parse("H27.06.05"));
cal2.setTime(df2.parse("平成 27/06/05"));
```

注釈:

new Locale("ja", "JP", "JP") を getInstance メソッドに指定することで、 和暦に対応した java.util.JapaneseImperialCalendar オブジェクトが作成される。
その他を指定すると java.util.GregorianCalendar オブジェクトが作成されるため、留意されたい。

7.5 システム時刻

7.5.1 Overview

アプリケーション開発中は、サーバーのシステム時刻ではなく、任意の日時でテストを行う必要が生じる。Production 環境においても日付を戻してリカバリ処理を行うことも想定される。

そのため、日時の取得ではサーバーのシステム時刻ではなく、開発・運用側で任意の日時に設定可能になっていることが望ましい。

共通ライブラリから提供しているコンポーネントについて

共通ライブラリでは、システム時刻を取得するためのコンポーネント (以降では、これらの API を総称して「Date Factory」と呼ぶ) を提供している。

共通ライブラリから提供しているコンポーネントは、terasoluna-gfw-common と terasoluna-gfw-jodatime の二つのアーティファクトに分かれており、

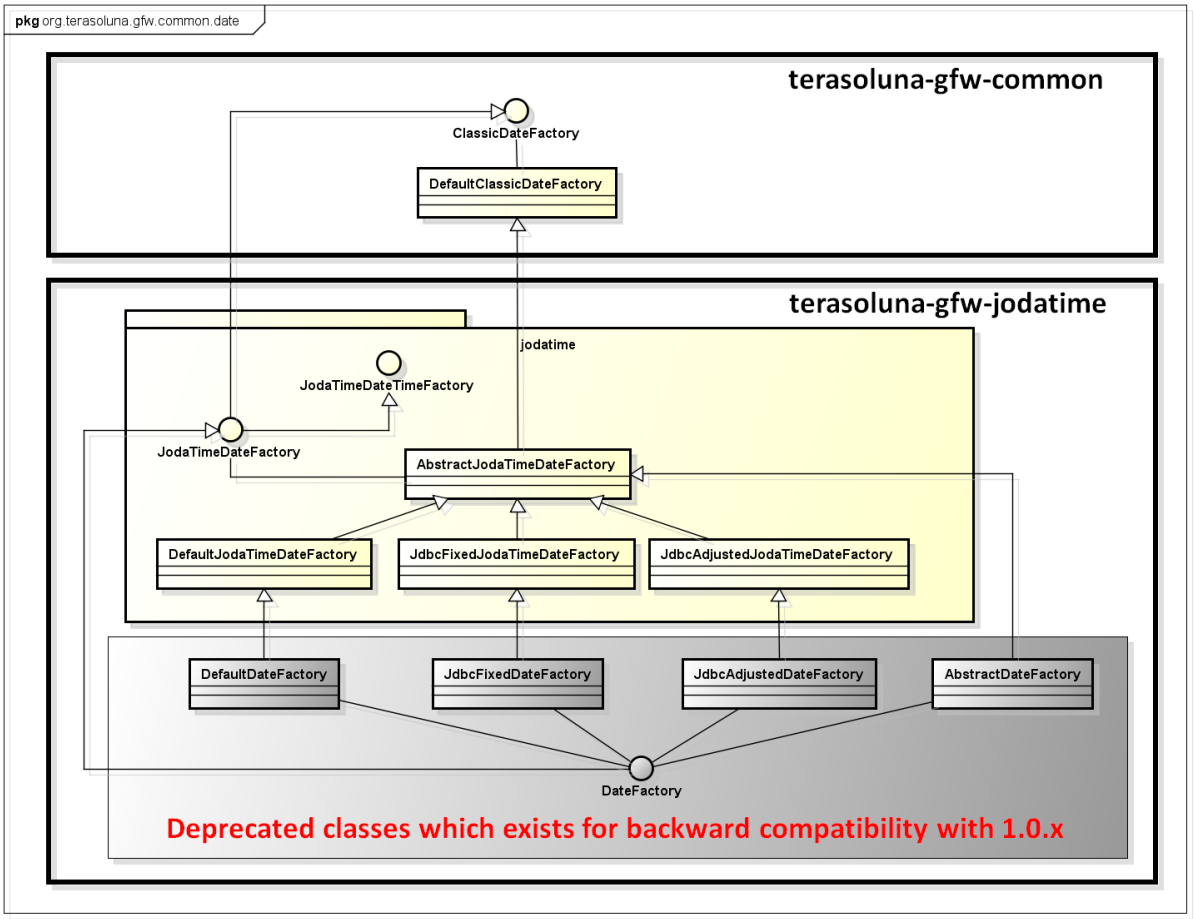
- terasoluna-gfw-common は、Java 標準の API のみを利用する Date Factory
- terasoluna-gfw-jodatime は、Joda Time の API を利用する Date Factory

を提供している。

共通ライブラリから提供しているコンポーネントのクラス図を以下に示す。

terasoluna-gfw-common

以下に、terasoluna-gfw-common のコンポーネントとして提供しているインタフェースについて説明する。



インタフェース	説明
org.terasoluna.gfw.common.date. ClassicDateFactory	Java から提供されている以下のクラスのインスタンスをシステム時刻として取得するためのインタフェース。 • `java.util.Date` • `java.sql.Timestamp` • `java.sql.Date` • `java.sql.Time` 共通ライブラリでは、本インタフェースの実装クラスとして以下のクラスを提供している。 • `org.terasoluna.gfw.common.date.DefaultClassicDateFactory`

terasoluna-gfw-jodatime

以下に、terasoluna-gfw-jodatime のコンポーネントとして提供しているインタフェースについて説明する。

インタフェース	説明
org.terasoluna.gfw.common.date.jodatime.JodaTimeDateTimeFactory	Joda Time から提供されている以下のクラスのインスタンスをシステム時刻として取得するためのインタフェース。 • `org.joda.time.DateTime`
org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory	`ClassicDateFactory` と `JodaTimeDateTimeFactory` を継承したインタフェース。 共通ライブラリでは、本インタフェースの実装クラスとして以下のクラスを提供している。 • `org.terasoluna.gfw.common.date.jodatime.DefaultJodaTimeDateFactory` • `org.terasoluna.gfw.common.date.jodatime.JdbcFixedJodaTimeDateFactory` • `org.terasoluna.gfw.common.date.jodatime.JdbcAdjustedJodaTimeDateFactory` **本ガイドラインでは、本インタフェースに対応する実装クラスを使用することを推奨する。**
org.terasoluna.gfw.common.date.DateFactory	`JodaTimeDateFactory` を継承したインタフェース (非推奨)。 本インタフェースは、terasoluna-gfw-common 1.0.x で提供している `DateFactory` との後方互換のために提供しているインタフェースである。 共通ライブラリでは、本インタフェースの実装クラスとして以下のクラスを提供している。 • `org.terasoluna.gfw.common.date.DefaultDateFactory`(非推奨) • `org.terasoluna.gfw.common.date.JdbcFixedDateFactory`(非推奨) • `org.terasoluna.gfw.common.date.JdbcAdjustedDateFactory`(非推奨) **本インタフェース及び対応する実装クラスは非推奨の API であるため、新規に開発するアプリケーションで使用する事を禁止する。**

注釈: Joda Time については、 日付操作 *(Joda Time)* を参照されたい。

7.5.2 How to use

Date Factory インタフェースの実装クラスを bean 定義ファイルに定義し、Date Factory のインスタンスを Java クラスにインジェクションして使用する。

実装クラスは使用用途に応じて、以下から選択する。

クラス名	概要	備考
org.terasoluna.gfw.common.date.jodatime.DefaultJodaTimeDateFactory	アプリケーションサーバーのシステム時刻を返却する。	`new DateTime()` での取得値と同等であり、時刻の変更はできない。
org.terasoluna.gfw.common.date.jodatime.JdbcFixedJodaTimeDateFactory	DB に登録した固定の時刻を返却する。	完全に時刻を固定する必要のある Integration Test 環境で使用されることを想定しており、Performance Test 環境や、Production 環境では使用しない。 このクラスを使用するためには、固定時刻を管理するためのテーブルが必要である。
org.terasoluna.gfw.common.date.jodatime.JdbcAdjustedJodaTimeDateFactory	アプリケーションサーバーのシステム時刻に DB に登録した差分(ミリ秒) を加算した時刻を返却する。	Integration Test 環境や System Test 環境で使用されることを想定しているが、差分値を 0 に設定することで Production 環境でも使用できる。 このクラスを使用するためには、差分値を管理するためのテーブルが必要である。

注釈: 実装クラスを設定する bean 定義ファイルは、環境ごとに切り替えられるように、[projectName]-env.xml に定義することを推奨する。Date Factory を利用することにより、bean 定義ファイルの設定を変更するだけで、ソースを変更せずに日時の変更が可能となる。bean 定義ファイルの記載例は後述する。

ちなみに: JUnit などで日時を変更して試験を行いたい場合、インタフェースの実装クラスを mock クラスに差し替えることで、任意の日時を設定することも可能である。差し替え方法については、「*Unit Test*」を参照されたい。

pom.xml の設定

terasoluna-gfw-jodatime への依存関係を追加する。

マルチプロジェクト構成の場合は、domain プロジェクトの pom.xml(projectName-domain/pom.xml) に追加する。

ブランクプロジェクト からプロジェクトを生成した場合は、terasoluna-gfw-jodatime への依存関係は、設定済みの状態である。

```
<dependencies>

    <!-- (1) -->
    <dependency>
        <groupId>org.terasoluna.gfw</groupId>
        <artifactId>terasoluna-gfw-jodatime-dependencies</artifactId>
        <type>pom</type>
    </dependency>

</dependencies>
```

項番	説明
1.	terasoluna-gfw-jodatime-dependencies を dependencies に追加する。terasoluna-gfw-jodatime-dependencies には、Joda Time 用の Date Factory と Joda Time 関連のライブラリへの依存関係が定義されている。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。

サーバーのシステム時刻を返却する

org.terasoluna.gfw.common.date.jodatime.DefaultJodaTimeDateFactory を使用する。

bean 定義ファイル ([projectname]-env.xml)

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.jodatime.
↪DefaultJodaTimeDateFactory" />  <!-- (1) -->
```

項番	説明
(1)	DefaultJodaTimeDateFactory クラスを bean 定義する。

Java クラス

```
@Inject
JodaTimeDateFactory dateFactory;  // (2)

public TourInfoSearchCriteria setUpTourInfoSearchCriteria() {

    DateTime dateTime = dateFactory.newDateTime();  // (3)

    // omitted
}
```

項番	説明
(2)	Date Factory を利用するクラスにインジェクションする。
(3)	利用したい日付のクラスインスタンスを返却するメソッドを呼び出す。 上記例では、`org.joda.time.DateTime` 型のインスタンスを取得している。

DB から取得した固定の時刻を返却する

`org.terasoluna.gfw.common.date.jodatime.JdbcFixedJodaTimeDateFactory` を使用する。

bean 定義ファイル

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.jodatime.
↪JdbcFixedJodaTimeDateFactory" >  <!-- (1) -->
    <property name="dataSource" ref="dataSource" />  <!-- (2) -->
    <property name="currentTimestampQuery" value="SELECT now FROM system_date" />
↪<!-- (3) -->
</bean>
```

項番	説明
(1)	JdbcFixedJodaTimeDateFactory を bean 定義する。
(2)	dataSource プロパティに、固定時刻を管理するためのテーブルが存在するデータソース (javax.sql.DataSource) を指定する。
(3)	currentTimestampQuery プロパティに、固定時刻を取得するための SQL を設定する。

テーブル設定例

以下のようにテーブルを作成し、レコードを追加する必要がある。

```
CREATE TABLE system_date(now timestamp NOT NULL);
INSERT INTO system_date(now) VALUES (current_date);
```

レコード番号	now
1	2013-01-01 01:01:01.000

Java クラス

```
@Inject
JodaTimeDateFactory dateFactory;

@RequestMapping(value="datetime", method = RequestMethod.GET)
public String listConfirm(Model model) {

    for (int i=0; i < 3; i++) {
        model.addAttribute("jdbcFixedDateFactory" + i, dateFactory.newDateTime());
↪// (4)
        model.addAttribute("DateTime" + i, new DateTime()); // (5)
    }
```

```
    return "date/dateTimeDisplay";
}
```

項番	説明
(4)	Date Factory から取得したシステム時刻を画面に渡す。 実行結果を確認すると、DB に設定した固定の値が出力されている事がわかる。
(5)	確認用に `new DateTime()` の結果を画面に渡す。 実行結果を確認すると、毎回異なる値 (アプリケーションサーバのシステム時刻) が出力されている事がわかる。

実行結果



SQL ログ

```
16. SELECT now FROM system_date {executed in 0 msec}
17. SELECT now FROM system_date {executed in 1 msec}
18. SELECT now FROM system_date {executed in 0 msec}
```

Date Factory のメソッドを呼び出すと、DB へのアクセスログが出力される。SQL ログを出力するために、データベースアクセス（共通編） で説明した `Log4jdbcProxyDataSource` を使用している。

サーバーのシステム時刻に DB に登録した差分値を加算した時刻を返却する

`org.terasoluna.gfw.common.date.jodatime.JdbcAdjustedJodaTimeDateFactory` を使用する。

bean 定義ファイル

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.jodatime.
↪JdbcAdjustedJodaTimeDateFactory" > <!-- (1) -->
  <property name="dataSource" ref="dataSource" /> <!-- (2) -->
  <property name="adjustedValueQuery" value="SELECT diff * 60 * 1000 FROM␣
↪operation_date" /> <!-- (3) -->
</bean>
```

項番	説明
(1)	`JdbcAdjustedJodaTimeDateFactory` を bean 定義する。
(2)	`dataSource` プロパティに、差分値を管理するためのテーブルが存在するデータソース (`javax.sql.DataSource`) を指定する。
(3)	`adjustedValueQuery` プロパティに、差分値を取得するための SQL を設定する。 上記 SQL は、差分値の単位を」minutes」にする場合の SQL である。

テーブル設定例

以下のようにテーブルを作成し、レコードを追加する必要がある。

```
CREATE TABLE operation_date(diff bigint NOT NULL);
INSERT INTO operation_date(diff) VALUES (-1440);
```

レコード番号	diff
1	-1440

本例では、差分値の単位を」minutes」としている。(DB のデータは-1440 分=1 日前を指定)
取得結果をミリ秒（整数値）に変換することで、DB 上の値の単位は、日・時・分・秒・ミリ秒のいずれでも問題ない。

注釈: 上記の SQL は PostgreSQL 用である。Oracle の場合は `BIGINT` の代わりに `NUMBER(19)` を使用すればよい。

ちなみに: 差分値の単位を」minutes」以外にしたい場合は、以下のような SQL を `adjustedValueQuery` プロパティに指定すればよい。

差分値の単位	SQL
milliseconds	SELECT diff FROM operation_date
seconds	SELECT diff * 1000 FROM operation_date
hours	SELECT diff * 60 * 60 * 1000 FROM operation_date
days	SELECT diff * 24 * 60 * 60 * 1000 FROM operation_date

Java クラス

```
@Inject
JodaTimeDateFactory dateFactory;

@RequestMapping(value="datetime", method = RequestMethod.GET)
public String listConfirm(Model model) {

    model.addAttribute("firstExpectedDate", new DateTime());  // (4)
    model.addAttribute("serverTime", dateFactory.newDateTime());  // (5)
    model.addAttribute("lastExpectedDate", new DateTime());  // (6)

    return "date/dateTimeDisplay";
}
```

項番	説明
(4)	確認用に、Date Factory のメソッドを呼び出す前の時刻を画面に渡す。
(5)	Date Factory から取得したシステム時刻を画面に渡す。 実行結果を確認すると、実行時から 1440 分を引いた時刻が出力されている事がわかる。
(6)	確認用に、Date Factory のメソッドを呼び出した後の時刻を画面に渡す。

実行結果



SQL ログ

```
17. SELECT diff * 60 * 1000 FROM operation_date {executed in 1 msec}
```

Date Factory のメソッドを呼び出すと、DB へのアクセスログが出力される。

差分のキャッシュとリロード方法

差分値を 0 にして、本番環境で利用する場合に、差分を毎回 DB から取得するのは性能が悪い。そこで、JdbcAdjustedJodaTimeDateFactory では、SQL を発行して取得した差分値をキャッシュすることを可能にしている。起動時に取得した値をキャッシュした後、リクエスト毎のテーブルアクセスは行わない。

bean 定義ファイル

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.jodatime.
↪JdbcAdjustedJodaTimeDateFactory" >
  <property name="dataSource" ref="dataSource" />
  <property name="adjustedValueQuery" value="SELECT diff * 60 * 1000 FROM␣
↪operation_date" />
  <property name="useCache" value="true" /> <!-- (1) -->
</bean>
```

項番	説明
(1)	true の場合、テーブルから取得した差分値をキャッシュする。デフォルトは false でキャッシュは行わない。 false の場合は Date Factory のメソッド呼び出し時に毎回 SQL を実行する。

キャッシュの設定をしたうえで差分値を変更したい場合は、テーブルの値を変更後、JdbcAdjustedJodaTimeDateFactory.reload() メソッドを実行することで、キャッシュする値を再読み込みすることができる。

Java クラス

```
@Controller
@RequestMapping(value = "reload")
public class ReloadAdjustedValueController {

    @Inject
    JdbcAdjustedJodaTimeDateFactory dateFactory;

    // omitted

    @RequestMapping(method = RequestMethod.GET)
    public String reload() {

        long adjustedValue = dateFactory.reload(); // (2)

        // omitted
```

```
    }
}
```

項番	説明
(2)	`JdbcAdjustedJodaTimeDateFactory` の `reload` メソッドを実行することで、テーブルから差分を読み直す。

7.5.3 Testing

テストを実施する際には、現在日時ではなく別の日時に変更することが必要になる場合がある。

環境	使用する Date Factory	試験内容
Unit Test	DefaultJodaTimeDateFactory	日付に関わる試験は Date Factory を mock 化する。
Integration Test	DefaultJodaTimeDateFactory	日付に関わらない試験
	JdbcFixedJodaTimeDateFactory	特定の日付、時刻に固定して試験を実施する場合
	JdbcAdjustedJodaTimeDateFactory	外部システムとの連携があり、1 日の試験の中で日付の流れを考慮して複数日の試験を実施する場合
System Test	JdbcAdjustedJodaTimeDateFactory	試験の日付を指定して実施する場合や、未来の日付における試験を実施する場合
Production	DefaultJodaTimeDateFactory	実際の時刻と変更する可能性が無い場合
	JdbcAdjustedJodaTimeDateFactory	時刻を変更する可能性を運用上残しておきたい場合。 通常時は差を **0** とし、必要な際のみ差を与える。必ず、 *useCache* を **true** に設定すること

Unit Test

Unit Test では、時刻を登録してその時刻が想定通りに更新されたのかを検証したい場合がある。

そのような場合、処理中にサーバー時刻をそのまま登録してしまうと、テスト実行のたびに値が異なるため、JUnit での回帰試験が難しくなる。そこで、Date Factory を用いることで、登録する時刻を任意の値に固定化することができる。

ミリ秒単位で時刻が一致するようにするため、mock を使用する。Date Factory に値を設定し、固定日付を返却する例を下記に示す。本例では、mock に mockito を使用する。

Java クラス

```
import org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory;

// omitted

@Inject
StaffRepository staffRepository;

@Inject
JodaTimeDateFactory dateFactory;

@Override
public Staff staffUpdateTel(String staffId, String tel) {

    // ex staffId=0001
    Staff staff = staffRepository.findOne(staffId);

    // ex tel = "0123456789"
    staff.setTel(tel);

    // set ChangeMillis
    staff.setChangeMillis(dateFactory.newDateTime()); // (1)

    staffRepository.save(staff);

    return staff;
}

// omitted
```

JUnit ソース

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import static org.mockito.Mockito.*;

import org.joda.time.DateTime;
import org.junit.Before;
import org.junit.Test;
import org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory;

public class StaffServiceTest {

    StaffService service;

    StaffRepository repository;

    JodaTimeDateFactory dateFactory;

    DateTime now;

    @Before
```

```
    public void setUp() {
        service = new StaffService();
        dateFactory = mock(JodaTimeDateFactory.class);
        repository = mock(StaffRepository.class);
        now = new DateTime();
        service.dateFactory = dateFactory;
        service.staffRepository = repository;
        when(dateFactory.newDateTime()).thenReturn(now); // (2)
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void testStaffUpdateTel() {

        Staff setDataStaff = new Staff();
        when(repository.findOne("0001")).thenReturn(setDataStaff);

        // execute
        Staff staff = service.staffUpdateTel("0001", "0123456789");

        //assert
        assertThat(staff.getChangeMillis(), is(now)); // (3)

    }
}
```

項番	説明
(1)	(2) の mock で指定した値が取得され設定される。
(2)	mock で日時を Data Factory の戻り値に設定。
(3)	設定した固定値と同じになるため、 **success** を返す。

日付によって処理が変わる場合の例

「予約したツアーは出発日の 7 日前を過ぎるとキャンセル出来ない」という仕様を実装した Service クラスを例に用いて説明する。

Java クラス

```
import org.terasoluna.gfw.common.date.jodatime.JodaTimeDateFactory;

// omitted

@Inject
JodaTimeDateFactory dateFactory;

// omitted

@Override
public void cancel(String reserveNo) throws BusinessException {
    // omitted

    LocalDate today = dateFactory.newDateTime().toLocalDate(); // (1)
    LocalDate cancelLimit = tourInfo.getDepDay().minusDays(7); // (2)

    if (today.isAfter(cancelLimit)) { // (3)
        // omitted (4)
    }

    // omitted
}
```

項番	説明
(1)	現在日時を取得する。`LocalDate` については 日付操作 *(Joda Time)* を参照されたい。
(2)	対象のツアーのキャンセル期限日を計算する。
(3)	今日がキャンセル期限日より後であるか判定する。
(4)	キャンセル期限日を過ぎた場合は `BusinessException` をスローする。

JUnit ソース

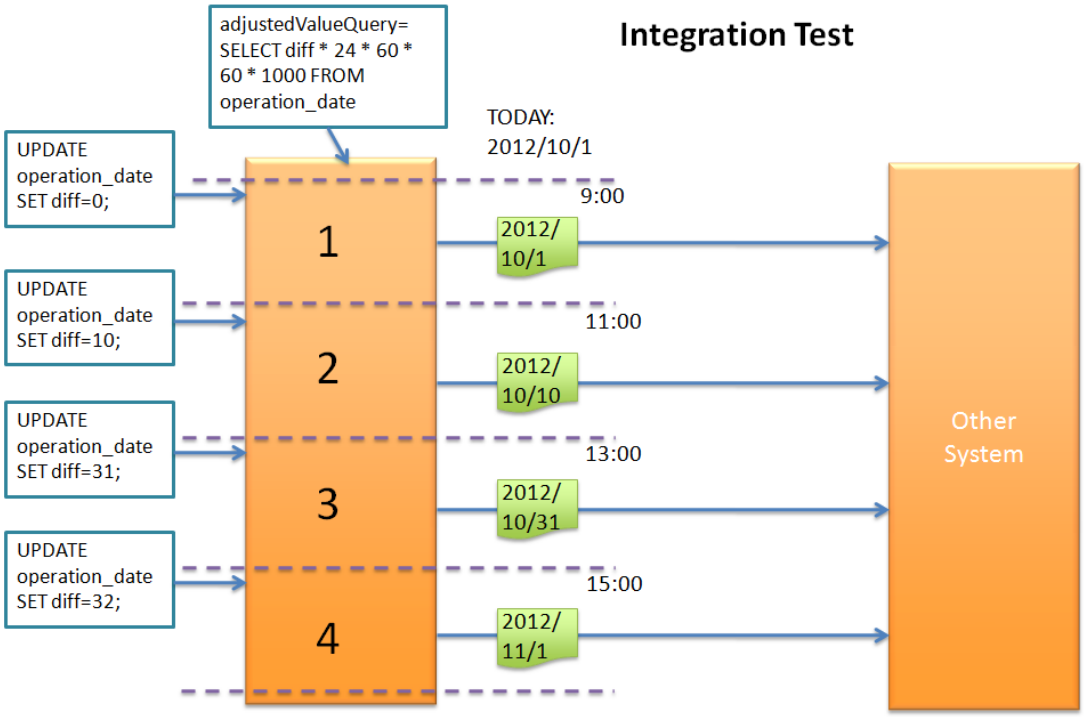
```
@Before
public void setUp() {
    service = new ReserveServiceImpl();

    // omitted

    Reserve reserveResult = new Reserve();
    reserveResult.setDepDay(new LocalDate(2012, 10, 10)); // (5)
    when(reserveRepository.findOne((String) anyObject())).thenReturn(
            reserveResult);
    dateFactory = mock(JodaTimeDateFactory.class);
    service.dateFactory = dateFactory;
}

@Test
public void testCancel01() {

  // omitted

  now = new DateTime(2012, 10, 1, 0, 0, 0, 0);
  when(dateFactory.newDateTime()).thenReturn(now); // (6)

  // run
  service.cancel(reserveNo); // (7)

  // omitted
}

@Test(expected = BusinessException.class)
public void testCancel02() {

  // omitted

  now = new DateTime(2012, 10, 9, 0, 0, 0, 0);
  when(dateFactory.newDateTime()).thenReturn(now); // (8)

  try {
      // run
      service.cancel(reserveNo); // (9)
      fail("Illegal Route");
  } catch (BusinessException e) {
      // assert message if required
      throw e;
  }
}
```

項番	説明
(5)	Repository クラスからの取得するツアー予約情報の出発日を 2012/10/10 とする。
(6)	dateFactory.newDateTime() の返り値を 2012/10/1 とする。
(7)	cancel を実行し、キャンセル可能な日付より前なので、キャンセルが成功する。
(8)	dateFactory.newDateTime() の返り値を 2012/10/9 とする。
(9)	cancel 実行し、キャンセル可能な日付より後なので、キャンセルが失敗する。

Integration Test

Integration Test では、システム連携先と疎通・連携確認のために 1 日の間に何日分ものデータ（例えばファイル）を作成して受け渡しを行う場合がある。

実際の日付が 2012/10/1 の場合、`JdbcAdjustedJodaTimeDateFactory` を使用し、試験対象の日付との差分を計算する SQL を設定する。

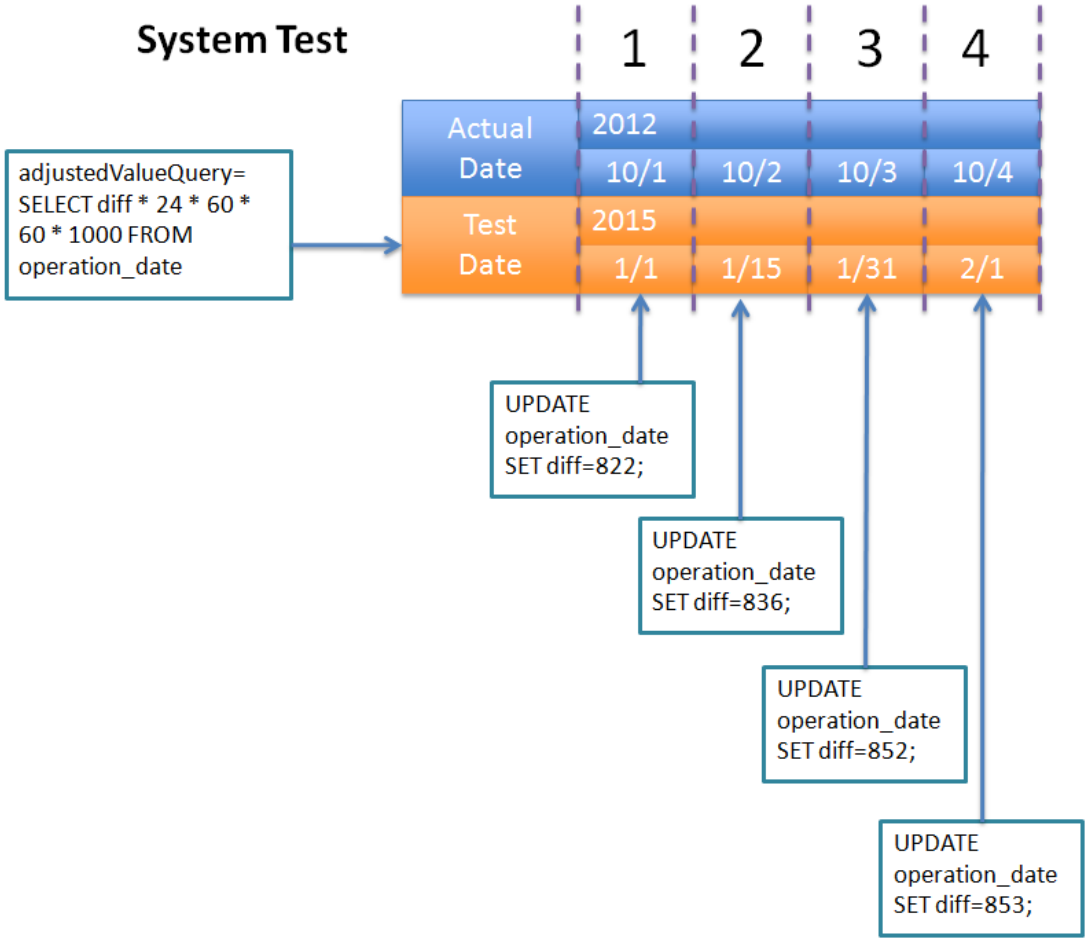


項番	説明
1	9:00-11:00 の間は差分値を」0 days」とし、Date Factory の返り値を 2012/10/1 とする。
2	11:00-13:00 の間は差分値を」9 days」とし、Date Factory の返り値を 2012/10/10 とする。
3	13:00-15:00 の間は差分値を」30 days」とし、Date Factory の返り値を 2012/10/31 とする。
4	15:00-17:00 の間は差分値を」31 days」とし、Date Factory の返り値を 2012/11/1 とする。

テーブルの値を変更するのみで、日付を変更することが可能である。

System Test

System Test では運用日を想定してテストシナリオを作成し、試験を実施することがある。



`JdbcAdjustedJodaTimeDateFactory` を使用し、日付差を計算する SQL を設定する。図中の 1,2,3,4 のように実際の日付と運用日の対応表を作成する。テーブルの差分値を変更するのみで、思い通りの日付でテストすることが可能となる。

Production

`JdbcAdjustedJodaTimeDateFactory` を使用し、差分値を 0 とすることで、ソースを変更せず Date Factory の返り値を、実際の日付と同じにできる。bean 定義ファイルも System Test の時から変更を必要としない。また、日時を変更する必要が生じてもテーブルの値を変更することで、Date Factory の返り値を変更できる。

警告: Production 環境で使用する場合は、production 環境で使用するテーブルの差分値が 0 となっていることを確認すること。

設定例

- production 環境で初めてテーブルを使用する場合

- INSERT INTO operation_date (diff) VALUES (0);

- production 環境で試験実施済みの場合
  - UPDATE operation_date SET diff=0;

を実行すること。

必ず、 *useCache* を **true** に設定すること

時間を変更することがない場合は、`DefaultJodaTimeDateFactory` に設定ファイルを変更することを推奨する。

7.6 文字列処理

7.6.1 Overview

Java の文字列標準 API では、日本語に特化した操作が少ない。
全角カタカナ/半角カタカナの変換や、半角カタカナのみで構成される文字列の判定を行う場合などは、
独自に処理を作りこむ必要がある。

また、Java では全ての文字列を Unicode で表現するが、
Unicode では 𠮷 のような特殊文字は、サロゲートペアと呼ばれる char 型 2 つ（32 ビット）で表される。
このような文字を扱う場合にも予期せぬ挙動が起きぬよう、様々な文字を扱うことを考慮した実装を行う必要がある。

本ガイドラインでは、日本語を処理するケースを想定し、
一般的な文字列操作の例と、共通ライブラリによる日本語操作 API の提供を行う。

7.6.2 How to use

トリム

半角空白のトリム操作を行う場合、`String#trim` メソッドを利用することも出来るが、前・後ろのみのトリム操作や、任意の文字列のトリム操作などの複雑な操作を行う場合は、Spring から提供されている `org.springframework.util.StringUtils` を利用するとよい。

以下に例を示す。

```
String str = "  Hello World!!";

StringUtils.trimWhitespace(str); // => "Hello World!!"

StringUtils.trimLeadingCharacter(str, ' '); // => "Hello World!!"

StringUtils.trimTrailingCharacter(str, '!'); // => "  Hello World"
```

注釈:

`StringUtils#trimLeadingCharacter` , `StringUtils#trimTrailingCharacter` の第 1 引数

にサロゲートペアの文字列は指定しても挙動に変化はない。なお、第 2 引数は char 型のため、サロゲートペアを指定することは出来ない。

パディング・サプレス

パディング（文字列埋め）操作・サプレス（文字列取り）操作を行う場合は、
`String` クラスから提供されているメソッドで行うことが出来る。

以下に例を示す。

```
int num = 1;

String paddingStr = String.format("%03d", num); // => "001"
String suppressStr = paddingStr.replaceFirst("^0+(?!$)", ""); // => "1"
```

> **警告:**
>
> `String#format` はサロゲートペアを考慮できないため、見た目上の長さでパディングを行いたい場合、サロゲートペアが含まれると正しい結果が得られない。
> サロゲートペアを考慮してパディングを実現するためには、後述するサロゲートペアを考慮した文字数のカウントを行い、パディングすべき正しい文字数を算出して文字列結合を行う必要がある。

サロゲートペアを考慮した文字列処理

文字列長の取得

サロゲートペアを考慮した文字列の長さを取得する場合、
単に `String#length` メソッドを使用することは出来ない。
サロゲートペアは 32 ビット（char 型 2 つ）で表現されるため、見た目上の文字数よりも多くカウントされてしまう。

下記例では、変数 `len` には 5 が代入される。

```
String str = "𠮷田太郎";
int len = str.length(); // => 5
```

そこで、Java SE 5 よりサロゲートペアを考慮した文字列の長さを取得するためのメソッド
`String#codePointCount` が定義された。

String#codePointCount の引数に、対象文字列の開始インデックスと終了インデックスを指定することで、文字列長を取得することが出来る。

以下に例を示す。

```
String str = "𠮷田太郎";
int lenOfChar = str.length(); // => 5
int lenOfCodePoint = str.codePointCount(0, lenOfChar); // => 4
```

また、Unicode では結合文字が存在する。

「が」を表す \u304c と、「か」と「濁点」を表す \u304b\u3099 は、見た目上の違いは存在しないが、「か」＋「濁点」の例は 2 文字としてカウントされてしまう。

こうした結合文字が入力されることも考慮して文字数をカウントする場合、 java.text.Normalizer を使用してテキストの正規化を行ってからカウントする。

以下に、結合文字とサロゲートペアを考慮した上で、文字列の長さを返却するメソッドを示す。

```
public int getStrLength(String str) {
  String normalizedStr  = Normalizer.normalize(str, Normalizer.Form.NFC);
  int length = normalizedStr.codePointCount(0, normalizedStr.length());

  return length;
}
```

指定範囲の文字列取得

指定範囲の文字列を取得する場合、単に String#substring を利用すると、想定していない結果になる可能性がある。

```
String str = "𠮷田 太郎";
int startIndex = 0;
int endIndex = 2;

String subStr = str.substring(startIndex, endIndex);

System.out.println(subStr); // => "𠮷"
```

上記の例では、0 文字目（先頭）から 2 文字を取り出し、「𠮷田」を取得しようと試みているが、サロゲート

ペアは 32 ビット（char 型 2 つ）で表現されるため「□」しか取得できない。

このような場合には、String#offsetByCodePoints を利用し、サロゲートペアを考慮した開始位置と終了位置を求めてから String#substring メソッドを使う必要がある。

以下に、先頭から 2 文字（苗字部分）を取り出す例を示す。

```
String str = "□田 太郎";
int startIndex = 0;
int endIndex = 2;

int startIndexSurrogate = str.offsetByCodePoints(0, startIndex); // => 0
int endIndexSurrogate = str.offsetByCodePoints(0, endIndex); // => 3

String subStrSurrogate = str.substring(startIndexSurrogate, endIndexSurrogate); //␣
↪=> "□田"
```

文字列分割

String#split メソッドは、サロゲートペアにデフォルトで対応している。

以下に例を示す。

```
String str = "□田 太郎";

str.split(" "); // => {"□田", "太郎"}
```

注釈:

サロゲートペアを区切り文字として、 String#split の引数に指定することも出来る。

注釈:

Java SE 7 までの環境と Java SE 8 で、 String#split に空文字を渡したときの挙動に変化があるため留意されたい。 参照： Pattern#split の Javadoc

```
String str = "ABC";
String[] elems = str.split("");

// Java SE 7 => {, A, B, C}
// Java SE 8 => {A, B, C}
```

全角・半角文字列変換

全角文字と半角文字の変換は、共通ライブラリが提供する `org.terasoluna.gfw.common.fullhalf.FullHalfConverter` クラスの API を使用して行う。

`FullHalfConverter` クラスは、変換対象にしたい全角文字と半角文字のペア定義 (`org.terasoluna.gfw.common.fullhalf.FullHalfPair`) を事前に登録しておくスタイルを採用している。共通ライブラリでは、デフォルトのペア定義が登録されている `FullHalfConverter` オブジェクトを、`org.terasoluna.gfw.common.fullhalf.DefaultFullHalf` クラスの `INSTANCE` 定数として提供している。デフォルトのペア定義については、DefaultFullHalf のソース を参照されたい。

注釈: 共通ライブラリが提供しているデフォルトのペア定義で変換要件が満たせない場合は、独自のペア定義を登録した `FullHalfConverter` オブジェクトを作成すればよい。具体的な作成方法については、独自の全角文字と半角文字のペア定義を登録した *FullHalfConverter* クラスの作成 を参照されたい。

共通ライブラリの適用方法

全角・半角文字列変換 を使う場合は、共通ライブラリを依存ライブラリとして以下の通り追加する必要がある。

```
<dependencies>
    <dependency>
        <groupId>org.terasoluna.gfw</groupId>
        <artifactId>terasoluna-gfw-string</artifactId>
    </dependency>
</dependencies>
```

注釈: 上記設定例では、依存ライブラリのバージョンは親プロジェクトで管理する前提である。そのため、<version>要素は指定していない。

全角文字列に変換

半角文字を全角文字へ変換する場合は、`FullHalfConverter` の `toFullwidth` メソッドを使用する。

```
String fullwidth = DefaultFullHalf.INSTANCE.toFullwidth("ｱﾞ!A8ｶﾞｻﾞ");     // (1)
```

項番	説明
(1)	半角文字が含まれる文字列を toFullwidth メソッドの引数に渡し、全角文字列へ変換する。 本例では、ｱﾞ ！Ａ８ｶﾞｻに変換される。なお、ペア定義されていない文字（本例の」ｻﾞ」 ）はそのまま返却される。

半角文字列に変換

全角文字を半角文字へ変換する場合は、FullHalfConverter の toHalfwidth メソッドを使用する。

```
String halfwidth = DefaultFullHalf.INSTANCE.toHalfwidth("Ａ！アガｻ");     // (1)
```

項番	説明
(1)	全角文字が含まれる文字列を toHalfwidth メソッドの引数に渡し、半角文字列へ変換する。 本例では、A!ｱｶﾞｻに変換される。なお、ペア定義されていない文字（本例の」ｻ」 ）はそのまま返却される。

注釈: FullHalfConverter は、2 文字以上で 1 文字を表現する結合文字（例：「」シ」 (\u30b7) + 濁点 (\u3099)」）を半角文字（例：」ｼﾞ」 ）へ変換することが出来ない。結合文字を半角文字へ変換する場合は、テキスト正規化を行って合成文字（例：」ジ」 (\u30b8)）に変換してから FullHalfConverter を使用する必要がある。

テキスト正規化を行う場合は、java.text.Normalizer を使用する。なお、結合文字を合成文字に変換する場合は、正規化形式として NFC または NFKC を利用する。

正規化形式として NFD（正準等価性によって分解する）を使用する場合の実装例

```
String str1 = Normalizer.normalize("モジ", Normalizer.Form.NFD); // str1 = "モシ + Voiced sound mark(\u3099)"
String str2 = Normalizer.normalize("ﾓｼﾞ", Normalizer.Form.NFD);  // str2 = "ﾓｼﾞ"
```

正規化形式として NFC（正準等価性によって分解し、再度合成する）を使用する場合の実装例

```
String mojiStr = "モシ\u3099";                                    // "モシ +␣
↪Voiced sound mark(\u3099)"
String str1 = Normalizer.normalize(mojiStr, Normalizer.Form.NFC); // str1 = "モジ (\u30b8) "
String str2 = Normalizer.normalize("ﾓｼﾞ", Normalizer.Form.NFC);   // str2 = "ﾓｼﾞ
↪"
```

正規化形式として NFKD（互換等価性によって分解する）を使用する場合の実装例

```
String str1 = Normalizer.normalize("モジ", Normalizer.Form.NFKD); // str1 = "モシ + Voiced sound mark(\u3099)"
String str2 = Normalizer.normalize("ﾓｼﾞ", Normalizer.Form.NFKD);  // str2 = "モシ + Voiced sound mark(\u3099)"
```

正規化形式として NFKC（互換等価性によって分解し、再度合成する）を使用する場合の実装例

```
String mojiStr = "モシ\u3099";                                    // "モシ + Voiced sound mark(\u3099)"
String str1 = Normalizer.normalize(mojiStr, Normalizer.Form.NFKC); // str1 = "モジ (\u30b8) "
String str2 = Normalizer.normalize("ﾓｼﾞ", Normalizer.Form.NFKC) ;  // str2 = "モジ"
```

詳細は Normalizer の JavaDoc を参照されたい。

独自の全角文字と半角文字のペア定義を登録した FullHalfConverter クラスの作成

DefaultFullHalf を使用せず、独自の全角文字と半角文字のペア定義を登録した FullHalfConverter を使用することも出来る。

以下に、独自の全角文字と半角文字のペア定義を登録した FullHalfConverter を使用する方法を示す。

独自のペア定義を登録した FullHalfConverter を提供するクラスの実装例

```
public class CustomFullHalf {

    private static final int FULL_HALF_CODE_DIFF = 0xFEE0;

    public static final FullHalfConverter INSTANCE;

    static {
        // (1)
        FullHalfPairsBuilder builder = new FullHalfPairsBuilder();

        // (2)
        builder.pair("ー", "-");

        // (3)
        for (char c = '!'; c <= '~'; c++) {
            String fullwidth = String.valueOf((char) (c + FULL_HALF_CODE_DIFF));
            builder.pair(fullwidth, String.valueOf(c));
        }

        // (4)
        builder.pair("。", "｡").pair("「", "｢").pair("」", "｣").pair("、", "､")
                .pair("・", "･").pair("ァ", "ｧ").pair("ィ", "ｨ").pair("ゥ", "ｩ")
                .pair("ェ", "ｪ").pair("ォ", "ｫ").pair("ャ", "ｬ").pair("ュ", "ｭ")
                .pair("ョ", "ｮ").pair("ッ", "ｯ").pair("ア", "ｱ").pair("イ", "ｲ")
                .pair("ウ", "ｳ").pair("エ", "ｴ").pair("オ", "ｵ").pair("カ", "ｶ")
```

```
                .pair("キ", "ｷ").pair("ク", "ｸ").pair("ケ", "ｹ").pair("コ", "ｺ")
                .pair("サ", "ｻ").pair("シ", "ｼ").pair("ス", "ｽ").pair("セ", "ｾ")
                .pair("ソ", "ｿ").pair("タ", "ﾀ").pair("チ", "ﾁ").pair("ツ", "ﾂ")
                .pair("テ", "ﾃ").pair("ト", "ﾄ").pair("ナ", "ﾅ").pair("ニ", "ﾆ")
                .pair("ヌ", "ﾇ").pair("ネ", "ﾈ").pair("ノ", "ﾉ").pair("ハ", "ﾊ")
                .pair("ヒ", "ﾋ").pair("フ", "ﾌ").pair("ヘ", "ﾍ").pair("ホ", "ﾎ")
                .pair("マ", "ﾏ").pair("ミ", "ﾐ").pair("ム", "ﾑ").pair("メ", "ﾒ")
                .pair("モ", "ﾓ").pair("ヤ", "ﾔ").pair("ユ", "ﾕ").pair("ヨ", "ﾖ")
                .pair("ラ", "ﾗ").pair("リ", "ﾘ").pair("ル", "ﾙ").pair("レ", "ﾚ")
                .pair("ロ", "ﾛ").pair("ワ", "ﾜ").pair("ヲ", "ﾝ").pair("ン", "ﾝ")
                .pair("ガ", "ｶﾞ").pair("ギ", "ｷﾞ").pair("グ", "ｸﾞ")
                .pair("ゲ", "ｹﾞ").pair("ゴ", "ｺﾞ").pair("ザ", "ｻﾞ")
                .pair("ジ", "ｼﾞ").pair("ズ", "ｽﾞ").pair("ゼ", "ｾﾞ")
                .pair("ゾ", "ｿﾞ").pair("ダ", "ﾀﾞ").pair("ヂ", "ﾁﾞ")
                .pair("ヅ", "ﾂﾞ").pair("デ", "ﾃﾞ").pair("ド", "ﾄﾞ")
                .pair("バ", "ﾊﾞ").pair("ビ", "ﾋﾞ").pair("ブ", "ﾌﾞ")
                .pair("ベ", "ﾍﾞ").pair("ボ", "ﾎﾞ").pair("パ", "ﾊﾟ")
                .pair("ピ", "ﾋﾟ").pair("プ", "ﾌﾟ").pair("ペ", "ﾍﾟ")
                .pair("ポ", "ﾎﾟ").pair("ヴ", "ｳﾞ").pair("\u30f7", "ﾜﾞ")
                .pair("\u30fa", "ｦﾞ").pair("ﾞ ", "ﾞ").pair("ﾟ ", "ﾟ").pair("  ", " ");

        // (5)
        INSTANCE = new FullHalfConverter(builder.build());
    }
}
```

項番	説明
(1)	`org.terasoluna.gfw.common.fullhalf.FullHalfPairsBuilder` を使用して、全角文字と半角文字のペア定義のセットを表現する `org.terasoluna.gfw.common.fullhalf.FullHalfPairs` を作成する。
(2)	`DefaultFullHalf` では、全角文字の」ー」 に対する半角文字を」-」 (`\uFF70`) に設定しているところを、本例では」−」 (`\u002D`) に変更している。 なお、」−」 (`\u002D`) は、下記 (3) の処理対象にも含まれているが、先に定義したペア定義が優先される仕組みになっている。
(3)	本例では、Unicode の全角の」！」 から」〜」 までと半角の」!」 から」~」 までのコード値を、コード値の並び順が同じであるという特徴を利用して、ループ処理を使ってペア定義を行っている。
(4)	上記 (3) 以外の文字はコード値の並び順が全角文字と半角文字で一致しないため、それぞれ個別にペア定義を行う。
(5)	`FullHalfPairsBuilder` より作成した `FullHalfPairs` を使用して、`FullHalfConverter` を作成する。

注釈: `FullHalfPairsBuilder#pair` メソッドの引数に指定可能な値については、FullHalfPair のコンストラクタの JavaDoc を参照されたい。

独自のペア定義を登録した **FullHalfConverter** の使用例

```
String halfwidth = CustomFullHalf.INSTANCE.toHalfwidth("ハローワールド！"); // (1)
```

項番	説明
(1)	独自のペア定義が登録された FullHalfConverter オブジェクトの toHalfwidth メソッドを使用して、全角文字が含まれる文字列を半角文字列へ変換する。 本例では、ﾊﾛｰﾜｰﾙﾄﾞ！に変換される。（」-」 は \u002D）

コードポイント集合チェック (文字種チェック)

文字種チェックを行う場合は、共通ライブラリから提供しているコードポイント集合機能を使用してチェックするとよい。

ここでは、コードポイント集合機能を使用した文字種チェックの実装方法を説明する。

- コードポイント集合の作成
- コードポイント集合同士の集合演算
- コードポイント集合を使った文字列チェック
- *Bean Validation* と連携した文字列チェック
- コードポイント集合クラスの新規作成

共通ライブラリの適用方法

コードポイント集合チェック *(*文字種チェック*)* を使う場合は、 共通ライブラリから提供しているコードポイント集合クラス 等を依存ライブラリとして追加する必要がある。

コードポイント集合の作成

org.terasoluna.gfw.common.codepoints.CodePoints は、コードポイント集合を表現するクラスである。

CodePoints のインスタンスの作成方法を以下に示す。

ファクトリメソッドを呼び出してインスタンスを作成する場合（キャッシュあり）

コードポイント集合クラス (Class<? extends CodePoints>) からインスタンスを作成し、作成したインスタンスをキャッシュする方法を以下に示す。

特定のコードポイント集合は、複数回作成する必要はないため、この方法を使用してキャッシュすることを推奨する。

```
CodePoints codePoints = CodePoints.of(ASCIIPrintableChars.class);  // (1)
```

項番	説明
(1)	CodePoints#of メソッド (ファクトリメソッド) にコードポイント集合クラスを渡してインスタンスを取得する。 本例では、 Ascii 印字可能文字のコードポイント集合クラス (org.terasoluna.gfw.common.codepoints.catalog.ASCIIPrintableChars) のインスタンスを取得している。

注釈: コードポイント集合クラスは、CodePoints クラスと同じモジュール内に複数存在する。その他にもコードポイント集合を提供するモジュールが存在するが、それらのモジュールは必要に応じて自プロジェクトに追加する必要がある。詳細は、共通ライブラリから提供しているコードポイント集合クラス を参照されたい。

また、 新規にコードポイント集合クラスを作成することも出来る。詳細は、 コードポイント集合クラスの新規作成 を参照されたい。

コードポイント集合クラスのコンストラクタを呼び出してインスタンスを作成する場合

コードポイント集合クラスからインスタンスを作成する方法を以下に示す。

この方法を使用した場合、作成したインスタンスはキャッシュされないため、キャッシュすべきでない処理（集合演算の引数等）で使用することを推奨する。

```
CodePoints codePoints = new ASCIIPrintableChars();  // (1)
```

項番	説明
(1)	new 演算子を使用してコンストラクタを呼び出し、コードポイント集合クラスのインスタンスを生成する。 本例では、 Ascii 印字可能文字のコードポイント集合クラス (ASCIIPrintableChars) のインスタンスを生成している。

CodePoints のコンストラクタを呼び出してインスタンスを作成する場合

CodePoints からインスタンスを作成する方法を以下に示す。
この方法を使用した場合、作成したインスタンスはキャッシュされないため、キャッシュすべきでない処理（集合演算の引数等）で使用することを推奨する。

- コードポイント (int) を可変長引数で渡す場合

```
CodePoints codePoints = new CodePoints(0x0061 /* a */, 0x0062 /* b */);  //␣
↪(1)
```

項番	説明
(1)	int のコードポイントを、CodePoints のコンストラクタに渡してインスタンスを生成する。 本例では、 文字」a」 と」b」 のコードポイント集合のインスタンスを生成している。

- コードポイント (int) の Set を渡す場合

```
Set<Integet> set = new HashSet<>();
set.add(0x0061 /* a */);
set.add(0x0062 /* b */);
CodePoints codePoints = new CodePoints(set);  // (1)
```

項番	説明
(1)	int のコードポイントを Set に追加し、Set を CodePoints のコンストラクタに渡してインスタンスを生成する。 本例では、 文字」a」 と」b」 のコードポイント集合のインスタンスを生成している。

- コードポイント集合文字列を可変長引数で渡す場合

```
CodePoints codePoints = new CodePoints("ab");          // (1)
```

```
CodePoints codePoints = new CodePoints("a", "b");  // (2)
```

項番	説明
(1)	コードポイント集合文字列を CodePoints のコンストラクタに渡してインスタンスを生成する。 本例では、 文字」a」 と」b」 のコードポイント集合のインスタンスを生成している。
(2)	コードポイント集合文字列を複数に分けて渡すことも出来る。(1) と同じ結果となる。

コードポイント集合同士の集合演算

コードポイント集合に対して集合演算を行い、新規のコードポイント集合のインスタンスを作成することが出来る。

なお、集合演算によって元のコードポイント集合の状態が変更されることは無い。

集合演算を使用してコードポイント集合のインスタンスを作成する方法を以下に示す。

和集合メソッドを使用してコードポイント集合のインスタンスを作成する場合

```
CodePoints abCp = new CodePoints(0x0061 /* a */, 0x0062 /* b */);
CodePoints cdCp = new CodePoints(0x0063 /* c */, 0x0064 /* d */);

CodePoints abcdCp = abCp.union(cdCp);    // (1)
```

項番	説明
(1)	CodePoints#union メソッドを使用して２つのコードポイント集合の和集合を計算し、新規のコードポイント集合のインスタンスを作成する。 本例では、「文字列 ab に含まれるコードポイント集合」と「文字列 cd に含まれるコードポイント集合」の和集合を計算し、新規のコードポイント集合（文字列 abcd に含まれるコードポイント集合）のインスタンスを作成している。

差集合メソッドを使用してコードポイント集合のインスタンスを作成する場合

```
CodePoints abcdCp = new CodePoints(0x0061 /* a */, 0x0062 /* b */,
        0x0063 /* c */, 0x0064 /* d */);
CodePoints cdCp = new CodePoints(0x0063 /* c */, 0x0064 /* d */);

CodePoints abCp = abcdCp.subtract(cdCp);    // (1)
```

項番	説明
(1)	CodePoints#subtract メソッドを使用して２つのコードポイント集合の差集合を計算し、新規のコードポイント集合のインスタンスを作成する。 本例では、「文字列 abcd に含まれるコードポイント集合」と「文字列 cd に含まれるコードポイント集合」の差集合を計算し、新規のコードポイントの集合（文字列 ab に含まれるコードポイント集合）のインスタンスを作成している。

積集合で新規のコードポイント集合のインスタンスを作成する場合

```
CodePoints abcdCp = new CodePoints(0x0061 /* a */, 0x0062 /* b */,
        0x0063 /* c */, 0x0064 /* d */);
CodePoints cdeCp = new CodePoints(0x0063 /* c */, 0x0064 /* d */, 0x0064 /* e */);

CodePoints cdCp = abcdCp.intersect(cdeCp);     // (1)
```

項番	説明
(1)	CodePoints#intersect メソッドを使用して 2 つのコードポイント集合の積集合を計算し、新規のコードポイント集合のインスタンスを作成する。 本例では、「文字列 abcd に含まれるコードポイント集合」と「文字列 cde に含まれるコードポイント集合」の積集合を計算し、新規のコードポイントの集合（文字列 cd に含まれるコードポイント集合）のインスタンスを作成している。

コードポイント集合を使った文字列チェック

CodePoints に用意されているメソッドを使用して文字列チェックを行うことが出来る。
以下に、文字列チェックを行う際に使用するメソッドの使用例を示す。

containsAll メソッド

チェック対象の文字列が全てコードポイント集合に含まれているか判定する。

```
CodePoints jisX208KanaCp = CodePoints.of(JIS_X_0208_Katakana.class);

boolean result;
result = jisX208KanaCp.containsAll("カ");      // true
result = jisX208KanaCp.containsAll("カナ");    // true
result = jisX208KanaCp.containsAll("カナ a");  // false
```

firstExcludedContPoint メソッド

チェック対象の文字列のうち、コードポイント集合に含まれない最初のコードポイントを返却する。なお、チェック対象の文字列が全てコードポイント集合に含まれている場合は、CodePoints#NOT_FOUND を返

却する。

```
CodePoints jisX208KanaCp = CodePoints.of(JIS_X_0208_Katakana.class);

int result;
result = jisX208KanaCp.firstExcludedCodePoint("カナ a");  // 0x0061 (a)
result = jisX208KanaCp.firstExcludedCodePoint("カ a ナ");  // 0x0061 (a)
result = jisX208KanaCp.firstExcludedCodePoint("カナ");   // CodePoints#NOT_FOUND
```

allExcludedCodePoints メソッド

チェック対象の文字列のうち、コードポイント集合に含まれないコードポイントの Set を返却する。

```
CodePoints jisX208KanaCp = CodePoints.of(JIS_X_0208_Katakana.class);

Set<Integer> result;
result = jisX208KanaCp.allExcludedCodePoints("カナ a");  // [0x0061 (a)]
result = jisX208KanaCp.allExcludedCodePoints("カ a ナ b"); // [0x0061 (a), 0x0062 (b)]
result = jisX208KanaCp.allExcludedCodePoints("カナ");   // []
```

Bean Validation と連携した文字列チェック

@org.terasoluna.gfw.common.codepoints.ConsistOf アノテーションにコードポイント集合クラスを指定することで、チェック対象の文字列が指定したコードポイント集合に全て含まれるかをチェックすることが出来る。

以下に使用例を示す。

チェックに用いるコードポイント集合が一つの場合

```
@ConsistOf(JIS_X_0208_Hiragana.class)    // (1)
private String firstName;
```

項番	説明
(1)	対象のフィールドに設定された文字列が、全て「JIS X 0208 のひらがな」であることをチェックする。

チェックに用いるコードポイント集合が複数の場合

```
@ConsistOf({JIS_X_0208_Hiragana.class, JIS_X_0208_Katakana.class})     // (1)
private String firstName;
```

項番	説明
(1)	対象のフィールドに設定された文字列が、全て「JIS X 0208 のひらがな」または「JIS X 0208 のカタカナ」であることをチェックする。

注釈: 長さ N の文字列を M 個のコードポイント集合でチェックした場合、N x M 回のチェック処理が発生する。文字列の長さが大きい場合は、性能劣化の要因となる恐れがある。そのため、チェックに使用するコードポイント集合の和集合となる新規コードポイント集合のクラスを作成し、そのクラスのみを指定したほうが良い。

コードポイント集合クラスの新規作成

コードポイント集合クラスを新規で作成する場合、`CodePoints` クラスを継承してコンストラクタでコードポイントを指定する。

コードポイント集合クラスを新規で作成する方法を以下に示す。

コードポイントを指定して新規にコードポイント集合のクラスを作成する場合

「数字のみ」からなるコードポイント集合の作成例

```
public class NumberChars extends CodePoints {
    public NumberCodePoints() {
        super(0x0030 /* 0 */, 0x0031 /* 1 */, 0x0032 /* 2 */, 0x0033 /* 3 */,
                0x0034 /* 4 */, 0x0035 /* 5 */, 0x0036 /* 6 */,
                0x0037 /* 7 */, 0x0038 /* 8 */, 0x0039 /* 9 */);
    }
}
```

コードポイント集合クラスの集合演算メソッドを使用して新規にコードポイント集合クラスを作成する場合

「ひらがな」と「カタカナ」からなる和集合を用いたコードポイント集合の作成例

```
public class FullwidthHiraganaKatakana extends CodePoints {
    public FullwidthHiraganaKatakana() {
        super(new X_JIS_0208_Hiragana().union(new X_JIS_0208_Katakana()));
    }
}
```

「記号（。「」、・）を除いた半角カタカナ」からなる差集合を用いたコードポイント集合の作成例

```
public class HalfwidthKatakana extends CodePoints {
    public HalfwidthKatakana() {
        super(new JIS_X_0201_Katakana().subtract(new CodePoints(0xFF61 /* 。 */,␣
↪0xFF62 /* 「 */,
                0xFF63 /* 」 */, 0xFF64 /* 、 */, 0xFF65 /* ・ */)));
    }
}
```

注釈: 集合演算で使用するコードポイント集合クラス（本例では X_JIS_0208_Hiragana や、X_JIS_0208_Katakana 等）を個別に使用するケースがない場合は、 new 演算子を使用してコンストラクタを呼び出し、コードポイント集合が無駄にキャッシュされないようにすべきである。CodePoints#of メソッドを使用してキャッシュさせてしまうと、集合演算の途中計算のみで使用するコードポイント集合がヒープに残り、メモリを圧迫してしまう。逆に個別に使用するケースがある場合は、CodePoints#of メソッドを使用してキャッシュすべきである。

共通ライブラリから提供しているコードポイント集合クラス

共通ライブラリから提供しているコードポイント集合クラス (org.terasoluna.gfw.common.codepoints.catalog パッケージのクラス) と、使用する際に取り込む必要があるアーティファクトの情報を以下に示す。

項番	クラス名	説明	アーティファクト情報
(1)	ASCIIControlChars	Ascii 制御文字の集合。(0x0000-0x001F、0x007F)	<dependency> <groupId>org.terasoluna.gfw</groupId> <artifactId>terasoluna-gfw-codepoints</artifactId> </dependency>
(2)	ASCIIPrintableChars	Ascii 印字可能文字の集合。(0x0020-0x007E)	(同上)
(3)	CRLF	改行コードの集合。0x000A(LINE FEED) と 0x000D(CARRIAGE RETURN)。	(同上)
(4)	JIS_X_0201_Katakana	JIS X 0201 のカタカナの集合。記号 (。「」、・) も含まれる。	<dependency> <groupId>org.terasoluna.gfw.codepoints</groupId> <artifactId>terasoluna-gfw-codepoints-jisx0201</artifactId> </dependency>
(5)	JIS_X_0201_LatinLetters	JIS X 0201 の Latin 文字の集合。	(同上)
(6)	JIS_X_0208_SpecialChars	JIS X 0208 の 1-2 区：特殊文字の集合。	<dependency> <groupId>org.terasoluna.gfw.codepoints</groupId> <artifactId>terasoluna-gfw-codepoints-jisx0208</artifactId> </dependency>
(7)	JIS_X_0208_LatinLetters	JIS X 0208 の 3 区：英数字の集合。	(同上)

Continued on next page

表 7.8 – continued from previous page

項番	クラス名	説明	アーティファクト情報
(8)	JIS_X_0208_Hiragana	JIS X 0208 の 4 区：ひらがなの集合。	(同上)
(9)	JIS_X_0208_Katakana	JIS X 0208 の 5 区：カタカナの集合。	(同上)
(10)	JIS_X_0208_GreekLetters	JIS X 0208 の 6 区：ギリシア文字の集合。	(同上)
(11)	JIS_X_0208_CyrillicLetters	JIS X 0208 の 7 区：キリル文字の集合。	(同上)
(12)	JIS_X_0208_BoxDrawingChars	JIS X 0208 の 8 区：罫線素片の集合。	(同上)

(13)	JIS_X_0208_Kanji	JIS X 208 で規定される漢字 6355 字。
		第一・第二水準漢字。
			`<dependency>`
			`  <groupId>org.terasoluna.gfw.codepoints</groupId>`
			`  <artifactId>terasoluna-gfw-codepoints-jisx0208kanji</artifactId>`
			`</dependency>`

(14)	JIS_X_0213_Kanji	JIS X 0213:2004 で規定される漢字 10050 字。
		第一・第二・第三・第四水準漢字。
			`<dependency>`
			`  <groupId>org.terasoluna.gfw.codepoints</groupId>`
			`  <artifactId>terasoluna-gfw-codepoints-jisx0213kanji</artifactId>`
			`</dependency>`

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。

注釈: <artifactId>が terasoluna-gfw-codepoints-xxx(terasoluna-gfw-codepoints-jisx0201 など）のライブラリでは依存関係として terasoluna-gfw-codepoints を取り込んでいる。

そのため、 terasoluna-gfw-codepoints-xxx のアーティファクト情報を取り込むことにより、terasoluna-gfw-codepoints が提供するコードポイント集合クラスも利用することができる。

注釈: JIS_X_0208_SpecialChars コードポイント集合クラスは JIS 漢字 (JIS X 0208) の 01-02 区に該当する特殊文字集合である。JIS 漢字の全角ダッシュ (—) は EM DASH であり、対応する UCS(ISO/IEC 10646-1, JIS X 0221, Unicode) のコードポイントは、一般的に U+2014 に相当する。しかし、Unicode コンソーシアムが提供する変換表では、Unicode で対応する文字が EM DASH でなく HORINZONTAL BAR (U+2015) になっている。実用されている一般的な変換ルールと、Unicode 変換表が異なっているため、Unicode 変換表通りにコードポイント集合を定義してしまうと実用上問題が出るケースが発生する可能性がある。そのため、JIS_X_0208_SpecialChars コードポイント集合クラスでは HORINZONTAL BAR (U+2015) を EM DASH (U+2014) に変更してコードポイント集合を定義している。

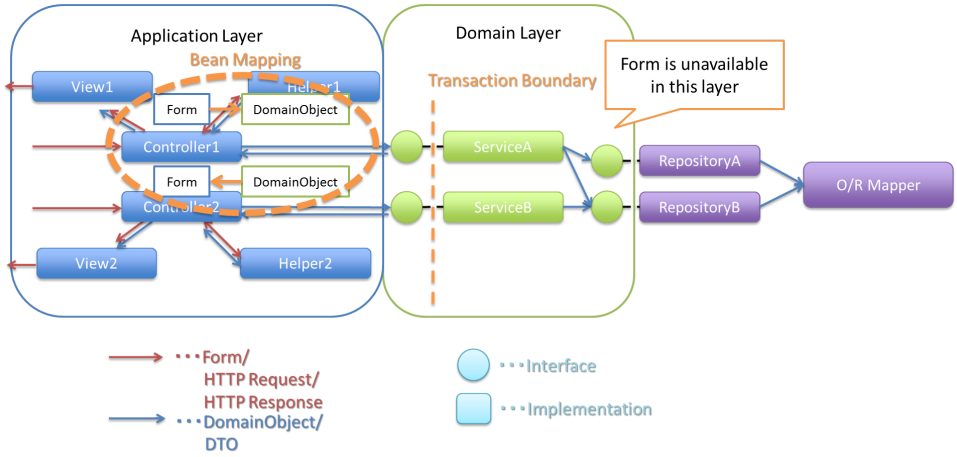
7.7 Bean マッピング (Dozer)

7.7.1 Overview

Bean マッピングは、一つの Bean を他の Bean にフィールド値をコピーすることである。

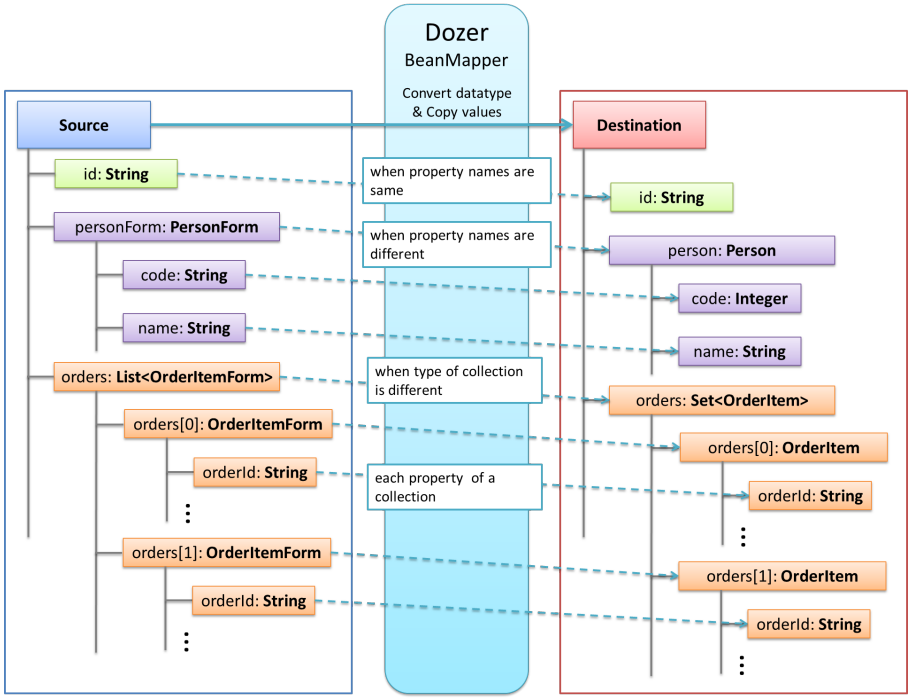
アプリケーションの異なるレイヤ間 (アプリケーション層とドメイン層) で、データの受け渡しをする場合など、Bean マッピングが必要となるケースは多い。

例として、アプリケーション層の `AccountForm` オブジェクトを、ドメイン層の `Account` オブジェクトに変換する場合を考える。
ドメイン層は、アプリケーション層に依存してはならないため、AccountForm オブジェクトをそのままドメイン層で使用できない。
そこで、`AccountForm` オブジェクトを、`Account` オブジェクトに Bean マッピングし、ドメイン層では、`Account` オブジェクトを使用する。
これによって、アプリケーション層と、ドメイン層の依存関係を一方向に保つことができる。



このオブジェクト間のマッピングは、Bean の getter/setter を呼び出して、データの受け渡しを行うことで実現できる。
しかしながら、処理が煩雑になり、プログラムの見通しが悪くなるため、本ガイドラインでは、Bean マッピングライブラリである OSS で利用可能な Dozer を使用することを推奨する。

Dozer を使用することで下図のように、コピー元クラスとコピー先クラスで型が異なるコピーや、ネストした Bean 同士のコピーも容易に行うことができる。



Dozer をした場合と使用しない場合のコード例を挙げる。

- 煩雑になり、プログラムの見通しが悪くなる例

```
User user = userService.findById(userId);

XxxOutput output = new XxxOutput();

output.setUserId(user.getUserId());
output.setFirstName(user.getFirstName());
output.setLastName(user.getLastName());
output.setTitle(user.getTitle());
output.setBirthDay(user.getBirthDay());
output.setGender(user.getGender());
output.setStatus(user.getStatus());
```

- Dozer を使用した場合の例

```
User user = userService.findById(userId);

XxxOutput output = beanMapper.map(user, XxxOutput.class);
```

以降は、Dozer の利用方法について説明する。

> **警告：** JSR-310 Date and Time API で追加されたクラス群はデフォルトのままでは例外が発生するため使用できず、カスタムコンバーターを作成する必要がある。

7.7.2 How to use

Dozer は、Java Bean のマッピング機能ライブラリである。変換元の Bean から変換先の Bean に、再帰的（ネストした構造）に、値をコピーする。

Dozer を使用するための Bean 定義

Dozer は、単独で使用するとき、以下のように、 `org.dozer.Mapper` のインスタンスを作成する。

```
Mapper mapper = new DozerBeanMapper();
```

Mapper のインスタンスを毎回作成するのは、効率が悪いため、Dozer が提供している `org.dozer.spring.DozerBeanMapperFactoryBean` を使用すること。

Bean 定義ファイル (applicationContext.xml) に、Mapper を作成する Factory クラスである `org.dozer.spring.DozerBeanMapperFactoryBean` を定義する

```
<bean class="org.dozer.spring.DozerBeanMapperFactoryBean">
    <property name="mappingFiles"
        value="classpath*:/META-INF/dozer/**/*-mapping.xml" /><!-- (1) -->
</bean>
```

項番	説明
(1)	mappingFiles に、マッピング定義 XML ファイルを指定する。 `org.dozer.spring.DozerBeanMapperFactoryBean` は、 interface として `org.dozer.Mapper` を保持している。そのため、 `@Inject` 時は `Mapper` を指定する。 この例では、クラスパス直下の、/META-INF/dozer の任意フォルダ内の (任意の値)-mapping.xml を、すべて読み込む。この XML ファイルの内容については、以降で説明する。

Bean マッピングを行いたいクラスに、 `Mapper` をインジェクトすればよい。

```
@Inject
Mapper beanMapper;
```

Bean 間のフィールド名、型が同じ場合のマッピング

デフォルトの動作として、Dozer は対象の Bean 間のフィールド名が同じであれば、マッピング定義 XML ファイルを作成せずにマッピングできる。

変換元の Bean 定義

```
public class Source {
    private int id;
    private String name;
    // omitted setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int id;
    private String name;
    // omitted setter/getter
}
```

以下のように、 Mapper の map メソッドを使って Bean マッピングを行う。下記メソッドを実行した後、Destination オブジェクトが新たに作成され、source の各フィールドの値が作成された Destination オブジェクトにコピーされる。

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
Destination destination = beanMapper.map(source, Destination.class); // (1)
System.out.println(destination.getId());
System.out.println(destination.getName());
```

項番	説明
(1)	第一引数に、コピー元のオブジェクトを渡し、第二引数に、コピー先の Bean のクラスを渡す。

上記のコードを実行すると以下のように出力される。作成されたオブジェクトにコピー元のオブジェクトの値が設定されていることが分かる。

```
1
SourceName
```

既に存在している destination オブジェクトに、source オブジェクトのフィールドをコピーしたい場合は、

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
Destination destination = new Destination();
destination.setId(2);
destination.setName("DestinationName");
beanMapper.map(source, destination); // (1)
System.out.println(destination.getId());
System.out.println(destination.getName());
```

項番	説明
(1)	第一引数に、コピー元のオブジェクトを渡し、第二引数に、コピー先のオブジェクトを渡す。

上記のコードを実行すると以下のように出力される。コピー元のオブジェクトの値がコピー先に反映されていることが分かる。

```
1
SourceName
```

注釈: `Destination` クラスのフィールドで `Source` クラスに存在しないものは、コピー前後で値は変わらない。

変換元の Bean 定義

```
public class Source {
    private int id;
    private String name;
    // omitted setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int id;
    private String name;
    private String title;
    // omitted setter/getter
}
```

マッピング例

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
Destination destination = new Destination();
destination.setId(2);
destination.setName("DestinationName");
destination.setTitle("DestinationTitle");
beanMapper.map(source, destination);
System.out.println(destination.getId());
System.out.println(destination.getName());
System.out.println(destination.getTitle());
```

上記のコードを実行すると以下のように出力される。Source クラスには title フィールドがないため、Destination オブジェクトの title フィールドは、コピー前のフィールド値から変更がない。

```
1
SourceName
DestinationTitle
```

Bean 間のフィールド名は同じ、型が異なる場合のマッピング

コピー元と、コピー先で Bean のフィールドの型が異なる場合、型変換がサポートされている型は、自動でマッピングできる。

以下のような変換は、マッピング定義 XML ファイル無しで変換できる。

例 : String -> BigDecimal

変換元の Bean 定義

```
public class Source {
    private String amount;
    // omitted setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private BigDecimal amount;
    // omitted setter/getter
}
```

マッピング例

```
Source source = new Source();
source.setAmount("123.45");
Destination destination = beanMapper.map(source, Destination.class);
System.out.println(destination.getAmount());
```

上記のコードを実行すると以下のように出力される。型が異なる場合でも値をコピーできていることが分かる。

```
123.45
```

サポートされている型変換については、 マニュアル を参照されたい。

Bean 間のフィールド名が異なる場合のマッピング

コピー元と、コピー先でフィールド名が異なる場合、マッピング定義 XML ファイルを作成し、Bean マッピングするフィールドを定義することで変換できる。

変換元の Bean 定義

```
public class Source {
    private int id;
    private String name;
    // omitted setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int destinationId;
    private String destinationName;
    // omitted setter/getter
}
```

Dozer を使用するための *Bean* 定義の定義がある場合、src/main/resources/META-INF/dozer フォルダ内に、(任意の値)-mapping.xml という、マッピング定義 XML ファイルを作成する。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">

    <mapping>
      <class-a>com.xx.xx.Source</class-a><!-- (1) -->
      <class-b>com.xx.xx.Destination</class-b><!-- (2) -->
      <field>
        <a>id</a><!-- (3) -->
        <b>destinationId</b><!-- (4) -->
      </field>
      <field>
        <a>name</a>
        <b>destinationName</b>
      </field>
    </mapping>

</mappings>
```

項番	説明
(1)	<class-a>タグ内にコピー元の Bean の、完全修飾クラス名 (FQCN) を指定する。
(2)	<class-b>タグ内にコピー先の Bean の、完全修飾クラス名 (FQCN) を指定する。
(3)	<field>タグ内の<a>タグ内にコピー元の Bean の、マッピング用のフィールド名を指定する。
(4)	<field>タグ内の<b>タグ内に (3) に対応するコピー先の Bean の、マッピング用のフィールド名を指定する。

マッピング例

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
Destination destination = beanMapper.map(source, Destination.class); // (1)
System.out.println(destination.getDestinationId());
System.out.println(destination.getDestinationName());
```

項番	説明
(1)	第一引数に、コピー元のオブジェクトを渡し、第二引数に、コピー先の Bean のクラスを渡す。(基本マッピングと違いはない。)

上記のコードを実行すると以下のように出力される。

```
1
SourceName
```

*Dozer を使用するための Bean 定義*の設定によって、mappingFiles プロパティにクラスパス直下の META-INF/dozer 配下に存在するマッピング定義 XML ファイルが読み込まれる。ファイル名は (任意の値)-mapping.xml である必要がある。いずれかのファイル内に Source クラスと Destination クラス間におけるマッピング定義があれば、その設定が適用される。

注釈: マッピング定義 XML ファイルは、Controller 単位で作成し、ファイル名は、(Controller 名から Controller

を除いた値)-mapping.xml にすることを推奨する。例えば、TodoController に対するマッピング定義 XML ファイルは、src/main/resources/META-INF/dozer/todo-mapping.xml に作成する。

単方向・双方向マッピング

マッピング XML で定義されているマッピングは、デフォルトで、双方向マッピングである。すなわち前述の例では Source オブジェクトから Destination オブジェクトへのマッピングを行ったが、Destination オブジェクトから Source オブジェクトのマッピングも可能である。

単方向のみを指定したい場合は、マッピング・フィールド定義に、<mapping>タグの type 属性に one-way を設定する。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
        <!-- omitted -->
        <mapping type="one-way">
              <class-a>com.xx.xx.Source</class-a>
              <class-b>com.xx.xx.Destination</class-b>
                <field>
                  <a>id</a>
                  <b>destinationId</b>
                </field>
                <field>
                  <a>name</a>
                  <b>destinationName</b>
                </field>
        </mapping>
        <!-- omitted -->
</mappings>
```

変換元の Bean 定義

```
public class Source {
    private int id;
    private String name;
    // omitted setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int destinationId;
    private String destinationName;
```

```
    // omitted setter/getter
}
```

マッピング例

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
Destination destination = beanMapper.map(source, Destination.class);
System.out.println(destination.getDestinationId());
System.out.println(destination.getDestinationName());
```

上記のコードを実行すると以下のように出力される。

```
1
SourceName
```

単方向を指定している場合に、逆方向のマッピングを行ってもエラーは発生しない。コピー処理は無視される。なぜなら、マッピング定義がないと Destination のフィールドに該当する Source のフィールドが存在ないとみなされるためである。

```
Destination destination = new Destination();
destination.setDestinationId(2);
destination.setDestinationName("DestinationName");

Source source = new Source();
source.setId(1);
source.setName("SourceName");

beanMapper.map(destination, source);

System.out.println(source.getId());
System.out.println(source.getName());
```

上記のコードを実行すると以下のように出力される。

```
1
SourceName
```

警告：単方向マッピングのバグについて

同名フィールドは<mapping>タグの type 属性に one-way を付与しても正常に単方向マッピングとならず、逆方向でもマッピングされるバグが存在する。

具体的には、以下のように定義した場合、同名フィールドは双方向マッピングとなる。

- 変換元の Bean 定義

```
public class Source {
    private String sameNameField;
    // omitted setter/getter
}
```

- 変換先の Bean 定義

```
public class Destination {
    private String sameNameField;
    // omitted setter/getter
}
```

- クラスベースでマッピング定義

```
<mapping type="one-way">
    <class-a>xxx.Source</class-a>
    <class-b>xxx.Destination</class-b>
</mapping>
```

- 同名フィールドを<field>タグを使用してマッピング定義

```
<mapping type="one-way">
    <class-a>xxx.Source</class-a>
    <class-b>xxx.Destination</class-b>
    <field>
        <a>sameNameField</a>
        <b>sameNameField</b>
    </field>
</mapping>
```

正常に単方向マッピングするためには、以下のように<field>タグの type 属性に one-way を付与する必要がある。

```
<mapping>
    <class-a>xxx.Source</class-a>
    <class-b>xxx.Destination</class-b>
    <field type="one-way">
        <a>sameNameField</a>
        <b>sameNameField</b>
    </field>
</mapping>
```

なお、このバグは TERASOLUNA Server Framework for Java 5.6.0.RELEASE が依存する Dozer 6.4.1 では、既に修正されている。

Nest したフィールドのマッピング

コピー元 Bean が持つフィールドを、コピー先 Bean が持つ Nest した Bean のフィールドにも、マッピングできることである。(Dozer の用語で、 Deep Mapping と呼ばれる。)

変換元の Bean 定義

```
public class EmployeeForm {
    private int id;
```

```
    private String name;
    private String deptId;
    // omitted setter/getter
}
```

変換先の Bean 定義

```
public class Employee {
    private Integer id;
    private String name;
    private Department department;
    // omitted setter/getter
}
```

```
public class Department {
    private String deptId;
    // omitted setter/getter and other fields
}
```

例：EmployeeForm オブジェクトが持つ deptId を、Employee オブジェクトが持つ Department の deptId にマップしたい場合、以下のように定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping map-empty-string="false" map-null="false">
        <class-a>com.xx.aa.EmployeeForm</class-a>
        <class-b>com.xx.bb.Employee</class-b>
        <field>
              <a>deptId</a>
              <b>department.deptId</b><!-- (1) -->
        </field>
    </mapping>
    <!-- omitted -->
</mappings>
```

項番	説明
(1)	Employee フォームの deptId に対する、Employee オブジェクトのフィールドを指定する。

マッピング例

```
EmployeeForm source = new EmployeeForm();
source.setId(1);
```

```
source.setName("John");
source.setDeptId("D01");

Employee destination = beanMapper.map(source, Employee.class);
System.out.println(destination.getId());
System.out.println(destination.getName());
System.out.println(destination.getDepartment().getDeptId());
```

上記のコードを実行すると以下のように出力される。

```
1
John
D01
```

上記の場合は、変換先クラスである Employee の新規インスタンスが作成される。Employee の中の department フィールドにも、新規に作成された Department インスタンスが設定され、EmployeeForm の deptId が、コピーされる。

下記のように Employee の中の department フィールドに既に Department オブジェクトが設定されている場合は、新規インスタンスは作成されず、既存の Department オブジェクトの deptId フィールドに、EmployeeForm の deptId がコピーされる。

```
EmployeeForm source = new EmployeeForm();
source.setId(1);
source.setName("John");
source.setDeptId("D01");

Employee destination = new Employee();
Department department = new Department();
destination.setDepartment(department);

beanMapper.map(source, destination);
System.out.println(department.getDeptId());
System.out.println(destination.getDepartment() == department);
```

上記のコードを実行すると以下のように出力される。

```
D01
true
```

Collection マッピング

Dozer は、以下の Collection タイプの双方向自動マッピングをサポートしている。フィールド名が同じである場合、マッピング定義 XML ファイルが不要である。

- java.util.List <=> java.util.List

- `java.util.List` <=> Array
- Array <=> Array
- `java.util.Set` <=> `java.util.Set`
- `java.util.Set` <=> Array
- `java.util.Set` <=> `java.util.List`

次のクラスのコレクションをもつ Bean のマッピングについて考える。

```
package com.example.dozer;

public class Email {
    private String email;

    public Email() {
    }

    public Email(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public String toString() {
        return email;
    }

    // generated by Eclipse
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((email == null) ? 0 : email.hashCode());
        return result;
    }

    // generated by Eclipse
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
```

```
            return false;
        Email other = (Email) obj;
        if (email == null) {
            if (other.email != null)
                return false;
        } else if (!email.equals(other.email))
            return false;
        return true;
    }

}
```

変換元の Bean

```
package com.example.dozer;

import java.util.List;

public class AccountForm {
    private List<Email> emails;

    public void setEmails(List<Email> emails) {
        this.emails = emails;
    }

    public List<Email> getEmails() {
        return emails;
    }
}
```

変換先の Bean

```
package com.example.dozer;

import java.util.List;

public class Account {
    private List<Email> emails;

    public void setEmails(List<Email> emails) {
        this.emails = emails;
    }

    public List<Email> getEmails() {
        return emails;
    }
}
```

マッピング例

```
AccountForm accountForm = new AccountForm();

List<Email> emailsSrc = new ArrayList<Email>();

emailsSrc.add(new Email("a@example.com"));
emailsSrc.add(new Email("b@example.com"));
emailsSrc.add(new Email("c@example.com"));

accountForm.setEmails(emailsSrc);

Account account = beanMapper.map(accountForm, Account.class);

System.out.println(account.getEmails());
```

上記のコードを実行すると以下のように出力される。

```
[a@example.com, b@example.com, c@example.com]
```

ここまではこれまで説明したことと特に変わりはない。

次の例のように、コピー先の **Bean** の **Collection** フィールドに既に要素が追加されている場合は要注意である。

```
AccountForm accountForm = new AccountForm();
Account account = new Account();

List<Email> emailsSrc = new ArrayList<Email>();
List<Email> emailsDest = new ArrayList<Email>();

emailsSrc.add(new Email("a@example.com"));
emailsSrc.add(new Email("b@example.com"));
emailsSrc.add(new Email("c@example.com"));

emailsDest.add(new Email("a@example.com"));
emailsDest.add(new Email("d@example.com"));
emailsDest.add(new Email("e@example.com"));

accountForm.setEmails(emailsSrc);
account.setEmails(emailsDest);

beanMapper.map(accountForm, account);

System.out.println(account.getEmails());
```

上記のコードを実行すると以下のように出力される。

```
[a@example.com, d@example.com, e@example.com, a@example.com, b@example.com,␣
↪c@example.com]
```

コピー元 Bean の Collection の全要素が、コピー先 Bean の Collection に追加されている。`a@exmample.com` をもつ 2 つの `Email` オブジェクトは」等価」であるが、単純に追加される。

(ここでいう」等価」とは Email.equals で比較すると true になり、Email.hashCode の値も同じであることを意味する。)

上記の振る舞いは、Dozer の用語では **cumulative** と呼ばれ、Collection をマッピングする際のデフォルトの挙動となっている。

この挙動はマッピング定義 XML ファイルにおいて変更することができる。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping>
        <class-a>com.example.dozer.AccountForm</class-a>
        <class-b>com.example.dozer.Account</class-b>
        <field relationship-type="non-cumulative"><!-- (1) -->
            <a>emails</a>
            <b>emails</b>
        </field>
    </mapping>
    <!-- omitted -->
</mappings>
```

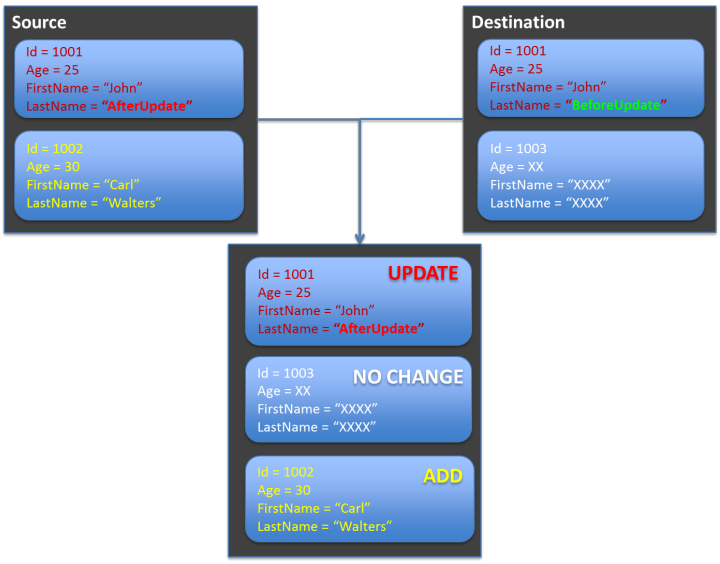
項番	説明
(1)	<field>タグの relationship-type 属性に non-cumulative を指定する。デフォルト値は cumulative である。 マッピング対象の Bean の全フィールドに対して non-cumulative を指定したい場合は、<mapping>タグの relationship-type 属性に non-cumulative を指定することもできる。

この設定のもと、前述のコードを実行すると以下のように出力される。

```
[a@example.com, d@example.com, e@example.com, b@example.com, c@example.com]
```

等価であるオブジェクトの重複がなくなっていることが分かる。

注釈: 変換元のオブジェクトが、変換先のオブジェクトで更新されることに注意されたい。上記の例では AccountForm の中の a@exmample.com がコピー先に格納される。



コピー先のコレクションにのみに存在する項目は除外したい場合も、マッピング定義 XML ファイルの設定で実現することができる。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping>
        <class-a>com.example.dozer.AccountForm</class-a>
        <class-b>com.example.dozer.Account</class-b>
        <field relationship-type="non-cumulative" remove-orphans="true" ><!-- (1) -
↪->
            <a>emails</a>
            <b>emails</b>
        </field>
    </mapping>
    <!-- omitted -->
</mappings>
```

項番	説明
(1)	<field>タグの remove-orphans 属性に true を設定する。デフォルト値は false である。

この設定のもと、前述のコードを実行すると以下のように出力される。

```
[a@example.com, b@example.com, c@example.com]
```

コピー元にあるオブジェクトだけがコピー先のコレクション内に残っていることが分かる。

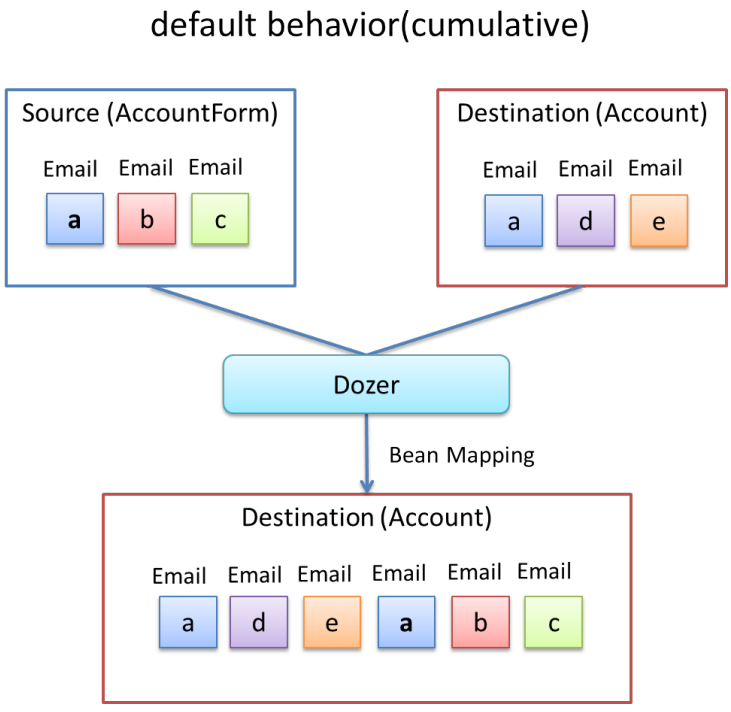
いかのように設定しても同じ結果が得られる。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping>
        <class-a>com.example.dozer.AccountForm</class-a>
        <class-b>com.example.dozer.Account</class-b>
        <field copy-by-reference="true"><!-- (1) -->
            <a>emails</a>
            <b>emails</b>
        </field>
    </mapping>
    <!-- omitted -->
</mappings>
```

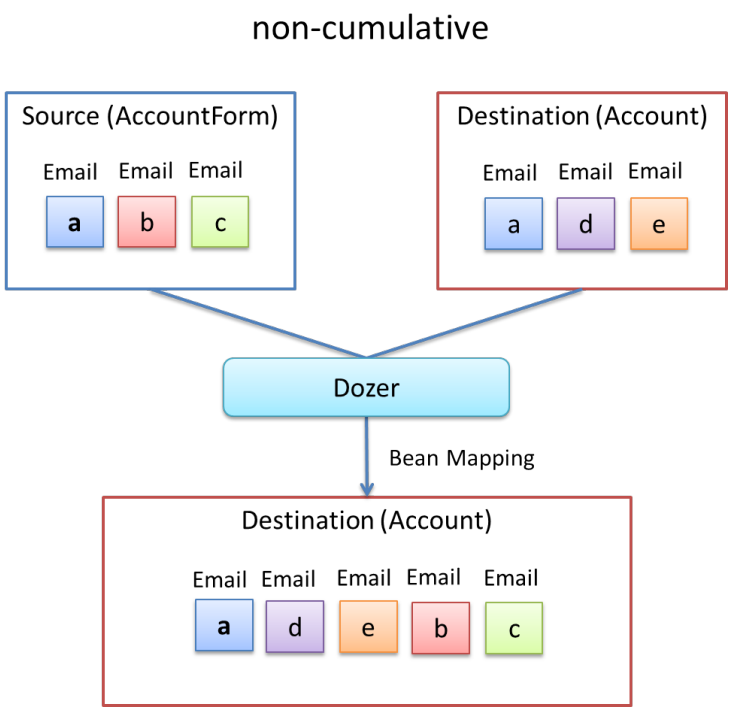
項番	説明
(1)	<field>タグの copy-by-reference 属性に true を設定する。デフォルト値は false である。

これまでの挙動を図で表現する。

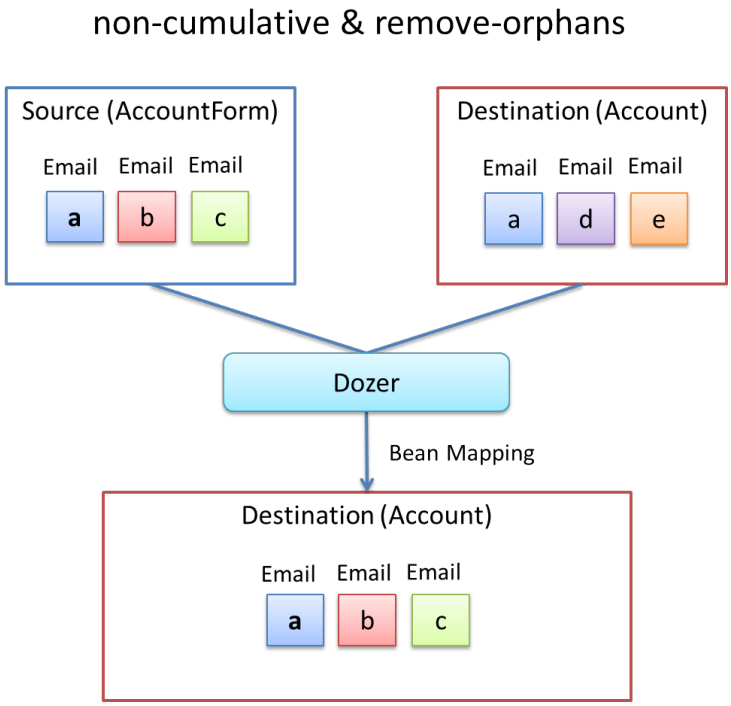
- デフォルトの挙動 (cumulative)



- non-cumulative



- non-cumulative かつ remove-orphans=true

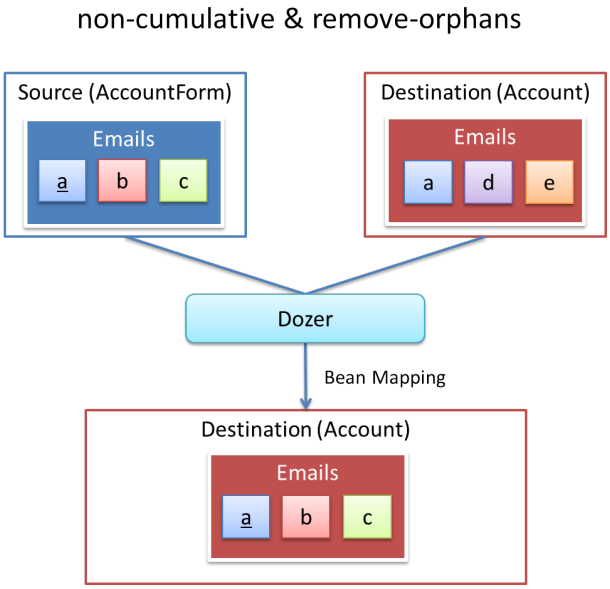


copy-by-reference もこのパターンである。

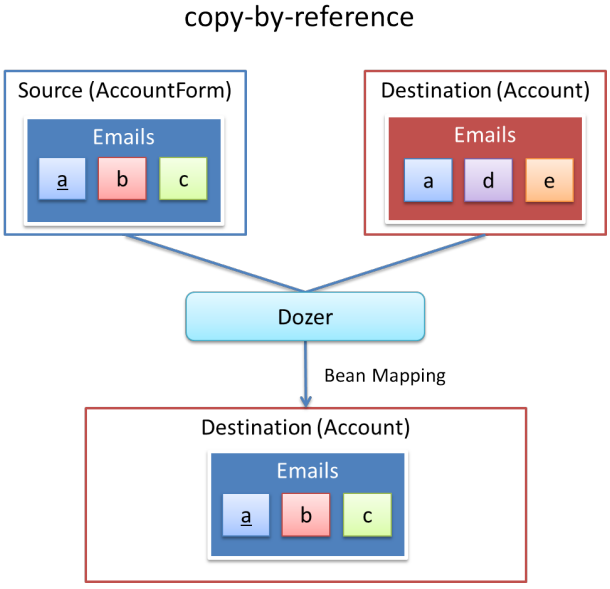
注釈: 「non-cumulative かつ remove-orphans=true」のパターンと「copy-by-reference」のパターンの違いは、Bean 変換後の Collection のコンテナがコピー先のものか、コピー元のものかで異なる点である。

「non-cumulative かつ remove-orphans=true」のパターンの場合は、Bean 変換後の Collection のコンテナはコピー先のものであり、「copy-by-reference」のパターンはコピー元のものである。以下に図で説明する。

- non-cumulative かつ remove-orphans=true



- copy-by-reference



コピー先が **JPA (Hibernate)** のエンティティで **1** 対多や多対多の関連を持つ場合は要注意である。コピー先のエンティティが EntityManager の管理下にある場合、予期せぬトラブルに遭うことがある。例えばコレクションのコンテナが変更されると全件 DELETE + 全件 INSERT の SQL が発行され、「non-cumulative かつ remove-orphans=true」でコピーした場合は変更内容を UPDATE(要素数が異なる場合は DELETE or INSERT)

の SQL が発行される場合がある。どちらが良いかは要件次第である。

警告: マッピング対象の Bean が `String` のコレクションを持つ場合、期待通りの挙動にならないバグがある。

```
StringListSrc src = new StringListSrc;
StringListDest dest = new StringListDest();

List<String> stringsSrc = new ArrayList<String>();
List<String> stringsDest = new ArrayList<String>();

stringsSrc.add("a");
stringsSrc.add("b");
stringsSrc.add("c");

stringsDest.add("a");
stringsDest.add("d");
stringsDest.add("e");

src.setStrings(stringsSrc);
dest.setStrings(stringsDest);

beanMapper.map(src, dest);

System.out.println(dest.getStrings());
```

上記のコードを non-cumulative かつ remove-orphans=true の設定で実行すると、

```
[a, b, c]
```

と出力されることを期待するが、実際には

```
[b, c]
```

と出力され、**重複した String が除かれてしまう。**

copy-by-reference=」true」の設定で実行すると、期待通り

```
[a, b, c]
```

と出力される。

ちなみに: Dozer では、Generics を使用しないリスト間でもマッピングできる。このとき、変換元と変換先に含まれているオブジェクトのデータ型を HINT として指定できる。詳細は、 Dozer の公式マニュアル -Collection and Array Mapping(Using Hints for Collection Mapping)- を参照されたい。

7.7.3 How to extend

カスタムコンバーターの作成

Dozer がサポートしていないデータ型のマッピングでは、同じ型同士の場合も異なる型の場合も、カスタムコンバーター経由でマッピングできる。

- 例：`java.lang.String` <=> `org.joda.time.DateTime`

カスタムコンバーターは、Dozer が提供している `org.dozer.CustomConverter` を実装したクラスである。
カスタムコンバーターの指定は、以下 3 パターンで行える。

- Global Configuration
- クラスレベル
- フィールドレベル

アプリケーション全体で、同様のロジックにより変換を行いたい場合は、Global Configuration を推奨する。

カスタムコンバーターを実装する場合は `org.dozer.DozerConverter` を継承するのが便利である。

```
package com.example.yourproject.common.bean.converter;

import org.dozer.DozerConverter;
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.springframework.util.StringUtils;

public class StringToJodaDateTimeConverter extends
                                        DozerConverter<String, DateTime> { // (1)
    public StringToJodaDateTimeConverter() {
        super(String.class, DateTime.class); // (2)
    }

    @Override
    public DateTime convertTo(String source, DateTime destination) {// (3)
        if (!StringUtils.hasLength(source)) {
            return null;
        }
        DateTimeFormatter formatter = DateTimeFormat
                .forPattern("yyyy-MM-dd HH:mm:ss");
        DateTime dt = formatter.parseDateTime(source);
        return dt;
    }

    @Override
    public String convertFrom(DateTime source, String destination) {// (4)
        if (source == null) {
            return null;
```

```
        }
        return source.toString("yyyy-MM-dd HH:mm:ss");
    }

}
```

項番	説明
(1)	org.dozer.DozerConverter を継承する。
(2)	コンストラクタで対象の 2 つのクラスを設定する。
(3)	String から DateTime の変換ロジックを記述する。本例ではデフォルト Locale を使用する。
(4)	DateTime から String の変換ロジックを記述する。本例ではデフォルト Locale を使用する。

作成したカスタムコンバーターを、マッピングに利用するために定義する必要がある。

dozer-configration-mapping.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">

    <configuration>
        <custom-converters><!-- (1) -->
            <!-- these are always bi-directional -->
            <converter
                type="com.example.yourproject.common.bean.converter.
↪StringToJodaDateTimeConverter"><!-- (2) -->
                <class-a>java.lang.String</class-a><!-- (3) -->
                <class-b>org.joda.time.DateTime</class-b><!-- (4) -->
            </converter>
        </custom-converters>
    </configuration>
    <!-- omitted -->
</mappings>
```

項番	説明
(1)	すべてのカスタムコンバーターが属する、`custom-converters` を定義する。
(2)	個別の変換の行う converter を定義する。converter のタイプに、実装クラスの完全修飾クラス名 (FQCN) を指定する。
(3)	変換元 Bean の完全修飾クラス名 (FQCN)
(4)	変換先 Bean の完全修飾クラス名 (FQCN)

上記のマッピングを行ったことで、アプリケーション全体で、`java.lang.String` <=> `org.joda.time.DateTime` の変換が必要な場合、標準のマッピングではなく、カスタムコンバーター呼び出しでマッピングが行われる。

例 :

変換元の Bean 定義

```
public class Source {
    private int id;
    private String date;
    // omitted setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int id;
    private DateTime date;
    // omitted setter/getter
}
```

マッピング (双方向例)

```
Source source = new Source();
source.setId(1);
source.setDate("2012-08-10 23:12:12");

DateTimeFormatter formatter = DateTimeFormat.forPattern("yyyy-MM-dd HH:mm:ss");
DateTime dt = formatter.parseDateTime(source.getDate());
```

```
// Source to Destination Bean Mapping (String to org.joda.time.DateTime)
Destination destination = dozerBeanMapper.map(source, Destination.class);
assertThat(destination.getId(), is(1));
assertThat(destination.getDate(),is(dt));

// Destination to Source Bean Mapping (org.joda.time.DateTime to String)
dozerBeanMapper.map(destination, source);

assertThat(source.getId(), is(1));
assertThat(source.getDate(),is("2012-08-10 23:12:12"));
```

カスタムコンバーターに関する詳細は、 Dozer の公式マニュアル -Custom Converters- を参照されたい。

注釈: `String` から `java.utl.Date` など標準の日付・時刻オブジェクトへの変換については」文字列から日付・時刻オブジェクトへのマッピング」で述べる。

7.7.4 Appendix

マッピング定義 XML ファイルで指定できるオプションを説明する。

すべてのオプションは、 Dozer の公式マニュアル -Custom Mappings Via Dozer XML Files- で確認できる。

フィールド除外設定 (field-exclude)

Bean を変換する際に、コピーしてほしくないフィールドを除外することができる。

以下のような Bean の変換を考える。

変換元の Bean 定義

```
public class Source {
    private int id;
    private String name;
    private String title;
    // omitted setter/getter
}
```

コピー先の Bean 定義

```
public class Destination {
    private int id;
    private String name;
    private String title;
    // omitted setter/getter
}
```

コピー元の Bean から任意のフィールドをマッピングから除外したい場合は以下のように定義する。

フィールド除外の設定は、マッピング定義 XML ファイルで、以下のように行う。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping>
        <class-a>com.xx.xx.Source</class-a>
        <class-b>com.xx.xx.Destination</class-b>
        <field-exclude><!-- (1) -->
            <a>title</a>
            <b>title</b>
        </field-exclude>
    </mapping>
    <!-- omitted -->
</mappings>
```

項番	説明
(1)	除外したいフィールドを、<field-exclude>要素で設定する。この例の場合、指定した上で map メソッドを実行すると、Source オブジェクトから Destination オブジェクトをコピーする際に、destination の title の値が、上書きされない。

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
source.setTitle("SourceTitle");

Destination destination = new Destination();
destination.setId(2);
destination.setName("DestinationName");
destination.setTitle("DestinationTitle");
beanMapper.map(source, destination);
System.out.println(destination.getId());
System.out.println(destination.getName());
System.out.println(destination.getTitle());
```

上記のコードを実行すると以下のように出力される。

```
1
SourceName
DestinationTitle
```

マッピング後、destination の title の値は、前の状態のままである。

マッピングの特定化 (map-id)

フィールド除外設定 *(field-exclude)* で示したマッピングは、アプリケーション全体で Bean 変換する際に適用される。マッピングの適用範囲を制限 (特定化) したい場合は、以下のように、map-id を指定して定義する。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping map-id="mapidTitleFieldExclude">
        <class-a>com.xx.xx.Source</class-a>
        <class-b>com.xx.xx.Destination</class-b>
        <field-exclude>
            <a>title</a>
            <b>title</b>
        </field-exclude>
    </mapping>
    <!-- omitted -->
</mappings>
```

上記の設定を行うと、map メソッドに map-id(mapidTitleFieldExclude) を渡すことで title のコピーを除外できる。map-id を指定しない場合はこの設定は適用されず、全フィールドがコピーされる。

map メソッドに map-id を渡す例を、以下に示す。

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
source.setTitle("SourceTitle");

Destination destination1 = new Destination();
destination1.setId(2);
destination1.setName("DestinationName");
destination1.setTitle("DestinationTitle");
beanMapper.map(source, destination1); // (1)
System.out.println(destination1.getId());
System.out.println(destination1.getName());
System.out.println(destination1.getTitle());

Destination destination2 = new Destination();
destination2.setId(2);
destination2.setName("DestinationName");
destination2.setTitle("DestinationTitle");
beanMapper.map(source, destination2, "mapidTitleFieldExclude"); // (2)
System.out.println(destination2.getId());
System.out.println(destination2.getName());
System.out.println(destination2.getTitle());
```

項番	説明
(1)	通常のマッピング。
(2)	第三引数に map-id を渡し、特定のマッピングルールを適用する。

上記のコードを実行すると以下のように出力される。

```
1
SourceName
SourceTitle

1
SourceName
DestinationTitle
```

ちなみに: map-id の指定は、mapping 項目だけでなく、フィールドの定義でも行える。詳細は、 Dozer の公式マニュアル -Context Based Mapping- を参照されたい。

注釈: Web アプリケーションにおいて、新規追加・更新両方の操作で同じフォームオブジェクトを使う場合がある。このとき、フォームオブジェクトをドメインオブジェクトにコピー (マップ) する上で、操作によってはコピーしたくないフィールドもある。この場合に、`<field-exclude>`を使用する。

- 例：新規作成のフォームでは userId を含むが、更新用のフォームでは userId を含まない。

この場合に同じフォームオブジェクトを使用すると、更新時に userId に null が設定される。コピー先のオブジェクトを DB から取得して、フォームオブジェクトをそのままコピーすると、コピー先の userId まで null となる。これを回避するために、更新用の map-id を用意し、更新時は userId に対して、フィールド除外の設定を行う。

コピー元の null・空フィールドを除外する設定 (map-null, map-empty)

コピー元の Bean のフィールドが、null の場合、あるいは空の場合に、マッピングから除外することができる。以下のように、マッピング定義 XML ファイルに設定する。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping map-null="false" map-empty-string="false"><!-- (1) -->
        <class-a>com.xx.xx.Source</class-a>
        <class-b>com.xx.xx.Destination</class-b>
    </mapping>
    <!-- omitted -->
</mappings>
```

項番	説明
(1)	コピー元の Bean のフィールドが null の場合にマッピングから除外したい場合は map-null 属性に false を設定する。デフォルト値は true である。 空の場合に、マッピングから除外したい場合は map-empty-string 属性に false を設定する。デフォルト値は true である。

変換元の Bean 定義

```
public class Source {
    private int id;
    private String name;
    private String title;
    // omitted setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int id;
    private String name;
    private String title;
    // omitted setter/getter
}
```

マッピング例

```
Source source = new Source();
source.setId(1);
source.setName(null);
source.setTitle("");

Destination destination = new Destination();
destination.setId(2);
destination.setName("DestinationName");
destination.setTitle("DestinationTitle");
beanMapper.map(source, destination);
System.out.println(destination.getId());
System.out.println(destination.getName());
System.out.println(destination.getTitle());
```

上記のコードを実行すると以下のように出力される。

```
1
DestinationName
DestinationTitle
```

コピー元 Bean の name と title フィールドは、null、あるいは空で、マッピングから除外されている。

文字列から日付・時刻オブジェクトへのマッピング

コピー元の文字列型のフィールドを、コピー先の日付・時刻系のフィールドにマッピングできる。

以下 6 種類の変換をサポートしている。

日付・時刻系

- java.lang.String **<=>** java.util.Date
- java.lang.String **<=>** java.util.Calendar
- java.lang.String **<=>** java.util.GregorianCalendar
- java.lang.String **<=>** java.sql.Timestamp

日付のみ

- java.lang.String **<=>** java.sql.Date

時刻のみ

- java.lang.String **<=>** java.sql.Time

日付・時刻系の変換は、以下のように行う。

例として、java.util.Date への変換を説明する。

java.util.Calendar,java.util.GregorianCalendar,java.sql.Timestamp も同じ方法で行える。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping>
        <class-a>com.xx.xx.Source</class-a>
        <class-b>com.xx.xx.Destination</class-b>
        <field>
            <a date-format="yyyy-MM-dd HH:mm:ss:SS">date</a><!-- (1) -->
            <b>date</b>
        </field>
    </mapping>
    <!-- omitted -->
</mappings>
```

項番	説明
(1)	コピー元のフィールド名と日付形式を指定する。

変換元の Bean 定義

```
public class Source {
    private String date;
    // omitted setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private Date date;
    // omitted setter/getter
}
```

マッピング

```
Source source = new Source();
source.setDate("2013-10-10 11:11:11.111");
Destination destination = beanMapper.map(source, Destination.class);
assert(destination.getDate().equals(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS
↪").parse("2013-10-10 11:11:11.111")));
```

日付形式は、個別のマッピング定義毎に設定するよりも、プロジェクトで一括して設定したいケースが多い。

その場合は Dozer の Global configuration ファイルで設定することを推奨する。

その場合、アプリケーション全体のマッピングで設定された日付形式が、適用される。

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance"
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <configuration>
        <date-format>yyyy-MM-dd HH:mm:ss.SSS</date-format>
        <!-- omitted other configuration -->
    </configuration>
    <!-- omitted -->
</mappings>
```

ファイル名には制限はないが、src/main/resources/META-INF/dozer/dozer-configration-mapping.xml を推奨する。

dozer-configration-mapping.xml 内の設定の範囲は、この設定ファイル内でアプリケーション全体に影響を与える、Global Configuration を行えばよい。

設定可能な項目の詳細について、 Dozer の公式マニュアル -Global Configuration- を参照されたい。

マッピングのエラー

マッピング中にマッピング処理が失敗したら、`org.dozer.MappingException`(実行時例外) がスローされる。

`MappingException` がスローされる代表的な例を、以下に挙げる。

- `map` メソッドに存在しない map-id が渡されている。
- `map` メソッドに存在する map-id を渡したが、マップ処理に渡したソース・ターゲット型は、その map-id に指定している定義とは異なる。
- Dozer がサポートしていない変換の場合、かつ、その変換用のカスタムコンバーターも存在しない場合。

これらは通常プログラムバグであるので、`map` メソッドの呼び出しの部分を正しく修正する必要がある。

第 8 章

メッセージ連携

8.1 E-mail 送信 (SMTP)

8.1.1 Overview

本節では、SMTP による E-mail の送信方法について説明する。

本ガイドラインでは、JavaMail の API と Spring Framework から提供されている Mail 連携用コンポーネントを利用することを前提としている。

注釈: 説明の対象としているのはメールを送信する部分のみである。メール送信に係る処理方式については言及していない。（処理方式において一例を紹介している。）

JavaMail について

JavaMail は、Java でメールの送受信を行うための API を提供している。Java EE に含まれるものであるが、Java SE でも追加パッケージとして利用可能である。JavaMail を利用することで、メール機能を容易に Java アプリケーションに組み込むことができる。

なお、本ガイドラインでは、Spring Framework の Mail 連携用コンポーネントを利用する前提であるため、JavaMail の API についての詳細には触れていない。JavaMail の API 仕様については、JavaMail API Design Specification を参照されたい。

注釈: メールセッション

メールセッション（Session）は、メールサーバに接続する際に必要となる情報を管理する。

メールセッションを取得するには以下のような方法がある。

- 典型的なエンタープライズアプリケーションにおいては、Java EE のコンテナで管理されたメールセッションを JNDI 経由で取得する。
- Tomcat の場合はリソースファクトリで定義したメールセッションを JNDI 経由で取得する。
- static ファクトリメソッドを利用して Bean 定義したメールセッションを DI コンテナから取得する。
- Java ソースから直接 `Session` の static ファクトリメソッドを利用して取得する。

なお、後述する Spring の `JavaMailSenderImpl` を使用すると、メールセッションを直接扱わずにメールサーバと接続することも可能である。

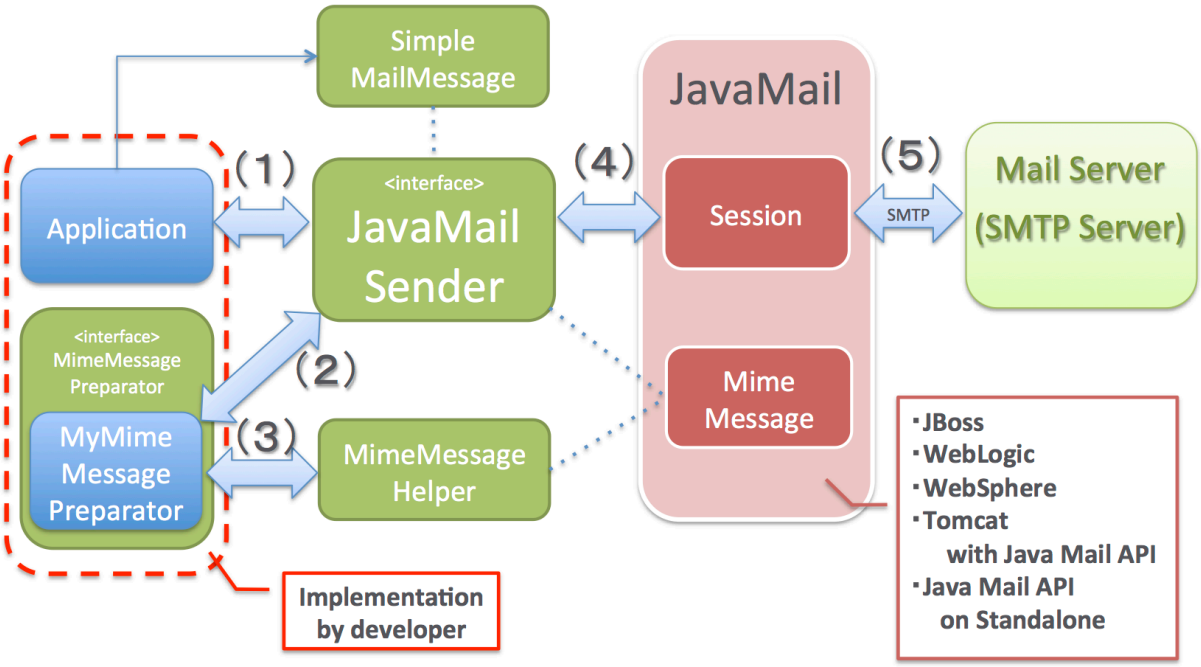
本ガイドラインでは、以下の二つの方法による実装例を紹介する。

- メールセッションを JNDI 経由で取得する方法
- セッションを直接扱わずに `JavaMailSenderImpl` のプロパティに接続情報を指定する方法

Spring Framework の Mail 連携用コンポーネントについて

Spring Framework はメール送信を行うためのコンポーネント（`org.springframework.mail` パッケージ）を提供している。このパッケージに含まれるコンポーネントはメール送信に係る詳細なロジックを隠蔽し、低レベルの API ハンドリング (JavaMail の API 呼び出し) を代行する。

具体的な実装方法の説明を行う前に、Spring Framework が提供するメール送信用のコンポーネントがどのようにメールを送信しているかを説明する。



項番	コンポーネント	説明
(1)	アプリケーション	JavaMailSender のメソッドを呼び出し、メールの送信依頼を行う。 * 単純なメッセージを送信する場合は、SimpleMailMessage を生成し宛先や本文を設定することでメールを送信することもできる。
(2)	JavaMailSender	アプリケーションから指定された MimeMessagePreparator(JavaMail の MimeMessage を作成するためのコールバックインターフェース) を呼び出し、メール送信用のメッセージ (MimeMessage) の作成依頼を行う。 * SimpleMailMessage を使用してメッセージを送信する場合はこの処理は呼びだされない。
(3)	アプリケーション (MimeMessagePreparator)	MimeMessageHelper のメソッドを利用して、メール送信用のメッセージ (MimeMessage) を作成する。 * SimpleMailMessage を使用してメッセージを送信する場合はこの処理は呼びだされない。
(4)	JavaMailSender	JavaMail の API を使用して、メールの送信依頼を行う。
(5)	JavaMail	メールサーバへメッセージを送信する。

本ガイドラインでは、以下のインタフェースやクラスを使用してメール送信処理を実装する方法について説明する。

- `JavaMailSender`

  JavaMail 用のメール送信インターフェース。

  JavaMail の MimeMessage と Spring の `SimpleMailMessage` の両方に対応している。

  また、JavaMail の `Session` の管理は `JavaMailSender` の実装クラスによって行われるため、メール送信処理をコーディングする際に `Session` を直接扱う必要がない。

- `JavaMailSenderImpl`

  `JavaMailSender` インタフェースの実装クラス。

  このクラスでは、設定済みの `Session` を DI する方法と、プロパティに指定した接続情報から `Session` を作成する方法をサポートしている。

- `MimeMessagePreparator`

  JavaMail の `MimeMessage` を作成するためのコールバックインターフェース。

  `JavaMailSender` の `send` メソッド内から呼び出される。

  `MimeMessagePreparator` の `prepare` メソッドで発生した例外は `MailPreparationException`（実行時例外）にラップされ再スローされる。

- `MimeMessageHelper`

  JavaMail の `MimeMessage` の作成を容易にするためのヘルパークラス。

  `MimeMessageHelper` には、`MimeMessage` に値を設定するための便利なメソッドがいくつも用意されている。

- `SimpleMailMessage`

  単純なメールメッセージを作成するためのクラス。

  英文のプレーンテキストメールを作成する際に使用できる。

  UTF-8 等の特定のエンコード指定、HTML メールや添付ファイル付きメールの送信、あるいはメールアドレスに個人名を付随させるといったリッチなメッセージの作成を行う際は、JavaMail の `MimeMessage` を使用する必要がある。

8.1.2 How to use

依存ライブラリについて

Spring Framework の Mail 連携用コンポーネントを利用する場合、以下のライブラリが追加で必要となる。

- JavaMail

上記ライブラリに対する依存関係を `pom.xml` に追加する。

マルチプロジェクト構成の場合は、domain プロジェクトの `pom.xml`(`projectName-domain/pom.xml`) に追加する。

```
<dependencies>

    <!-- (1) -->
    <dependency>
        <groupId>com.sun.mail</groupId>
        <artifactId>javax.mail</artifactId>
    </dependency>

</dependencies>
```

項番	説明
(1)	JavaMail のライブラリを dependencies に追加する。 アプリケーションサーバ提供のメールセッションを使用する場合、<scope>を provided に設定する。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

JavaMailSender の設定方法

JavaMailSender を DI するための Bean 定義を行う。

注釈: マルチプロジェクト構成の場合は、env プロジェクトの projectName-env.xml に設定することを推奨する。なお、本ガイドラインでは、マルチプロジェクト構成を採用することを推奨している。

アプリケーションサーバ提供のメールセッションを使用する場合

アプリケーションサーバ提供のメールセッションを使用する場合の設定例を以下に示す。

表 8.2　アプリケーションサーバから提供されているメールセッション

項番	アプリケーションサーバ	参照ページ
1.	Apache Tomcat 9.0	Apache Tomcat 9.0 User Guide(JNDI Resources HOW-TO)(JavaMail Sessions) を参照されたい。
2.	Apache Tomcat 8.5	Apache Tomcat 8.5 User Guide(JNDI Resources HOW-TO)(JavaMail Sessions) を参照されたい。
3.	Oracle WebLogic Server 12c	Oracle WebLogic Server 12.2.1.3 Documentation を参照されたい。
4.	IBM WebSphere Application Server Version 9.0	WebSphere Application Server Version 9.0.0 documentation を参照されたい。
5.	Red Hat JBoss Enterprise Application Platform Version 7.2	JBoss Enterprise Application Platform 7.2 Product Documentation を参照されたい。
6.	Red Hat JBoss Enterprise Application Platform Version 6.4	JBoss Enterprise Application Platform 6.4 Product Documentation を参照されたい。

JNDI 経由で取得したメールセッションを Bean として登録するための設定を行う。

```
<jee:jndi-lookup id="mailSession" jndi-name="mail/Session" /> <!-- (1) -->
```

項番	説明
(1)	<jee:jndi-lookup>要素の jndi-name 属性に、アプリケーションサーバ提供のメールセッションの JNDI 名を指定する。

次に、JavaMailSender を Bean 定義する。

```
<!-- (1) -->
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="session" ref="mailSession" /> <!-- (2) -->
</bean>
```

項番	説明
(1)	JavaMailSenderImpl を Bean 定義する。
(2)	session プロパティに設定済みのメールセッションの Bean を指定する。

アプリケーションサーバ提供のメールセッションを使用しない場合（認証なし）

認証が必要ない場合の設定例を以下に示す。

JavaMailSender を Bean 定義する。

```
<!-- (1) -->
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="${mail.smtp.host}"/> <!-- (2) -->
    <property name="port" value="${mail.smtp.port}"/> <!-- (3) -->
</bean>
```

項番	説明
(1)	JavaMailSenderImpl を Bean 定義する。
(2)	host プロパティに SMTP サーバのホスト名を指定する。 この例では、プロパティファイルで定義した値（キー「mail.smtp.host」に対する値）を設定している。
(3)	port プロパティに SMTP サーバのポート番号を指定する。 この例では、プロパティファイルで定義した値（キー「mail.smtp.port」に対する値）を設定している。

注釈: プロパティファイルについての詳細は、プロパティ管理 を参照されたい。

アプリケーションサーバ提供のメールセッションを使用しない場合（認証あり）

認証が必要な場合の設定例を以下に示す。

JavaMailSender を Bean 定義する。

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="${mail.smtp.host}"/>
    <property name="port" value="${mail.smtp.port}"/>
    <property name="username" value="${mail.smtp.user}"/> <!-- (1) -->
    <property name="password" value="${mail.smtp.password}"/> <!-- (2) -->
    <property name="javaMailProperties">
        <props>
            <prop key="mail.smtp.auth">true</prop> <!-- (3) -->
        </props>
    </property>
</bean>
```

項番	説明
(1)	username プロパティに SMTP サーバのユーザ名を指定する。 この例では、プロパティファイルで定義した値（キー「mail.smtp.user」に対する値）を設定している。
(2)	password プロパティに SMTP サーバのパスワードを指定する。 この例では、プロパティファイルで定義した値（キー「mail.smtp.password」に対する値）を設定している。
(3)	javaMailProperties プロパティにキー「mail.smtp.auth」として true を設定する。

注釈: プロパティファイルについての詳細は、プロパティ管理 を参照されたい。

ちなみに: TLS による接続が必要な場合、javaMailProperties プロパティにキー「mail.smtp.starttls.enable」として true を設定する。なお、左記のとおり指定した場合でも SMTP サーバが STARTTLS をサポートしていない場合は平文による通信が行われる。必要に応じて javaMailProperties プロパティにキー「mail.smtp.starttls.required」として true を設定することで、STARTTLS を利用できない場合にエラーとすることも可能である。

SimpleMailMessage によるメール送信方法

英文のプレーンテキストメール（エンコードの指定や添付ファイル等が不要なメール）を送信する場合は、Spring が提供している SimpleMailMessage クラスを使用する。

以下に、SimpleMailMessage クラスを使用したメール送信方法を説明する。

Bean 定義例

```
<!-- (1) -->
<bean id="templateMessage" class="org.springframework.mail.SimpleMailMessage">
    <property name="from" value="info@example.com" /> <!-- (2) -->
    <property name="subject" value="Registration confirmation." /> <!-- (3) -->
</bean>
```

項番	説明
(1)	テンプレートとして SimpleMailMessage を Bean 定義する。 テンプレートの SimpleMailMessage を利用するのは必須ではないが、メールメッセージで固定的な箇所（例えば送信元メールアドレス等）をテンプレート化しておくことで、メールメッセージ作成時に個別に設定する必要がなくなる。
(2)	from プロパティに From ヘッダの内容を指定する。
(3)	subject プロパティに Subject ヘッダの内容を指定する。

Java クラスの実装例

```
@Inject
JavaMailSender mailSender; // (1)

@Inject
SimpleMailMessage templateMessage; // (2)

public void register(User user) {
    // omitted

    // (3)
    SimpleMailMessage message = new SimpleMailMessage(templateMessage);
    message.setTo(user.getEmailAddress());
    String text = "Hi "
            + user.getUserName()
            + ", welcome to EXAMPLE.COM!\r\n"
            + "If you were not an intended recipient, Please notify the sender.";
```

```
    message.setText(text);
    mailSender.send(message);

    // omitted
}
```

項番	説明
(1)	JavaMailSender をインジェクションする。
(2)	テンプレートとして Bean 定義した SimpleMailMessage をインジェクションする。
(3)	テンプレートの Bean を利用して SimpleMailMessage のインスタンスを生成し、To ヘッダと本文を設定して送信する。

注釈:

表 8.3 SimpleMailMessage で設定可能なプロパティ

項番	プロパティ	説明
1.	`from`	From ヘッダを設定する。
2.	`to`	To ヘッダを設定する。
3.	`cc`	Cc ヘッダを設定する。
4.	`bcc`	Bcc ヘッダを設定する。
5.	`subject`	Subject ヘッダを設定する。
6.	`replyTo`	Reply-To ヘッダを設定する。
7.	`sentDate`	Date ヘッダを設定する。 なお、明示的に設定しない場合は送信時にシステム時刻（`new Date()`）が自動設定される。
8.	`text`	本文を設定する。

To、Cc、Bcc に複数の宛先を設定する場合は配列にして設定する。

警告: メールヘッダを設定する場合、メールヘッダ・インジェクションに対する考慮が必要となる。詳細はメールヘッダ・インジェクション対策を参照されたい。

MimeMessage によるメール送信方法

英文以外のメールや HTML メール、添付ファイルの送信を行う場合、javax.mail.internet.MimeMessage クラスを使用する。本ガイドラインでは MimeMessageHelper クラスを使用して MimeMessage を作成する方法を推奨している。

本項では、MimeMessageHelper クラスを使用した以下のメール送信方法を説明する。

- テキストメールの送信
- *HTML* メールの送信
- 添付ファイル付きメールの送信
- インラインリソース付きメールの送信

テキストメールの送信

MimeMessageHelper クラスを使用して、テキストメールを送信する実装例を以下に示す。

Java クラスの実装例

```
@Inject
JavaMailSender mailSender; // (1)

public void register(User user) {
    // omitted

    // (2)
    mailSender.send(new MimeMessagePreparator() {

        @Override
        public void prepare(MimeMessage mimeMessage) throws Exception {
            MimeMessageHelper helper = new MimeMessageHelper(mimeMessage,
                    StandardCharsets.UTF_8.name()); // (3)
            helper.setFrom("EXAMPLE.COM <info@example.com>"); // (4)
            helper.setTo(user.getEmailAddress()); // (5)
            helper.setSubject("Registration confirmation."); // (6)
            String text = "Hi "
                    + user.getUserName()
                    + ", welcome to EXAMPLE.COM!\r\n"
                    + "If you were not an intended recipient, Please notify the␣
↪sender.";
            helper.setText(text); // (7)
        }
    });

    // omitted
}
```

項番	説明
(1)	`JavaMailSender` をインジェクションする。
(2)	`JavaMailSender` の `send` メソッドを利用してメールを送信する。 引数には `MimeMessagePreparator` を実装した匿名内部クラスを定義する。
(3)	文字コードを指定して、`MimeMessageHelper` のインスタンスを生成する。 この例では、文字コードに UTF-8 を指定している。
(4)	From ヘッダの内容を設定する。 この例では、」名前 <アドレス>」の形式で設定している。
(5)	To ヘッダの内容を設定する。
(6)	Subject ヘッダの内容を設定する。
(7)	本文の内容を設定する。

警告: メールヘッダを設定する場合、メールヘッダ・インジェクションに対する考慮が必要となる。詳細はメールヘッダ・インジェクション対策を参照されたい。

注釈: 日本語のメールを送信する際、UTF-8 をサポートしていないメールクライアントもサポートする必要がある場合はエンコードに ISO-2022-JP を利用することも考えられる。エンコードに ISO-2022-JP を利用する際に考慮すべき事項について、*ISO-2022-JP のエンコードについての考慮*を参照されたい。

HTML メールの送信

`MimeMessageHelper` クラスを使用して、HTML メールを送信する実装例を以下に示す。

Java クラスの実装例

```
@Inject
JavaMailSender mailSender; // (1)

public void register(User user) {
    // omitted

    // (2)
    mailSender.send(new MimeMessagePreparator() {

        @Override
        public void prepare(MimeMessage mimeMessage) throws Exception {
            MimeMessageHelper helper = new MimeMessageHelper(mimeMessage,
                    StandardCharsets.UTF_8.name()); // (3)
            helper.setFrom("EXAMPLE.COM <info@example.com>"); // (4)
            helper.setTo(user.getEmailAddress()); // (5)
            helper.setSubject("Registration confirmation."); // (6)
            String text = "<html><body><h3>Hi "
                    + user.getUserName()
                    + ", welcome to EXAMPLE.COM!</h3>"
                    + "If you were not an intended recipient, Please notify the␣
↪sender.</body></html>";
            helper.setText(text, true); // (7)
        }
    });

    // omitted
}
```

項番	説明
(1)	JavaMailSender をインジェクションする。
(2)	JavaMailSender の send メソッドを利用してメールを送信する。 引数には MimeMessagePreparator を実装した匿名内部クラスを定義する。
(3)	文字コードを指定して、MimeMessageHelper のインスタンスを生成する。 この例では、文字コードに UTF-8 を指定している。
(4)	From ヘッダの内容を設定する。 この例では、」名前 <アドレス>」の形式で設定している。
(5)	To ヘッダの内容を設定する。
(6)	Subject ヘッダの内容を設定する。
(7)	本文の内容を設定する。setText メソッドの第二引数に true を指定することで、Content-Type が text/html になる。

警告: メール本文の HTML を生成する際に外部から入力された値を使用する場合は XSS 攻撃への対策を行うこと。

添付ファイル付きメールの送信

MimeMessageHelper クラスを使用して、添付ファイル付きメールを送信する実装例を以下に示す。

Java クラスの実装例

```
@Inject
JavaMailSender mailSender; // (1)
```

```
public void register(User user) {
    // omitted

    // (2)
    mailSender.send(new MimeMessagePreparator() {

        @Override
        public void prepare(MimeMessage mimeMessage) throws Exception {
            MimeMessageHelper helper = new MimeMessageHelper(mimeMessage,
                    true, StandardCharsets.UTF_8.name()); // (3)
            helper.setFrom("EXAMPLE.COM <info@example.com>"); // (4)
            helper.setTo(user.getEmailAddress()); // (5)
            helper.setSubject("Registration confirmation."); // (6)
            String text = "Hi "
                    + user.getUserName()
                    + ", welcome to EXAMPLE.COM!\r\n"
                    + "Please find attached the file.\r\n\r\n"
                    + "If you were not an intended recipient, Please notify the␣
↪sender.";
            helper.setText(text); // (7)
            ClassPathResource file = new ClassPathResource("doc/quickstart.pdf");
            helper.addAttachment("QuickStart.pdf", file); // (8)
        }
    });

    // omitted
}
```

項番	説明
(1)	JavaMailSender をインジェクションする。
(2)	JavaMailSender の send メソッドを利用してメールを送信する。 引数には MimeMessagePreparator を実装した匿名内部クラスを定義する。
(3)	文字コードを指定して、MimeMessageHelper のインスタンスを生成する。 この例では、文字コードに UTF-8 を指定している。 MimeMessageHelper のコンストラクタの第二引数に true を指定することで、マルチパートモード（デフォルトの MULTIPART_MODE_MIXED_RELATED）になる。
(4)	From ヘッダの内容を設定する。
(5)	To ヘッダの内容を設定する。
(6)	Subject ヘッダの内容を設定する。
(7)	本文の内容を設定する。
(8)	添付ファイル名を指定して添付するファイルを設定する。 この例では、QuickStart.pdf というファイル名で、クラスパス上にある doc/quickstart.pdf というファイルを添付している。

インラインリソース付きメールの送信

MimeMessageHelper クラスを使用して、インラインリソース付きメールを送信する実装例を以下に示す。

Java クラスの実装例

```
@Inject
JavaMailSender mailSender; // (1)

public void register(User user) {
    // omitted

    // (2)
    mailSender.send(new MimeMessagePreparator() {

        @Override
        public void prepare(MimeMessage mimeMessage) throws Exception {
            MimeMessageHelper helper = new MimeMessageHelper(mimeMessage,
                    true, StandardCharsets.UTF_8.name()); // (3)
            helper.setFrom("EXAMPLE.COM <info@example.com>"); // (4)
            helper.setTo(user.getEmailAddress()); // (5)
            helper.setSubject("Registration confirmation."); // (6)
            String cid = "identifier1234";
            String text = "<html><body><img src='cid:"
                    + cid
                    + "' /><h3>Hi "
                    + user.getUserName()
                    + ", welcome to EXAMPLE.COM!\r\n</h3>"
                    + "If you were not an intended recipient, Please notify the␣
↪sender.</body></html>";
            helper.setText(text, true); // (7)
            ClassPathResource res = new ClassPathResource("image/logo.jpg");
            helper.addInline(cid, res); // (8)
        }
    });

    // omitted
}
```

項番	説明
(1)	`JavaMailSender` をインジェクションする。
(2)	`JavaMailSender` の `send` メソッドを利用してメールを送信する。 引数には `MimeMessagePreparator` を実装した匿名内部クラスを定義する。
(3)	文字コードを指定して、`MimeMessageHelper` のインスタンスを生成する。 この例では、文字コードに UTF-8 を指定している。 `MimeMessageHelper` のコンストラクタの第二引数に `true` を指定することで、マルチパートモードになる。
(4)	From ヘッダの内容を設定する。
(5)	To ヘッダの内容を設定する。
(6)	Subject ヘッダの内容を設定する。
(7)	本文の内容を設定する。`setText` メソッドの第二引数に `true` を指定することで、Content-Type が text/html になる。
(8)	インラインリソースのコンテンツ ID を指定してインラインリソースを設定する。 この例では、`identifier1234` というコンテンツ ID で、クラスパス上にある `image/logo.jpg` というファイルを設定している。

注釈: `addInline` メソッドは、`setText` メソッドの後に呼び出すこと。そうしないと、メールクライアントがインラインリソースを正しく参照できないことがある。

メール送信時の例外について

JavaMailSender の send メソッドを利用してメール送信を行う際に発生する例外は org.springframework.mail.MailException を継承した例外である。MailException を継承した例外クラスと、それぞれの例外の発生条件について、以下の表に示す。

表 8.4　メール送信時の例外

項番	例外クラス	発生条件
1.	MailAuthenticationException	認証失敗時に発生する。
2.	MailParseException	メールメッセージのプロパティに不正な値が設定されている場合に発生する。
3.	MailPreparationException	メールメッセージを作成中に想定外のエラーが起きた場合に発生する。想定外のエラーとしては、例えばテンプレートライブラリで発生するエラーといったものがある。 MimeMessagePreparator で発生した例外が MailPreparationException にラップされてスローされる。
4.	MailSendException	メールの送信エラーが起きた場合に発生する。

注釈: 特定の例外に対するエラー画面遷移については、 例外ハンドリング を参照されたい。

8.1.3 How to extend

テンプレートを使用したメール本文の作成方法

上で示した実装例のように Java ソースでメール本文を直接組み立てるのは、以下の理由から推奨しない。

- メール本文を Java ソースで組み立てるのは可読性が悪くエラーを作りやすい。
- 表示ロジックとビジネスロジックの境界が曖昧となる。
- メール本文のデザインを変更するために、Java ソースの修正、コンパイル、デプロイが必要になる。

よって、メール本文のデザインを定義するためにテンプレートライブラリを使用することを推奨する。特にメール本文が複雑になるような場合はテンプレートライブラリを使用すべきである。

FreeMarker を使用したメール本文の作成

本ガイドラインでは、テンプレートライブラリとして FreeMarker を使用する方法について説明する。

- FreeMarker を使用するために、依存ライブラリを設定する。

pom.xml の設定例

```
<dependencies>

    <!-- (1) -->
    <dependency>
        <groupId>org.freemarker</groupId>
        <artifactId>freemarker</artifactId>
    </dependency>

</dependencies>
```

項番	説明
(1)	FreeMarker のライブラリを dependencies に追加する。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

- `freemarker.template.Configuration` を生成するための FactoryBean を Bean 定義する。

Bean 定義ファイルの設定例

```
<!-- (1) -->
<bean id="freemarkerConfiguration"
    class="org.springframework.ui.freemarker.
↪FreeMarkerConfigurationFactoryBean">
    <property name="templateLoaderPath" value="classpath:/META-INF/
↪freemarker/" /> <!-- (2) -->
```

```
    <property name="defaultEncoding" value="UTF-8" /> <!-- (3) -->
</bean>
```

項番	説明
(1)	FreeMarkerConfigurationFactoryBean を Bean 定義する。
(2)	templateLoaderPath プロパティにテンプレートファイルの格納された場所を指定する。 この例では、クラスパス上にある META-INF/freemarker/ディレクトリを設定している。
(3)	defaultEncoding プロパティにデフォルトのエンコードを指定する。 この例では、UTF-8 を設定している。

注釈: 上記以外の設定については、FreeMarkerConfigurationFactoryBean の JavaDoc を参照されたい。また、FreeMarker 自体の設定については、FreeMarker Manual (Programmer's Guide / The Configuration) を参照されたい。

- メール本文のテンプレートファイルを作成する。

テンプレートファイルの設定例

```
<#escape x as x?html> <#-- (1) -->
<html>
    <body>
        <h3>Hi ${userName}, welcome to TERASOLUNA.ORG!</h3> <#-- (2) -->

        <div>
            If you were not an intended recipient, Please notify the␣
↪sender.
        </div>
    </body>
</html>
</#escape>
```

項番	説明
(1)	XSS 攻撃への対策として HTML エスケープを行うように設定している。
(2)	データモデルに設定された `userName` の値を埋め込む。

注釈: テンプレート言語（FTL）の詳細については、FreeMarker Manual (Template Language Reference) を参照されたい。

- テンプレートを使用してメール本文を生成し、メール送信する。

Java クラスの実装例

```
@Inject
JavaMailSender mailSender;

@Inject
Configuration freemarkerConfiguration; // (1)

public void register(User user) {
    // omitted

    mailSender.send(new MimeMessagePreparator() {

        @Override
        public void prepare(MimeMessage mimeMessage) throws Exception {
            MimeMessageHelper helper = new MimeMessageHelper(mimeMessage,
                    StandardCharsets.UTF_8.name());
            helper.setFrom("EXAMPLE.COM <info@example.com>");
            helper.setTo(user.getEmailAddress());
            helper.setSubject("Registration confirmation.");
            Template template = freemarkerConfiguration
                    .getTemplate("registration-confirmation.ftl"); // (2)
            String text = FreeMarkerTemplateUtils
                    .processTemplateIntoString(template, user); // (3)
            helper.setText(text, true);
        }
    });

    // omitted
}
```

項番	説明
(1)	Configuration をインジェクションする。
(2)	Configuration の getTemplate メソッドを利用して Template を取得する。 この例では、テンプレートファイルとして」registration-confirmation.ftl」を指定している。
(3)	取得した Template をもとに、org.springframework.ui.freemarker.FreeMarkerTemplateUtils の processTemplateIntoString メソッドを利用してテンプレートから文字列を生成する。 この例では、データモデルとして userName プロパティを持つ User オブジェクト（JavaBeans）を指定している。これにより、テンプレートファイルの${userName}の箇所に userName プロパティの値が埋め込まれる。

8.1.4 Appendix

ISO-2022-JP のエンコードについての考慮

日本語のメールを送信する際、送信したメールを受信するメールクライアントを限定できない場合は、エンコードに ISO-2022-JP を利用することを検討する必要がある。この理由としては、レガシーなメールクライアントが UTF-8 に対応していない場合を考慮するためである。

MS932 で入力された文字列に対し、エンコードに ISO-2022-JP をはじめとする JIS X 0208 の文字集合をベースとしたエンコードを設定した場合、以下の表に記載する 7 文字において文字化けが発生する。

変換前			変換後		
MS932 入力文字	入力値 （SJIS）	Unicode （UTF-16）	Unicode （UTF-16）	ISO-2022-JP （JIS）	JIS X 0208 代替文字
―（全角ハイフン）	815D	U+2015	U+2014	213E	—（EM ダッシュ）
－（ハイフンマイナス）	817C	U+FF0D	U+2212	215D	－（全角マイナス）
～（全角チルド）	8160	U+FF5E	U+301C	2141	～（波ダッシュ）
∥（平行記号）	8161	U+2225	U+2016	2142	‖（双柱）
￠（全角セント記号）	8191	U+FFE0	U+00A2	2171	¢（セント記号）
￡（全角ポンド記号）	8192	U+FFE1	U+00A3	2172	£（ポンド記号）
￢（全角否定記号）	81CA	U+FFE2	U+00AC	224C	¬（否定記号）

この問題は、Unicode を介して文字コード変換を行う際に、MS932 に有り JIS X 0208 に無い文字が存在するためであり、文字化けを回避するためには、文字化けする文字について代替文字に文字コードを置き換えるなどの対処を行う必要がある。なお、後述する x-windows-iso2022jp を使用する場合、変換処理は不要である。

以下に、変換処理の実装例を示す。

```
public static String convertISO2022JPCharacters(String targetStr) {

    if (targetStr == null) {
        return null;
    }
```

```
    char[] ch = targetStr.toCharArray();

    for (int i = 0; i < ch.length; i++) {
        switch (ch[i]) {

        // '―'（全角ハイフン）
        case '\u2015':
            ch[i] = '\u2014';
            break;
        // 'ー'（全角マイナス）
        case '\uff0d':
            ch[i] = '\u2212';
            break;
        // '～'（波ダッシュ）
        case '\uff5e':
            ch[i] = '\u301c';
            break;
        // '∥'（双柱）
        case '\u2225':
            ch[i] = '\u2016';
            break;
        // '¢'（セント記号）
        case '\uffe0':
            ch[i] = '\u00A2';
            break;
        // '£'（ポンド記号）
        case '\uffe1':
            ch[i] = '\u00A3';
            break;
        // '¬'（否定記号）
        case '\uffe2':
            ch[i] = '\u00AC';
            break;
        default:
            break;
        }
    }

    return String.valueOf(ch);
}
```

注釈: Unicode へのマッピング時の問題であるため、入力値の文字コードに依らず変換は必要である。変換対象となるのは日本語を含む文字列が設定される可能性のあるヘッダおよび本文の文字列である。日本語を含む可能性があり一般的によく使われると考えられるヘッダとしては、From、To、Cc、Bcc、Reply-To、Subject が挙げられる。

また、エンコードに ISO-2022-JP を設定する場合、以下のような範囲外となる拡張文字が文字化けする。

これらの文字は本来使用すべきではない。もし、これらの文字を使用する必要がある場合、JVM の起動



図 8.1　図-範囲外となる拡張文字の例

オプションとして以下のように設定することで ISO-2022-JP のエンコードが指定された場合に x-windows-iso2022jp でマッピングするように差し替えることが可能である。

```
-Dsun.nio.cs.map=x-windows-iso2022jp/ISO-2022-JP
```

警告: x-windows-iso2022jp は ISO-2022-JP の規格と異なるマッピング（MS932 ベース）を含む ISO-2022-JP 実装である。メールヘッダで ISO-2022-JP のエンコードが指定された場合に範囲外の拡張文字を扱えるような実装となっているかはメールクライアントに依存する。このため、x-windows-iso2022jp を使用してマッピングした場合でも、すべてのメールクライアントで確実に文字化けしないことが保証されるわけではない。

拡張文字を代替文字に変換してもよい場合、前述した 7 文字と同様にアプリケーションで独自に変換を行う方法も合わせて検討されたい。

JavaMail でマルチバイト文字を使用する際の注意点

JavaMail では、送信するメールの本文の終端がマルチバイト文字で終わっていると、終端に余計な文字（「?」や「w)」等）が出力される場合がある。本事象は以下の方法で回避できる。

- メール本文の終端文字を半角文字にする
- メール本文の終端を改行コード（CRLF）にする

メールヘッダ・インジェクション対策

メールヘッダ・インジェクション攻撃が成功すると、本来意図していない宛先にメール送信され、迷惑メール送信の踏み台に悪用される可能性がある。メールヘッダ（Subject 等）の内容に外部から入力された文字列を利用する場合、メールヘッダ・インジェクション攻撃への対策が必要となる。

例えば、`MimeMessageHelper` の `setSubject` メソッドで以下の文字列を設定すると、Bcc ヘッダを追加し本文を改ざんすることが可能となる。

```
Notification\r\nBcc: attacker@exapmle.com\r\n\r\nManipulated body.
```

メールヘッダ・インジェクション攻撃への対策としては、以下のような方法が考えられる。

- メールヘッダに設定する内容は固定値とし、外部から入力された文字列はすべてメール本文に出力する。
- メールヘッダに設定する内容に改行文字が含まれないことをチェックする。

処理方式

メール送信は時間のかかる処理であるため、Web アプリケーションのリクエストの中で送信処理を行うと応答時間が長くなってしまう。このため、通常は Web アプリケーションのリクエストの中では送信処理を行わず、非同期でメール送信を行う処理方式とすることが多い。メール送信の処理方式について詳細については言及しないが、以下に一例を示すので参考にされたい。

データベースまたはメッセージキューに保持されたメール情報をもとにメール送信を行う

データベースまたはメッセージキューに保持されたメール情報をもとにメール送信を行うには、以下のような機能をアプリケーションに組み込む。

- 送信するメールの情報（宛先や本文、添付ファイル等）をデータベース（またはメッセージキュー）に登録する。
- データベース（またはメッセージキュー）から未送信のメール情報を定期的に取得し、SMTP によるメール送信を行う。
- 送信結果をデータベース（またはメッセージキュー）に登録する。

なお、以下の点を含めて検討する必要がある。

- 登録されたメール情報やメール送信結果の確認方法
- メール送信エラー時の取り扱い

ちなみに： メールサービスによっては、連続してメールが送信された場合に、スパムメールと判定されることがある。左記への対策としては、同一ドメインに対し連続で送信処理を行わないように、送信順序をランダムにする方法が考えられる。

GreenMail を利用したテスト

メール送信機能をテストするためにフェイクサーバとして GreenMail を利用する方法を紹介する。GreenMail はライブラリとして利用する以外に、war ファイルをデプロイして利用することも可能である。

GreenMail を利用したテストコードの実装例を以下に示す。

pom.xml の設定例

```
<dependencies>

    <!-- (1) -->
    <dependency>
        <groupId>com.icegreen</groupId>
        <artifactId>greenmail</artifactId>
        <version>1.4.1</version>
        <scope>test</scope>
    </dependency>

</dependencies>
```

項番	説明
(1)	GreenMail のライブラリを dependencies に追加する。

JUnit ソースの実装例

```
@Inject
EmailService emailService;

@Rule
public final GreenMailRule greenMail = new GreenMailRule(
        ServerSetupTest.SMTP); // (1)

@Test
public void testSend() {

    String from = "info@example.com";
    String to = "foo@example.com";
    String subject = "Registration confirmation.";
    String text = "Hi "
            + to
            + ", welcome to EXAMPLE.COM!\r\n"
            + "If you were not an intended recipient, Please notify the sender.";
    emailService.send(from, to, subject, text);

    assertTrue(greenMail.waitForIncomingEmail(3000, 1)); // (2)

    Message[] messages = greenMail.getReceivedMessages(); // (3)
```

```
    assertNotNull(messages);
    assertEquals(1, messages.length);
    // omitted
}
```

項番	説明
(1)	`ServerSetupTest.SMTP` を指定した `GreenMailRule` をルールとして設定する。 SMTP のポート番号はデフォルトで `3025` が使用される。
(2)	`waitForIncomingEmail` メソッドを利用してメールの到達を待機する。 別スレッドで非同期にメール送信が行われる際に利用する。 この例では、メール送信が非同期で行われている前提で、1 通のメールが到達するまで最大 3 秒待機する。
(3)	`getReceivedMessages` メソッドを利用してすべての受信メールを取得する。 GreenMail で送信したメールは宛先に係らず、すべて GreenMail で受信される。

8.2 JMS(Java Message Service)

8.2.1 Overview

本節では、JMS API と Spring Framework の JMS 連携用コンポーネントを使用したメッセージの送受信方法について説明する。

JMS とは

JMS は Java で MOM（Message Oriented Middleware）を利用するための標準 API である。

JMS のアーキテクチャは、JMS プロバイダを経由してクライアントからクライアントへメッセージを交換する。

JMS は非同期メッセージングをサポートしているため、クライアント間を疎結合にすることができる。

また、後述する Point-to-Point モデルを採用することでメッセージを Queue に格納できるため、クライアントの性能に応じたメッセージ受信が可能となる。

その反面、クライアントからクライアントへのメッセージングにはタイムラグが発生しうるので、リアルタイムな応答が求められる処理に向かない傾向がある。

JMS の詳細については、Java Message Service (JMS) を参照されたい。

JMS を使用することで、同期または非同期でのメッセージングが可能となる。

注釈: 本ガイドラインでは JMS1.1 を使用することを前提としている。

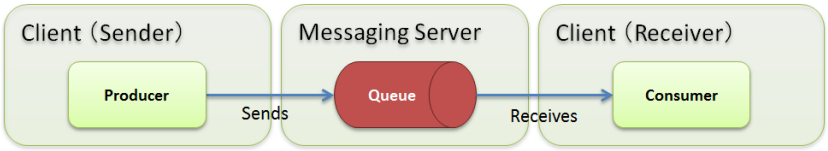
利用する際には、下記に説明する配信モデルとメッセージ送受信方式を要件に合わせて選択する必要がある。

- 配信モデル

  配信モデルは、Point-to-Point（PTP）と Publisher-Subscriber（Pub/Sub）の 2 つのモデルが存在する。

  2 つのモデルの大きな違いは送信者と受信者が 1 対 1 であるか、1 対多であるかであり、用途によって選択する必要がある。

  1. Point-To-Point（PTP）モデル



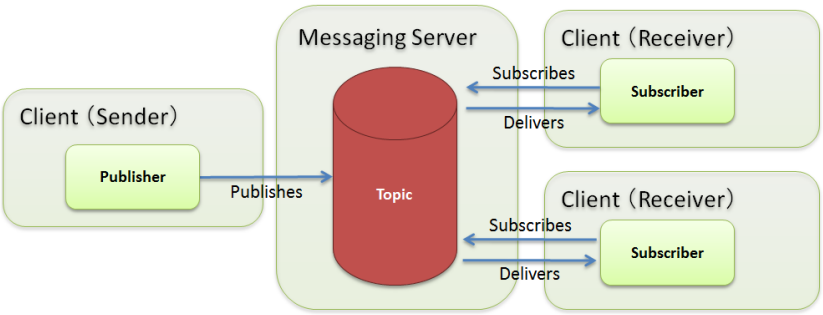
PTP モデルとは、2 つのクライアント間において、一方のクライアント（Producer）からメッセージを送信し、もう一方のクライアント（Consumer）のみがそのメッセージを受信するモデルである。

PTP モデルにおけるメッセージのあて先（Destination）を Queue と呼ぶ。

Producer は Queue にメッセージを送信し、Consumer は Queue からメッセージを取得し、処理を行う。

Consumer からメッセージが取得されるか、メッセージが有効期限に達すると Queue からメッセージが削除される。

2. Publisher-Subscriber（Pub/Sub）モデル



Pub/Sub モデルとは、一方のクライアント（Publisher）からメッセージを発行 (Publishes) し、他方の複数クライアント（Subscriber）にそのメッセージを配信 (Delivers) するモデルである。

Pub/Sub モデルにおけるメッセージのあて先（Destination) を Topic と呼ぶ。

Subscriber は Topic に対し購読依頼 (Subscribes) を行い、Publisher は Topic にメッセージを発行する。

Topic に購読依頼している全ての Subscriber にメッセージが配信される。

本ガイドラインでは、一般的に利用されることが多い PTP モデルの実装方法について説明する。

- メッセージ送信方式

Queue または Topic へのメッセージ送信方式には、同期送信方式と非同期送信方式の 2 通りの処理方式が考えられるが、JMS1.1 では同期送信方式のみがサポートされる。

1. 同期送信方式

明示的にメッセージを送信する機能を呼び出すことで、メッセージに対する処理と送信が開始される。

JMS プロバイダからの応答があるまで待機するため、後続処理がブロックされる。

2. 非同期送信方式

   明示的にメッセージを送信する機能を呼び出すことで、メッセージに対する処理と送信が開始される。

   JMS プロバイダからの応答を待たないため、後続処理を続けて実行する。

   非同期送信方式の詳細については、Java Message Service(Version 2.0) の」7.3. Asynchronous send」を参照されたい。

- メッセージ受信方式

Queue または Topic に受信したメッセージに対する処理を実装する際には、同期受信方式と非同期受信方式の 2 通りの処理方式を選択することができる。

後述するように、同期受信方式の利用ケースは限定的であるため、一般的には非同期受信方式が利用されることが多い。

1. 非同期受信方式

   Queue または Topic がメッセージを受信すると、受信したメッセージに対する処理が開始される。

   1 つのメッセージに対する処理が終了しなくても別のメッセージの処理が開始されるため、並列処理に向いている。

2. 同期受信方式

   明示的にメッセージを受信する機能を呼び出すことで、受信とメッセージに対する処理が開始される。

   メッセージを受信する機能は、Queue または Topic にメッセージが存在しない場合、受信するまで待機する。

   そのため、タイムアウト値を設定することで、メッセージの待ち時間を指定する必要がある。

   メッセージの同期受信を使用する一例として、Web アプリケーションにおいて Queue に溜まったメッセージを、画面操作時など任意のタイミングで取得・処理したい場合や、バッチで定期的にメッセージの処理を行いたい場合に使用することができる。

JMS ではメッセージは以下のパートで構成される。

詳細は Java Message Service(Version 1.1) の」3. JMS Message Model」を参照されたい。

構成	説明
ヘッダ	JMS プロバイダやアプリケーションに対して、メッセージの Destination や識別子などの制御情報や JMS の拡張ヘッダ (JMSX)、JMS プロバイダ独自のヘッダ、アプリケーション独自のヘッダを格納する。
プロパティ	ヘッダに追加する制御情報を格納する。
ペイロード	メッセージ本体を格納する。 データ種別によって、`javax.jms.BytesMessage`、`javax.jms.MapMessage`、`javax.jms.ObjectMessage`、`javax.jms.StreamMessage`、`javax.jms.TextMessage` の 5 つのメッセージタイプを提供している。 JavaBean を送信したい場合は、`ObjectMessage` を利用する。 その場合は、JavaBean をクライアント間で共有する必要がある。

警告: デシリアライズ時の注意点

Queue に `ObjectMessage` が入るとメッセージを取り出す際にデシリアライズが行われる。

デシリアライズ処理は、不正なデータや予期しないデータを使用して業務ロジックの乱用、サービスの拒否、任意のコードの実行が行われる危険があるため、信頼できない送信元から受信しうるものをデシリアライズしてはならない。そのため、Queue も (信頼できない送信元を含み得る) 不特定多数からのメッセージを受け付ける構成であってはならない。

詳細については Deserialization of untrusted data を参照されたい。

JMS の利用

JMS を用いた処理を実装する場合、Java EE で定義された JMS API（以下、JMS API）を使用することで、処理を実現できる。

ただし、本ガイドラインでは、JMS API をそのまま使用する場合に比べてメリット（記述が容易など）が多い、Spring Framework の JMS 連携用コンポーネントを利用する前提としている。

そのため、JMS API の詳細については説明しない。

詳細については Java API を参照されたい。

注釈: JMS は Java API の標準化はしているが、メッセージの物理的なプロトコルの標準化はしていない。

注釈: Java EE サーバでは JMS 実装が標準で組み込まれているためデフォルトで利用可能 (Java EE サーバに組み込まれている JMS プロバイダを使う場合に限られる) だが、Apache Tomcat などのように JMS 実装が組み込まれていない Java EE サーバでは、別途 JMS 実装が必要になる。

Spring Framework のコンポーネントを使用した JMS の利用

Spring Framework では、メッセージ送受信を行うためのライブラリとして以下を提供している。

- `spring-jms`

  JMS を利用したメッセージングを行うためのコンポーネントを提供する。

  このライブラリに含まれるコンポーネントを利用することで、低レベルの JMS API 呼び出しが不要となり、実装を簡素化できる。

  `spring-messaging` を利用することが可能である。

- `spring-messaging`

  メッセージングベースのアプリケーションを作成する際に必要となる基盤機能を抽象化するためのコンポーネントを提供する。

  メッセージとそれを処理するメソッドを対応付けるためのアノテーションのセットが含まれている。

  このライブラリに含まれるコンポーネントを利用することで、メッセージングの実装スタイルを合わせることができる。

`spring-jms` のみでも実装可能であるが、`spring-messaging` を利用することで実装方式を合わせることが可能である。

本ガイドラインでは、`spring-messaging` も利用することを推奨している。

ここでは、具体的な実装方法の説明を行う前に、Spring Framework が提供する JMS 連携用のコンポーネントがどのようにメッセージを送受信しているかを説明する。

まずは、説明に登場するコンポーネントを紹介する。

Spring Framework は、以下にあげるインタフェースやクラスなどを利用して JMS API 経由でメッセージ送受信を行う。

- `javax.jms.ConnectionFactory`

  JMS プロバイダへのコネクション作成用インタフェース。
  アプリケーションから JMS プロバイダへの接続を作成する機能を提供する。

- `javax.jms.Destination`

  あて先 (Queue や Topic) であることを示すインタフェース。

- `javax.jms.MessageProducer`

  メッセージの送信用インタフェース。

- `javax.jms.MessageConsumer`

  メッセージの受信用インタフェース。

- `javax.jms.Message`

  ヘッダとボディを保持するメッセージであることを示すインタフェース。
  送受信はこのインタフェースの実装クラスがやり取りされる。

- `org.springframework.messaging.Message`

  さまざまなメッセージングで扱うメッセージを抽象化したインタフェース。
  JMS でも利用可能である。
  前述したとおり、メッセージングの実装方式を合わせるため、基本的には spring-messaging で提供されている `org.springframework.messaging.Message` を使用する。
  ただし、`org.springframework.jms.core.JmsTemplate` を使用したほうがよい場合が存在するので、その場合には `javax.jms.Message` を使用する。

- `org.springframework.jms.core.JmsMessagingTemplate` および `org.springframework.jms.core.`

  JMS API を利用するためのリソースの生成や解放などをテンプレート化したクラス。
  メッセージの送信及びメッセージの同期受信機能を行う際に使用することで実装を簡素化できる。
  基本的には、`org.springframework.messaging.Message` を扱うことができる `JmsMessagingTemplate` を使用する。
  `JmsMessagingTemplate` は `JmsTemplate` をラップしているため、`JmsTemplate` のプロパティを利用することで設定を行うことができる。
  ただし、一部 `JmsTemplate` をそのまま使用したほうがよい場合が存在する。具体的な使用例については後ほど説明する。

- `org.springframework.jms.listener.DefaultMessageListenerContainer`

`DefaultMessageListenerContainer` は Queue からメッセージを受け取り、受け取ったメッセージを処理する `MessageListener` を起動させる。

- `@org.springframework.jms.annotation.JmsListener`

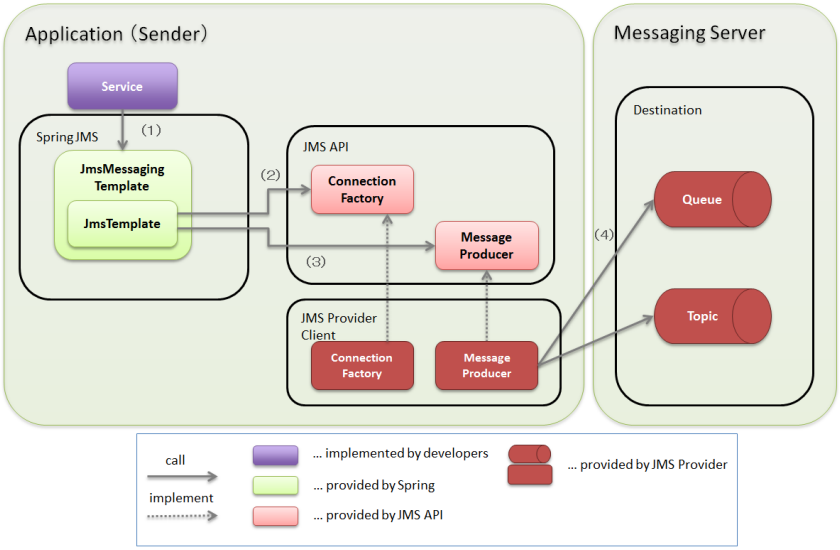
  JMS の `MessageListener` として扱うメソッドであることを示すマーカアノテーション。
  メッセージを受け取った際に処理を行うメソッドに対して`@JmsListener` アノテーションを付与する。

- `org.springframework.jms.connection.JmsTransactionManager`

  JMS(`javax.jms.Connection`/ `javax.jms.Session`) の API を呼び出して、トランザクションを管理するための実装クラス。

メッセージを同期送信する場合

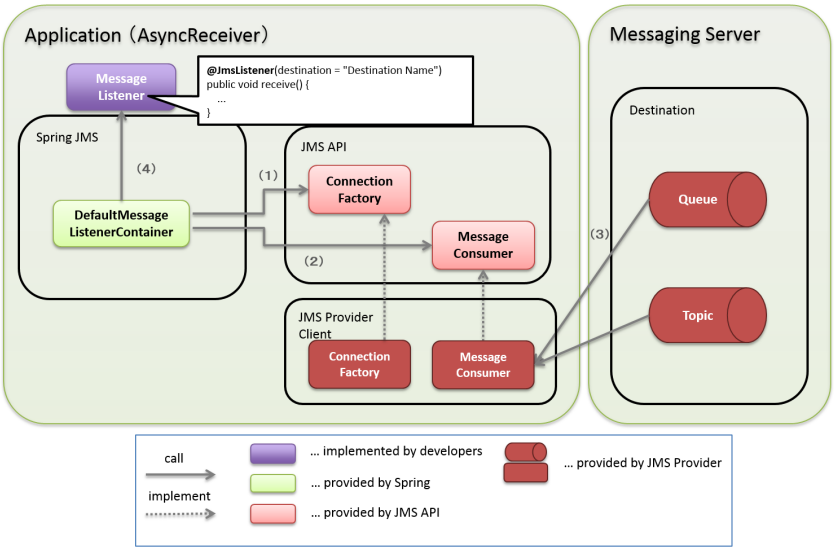
メッセージを同期送信する処理の流れについて図を用いて説明する。



項番	説明
(1)	Service 内で、JmsMessagingTemplate に対して「送信対象の Destination 名」と「送信するメッセージのペイロード」を渡して処理を実行する。 JmsMessagingTemplate は JmsTemplate に処理を委譲する。
(2)	JmsTemplate は JNDI 経由で取得された ConnectionFactory から javax.jms.Connection を取得する。
(3)	JmsTemplate は MessageProducer に Destination とメッセージを渡す。 MessageProducer は javax.jms.Session から生成される。(Session は (2) で取得した Connection から生成される。) また、Destination は (1) で渡された「送信対象の Destination 名」をもとに JNDI 経由で取得される。
(4)	MessageProducer は送信対象の Destination へメッセージを送信する。

メッセージを非同期受信する場合

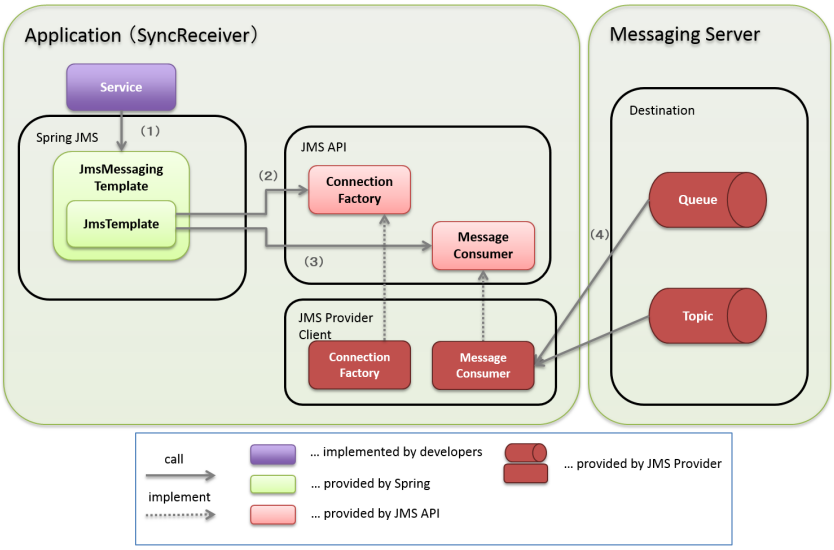
メッセージを非同期受信する処理の流れについて図を用いて説明する。



項番	説明
(1)	JNDI 経由で取得された `ConnectionFactory` から `Connection` を取得する。
(2)	`DefaultMessageListenerContainer` は `MessageConsumer` に `Destination` を渡す。 `MessageConsumer` は `Session` から生成される。(`Session` は (1) で取得した `Connection` から生成される。) また、`Destination` は`@JmsListener` アノテーションで指定された「受信対象の Destination 名」をもとに JNDI 経由で取得される。
(3)	`MessageConsumer` は `Destination` からメッセージを受信する。
(4)	受信したメッセージを引数として、`MessageListener` 内の`@JmsListener` アノテーションが設定されたメソッド (リスナーメソッド) が呼び出される。リスナーメソッドは `DefaultMessageListenerContainer` で管理される。

メッセージを同期受信する場合

メッセージを同期受信する処理の流れについて図を用いて説明する。



項番	説明
(1)	Service 内で、JmsMessagingTemplate に対して、「受信対象の Destination 名」を渡す。 JmsMessagingTemplate は JmsTemplate に処理を委譲する。
(2)	JmsTemplate は JNDI 経由で取得された ConnectionFactory から Connection を取得する。
(3)	JmsTemplate は MessageConsumer に Destination とメッセージを渡す。 MessageConsumer は Session から生成される。(Session は (2) で取得した Connection から生成される。) また、Destination は (1) で渡された「受信対象の Destination 名」をもとに JNDI 経由で取得される。
(4)	MessageConsumer は Destination からメッセージを受信する。 メッセージは JmsTemplate や JmsMessagingTemplate を経由して Service に返却される。

プロジェクト構成について

JMS を利用する場合のプロジェクトの推奨構成について説明する。

シリアライズした JavaBean を ObjectMessage 経由で送受信する場合、この JavaBean を送信側と受信側で共有する必要がある。

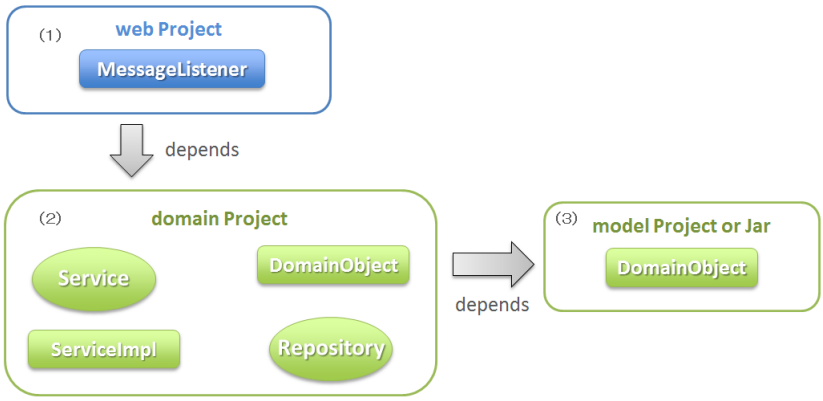
この場合、既存のブランクプロジェクトとは別に model プロジェクトを追加することを推奨する。

- **model の共有**
  - 送信または受信側のクライアントが model を提供していない場合

    model プロジェクトを追加して、通信先のクライアントに Jar ファイルを配布する。
  - 送信または受信側のクライアントが model を提供している場合

    提供された model をライブラリに追加する。

model プロジェクト、または、配布されたアーカイブファイルと既存のプロジェクトとの関係は以下のようになる。



項番	プロジェクト名	説明
(1)	web プロジェクト	非同期受信を行うためのリスナークラスを配置する。
(2)	domain プロジェクト	非同期受信を行うためのリスナークラスから実行される Service を配置する。 その他、Repository などは従来と同じである。
(3)	model プロジェクトもしくは Jar ファイル	ドメイン層に属するクラスのうち、クライアント間で共有するクラスを使用する。

model プロジェクトを追加するためには、以下を実施する。

- model プロジェクトの作成
- domain プロジェクトから model プロジェクトへの依存関係の追加

詳細な追加方法については、同じように JavaBean の共有を行っている *SOAP Web Service（サーバ/クライアント）の SOAP サーバ用にプロジェクトの設定を変更する* を参照されたい。

8.2.2 How to use

メッセージの送受信に共通する設定

本節では、メッセージの送受信に必要となる共通的な設定について説明する。

依存ライブラリの設定

Spring Framework の JMS 連携用コンポーネントを利用するために、domain プロジェクトの pom.xml に Spring Framework の `spring-jms` を追加する。

- `[projectName]-domain/pom.xml`

```
<dependencies>

    <!-- (1) -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jms</artifactId>
    </dependency>

</dependencies>
```

項番	説明
(1)	`spring-jms` を dependencies に追加する。 `spring-jms` は `spring-messaging` に依存するため、`spring-messaging` も推移的に依存ライブラリとして追加される。

`spring-jms` の他に、pom.xml に JMS プロバイダのライブラリを追加する。

pom.xml へのライブラリの追加例については、*JMS プロバイダに依存する設定* を参照されたい。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

`ConnectionFactory` の設定

`ConnectionFactory` の定義の方法には、アプリケーションサーバで定義する方法と、Bean 定義ファイル

で定義する方法がある。

特別な理由がない場合、Bean 定義ファイルを JMS プロバイダ非依存とするため、アプリケーションサーバで定義する方法を選択する。

本節では、アプリケーションサーバで定義する方法についてのみ説明する。

アプリケーションサーバで定義した `ConnectionFactory` を使用するためには、Bean 定義ファイルに JNDI 経由で取得した JavaBean を利用するための設定を行う必要がある。

- `[projectName]-env/src/main/resources/META-INF/spring/[projectName]-env.xml`

```
<!-- (1) -->
<jee:jndi-lookup id="connectionFactory" jndi-name="jms/ConnectionFactory"/>
```

項番	説明
(1)	`jndi-name` 属性に、アプリケーションサーバ提供の `ConnectionFactory` の JNDI 名を指定する。 `resource-ref` 属性がデフォルトで `true` のため、JNDI 名にプレフィックス (java:comp/env/) がない場合は、自動的に付与される。

注釈: Bean 定義した ConnectionFactory を使用する場合

JNDI を利用しない場合、`ConnectionFactory` の実装クラスを Bean 定義することでも `ConnectionFactory` を利用することが可能である。この場合、`ConnectionFactory` の実装クラスは JMS プロバイダ依存となる。詳細については、*JMS プロバイダに依存する設定* の」JNDI を使用しない場合の設定」を参照されたい。

`DestinationResolver` の設定

Destination の名前解決には、JNDI による解決と JMS プロバイダでの解決の二通りの方法がある。

デフォルトでは JMS プロバイダでの解決が行われるが、ポータビリティや管理の観点から、特別な理由がない場合は JNDI による解決を推奨する。

`org.springframework.jms.support.destination.JndiDestinationResolver` を使用することで、JNDI 名により Destination の名前解決を行うことができる。

以下に `JndiDestinationResolver` の定義例を示す。

- `[projectName]-env/src/main/resources/META-INF/spring/`

[projectName]-env.xml

```
<!-- (1) -->
<bean id="destinationResolver"
   class="org.springframework.jms.support.destination.JndiDestinationResolver
↪">
   <property name="resourceRef" value="true" /> <!-- (2) -->
</bean>
```

項番	説明
(1)	JndiDestinationResolver を Bean 定義する。
(2)	JNDI 名にプレフィックス (java:comp/env/) がないときに、自動的に付与させる場合は true を設定する。デフォルトは false である。 **警告:** <jee:jndi-lookup/>の resource-ref 属性とはデフォルト値が異なることに注意されたい。

注釈: DynamicDestinationResolver を使用する場合

JNDI を利用せずに JMS プロバイダで Destination の名前解決する場合、DynamicDestinationResolver を利用する。DynamicDestinationResolver の設定については、*JMS プロバイダに依存する設定* の」JNDI を使用しない場合の設定」を参照されたい。

メッセージを同期送信する方法

PTP モデルにて、クライアント（Producer）から JMS プロバイダへメッセージを同期送信する方法を説明する。

基本的な同期送信

JmsMessagingTemplate を利用して、JMS プロバイダへの同期送信処理を実現する。

Todo クラスのオブジェクトをメッセージ同期送信する場合の実装例を示す。

最初に JmsMessagingTemplate の設定方法を示す。

- [projectName]-env/src/main/resources/META-INF/spring/[projectName]-env.xml

```
<bean id="cachingConnectionFactory"
   class="org.springframework.jms.connection.CachingConnectionFactory"> <!--
↪(1) -->
   <property name="targetConnectionFactory" ref="connectionFactory" /> <!--
↪(2) -->
   <property name="sessionCacheSize" value="1" />  <!-- (3) -->
</bean>
```

項番	説明
(1)	Session、MessageProducer/Consumer のキャッシュを行う org.springframework.jms.connection.CachingConnectionFactory を Bean 定義する。 Bean 定義もしくは JNDI 名でルックアップした JMS プロバイダ固有の ConnectionFactory をそのまま使うのではなく、CachingConnectionFactory にラップして使用することで、キャッシュ機能を使用することができる。
(2)	Bean 定義もしくは JNDI 名でルックアップした JMS プロバイダ固有の ConnectionFactory を指定する。
(3)	Session のキャッシュ数を設定する。（デフォルト値は 1） この例では 1 を指定しているが、性能要件に応じて適宜キャッシュ数を変更すること。 このキャッシュ数を超えてセッションが必要になるとキャッシュを使用せず、新しいセッションの作成と破棄を繰り返すことになる。 すると処理効率が下がり、性能劣化の原因になるので注意すること。

- [projectName]-domain/src/main/resources/META-INF/spring/[projectName]-infra.xml

```
<!-- (1) -->
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
   <property name="connectionFactory" ref="cachingConnectionFactory" />
   <property name="destinationResolver" ref="destinationResolver" />
</bean>

<!-- (2) -->
<bean id="jmsMessagingTemplate" class="org.springframework.jms.core.
↪JmsMessagingTemplate">
```

```
    <property name="jmsTemplate" ref="jmsTemplate"/>
</bean>
```

項番	説明
(1)	JmsTemplate を Bean 定義する。 JmsTemplate は低レベルの API ハンドリング（JMS API 呼び出し）を代行する。 設定可能な属性に関しては、下記の JmsTemplate の属性一覧を参照されたい。
(2)	JmsMessagingTemplate を Bean 定義する。同期送信処理を代行する JmsTemplate をインジェクションする。

同期送信に関連する JmsTemplate の属性は以下が存在する。
必要に応じて設定を行う必要がある。

項番	設定項目	内容	必須	デフォルト値
1.	connectionFactory	使用する ConnectionFactory を設定する。	○	なし（必須であるため）
2.	pubSubDomain	メッセージングモデルについて設定する。 PTP（Queue）モデルは false、Pub/Sub（Topic）は true に設定する。	-	false
3.	sessionTransacted	セッションでのトランザクション管理をするかどうか設定する。 本ガイドラインでは、後述するトランザクション管理を使用するため、デフォルトのままの false を推奨する。	-	false

Continued on next page

表 8.5 – continued from previous page

項番	設定項目	内容	必須	デフォルト値
4.	messageConverter	メッセージコンバータを設定する。 本ガイドラインで紹介している範囲では、デフォルトのままで問題ない。	-	SimpleMessageCo が使用される。
5.	destinationResolver	DestinationResolver を設定する。 本ガイドラインでは、JNDI で名前解決を行う、JndiDestinationResolver を設定することを推奨する。	-	DynamicDestinat が使用される。 (DynamicDestina を利用すると JMS プロバイダで Destination の名前解決が行われる。)
6.	defaultDestination	既定の Destination を設定する。 Destination を明示的に指定しない場合、この Destination が使用される。	-	null(既定の Destination なし)

Continued on next page

表 8.5 – continued from previous page

項番	設定項目	内容	必須	デフォルト値
7.	deliveryMode	配信モードを 1(NON_PERSISTENT)、2(PERSISTENT) から設定する。 2(PERSISTENT) は、メッセージの永続化を行う。 1(NON_PERSISTENT) は、メッセージの永続化を行わない。 そのため、性能は上がるが、JMS プロバイダの再起動などが起こるとメッセージが消失する可能性がある。 本ガイドラインでは、メッセージの消失を避けるため、 2(PERSISTENT) を使用することを推奨する。 この設定を使用する場合、後述する `explicitQosEnabled` に `true` を設定する必要があるので注意すること。	-	2(PERSISTENT)
8.	priority	メッセージの優先度を設定する。優先度は 0 から 9 まで設定できる。 数値が大きいほど優先度が高くなる。 同期送信時にメッセージが Queue に格納される時点で優先度が評価され、優先度が高いメッセージは低いメッセージより先に取り出されるように格納される。 優先度が同じメッセージは FIFO（First-In First-Out）で扱われる。 この設定を使用する場合、後述する `explicitQosEnabled` に `true` を設定する必要があるので注意すること。	-	4
9.	timeToLive	メッセージの有効期限をミリ秒で設定する。 メッセージが有効期限に達すると、JMS プロバイダは Queue からメッセージを削除する。 この設定を使用する場合、後述する `explicitQosEnabled` に `true` を設定する必要があるので注意すること。	-	0（無制限）

Continued on next page

表 8.5 – continued from previous page

項番	設定項目	内容	必須	デフォルト値
10.	explicitQosEnabled	deliveryMode、priority、timeToLive を有効にする場合は true を設定する。	-	false

(*1)org.springframework.jms.support.converter.SimpleMessageConverter

(*2)org.springframework.jms.support.destination.DynamicDestinationResolver

次に送信対象の JavaBean を作成する。

- [projectName]-domain/src/main/java/com/example/domain/model/Todo.java

```
package com.example.domain.model;

import java.io.Serializable;

public class Todo implements Serializable { // (1)

    private static final long serialVersionUID = -1L;

    // omitted

    private String description;

    // omitted

    private boolean finished;

    // omitted

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public boolean isFinished() {
        return finished;
    }

    public void setFinished(boolean finished) {
        this.finished = finished;
    }

}
```

項番	説明
(1)	基本的には通常の JavaBean で問題ないが、シリアライズして送信するため、 `java.io.Serializable` インタフェース を実装する必要がある。

最後に実際に同期送信を行う処理を記述する。

以下では、指定したテキストをもつ `Todo` オブジェクトを Queue に同期送信する実装例を示す。

- `[projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java`

```
package com.example.domain.service.todo;

import javax.inject.Inject;
import org.springframework.jms.core.JmsMessagingTemplate;
import org.springframework.stereotype.Service;
import com.example.domain.model.Todo;

@Service
public class TodoServiceImpl implements TodoService {

    @Inject
    JmsMessagingTemplate jmsMessagingTemplate;    // (1)

    @Override
    public void sendMessage(String message) {

        Todo todo = new Todo();
        // omitted

        jmsMessagingTemplate.convertAndSend("jms/queue/TodoMessageQueue",␣
↪todo);  // (2)

    }
}
```

項番	説明
(1)	`JmsMessagingTemplate` をインジェクションする。
(2)	`JmsMessagingTemplate` の `convertAndSend` メソッドを使用して、引数の JavaBean を `org.springframework.messaging.Message` インタフェースの実装クラスに変換し、指定した Destination に対しメッセージを同期送信する。 デフォルトで変換には、`org.springframework.jms.support.converter.SimpleMessageConverter` が使用される。 `SimpleMessageConverter` を使用すると、`javax.jms.Message`、`java.lang.String`、byte 配列、`java.util.Map`、`java.io.Serializable` インタフェースを実装したクラスを送信可能である。

注釈: 業務ロジック内で JMS の例外ハンドリング

JMS (Java Message Service) の Introduction で触れられているように、Spring Framework では検査例外を非検査例外に変換している。そのため、業務ロジック内で JMS の例外をハンドリングする場合は、非検査例外を扱う必要がある。

Template クラス	例外の変換を行うメソッド	変換後の例外
`JmsMessagingTemplate`	`JmsMessagingTemplate` の `convertJmsException` メソッド	`MessagingException`(*1) 及びそのサブ例外
`JmsTemplate`	`JmsAccessor` の `convertJmsAccessException` メソッド	`JmsException`(*2) 及びそのサブ例外

(*1) `org.springframework.messaging.MessagingException`

(*2) `org.springframework.jms.JmsException`

メッセージヘッダを編集して同期送信する場合

JmsMessagingTemplate の convertAndSend メソッドの引数に Key-Value 形式のヘッダ属性と値を指定することで、ヘッダ属性を編集して同期送信することが可能である。ヘッダの詳細については、javax.jms.Message を参照されたい。送信、応答メッセージなどを紐づける役割の JMSCorrelationID を同期送信時に指定する場合の実装例を示す。

- [projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java

```
package com.example.domain.service.todo;

import java.util.Map;
import javax.inject.Inject;
import org.springframework.jms.core.JmsMessagingTemplate;
import org.springframework.stereotype.Service;
import org.springframework.jms.support.JmsHeaders;
import com.example.domain.model.Todo;

@Service
public class TodoServiceImpl implements TodoService {

@Inject
JmsMessagingTemplate jmsMessagingTemplate;

  public void sendMessageWithCorrelationId(String correlationId) {

    Todo todo = new Todo();
    // omitted

    Map<String, Object> headers = new HashMap<>();
    headers.put(JmsHeaders.CORRELATION_ID, correlationId);// (1)

    jmsMessagingTemplate.convertAndSend("jms/queue/TodoMessageQueue",
            todo, headers); // (2)

  }
}
```

項番	説明
(1)	Map の実装クラスに対し、ヘッダ属性名とその値を設定してヘッダ情報を作成する。
(2)	JmsMessagingTemplate の convertAndSend メソッドを使用することで、(2) で作成したヘッダ情報を付与したメッセージを同期送信する。

> **警告: 編集可能なヘッダ属性について**
>
> Spring Framework の SimpleMessageConverter によるメッセージ変換時には、ヘッダ属性の一部 (JMSDestination、JMSDeliveryMode、JMSExpiration、JMSMessageID、JMSPriority、JMSRedelivered と JMSTimestamp) を read-only として扱っている。そのため、上記の実装例のように read-only のヘッダ属性を設定しても、送信したメッセージのヘッダには格納されない。（メッセージのプロパティとして保持される。）read-only のヘッダ属性うち、JMSDeliveryMode や JMSPriority については、JmsTemplate 単位での設定が可能である。詳細については、基本的な同期送信 の JmsTemplate の属性一覧を参照されたい。

トランザクション管理

データの一貫性を保証する必要がある場合は、トランザクション管理機能を使用する。
本ガイドラインで推奨する「宣言型トランザクション管理」を利用した実装例を以下に示す。
「宣言型トランザクション管理」の詳細は、トランザクション管理について を参照されたい。

トランザクション管理を実現するためには、
org.springframework.jms.connection.JmsTransactionManager を利用する。
最初に設定例を示す。

- [projectName]-domain/src/main/resources/META-INF/spring/[projectName]-infra.xml

```
<!-- (1) -->
<bean id="sendJmsTransactionManager"
   class="org.springframework.jms.connection.JmsTransactionManager">
   <!-- (2) -->
   <property name="connectionFactory" ref="cachingConnectionFactory" />
</bean>
```

項番	説明
(1)	JmsTransactionManager を Bean 定義する。 注釈: **TransactionManager の bean 名について** @Transactional アノテーションを付与した場合、デフォルトでは Bean 名 transactionManager で登録されている Bean が使用される。(詳細は、トランザクション管理を使うための設定について を参照されたい) Blank プロジェクトには、transactionManager という Bean 名で DataSourceTransactionManager が定義されているため、上記の設定では別名で Bean を定義している。 そのため、アプリケーション内で、TransactionManager を 1 つしか使用しない場合は、bean 名を transactionManager にすることで@Transactional アノテーションでの transactionManager 属性の指定を省略することができる。
(2)	トランザクションを管理する CachingConnectionFactory を指定する。

トランザクション管理を行い、Todo オブジェクトを Queue に同期送信する実装例を以下に示す。

- [projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java

```
package com.example.domain.service.todo;

import javax.inject.Inject;
import org.springframework.jms.core.JmsMessagingTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.example.domain.model.Todo;

@Service
@Transactional("sendJmsTransactionManager")  // (1)
public class TodoServiceImpl implements TodoService {
    @Inject
    JmsMessagingTemplate jmsMessagingTemplate;

    @Override
    public void sendMessage(String message) {

        Todo todo = new Todo();
        // omitted
```

```
        jmsMessagingTemplate.convertAndSend("jms/queue/TodoMessageQueue", todo);
↪   // (2)
    }

}
```

項番	説明
(1)	`@Transactional` アノテーションを利用してトランザクション境界を宣言する。 これにより、クラス内の各メソッドの開始時にトランザクションが開始され、メソッドの終了時にトランザクションがコミットされる。
(2)	Queue にメッセージを同期送信する。 ただし、実際にメッセージが Queue に送信されるのはトランザクションがコミットされるタイミングとなるので注意すること。

DB のトランザクション管理を行う必要があるアプリケーションでは、業務の要件をもとに JMS と DB のトランザクションの関連を精査した上でトランザクションの管理方針を決定すること。

JMS と DB のトランザクションの連携には JTA によるグローバルトランザクションを使用する方法があるが、プロトコルの特性上、性能面のオーバーヘッドがかかるため、」Best Effort 1 Phase Commit」の使用を推奨する。詳細は以下を参照されたい。

Distributed transactions in Spring, with and without XA

Spring Distributed transactions using Best Effort 1 Phase Commit

警告: メッセージ受信後に JMS プロバイダとの接続が切れるなどで JMS プロバイダにトランザクションの処理結果が返らない場合

メッセージ受信後に JMS プロバイダとの接続が切れるなどで、JMS プロバイダにトランザクションの処理結果が返らない場合、トランザクションの扱いは JMS プロバイダに依存する。そのため、受信したメッセージの消失などを考慮した設計を行うこと。特に、メッセージの消

> 失が絶対に許されないような場合には、メッセージの消失を補う仕組みを用意するか、グローバルトランザクションなどの利用を検討する必要がある。

「Best Effort 1 Phase Commit」は `org.springframework.data.transaction.ChainedTransactionManager` を利用することで実現する。

以下に、JMS のトランザクション管理にトランザクション管理の `sendJmsTransactionManager` を使用し、DB のトランザクション管理に Blank プロジェクトのデフォルトの設定で定義されている `transactionManager` を使用する設定例を示す。

- `[projectName]-domain/src/main/resources/META-INF/spring/[projectName]-infra.xml`

```
<!-- (1) -->
<bean id="sendChainedTransactionManager" class="org.springframework.
↪data.transaction.ChainedTransactionManager">
    <constructor-arg>
        <list>
            <!-- (2) -->
            <ref bean="sendJmsTransactionManager" />
            <ref bean="transactionManager" />
        </list>
    </constructor-arg>
</bean>
```

項番	説明
(1)	`ChainedTransactionManager` を Bean 定義する。
(2)	JMS と DB のトランザクションマネージャを指定する。 登録した順にトランザクションが開始され、登録した逆順にトランザクションがコミットされる。

上記の設定を利用した実装例を以下に示す。

- `[projectName]-domain/src/main/java/com/example/domain/service/todo/ChainedTransactionalTodoServiceImpl.java`

```
package com.example.domain.service.todo;

import javax.inject.Inject;
import org.springframework.jms.core.JmsMessagingTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.example.domain.model.Todo;

@Service
@Transactional("sendChainedTransactionManager")  // (1)
public class ChainedTransactionalTodoServiceImpl implements␣
↪ChainedTransactionalTodoService {
    @Inject
    JmsMessagingTemplate jmsMessagingTemplate;

    @Inject
    TodoSharedService todoSharedService;

    @Override
    public void sendMessage(String message) {
        Todo todo = new Todo();
        // omitted

        jmsMessagingTemplate.convertAndSend("jms/queue/TodoMessageQueue",␣
↪todo); // (2)

        // omitted
        todoSharedService.insert(todo); // (3)
    }

}
```

項番	説明
(1)	`@Transactional` アノテーションに `sendChainedTransactionManager` を指定することで、JMS と DB のトランザクション管理を行う。 `@Transactional` アノテーションの詳細については、ドメイン層の実装のトランザクション管理についてを参照されたい。
(2)	メッセージの同期送信を行う。
(3)	DB アクセスを伴う処理を実行する。この例では、DB の更新を伴う SharedService を実行している。

注釈: 業務上、JMS と DB など複数のトランザクションをまとめて管理する必要がある場合、グローバルトランザクションを検討する。グローバルトランザクションについては、トランザクション管理を使うための設定についての」複数 DB（複数リソース）に対するトランザクション管理（グローバルトランザクションの管理）が必要な場合」を参照されたい。

メッセージを非同期受信する方法

*JMS とは*の」メッセージ受信方式」で述べたように、一般的に受信処理を行う場合には非同期受信を利用する。

非同期受信機能を司る `DefaultMessageListenerContainer` に対し、`@JmsListener` アノテーションが付与されたリスナーメソッドを登録することで非同期受信処理を実現する。

非同期受信時の処理を行うリスナーメソッドの役割として、以下が存在する。

1. メッセージを受け取るためのメソッドを提供する。

   `@JmsListener` アノテーションが付与されたメソッドを実装することで、メッセージを受け取ることができる。

2. 業務処理の呼び出しを行う。

   リスナーメソッドでは業務処理の実装は行わず、Service のメソッドに処理を委譲する。

3. 業務ロジックで発生した例外のハンドリングを行う。

   ビジネス例外や正常稼働時に発生するライブラリ例外のハンドリングを行う。

4. 処理結果をメッセージ送信する。

   応答メッセージなどの送信が必要なメソッドでは、`org.springframework.jms.listener.adapter.JmsResponse` を利用することで、指定した Destination に対してリスナーメソッドや業務ロジックの処理結果をメッセージ送信することができる。

基本的な非同期受信

`@JmsListener` アノテーションを利用した非同期受信の方法について説明をする。

非同期受信の実装には下記の設定が必要となる。

- JMS Namespace を定義する。
- `@JmsListener` アノテーションを有効化する。
- DI コンテナで管理しているコンポーネントのメソッドに`@JmsListener` アノテーションを指定する。

それぞれの詳細な実装方法について、以下に記述する。

- [projectName]-web/src/main/resources/META-INF/spring/applicationContext.xml

```
<!-- (1) -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jms="http://www.springframework.org/schema/jms"
    xsi:schemaLocation="http://www.springframework.org/schema/beans https://
↪www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jms https://www.springframework.
↪org/schema/jms/spring-jms.xsd">

    <!-- (2) -->
    <jms:annotation-driven />


    <!-- (3) -->
    <jms:listener-container
        factory-id="jmsListenerContainerFactory"
        destination-resolver="destinationResolver"
        concurrency="1"/>
```

項番	属性名	内容
(1)	xmlns:jms	JMS Namespace を定義する。 値として http://www.springframework.org/schema/jms を指定する。 JMS Namespace の詳細については、JMS Namespace Support を参照されたい。
	xsi:schemaLocation	スキーマの URL を指定する。 値に http://www.springframework.org/schema/jms と https://www.springframework.org/schema/jms/spring-jms.xsd を追加する。
(2)	-	<jms:annotation-driven />を利用して、@JmsListener アノテーションや@SendTo アノテーション等の JMS 関連のアノテーション機能を有効化する。

Continued on next page

表 8.6 – continued from previous page

項番	属性名	内容
(3)	-	<jms:listener-container/>を利用して DefaultMessageListenerContainer を生成するファクトリへパラメータを与えることで、DefaultMessageListenerContainer の設定を行う。 <jms:listener-container/>の属性には、利用したい ConnectionFactory の Bean を指定できる connection-factory 属性が存在する。connection-factory 属性のデフォルト値は connectionFactory である。 この例では、*ConnectionFactory の設定*で示した ConnectionFactory の Bean(Bean 名は connectionFactory) を利用するため、connection-factory 属性を省略している。 <jms:listener-container/>には、ここで紹介した以外の属性も存在する。 詳細については、Attributes of the JMS <listener-container> element を参照されたい。 **警告:** DefaultMessageListenerContainer 内部には独自のキャッシュ機能が備わっている。一方で、AP サーバ製品や MOM 製品によって関連リソースをキャッシュする場合もある。両者の管理に不整合が生じないように cache 属性でキャッシュレベルを指定すること。詳細については、DefaultMessageListenerContainer の Javadoc を参照されたい。本ガイドラインでは、<jms:listener-container/>の connection-factory 属性には、*ConnectionFactory の設定*で定義した ConnectionFactory を指定する。

Continued on next page

表 8.6 – continued from previous page

項番	属性名	内容
	concurrency	DefaultMessageListenerContainer が管理するリスナーメソッドごとの並列数に対する上限を指定する。 concurrency 属性のデフォルトは 1 である。 並列数の下限と上限を指定することも可能である。例えば、下限を 5、上限を 10 とする場合は」5-10」と指定する。 リスナーメソッドの並列数が設定した上限値に達した場合は、並列に処理されず待ち状態となる。 必要に応じて値を設定すること。 --- **注釈:** リスナーメソッド単位で並列数を指定したい場合は、@JmsListener アノテーションの concurrency 属性を利用することができる。 ---
	destination-resolver	非同期受信時の Destination 名解決で使用する DestinationResolver の Bean 名を設定する。 DestinationResolver の Bean 定義については、*DestinationResolver の設定*を参照されたい。 destination-resolver 属性を指定していない場合は DefaultMessageListenerContainer 内で生成された DynamicDestinationResolver が利用される。
	factory-id	Bean 定義を行う DefaultJmsListenerContainerFactory の名前を設定する。 @JmsListener アノテーションがデフォルトで Bean 名 jmsListenerContainerFactory を参照するため、<jms:listener-container/>が一つの場合は Bean 名を jmsListenerContainerFactory とすることを推奨する。

Continued on next page

表 8.6 – continued from previous page

項番	属性名	内容
	cache	Connection、Session や MessageConsumer などのキャッシュ対象を決定するために、キャッシュレベルを指定する。 connection, session, consumer, none(キャッシュしない), auto(自動的に選択) のいずれかより選択する。 ここではデフォルトの auto を指定するため、cache 属性を省略している。 **注釈:** auto を指定した場合、transaction-manager 属性の指定有無によって、挙動が変わる。指定した場合は none 指定時と同様となり、指定しない場合は consumer 指定時と同様となる。これは、transaction-manager 属性が、JTA トランザクションを使用する場合にのみ指定することに起因している。アプリケーションサーバ内で Connection や Session などをプールしない場合は、性能向上のため consumer を指定することを検討すること。

DI コンテナで管理しているコンポーネントのメソッドに`@JmsListener` アノテーションを指定することで、指定した Destination より非同期でメッセージを受信する。実装方法を以下に示す。

- `[projectName]-web/src/main/java/com/example/listener/todo/TodoMessageListener.java`

```
package com.example.listener.todo;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;
import com.example.domain.model.Todo;
@Component
public class TodoMessageListener {

    @JmsListener(destination = "jms/queue/TodoMessageQueue")    // (1)
    public void receive(Todo todo) {
        // omitted
    }

}
```

項番	説明
(1)	非同期受信用のメソッドに対し`@JmsListener` アノテーションを設定する。 `destination` 属性には、受信先の Destination 名を指定する。

`@JmsListener` アノテーションの主な属性の一覧を以下に示す。詳細やその他の属性については、@JmsListener アノテーションの Javadoc を参照されたい。

項番	項目	内容
1.	destination	受信する Destination を指定する。
2.	containerFactory	リスナーメソッドの管理を行う DefaultJmsListenerContainerFactory の Bean 名を指定する。 デフォルトは jmsListenerContainerFactory である。
3.	selector	受信するメッセージを限定するための条件であるメッセージセレクタを指定する。 明示的に値を指定しない場合、デフォルトは」「(空文字) であり、すべてのメッセージが受信対象となる。 利用方法については、非同期受信するメッセージを限定する場合を参照されたい。
3.	concurrency	リスナーメソッドの並列数の上限を指定する。 concurrency 属性のデフォルトは 1 である。 並列数の下限と上限を指定することも可能である。例えば、下限を 5、上限を 10 とする場合は」5-10」と指定する。 リスナーメソッドの並列数が設定した上限値に達した場合は、並列に処理されず待ち状態となる。 必要に応じて値を設定すること。

メッセージのヘッダ情報を取得する

非同期受信の処理結果を Producer 側で指定した Destination(ヘッダ属性 JMSReplyTo の値) に送信する場合など、メッセージのヘッダ情報をリスナーメソッド内で利用する場合には、@org.springframework.messaging.handler.annotation.Header アノテーションを利用する。

実装例を以下に示す。

- [projectName]-web/src/main/java/com/example/listener/todo/TodoMessageListener.java

```
@JmsListener(destination = "jms/queue/TodoMessageQueue")
public JmsResponse<Todo> receiveAndResponse(
        Todo todo, @Header("jms_replyTo") Destination␣
↪storeResponseMessageQueue) { // (1)

    // omitted

     return JmsResponse.forDestination(todo, storeResponseMessageQueue);
}
```

項番	説明
(1)	受信メッセージのヘッダ属性 JMSReplyTo の値を取得するために、@Header アノテーションを指定する。 JMS の標準ヘッダ属性を取得する場合に指定するキーの値については、JmsHeaders の定数の定義を参照されたい。

非同期受信後の処理結果をメッセージ送信

@JmsListener アノテーションを定義したメソッドの処理結果を、応答メッセージとして Destination に送信する方法が用意されている。
処理結果の送信先を指定する方法として、以下の 2 つが存在する。

- 処理結果の送信先を静的に指定する場合
- 処理結果の送信先を動的に指定する場合

それぞれについて、以下に説明する。

- **処理結果の送信先を静的に指定する場合**

  @JmsListener アノテーションが定義されているメソッドに対し、Destination を指定した@SendTo アノテーションを定義することで、固定の Destination への処理結果のメッセージ送信を実現する。

  実装例を以下に示す。

- [projectName]-web/src/main/java/com/example/listener/todo/TodoMessageListener.java

```
@JmsListener(destination = "jms/queue/TodoMessageQueue")
@SendTo("jms/queue/ResponseMessageQueue") // (1)
public Todo receiveMessageAndSendTo(Todo todo) {

    // omitted
    return todo; // (2)
}
```

項番	説明
(1)	`@SendTo` アノテーションを定義することで、処理結果の送信先に対するデフォルトの Destination を指定できる。
(2)	`@SendTo` アノテーションに定義した Destination に送信するデータを返却する。 許可されている返却値の型は `org.springframework.messaging.Message`、`javax.jms.Message`、`String`、`byte` 配列、`Map`、`Serializable` インタフェースを実装したクラス である。

- 処理結果の送信先を動的に変更する場合

  動的に送信先の Destination を変更する場合は `JmsResponse` クラスの `forDestination` や `forQueue` メソッドを用いる。

  送信先の Destination や Destination 名を動的に変更することで、任意の Destination に処理結果を送信することができる。実装例を以下に示す。

  - `[projectName]-web/src/main/java/com/example/listener/todo/TodoMessageListener.java`

```
@JmsListener(destination = "jms/queue/TodoMessageQueue")
public JmsResponse<Todo> receiveMessageJmsResponse(Todo todo) {

    // omitted

    String resQueue = null;

    if (todo.isFinished()) {
        resQueue = "jms/queue/FinihedTodoMessageQueue";
    } else {
        resQueue = "jms/queue/ActiveTodoMessageQueue";
    }

    return JmsResponse.forQueue(todo, resQueue); // (1)
}
```

項番	説明
(1)	処理内容に応じて送信先の Queue を変更する場合は `JmsResponse` クラスの `forDestination` や `forQueue` メソッドを使用する。 この例では、`forQueue` メソッドを利用して、Destination 名から送信を行っている。 **注釈:** `JmsResponse` クラスの `forQueue` メソッドを利用する場合は、文字列である Destination 名を利用する。Destination 名の解決には、`DefaultMessageListenerContainer` に指定した `DestinationResolver` が利用される。

注釈: 処理結果の送信先を Producer 側で指定する場合

以下のように実装することで、Producer 側で指定した任意の Destination に処理結果のメッセージを送信することができる。

実装箇所	実装内容
Producer 側	JMS 標準に則りメッセージのヘッダ属性 `JMSReplyTo` に Destination を指定する。 ヘッダ属性の編集については、メッセージヘッダを編集して同期送信する場合を参照されたい。
Consumer 側	メッセージ送信するオブジェクトを返却する。

ヘッダ属性 `JMSReplyTo` は Consumer 側で指定したデフォルトの Destination よりも優先される。詳細については、Response management を参照されたい。

非同期受信するメッセージを限定する場合

受信時にメッセージセレクタを指定することで受信するメッセージを限定することができる。

- `[projectName]-web/src/main/java/com/example/listener/todo/TodoMessageListener.java`

```
@JmsListener(destination = "jms/queue/MessageQueue" , selector = "TodoStatus
↪= 'deleted'")     // (1)
public void receive(Todo todo) {
    // omitted
}
```

項番	説明
(1)	selector 属性を利用することで受信対象の条件を設定することができる。 ヘッダ属性の TodoStatus が deleted のメッセージのみ受信する。 メッセージセレクタは SQL92 条件式構文のサブセットに基づいている。 詳細は Message Selectors を参照されたい。

非同期受信したメッセージの入力チェック

セキュリティなどの観点から、不正なデータを保持したメッセージを業務ロジック内で処理しないよう、入力チェックを行うべきである。

Method Validation を利用して Service のメソッドで入力チェックを実装し、入力チェックエラー時の例外をリスナーメソッドでハンドリングする。

これは、トランザクション管理を行う場合に、入力チェックエラー時の例外によって無用なロールバック処理が起こることを回避するためである。トランザクション管理については、トランザクション管理を参照されたい。

Method Validation の設定や実装方法の詳細は、入力チェックの *Method Validation* を参照されたい。

基本的な同期送信で示した Todo のオブジェクトに対して入力チェックを行う実装例を以下に示す。

- [projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java

```
package com.example.domain.service.todo;

import javax.validation.Valid;
import org.springframework.validation.annotation.Validated;
import com.example.domain.model.Todo;

@Validated // (1)
public interface TodoService {

    void updateTodo(@Valid Todo todo); // (2)

}
```

項番	説明
(1)	@Validated アノテーションを付けることで、このインタフェースが入力チェック対象であることを宣言する。
(2)	Bean Validation の制約アノテーションをメソッドの引数として指定する。

- [projectName]-domain/src/main/java/com/example/domain/model/Todo.java

```
package com.example.domain.model;

import java.io.Serializable;
import javax.validation.constraints.Null;

// (1)
public class Todo implements Serializable {

    private static final long serialVersionUID = -1L;

    // omitted

    @Null
    private String description;

    // omitted

    private boolean finished;

    // omitted

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public boolean isFinished() {
        return finished;
    }

    public void setFinished(boolean finished) {
        this.finished = finished;
    }

}
```

項番	説明
(1)	Bean Validation で JavaBean の入力チェックを定義する。 この例では一例として`@Null` アノテーションを設定している。 詳細は「入力チェック」を参照されたい。

- `[projectName]-web/src/main/java/com/example/listener/todo/TodoMessageListener.java`

```
@Inject
TodoService todoService;

@JmsListener(destination = "jms/queue/MessageQueue")
public void receive(Todo todo) {
    try {
        todoService.updateTodo(todo); // (1)
    } catch (ConstraintViolationException e) { // (2)
        // omitted
    }
}
```

項番	説明
(1)	入力チェックを行う Service のメソッドを実行する。
(2)	制約違反時に発生する `ConstraintViolationException` を捕捉する。 捕捉後には任意の処理を実行可能である。 論理的なエラーメッセージを格納するための Queue を利用する場合など、別の Queue にメッセージ送信する例については、非同期受信時の例外ハンドリングを参照されたい。

トランザクション管理

データの一貫性を保証する必要がある場合は、トランザクション管理機能を使用する。

非同期受信で使用する Spring JMS の `DefaultMessageListenerContainer` には、JMS のトランザクション管理の仕組みが組み込まれている。`listener-container` の `acknowledge` 属性でその機能を切

り替えられる。それを利用した場合の実装例を以下に示す。

注釈: メッセージが Queue に戻されると、そのメッセージが再度非同期受信されるため、エラーの原因が解決していない場合は、ロールバック、非同期受信を繰り返すこととなる。JMS プロバイダによっては、ロールバック後の再送信回数に閾値を設定でき、再送信された回数が閾値を超えた場合、Dead Letter Queue にメッセージを格納する。

- `[projectName]-web/src/main/resources/META-INF/spring/applicationContext.xml`

```
<!-- (1) -->
<jms:listener-container
    factory-id="jmsListenerContainerFactory"
    destination-resolver="destinationResolver"
    concurrency="1"
    error-handler="jmsErrorHandler"
    acknowledge="transacted"/>
```

項番	属性名	内容
(1)	cache	Connection、Session や MessageConsumer などのキャッシュ対象を決定するために、キャッシュレベルを指定する。 ここではデフォルトの auto を指定するため、cache 属性を省略している。 基本的な非同期受信の説明を合わせて参照されたい。
	acknowledge	トランザクションを有効にするため、確認応答モードに transacted を指定する。デフォルトは auto である。

警告: アプリケーションサーバによっては、アプリケーション内での Connection や Session のキャッシュを禁止している場合があるため、使用するアプリケーションサーバの仕様に応じてキャッシュの有効化、無効化を決定すること。

警告: 非同期受信と同期送信・受信を併用し、かつ、単一のトランザクションで管理したい場合、jms:listener-container の connection-factory 属性と JmsTemplate の connectionFactory プロパティで指定する ConnectionFactory のインスタンスを同一

にすること。これによって、Spring は非同期受信と同期送受信で利用する Session を共有するため、単一のトランザクションとなる。このとき、jms:listener-container および JmsTemplate の両方でキャッシュを有効にするには、以下のような手段が候補となる。

- JMS 関連リソースのキャッシュを AP サーバ製品に任せ、JNDI ルックアップ経由で取得したオブジェクトを非同期受信と同期送信・受信の両方でそのまま使用する。
- MOM 製品が connectionfactory の cache 機能を持っている場合、それを非同期受信と同期送信・受信の両方でそのまま使用する。
- org.springframework.jms.connection.CachingConnectionFactory を非同期受信と同期送信・受信の両方でそのまま使用する。

いずれの場合も listener-container の cache には none を指定すること。

注釈: 特定の例外の場合にロールバック以外の例外ハンドリングを行う方法

トランザクション管理を有効にした場合、入力チェックなどで発生した例外を捕捉せずに throw すると、ロールバックによってメッセージが Queue に戻される。リスナーメソッドは Queue に戻されたメッセージを再度非同期受信するため、非同期受信 → エラー発生 → ロールバックが JMS プロバイダの設定回数分繰り返されることになる。リトライによってエラーの原因が解消されないような例外の場合は、上記のような無駄な処理を抑えるため、例外を補足してリスナーメソッドから throw しないようにハンドリングを行う。詳細については、非同期受信時の例外ハンドリングを参照されたい。

DB のトランザクション管理を行う必要があるアプリケーションでは、業務の要件をもとに JMS と DB のトランザクションの関連を精査した上でトランザクションの管理方針を決定すること。非同期受信で JMS と DB のトランザクションを連携させるには以下のような方法が考えられる。

1. JTA によるグローバルトランザクションを使用する方法
2. ”Best Effort 1 Phase Commit”を使用する方法
3. JMS と DB のトランザクションを個別に指定する方法

このうち、以下を理由に「JMS と DB のトランザクションを個別に指定する方法」の利用を検討されたい。同期送信のトランザクション管理 (トランザクション管理) でも紹介したように、JTA によるグローバルトランザクションはプロトコルの特性上、性能面のオーバヘッドがかかる。これを解消するため、同期送信では”Best Effort 1 Phase Commit”を使用するトランザクション管理方法を紹介したが、非同期受信ではトランザクションが不適切な構成になるため推奨されない。一般的にリカバリの観点から DB トランザクション境界より JMS トランザクション境界を外側に置く構成をとるが、Spring の DefaultMessageListenerContainer は独自のトランザクション管理機構を持つために、JTA 用の設定である jms:listener-container の

transaction-manager 属性を活用し”Best Effort 1 Phase Commit”を実現しようとすると、DB トランザクション境界が JMS トランザクション境界の外側になってしまう。結果、非同期で受信したメッセージが正常に処理されたにもかかわらず DB トランザクションがロールバックされる可能性が生じる。

> **警告: メッセージ受信後に JMS プロバイダとの接続が切れた場合などで JMS プロバイダにトランザクションの処理結果が返らない場合**
>
> メッセージ受信後に JMS プロバイダとの接続が切れた場合などで、JMS プロバイダにトランザクションの処理結果が返らない場合、トランザクションの扱いは JMS プロバイダに依存する。 そのため、受信したメッセージの消失や、ロールバックによるメッセージの再処理などを考慮した設計を行うこと。 特に、メッセージの消失が許されないような場合には、メッセージの消失を補う仕組みを用意するか、グローバルトランザクションなどの利用を検討する必要がある。

本ガイドラインではグローバルトランザクションは使わずに、上記の通り JMS のトランザクションは Spring JMS が内部で実装しているトランザクション管理に委ね、DB のトランザクションをブランクプロジェクトのデフォルトの設定で定義されている transactionManager で管理する方法を推奨する。その実装例を以下に示す。

- [projectName]-web/src/main/java/com/example/listener/todo/TodoMessageListener.java

```
package com.example.listener.todo;

import javax.inject.Inject;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;
import com.example.domain.service.todo.TodoService;
import com.example.domain.model.Todo;
@Component
public class TodoMessageListener {
    @Inject
    TodoService todoService;

    @JmsListener(destination = "TransactedQueue") // (1)
    public void receiveTransactedMessage(Todo todo) {

        todoService.update(todo);

    }
}
```

- [projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java

```
package com.example.domain.service.todo;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
```

```
import com.example.domain.model.Todo;

@Transactional // (2)
@Service
public class TodoServiceImpl implements TodoService {

    @Override
    public void update(Todo todo) {
        // omitted
    }
}
```

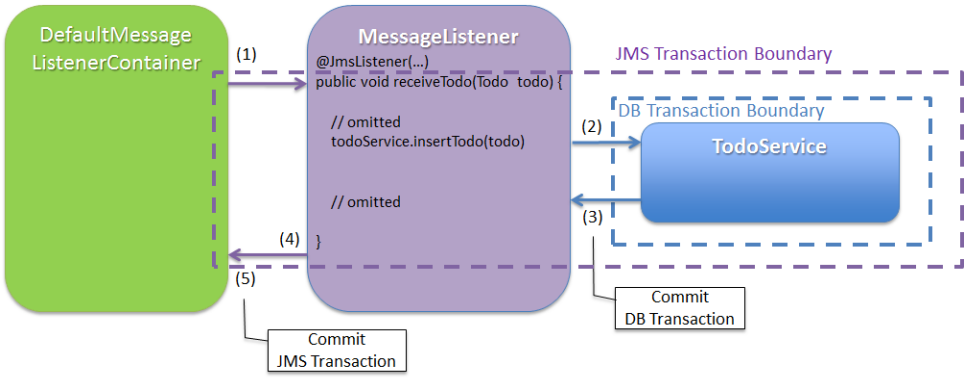
項番	説明
(1)	`@JmsListener` アノテーションを定義し、JMS のトランザクション管理を有効にした `DefaultJmsListenerContainerFactory` を指定する。 `@JmsListener` アノテーションはデフォルトで Bean 名 `jmsListenerContainerFactory` を参照するため、`containerFactory` 属性を省略している。
(2)	DB のトランザクション境界を定義する。 value を省略しているため、デフォルトで、Bean 名 `transactionManager` を参照する。同期送信では `JmsTransactionManager` や `ChainedTransactionManager` の Bean 名を指定したが、非同期受信では JMS のトランザクションは Spring に委ねるため DB のトランザクションマネージャを参照させる。 `@Transactional` アノテーションの詳細については、ドメイン層の実装のトランザクション管理についてを参照されたい。

注釈: トランザクション境界のネストの順序は業務要件によるが、JMS プロバイダは外部システムとの連携に使用される場合が多い。その場合は JMS トランザクション境界を DB トランザクション境界の外側に置き、内向きの DB トランザクションを先に完結する方がリカバリは容易である。DB のトランザクションをコミットし、JMS のトランザクションがロールバックした場合、メッセージが Queue に戻されるため、同じメッセージを再度処理することになる。設計上の考慮点として、業務処理の再実行時に DB 更新処理を再試行しても問題ないように設計する必要がある。

上記の設定、実装例に従ってアプリケーションを作成した場合の挙動について説明する。

- リスナーメソッドの処理が正常に終了した場合

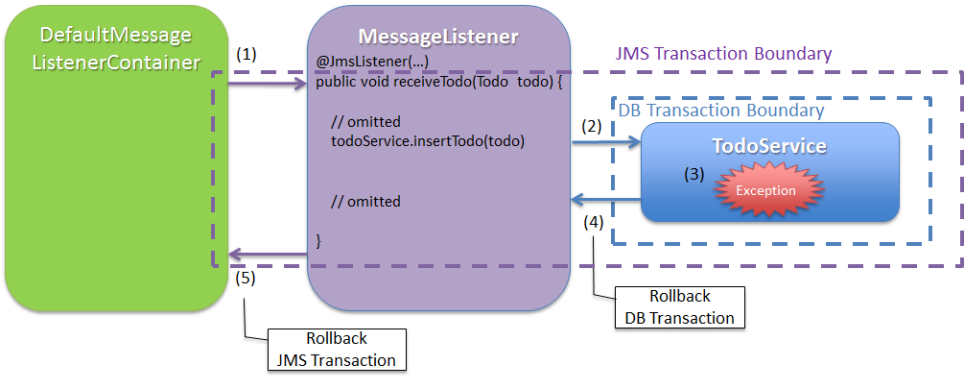
  `DefaultMessageListenerContainer` が JMS トランザクションを開始・コミットし、DB のトランザクションマネージャが DB のトランザクションを開始・コミットする。



項番	説明
(1)	JMS のトランザクションを開始する。
(2)	DB のトランザクションを開始する。
(3)	DB のトランザクションをコミットし、DB のトランザクションを終了する。
(4)	リスナーメソッドが正常終了する。
(5)	JMS のトランザクションをコミットし、JMS のトランザクションを終了する。

- 業務ロジックで予期せぬ例外が発生した場合

  サービスで例外が発生した場合 JMS トランザクションと DB トランザクションの両方をロールバックする。



項番	説明
(1)	JMS のトランザクションを開始する。
(2)	DB のトランザクションを開始する。
(3)	業務ロジックで予期しない例外が発生する。
(4)	DB のトランザクションをロールバックし、DB のトランザクションを終了する。
(5)	JMS のトランザクションをロールバックし、JMS のトランザクションを終了する。 JMS のトランザクションがロールバックするため、メッセージが Queue に戻される。

- **メッセージ受信後に JMS プロバイダとの接続が切れた場合などで、DB のトランザクションのみコミットしてしまう場合**

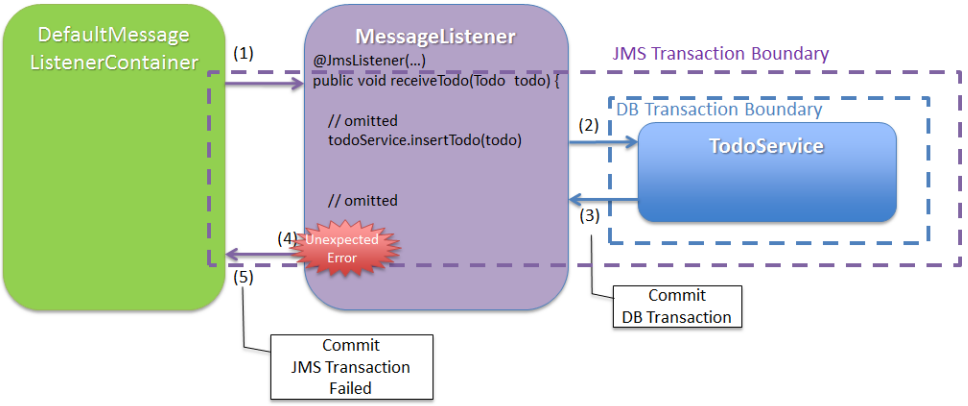
  非同期受信を伴う処理をグローバルトランザクションで管理しない場合は、DB トランザクションと JMS トランザクションは別々にコミットすることになるため、JMS と DB の状態に不整合が生じる可能性がある。具体的には以下の様な場合が該当する。

  - JMS コネクションの切断を検知できずに DB の更新処理を続け、コミットしてしまう場合
  - DB トランザクションのコミット後で JMS トランザクションをコミットする前に例外が発生した場合

  そのような場合に、JMS のトランザクションをロールバックした後に再度同じメッセージを処理することもあれば、送信側によって同一内容のメッセージを複数回送信してしまうことがある。そ

のような背景で同じメッセージを複数受信した場合でもデータの完全性を保障する必要がある。その対策として、JMSMessageID、または、Property や Body に含まれる、リクエストを一意に特定するための情報を記録する方法がある。これは、メッセージの受信ごとに過去に記録した情報と比較し、処理の状況に応じて処理し分けることを意味する。なお、以下のとおり、利用する情報によって対応できる事象に差がある。

- JMSMessageID を記録する場合、メッセージがロールバックされた際の二重処理にのみ対応できる。
- Property や Body の一部を記録する場合、メッセージがロールバックされた際に加えて、異常時などに業務上同一の意味をもつメッセージが複数回送信された際の二重処理にも対応できる。



項番	説明
(1)	JMS のトランザクションを開始する。
(2)	DB のトランザクションを開始する。
(3)	DB のトランザクションをコミットし、DB のトランザクションを終了する。
(4)	JMS のトランザクションのコミット前に JMS プロバイダとの接続が切れるなどの予期せぬエラーが発生する。
(5)	JMS のトランザクションのコミットに失敗する。 そのため、メッセージ消失などに備え、整合性を担保するための仕組みを用意する必要がある。

注釈: 上記のような事象を避け、JMS と DB など複数のトランザクションを厳密に管理する必要がある場合には、グローバルトランザクションの利用を検討する。グローバルトランザクションについては、各種製品マニュアルを参照されたい。

非同期受信時の例外ハンドリング

トランザクション管理を行う場合には、ロールバック処理を考慮した例外のハンドリングを行う必要がある。
トランザクション管理の詳細については、トランザクション管理を参照されたい。
JMS の例外ハンドリングは、目的に応じて以下の 2 種類のパターンに分類される。

表 8.7　表-例外ハンドリングのパターン

項番	ハンドリングの目的	ハンドリング対象となり得る例外の例	ハンドリング方法	ハンドリング単位
(1)	ビジネス層で発生した例外を個別にハンドリングする場合	入力チェックエラーなどのビジネス例外	リスナーメソッド (try-catch)	リスナーメソッド単位
(2)	リスナーメソッドから throw された例外を統一的にハンドリングする場合	入出力エラーなどのシステム例外	`ErrorHandler`	JMSListenerContainer 単位

- ビジネス層で発生した例外を個別にハンドリングする場合

メッセージの内容が不正である場合など、ビジネス層で発生した例外をリスナーメソッドで捕捉 (try-catch) し、リスナーメソッド単位でハンドリングを行う。
トランザクション管理を行う場合、ロールバックが必要なケースは例外を `DefaultMessageListenerContainer` に throw する必要があるため、補足した例外を throw し直すこと。
実装例を以下に示す。

- `[projectName]-web/src/main/java/com/example/listener/todo/TodoMessageListener.java`

```
@Inject
TodoService todoService;

@JmsListener(destination = "jms/queue/TodoMessageQueue")
public JmsResponse<Todo> receiveTodo(Todo todo) {
    try {
        todoService.insertTodo(todo);
    } catch (BusinessException e) {
        return JmsResponse.forQueue(todo, "jms/queue/ErrorMessageQueue"); // (1)
    }
    return null; // (2)
}
```

項番	説明
(1)	JmsResponse クラスの forQueue メソッドを利用し、任意のオブジェクトを論理的なエラーメッセージを格納するための Queue に送信することができる。 この例では、AOP でログ出力が行われる BusinessException を捕捉しているため、明示的にログ出力処理などを記述していないが、例外の原因を消失させないように例外をハンドリングする必要がある。 トランザクション管理を行い、ロールバックしてメッセージの再処理を行いたい場合には、捕捉した例外を throw する必要がある。
(2)	メッセージを送信しない場合は、返り値を null にする。

- リスナーメソッドから **throw** された例外を統一的にハンドリングする場合

例外ごとに共通的なハンドリングを行う場合には、<jms:listener-container/>の error-handler 属性に定義した ErrorHandler の実装クラスを利用する。
設定方法を以下に示す。

- [projectName]-web/src/main/resources/META-INF/spring/applicationContext.xml

```
<!-- (1) -->
<jms:listener-container
    factory-id="jmsListenerContainerFactory"
    destination-resolver="destinationResolver"
    concurrency="1"
    error-handler="jmsErrorHandler"
    acknowledge="transacted"/>

<!-- (2) -->
<bean id="jmsErrorHandler"
    class="com.example.domain.service.todo.JmsErrorHandler">
</bean>
```

項番	説明
(1)	<jms:listener-container/>の error-handler 属性にエラーハンドリングクラスの Bean 名を定義する。
(2)	エラーハンドリングクラスを Bean 定義する。

実装方法を以下に示す。

- [projectName]-web/src/main/java/com/example/listener/todo/JmsErrorHandler.java

```
package com.example.listener.todo;

import org.springframework.util.ErrorHandler;
import org.terasoluna.gfw.common.exception.SystemException;

public class JmsErrorHandler implements ErrorHandler {  // (1)

   @Override
    public void handleError(Throwable t) { // (2)
        // omitted
        if (t.getCause() instanceof SystemException) {  // (3)

            // omitted system error handling

        } else {
            // omitted error handling
        }
    }
}
```

項番	説明
(1)	`ErrorHandler` インタフェースを実装したエラーハンドリングクラスを作成する。
(2)	リスナーメソッド内で発生した例外は `org.springframework.jms.listener.adapter.ListenerExecutionFailedException` にラップされ、引数として渡される。
(3)	任意の例外クラスを判定し、例外に沿ったエラーハンドリングを実施する。 アプリケーション内で発生した例外を取得するには `t.getCause()` を実行する必要がある。

メッセージを同期受信する方法

`JmsMessagingTemplate` を利用して、JMS プロバイダへの同期受信処理を実現する。

同期受信を利用することで、任意のタイミングでメッセージの受信が可能である。

同期受信を利用しない実現方法を十分に検討した上で、アーキテクチャを決定すること。

同期受信の Bean 定義ファイルの設定を以下に示す。

- `[projectName]-env/src/main/resources/META-INF/spring/[projectName]-env.xml`

```
<bean id="cachingConnectionFactory"
   class="org.springframework.jms.connection.CachingConnectionFactory"> <!--␣
↪(1) -->
   <property name="targetConnectionFactory" ref="connectionFactory" /> <!--␣
↪(2) -->
   <property name="sessionCacheSize" value="1" />  <!-- (3) -->
</bean>
```

項番	説明
(1)	Session、MessageProducer,MessageConsumer のキャッシュを行う org.springframework.jms.connection.CachingConnectionFactory を Bean 定義する。 Bean 定義もしくは JNDI 名でルックアップした JMS プロバイダ固有の ConnectionFactory をそのまま使うのではなく、CachingConnectionFactory にラップして使用することで、キャッシュ機能を使用することができる。
(2)	Bean 定義もしくは JNDI 名でルックアップした ConnectionFactory を指定する。
(3)	Session のキャッシュ数を設定する。（デフォルト値は 1） この例では 1 を指定しているが、性能要件に応じて適宜キャッシュ数を変更すること。 このキャッシュ数を超えてセッションが必要になるとキャッシュを使用せず、新しいセッションの作成と破棄を繰り返すことになる。 すると処理効率が下がり、性能劣化の原因になるので注意すること。

- [projectName]-domain/src/main/resources/META-INF/spring/[projectName]-infra.xml

```
<!-- (1) -->
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
   <property name="connectionFactory" ref="cachingConnectionFactory" />
   <property name="destinationResolver" ref="destinationResolver" />
</bean>

<!-- (2) -->
<bean id="jmsMessagingTemplate" class="org.springframework.jms.core.
↪JmsMessagingTemplate">
    <property name="jmsTemplate" ref="jmsTemplate"/>
</bean>
```

項番	説明
(1)	JmsTemplate を Bean 定義する。 JmsTemplate は低レベルの API ハンドリング（JMS API 呼び出し）を代行する。 設定可能な属性に関しては、下記の JmsTemplate の属性一覧を参照されたい。
(2)	JmsMessagingTemplate を Bean 定義する。同期受信処理を代行する JmsTemplate をインジェクションする。

同期受信に関連する JmsTemplate の属性一覧を以下に示す。
必要に応じて設定を行う必要がある。

項番	設定項目	内容	必須	デフォルト値
1.	connectionFactory	使用する ConnectionFactory を設定する。	○	なし（必須であるため）
2.	pubSubDomain	メッセージングモデルについて設定する。 PTP（Queue）モデルは false、Pub/Sub（Topic）は true に設定する。	-	false
3.	sessionTransacted	セッションでのトランザクション管理をするかどうか設定する。 本ガイドラインでは、後述するトランザクション管理を使用するため、デフォルトのままの false を推奨する。	-	false
4.	sessionAcknowledgeMode	sessionAcknowledgeMode はセッションの確認応答モードを設定する。 詳細については JmsTemplate の JavaDoc を参照されたい。	-	1

Continued on next page

表 8.8 – continued from previous page

項番	設定項目	内容	必須	デフォルト値
5.	`receiveTimeout`	同期受信時のタイムアウト時間（ミリ秒）を設定する。未設定の場合、メッセージを受信するまで待機する。 未設定の状態だと、後続の処理に影響が出てしまうため、必ず適切なタイムアウト時間を設定すること。	-	0
6.	`messageConverter`	メッセージコンバータを設定する。 本ガイドラインで紹介している範囲では、デフォルトのままで問題ない。	-	`SimpleMessageCo` が使用される。
7.	`destinationResolver`	DestinationResolver を設定する。 本ガイドラインでは、JNDI で名前解決を行う、`JndiDestinationResolver` を設定することを推奨する。	-	`DynamicDestinat` が使用される。 (`DynamicDestina` を利用すると JMS プロバイダで Destination の名前解決が行われる。)
8.	`defaultDestination`	既定の Destination を設定する。 Destination を明示的に指定しない場合、この Destination が使用される。	-	null(既定の Destination なし)

(*1)org.springframework.jms.support.converter.SimpleMessageConverter

(*2)org.springframework.jms.support.destination.DynamicDestinationResolver

JmsMessagingTemplate クラスの receiveAndConvert メソッドにより、メッセージの同期受信を行う。実装例を以下に示す。

- [projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java

```
package com.example.domain.service.todo;

import javax.inject.Inject;
import org.springframework.jms.core.JmsMessagingTemplate;
import org.springframework.stereotype.Service;
import com.example.domain.model.Todo;

@Service
public class TodoServiceImpl implements TodoService {
    @Inject
    JmsMessagingTemplate jmsMessagingTemplate;

    @Override
    public String receiveTodo() {

        // omitted
        Todo retTodo = jmsMessagingTemplate.receiveAndConvert("jms/queue/
↪TodoMessageQueue", Todo.class);    // (1)

    }
}
```

項番	説明
(1)	JmsMessagingTemplate の receiveAndConvert メソッドにより、指定した Destination からメッセージを受信する。 receiveAndConvert メソッドは、第 2 引数に変換先のクラスを指定することで型変換したクラスが取得できる。 ヘッダ項目を参照する場合は receive メソッドを使用することにより、Spring Framework の Message オブジェクトで取得することができる。

8.2.3 Appendix

JMS プロバイダに依存する設定

JMS プロバイダごとに設定が異なる場合がある。以下に JMS プロバイダごとの設定について説明する。

Apache ActiveMQ を利用する場合

Apache ActiveMQ を利用する場合の設定について説明する。

- アプリケーションサーバに対する JMS プロバイダ固有の設定

  JMS プロバイダによっては、固有の設定が必要な場合がある。

  Apache ActiveMQ では、受信するメッセージのペイロードが許可されたオブジェクトで構成されていることを保障するために、環境変数をアプリケーションサーバの起動引数に追加する必要がある。

  詳細については、ObjectMessage を参照されたい。

  環境変数を Apache Tomcat の起動引数に追加する例を以下に示す。JBoss Enterprise Application Platform 7.2 の場合は Configuring JBoss EAP to Run as a Service を、JBoss Enterprise Application Platform 6.4 の場合は Service Configuration を、Weblogic の場合は Starting Managed Servers with a Startup Script を参照されたい。

  - `$CATALINA_HOME/bin/setenv.sh`

```
# omitted
# (1)
-Dorg.apache.activemq.SERIALIZABLE_PACKAGES=java.lang,java.util,org.apache.
↪activemq,org.fusesource.hawtbuf,com.thoughtworks.xstream.mapper,com.
↪example.domain.model
# omitted
```

項番	説明
(1)	許可する任意のオブジェクトのパッケージを追加する。`java.lang`, `java.util`, `org.apache.activemq`, `org.fusesource.hawtbuf`, `com.thoughtworks.xstream.mapper` は Apache ActiveMQ を使用する場合に必要な設定である。 このサンプルで必要な設定値として、」com.example.domain.model」を追加している。

- ライブラリの追加

  `spring-jms` ライブラリには JMS API が含まれない。

  JMS プロバイダのライブラリには JMS API を含むことが多いが、JMS プロバイダのライブラリに JMS API が含まれない場合は、pom.xml に JMS API を追加する。

domain プロジェクトと web プロジェクトの pom.xml に `activemq-client` をビルド用のライブラリとして追加する。
また、アプリケーションサーバに `activemq-client` とその依存ライブラリを追加する。

- `[projectName]-domain/pom.xml`
- `[projectName]-web/pom.xml`

```
<dependencies>

    <!-- (1) -->
    <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-client</artifactId>
        <scope>provided</scope>
    </dependency>

</dependencies>
```

項番	説明
(1)	Apache ActiveMQ のクライアントライブラリをビルド用として dependencies に追加する。`activemq-client` のライブラリには JMS API も含まれているため、JMS API をライブラリとして追加する必要はない。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

警告: TERASOLUNA Server Framework for Java で使用している Spring IO Platform では、Apache ActiveMQ と接続などを行う際に使用するライブラリのバージョンを定義している。そのため、Apache ActiveMQ のバージョンを決定する際には注意すること。また、TERASOLUNA Server Framework for Java のバージョンアップの際には、ライブラリとミドルウェアのバージョンの整合性が取れなくなる可能性があるので注意すること。

- アプリケーションサーバへの **JNDI** 登録

  アプリケーションサーバへの JNDI 登録については、Manually integrating Tomcat and ActiveMQ を参照されたい。

- **JNDI** を使用しない場合の設定

  本ガイドラインでは JNDI による名前解決する方法を推奨しているが、
  アプリケーションサーバ上で動かせない単体テストの実施において、JMS プロバイダと接続する場合などには、JNDI を利用しないケースがある。
  その場合、`ConnectionFactory` の実装クラスの Bean の生成と、JMS プロバイダで Destination の名前解決を行うために `DynamicDestinationResolver` を設定する必要がある。
  ただし、`JmsTemplate` の `destinationResolver` 属性や `DefaultMessageListenerContainer` の `destination-resolver` 属性を省略した場合は、内部的に生成された `DynamicDestinationResolver` が使用されるため、`DynamicDestinationResolver` の Bean 定義を省略可能である。
  また、Queue についても JNDI を用いて指定していたが、JMS プロバイダーの機能を用いて Destination に指定した Queue が存在しない場合に、指定した名前の Queue を動的に生成させることができる。
  アプリケーションサーバを介さずに接続を行うには Apache ActiveMQ の内部 Broker を用いる必要がある。
  Apache ActiveMQ の内部 Broker の設定については How do I embed a Broker inside a Connection を参照されたい。
  テスト用のコンテキストに下記の設定を追加すること。

  - `[projectName]-env/src/main/resources/META-INF/spring/[projectName]-env.xml`

```
<!-- (1) -->
<bean id="connectionFactory" class="org.apache.activemq.
↪ActiveMQConnectionFactory">
    <constructor-arg value="tcp://localhost:61616"/>  <!-- (2) -->
</bean>

<!-- (3) -->
<bean id="destinationResolver" class="org.springframework.jms.support.
↪destination.DynamicDestinationResolver">
</bean>
```

項番	説明
(1)	Apache ActiveMQ の `ConnectionFactory` を Bean 定義する。
(2)	Apache ActiveMQ の起動 URL を指定する。起動 URL は各環境に沿った値を設定する。
(3)	`DynamicDestinationResolver` を Bean 定義する。 Destination が指定されていない場合には、省略可能である。

同一メッセージの大量送信

同一メッセージの大量送信の実装において `JmsMessageTemplate` を使用する場合、メモリ使用量が増えてしまう可能性がある。

そのため、`JmsTemplate` クラスの `send` メソッドを使用して実装を行うことを検討する必要がある。

理由としては、`JmsMessageTemplate` ではメッセージ送信処理を行うたびに `org.springframework.jms.core.MessageCreator` というクラスのインスタンスが生成されてしまう。

無駄なインスタンスの生成を防ぐために、送信処理時に `MessageCreator` のインスタンスが生成されない `JmsTemplate` クラスの `send` メソッドで送信を行うことでメモリの使用量を削減するようにする。

以下に、ある文字列を同一 Destination に 100 件送信を行う場合コード例を示す。

- `[projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java`

```
package com.example.domain.service.todo;

import java.io.IOException;
import javax.inject.Inject;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.stereotype.Service;

@Service
public class TodoServiceImpl implements TodoService {

    @Inject
    JmsTemplate jmsTemplate; // (1)
```

```
    @Override
    public void sendManyMessage(final String messageStr) throws IOException {
        MessageCreator mc = new MessageCreator() { // (2)
            public Message createMessage(Session session) throws JMSException
↪{
                TextMessage message = session.createTextMessage(); // (3)
                message.setText(messageStr);

                // omitted
                return message;
            }
        };
        for (int i = 0; i < 100; i++) {
            jmsTemplate.send("jms/queue/TodoMessageQueue", mc); // (4)
        }
    }
}
```

項番	説明
(1)	JmsMessagingTemplate を使用すると、送信のたびに MessageCreater の生成が行われてしまうため、MessageCreater の生成を送信と分離して定義できる JmsTemplate を利用する。
(2)	JMS の Message を作成するために MessageCreator のインスタンスを生成する。
(3)	JmsTemplate クラスの send メソッドでメッセージを送信することで、ループごとに MessageCreator のインスタンスを生成が行われなくなり、メモリの使用量を削減させることができるようになる。

サイズの大きなデータの送受信

画像データなどサイズの大きなデータ (目安として 1 MB 以上) を扱う場合、同時トランザクション数やヒープサイズによっては OutOfMemoryError が発生する可能性がある。JMS の標準 API ではサイズの大きなデータをストリームとして扱うことができるのはプリミティブ型のデータの送信を行う StreamMessage と未解釈のバイトストリームの送信を行える ByteMessage のみである。そのため、JMS API ではなく、JMS プロバイダベンダ毎に用意している固有の API を使用するケースがある。

Apache ActiveMQ を利用する場合

Blob Message を使用することでサイズの大きなメッセージを送受信することができる。実装例を以下に示す。

注釈: org.apache.activemq.BlobMessage を使用する場合、Apache ActiveMQ 独自の API を使用することになるため、Spring Framework が提供している Message や CachingConnectionFactory を使用することはできない。性能影響を考慮し、BlobMessage を使用する場合は BlobMessage 用の JmsTemplate を別途定義することを推奨する。

- 設定

BlobMessage を用いたメッセージの送信では、メッセージはヒープ領域ではなく、一時的に Apache ActiveMQ が起動しているサーバに格納される。メッセージの格納先の定義例を以下に示す。

  - [projectName]-env/src/main/resources/META-INF/spring/[projectName]-env.xml

```
<bean id="connectionFactory"
   class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL">
      <!-- (1) -->
      <value>tcp://localhost:61616?jms.blobTransferPolicy.uploadUrl=/tmp</
↪value>
    </property>
</bean>
```

項番	説明
(1)	一時的にメッセージを格納する Apache ActiveMQ のサーバのディレクトリを定義する。 jms.blobTransferPolicy.uploadUrl にはデフォルトで http://localhost:8080/uploads/が設定されており、デフォルトか brokerURL をオーバーロードすることで一時ファイルの置き場を指定できる。 例では/tmp に一時的にファイルを格納している。

- 送信

Blob Message を利用した送信クラスの実装例を以下に示す。

  - [projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java

```
package com.example.domain.service.todo;

import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import javax.inject.Inject;
```

```
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import org.apache.activemq.ActiveMQSession;
import org.apache.activemq.BlobMessage;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.stereotype.Service;

@Service
public class TodoServiceImpl implements TodoService {
    @Inject
    JmsTemplate jmsTemplate;

    @Override
    public void sendBlobMessage(String inputFilePath) throws IOException {

        Path path = Paths.get(inputFilePath);
        try (final InputStream inputStream = Files.newInputStream(path)) {

            jmsTemplate.send("jms/queue/TodoMessageQueue", new␣
↪MessageCreator() {
                public Message createMessage(Session session) throws␣
↪JMSException {

                    ActiveMQSession activeMQSession = (ActiveMQSession)␣
↪session;  // (1)

                    BlobMessage blobMessage = activeMQSession.
↪createBlobMessage(inputStream);  // (2)
                    return blobMessage;
                }
            });
        }
    }
}
```

項番	説明
(1)	`BlobMessage` を使用するには Apache ActiveMQ 独自 API である `org.apache.activemq.ActiveMQSession` を使用する。
(2)	`ActiveMQSession` より、送信データを指定して `BlobMessage` を生成する。 `createBlobMessage` メソッドの引数は `File`、`InputStream`、`URL` クラスが指定可能である。

- 受信

  受信クラスの実装例を以下に示す。

  - [projectName]-web/src/main/java/com/example/listener/todo/TodoMessageListener.java

```
package com.example.listener.todo;

import java.io.IOException;
import javax.inject.Inject;
import javax.jms.JMSException;
import org.apache.activemq.BlobMessage;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;
import com.example.domain.service.todo.TodoService;
@Component
public class TodoMessageListener {
    @Inject
    TodoService todoService;
    @JmsListener(destination = "jms/queue/TodoMessageQueue")
    public void receiveBlobMessage(BlobMessage message) throws IOException,
↪ JMSException {
     todoService.fileInputBlobMessage(message);
        // omitted
    }
}
```

  - [projectName]-domain/src/main/java/com/example/domain/service/todo/TodoServiceImpl.java

```
package com.example.domain.service.todo;

import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import org.apache.activemq.BlobMessage;
import org.springframework.stereotype.Service;

@Service
public class TodoServiceImpl implements TodoService {

    @Override
    public void fileInputBlobMessage(BlobMessage message) throws␣
↪IOException {
        try(InputStream is =  message.getInputStream()){    // (1)
            Path path = Paths.get("outputFilePath");
            Files.copy(is, path);
            // omitted
        }
    }
```

```
}
```

項番	説明
(1)	受信した `BlobMessage` より、`InputStream` としてデータを取得する。

第 9 章

セキュリティ対策

9.1 Spring Security 概要

Spring Security は、アプリケーションにセキュリティ対策機能を実装する際に使用するフレームワークである。Spring Security はスタンドアロンなアプリケーションでも利用できるが、サーブレットコンテナにデプロイする Web アプリケーションに対してセキュリティ対策を行う際に利用するのが一般的である。本章では、Spring Security が提供する機能のうち、一般的な Web アプリケーションでの利用頻度が高いと思われる機能にしぼって説明する。

ちなみに：ガイドラインで紹介していない機能

Spring Security は、本ガイドラインで紹介していない機能も多く提供している。Spring Security が提供するすべての機能を知りたい場合は、Spring Security Reference を参照されたい。

注釈：Spring Security のバージョン

本ガイドラインでは、Spring Security 4 を使用することを前提としている。Spring Security が 4.0 にバージョンアップするにあたり、様々な変更が適用されており、以降で記述されるサンプルについても、Spring Security 4 を使用したサンプルとなっている。

変更内容については Migrating from Spring Security 3.x to 4.x (XML Configuration) を参照されたい。

9.1.1 Spring Security の機能

セキュリティ対策の基本機能

Spring Security は、セキュリティ対策の基本機能として以下の機能を提供している。

表 9.1　セキュリティ対策の基本機能

機能	説明
認証機能	アプリケーションを利用するユーザーの正当性を確認する機能。
認可機能	アプリケーションが提供するリソースや処理に対してアクセスを制御する機能。

セキュリティ対策の強化機能

Spring Security では認証と認可という基本的な機能に加え、Web アプリケーションのセキュリティを強化するための機能をいくつか提供している。

表 9.2　セキュリティ対策の強化機能

機能	説明
セッション管理機能	セッションハイジャック攻撃やセッション固定攻撃からユーザーを守る機能、セッションのライフサイクル (生成、破棄、タイムアウト) を制御するための機能。
CSRF 対策機能	クロスサイトリクエストフォージェリ (CSRF) 攻撃からユーザーを守るための機能。
セキュリティヘッダ出力機能	Web ブラウザのセキュリティ対策機能と連携し、ブラウザの機能を悪用した攻撃からユーザーを守るための機能。

9.1.2 Spring Security のアーキテクチャ

各機能の詳細な説明を行う前に、Spring Security のアーキテクチャ概要と Spring Security を構成する主要なコンポーネントの役割を説明する。

注釈: ここで説明する内容は、Spring Security が提供するデフォルトの動作をそのまま利用する場合や、Spring Security のコンフィギュレーションをサポートする仕組みを利用する場合は、開発者が直接意識する必要ない。そのため、まず各機能の使い方を知りたい場合は、本節を読み飛ばしても問題はない。

ただし、ここで説明する内容は、Spring Security のデフォルトの動作をカスタマイズする際に必要になるので、アプリケーションのアーキテクトは一読しておくことを推奨する。

Spring Security のモジュール

まずフレームワークスタックとなっている Spring Security の提供モジュールを紹介する。

フレームワークスタックモジュール群

フレームワークスタックモジュールは、以下の通りである。本ガイドラインでもこれらのモジュールを使用してセキュリティ対策を行う方法について説明する。

表 9.3　フレームワークスタックモジュール群

モジュール名	説明
spring-security-core	認証と認可機能を実現するために必要となるコアなコンポーネントが格納されている。このモジュールに含まれるコンポーネントは、スタンドアロン環境で実行するアプリケーションでも使用することができる。
spring-security-web	Web アプリケーションのセキュリティ対策を実現するために必要となるコンポーネントが格納されている。このモジュールに含まれるコンポーネントは、Web 層 (サーブレット API など) に依存する処理を行う。
spring-security-config	各モジュールから提供されているコンポーネントのセットアップをサポートするためのコンポーネント (コンフィギュレーションをサポートするクラスや XML ネームスペースを解析するクラスなど) が格納されている。このモジュールを使用すると、Spring Security の bean 定義を簡単に行うことができる。
spring-security-taglibs	認証情報や認可機能にアクセスするための JSP タグライブラリが格納されている。
spring-security-acl	Entity などのドメインオブジェクトを Access Control List(ACL) を使用して認可制御するために必要となるコンポーネントが格納されている。本モジュールは依存関係の都合上、フレームワークスタックに含まれているモジュールであるため、本ガイドラインにおいて使用方法の説明は行わない。

要件に合わせて使用するモジュール群

フレームワークスタックではないが、一般的に利用される認証方法などをサポートするために、以下のようなモジュールも提供されている。セキュリティ要件に応じて、これらのモジュールの使用も検討されたい。

表 9.4　要件に合わせて使用するモジュール群

モジュール名	説明
spring-security-remoting	JNDI 経由で DNS にアクセス、Basic 認証が必要な Web サイトにアクセス、Spring Security を使用してセキュリティ対策しているメソッドに RMI 経由でアクセスする際に必要となるコンポーネントが格納されている。
spring-security-aspects	AspectJ を使用して Java のメソッドに認可機能を適用する際、必要となるコンポーネントが格納されている。このモジュールは、AOP として Spring AOP を使う場合は不要である。
spring-security-messaging[*5]	Spring の Web Socket 機能に対してセキュリティ対策を追加するためのコンポーネントが格納されている。
spring-security-data[*5]	Spring Data の機能から認証情報にアクセスできるようにするためのコンポーネントが格納されている。
spring-security-ldap	Lightweight Directory Access Protocol(LDAP) を使用した認証を実現するために必要となるコンポーネントが格納されている。
spring-security-openid	OpenID[*1] を使用した認証を実現するために必要となるコンポーネントが格納されている。
spring-security-cas	Central Authentication Service(CAS)[*2] と連携するために必要となるコンポーネントが格納されている。
spring-security-crypto	暗号化、キーの生成、ハッシュアルゴリズムを利用したパスワードエンコーディングを行うためのコンポーネントが格納されている。このモジュールに含まれるクラスは、フレームワークスタックモジュールである spring-security-core にも含まれている。

テスト用のモジュール

Spring Security 4.0 からはテストを支援するためのモジュールが追加されている。

表 9.5　テスト用のモジュール

モジュール名	説明
spring-security-test[*5]	Spring Security に依存しているクラスのテストを支援するためのコンポーネントが格納されている。このモジュールを使用すると、JUnit テスト時に必要となる認証情報を簡単にセットアップすることができる。また、Spring MVC のテスト用コンポーネント (MockMvc) と連携して使用するコンポーネントも含まれている。

要件に合わせて利用する関連モジュール群

また、いくつかの関連モジュールも提供されている。

[*5] Spring Security 4.0 から追加されたモジュールである。
[*1] OpenID は、簡単に言うと「1 つの ID で複数のサイトにログインできるようにする」ための仕組みである。
[*2] CAS は、OSS として提供されているシングルサインオン用のサーバーコンポーネントである。詳細は https://www.apereo.org/cas を参照されたい。

表 9.6　要件に合わせて利用する主な関連モジュール群

モジュール名	説明
spring-security-oauth2[*3]	OAuth 2.0[*4]の仕組みを使用して API の認可を実現するために必要となるコンポーネントが格納されている。
spring-security-oauth[*3]	OAuth 1.0 の仕組みを使用して API の認可を実現するために必要となるコンポーネントが格納されている。

フレームワーク処理

Spring Security は、サーブレットフィルタの仕組みを使用して Web アプリケーションのセキュリティ対策を行うアーキテクチャを採用しており、以下のような流れで処理を実行している。

項番	説明
(1)	クライアントは、Web アプリケーションに対してリクエストを送る。
(2)	Spring Security の FilterChainProxy クラス (サーブレットフィルタ) がリクエストを受け取り、HttpFirewall インタフェースのメソッドを呼び出して HttpServletRequest と HttpServletResponse に対してファイアウォール機能を組み込む。
(3)	FilterChainProxy クラスは、Spring Security が提供しているセキュリティ対策用の Security Filter(サーブレットフィルタ) クラスに処理を委譲する。
(4)	Security Filter は複数のクラスで構成されており、サーブレットフィルタの処理が正常に終了すると後続のサーブレットフィルタが呼び出される。
(5)	最後の Security Filter の処理が正常に終了した場合、後続処理 (サーブレットフィルタやサーブレットなど) を呼びだし、Web アプリケーション内のリソースへアクセスする。
(6)	FilterChainProxy クラスは、Web アプリケーションから返却されたリソースをクライアントへレスポンスする。

Web アプリケーション向けのフレームワーク処理を構成する主要なコンポーネントは以下の通りである。詳細は Spring Security Reference -The Security Filter Chain-を参照されたい。

[*3] 詳細は http://projects.spring.io/spring-security-oauth/ を参照されたい。

[*4] OAuth 2.0 は、OAuth 1.0 が抱えていた課題 (署名と認証フローの複雑さ、モバイルやデスクトップのクライアントアプリの未対応など) を改善したバージョンで、OAuth 1.0 との後方互換性はない。

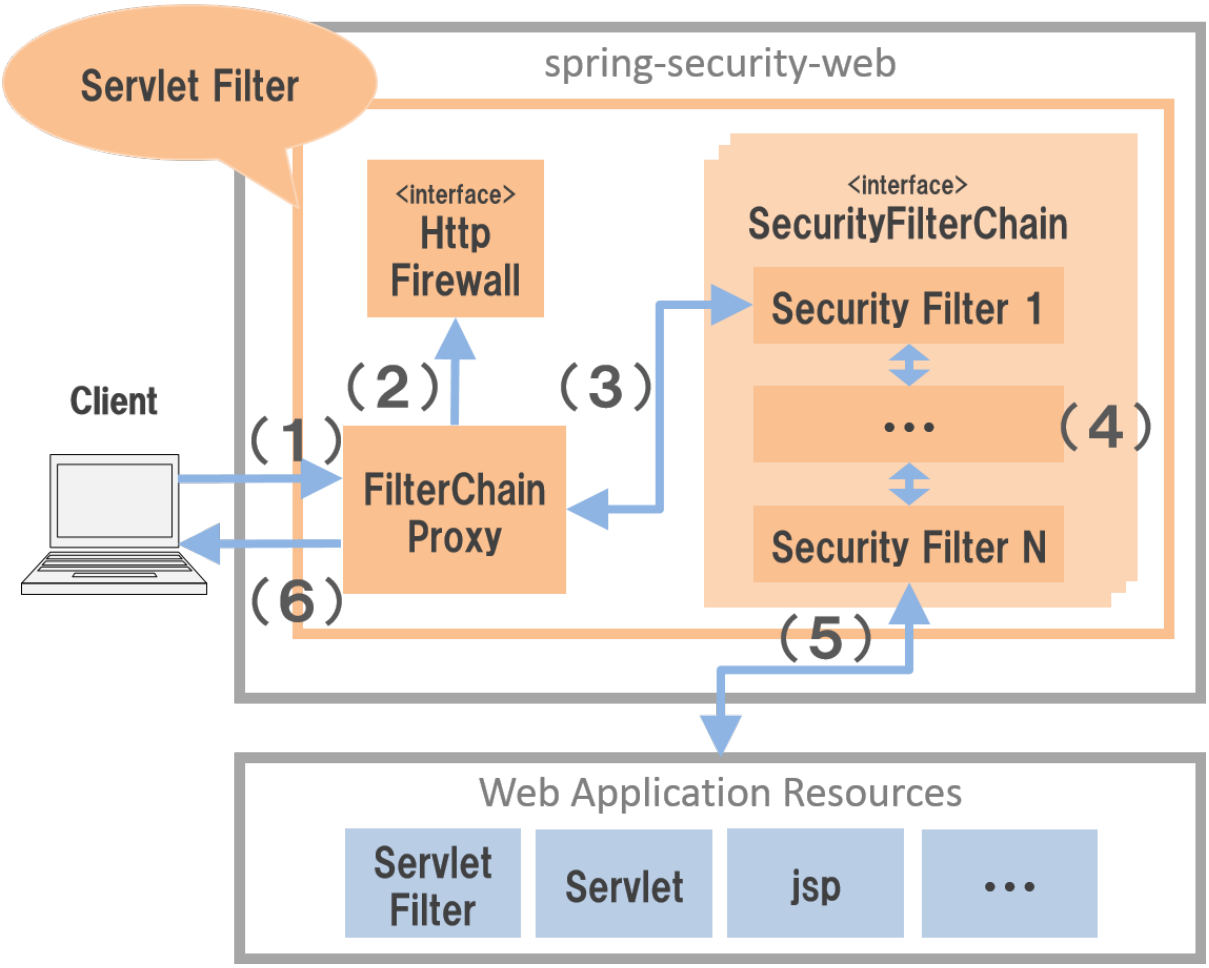


図 9.1　**Spring Security** のフレームワークアーキテクチャ

FilterChainProxy

`FilterChainProxy` クラスは、Web アプリケーション向けのフレームワーク処理のエントリーポイントとなるサーブレットフィルタクラスである。このクラスはフレームワーク処理の全体の流れを制御するクラスであり、具体的なセキュリティ対策処理は Security Filter に委譲している。

HttpFirewall

`HttpFirewall` インタフェースは、`HttpServletRequest` と `HttpServletResponse` に対してファイアウォール機能を組み込むためのインタフェースである。デフォルトでは、`DefaultHttpFirewall` クラスが使用され、ディレクトリトラバーサル攻撃や HTTP レスポンス分割攻撃に対するチェックなどが実装されている。

SecurityFilterChain

`SecurityFilterChain` インタフェースは、`FilterChainProxy` が受け取ったリクエストに対して、適用する Security Filter のリストを管理するためのインタフェースである。デフォルトでは `DefaultSecurityFilterChain` クラスが使用され、適用する Security Filter のリストを、リクエスト URL のパターン毎に管理する。

たとえば、以下のような bean 定義を行うと、URL に応じて異なる内容のセキュリティ対策を適用することが

できる。

- xxx-web/src/main/resources/META-INF/spring/spring-security.xml の定義例

```
<sec:http pattern="/api/**">
    <!-- ... -->
</sec:http>

<sec:http pattern="/ui/**">
    <!-- ... -->
</sec:http>
```

Security Filter

Security Filter クラスは、フレームワーク機能やセキュリティ対策機能を実現する上で必要となる処理を提供するサーブレットフィルタクラスである。

Spring Security は、複数の Security Filter を連鎖させることで Web アプリケーションのセキュリティ対策を行う仕組みになっている。ここでは、認証と認可機能を実現するために必要となるコアなクラスを紹介する。詳細は Spring Security Reference -Core Security Filters-を参照されたい。

表 9.7　コアな **Security Filter**

クラス名	説明
`SecurityContextPersistenceFilter`	認証情報についてリクエストを跨いで共有するための処理を提供するクラス。デフォルトの実装では、`HttpSession` に認証情報を格納することで、リクエストをまたいで認証情報を共有している。
`UsernamePasswordAuthenticationFilter`	リクエストパラメータで指定されたユーザー名とパスワードを使用して認証処理を行うクラス。フォーム認証を行う際に使用する。
`LogoutFilter`	ログアウト処理を行うクラス。
`FilterSecurityInterceptor`	HTTP リクエスト (`HttpServletRequest`) に対して認可処理を実行するためのクラス。
`ExceptionTranslationFilter`	`FilterSecurityInterceptor` で発生した例外をハンドリングし、クライアントへ返却するレスポンスを制御するクラス。デフォルトの実装では、未認証ユーザーからのアクセスの場合は認証を促すレスポンス、認証済みのユーザーからのアクセスの場合は認可エラーを通知するレスポンスを返却する。

9.1.3 Spring Security のセットアップ

Web アプリケーションに Spring Security を適用するためのセットアップ方法について説明する。

ここでは、Web アプリケーションに Spring Security を適用し、Spring Security が提供しているデフォルトのログイン画面を表示させる最もシンプルなセットアップ方法を説明する。実際のアプリケーション開発で必要

となるカスタマイズ方法や拡張方法については、次節以降で順次説明する。

注釈: 開発プロジェクトをブランクプロジェクトから作成すると、ここで説明する各設定はセットアップ済みの状態になっている。開発プロジェクトの作成方法については、「*Web* アプリケーション向け開発プロジェクトの作成」を参照されたい。

依存ライブラリの適用

まず、Spring Security を依存関係として使用している共通ライブラリを適用する。Spring Security と共通ライブラリの関連については、共通ライブラリの構成要素 を参照されたい。

本ガイドラインでは、Maven を使って開発プロジェクトを作成していることを前提とする。

- xxx-domain/pom.xml の設定例

```
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-security-core</artifactId>  <!-- (1) -->
</dependency>
```

- xxx-web/pom.xml の設定例

```
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-security-web</artifactId>  <!-- (2) -->
</dependency>
```

項番	説明
(1)	ドメイン層のプロジェクトで Spring Security の機能を使用する場合は、terasoluna-gfw-security-core を dependency に追加する。
(2)	アプリケーション層のプロジェクトで Spring Security の機能を使用する場合は、terasoluna-gfw-security-web を dependency に追加する。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。

---

bean 定義ファイルの作成

Spring Security のコンポーネントを bean 定義するため、以下のような XML ファイルを作成する。（ブランクプロジェクトより抜粋）

- xxx-web/src/main/resources/META-INF/spring/spring-security.xml の定義例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://www.springframework.org/schema/security"
    xsi:schemaLocation="
        http://www.springframework.org/schema/security https://www.springframework.
↪org/schema/security/spring-security.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
    "> <!-- (1) -->

    <sec:http pattern="/resources/**" security="none"/> <!-- (2) -->
    <sec:http> <!-- (3) -->
        <sec:form-login /> <!-- (4) -->
        <sec:logout /> <!-- (5) -->
        <sec:access-denied-handler ref="accessDeniedHandler"/> <!-- (6) -->
        <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/> <!--
↪ (7) -->
        <sec:session-management /> <!-- (8) -->
    </sec:http>

    <sec:authentication-manager /> <!-- (9) -->

    <!-- CSRF Protection -->
    <bean id="accessDeniedHandler"
        class="org.springframework.security.web.access.
↪DelegatingAccessDeniedHandler"> <!-- (10) -->
        <!-- omitted -->
    </bean>

    <!-- Put UserID into MDC -->
    <bean id="userIdMDCPutFilter" class="org.terasoluna.gfw.security.web.logging.
↪UserIdMDCPutFilter">  <!-- (11) -->
    </bean>

</beans>
```

項番	説明
(1)	Spring Security から提供されている XML ネームスペースを有効にする。上記例では、sec という名前を割り当てている。XML ネームスペースを使用すると、Spring Security のコンポーネントの bean 定義を簡単に行うことができる。
(2)	<sec:http>タグを定義し、セキュリティ対策が不要なリソースパスの設定を行う。詳細は セキュリティ対策を適用しないため設定 を参照されたい。
(3)	<sec:http>タグを定義する。<sec:http>タグを定義すると、Spring Security を利用するために必要となるコンポーネントの bean 定義が自動的に行われる。
(4)	<sec:form-login>タグを定義し、フォーム認証を使用したログインに関する設定を行う。詳細は フォーム認証 を参照されたい。
(5)	<sec:logout>タグ を定義し、ログアウトに関する設定を行う。詳細は ログアウト を参照されたい。
(6)	<sec:access-denied-handler>タグを定義し、アクセスエラー時の制御を行うための設定を定義する。詳細は *AccessDeniedHandler* の適用 、 認可エラー時の遷移先 を参照されたい。
(7)	ログ出力するユーザ情報を MDC に格納するための共通ライブラリのフィルタを定義する。
(8)	<sec:session-management>タグ を定義し、セッション管理に関する設定を行う。詳細は セッション管理 を参照されたい。
(9)	<sec:authentication-manager>タグを定義して、認証機能用のコンポーネントを bean 定義する。このタグを定義しておかないとサーバ起動時にエラーが発生する。
(10)	アクセスエラー時のエラーハンドリングを行うコンポーネントを bean 定義する。
(11)	ログ出力するユーザ情報を MDC にする共通ライブラリのコンポーネントを bean 定義する。

作成した bean 定義ファイルを使用して Spring の DI コンテナを生成するように定義する。

- xxx-web/src/main/webapp/WEB-INF/web.xml の設定例

```
<!-- (1) -->
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<!-- (2) -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath*:META-INF/spring/applicationContext.xml
        classpath*:META-INF/spring/spring-security.xml
    </param-value>
</context-param>
```

項番	説明
(1)	サーブレットコンテナのリスナクラスとして、ContextLoaderListener クラスを指定する。
(2)	サーブレットコンテナの contextClass パラメータに、applicationContext.xml に加えて、Spring Security 用の bean 定義ファイルを追加する。

サーブレットフィルタの設定

最後に、Spring Security が提供しているサーブレットフィルタクラス (FilterChainProxy) をサーブレットコンテナに登録する。

- xxx-web/src/main/webapp/WEB-INF/web.xml の設定例

```
<!-- (1) -->
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
<!-- (2) -->
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

項番	説明
(1)	Spring Framework から提供されている DelegatingFilterProxy を使用して、Spring の DI コンテナで管理されている bean(FilterChainProxy) をサーブレットコンテナに登録する。サーブレットフィルタの名前には、Spring の DI コンテナで管理されている bean の bean 名 (springSecurityFilterChain) を指定する。
(2)	Spring Security を適用する URL のパターンを指定する。上記例では、すべてのリクエストに対して Spring Security を適用する。

セキュリティ対策を適用しないため設定

セキュリティ対策が不要なリソースのパス (css ファイルや image ファイルにアクセスするためのパスなど) に対しては、<sec:http>タグを使用して、Spring Security のセキュリティ機能 (Security Filter) が適用されないように制御することができる。

- xxx-web/src/main/resources/META-INF/spring/spring-security.xml の定義例

```
<sec:http pattern="/resources/**" security="none"/>  <!-- (1) (2) -->
<sec:http>
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	pattern 属性にセキュリティ機能を適用しないパスのパターンを指定する。
(2)	security 属性に none を指定する。 none を指定すると、Spring Security のセキュリティ機能 (Security Filter) が適用されない。

9.2 認証

9.2.1 Overview

本節では、Spring Security が提供している認証機能について説明する。

認証処理は、アプリケーションを利用するユーザーの正当性を確認するための処理である。

ユーザーの正当性を確認するためのもっとも標準的な方法は、アプリケーションを使用できるユーザーをデータストアに登録しておき、利用者が入力した認証情報（ユーザー名とパスワードなど）と照合する方法である。ユーザーの情報を登録しておくデータストアにはリレーショナルデータベースを利用するのが一般的だが、ディレクトリサービスや外部システムなどを利用するケースもある。

また、利用者に認証情報を入力してもらう方式もいくつか存在する。HTML の入力フォームを使う方式や RFC で定められている HTTP 標準の認証方式 (Basic 認証や Digest 認証など) を利用するのが一般的だが、OpenID 認証やシングルサインオン認証などの認証方式を利用するケースもある。

本節では、HTML の入力フォームで入力した認証情報とリレーショナルデータベースに格納されているユーザー情報を照合して認証処理を行う実装例を紹介しながら、Spring Security の認証機能の使い方を説明する。

認証処理のアーキテクチャ

Spring Security は、以下のような流れで認証処理を行う。

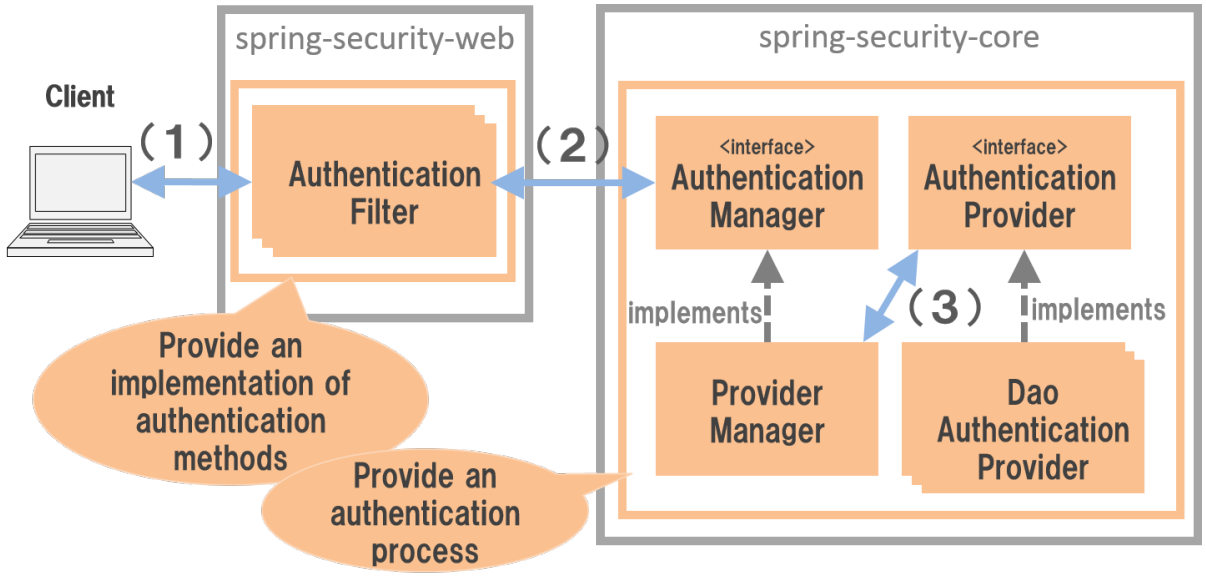


図 9.2　認証処理のアーキテクチャ

項番	説明
(1)	クライアントは、認証処理を行うパスに対して資格情報（ユーザー名とパスワード）を指定してリクエストを送信する。
(2)	Authentication Filter は、リクエストから資格情報を取得して、`AuthenticationManager` クラスの認証処理を呼び出す。
(3)	`ProviderManager`(デフォルトで使用される `AuthenticationManager` の実装クラス) は、実際の認証処理を `AuthenticationProvider` インタフェースの実装クラスに委譲する。

Authentication Filter

Authentication Filter は、認証方式に対する実装を提供するサーブレットフィルタである。Spring Security がサポートしている主な認証方式は以下の通り。

表 9.8 Spring Security が提供している主な Authentication Filter

クラス名	説明
`UsernamePasswordAuthenticationFilter`	フォーム認証用のサーブレットフィルタクラスで、HTTP リクエストのパラメータから資格情報を取得する。
`BasicAuthenticationFilter`	Basic 認証用のサーブレットフィルタクラスで、HTTP リクエストの認証ヘッダから資格情報を取得する。
`DigestAuthenticationFilter`	Digest 認証用のサーブレットフィルタクラスで、HTTP リクエストの認証ヘッダから資格情報を取得する。
`RememberMeAuthenticationFilter`	Remember Me 認証用のサーブレットフィルタクラスで、HTTP リクエストの Cookie から資格情報を取得する。 Remember Me 認証を有効にすると、ブラウザを閉じたりセッションタイムアウトが発生しても、ログイン状態を保つことができる。

これらのサーブレットフィルタは、 フレームワーク処理で紹介した Authentication Filter の 1 つである。

注釈: Spring Security によってサポートされていない認証方式を実現する必要がある場合は、認証方式を実現するための `Authentication Filter` を作成し、Spring Security に組み込むことで実現することが可能である。

AuthenticationManager

`AuthenticationManager` は、認証処理を実行するためのインタフェースである。Spring Security が提供するデフォルト実装 (`ProviderManager`) では、実際の認証処理は `AuthenticationProvider` に委譲し、`AuthenticationProvider` で行われた認証処理の処理結果をハンドリングする仕組みになっている。

AuthenticationProvider

`AuthenticationProvider` は、認証処理の実装を提供するためのインタフェースである。Spring Security が提供している主な `AuthenticationProvider` の実装クラスは以下の通り。

表 9.9　**Spring Security が提供している主な AuthenticationProvider**

クラス名	説明
`DaoAuthenticationProvider`	データストアに登録しているユーザーの資格情報とユーザーの状態をチェックして認証処理を行う実装クラス。 チェックで必要となる資格情報とユーザーの状態は `UserDetails` というインタフェースを実装しているクラスから取得する。

注釈: Spring Security が提供していない認証処理を実現する必要がある場合は、認証処理を実現するための `AuthenticationProvider` を作成し、Spring Security に組み込むことで実現することが可能である。

9.2.2 How to use

認証機能を使用するために必要となる bean 定義例や実装方法について説明する。

本項では *Overview* で説明したとおり、HTML の入力フォームで入力した認証情報とリレーショナルデータベースに格納されているユーザー情報を照合して認証処理を行う方法について説明する。

フォーム認証

Spring Security は、以下のような流れでフォーム認証を行う。

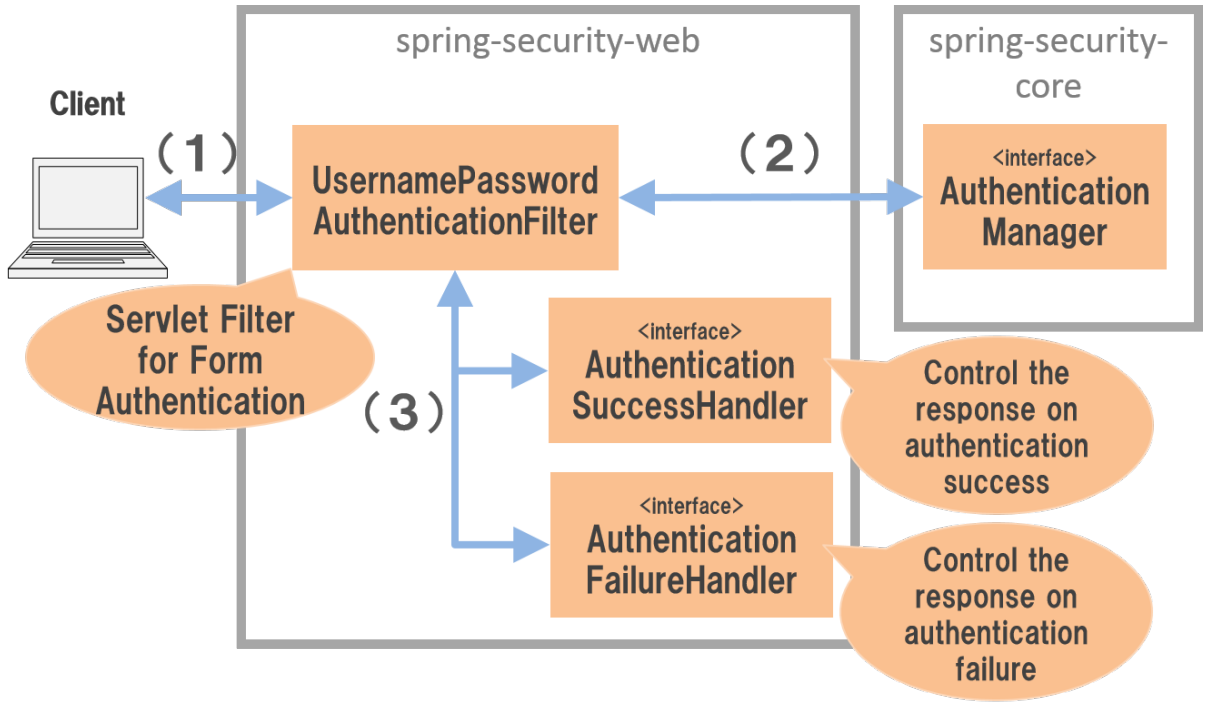


図 9.3　フォーム認証の仕組み

項番	説明
(1)	クライアントは、フォーム認証を行うパスに対して資格情報（ユーザー名とパスワード）をリクエストパラメータとして送信する。
(2)	`UsernamePasswordAuthenticationFilter` クラスは、リクエストパラメータから資格情報を取得して、`AuthenticationManager` の認証処理を呼び出す。
(3)	`UsernamePasswordAuthenticationFilter` クラスは、`AuthenticationManager` から返却された認証結果をハンドリングする。 認証処理が成功した場合は `AuthenticationSuccessHandler` のメソッドを呼び出し、認証処理が失敗した場合は `AuthenticationFailureHandler` のメソッドを呼び出し、画面遷移を行う。

フォーム認証の適用

フォーム認証を使用する場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<sec:http>
    <sec:form-login />     <!-- (1) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	<sec:form-login>タグを定義することで、フォーム認証が有効になる。

ちなみに: auto-config 属性について

<sec:http>には、フォーム認証 (<sec:form-login>タグ)、Basic 認証 (<sec:http-basic>タグ)、ログアウト (<sec:logout>タグ) に対するコンフィギュレーションを自動で行うか否かを指定する auto-config 属性が用意されている。デフォルト値は false(自動でコンフィギュレーションしない) となっており、Spring Security のリファレンスドキュメントでもデフォルト値の使用が推奨されている。

本ガイドラインでも、明示的にタグを指定するスタイルを推奨する。

要素名	説明
<form-login>	フォーム認証処理を行う Security Filter(UsernamePasswordAuthenticationFilter) が適用される。
<http-basic>	RFC1945 に準拠した Basic 認証を行う Security Filter(BasicAuthenticationFilter) が適用される。 詳細な利用方法は、BasicAuthenticationFilter の JavaDoc を参照されたい。
<logout>	ログアウト処理を行う Security Filter(LogoutFilter) が適用される。 ログアウト処理の詳細については、「ログアウト」を参照されたい。

なお、auto-config を定義しない場合は、フォーム認証 (<sec:form-login>タグ)、もしくは Basic 認証 (<sec:http-basic>タグ) を定義する必要がある。これは、ひとつの

SecurityFilterChain(<sec:http>) 内には、ひとつ以上の Authentication Filter の Bean 定義が必要であるという、Spring Security の仕様をみたすためである。

デフォルトの動作

Spring Security のデフォルトの動作では、/login に対して GET メソッドでアクセスすると Spring Security が用意しているデフォルトのログインフォームが表示され、ログインボタンを押下すると/login に対して POST メソッドでアクセスして認証処理を行う。

ログインフォームの作成

Spring Security はフォーム認証用のログインフォームをデフォルトで提供しているが、そのまま利用するケースは少ない。ここでは、自身で作成したログインフォームを Spring Security に適用する方法を説明する。

まず、ログインフォームを表示するための JSP を作成する。ここでは、Spring MVC でリクエストをうけてログインフォームを表示する際の実装例になっている。

- ログインフォームを表示するための JSP の作成例 (xxx-web/src/main/webapp/WEB-INF/views/login/loginForm.jsp)

```
<%@ page contentType="text/html;charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<%-- omitted --%>
<div id="wrapper">
    <h3>Login Screen</h3>
    <%-- (1) --%>
    <c:if test="${param.containsKey('error')}">
        <t:messagesPanel messagesType="error"
            messagesAttributeName="SPRING_SECURITY_LAST_EXCEPTION"/> <%-- (2) --%>
    </c:if>
    <form:form action="${pageContext.request.contextPath}/login" method="post"> <%-
↪- (3) --%>
        <table>
            <tr>
                <td><label for="username">User Name</label></td>
                <td><input type="text" id="username" name="username"></td>
            </tr>
            <tr>
                <td><label for="password">Password</label></td>
                <td><input type="password" id="password" name="password"></td>
            </tr>
            <tr>
                <td> </td>
                <td><button>Login</button></td>
            </tr>
```

```
        </table>
    </form:form>
</div>
<%-- omitted --%>
```

項番	説明
(1)	認証エラーを表示するためのエリア。
(2)	認証エラー時に出力させる例外メッセージを出力する。 共通ライブラリで提供している<t:messagesPanel>タグを使用して出力することを推奨する。 <t:messagesPanel>タグの使用方法については、「メッセージ管理」を参照されたい。 なお、認証エラーが発生した場合、Spring Security のデフォルトの設定で使用される、org.springframework.security.web.authentication.SimpleUrlAuthenticationFailureHandler では、認証エラー時に発生した例外オブジェクトを SPRING_SECURITY_LAST_EXCEPTION という属性名で、リダイレクト時はセッション、フォワード時はリクエストスコープに格納する。
(3)	ユーザー名とパスワードを入力するためのログインフォーム。 ここではユーザー名を username、パスワードを passowrd というリクエストパラメータで送信する。 また、<form:form>を使用することで、CSRF 対策用のトークン値がリクエストパラメータで送信される。 CSRF 対策については、「*CSRF 対策*」で説明する。

つぎに、作成したログインフォームを Spring Security に適用する。

- spring-security.xml の定義例

```
<sec:http>
    <sec:form-login
        login-page="/login/loginForm"
        login-processing-url="/login"  /> <!-- (1)(2) -->
    <sec:intercept-url pattern="/login/**" access="permitAll"/>  <!-- (3) -->
    <sec:intercept-url pattern="/**" access="isAuthenticated()"/> <!-- (4) -->
```

```
</sec:http>
```

項番	説明
(1)	login-page 属性にログインフォームを表示するためのパスを指定する。 匿名ユーザーが認証を必要とする Web リソースにアクセスした場合は、この属性に指定したパスにリダイレクトしてログインフォームを表示する。 ここでは、Spring MVC でリクエストを受けてログインフォームを表示している。 詳細は 「*Spring MVC でリクエストを受けてログインフォームを表示する*」を参照されたい。
(2)	login-processing-url 属性に認証処理を行うためのパスを指定する。 デフォルトのパスも/login であるが、ここでは明示的に指定することとする。
(3)	ログインフォームが格納されている/login パス配下に対し、すべてのユーザーがアクセスできる権限を付与する。 Web リソースに対してアクセスポリシーの指定方法については、「認可」を参照されたい。
(4)	アプリケーションで扱う Web リソースに対してアクセス権を付与する。 上記例では、Web アプリケーションのルートパスの配下に対して、認証済みユーザーのみがアクセスできる権限を付与している。 Web リソースに対してアクセスポリシーの指定方法については、「認可」を参照されたい。

注釈: **Spring Security 4.0 における変更**

Spring Security 4.0 から、以下の設定のデフォルト値が変更されている

- username-parameter
- password-parameter
- login-processing-url
- authentication-failure-url

認証成功時のレスポンス

Spring Security は、認証成功時のレスポンスを制御するためのコンポーネントとして、AuthenticationSuccessHandler というインタフェースと実装クラスを提供している。

表 9.10 AuthenticationSuccessHandler の実装クラス

実装クラス	説明
SavedRequestAwareAuthenticationSuccessHandler	認証前にアクセスを試みた URL にリダイレクトを行う実装クラス。 デフォルトで使用される実装クラス。
SimpleUrlAuthenticationSuccessHandler	defaultTargetUrl にリダイレクト又はフォワードを行う実装クラス。

デフォルトの動作

Spring Security のデフォルトの動作では、認証前にアクセスを拒否したリクエストを HTTP セッションに保存しておいて、認証が成功した際にアクセスを拒否したリクエストを復元してリダイレクトする。認証したユーザーにリダイレクト先へのアクセス権があればページが表示され、アクセス権がなければ認可エラーとなる。この動作を実現するために使用されるのが、SavedRequestAwareAuthenticationSuccessHandler クラスである。

ログインフォームを明示的に表示してから認証処理を行った後の遷移先は、Spring Security のデフォルトの設定では Web アプリケーションのルートパス (「/」) となっているため、認証成功時は Web アプリケーションのルートパスにリダイレクトされる。

認証失敗時のレスポンス

Spring Security は、認証失敗時のレスポンスを制御するためのコンポーネントとして、AuthenticationFailureHandler というインタフェースと実装クラスを提供している。

表 9.11 AuthenticationFailureHandler の実装クラス

実装クラス	説明
SimpleUrlAuthenticationFailureHandler	指定したパス (defaultFailureUrl) にリダイレクト又はフォワードを行う実装クラス。
ExceptionMappingAuthenticationFailureHandler	認証例外と遷移先の URL をマッピングすることができる実装クラス。 Spring Security はエラー原因毎に発生する例外クラスが異なるため、この実装クラスを使用するとエラーの種類毎に遷移先を切り替えることが可能である。
DelegatingAuthenticationFailureHandler	認証例外と AuthenticationFailureHandler をマッピングすることができる実装クラス。 ExceptionMappingAuthenticationFailureHandler と似ているが、認証例外毎に AuthenticationFailureHandler を指定できるので、より柔軟な振る舞いをサポートすることができる。

デフォルトの動作

Spring Security のデフォルトの動作では、ログインフォームを表示するためのパスに error というクエリパラメータが付与された URL にリダイレクトする。

例として、ログインフォームを表示するためのパスが/login の場合は/login?error にリダイレクトされる。

DB 認証

Spring Security は、以下のような流れで DB 認証を行う。

項番	説明
(1)	Spring Security はクライアントからの認証依頼を受け、DaoAuthenticationProvider の認証処理を呼び出す。

Continued on next page

表 9.12 – continued from previous page

項番	説明
(2)	DaoAuthenticationProvider は、UserDetailsService のユーザー情報取得処理を呼び出す。
(3)	UserDetailsService の実装クラスは、データストアからユーザー情報を取得する。
(4)	UserDetailsService の実装クラスは、データストアから取得したユーザー情報から UserDetails を生成する。
(5)	DaoAuthenticationProvider は、UserDetailsService から返却された UserDetails とクライアントが指定した認証情報との照合を行い、クライアントが指定したユーザーの正当性をチェックする。

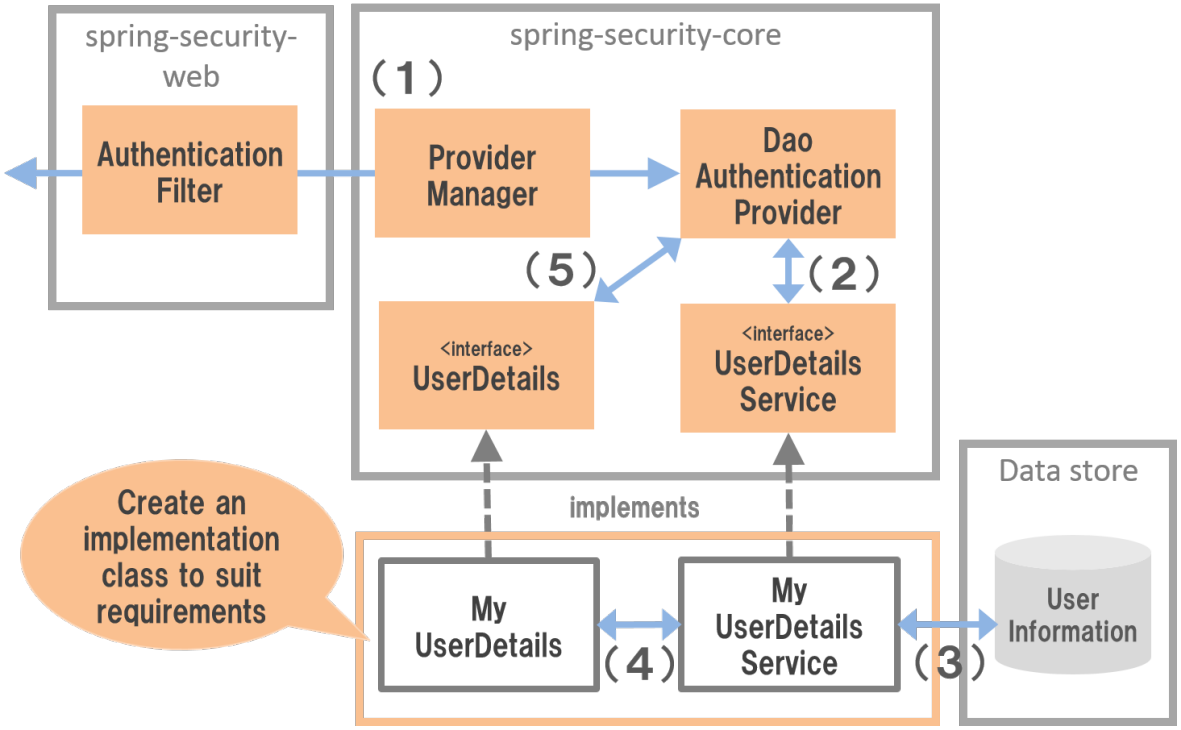


図 9.4　**DB** 認証の仕組み

注釈: **Spring Security** が提供する **DB** 認証

Spring Security は、ユーザー情報をリレーショナルデータベースから JDBC 経由で取得するための実装クラスを提供している。

- `org.springframework.security.core.userdetails.User`(`UserDetails` の実装クラス)
- `org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl` (`UserDetailsService` の実装クラス)

これらの実装クラスは最低限の認証処理 (パスワードの照合、有効ユーザーの判定) しか行わないため、そのまま利用できるケースは少ない。そのため、本ガイドラインでは、`UserDetails` と `UserDetailsService` の実装クラスを作成する方法について説明する。

UserDetails の作成

`UserDetails` は、認証処理で必要となる資格情報 (ユーザー名とパスワード) とユーザーの状態を提供するためのインタフェースで、以下のメソッドが定義されている。`AuthenticationProvider` として `DaoAuthenticationProvider` を使用する場合は、アプリケーションの要件に合わせて `UserDetails`

の実装クラスを作成する。

UserDetails インタフェース

```
public interface UserDetails extends Serializable {
    String getUsername(); // (1)
    String getPassword(); // (2)
    boolean isEnabled(); // (3)
    boolean isAccountNonLocked(); // (4)
    boolean isAccountNonExpired(); // (5)
    boolean isCredentialsNonExpired(); // (6)
    Collection<? extends GrantedAuthority> getAuthorities(); // (7)
}
```

項番	メソッド名	説明
(1)	getUsername	ユーザー名を返却する。
(2)	getPassword	登録されているパスワードを返却する。 このメソッドで返却したパスワードとクライアントから指定されたパスワードが一致しない場合は、DaoAuthenticationProvider は BadCredentialsException を発生させる。
(3)	isEnabled	有効なユーザーかを判定する。有効な場合は true を返却する。 無効なユーザーの場合は、DaoAuthenticationProvider は DisabledException を発生させる。
(4)	isAccountNonLocked	アカウントのロック状態を判定する。ロックされていない場合は true を返却する。 アカウントがロックされている場合は、DaoAuthenticationProvider は LockedException を発生させる。
(5)	isAccountNonExpired	アカウントの有効期限の状態を判定する。有効期限内の場合は true を返却する。 有効期限切れの場合は、DaoAuthenticationProvider は AccountExpiredException を発生させる。

Continued on next page

表 9.13 – continued from previous page

項番	メソッド名	説明
(6)	isCredentialsNonExpired	資格情報の有効期限の状態を判定する。有効期限内の場合は true を返却する。 有効期限切れの場合は、DaoAuthenticationProvider は CredentialsExpiredException を発生させる。
(7)	getAuthorities	ユーザーに与えられている権限リストを返却する。 このメソッドは認可処理で使用される。

注釈: 認証例外による遷移先の切り替え

DaoAuthenticationProvider が発生させる例外毎に画面遷移を切り替えたい場合は、AuthenticationFailureHandler として ExceptionMappingAuthenticationFailureHandler を使用すると実現することができる。

例として、ユーザーのパスワードの有効期限が切れた際にパスワード変更画面に遷移させたい場合は、ExceptionMappingAuthenticationFailureHandler を使って CredentialsExpiredException をハンドリングすると画面遷移を切り替えることができる。

詳細は、認証失敗時のレスポンスのカスタマイズを参照されたい。

注釈: Spring Security が提供する資格情報

Spring Security は、資格情報 (ユーザー名とパスワード) とユーザーの状態を保持するための実装クラス (org.springframework.security.core.userdetails.User) を提供しているが、このクラスは認証処理に必要な情報しか保持することができない。一般的なアプリケーションでは、認証処理で使用しないユーザーの情報（ユーザーの氏名など）も必要になるケースが多いため、User クラスをそのまま利用できるケースは少ない。

ここでは、アカウントの情報を保持する UserDetails の実装クラスを作成する。本例は User を継承することでも実現することができるが、UserDetails を実装する方法の例として紹介している。

- UserDetails の実装クラスの作成例

```
public class AccountUserDetails implements UserDetails { // (1)

    private final Account account;
    private final Collection<GrantedAuthority> authorities;

    public AccountUserDetails(
        Account account, Collection<GrantedAuthority> authorities) {
        // (2)
        this.account = account;
        this.authorities = authorities;
    }

    // (3)
    public String getPassword() {
        return account.getPassword();
    }
    public String getUsername() {
```

```
        return account.getUsername();
    }
    public boolean isEnabled() {
        return account.isEnabled();
    }
    public Collection<GrantedAuthority> getAuthorities() {
        return authorities;
    }

    // (4)
    public boolean isAccountNonExpired() {
        return true;
    }
    public boolean isAccountNonLocked() {
        return true;
    }
    public boolean isCredentialsNonExpired() {
        return true;
    }

    // (5)
    public Account getAccount() {
        return account;
    }

}
```

項番	説明
(1)	UserDetails インタフェースを実装したクラスを作成する。
(2)	ユーザー情報と権限情報をプロパティに保持する。
(3)	UserDetails インタフェースに定義されているメソッドを実装する。
(4)	本節の例では、「アカウントのロック」「アカウントの有効期限切れ」「資格情報の有効期限切れ」に対するチェックは未実装であるが、要件に合わせて実装されたい。
(5)	認証処理成功後の処理でアカウント情報にアクセスできるようにするために、getter メソッドを用意する。

Spring Security は、UserDetails の実装クラスとして User クラスを提供している。User クラスを継承すると資格情報とユーザーの状態を簡単に保持することができる。

- User クラスを継承した UserDetails 実装クラスの作成例

```
public class AccountUserDetails extends User {

    private final Account account;

    public AccountUserDetails(Account account, boolean accountNonExpired,
            boolean credentialsNonExpired, boolean accountNonLocked,
            Collection<GrantedAuthority> authorities) {
        super(account.getUsername(), account.getPassword(),
                account.isEnabled(), true, true, true, authorities);
        this.account = account;
    }

    public Account getAccount() {
        return account;
    }
}
```

UserDetailsService の作成

UserDetailsService は、認証処理で必要となる資格情報とユーザーの状態をデータストアから取得するためのインタフェースで、以下のメソッドが定義されている。AuthenticationProvider として DaoAuthenticationProvider を使用する場合は、アプリケーションの要件に合わせて UserDetailsService の実装クラスを作成する。

- UserDetailsService インタフェース

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username) throws␣
↪UsernameNotFoundException;
}
```

ここでは、データベースからアカウント情報を検索して、UserDetails のインスタンスを生成するための

サービスクラスを作成する。本サンプルでは、SharedService を使用して、アカウント情報を取得している。SharedService については、*Service の実装*を参照されたい。

- AccountSharedService インタフェースの作成例

```
public interface AccountSharedService {
    Account findOne(String username);
}
```

- AccountSharedService の実装クラスの作成例

```
// (1)
@Service
@Transactional
public class AccountSharedServiceImpl implements AccountSharedService {
    @Inject
    AccountRepository accountRepository;

    // (2)
    @Override
    public Account findOne(String username) {
        Account account = accountRepository.findOneByUsername(username);
        if (account == null) {
            throw new ResourceNotFoundException("The given account is not found!
↪username="
                    + username);
        }
        return account;
    }
}
```

項番	説明
(1)	AccountSharedService インタフェースを実装したクラスを作成し、@Service を付与する。 上記例では、コンポーネントスキャン機能を使って AccountSharedServiceImpl を DI コンテナに登録している。
(2)	データベースからアカウント情報を検索する。 アカウント情報が見つからない場合は、共通ライブラリの例外である ResourceNotFoundException を発生させる。 Repository の作成例については、「*Spring Security チュートリアル*」を参照されたい。

- UserDetailsService の実装クラスの作成例

```
// (1)
@Service
@Transactional
public class AccountUserDetailsService implements UserDetailsService {
    @Inject
    AccountSharedService accountSharedService;

    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {

        try {
            Account account = accountSharedService.findOne(username);
            // (2)
            return new AccountUserDetails(account, getAuthorities(account));
        } catch (ResourceNotFoundException e) {
            // (3)
            throw new UsernameNotFoundException("user not found", e);
        }
    }

    // (4)
    private Collection<GrantedAuthority> getAuthorities(Account account) {
        if (account.isAdmin()) {
            return AuthorityUtils.createAuthorityList("ROLE_USER", "ROLE_ADMIN");
        } else {
            return AuthorityUtils.createAuthorityList("ROLE_USER");
        }
    }
}
```

項番	説明
(1)	UserDetailsService インタフェースを実装したクラスを作成し、@Service を付与する。 上記例では、コンポーネントスキャン機能を使って UserDetailsService を DI コンテナに登録している。
(2)	AccountSharedService を使用してアカウント情報を取得する。 アカウント情報が見つかった場合は、UserDetails を生成する。 上記例では、ユーザー名、パスワード、ユーザーの有効状態をアカウント情報から取得している。
(3)	アカウント情報が見つからない場合は、UsernameNotFoundException を発生させる。
(4)	ユーザーが保持する権限 (ロール) 情報を生成する。ここで生成した権限 (ロール) 情報は、認可処理で使用される。

注釈: 認可で使用する権限情報

Spring Security の認可処理は、ROLE_で始まる権限情報をロールとして扱う。そのため、ロールを使用してリソースへのアクセス制御を行う場合は、 ロールとして扱う権限情報に ROLE_プレフィックスを付与する必要がある。

注釈: 認証例外情報の隠蔽

Spring Security のデフォルトの動作では、UsernameNotFoundException は BadCredentialsException という例外に変換してからエラー処理を行う。BadCredentialsException は、クライアントから指定された資格情報のいずれかの項目に誤りがあることを通知するための例外であり、具体的なエラー理由がクライアントに通知されることはない。

DB 認証の適用

作成した UserDetailsService を使用して認証処理を行うためには、DaoAuthenticationProvider を有効化して、作成した UserDetailsService を適用する必要がある。

- spring-security.xml の定義例

```
<sec:authentication-manager> <!-- (1) -->
    <sec:authentication-provider user-service-ref="accountUserDetailsService"> <!--
↪ (2) -->
        <sec:password-encoder ref="passwordEncoder" /> <!-- (3) -->
    </sec:authentication-provider>
</sec:authentication-manager>

<bean id="passwordEncoder"
    class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" /> <!-
↪- (4) -->
```

項番	説明
(1)	AuthenticationManager を bean 定義する。
(2)	<sec:authentication-manager>要素内に <sec:authentication-provider>要素を定義する。 user-service-ref 属性に「*UserDetailsService の作成*」で作成した AccountUserDetailsService の bean を指定する。 本定義により、デフォルト設定の DaoAuthenticationProvider が有効になる。
(3)	パスワード照合時に使用する PasswordEncoder の bean を指定する。
(4)	パスワード照合時に使用する PasswordEncoder を Bean 定義する。 上記例では、パスワードを BCrypt アルゴリズムでハッシュ化する BCryptPasswordEncoder を定義している。 パスワードのハッシュ化については、「パスワードのハッシュ化」を参照されたい。

パスワードのハッシュ化

パスワードをデータベースなどに保存する場合は、パスワードそのものではなくパスワードのハッシュ値を保存するのが一般的である。

Spring Security は、パスワードをハッシュ化するためのインタフェースと実装クラスを提供しており、認証機能と連携して動作する。

Spring Security が提供するインタフェースには、以下の 2 種類がある。

- `org.springframework.security.crypto.password.PasswordEncoder`
- `org.springframework.security.authentication.encoding.PasswordEncoder`

どちらも `PasswordEncoder` という名前のインタフェースであるが、`org.springframework.security.authentication.encoding` パッケージの `PasswordEncoder` は非推奨になっている。パスワードのハッシュ化要件に制約がない場合は、`org.springframework.security.crypto.password` パッケージの `PasswordEncoder` インタフェースの実装クラスを使用することを推奨する。

org.springframework.security.crypto.password.PasswordEncoder のメソッド定義

```
public interface PasswordEncoder {
    String encode(CharSequence rawPassword);
    boolean matches(CharSequence rawPassword, String encodedPassword);
}
```

表 9.14　**PasswordEncoder** に定義されているメソッド

メソッド名	説明
`encode`	パスワードをハッシュ化するためのメソッド。 アカウントの登録処理やパスワード変更処理などでデータストアに保存するパスワードをハッシュ化する際に使用できる。
`matches`	平文のパスワードとハッシュ化されたパスワードを照合するためのメソッド。 このメソッドは Spring Security の認証処理でも利用されるが、パスワード変更処理などで現在のパスワードや過去に使用していたパスワードと照合する際にも使用できる。

Spring Security は、`PasswordEncoder` インタフェースの実装クラスとして以下の 3 つのいずれかを使用することを推奨している。

表 9.15　PasswordEncoder の実装クラス

実装クラス	説明
BCryptPasswordEncoder	BCrypt アルゴリズムを使用してパスワードのハッシュ化及び照合を行う実装クラス。 詳細は、BCryptPasswordEncoder の JavaDoc を参照されたい。
Pbkdf2PasswordEncoder	PBKDF2 アルゴリズムを使用してパスワードのハッシュ化及び照合を行う実装クラス。Pbkdf2PasswordEncoder は Spring Security 4.1 から追加されたクラスである。 詳細は、Pbkdf2PasswordEncoder の JavaDoc を参照されたい。
SCryptPasswordEncoder	SCrypt アルゴリズムを使用してパスワードのハッシュ化及び照合を行う実装クラス。SCryptPasswordEncoder は Spring Security 4.1 から追加されたクラスである。 詳細は、SCryptPasswordEncoder の JavaDoc を参照されたい。

本節では、BCryptPasswordEncoder の使い方について説明する。

注釈: ガイドラインでは BCryptPasswordEncoder の使い方を紹介しているが、OWASP では FIPS に準ずる PBKDF2 アルゴリズムが推奨されている。

注釈: ブランクプロジェクトで SCryptPasswordEncoder を使用する場合は、デフォルト設定から変更する必要がある。

依存ライブラリとして不足している bcprov-jdk15on を追加する。pom.xml に以下の dependency を追加すれば良い。

pom.xml

```
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcprov-jdk15on</artifactId>
```

```
    <version>${bouncycastle.version}</version> <!-- (1) -->
</dependency>
```

項番	説明
(1)	任意のバージョンを指定する。

applicationContext.xml の既存の PasswordEncoder を変更し、SCryptPasswordEncoder を有効化する。

applicationContext.xml

```
<bean id="passwordEncoder" class="org.springframework.security.crypto.
↪password.SCryptPasswordEncoder">
```

Pbkdf2PasswordEncoder を使用する場合は、applicationContext.xml の変更のみ行えば良い。

注釈: アプリケーションの要件によっては、上記以外の非推奨な PasswordEncoder の実装クラスを利用する必要がある場合もある。「非推奨アルゴリズムの *PasswordEncoder* の利用」では非推奨の実装クラスの一つである MessageDigestPasswordEncoder を利用する方法について解説する

BCryptPasswordEncoder

BCryptPasswordEncoder は、BCrypt アルゴリズムを使用してパスワードのハッシュ化及びパスワードの照合を行う実装クラスである。ソルト には 16 バイトの乱数 (java.security.SecureRandom) が使用され、デフォルトでは 1,024(2 の 10 乗) 回 ストレッチング を行う。

- applicationContext.xml の定義例

```
<bean id="passwordEncoder"
    class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" > <!--
↪ (1) -->
```

```
    <constructor-arg name="strength" value="11" /> <!-- (2) -->
</bean>
```

項番	説明
(1)	passwordEncoder のクラスに BCryptPasswordEncoder を指定する。
(2)	コンストラクタの引数に、ハッシュ化のストレッチング回数のラウンド数を指定する。 本引数は省略可能であり、指定できる値は」4」から 31 である。 なお、未指定時のデフォルト値は 10 である。 本ガイドラインでは説明を省略するが、コンストラクタ引数として java.security.SecureRandom.SecureRandom を指定することも可能である。

警告: SecureRandom の使用について

Linux 環境で SecureRandom を使用する場合、処理の遅延やタイムアウトが発生する場合がある。これは使用する乱数生成器に左右される事象であり、以下のドキュメントに説明がある。

- https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html

本事象が発生する場合は、以下のいずれかの設定を追加することで回避することができる。

- Java コマンド実行時に -Djava.security.egd=file:/dev/urandom を指定する。
- ${JAVA_HOME}/jre/lib/security/java.security 内の securerandom.source=/dev/random を securerandom.source=/dev/urandom に変更する。

Java SE 7 の b19 以前のバージョン (正式リリース前) で本事象が発生する場合は、/dev/urandom ではなく /dev/./urandom を指定する必要がある。ただし、SecureRandom で使用するアルゴリズムが NativePRNG の場合は回避することができない。

BCryptPasswordEncoder を使用して処理を行うクラスでは、PasswordEncoder を DI コンテナからインジェクションして使用する。

```
@Service
@Transactional
public class AccountServiceImpl implements AccountService {
```

```
    @Inject
    AccountRepository accountRepository;

    @Inject
    PasswordEncoder passwordEncoder; // (1)

    public Account register(Account account, String rawPassword) {
        // omitted
        String encodedPassword = passwordEncoder.encode(rawPassword); // (2)
        account.setPassword(encodedPassword);
        // omitted
        return accountRepository.save(account);
    }

}
```

項番	説明
(1)	`PasswordEncoder` をインジェクションする。
(2)	インジェクションした `PasswordEncoder` のメソッドを呼び出す。 ここでは、データストアに保存するパスワードをハッシュ化している。

注釈: ソルト

ハッシュ化対象のデータに追加する文字列のことである。ソルトをパスワードに付与することで、実際のパスワードより桁数が長くなるため、レインボークラックなどのパスワード解析を困難にすることができる。なお、**ソルトはユーザーごとに異なる値（ランダム値等）を設定することを推奨する。**これは、同じソルトを使用していると、ハッシュ値からハッシュ化前の文字列 (パスワード) がわかってしまう可能性があるためである。

注釈: ストレッチング

ハッシュ関数の計算を繰り返し行うことで、保管するパスワードに関する情報を繰り返し暗号化することである。パスワードの総当たり攻撃への対策として、パスワード解析に必要な時間を延ばすために行う。しかし、ストレッチングはシステムの性能に影響を与えるので、システムの性能を考慮してストレッチング回数を決める必要がある。

Spring Security のデフォルトでは 1,024(2 の 10 乗) 回ストレッチングを行うが、この回数はコンストラクタ引数 (`strength`) で変更することができる。`strength` には 4(16 回) から 31(2,147,483,648 回) を指定するこ

とが可能である。ストレッチング回数が多いほどパスワードの強度は増すが、計算量が多くなるため性能にあたえる影響も大きくなる。

認証イベントのハンドリング

Spring Security は、Spring Framework が提供しているイベント通知の仕組みを利用して、認証処理の処理結果を他のコンポーネントと連携する仕組みを提供している。

この仕組みを利用すると、以下のようなセキュリティ要件を Spring Security の認証機能に組み込むことが可能である。

- 認証成功、失敗などの認証履歴をデータベースやログに保存する。
- パスワードを連続して間違った場合にアカウントをロックする。

認証イベントの通知は、以下のような仕組みで行われる。

項番	説明
(1)	Spring Security の認証機能は、認証結果 (認証情報や認証例外) を `AuthenticationEventPublisher` に渡して認証イベントの通知依頼を行う。
(2)	`AuthenticationEventPublisher` インタフェースのデフォルトの実装クラスは 認証結果に対応する認証イベントクラスのインスタンスを生成し、 `ApplicationEventPublisher` に渡してイベントの通知依頼を行う。
(3)	`ApplicationEventPublisher` インタフェースの実装クラスは、`ApplicationListener` インタフェースの実装クラスにイベントを通知する。
(4)	`ApplicationListener` の実装クラスの一つである `ApplicationListenerMethodAdaptor` は、 `@org.springframework.context.event.EventListener` が付与されているメソッドを呼び出してイベントを通知する。

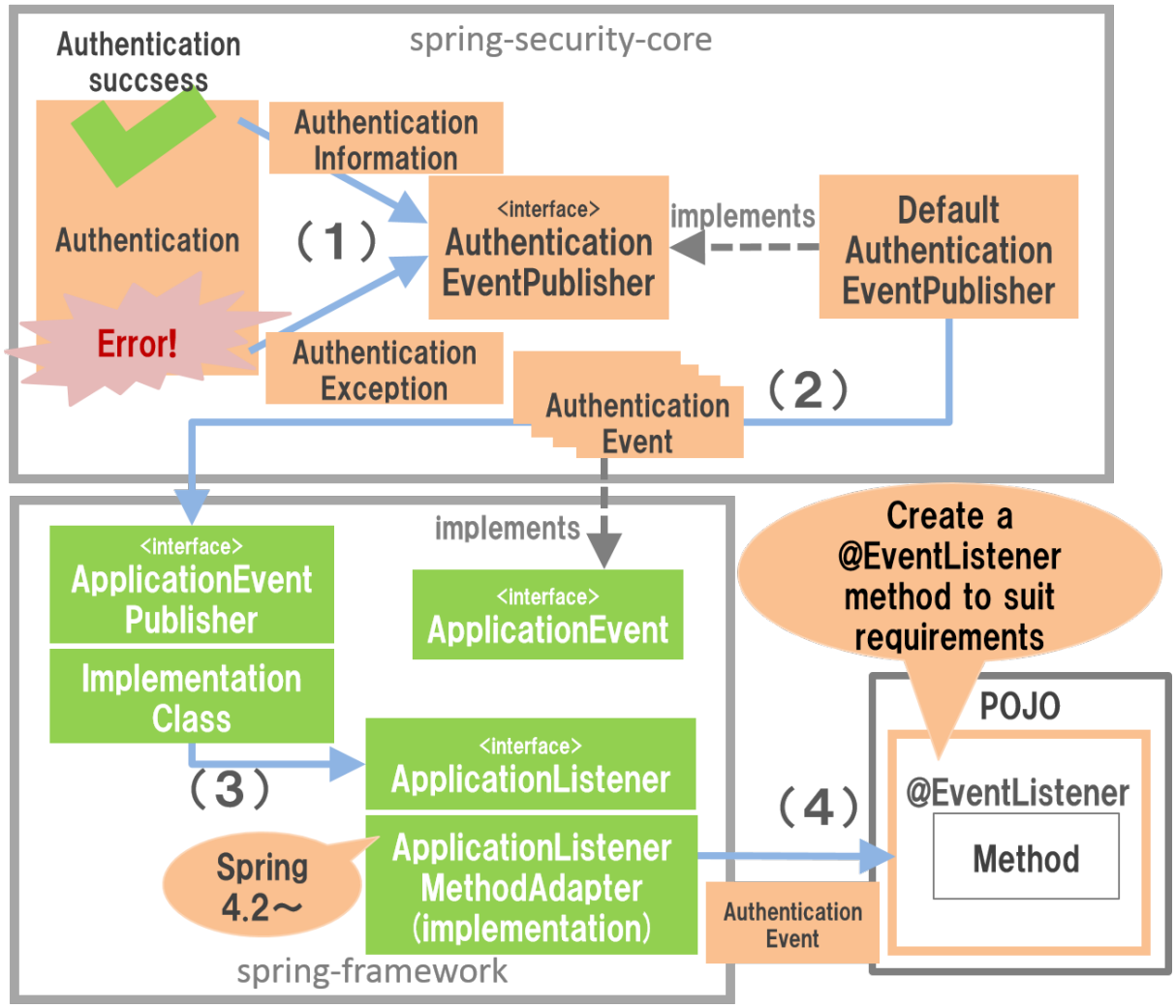


図 9.5　イベント通知の仕組み

注釈: メモ

Spring 4.1 までは `ApplicationListener` インタフェースの実装クラスを作成してイベントを受け取る必要があったが、Spring 4.2 からは POJO に`@EventListener` を付与したメソッドを実装するだけでイベントを受け取ることが可能である。なお、Spring 4.2 以降でも、従来通り `ApplicationListener` インタフェースの実装クラスを作成してイベントを受け取ることも可能である。

Spring Security 使用しているイベントは、認証が成功したことを通知するイベントと認証が失敗したことを通知するイベントの 2 種類に分類される。以下に Spring Security が用意しているイベントクラスを説明する。

認証成功イベント

認証が成功した時に Spring Security が通知する主なイベントは以下の 3 つである。この 3 つのイベントは途中でエラーが発生しなければ、以下の順番ですべて通知される。

表 9.16　認証が成功したことを通知するイベントクラス

イベントクラス	説明
`AuthenticationSuccessEvent`	`AuthenticationProvider` による認証処理が成功したことを通知するためのイベントクラス。このイベントをハンドリングすると、クライアントが正しい認証情報を指定したことを検知することが可能である。なお、このイベントをハンドリングした後の後続処理でエラーが発生する可能性がある点に注意されたい。
`SessionFixationProtectionEvent`	セッション固定攻撃対策の処理 (セッション ID の変更処理) が成功したことを通知するためのイベントクラス。このイベントをハンドリングすると、変更後のセッション ID を検知することが可能になる。
`InteractiveAuthenticationSuccessEvent`	認証処理がすべて成功したことを通知するためのイベントクラス。このイベントをハンドリングすると、画面遷移を除くすべての認証処理が成功したことを検知することが可能になる。

認証失敗イベント

認証が失敗した時に Spring Security が通知する主なイベントは以下の通り。認証に失敗した場合は、いずれか一つのイベントが通知される。

表 9.17　認証が失敗したことを通知するイベントクラス

イベントクラス	説明
AuthenticationFailureBadCredentialsEvent	BadCredentialsException が発生したことを通知するためのイベントクラス。
AuthenticationFailureDisabledEvent	DisabledException が発生したことを通知するためのイベントクラス。
AuthenticationFailureLockedEvent	LockedException が発生したことを通知するためのイベントクラス。
AuthenticationFailureExpiredEvent	AccountExpiredException が発生したことを通知するためのイベントクラス。
AuthenticationFailureCredentialsExpiredEvent	CredentialsExpiredException が発生したことを通知するためのイベントクラス。
AuthenticationFailureServiceExceptionEvent	AuthenticationServiceException が発生したことを通知するためのイベントクラス。

イベントリスナの作成

認証イベントの通知を受け取って処理を行いたい場合は、@EventListener を付与したメソッドを実装したクラスを作成し、DI コンテナに登録する。

- イベントリスナクラスの実装例

```
@Component
public class AuthenticationEventListeners {

```

```
    private static final Logger logger =
            LoggerFactory.getLogger(AuthenticationEventListeners.class);

@EventListener // (1)
public void handleBadCredentials(
    AuthenticationFailureBadCredentialsEvent event) { // (2)
    log.info("Bad credentials is detected. username : {}", event.
↪getAuthentication().getName());
    // omitted
}
```

項番	説明
(1)	`@EventListener` をメソッドに付与したメソッドを作成する。
(2)	メソッドの引数にハンドリングしたい認証イベントクラスを指定する。

上記例では、クライアントが指定した認証情報に誤りがあった場合に通知される `AuthenticationFailureBadCredentialsEvent` をハンドリングするクラスを作成する例としているが、他のイベントも同じ要領でハンドリングすることが可能である。

ちなみに: 総当たり攻撃による不正ログインの兆候を検出するための方法として、ログイン認証時のログを監視することがあげられる。実装例のような `AuthenticationFailureBadCredentialsEvent` をハンドリングするイベントリスナを作成して認証情報の誤りをログ情報として出力することで、Spring Security を使用した認証時のログを監視することが可能になる。

ログアウト

Spring Security は、以下のような流れでログアウト処理を行う。

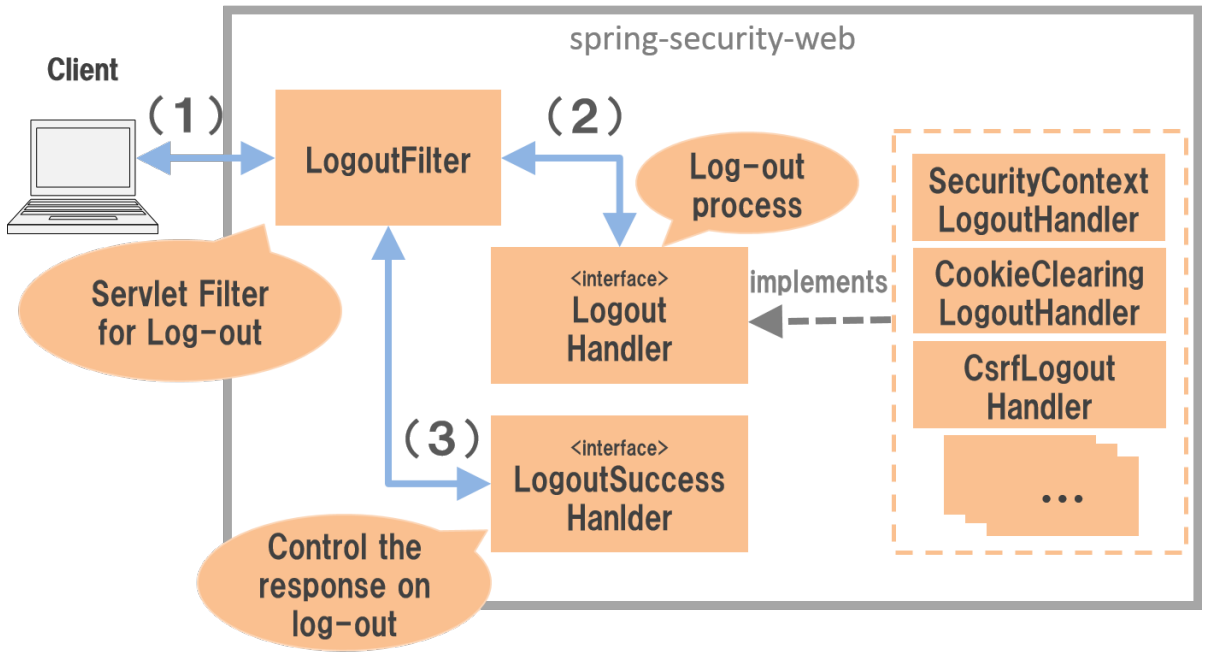


図 9.6　ログアウト処理の仕組み

項番	説明
(1)	クライアントは、ログアウト処理を行うためのパスにリクエストを送信する。
(2)	`LogoutFilter` は、`LogoutHandler` のメソッドを呼び出し、実際のログアウト処理を行う。
(3)	`LogoutFilter` は、`LogoutSuccessHandler` のメソッドを呼び出し、画面遷移を行う。

`LogoutHandler` の実装クラスは複数存在し、それぞれ以下の役割をもっている。

表 9.18　主な LogoutHandler の実装クラス

実装クラス	説明
SecurityContextLogoutHandler	ログインユーザーの認証情報のクリアとセッションの破棄を行うクラス。
CookieClearingLogoutHandler	指定したクッキーを削除するためのレスポンスを行うクラス。
CsrfLogoutHandler	CSRF 対策用トークンの破棄を行うクラス。

これらの LogoutHandler は、Spring Security が提供している bean 定義をサポートするクラスが自動で LogoutFilter に設定する仕組みになっているため、基本的にはアプリケーションの開発者が直接意識する必要はない。また、*Remember Me 認証機能* を有効にすると、Remember Me 認証用の Token を破棄するための LogoutHandler の実装クラスも設定される。

ログアウト処理の適用

ログアウト処理を適用するためには、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<sec:http>
    <!-- omitted -->
    <sec:logout /> <!-- (1) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	<sec:logout>タグを定義することで、ログアウト処理が有効となる。

注釈: **Spring Security 4.0** における変更

Spring Security 4.0 から、以下の設定のデフォルト値が変更されている

- logout-url

ちなみに：Cookie の削除

本ガイドラインでは説明を割愛するが、 `<sec:logout>`タグには、ログアウト時に指定した Cookie を削除するための `delete-cookies` 属性が存在する。ただし、この属性を使用しても正常に Cookie が削除できないケースが報告されている。

詳細は Spring Security の以下の JIRA を参照されたい。

- https://jira.spring.io/browse/SEC-2091?redirect=false

デフォルトの動作

Spring Security のデフォルトの動作では、`/logout` というパスにリクエストを送るとログアウト処理が行われる。ログアウト処理では、「ログインユーザーの認証情報のクリア」「セッションの破棄」が行われる。

また、

- CSRF 対策を行っている場合は、「CSRF 対策用トークンの破棄」
- Remember Me 認証機能を使用している場合は、「Remember Me 認証用の Token の破棄」

も行われる

- ログアウト処理を呼び出すための JSP の実装例

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<%-- omitted --%>
<form:form action="${pageContext.request.contextPath}/logout" method="post"> <%--␣
↪(1) --%>
    <button>ログアウト</button>
</form:form>
```

項番	説明
(1)	ログアウト用のフォームを作成する。 また、`<form:form>`を使用することで、CSRF 対策用のトークン値がリクエストパラメータで送信される。 CSRF 対策については、「*CSRF 対策*」で説明する。

注釈: **CSRF** トークンの送信

CSRF 対策を有効にしている場合は、CSRF 対策用のトークンを POST メソッドで送信する必要がある。

ログアウト成功時のレスポンス

Spring Security は、ログアウト成功時のレスポンスを制御するためのコンポーネントとして、`LogoutSuccessHandler` というインタフェースと実装クラスを提供している。

表 9.19 **AuthenticationFailureHandler** の実装クラス

実装クラス	説明
`SimpleUrlLogoutSuccessHandler`	指定したパス (`defaultTargetUrl`) にリダイレクトを行う実装クラス。

デフォルトの動作

Spring Security のデフォルトの動作では、ログインフォームを表示するためのパスに `logout` というクエリパラメータが付与された URL にリダイレクトする。

例として、ログインフォームを表示するためのパスが`/login` の場合は`/login?logout` にリダイレクトされる。

認証情報へのアクセス

認証されたユーザーの認証情報は、Spring Security のデフォルト実装ではセッションに格納される。セッションに格納された認証情報は、リクエスト毎に `SecurityContextPersistenceFilter` クラスによって `SecurityContextHolder` というクラスに格納され、同一スレッド内であればどこからでもアクセスすることができるようになる。

ここでは、認証情報から `UserDetails` を取得し、取得した `UserDetails` が保持している情報にアクセスする方法を説明する。

Java からのアクセス

一般的な業務アプリケーションでは、「いつ」「誰が」「どのデータに」「どのようなアクセスをしたか」を記録する監査ログを取得することがある。このような要件を実現する際の「誰が」は、認証情報から取得すること

ができる。

- Java から認証情報へアクセスする実装例

```
Authentication authentication =
        SecurityContextHolder.getContext().getAuthentication(); // (1)
String userUuid = null;
if (authentication.getPrincipal() instanceof AccountUserDetails) {
    AccountUserDetails userDetails =
            AccountUserDetails.class.cast(authentication.getPrincipal()); // (2)
    userUuid = userDetails.getAccount().getUserUuid(); // (3)
}
if (log.isInfoEnabled()) {
    log.info("type:Audit\tuserUuid:{}\tresource:{}\tmethod:{}",
            userUuid, httpRequest.getRequestURI(), httpRequest.getMethod());
}
```

項番	説明
(1)	SecurityContextHolder から認証情報 (Authentication オブジェクト) を取得する。
(2)	Authentication#getPrincipal() メソッドを呼び出して、UserDetails オブジェクトを取得する。 認証済みでない場合 (匿名ユーザーの場合) は、匿名ユーザーであることを示す文字列が返却されるため注意されたい。
(3)	UserDetails から処理に必要な情報を取得する。 ここでは、ユーザーを一意に識別するための値 (UUID) を取得している。

警告: 認証情報へのアクセスと結合度

Spring Security のデフォルト実装では、認証情報をスレッドローカルの変数に格納しているため、リクエストを受けたスレッドと同じスレッドであればどこからでもアクセス可能である。この仕組みは便利ではあるが、認証情報を必要とするクラスが SecurityContextHolder クラスに直接依存してしまうため、乱用するとコンポーネントの疎結合性が低下するので注意が必要である。

Spring Security では、Spring MVC の機能と連携してコンポーネント間の疎結合性を保つための仕組みを別途提供している。Spring MVC との連携方法については、「*認証処理と Spring MVC の連携*」で説明する。**本ガイドラインでは Spring MVC との連携を使用して認証情報を取得することを推奨する。**

注釈: 認証処理用のフィルタ (FORM_LOGIN_FILTER) をカスタマイズする場合は、<sec:concurrency-control>要素の指定に加えて、以下の 2 つの SessionAuthenticationStrategy クラスを有効化する必要がある。

- `org.springframework.security.web.authentication.session.ConcurrentSessionControlAuthenticationStrategy`
  認証成功後にログインユーザ毎のセッション数をチェックするクラス。
- `org.springframework.security.web.authentication.session.RegisterSessionAuthenticationStrategy`
  認証に成功したセッションをセッション管理領域に登録するクラス。

version 1.0.x.RELEASE で依存している Spring Security 3.1 では、`org.springframework.security.web.authentication.session.ConcurrentSessionControlStrategy` というクラスが提供されていたが、Spring Security 3.2 より非推奨の API になり、Spring Security 4.0 より廃止になっている。Spring Security 3.1 から Spring Security 3.2 以降にバージョンアップする場合は、以下のクラスを組み合わせて使用するように変更する必要がある。

- `ConcurrentSessionControlAuthenticationStrategy` (Spring Security 3.2 で追加)
- `RegisterSessionAuthenticationStrategy` (Spring Security 3.2 で追加)
- `org.springframework.security.web.authentication.session.SessionFixationProtectionStrategy`

具体的な定義方法については、Spring Security Reference -Web Application Security (Concurrency Control)- のサンプルコードを参考にされたい。

JSP からのアクセス

一般的な Web アプリケーションでは、ログインユーザーのユーザー情報などを画面に表示することがある。このような要件を実現する際のログインユーザーのユーザー情報は、認証情報から取得することができる。

- JSP から認証情報へアクセスする実装例

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<%-- omitted --%>
ようこそ、
<sec:authentication property="principal.account.lastName"/> <%-- (1) --%>
さん。
```

項番	説明
(1)	Spring Security から提供されている<sec:authentication>タグを使用して、認証情報 (Authentication オブジェクト) を取得する。 property 属性にアクセスしたいプロパティへのパスを指定する。 ネストしているオブジェクトへアクセスしたい場合は、プロパティ名を」.」 でつなげればよい。

ちなみに：認証情報の表示方法

ここでは、認証情報が保持するユーザー情報を表示する際の実装例を説明したが、var 属性と scope 属性を組み合わせて任意のスコープ変数に値を格納することも可能である。ログインユーザーの状態によって表示内容を切り替えたい場合は、ユーザー情報を変数に格納しておき、JSTL のタグライブラリなどを使って表示を切り替えることが可能である。

上記の例は、以下のように記述することでも実現することができる。本例では、scope 属性を省略しているため、page スコープが適用される。

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<%-- omitted --%>
<sec:authentication var="principal" property="principal"/>
<%-- omitted --%>
ようこそ、
${f:h(principal.account.lastName)}
さん。
```

認証処理と Spring MVC の連携

Spring Security は、Spring MVC と連携するためのコンポーネントをいくつか提供している。ここでは、認証処理と連携するためのコンポーネントの使い方を説明する。

認証情報へのアクセス

Spring Security は、認証情報 (UserDetails) を Spring MVC のコントローラーのメソッドに引き渡すためのコンポーネントとして、AuthenticationPrincipalArgumentResolver クラスを提供している。AuthenticationPrincipalArgumentResolver を使用すると、コントローラーのメソッド引数として UserDetails インタフェースまたはその実装クラスのインスタンスを受け取ることができるため、コンポーネントの疎結合性を高めることができる。

認証情報 (UserDetails) をコントローラーの引数として受け取るためには、まず

AuthenticationPrincipalArgumentResolver を Spring MVC に適用する必要がある。AuthenticationPrincipalArgumentResolver を適用するための bean 定義は以下の通りである。なお、ブランクプロジェクトには AuthenticationPrincipalArgumentResolver が設定済みである。

- spring-mvc.xml の定義例

```
  <mvc:annotation-driven>
      <mvc:argument-resolvers>
          <!-- omitted -->
          <!-- (1) -->
          <bean class="org.springframework.security.web.method.annotation.
↪AuthenticationPrincipalArgumentResolver" />
          <!-- omitted -->
      </mvc:argument-resolvers>
</mvc:annotation-driven>
```

項番	説明
(1)	HandlerMethodArgumentResolver の実装クラスとして、AuthenticationPrincipalArgumentResolver を Spring MVC に適用する。

認証情報 (UserDetails) をコントローラーのメソッドで受け取る際は、以下のようなメソッドを作成する。

- 認証情報 (UserDetails) を受け取るメソッドの作成例

```
@RequestMapping("account")
@Controller
public class AccountController {

    public String view(
            @AuthenticationPrincipal AccountUserDetails userDetails, // (1)
            Model model) {
        model.addAttribute(userDetails.getAccount());
        return "profile";
    }

}
```

項番	説明
(1)	認証情報 (UserDetails) を受け取るための引数を宣言し、@org.springframework.security.core.annotation.AuthenticationPrincipal を引数アノテーションとして指定する。 AuthenticationPrincipalArgumentResolver は、@AuthenticationPrincipal が付与されている引数に認証情報 (UserDetails) が設定される。

9.2.3 How to extend

本節では、Spring Security が用意しているカスタマイズポイントや拡張方法について説明する。

Spring Security は、多くのカスタマイズポイントを提供しているため、すべてのカスタマイズポイントを紹介することはできないため、ここでは代表的なカスタマイズポイントに絞って説明を行う。

フォーム認証のカスタマイズ

フォーム認証処理のカスタマイズポイントを説明する。

認証パスの変更

Spring Security のデフォルトでは、認証処理を実行するためのパスは「/login」であるが、以下のような bean 定義を行うことで変更することが可能である。

- spring-security.xml の定義例

```
<sec:http>
    <sec:form-login login-processing-url="/authentication" /> <!-- (1) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	login-processing-url 属性に認証処理を行うためのパスを指定する。

注釈: 認証処理のパスを変更した場合は、ログインフォーム のリクエスト先も変更する必要がある。

資格情報を送るリクエストパラメータ名の変更

Spring Security のデフォルトでは、資格情報 (ユーザー名とパスワード) を送るためのリクエストパラメータは「username」と「password」であるが、以下のような bean 定義を行うことで変更することが可能である。

- spring-security.xml の定義例

```
<sec:http>
    <sec:form-login
        username-parameter="uid"
        password-parameter="pwd" /> <!-- (1) (2) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	username-parameter 属性にユーザー名のリクエストパラメータ名を指定する。
(2)	password-parameter 属性にパスワードのリクエストパラメータ名を指定する。

注釈: リクエストパラメータ名を変更した場合は、ログインフォーム 内の項目名も変更する必要がある。

認証成功時のレスポンスのカスタマイズ

認証成功時のレスポンスのカスタマイズポイントを説明する。

デフォルト遷移先の変更

ログインフォームを自分で表示して認証処理を行った後の遷移先 (デフォルト URL) は、Web アプリケーションのルートパス (「/」) だが、以下のような bean 定義を行うことで変更することが可能である。

- spring-security.xml の定義例

```
<sec:http>
    <sec:form-login default-target-url="/menu" /> <!-- (1) -->
</sec:http>
```

項番	説明
(1)	`default-target-url` 属性に認証成功時に遷移するデフォルトのパスを指定する。

遷移先の固定化

Spring Security のデフォルトの動作では、未認証時に認証が必要なページへのリクエストを受信した場合は、受信したリクエストを一旦 HTTP セッションに保存し、認証ページに遷移する。認証成功時にリクエストを復元してリダイレクトするが、以下のような bean 定義を行うことで常に同じ画面に遷移させることが可能である。

- spring-security.xml の定義例

```
<sec:http>
    <sec:form-login
        default-target-url="/menu"
        always-use-default-target="true" /> <!-- (1) -->
</sec:http>
```

項番	説明
(1)	`always-use-default-target` 属性に `true` を指定する。

AuthenticationSuccessHandler の適用

Spring Security が提供しているデフォルトの動作をカスタマイズする仕組みだけでは要件をみたせない場合は、以下のような bean 定義を行うことで `AuthenticationSuccessHandler` インタフェースの実装クラスを直接適用することができる。

- spring-security.xml の定義例

```
<bean id="authenticationSuccessHandler" class="com.example.app.security.handler.
↪MyAuthenticationSuccessHandler"> <!-- (1) -->

<sec:http>
    <sec:form-login authentication-success-handler-ref=
↪"authenticationSuccessHandler" /> <!-- (2) -->
</sec:http>
```

項番	説明
(1)	`AuthenticationSuccessHandler` インタフェースの実装クラスを bean 定義する。
(2)	`authentication-success-handler-ref` 属性に定義した `authenticationSuccessHandler` を指定する。

警告: AuthenticationSuccessHandler の責務

`AuthenticationSuccessHandler` は、認証成功時における Web 層の処理 (主に画面遷移に関する処理) を行うためのインタフェースである。そのため、認証失敗回数のクリアなどのビジネスルールに依存する処理（ビジネスロジック）をこのインタフェースの実装クラスを経由して呼び出すべきではない。

ビジネスルールに依存する処理の呼び出しは、前節で紹介している「認証イベントのハンドリング」の仕組みを使用されたい。

認証失敗時のレスポンスのカスタマイズ

認証失敗時のレスポンスのカスタマイズポイントを説明する。

遷移先の変更

Spring Security のデフォルトの動作では、ログインフォームを表示するためのパスに `error` というクエリパラメータが付与された URL にリダイレクトするが、以下のような bean 定義を行うことで変更することが可能である。

- spring-security.xml の定義例

```
<sec:http>
    <sec:form-login authentication-failure-url="/loginFailure" /> <!-- (1) -->
</sec:http>
```

項番	説明
(1)	authentication-failure-url 属性に認証失敗時に遷移するパスを指定する。

AuthenticationFailureHandler の適用

Spring Security が提供しているデフォルトの動作をカスタマイズする仕組みだけでは要件をみたせない場合は、以下のような bean 定義を行うことで AuthenticationFailureHandler インタフェースの実装クラスを直接適用することができる。

- spring-security.xml の定義例

```
 <!-- (1) -->
<bean id="authenticationFailureHandler"
    class="org.springframework.security.web.authentication.
↪ExceptionMappingAuthenticationFailureHandler" />
    <property name="defaultFailureUrl" value="/login/systemError" /> <!-- (2) -->
    <property name="exceptionMappings"> <!-- (3) -->
        <props>
            <prop key="org.springframework.security.authentication.
↪BadCredentialsException"> <!-- (4) -->
                /login/badCredentials
            </prop>
            <prop key="org.springframework.security.core.userdetails.
↪UsernameNotFoundException"> <!-- (5) -->
                /login/usernameNotFound
            </prop>
            <prop key="org.springframework.security.authentication.
↪DisabledException"> <!-- (6) -->
                /login/disabled
            </prop>
            <!-- omitted -->
        </props>
    </property>
</bean>

<sec:http>
    <sec:form-login authentication-failure-handler-ref=
↪"authenticationFailureHandler" /> <!-- (7) -->
```

```
</sec:http>
```

項番	説明
(1)	AuthenticationFailureHandler インタフェースの実装クラスを bean 定義する。
(2)	defaultFailureUrl 属性にデフォルトの遷移先の URL を指定する。 下記 (4)-(6) の定義に合致しない例外が発生した際は、本設定の遷移先に遷移する。
(3)	exceptionMappings プロパティにハンドルする org.springframework.security.authentication.AuthenticationServiceException の実装クラスと例外発生時の遷移先を Map 形式で設定する。 キーに org.springframework.security.authentication.AuthenticationServiceException 実装クラスを設定し、値に遷移先 URL を設定する。
(4)	BadCredentialsException パスワード照合失敗による認証エラー時にスローされる。
(5)	UsernameNotFoundException 不正ユーザー ID（存在しないユーザー ID）による認証エラー時にスローされる。 org.springframework.security.authentication.dao.AbstractUserDetailsAuthenticationProvider を 継承したクラスを認証プロバイダに指定している場合、 hideUserNotFoundExceptions プロパティを false に変更しないと本例外は、BadCredentialsException に変更される。
(6)	DisabledException 無効ユーザー ID による認証エラー時にスローされる。

Continued on next page

表 9.20 – continued from previous page

項番	説明
(7)	`authentication-failure-handler-ref` 属性に `authenticationFailureHandler` を設定する。

注釈: 例外発生時の制御

`exceptionMappings` プロパティに定義した例外が発生した場合、例外にマッピングした遷移先にリダイレクトされるが、発生した例外オブジェクトがセッションスコープに格納されないため、Spring Security が生成したエラーメッセージを画面に表示する事ができない。

そのため、遷移先の画面で表示するエラーメッセージは、リダイレクト先の処理 (Controller 又は View の処理) で生成する必要がある。

また、以下のプロパティを参照する処理が呼び出されないため、設定値を変更しても動作が変わらないという点を補足しておく。

- `useForward`
- `allowSessionCreation`

ログアウト処理のカスタマイズ

ログアウト処理のカスタマイズポイントを説明する。

ログアウトパスの変更

Spring Security のデフォルトでは、ログアウト処理を実行するためのパスは「`/logout`」であるが、以下のような bean 定義を行うことで変更することが可能である。

- spring-security.xml の定義例

```
<sec:http>
    <!-- omitted -->
    <sec:logout logout-url="/auth/logout" /> <!-- (1) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	`logout-url` 属性を設定し、ログアウト処理を行うパスを指定する。

注釈: ログアウトパスを変更した場合は、ログアウトフォーム のリクエスト先も変更する必要がある。

ちなみに：システムエラー発生時の振る舞いシステムエラーが発生した場合は、業務継続不可となるケースが多いと考えられる。システムエラー発生後、業務を継続させたくない場合は、以下のような対策を講じることを推奨する。

- システムエラー発生時にセッション情報をクリアする。
- システムエラー発生時に認証情報をクリアする。

ここでは、共通ライブラリの例外ハンドリング機能を使用してシステム例外発生時に認証情報をクリアする例を説明する。例外ハンドリング機能の詳細については「例外ハンドリング」を参照されたい。

```
// (1)
public class LogoutSystemExceptionResolver extends SystemExceptionResolver {
    // (2)
    @Override
    protected ModelAndView doResolveException(HttpServletRequest request,
            HttpServletResponse response, java.lang.Object handler,
            java.lang.Exception ex) {

        // SystemExceptionResolver の処理を行う
        ModelAndView resulut = super.doResolveException(request, response,
                handler, ex);

        // 認証情報をクリアする (2)
        SecurityContextHolder.clearContext();

        return resulut;
    }
}
```

項番	説明
(1)	org.terasoluna.gfw.web.exception.SystemExceptionResolver.SystemExceptionResolver を拡張する。
(2)	認証情報をクリアする。

なお、認証情報をクリアする方法以外にも、セッションをクリアすることでも、同様の要件を満たすことができる。プロジェクトの要件に合わせて実装されたい。

ログアウト成功時のレスポンスのカスタマイズ

ログアウト処理成功時のレスポンスのカスタマイズポイントを説明する。

遷移先の変更

- spring-security.xml の定義例

```
<sec:http>
    <!-- omitted -->
    <sec:logout logout-success-url="/logoutSuccess" /> <!-- (1) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	`logout-success-url` 属性を設定し、ログアウト成功時に遷移するパスを指定する。

LogoutSuccessHandler の適用

- spring-security.xml の定義例

```
<!-- (1) -->
<bean id="logoutSuccessHandler" class="com.example.app.security.handler.
↪MyLogoutSuccessHandler" />

<sec:http>
    <!-- omitted -->
    <sec:logout success-handler-ref="logoutSuccessHandler" /> <!-- (2) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	`LogoutSuccessHandler` インタフェースの実装クラスを bean 定義する。
(2)	`success-handler-ref` 属性に `LogoutSuccessHandler` を設定する。

エラーメッセージのカスタマイズ

認証に失敗した場合、Spring Security が用意しているエラーメッセージが表示されるが、このエラーメッセージは変更することが可能である。

メッセージ変更方法の詳細については、メッセージ管理を参照されたい。

システムエラー時のメッセージ

認証処理の中で予期しないエラー（システムエラーなど）が発生した場合、InternalAuthenticationServiceException という例外が発生する。InternalAuthenticationServiceException が保持するメッセージには、原因例外のメッセージが設定されるため、画面にそのまま表示するのは適切ではない。

例えばユーザー情報をデーターベースから取得する時に DB アクセスエラーが発生した場合、SQLException が保持する例外メッセージが画面に表示されることになる。システムエラーの例外メッセージを画面に表示させないためには、ExceptionMappingAuthenticationFailureHandler を使用して InternalAuthenticationServiceException をハンドリングし、システムエラーが発生したことを通知するためのパスに遷移させるなどの対応が必要となる。

- spring-security.xml の定義例

```
  <bean id="authenticationFailureHandler"
      class="org.springframework.security.web.authentication.
↪ExceptionMappingAuthenticationFailureHandler">
      <property name="defaultFailureUrl" value="/login?error" />
      <property name="exceptionMappings">
          <props>
              <prop key="org.springframework.security.authentication.
↪InternalAuthenticationServiceException">
                  /login?systemError
              </prop>
              <!-- omitted -->
          </props>
      </property>
  </bean>

<sec:http>
    <sec:form-login authentication-failure-handler-ref=
↪"authenticationFailureHandler" />
</sec:http>
```

ここでは、システムエラーが発生したことを識別するためのクエリパラメータ (systemError) を付け

てログインフォームに遷移させている。遷移先に指定したログインフォームでは、クエリパラメータに systemError が指定されている場合は、認証例外のメッセージを表示するのではなく、固定のエラーメッセージを表示するようにしている。

- ログインフォームの実装例

```
<c:choose>
    <c:when test="${param.containsKey('error')}">
        <span style="color: red;">
            <c:out value="${SPRING_SECURITY_LAST_EXCEPTION.message}"/>
        </span>
    </c:when>
    <c:when test="${param.containsKey('systemError')}">
        <span style="color: red;">
            System Error occurred.
        </span>
    </c:when>
</c:choose>
```

注釈: ここでは、ログインフォームに遷移させる場合の実装例を紹介したが、システムエラー画面に遷移させてもよい。

認証時の入力チェック

DB サーバへの負荷軽減等で、認証ページにおける、あきらかな入力誤りに対しては、事前にチェックを行いたい場合がある。このような場合は、Bean Validation を使用した入力チェックも可能である。

Bean Validation による入力チェック

以下に Bean Validation を使用した入力チェックの例を説明する。Bean Validation に関する詳細は 入力チェックを参照すること。

- フォームクラスの実装例

```
public class LoginForm implements Serializable {

    // omitted
    @NotEmpty // (1)
    private String username;

    @NotEmpty // (1)
    private String password;
    // omitted

}
```

項番	説明
(1)	本例では、username、password をそれぞれ必須入力としている。

- コントローラクラスの実装例

```
@ModelAttribute
public LoginForm setupForm() { // (1)
    return new LoginForm();
}

@RequestMapping(value = "login")
public String login(@Validated LoginForm form, BindingResult result) {
    // omitted
    if (result.hasErrors()) {
        // omitted
    }
    return "forward:/authenticate"; // (2)
}
```

項番	説明
(1)	LoginForm を初期化する。
(2)	forward で<sec:form-login>要素の login-processing-url 属性に指定したパスに **Forward** する。 認証に関する設定は、フォーム認証のカスタマイズを参照すること。

加えて、Forward による遷移でも Spring Security の処理が行われるよう、認証パスを Spring Security サーブレットフィルタに追加する。

- web.xml の設定例

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

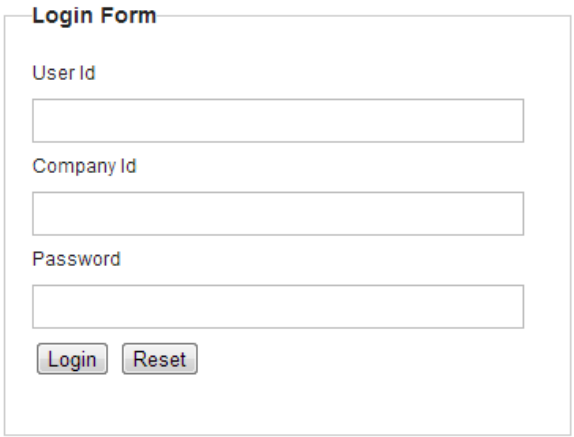
```
<!-- (1) -->
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/authenticate</url-pattern>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

項番	説明
(1)	Forward で認証するためのパターンを指定する ここでは認証パスである/authenticate を指定している。

認証処理の拡張

Spring Security から提供されている認証プロバイダで対応できない認証要件がある場合は、org.springframework.security.authentication.AuthenticationProvider インタフェースを実装したクラスを作成する必要がある。

ここでは、ユーザー名、パスワード、**会社識別子 (独自の認証パラメータ)** の 3 つのパラメータを使用して DB 認証を行うための拡張例を示す。



上記の要件を実現するためには、以下に示すクラスを作成する必要がある。

項番	説明
(1)	ユーザー名、パスワード、会社識別子を保持する `org.springframework.security.core.Authentication` インタフェースの実装クラス。 ここでは、`org.springframework.security.authentication.UsernamePasswordAuthenticationToken` クラスを継承して作成する。
(2)	ユーザー名、パスワード、会社識別子を使用して DB 認証を行う `org.springframework.security.authentication.AuthenticationProvider` の実装クラス。 ここでは、`org.springframework.security.authentication.dao.DaoAuthenticationProvider` クラスを継承して作成する。
(3)	ユーザー名、パスワード、会社識別子をリクエストパラメータから取得して、`AuthenticationManager(AuthenticationProvider)` に渡す `Authentication` を生成するための Authentication Filter クラス。 ここでは、`org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter` クラスを継承して作成する。

注釈: ここでは、認証用のパラメータとして独自のパラメータを追加する例にしているため、`Authentication` インタフェースの実装クラスと `Authentication` を生成するための Authentication Filter クラスの拡張が必要となる。

ユーザー名とパスワードのみで認証する場合は、`AuthenticationProvider` インタフェースの実装クラスを作成するだけで、認証処理を拡張することができる。

Authentication インターフェースの実装クラスの作成

`UsernamePasswordAuthenticationToken` クラスを継承し、ユーザー名とパスワードに加えて、会社

識別子 (独自の認証パラメータ) を保持するクラスを作成する。

```
// import omitted
public class CompanyIdUsernamePasswordAuthenticationToken extends
    UsernamePasswordAuthenticationToken {

    private static final long serialVersionUID = SpringSecurityCoreVersion.SERIAL_
↪VERSION_UID;

    // (1)
    private final String companyId;

    // (2)
    public CompanyIdUsernamePasswordAuthenticationToken(
            Object principal, Object credentials, String companyId) {
        super(principal, credentials);
        this.companyId = companyId;
    }

    // (3)
    public CompanyIdUsernamePasswordAuthenticationToken(
            Object principal, Object credentials, String companyId,
            Collection<? extends GrantedAuthority> authorities) {
        super(principal, credentials, authorities);
        this.companyId = companyId;
    }

    public String getCompanyId() {
        return companyId;
    }

}
```

項番	説明
(1)	会社識別子を保持するフィールドを作成する。
(2)	認証前の情報 (リクエストパラメータで指定された情報) を保持するインスタンスを作成する際に使用するコンストラクタを作成する。
(3)	認証済みの情報を保持するインスタンスを作成する際に使用するコンストラクタを作成する。 親クラスのコンストラクタの引数に認可情報を渡すことで、認証済みの状態となる。

AuthenticationProvider インターフェースの実装クラスの作成

DaoAuthenticationProvider クラスを継承し、ユーザー名、パスワード、会社識別子を使用して DB 認証を行うクラスを作成する。

```
// import omitted
public class CompanyIdUsernamePasswordAuthenticationProvider extends
    DaoAuthenticationProvider {

    // omitted

    @Override
    protected void additionalAuthenticationChecks(UserDetails userDetails,
            UsernamePasswordAuthenticationToken authentication)
            throws AuthenticationException {

        // (1)
        super.additionalAuthenticationChecks(userDetails, authentication);

        // (2)
        CompanyIdUsernamePasswordAuthenticationToken␣
↪companyIdUsernamePasswordAuthentication =
                (CompanyIdUsernamePasswordAuthenticationToken) authentication;
        String requestedCompanyId = companyIdUsernamePasswordAuthentication.
↪getCompanyId();
        String companyId = ((SampleUserDetails) userDetails).getAccount().
↪getCompanyId();
        if (!companyId.equals(requestedCompanyId)) {
            throw new BadCredentialsException(messages.getMessage(
                    "AbstractUserDetailsAuthenticationProvider.badCredentials",
                    "Bad credentials"));
        }
    }

    @Override
    protected Authentication createSuccessAuthentication(Object principal,
            Authentication authentication, UserDetails user) {
        String companyId = ((SampleUserDetails) user).getAccount()
                .getCompanyId();
        // (3)
        return new CompanyIdUsernamePasswordAuthenticationToken(user,
                authentication.getCredentials(), companyId,
                user.getAuthorities());
    }

    @Override
    public boolean supports(Class<?> authentication) {
        // (4)
        return CompanyIdUsernamePasswordAuthenticationToken.class
```

```
                .isAssignableFrom(authentication);
    }

}
```

項番	説明
(1)	親クラスのメソッドを呼び出し、Spring Security が提供しているチェック処理を実行する。 この処理にはパスワード認証処理も含まれる。
(2)	パスワード認証が成功した場合は、会社識別子 (独自の認証パラメータ) の妥当性をチェックする。 上記例では、リクエストされた会社識別子とテーブルに保持している会社識別子が一致するかをチェックしている。
(3)	パスワード認証及び独自の認証処理が成功した場合は、認証済み状態の `CompanyIdUsernamePasswordAuthenticationToken` を作成して返却する。
(4)	`CompanyIdUsernamePasswordAuthenticationToken` にキャスト可能な `Authentication` が指定された場合に、本クラスを使用して認証処理を行うようにする。

注釈: ユーザーの存在チェック、ユーザーの状態チェック (無効ユーザー、ロック中ユーザー、利用期限切れユーザーなどのチェック) は、`additionalAuthenticationChecks` メソッドが呼び出される前に親クラスの処理として行われる。

Authentication Filter の作成

`UsernamePasswordAuthenticationFilter` クラスを継承し、認証情報 (ユーザー名、パスワード、会社識別子) を `AuthenticationProvider` に引き渡すための Authentication Filter クラスを作成する。

`attemptAuthentication` メソッドの実装は、`UsernamePasswordAuthenticationFilter` クラスのメソッドをコピーしてカスタマイズしたものである。

```
// import omitted
public class CompanyIdUsernamePasswordAuthenticationFilter extends
    UsernamePasswordAuthenticationFilter {

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request,
            HttpServletResponse response) throws AuthenticationException {

        if (!request.getMethod().equals("POST")) {
            throw new AuthenticationServiceException("Authentication method not
↪supported: "
                    + request.getMethod());
        }

        // (1)
        // Obtain UserName, Password, CompanyId
        String username = super.obtainUsername(request);
        String password = super.obtainPassword(request);
        String companyId = obtainCompanyId(request);
        if (username == null) {
            username = "";
        } else {
            username = username.trim();
        }
        if (password == null) {
            password = "";
        }
        CompanyIdUsernamePasswordAuthenticationToken authRequest =
            new CompanyIdUsernamePasswordAuthenticationToken(username, password,
↪companyId);

        // Allow subclasses to set the "details" property
        setDetails(request, authRequest);

        return this.getAuthenticationManager().authenticate(authRequest); // (2)
    }

    // (3)
    protected String obtainCompanyId(HttpServletRequest request) {
        return request.getParameter("companyId");
    }
}
```

項番	説明
(1)	リクエストパラメータから取得した認証情報 (ユーザー名、パスワード、会社識別子) より、`CompanyIdUsernamePasswordAuthenticationToken` のインスタンスを生成する。
(2)	リクエストパラメータで指定された認証情報 (`CompanyIdUsernamePasswordAuthenticationToken` のインスタンス) を指定して、`org.springframework.security.authentication.AuthenticationManager` の `authenticate` メソッドを呼び出す。 `AuthenticationManager` のメソッドを呼び出すと、`AuthenticationProvider` の認証処理が呼び出される。
(3)	会社識別子は、`companyId` というリクエストパラメータより取得する。

ログインフォームの修正

ログインフォームの作成で作成したログインフォーム (JSP) に対して、会社識別子を追加する。

```
<form:form action="${pageContext.request.contextPath}/login" method="post">
    <!-- omitted -->
        <tr>
            <td><label for="username">User Name</label></td>
            <td><input type="text" id="username" name="username"></td>
        </tr>
        <tr>
            <td><label for="companyId">Company Id</label></td>
            <td><input type="text" id="companyId" name="companyId"></td> <!-- (1) -
↪->
        </tr>
        <tr>
            <td><label for="password">Password</label></td>
            <td><input type="password" id="password" name="password"></td>
        </tr>
    <!-- omitted -->
</form:form>
```

項番	説明
(1)	会社識別子の入力フィールド名に companyId を指定する。

拡張した認証処理の適用

ユーザー名、パスワード、会社識別子 (独自の認証パラメータ) を使用した DB 認証機能を Spring Security に適用する。

- spring-security.xml の定義例

```
<!-- omitted -->

<!-- (1) -->
<sec:http
    entry-point-ref="loginUrlAuthenticationEntryPoint">

    <!-- omitted -->

    <!-- (2) -->
    <sec:custom-filter
        position="FORM_LOGIN_FILTER" ref=
↪"companyIdUsernamePasswordAuthenticationFilter" />

    <!-- omitted -->

    <sec:csrf token-repository-ref="csrfTokenRepository" />

    <sec:logout
        logout-url="/logout"
        logout-success-url="/login" />

    <!-- omitted -->

    <sec:intercept-url pattern="/login" access="permitAll" />
    <sec:intercept-url pattern="/**" access="isAuthenticated()" />

    <!-- omitted -->

</sec:http>

<!-- (3) -->
<bean id="loginUrlAuthenticationEntryPoint"
    class="org.springframework.security.web.authentication.
↪LoginUrlAuthenticationEntryPoint">
```

```
    <constructor-arg value="/login" />
</bean>

<!-- (4) -->
<bean id="companyIdUsernamePasswordAuthenticationFilter"
    class="com.example.app.common.security.
↪CompanyIdUsernamePasswordAuthenticationFilter">
    <!-- (5) -->
    <property name="requiresAuthenticationRequestMatcher">
        <bean class="org.springframework.security.web.util.matcher.
↪AntPathRequestMatcher">
            <constructor-arg index="0" value="/authentication" />
            <constructor-arg index="1" value="POST" />
        </bean>
    </property>
    <!-- (6) -->
    <property name="authenticationManager" ref="authenticationManager" />
    <!-- (7) -->
    <property name="sessionAuthenticationStrategy" ref=
↪"sessionAuthenticationStrategy" />
    <!-- (8) -->
    <property name="authenticationFailureHandler">
        <bean class="org.springframework.security.web.authentication.
↪SimpleUrlAuthenticationFailureHandler">
            <constructor-arg value="/login?error=true" />
        </bean>
    </property>
    <!-- (9) -->
    <property name="authenticationSuccessHandler">
        <bean class="org.springframework.security.web.authentication.
↪SimpleUrlAuthenticationSuccessHandler" />
    </property>
</bean>

<!-- (6') -->
<sec:authentication-manager alias="authenticationManager">
    <sec:authentication-provider ref=
↪"companyIdUsernamePasswordAuthenticationProvider" />
</sec:authentication-manager>
<bean id="companyIdUsernamePasswordAuthenticationProvider"
    class="com.example.app.common.security.
↪CompanyIdUsernamePasswordAuthenticationProvider">
    <property name="userDetailsService" ref="sampleUserDetailsService" />
    <property name="passwordEncoder" ref="passwordEncoder" />
</bean>

<!-- (7') -->
<bean id="sessionAuthenticationStrategy"
    class="org.springframework.security.web.authentication.session.
↪CompositeSessionAuthenticationStrategy">
    <constructor-arg>
        <util:list>
            <bean class="org.springframework.security.web.csrf.
↪CsrfAuthenticationStrategy">
```

```
                <constructor-arg ref="csrfTokenRepository" />
            </bean>
            <bean class="org.springframework.security.web.authentication.session.
↪SessionFixationProtectionStrategy" />
        </util:list>
    </constructor-arg>
</bean>

<bean id="csrfTokenRepository"
    class="org.springframework.security.web.csrf.HttpSessionCsrfTokenRepository" />


<!-- omitted -->
```

項番	説明
(1)	(2) の<sec:custom-filter>タグを使用して FORM_LOGIN_FILTER を差し替える場合は、<sec:http>タグの属性に以下の設定を行う必要がある。 • 自動設定を使用することができないため、auto-config="false"を指定するか、auto-config 属性を削除する。 • <sec:form-login>タグが使用できないため、entry-point-ref 属性を使用して AuthenticationEntryPoint を明示的に指定する。
(2)	<sec:custom-filter>タグを使用して FORM_LOGIN_FILTER を差し替える。 <sec:custom-filter>タグの position 属性に FORM_LOGIN_FILTER を指定し、ref 属性に拡張した Authentication Filter の bean を指定する。
(3)	<sec:http>タグの entry-point-ref 属性に使用する AuthenticationEntryPoint の bean を指定する。 ここでは、<sec:form-login>タグを指定した際に使用される org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint クラスの bean を指定している。
(4)	FORM_LOGIN_FILTER として使用する Authentication Filter クラスの bean を定義する。 ここでは、拡張した Authentication Filter クラス (CompanyIdUsernamePasswordAuthenticationFilter) の bean を定義している。

Continued on next page

表 9.21 – continued from previous page

項番	説明
(5)	`requiresAuthenticationRequestMatcher` プロパティに、認証処理を行うリクエストを検出するための `RequestMatcher` インスタンスを指定する。 ここでは、`/authentication` というパスにリクエストがあった場合に認証処理を行うように設定している。 これは、`<sec:form-login>`タグの `login-processing-url` 属性に`/authentication` を指定したのと同義である。
(6)	`authenticationManager` プロパティに、`<sec:authentication-manager>`タグの `alias` 属性に設定した値を指定する。 `<sec:authentication-manager>`タグの `alias` 属性を指定すると、 Spring Security が生成した `AuthenticationManager` の bean を、他の bean へ DI することができる様になる。
(6』)	Spring Security が生成する `AuthenticationManager` に対して、拡張した `AuthenticationProvider`(`CompanyIdUsernamePasswordAuthenticationProvider`) を設定する。
(7)	`sessionAuthenticationStrategy` プロパティに、認証成功時のセッションの取り扱いを制御するコンポーネント (`SessionAuthenticationStrategy`) の bean を指定する。
(7』)	認証成功時のセッションの取り扱いを制御するコンポーネント (`SessionAuthenticationStrategy`) の bean を定義する。 ここでは、Spring Security から提供されている、 • CSRF トークンを作り直すコンポーネント (`CsrfAuthenticationStrategy`) • セッション・フィクセーション攻撃を防ぐために新しいセッションを生成するコンポーネント (`SessionFixationProtectionStrategy`) を有効化している。

Continued on next page

表 9.21 – continued from previous page

項番	説明
(8)	`authenticationFailureHandler` プロパティに、認証失敗時に呼ばれるハンドラクラスを指定する。
(9)	`authenticationSuccessHandler` プロパティに、認証成功時に呼ばれるハンドラクラスを指定する。

注釈: auto-config について

auto-config="false"を指定又は指定を省略した際に Basic 認証処理とログアウト処理を有効化したい場合は、<sec:http-basic>タグと<sec:logout>タグを明示的に定義する必要がある。

非推奨アルゴリズムの PasswordEncoder の利用

セキュリティ要件によっては、前述した PasswordEncoder を実装したクラスでは実現できない場合がある。特に、既存のアカウント情報で使用しているハッシュ化要件を踏襲する必要がある場合は、前述の PasswordEncoder では要件を満たせないことがある。

具体的には、既存のハッシュ化要件が以下のようなケースである。

- アルゴリズムが SHA-512 である。
- ハッシュ化回数が 1000 回である。

このようなケースでは、org.springframework.security.crypto.password.PasswordEncoder の実装クラスの一つである MessageDigestPasswordEncoder を利用することで要件を満たすことができる。

警告: 非推奨アルゴリズムの PasswordEncoder として org.springframework.security.authentication.encoding.PasswordEncoder の実装クラスが存在するが、これらのクラスは固定のソルトを使用する必要があり、セキュリティ上のリスクを伴う。

org.springframework.security.authentication.encoding.PasswordEncoder の実装クラスを利用している場合は、ランダムなソルトを利用する MessageDigestPasswordEncoder への移行を検討されたい。

MessageDigestPasswordEncoder の利用

MessageDigestPasswordEncoder は Java が提供する java.security.MessageDigest クラスを利用してハッシュ化を行う。MessageDigest クラスは MD-5、SHA-1、SHA-256 等のハッシュアルゴリズムを提供している。詳細については Javadoc を参照されたい。

警告: MessageDigestPasswordEncoder は旧式の実装であることを示すため非推奨となっている。このクラスが廃止される予定はないが、セキュリティ上のリスクが想定されるため、Pbkdf2 アルゴリズムや BCrypt アルゴリズムを利用することを検討されたい。

ここでは MessageDigestPasswordEncoder を利用して、以下のハッシュ化要件を満たす PasswordEncoder を実装する。

- アルゴリズムが SHA-512 である。
- ハッシュ化回数が 1000 回である。

注釈: MessageDigestPasswordEncoder の仕様

MessageDigestPasswordEncoder でハッシュ化を行うと以下のフォーマットで出力される。

```
{salt}hashValue
```

ソルトは Spring Security 4.2.6 からランダムに生成するようになったため、安全性が向上している。

ハッシュ化したパスワードに付与されたソルトは照合の際に使用される。

既に固定のソルトを用いてハッシュ化したパスワードについて

先述の通り、Spring Security 4.2.6 以降の MessageDigestPasswordEncoder はソルトをランダムに生成するが、既に固定のソルトを用いてパスワードをハッシュ化していた場合も、パスワードにソルトを付与する移行処理を行うことで、照合することができるようになる。

パスワードデータの移行については、MessageDigestPasswordEncoder の Javadoc を参照されたい。

この場合、既存のパスワードは固定のソルトを用いて照合が行われるが、パスワードを新規に設定または変更した場合はランダムなソルトが用いられる。

警告: MessageDigestPasswordEncoder を使用する際は、以下の点に注意する必要がある。

- 既存のハッシュ値より文字数が多くなる
- エンコードしたパスワードからソルトの情報が得られる

MessageDigestPasswordEncoder の bean を定義する。

- applicationContext.xml の定義例

```
<bean id="passwordEncoder" class="org.springframework.security.crypto.password.
↪MessageDigestPasswordEncoder"> <!-- (1) -->
    <constructor-arg name="algorithm" value="SHA-512" /> <!-- (2) -->
    <property name="iterations" value="1000" /> <!-- (3) -->
</bean>
```

項番	説明
(1)	ハッシュ化に使用する PasswordEncoder として MessageDigestPasswordEncoder を bean 定義する。
(2)	MessageDigestPasswordEncoder で使用するハッシュアルゴリズムを指定する。 ここには MessageDigest クラスが対応するアルゴリズムを指定することができる。 指定できる値については、Java 暗号化アーキテクチャ標準アルゴリズム名のドキュメント を参照されたい。
(3)	ハッシュ化の iterations(反復) 回数を指定する。 設定を行わなかった場合、1 回となる。

9.2.4 Appendix

Spring MVC でリクエストを受けてログインフォームを表示する

Spring MVC でリクエストを受けてログインフォームを表示する方法を説明する。

- ログインフォームを表示する Controller の定義例

```
@Controller
@RequestMapping("/login")
public class LoginController { // (1)
```

```
    @RequestMapping
    public String index() {
        return "login";
    }
}
```

項番	説明
(1)	view 名として」login」を返却する。`InternalResourceViewResolver` によって src/main/webapp/WEB-INF/views/login.jsp が出力される。

本例のように、単純に view 名を返すだけのメソッドが一つだけある Controller であれば、<mvc:view-controller>を使用して代用することも可能である。

詳しくは、*HTML を応答する*を参照されたい。

Remember Me 認証の利用

「Remember Me 認証」とは、Web サイトに頻繁にアクセスするユーザーの利便性を高めるための機能の一つで、ログイン状態を通常のライフサイクルより長く保持するための機能である。本機能を使用すると、ブラウザを閉じた後やセッションタイムが発生した後でも、Cookie に保持している Remember Me 認証用の Token を使用して、ユーザ名とパスワードを再入力することなく自動でログインすることができる。なお、本機能は、ユーザーがログイン状態を保持することを許可した場合のみ有効となる。

Spring Security は、「Hash-Based Token 方式の Remember Me 認証」と「Persistent Token 方式の Remember Me 認証」をサポートしており、デフォルトでは Hash-Based Token 方式が使用される。

Remember Me 認証を利用する場合は、<sec:remember-me>タグを追加する。

- spring-security.xml の定義例

```
<sec:http>
    <!-- omitted -->
    <sec:remember-me key="terasoluna-tourreservation-km/ylnHv"
        token-validity-seconds="#{30 * 24 * 60 * 60}" />  <!-- (1) (2) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	key 属性に、Remember Me 認証用の Token を生成したアプリケーションを識別するキー値を指定する。 キー値の指定が無い場合、アプリケーションの起動毎にユニークな値が生成される。 なお、Hash-Based Token が保持しているキー値とサーバーで保持しているキー値が異なる場合、無効な Token として扱われる。 つまり、アプリケーションを再起動する前に生成した Hash-Based Token を有効な Token として扱いたい場合は、key 属性の指定は必須である。
(2)	token-validity-seconds 属性に、Remember Me 認証用の Token の有効時間を秒単位で指定する。 指定が無い場合、デフォルトで 14 日間が有効時間になる。 上記例では、有効時間として 30 日間を設定している。

上記以外の属性については、Spring Security Reference -The Security Namespace (<remember-me>) -を参照されたい。

注釈: Spring Security 4.0 における変更

Spring Security 4.0 から、以下の設定のデフォルト値が変更されている

- remember-me-parameter
- remember-me-cookie

ログインフォームには、「Remember Me 認証」機能の利用有無を指定するためのフラグ (チェックボックス項目) を用意する。

- ログインフォームの JSP の実装例

```
<form:form action="${pageContext.request.contextPath}/login" method="post">
        <!-- omitted -->
        <tr>
            <td><label for="remember-me">Remember Me : </label></td>
            <td><input name="remember-me" id="remember-me" type="checkbox" checked=
↪"checked" value="true"></td> <!-- (1) -->
```

```
        </tr>
        <!-- omitted -->
</form:form>
```

項番	説明
(1)	「Remember Me 認証」機能の利用有無を指定するためのフラグ (チェックボックス項目) を追加し、フィールド名 (リクエストパラメータ名) には、`remember-me-parameter` のデフォルト値である `remember-me` を指定する。 チェックボックスの `value` 属性には、`true` を設定する。 チェックボックスをチェック状態にしてから認証処理を実行すると、以降のリクエストから「Remember Me 認証」機能が適用される。

ちなみに：value 属性の設定値について

`value` 属性には、`true` を設定する旨が rememberMeRequested の JavaDoc に記載されているが、実装上は `on`、`yes`、」`1`」 も設定可能である。

9.3 認可

9.3.1 Overview

本節では、Spring Security が提供している認可機能について説明する。

認可処理は、アプリケーションの利用者がアクセスできるリソースを制御するための処理である。利用者がアクセスできるリソースを制御するためのもっとも標準的な方法は、リソース (又はリソースの集合) 毎にアクセスポリシーを定義しておき、利用者がリソースにアクセスしようとした時にアクセスポリシーを調べて制御する方法である。

アクセスポリシーには、どのリソースにどのユーザーからのアクセスを許可するかを定義する。Spring Security では、以下の 3 つのリソースに対してアクセスポリシーを定義することができる。

- Web リソース
- Java メソッド
- ドメインオブジェクト[*1]
- JSP の画面項目

本節では、「Web リソース」「Java メソッド」「JSP の画面項目」のアクセスに対して認可処理を適用するための実装例 (定義例) を紹介しながら、Spring Security の認可機能について説明する。

認可処理のアーキテクチャ

Spring Security は、以下のような流れで認可処理を行う。

[*1] ドメインオブジェクトのアクセスに対する認可処理については、 Spring Security Reference -Domain Object Security (ACLs)-を参照されたい。

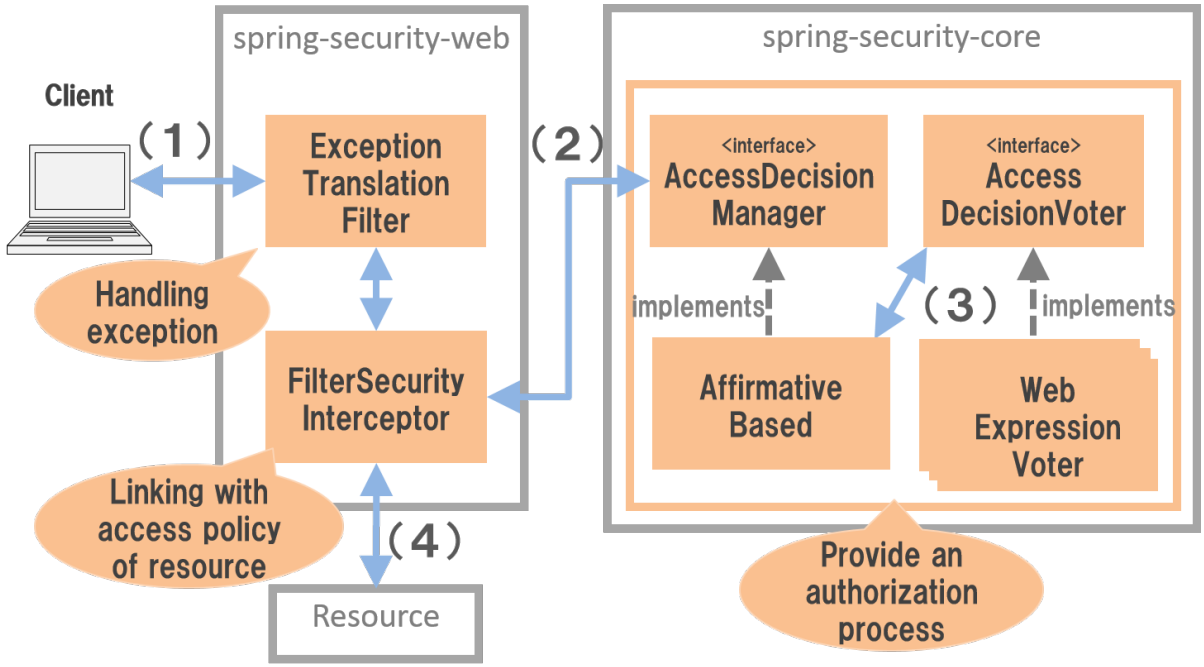


図 9.7　認可処理のアーキテクチャ

項番	説明
(1)	クライアントは、任意のリソースにアクセスする。
(2)	`FilterSecurityInterceptor` クラスは、`AccessDecisionManager` インタフェースのメソッドを呼び出し、リソースへのアクセス権の有無をチェックする。
(3)	`AffirmativeBased` クラス (デフォルトで使用される `AccessDecisionManager` の実装クラス) は、`AccessDecisionVoter` インタフェースのメソッドを呼び出し、アクセス権の有無を投票させる。
(4)	`FilterSecurityInterceptor` は、`AccessDecisionManager` によってアクセス権が付与された場合に限り、リソースへアクセスする。

ExceptionTranslationFilter

ExceptionTranslationFilter は、認可処理 (AccessDecisionManager) で発生した例外をハンドリングし、クライアントへ適切なレスポンスを行うための Security Filter である。デフォルトの実装では、未認証ユーザーからのアクセスの場合は認証を促すレスポンス、認証済みのユーザーからのアクセスの場合は認可エラーを通知するレスポンスを返却する。

FilterSecurityInterceptor

FilterSecurityInterceptor は、HTTP リクエストに対して認可処理を適用するための Security Filter で、実際の認可処理は AccessDecisionManager に委譲する。AccessDecisionManager インタフェースのメソッドを呼び出す際には、クライアントがアクセスしようとしたリソースに指定されているアクセスポリシーを連携する。

AccessDecisionManager

AccessDecisionManager は、アクセスしようとしたリソースに対してアクセス権があるかチェックを行うためのインタフェースである。

Spring Security が提供する実装クラスは 3 種類存在するが、いずれも AccessDecisionVoter というインタフェースのメソッドを呼び出してアクセス権を付与するか否かを判定させている。AccessDecisionVoter は「付与」「拒否」「棄権」のいずれかを投票し、AccessDecisionManager の実装クラスが投票結果を集約して最終的なアクセス権を判断する。アクセス権がないと判断した場合は、AccessDeniedException を発生させアクセスを拒否する。

なお、すべての投票結果が「棄権」であった場合、Spring Security のでデフォルトでは、「アクセス権なし」と判定される。

表 9.22 Spring Security が提供する AccessDecisionManager の実装クラス

クラス名	説明
AffirmativeBased	AccessDecisionVoter に投票させ、「付与」が 1 件投票された時点でアクセス権を与える実装クラス。 **デフォルトで使用される実装クラス。**
ConsensusBased	全ての AccessDecisionVoter に投票させ、「付与」の投票数が多い場合にアクセス権を与える実装クラス。 「付与」「拒否」が 1 件以上、且つ同数の場合、Spring Security のデフォルトでは、「アクセス権あり」と判定される。
UnanimousBased	AccessDecisionVoter に投票させ、「拒否」が 1 件投票された時点で **アクセス権を与えない** 実装クラス。

注釈: **AccessDecisionVoter の選択**

使用する AccessDecisionVoter が 1 つの場合はどの実装クラスを使っても動作に違いはない。複数の AccessDecisionVoter を使用する場合は、要件に合わせて実装クラスを選択されたい。

AccessDecisionVoter

AccessDecisionVoter は、アクセスしようとしたリソースに指定されているアクセスポリシーを参照してアクセス権を付与するかを投票するためのインタフェースである。

Spring Security が提供する主な実装クラスは以下の通り。

表 9.23　Spring Security が提供する AccessDecisionVoter の主な実装クラス

クラス名	説明
WebExpressionVoter	SpEL 経由で認証情報 (Authentication) が保持する権限情報とリクエスト情報 (HttpServletRequest) を参照して投票を行う実装クラス。
RoleVoter	利用者が持つロールを参照して投票を行う実装クラス。
RoleHierarchyVoter	利用者が持つ階層化されたロールを参照して投票を行う実装クラス。
AuthenticatedVoter	認証状態を参照して投票を行う実装クラス。

注釈: デフォルトで適用される AccessDecisionVoter

デフォルトで適用される AccessDecisionVoter インタフェースの実装クラスは、Spring Security 4.0 から WebExpressionVoter に統一されている。WebExpressionVoter は、RoleVoter、RoleHierarchyVoter、AuthenticatedVoter を使用した時と同じことが実現できるため、本ガイドラインでも、デフォルトの WebExpressionVoter を使って認可処理を行う前提で説明を行う。

9.3.2 How to use

認可機能を使用するために必要となる bean 定義例 (アクセスポリシーの指定方法) や実装方法について説明する。

アクセスポリシーの記述方法

アクセスポリシーの記述方法を説明する。

Spring Security は、アクセスポリシーを指定する記述方法として Spring Expression Language(SpEL) をサポー

トしている。SpEL を使わない方法もあるが、本ガイドラインでは Expression を使ってアクセスポリシーを指定する方法で説明を行う。SpEL の使い方については本節でも紹介するが、より詳しい使い方を知りたい場合は Spring Framework Reference Documentation -Spring Expression Language (SpEL)-を参照されたい。

Built-In の Common Expressions

Spring Security が用意している共通的な Expression は以下の通り。

表 9.24: **Spring Security が提供している共通的な Expression**

Expression	説明
`hasRole(String role)`	ログインユーザーが、引数に指定したロールを保持している場合に `true` を返却する。
`hasAnyRole(String... roles)`	ログインユーザーが、引数に指定したロールのいずれかを保持している場合に `true` を返却する。
`isAnonymous()`	ログインしていない匿名ユーザーの場合に `true` を返却する。
`isRememberMe()`	Remember Me 認証によってログインしたユーザーの場合に `true` を返却する。
`isAuthenticated()`	ログイン中の場合に `true` を返却する。
`isFullyAuthenticated()`	Remember Me 認証ではなく通常の認証プロセスによってログインしたユーザーの場合に `true` を返却する。
`permitAll`	常に `true` を返却する。
`denyAll`	常に `false` を返却する。

Continued on next page

表 9.24 – continued from previous page

Expression	説明
`principal`	認証されたユーザーのユーザー情報 (`UserDetails` インタフェースを実装したクラスのオブジェクト) を返却する。
`authentication`	認証されたユーザーの認証情報 (`Authentication` インタフェースを実装したクラスのオブジェクト) を返却する。

注釈: **Expression** を使用した認証情報へのアクセス

Expression として `principal` や `authentication` を使用すると、ログインユーザーのユーザー情報や認証情報を参照することができるため、ロール以外の属性を使ってアクセスポリシーを設定することが可能になる。

注釈: ロール名のプレフィックス

Spring Security 3.2 までは、ロール名には `ROLE_` プレフィックスを指定する必要があったが、Spring Security 4.0 から `ROLE_` プレフィックスの指定が不要となっている。

例）

- Spring Secuirty 3.2 以前 : `hasRole('ROLE_USER')`
- Spring Security 4.0 以降 : `hasRole('USER')`

Built-In の Web Expressions

Spring Security が用意している Web アプリケーション向け Expression は以下の通り。

表 9.25 **Spring Security が提供する Web アプリケーション向け Expression**

Expression	説明
`hasIpAddress(String ipAddress)`	リクエスト元の IP アドレスが、引数に指定した IP アドレス体系に一致する場合に `true` を返却する。

演算子の使用

演算子を使用した判定も行うことができる。以下の例では、ロールと、リクエストされた IP アドレス両方に合致した場合、アクセス可能となる。

- spring-security.xml の定義例

```
<sec:http>
    <sec:intercept-url pattern="/admin/**" access="hasRole('ADMIN') and␣
↪hasIpAddress('192.168.10.1')"/>
    <!-- omitted -->
</sec:http>
```

使用可能な演算子一覧

演算子	説明
[式 1] and [式 2]	式 1、式 2 が、どちらも真の場合に、真を返す。
[式 1] or [式 2]	いずれかの式が、真の場合に、真を返す。
![式]	式が真の場合は偽を、偽の場合は真を返す。

Web リソースへの認可

Spring Security は、サーブレットフィルタの仕組みを利用して Web リソース (HTTP リクエスト) に対して認可処理を行う。

認可処理の適用

Web リソースに対して認可処理を適用する場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<sec:http>
    <!-- omitted -->
    <sec:intercept-url pattern="/**" access="isAuthenticated()" />  <!-- (1) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	<sec:intercept-url>タグに、HTTP リクエストに対してアクセスポリシーを定義する。 ここでは、SpEL を使用して「Web アプリケーション配下の全てのリクエストに対して認証済みのユーザーのみアクセスを許可する」というアクセスポリシーを定義している。

注釈: **use-expressions のデフォルト定義**

Spring Security 4.0 から、<sec:http> タグの use-expressions 属性のデフォルト値が true に変更になっているため、true を使用する場合に明示的な記述は不要となった。

アクセスポリシーの定義

bean 定義ファイルを使用して、Web リソースに対してアクセスポリシーを定義する方法について説明する。

アクセスポリシーを適用する Web リソースの指定

まず、アクセスポリシーを適用するリソース (HTTP リクエスト) を指定する。アクセスポリシーを適用するリソースの指定は、<sec:intercept-url>タグの以下の属性を使用する。

表 9.26　アクセスポリシーを適用するリソースを指定するための属性

属性名	説明
pattern	Ant 形式又は正規表現で指定したパスパターンに一致するリソースを適用対象にするための属性。
method	指定した HTTP メソッド (GET,POST など) を使ってアクセスがあった場合に適用対象にするための属性。
requires-channel	「http」、もしくは「https」を指定する。指定したプロトコルでのアクセスを強制するための属性。 指定しない場合、どちらでもアクセス可能である。

上記以外の属性については、<intercept-url>を参照されたい。

- <sec:intercept-url>タグ pattern 属性の定義例（spring-security.xml）

```
<sec:http >
    <sec:intercept-url pattern="/admin/accounts/**" access="..."/>
    <sec:intercept-url pattern="/admin/**" access="..."/>
    <sec:intercept-url pattern="/**" access="..."/>
    <!-- omitted -->
</sec:http>
```

Spring Security は定義した順番でリクエストとのマッチング処理を行い、最初にマッチした定義を適用する。そのため、bean 定義ファイルを使用してアクセスポリシーを指定する場合も定義順番には注意が必要である。

ちなみに：パスパターンの解釈

Spring Security のデフォルトの動作では、パスパターンは Ant 形式で解釈する。パスパターンを正規表現で指

定したい場合は、<sec:http>タグの request-matcher 属性に regex を指定すること。

```
<sec:http request-matcher="regex">
    <sec:intercept-url pattern="/admin/accounts/.*" access="hasRole('ACCOUNT_
↪MANAGER')" />
    <!-- omitted -->
</sec:http>
```

警告: Spring Security 4.1 以降、Spring Security がデフォルトで使用している *AntPathRequestMatcher* のパスマッチングの仕様が大文字・小文字を区別する様になった。

例えば以下に示すように、/Todo/List というパスが割り当てられた Spring MVC のエンドポイントに対してアクセスポリシーを定義する場合は、<sec:intercept-url>タグの pattern 属性に指定する値は /Todo/List や /Todo/*など大文字・小文字をそろえる必要がある。誤って/todo/list や/todo/**など大文字・小文字がそろっていない値を指定してしまうと、意図した認可制御が行われなくなるので注意されたい。

- Spring MVC のエンドポイントの実装例

```
@RequestMapping(value="/Todo/List")
public String viewTodoList(){
    //...
}
```

- アクセスポリシーの定義例

```
<sec:http>
    <sec:intercept-url pattern="/Todo/List" access="isAuthenticated()" />
    <!-- omitted -->
</sec:http>
```

注釈: Spring MVC と Spring Security では、リクエストとのマッチングの仕組みが厳密には異なっており、この差異を利用して Spring Security の認可機能を突破し、ハンドラメソッドにアクセスできる脆弱性が存在する。本事象の詳細は「CVE-2016-5007 Spring Security / MVC Path Matching Inconsistency」を参照されたい。

trimTokens プロパティに *true* を設定した *org.springframework.util.AntPathMatcher* の Bean が Spring MVC に適用されている場合に、本事象が発生する。Spring Framework 4.2 以前は *trimTokens* プロパティのデフォルト値が *true* となっていたが、Spring Framework 4.3 からデフォルト値は *false* となったため、意図的に変更しない限り本事象は発生しない。

警告: 特定の URL に対してアクセスポリシーを設ける (pattern 属性に」*」や**などのワイルドカード指定を含めない) 場合、拡張子を付けたパターンとリクエストパスの末尾に」/」を付けたパターンに対するアクセスポリシーの追加が必須である。

下記の設定例は、/restrict に対して「ROLE_ADMIN」ロールを持つユーザからのアクセスのみを許可している。

```
<sec:http>
    <sec:intercept-url pattern="/restrict.*" access="hasRole('ADMIN')" />
↪<!-- (1) -->
    <sec:intercept-url pattern="/restrict/" access="hasRole('ADMIN')" /> <!
↪-- (2) -->
    <sec:intercept-url pattern="/restrict" access="hasRole('ADMIN')" /> <!-
↪- (3) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	/restrict に拡張子を付けたパターン (/restrict.json など) のアクセスポリシーを定義する。
(2)	/restrict にリクエストパスの末尾に」/」を付けたパターン (/restrict/など) のアクセスポリシーを定義する。
(3)	/restrict に対するアクセスポリシーを定義する。

アクセスポリシーの指定

つぎに、アクセスポリシーを指定する。アクセスポリシーの指定は、<sec:intercept-url>タグの access 属性に指定する。

- <sec:intercept-url>タグ access 属性の定義例（spring-security.xml）

```
<sec:http>
    <sec:intercept-url pattern="/admin/accounts/**" access="hasRole('ACCOUNT_
↪MANAGER')"/>
    <sec:intercept-url pattern="/admin/configurations/**" access=
↪"hasIpAddress('127.0.0.1') and hasRole('CONFIGURATION_MANAGER')" />
    <sec:intercept-url pattern="/admin/**" access="hasRole('ADMIN')" />
    <!-- omitted -->
</sec:http>
```

表 9.28　アクセスポリシーを指定するための属性

属性名	説明
access	SpEL でのアクセス制御式や、アクセス可能なロールを指定する。

ログインユーザーに「ROLE_USER」「ROLE_ADMIN」というロールがある場合を例に、設定例を示す。

- <sec:intercept-url>タグ pattern 属性の定義例（spring-security.xml）

```
<sec:http>
    <sec:intercept-url pattern="/reserve/**" access="hasAnyRole('USER','ADMIN
↪')" /> <!-- (1) -->
    <sec:intercept-url pattern="/admin/**" access="hasRole('ADMIN')" /> <!--␣
↪(2) -->
    <sec:intercept-url pattern="/**" access="denyAll" /> <!-- (3) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	「/reserve/**」にアクセスするためには、「ROLE_USER」もしくは「ROLE_ADMIN」ロールが必要である。 hasAnyRole については、後述する。
(2)	「/admin/**」にアクセスするためには、「ROLE_ADMIN」ロールが必要である。 hasRole については、後述する。
(3)	denyAll を全てのパターンに設定し、 権限設定が記述されていない URL に対してはどのユーザーもアクセス出来ない設定としている。 denyAll については、後述する。

注釈: URL パターンの記述順序について

クライアントからのリクエストに対して、intercept-url で記述されているパターンに、上から順にマッチさせ、マッチしたパターンに対してアクセス認可を行う。そのため、パターンの記述は、必ず、より限定されたパターンから記述すること。

Spring Securiy ではデフォルトで、SpEL が有効になっている。access 属性に記述した SpEL は真偽値で評価され、式が真の場合に、アクセスが認可される。以下に使用例を示す。

- spring-security.xml の定義例

```
<sec:http>
    <sec:intercept-url pattern="/admin/**" access="hasRole('ADMIN')"/>  <!--
↪(1) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	hasRole('ロール名') を指定することで、ログインユーザーが指定したロールを保持していれば真を返す。

使用可能な主な Expression は、アクセスポリシーの記述方法 を参照されたい。

パス変数の参照

Spring Security 4.1 以降では、アクセスポリシーを適用するリソースを指定する際にパス変数[*2]を使用することができ、アクセスポリシーの定義内で#**パス変数名**と指定することで参照できる。

ただし、拡張子を付けてアクセス可能なパスに対してパス変数を使用するアクセスポリシーを定義する場合は、パス変数値に拡張子部分が格納されない様に定義する必要がある。

例えば、パターンに/users/{userName}と定義し、/users/personName.json というリクエストパスを送信した際、アクセスポリシーの定義内で参照しているパス変数#userName には personName ではなく personName.json が格納され、意図しない認可制御が行われてしまう。

この事象を防ぐためには、「拡張子を付けたパスに対するアクセスポリシー」を定義した後に、「拡張子を付けないパスに対するアクセスポリシー」を定義する必要がある。

以下の例は、ログインユーザが自身のユーザ情報のみアクセスできる様にアクセスポリシーを定義している。

- spring-security.xml の定義例（ワイルドカードを使用する場合）

```
<sec:http>
    <!-- (1) -->
    <sec:intercept-url pattern="/users/{userName}.*"  access=
↪"isAuthenticated() and #userName == principal.username"/>
    <!-- (2) -->
    <sec:intercept-url pattern="/users/{userName}/**" access=
↪"isAuthenticated() and #userName == principal.username"/>
    <!-- omitted -->
</sec:http>
```

[*2] パス変数の説明は アプリケーション層の実装 の *URL のパスから値を取得する*を参照されたい。

項番	説明
(1)	「拡張子を付けたパスに対するアクセスポリシー」を定義する。
(2)	「拡張子を付けないパスに対するアクセスポリシー」を定義する。 ワイルドカードを使用して`/users/{userName}`で始まるパスに対するアクセスポリシーを定義する。

- spring-security.xml の定義例（ワイルドカードを使用しない場合）

```
<sec:http>
    <!-- (1) -->
    <sec:intercept-url pattern="/users/{userName}.*" access=
↪"isAuthenticated() and #userName == principal.username"/>
    <!-- (2) -->
    <sec:intercept-url pattern="/users/{userName}/"  access=
↪"isAuthenticated() and #userName == principal.username"/>
    <sec:intercept-url pattern="/users/{userName}"   access=
↪"isAuthenticated() and #userName == principal.username"/>
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	「拡張子を付けたパスに対するアクセスポリシー」を定義する。
(2)	「拡張子を付けないパスに対するアクセスポリシー」を定義する。 ワイルドカードを使用しない場合、Spring MVC と Spring Security のパスマッチングの差を吸収するために 末尾が」/」 で終わるパスに対するアクセスポリシーも定義する。

メソッドへの認可

Spring Security は、Spring AOP の仕組みを利用して DI コンテナで管理している Bean のメソッド呼び出しに対して認可処理を行う。

メソッドに対する認可処理は、ドメイン層 (サービス層) のメソッド呼び出しに対して行うことを想定して提供されている。メソッドに対する認可処理を使用すると、ドメインオブジェクトのプロパティを参照することができるため、きめの細かいアクセスポリシーの定義を行うことが可能になる。

AOP の有効化

メソッドへの認可処理を使用する場合は、メソッド呼び出しに対して認可処理を行うためのコンポーネント (AOP) を有効化する必要がある。AOP を有効化すると、アクセスポリシーをメソッドのアノテーションに定義できるようになる。

Spring Security は、以下のアノテーションをサポートしている。

- `@PreAuthorize`、`@PostAuthorize`、`@PreFilter`、`@PostFilter`
- JSR-250 (`javax.annotation.security` パッケージ) のアノテーション (`@RolesAllowed` など)
- `@Secured`

本ガイドラインでは、アクセスポリシーを Expression で使用することができる`@PreAuthorize`、`@PostAuthorize` を使用する方法を説明する。

- spring-security.xml の定義例

```
<sec:global-method-security pre-post-annotations="enabled" /> <!-- (1) (2) -->
```

項番	説明
(1)	`<sec:global-method-security>`タグを付与すると、メソッド呼び出しに対する認可処理を行う AOP が有効になる。
(2)	`pre-post-annotations` 属性に `enabled` を指定する。 `pre-post-annotations` 属性に `enabled` を指定すると、Expression を指定してアクセスポリシーを定義できるアノテーションが有効になる。

認可処理の適用

メソッドに対して認可処理を適用する際は、アクセスポリシーを指定するアノテーションを使用して、メソッド毎にアクセスポリシーを定義する。

アクセスポリシーの定義

メソッド実行前に適用するアクセスポリシーの指定

メソッドの実行前に適用するアクセスポリシーを指定する場合は、`@PreAuthorize` を使用する。

`@PreAuthorize` の `value` 属性に指定した Expression の結果が `true` になるとメソッドの実行が許可される。下記例では、管理者以外は、他人のアカウント情報にアクセスできないように定義している。

- `@PreAuthorize` の定義例

```
// (1) (2)
@PreAuthorize("hasRole('ADMIN') or (#username == principal.username)")
public Account findOne(String username) {
    return accountRepository.findOne(username);
}
```

項番	説明
(1)	認可処理を適用したいメソッドに、`@PreAuthorize` を付与する。
(2)	`value` 属性に、メソッドに対してアクセスポリシーを定義する。 ここでは、「管理者の場合は全てのアカウントへのアクセスを許可する」「管理者以外の場合は自身のアカウントへのアクセスのみ許可する」というアクセスポリシーを定義している。

ここでポイントになるのは、Expression の中からメソッドの引数にアクセスしている部分である。具体的には、「`#username`」の部分が引数にアクセスしている部分である。Expression 内で「# + 引数名」形式の Expression を指定することで、メソッドの引数にアクセスすることができる。なお、引数のプロパティやメソッドにアクセスする場合などは、実行時エラーを防止するため引数の `null` チェックを行うよう注意されたい。

ちなみに：引数名を指定するアノテーション

Spring Security は、クラスに出力されているデバッグ情報から引数名を解決する仕組みになっているが、アノテーション (`@org.springframework.security.access.method.P`) を使用して明示的に引数名を指定することもできる。

以下のケースにあてはまる場合は、アノテーションを使用して明示的に変数名を指定する。

- クラスに変数のデバッグ情報を出力しない
- Expression の中から実際の変数名とは別の名前を使ってアクセスしたい (例えば短縮した名前)

```
@PreAuthorize("hasRole('ADMIN') or (#username == principal.username)")
public Account findOne(@P("username") String username) {
    return accountRepository.findOne(username);
}
```

なお、#username と、メソッドの引数である username の名称が一致している場合は @P を省略することが可能である。ただし、Spring Security は引数名の解決を、実装クラスの引数名を使用して行っているため @PreAuthorize アノテーションをインターフェースに定義している場合には、**実装クラスの引数名を、@PreAuthorize 内で指定した #username と一致させる必要がある** ので、注意されたい。

JDK 8 から追加されたコンパイルオプション (-parameters) を使用すると、メソッドパラメータにリフレクション用のメタデータが生成されるため、アノテーションを指定しなくても引数名が解決される。

メソッド実行後に適用するアクセスポリシーの指定

メソッドの実行後に適用するアクセスポリシーを指定する場合は、@PostAuthorize を使用する。

@PostAuthorize の value 属性に指定した Expression の結果が true になるとメソッドの実行結果が呼び出し元に返却される。下記例では、所属する部署が違うユーザーのアカウント情報にアクセスできないように定義している。

- @PostAuthorize の定義例

```
@PreAuthorize("...")
@PostAuthorize("(returnObject == null) " +
        "or (returnObject.departmentCode == principal.account.departmentCode)")
public Account findOne(String username) {
    return accountRepository.findOne(username);
}
```

ここでポイントになるのは、Expression の中からメソッドの返り値にアクセスしている部分である。具体的には、「returnObject.departmentCode」の部分が返り値にアクセスしている部分である。Expression 内で「returnObject」を指定すると、メソッドの返り値にアクセスすることができる。なお、返り値のプロパティやメソッドにアクセスする場合などは、実行時エラーを防止するため返り値の null チェックを行うよう注意されたい。

JSP の画面項目への認可

Spring Security は、JSP タグライブラリを使用して JSP の画面項目に対して認可処理を適用することができる。

ここでは最もシンプルな定義を例に、JSP の画面項目のアクセスに対して認可処理を適用する方法について説明する。

アクセスポリシーの定義

JSP タグライブラリを使用して JSP の画面項目に対してアクセスポリシーを定義する際は、表示を許可する条件 (アクセスポリシー) を JSP に定義する。

- アクセスポリシー定義例

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<!-- (1) -->
<sec:authorize access="hasRole('ADMIN')"> <!-- (2) -->
    <h2>Admin Menu</h2>
    <!-- omitted -->
</sec:authorize>
```

項番	説明
(1)	アクセスポリシーを適用したい部分を<sec:authorize>タグで囲む。
(2)	access 属性にアクセスポリシーを定義する。ここでは、「管理者の場合は表示を許可する」というアクセスポリシーを定義している。

Web リソースに指定したアクセスポリシーとの連動

ボタンやリンクなど (サーバーへのリクエストを伴う画面項目) に対してアクセスポリシーを定義する際は、リクエスト先の Web リソースに定義されているアクセスポリシーと連動させる。Web リソースに指定したアクセスポリシーと連動させる場合は、<sec:authorize>タグの url 属性を使用する。

url 属性に指定した Web リソースにアクセスできる場合に限り<sec:authorize>タグの中に実装した JSP の処理が実行される。

- Web リソースに定義されているアクセスポリシーとの連携例

```
<ul>
    <!-- (1) -->
    <sec:authorize url="/admin/accounts"> <!-- (2) -->
        <li>
```

```
            <a href="<c:url value='/admin/accounts' />">Account Management</a>
        </li>
    </sec:authorize>
</ul>
```

項番	説明
(1)	ボタンやリンクを出力する部分を<sec:authorize>タグで囲む。
(2)	<sec:authorize>タグの url 属性に Web リソースへアクセスするための URL を指定する。 ここでは、「/admin/accounts という URL が割り振られている Web リソースにアクセス可能な場合は表示を許可する」というアクセスポリシーを定義しており、Web リソースに定義されているアクセスポリシーを直接意識する必要がない。

注釈: HTTP メソッドによるポリシーの指定

Web リソースのアクセスポリシーの定義をする際に、HTTP メソッドによって異なるアクセスポリシーを指定している場合は、<sec:authorize>タグの method 属性を指定して、連動させる定義を特定すること。

警告: 表示制御に関する留意点

ボタンやリンクなどの表示制御を行う場合は、必ず Web リソースに定義されているアクセスポリシーと連動させること。

ボタンやリンクに対して直接アクセスポリシーの指定を行い、Web リソース自体にアクセスポリシーを定義していないと、URL を直接してアクセスするような不正なアクセスを防ぐことができない。

認可処理の判定結果を変数に格納

<sec:authorize>タグを使って呼び出した認可処理の判定結果は、変数に格納して使いまわすことができる。

- JSP の実装例

```
<sec:authorize url="/admin/accounts"
               var="hasAccountsAuthority"/> <!-- (1) -->

<c:if test="${hasAccountsAuthority}"> <!-- (2) -->
    <!-- omitted -->
</c:if>
```

項番	説明
(1)	var 属性に判定結果を格納するための変数名を指定する。 アクセスが許可された場合は、変数に true が設定される。
(2)	変数の値を参照して表示処理を実装する。

認可エラー時のレスポンス

Spring Security は、リソースへのアクセスを拒否した場合、以下のような流れでエラーをハンドリングしてレスポンスの制御を行う。

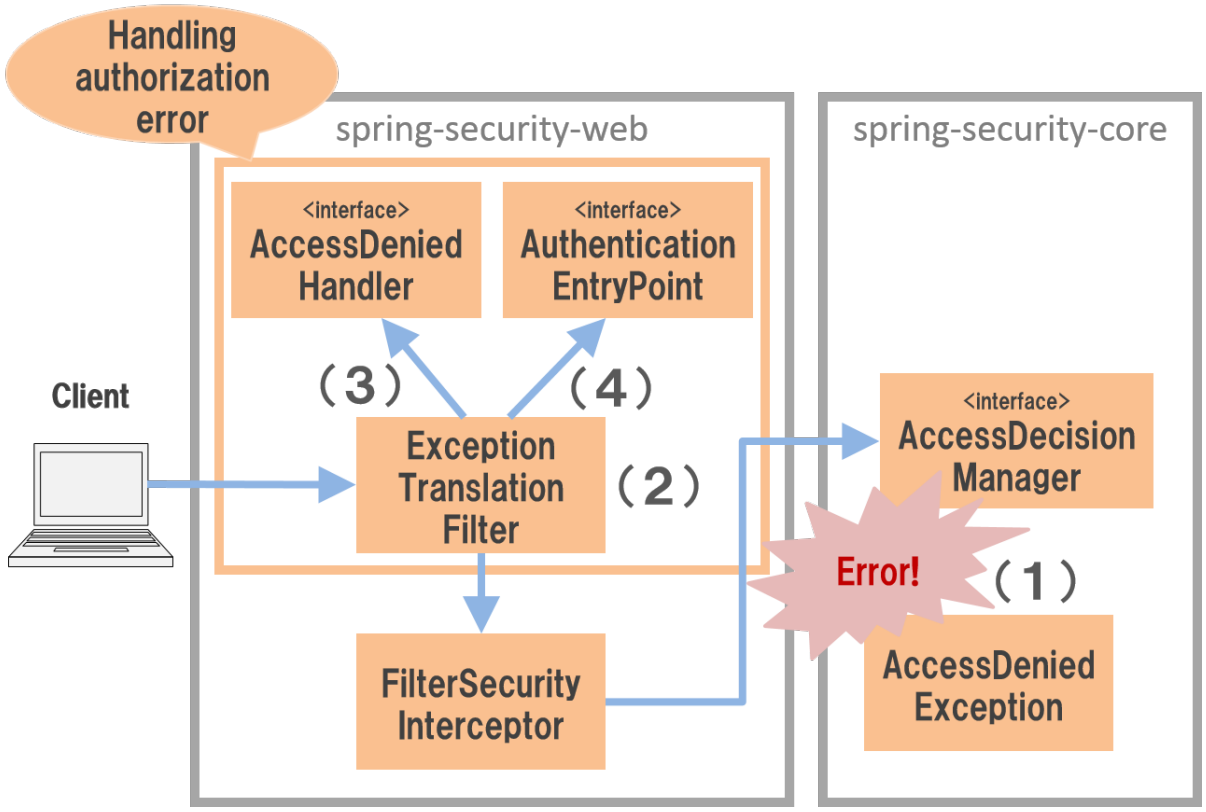


図 9.8　認可エラーのハンドリングの仕組み

項番	説明
(1)	Spring Security は、リソースやメソッドへのアクセスを拒否するために、`AccessDeniedException` を発生させる。
(2)	`ExceptionTranslationFilter` クラスは、`AccessDeniedException` をキャッチし、`AccessDeniedHandler` または `AuthenticationEntryPoint` インタフェースのメソッドを呼び出してエラー応答を行う。
(3)	認証済みのユーザーからのアクセスの場合は、`AccessDeniedHandler` インタフェースのメソッドを呼び出してエラー応答を行う。
(4)	未認証のユーザーからのアクセスの場合は、`AuthenticationEntryPoint` インタフェースのメソッドを呼び出してエラー応答を行う。

AccessDeniedHandler

AccessDeniedHandler インタフェースは、認証済みのユーザーからのアクセスを拒否した際のエラー応答を行うためのインタフェースである。Spring Security は、AccessDeniedHandler インタフェースの実装クラスとして以下のクラスを提供している。

表 9.31 Spring Security が提供する AccessDeniedHandler の実装クラス

クラス名	説明
AccessDeniedHandlerImpl	HTTP レスポンスコードに 403(Forbidden) を設定し、指定されたエラーページに遷移する。 エラーページの指定がない場合は、HTTP レスポンスコードに 403(Forbidden) を設定してエラー応答 (HttpServletResponse#sendError) を行う。
InvalidSessionAccessDeniedHandler	InvalidSessionStrategy インタフェースの実装クラスに処理を委譲する。 このクラスは、CSRF 対策とセッション管理機能を使用してセッションタイムアウトを検知する設定を有効にした際に、CSRF トークンがセッションに存在しない (つまりセッションタイムアウトが発生している) 場合に使用される。
DelegatingAccessDeniedHandler	AccessDeniedException と AccessDeniedHandler インタフェースの実装クラスのマッピングを行い、発生した AccessDeniedException に対応する AccessDeniedHandler インタフェースの実装クラスに処理を委譲する。 InvalidSessionAccessDeniedHandler はこの仕組みを利用して呼び出されている。

Spring Security のデフォルトの設定では、エラーページの指定がない AccessDeniedHandlerImpl が使用される。

AuthenticationEntryPoint

AuthenticationEntryPoint インタフェースは、未認証のユーザーからのアクセスを拒否した際のエラー応答を行うためのインタフェースである。Spring Security は、AuthenticationEntryPoint インタ

フェースの実装クラスとして以下のクラスを提供している。

表 9.32 Spring Security が提供する主な AuthenticationEntryPoint の実装クラス

クラス名	説明
LoginUrlAuthenticationEntryPoint	フォーム認証用のログインフォームを表示する。
BasicAuthenticationEntryPoint	Basic 認証用のエラー応答を行う。 具体的には、HTTP レスポンスコードに 401(Unauthorized) を、レスポンスヘッダとして Basic 認証用の「WWW-Authenticate」ヘッダを設定してエラー応答 (HttpServletResponse#sendError) を行う。
DigestAuthenticationEntryPoint	Digest 認証用のエラー応答を行う。 具体的には、HTTP レスポンスコードに 401(Unauthorized) を、レスポンスヘッダとして Digest 認証用の「WWW-Authenticate」ヘッダを設定してエラー応答 (HttpServletResponse#sendError) を行う。
Http403ForbiddenEntryPoint	HTTP レスポンスコードに 403(Forbidden) を設定してエラー応答 (HttpServletResponse#sendError) を行う。
DelegatingAuthenticationEntryPoint	RequestMatcher と AuthenticationEntryPoint インタフェースの実装クラスのマッピングを行い、HTTP リクエストに対応する AuthenticationEntryPoint インタフェースの実装クラスに処理を委譲する。

Spring Security のデフォルトの設定では、認証方式に対応する AuthenticationEntryPoint インタフェースの実装クラスが使用される。

認可エラー時の遷移先

Spring Security のデフォルトの設定だと、認証済みのユーザーからのアクセスを拒否した際は、アプリケーションサーバのエラーページが表示される。アプリケーションサーバーのエラーページを表示してしまうと、システムのセキュリティを低下させる要因になるため、適切なエラー画面を表示することを推奨する。エラーページの指定は、以下のような bean 定義を行うことで可能である。

- spring-security.xml の定義例

```
<sec:http>
    <!-- omitted -->
    <sec:access-denied-handler
        error-page="/WEB-INF/views/common/error/accessDeniedError.jsp" /> <!-- (1)
↪-->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	<sec:access-denied-handler>タグの error-page 属性に認可エラー用のエラーページを指定する。

ちなみに：サーブレットコンテナのエラーページ機能の利用

認可エラーのエラーページは、サーブレットコンテナのエラーページ機能を使って指定することもできる。

サーブレットコンテナのエラーページ機能を使う場合は、web.xml の<error-page>タグを使用してエラーページを指定する。

```
<error-page>
    <error-code>403</error-code>
    <location>/WEB-INF/views/common/error/accessDeniedError.jsp</location>
</error-page>
```

9.3.3 How to extend

本節では、Spring Security が用意しているカスタマイズポイントや拡張方法について説明する。

Spring Security は、多くのカスタマイズポイントを提供しているため、すべてのカスタマイズポイントは紹介しない。本節では代表的なカスタマイズポイントに絞って説明を行う。

認可エラー時のレスポンス (認証済みユーザー編)

ここでは、認証済みユーザーからのアクセスを拒否した際の動作をカスタマイズする方法を説明する。

AccessDeniedHandler の適用

Spring Security が提供しているデフォルトの動作をカスタマイズする仕組みだけでは要件をみたせない場合は、AccessDeniedHandler インタフェースの実装クラスを直接適用することができる。

例えば、Ajax のリクエスト (REST API など) で認可エラーが発生した場合は、エラーページ (HTML) ではなく JSON 形式でエラー情報を応答することが求められるケースがある。そのような場合は、AccessDeniedHandler インタフェースの実装クラスを作成して Spring Security に適用することで実現することができる。

- AccessDeniedHandler インタフェースの実装クラスの作成例

```
public class JsonDelegatingAccessDeniedHandler implements AccessDeniedHandler {

    private final RequestMatcher jsonRequestMatcher;
    private final AccessDeniedHandler delegateHandler;

    public JsonDelegatingAccessDeniedHandler(
            RequestMatcher jsonRequestMatcher, AccessDeniedHandler␣
↪delegateHandler) {
        this.jsonRequestMatcher = jsonRequestMatcher;
        this.delegateHandler = delegateHandler;
    }

    public void handle(HttpServletRequest request, HttpServletResponse response,
                       AccessDeniedException accessDeniedException)
            throws IOException, ServletException {
        if (jsonRequestMatcher.matches(request)) {
            // response error information of JSON format
            response.setStatus(HttpServletResponse.SC_FORBIDDEN);
            // omitted
        } else {
            // response error page of HTML format
            delegateHandler.handle(
                    request, response, accessDeniedException);
        }
    }

}
```

- spring-security.xml の定義例

```
<!-- (1) -->
<bean id="accessDeniedHandler"
      class="com.example.web.security.JsonDelegatingAccessDeniedHandler">
    <constructor-arg>
        <bean class="org.springframework.security.web.util.matcher.
↪AntPathRequestMatcher">
            <constructor-arg value="/api/**"/>
```

```
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
            <property name="errorPage"
                      value="/WEB-INF/views/common/error/accessDeniedError.jsp"/>
        </bean>
    </constructor-arg>
</bean>

<sec:http>
    <!-- omitted -->
    <sec:access-denied-handler ref="accessDeniedHandler" />  <!-- (2) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	AccessDeniedHandler インタフェースの実装クラスを bean 定義して DI コンテナに登録する。
(2)	<sec:access-denied-handler>タグの ref 属性に AccessDeniedHandler の bean を指定する。

認可エラー時のレスポンス (未認証ユーザー編)

ここでは、未認証ユーザーからのアクセスを拒否した際の動作をカスタマイズする方法を説明する。

リクエスト毎に AuthenticationEntryPoint を適用

認証済みユーザーと同様に、Ajax のリクエスト (REST API など) で認可エラーが発生した場合は、ログインページ (HTML) ではなく JSON 形式でエラー情報を応答することが求められるケースがある。そのような場合は、リクエストのパターン毎に AuthenticationEntryPoint インタフェースの実装クラスを Spring Security に適用することで実現することができる。

- spring-security.xml の定義例

```
<!-- (1) -->
<bean id="authenticationEntryPoint"
      class="org.springframework.security.web.authentication.
↪DelegatingAuthenticationEntryPoint">
    <constructor-arg>
        <map>
            <entry>
                <key>
                    <bean class="org.springframework.security.web.util.matcher.
↪AntPathRequestMatcher">
```

```
                        <constructor-arg value="/api/**"/>
                    </bean>
                </key>
                <bean class="com.example.web.security.JsonAuthenticationEntryPoint
↪"/>
            </entry>
        </map>
    </constructor-arg>
    <property name="defaultEntryPoint">
        <bean class="org.springframework.security.web.authentication.
↪LoginUrlAuthenticationEntryPoint">
            <constructor-arg value="/login"/>
        </bean>
    </property>
</bean>

<sec:http entry-point-ref="authenticationEntryPoint"> <!-- (2) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	AuthenticationEntryPoint インタフェースの実装クラスを bean 定義して DI コンテナに登録する。 ここでは、Spring Security が提供している DelegatingAuthenticationEntryPoint クラスを利用して、リクエストのパターン毎に AuthenticationEntryPoint インタフェースの実装クラスを適用している。
(2)	<sec:http>タグの entry-point-ref 属性に AuthenticationEntryPoint の bean を指定する。

注釈: デフォルトで適用される AuthenticationEntryPoint

リクエストに対応する AuthenticationEntryPoint インタフェースの実装クラスの指定がない場合は、Spring Security がデフォルトで定義する AuthenticationEntryPoint インタフェースの実装クラスが使用される仕組みになっている。認証方式としてフォーム認証を使用する場合は、LoginUrlAuthenticationEntryPoint クラスが使用されログインフォームが表示される。

ロールの階層化

認可処理では、ロールに階層関係を設けることができる。

上位に指定したロールは、下位のロールにアクセスが許可されているリソースにもアクセスすることができる。ロールの関係が複雑な場合は、階層関係も設けることも検討されたい。

例えば、「ROLE_ADMIN」が上位ロール、「ROLE_USER」が下位ロールという階層関係を設けた場合、下記のようアクセスポリシーを設定すると、「ROLE_ADMIN」権限を持つユーザーは、`/user` 配下のパス (「ROLE_USER」権限を持つユーザーがアクセスできるパス) にアクセスすることができる。

- spring-security.xml の定義例

```
<sec:http>
    <sec:intercept-url pattern="/user/**" access="hasAnyRole('USER')" />
    <!-- omitted -->
</sec:http>
```

階層関係の設定

ロールの階層関係は、`org.springframework.security.access.hierarchicalroles.RoleHierarchy` インタフェースの実装クラスで解決する。

- spring-security.xml の定義例

```
<bean id="roleHierarchy"
    class="org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl
↪"> <!-- (1) -->
    <property name="hierarchy"> <!-- (2) -->
        <value>
            ROLE_ADMIN > ROLE_STAFF
            ROLE_STAFF > ROLE_USER
        </value>
    </property>
</bean>
```

項番	説明
(1)	`org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl` クラスを指定する。 `RoleHierarchyImpl` は、Spring Security が提供するデフォルトの実装クラスである。
(2)	`hierarchy` プロパティに階層関係を定義する。 書式: [上位ロール] > [下位ロール] 上記例では、 STAFF は、USER に認可されたリソースにもアクセス可能である。 ADMIN は、USER と STAFF に認可されたリソースにもアクセス可能である。

Web リソースの認可処理への適用

ロールの階層化を、Web リソースと JSP の画面項目に対する認可処理に適用する方法を説明する。

- spring-security.xml の定義例

```
<!-- (1) -->
<bean id="webExpressionHandler"
    class="org.springframework.security.web.access.expression.
↪DefaultWebSecurityExpressionHandler">
    <property name="roleHierarchy" ref="roleHierarchy"/>  <!-- (2) -->
</bean>

<sec:http>
    <!-- omitted -->
    <sec:expression-handler ref="webExpressionHandler" />  <!-- (3) -->
</sec:http>
```

項番	説明
(1)	`org.springframework.security.web.access.expression.DefaultWebSecurityExpressionHandler` の Bean を定義する。
(2)	`roleHierarchy` プロパティに `RoleHierarchy` インタフェースの実装クラスの Bean を指定する。
(3)	`<sec:expression-handler>`タグの `ref` 属性に、`org.springframework.security.access.expression.SecurityExpressionHandler` インタフェースの実装クラスの Bean を指定する。

メソッドの認可処理への適用

ロールの階層化を、Java メソッドに対する認可処理に適用する方法を説明する。

- spring-security.xml の定義例

```
<bean id="methodExpressionHandler"
    class="org.springframework.security.access.expression.method.
↪DefaultMethodSecurityExpressionHandler"> <!-- (1) -->
    <property name="roleHierarchy" ref="roleHierarchy"/> <!-- (2) -->
</bean>

<sec:global-method-security pre-post-annotations="enabled">
    <sec:expression-handler ref="methodExpressionHandler" /> <!-- (3) -->
</sec:global-method-security>
```

項番	説明
(1)	`org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler` の Bean を定義する。
(2)	`roleHierarchy` プロパティに `RoleHierarchy` インタフェースの実装クラスの Bean を指定する。
(3)	`<sec:expression-handler>` タグの `ref` 属性に、`org.springframework.security.access.expression.SecurityExpressionHandler` インタフェースの実装クラスの Bean を指定する。

9.4 セッション管理

9.4.1 Overview

本節では、「Web アプリケーションでセッションを扱う際に必要となるセキュリティ対策」及び「Spring Security が提供しているセッション関連の機能」について説明する。

セッション利用時のセキュリティ対策

Web アプリケーションでセッションを扱う場合、一般的には以下の攻撃に対して対策が必要となる。

対策	説明
セッションハイジャック攻撃	通信の盗聴、規則性からの類推、クロスサイトスクリプティングなどを駆使してセッション ID を盗みとり、盗みとったセッション ID をつかっているユーザーになりすましてシステムを利用する攻撃。
セッション固定攻撃	攻撃者が事前に払い出したセッション ID を他人に使わせてシステムにログインさせ、攻撃者がログインしたユーザーになりすましてシステムを利用する攻撃。

セッションハイジャック攻撃への対策

セッションハイジャック攻撃への対策は、セッション ID が盗み取られないようにするしかない。いったん盗み取られてしまうと、アプリケーションサーバは正規のユーザーからのリクエストなのか、攻撃者からのリクエストなのかを判断することができない。

このようなセッションハイジャック攻撃からアプリケーションを守るためには、以下のような対策が必要である。

表 9.33　セッションハイジャック攻撃への対策

対策	説明
推測困難なセッション ID の生成する	連番など推測できる値をセッション ID に使用せず、推測が困難な (セキュアな) ランダム値を使用する。 基本的にはアプリケーションサーバが提供するセッション ID の生成機構を利用すればよい。
HTTPS を使って通信を暗号化する	盗まれると困る情報をやりとりする通信は、HTTPS プロトコルを使って暗号化する。 通信の盗聴はフリーのソフトなどを使って簡単に行うことができため、盗聴されても解読されないように暗号化しておくことが重要である。
セッション ID は Cookie を使って連携する	クライアントとサーバーとの間でセッション ID を連携する際は、Cookie を使って連携するように設定し、URL Rewriting 機能を無効化する。
Cookie の `HttpOnly` 属性を指定する	Cookie の `HttpOnly` 属性を指定すると、JavaScript から Cookie にアクセスすることができなくため、クロスサイトスクリプティングを使ってセッション ID を盗むことができなくなる。
Cookie に `Secure` 属性を指定する	Cookie に `Secure` 属性を指定すると、HTTPS 通信の時だけ Cookie をサーバーに送信するため、誤って HTTP 通信を使ってしまった時にセッション ID が盗み取られるリスクを減らすことができる。

注釈: URL Rewriting

URL Rewriting は、Cookie を使用できないクライアントとセッションを維持するための仕組みである。具体的には、URL のリクエストパラメータの中にセッション ID を含めることでクライアントとサーバーの間でセッション ID を連携する。

- URL Rewriting が行われた URL 例

```
http://localhost:8080/;jsessionid=7E6EDE4D3317FC5F14FD912BEAC96646
```

`jsessionid=7E6EDE4D3317FC5F14FD912BEAC96646` の部分が URL Rewriting されたセッション ID

になる。Servlet の API 仕様では、以下のメソッドを呼び出すと URL Rewriting が行われる可能性があり、JSTL や Spring が提供している JSP タグライブラリの中でもこれらのメソッドを呼び出している。

- `HttpServletResponse#encodeURL(String)`
- `HttpServletResponse#encodeRedirectURL(String)`

URL Rewriting が行われると URL 内にセッション ID が露出してしまうため、セッション ID を盗まれるリスクが高くなる。そのため、Cookie を使うことができるクライアントのみをサポートする場合は、サーブレットコンテナの URL Rewriting 機能を無効化することを推奨する。

セッション固定攻撃への対策

セッション固定攻撃からアプリケーションを守るためには、以下のような対策が必要になる。

表 9.34　セッション固定攻撃への対策

対策	説明
URL Rewriting 機能を無効化する	URL Rewriting 機能を無効化すると、攻撃者が事前に払い出したセッション ID が使われず、新たにセッションが開始される。
ログイン後にセッション ID を変更する	ログイン後にセッション ID を変更することで、攻撃者が事前に払い出したセッション ID が使用できなくなる。

Spring Security が提供するセッション管理機能

Spring Security では、セッションについて、主に以下の機能が提供されている。

表 9.35　セッションに関する提供機能

機能	説明
セキュリティ対策	セッションハイジャック攻撃等のセッション ID を使用した攻撃への対策機能。
ライフサイクル制御	セッションの生成～破棄までのライフサイクルを制御する機能。
タイムアウト制御	タイムアウトにより、セッションを破棄する機能。
多重ログイン制御	同一ユーザーによる多重ログイン時のセッションを制御する機能。

9.4.2 How to use

セッションハイジャック攻撃への対策

ここでは URL Rewriting 機能を無効化し、Cookie を使用してセッション ID を連携する方法を説明する。

Spring Security による URL Rewriting 機能の無効化

Spring Security は URL Rewriting を無効化するための仕組みを提供しており、この機能はデフォルトで適用されている。Cookie を使えないクライアントをサポートする必要がある場合は、URL Rewriting を許可するように Bean 定義する。

- spring-security.xml の定義例

```
<sec:http disable-url-rewriting="false"> <!-- false を指定して URL Rewriting を有効化 -->
```

項番	説明
(1)	Spring Security のデフォルトでは、`disable-url-rewriting` の値は `true` であるため、URL Rewriting は行われない。 URL Rewriting を有効にする際は、`<sec:http>`要素の `disable-url-rewriting` 属性に `false` を設定する。

サーブレットコンテナによる URL Rewriting 機能の無効化

Servlet の標準仕様の仕組みを使ってセッションをセキュアに扱うことが可能である。

- web.xml の定義例

```
<session-config>
    <cookie-config>
        <http-only>true</http-only> <!-- (1)  -->
    </cookie-config>
    <tracking-mode>COOKIE</tracking-mode> <!-- (2) -->
</session-config>
```

項番	説明
(1)	Cookie に HttpOnly 属性を付与する場合は、<http-only>要素に true を指定する。 使用するアプリケーションサーバによっては、デフォルト値が true になっている。
(3)	URL Rewriting 機能を無効化する場合は、<tracking-mode>要素に COOKIE を指定する。

上記の定義例からは省略しているが、<cookie-config>に <secure>true</secure>を追加することで、 Cookie に Secure 属性を付与することができる。ただし、cookie の secure 化は、web.xml で指定するのではなく、クライアントと HTTPS 通信を行うミドルウェア (SSL アクセラレータや Web サーバーなど) で付与する方法を検討されたい。

実際のシステム開発の現場において、ローカルの開発環境で HTTPS を使うケースはほとんどない。また、本番環境においても、HTTPS を使うのは SSL アクセラレータや Web サーバーとの通信までで、アプリケーションサーバへの通信は HTTP で行うケースも少なくない。このような環境下で Secure 属性の指定を web.xml で行ってしまうと、実行環境毎に web.xml や web-fragment.xml を用意することになり、ファイルの管理が煩雑になるため推奨されない。

セッション管理機能の適用

Spring Security のセッション管理機能を適用する方法を説明する。Spring Security のセッション管理機能の処理を使用する場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<sec:http>
    <!-- ommited -->
    <sec:session-management /> <!-- (1) -->
    <!-- ommited -->
</sec:http>
```

項番	説明
(1)	<sec:http>要素の子要素として<sec:session-management>要素を指定する。 <sec:session-management>要素を指定すると、セッション管理機能が適用される。

セッション固定攻撃への対策

Spring Security は、セッション固定攻撃対策として、ログイン成功時にセッション ID を変更するためのオプションを 4 つ用意している。

表 9.36　セッション固定攻撃への対策のオプション

オプション	説明
changeSessionId	Servlet 3.1 で追加された HttpServletRequest#changeSessionId() を使用してセッション ID を変更する。 (これは Servlet 3.1 以上のコンテナ上でのデフォルトの動作である)
migrateSession	ログイン前に使用していたセッションを破棄し、新たにセッションを作成する。 このオプションを使用すると、ログイン前にセッションに格納されていたオブジェクトは新しいセッションに引き継がれる。 (Servlet 3.0 以下のコンテナ上でのデフォルトの動作である)
newSession	このオプションは migrateSession と同じ方法でセッション ID を変更するが、ログイン前に格納されていたオブジェクトは新しいセッションに引き継がれない。
none	Spring Security は、セッション ID を変更しない。

デフォルトの動作を変更したい場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<sec:session-management
        session-fixation-protection="newSession"/> <!-- (1) -->
```

項番	説明
(1)	<sec:session-management>要素の session-fixation-protection 属性にセッション固定攻撃の対策方法を指定する。

セッションのライフサイクル制御

Spring Security は、リクエストを跨いで認証情報などのオブジェクトを共有するための手段として HTTP セッションを使用しており、Spring Security の処理の中でセッションのライフサイクル (セッションの作成と破棄) を制御している。

注釈: セッション情報の格納先

Spring Security が用意しているデフォルト実装では HTTP セッションを使用するが、HTTP セッション以外 (データベースやキーバリューストアなど) にオブジェクトを格納することも可能なアーキテクチャになっている。

セッションの作成

Spring Security の処理の中でどのような方針でセッションを作成して利用するかは、以下のオプションから選択することができる。

表 9.37　セッションの作成方針

オプション	説明
always	セッションが存在しない場合は、無条件に新たなセッションを生成する。 このオプションを指定すると、Spring Security の処理でセッションを使わないケースでもセッションが作成される。
ifRequired	セッションが存在しない場合は、セッションにオブジェクトを格納するタイミングで新たなセッションを作成して利用する。(デフォルトの動作)
never	セッションが存在しない場合は、セッションの生成及び利用は行わない。 ただし、既にセッションが存在している場合はセッションを利用する。
stateless	セッションの有無に関係なく、セッションの生成及び利用は行わない。

デフォルトの振る舞いを変更したい場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<sec:http create-session="stateless"> <!-- (1) -->
    <!-- ommited -->
</sec:http>
```

項番	説明
(1)	<sec:http>要素の create-session 属性に、変更したいセッションの作成方針を指定する。

セッションの破棄

Spring Security は、以下のタイミングでセッションを破棄する。

- ログアウト処理が実行されたタイミング
- 認証処理が成功したタイミング (セッション固定攻撃対策として migrateSession 又は newSession が適用されるとセッションが破棄される)

セッションタイムアウトの制御

セッションにオブジェクトを格納する場合、適切なセッションタイムアウト値を指定して、一定時間操作がないユーザーとのセッションを自動で破棄するようにするのが一般的である。

セッションタイムアウトの指定

セッションタイムアウトは、サーブレットコンテナに対して指定する。アプリケーションサーバーによっては、サーバー独自の指定方法を用意しているケースもあるが、ここでは、Servlet 標準仕様で定められた指定方法を説明する。

- web.xml の定義例

```
<session-config>
    <session-timeout>60</session-timeout> <!-- (1) -->
    <!-- ommited -->
</session-config>
```

項番	説明
(1)	<session-timeout>要素に適切なタイムアウト値 (分単位) を指定する。 タイムアウト値を指定しない場合は、サーブレットコンテナが用意しているデフォルト値が適用される。 また、0 以下の値を指定するとサーブレットコンテナのセッションタイム機能が無効化される。

無効なセッションを使ったリクエストの検知

Spring Security は、無効なセッションを使ったリクエストを検知する機能を提供している。無効なセッションとして扱われるリクエストの大部分は、セッションタイムアウト後のリクエストである。デフォルトではこの機能は無効になっているが、以下のような bean 定義を行うことで有効化することができる。

- spring-security.xml の定義例

```
<sec:session-management
        invalid-session-url="/error/invalidSession"/>
```

項番	説明
(1)	<sec:session-management>要素の invalid-session-url 属性に、無効なセッションを使ったリクエストを検知した際のリダイレクト先のパスを指定する。

除外パスの指定

無効なセッションを使ったリクエストを検知する機能を有効にすると、Spring Security のサーブレットフィルタを通過するすべてのリクエストに対してチェックが行われる。そのため、セッションが無効な状態でアクセスしても問題がないページにアクセスした場合もチェックが行われる。

この動作を変更したい場合は、チェック対象から除外したいパスに対して個別に bean 定義を行うことで実現することが可能である。例として、トップページを開くためのパス (「/」) を除外パスに指定したい場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<!-- (1) -->
<sec:http pattern="/"> <!-- (2) -->
    <sec:session-management />
</sec:http>

<!-- (3) -->
<sec:http>
    <!-- ommited -->
    <sec:session-management
            invalid-session-url="/error/invalidSession"/>
    <!-- ommited -->
</sec:http>
```

項番	説明
(1)	トップページを開くためのパス (「/」) に適用する SecurityFilterChain を作成するための<sec:http>要素を新たに追加する。
(2)	(1) の<sec:http>要素を使って生成した SecurityFilterChain を適用するパスパターンを指定する。 指定可能なパスパターンは Ant 形式のパス表記と正規表現の 2 つの形式であり、デフォルトでは Ant 形式のパスとして扱われる。 また、パスパターンではなく RequestMatcher オブジェクトを直接指定することも可能である。
(3)	個別定義していないパスに適用する SecurityFilterChain を作成するための<sec:http>要素を定義する。 この定義は、個別定義用の<sec:http>要素より下に定義すること。 これは<sec:http>要素の定義順番が SecurityFilterChain の優先順位となるためである。

多重ログインの制御

Spring Security は、同じユーザー名 (ログイン ID) を使った多重ログインを制御する機能を提供している。デフォルトではこの機能は無効になっているが、セッションのライフサイクル検知の有効化 を行うことで有効化することができる。

警告: 多重ログイン制御における制約

Spring Security が提供しているデフォルト実装では、ユーザー毎のセッション情報をアプリケーションサーバーのメモリ内で管理しているため、以下の 2 つの制約がある。

ひとつめの制約として、複数のアプリケーションサーバーを同時に起動するシステムでは、デフォルト実装を利用することができないことが挙げられる。複数のアプリケーションサーバーを同時に使用する場合は、ユーザー毎のセッション情報をデータベースやキーバリューストア (キャッシュサーバー) などの共有領域で管理する実装クラスの作成が必要になる。

ふたつめの制約は、アプリケーションサーバーを停止または再起動時した際に、セッション情報が復元されると、正常動作しない可能性があるという点である。使用するアプリケーションサーバーによっては、停止または再起動時のセッション状態を復元する機能をもっているため、実際のセッション状態と Spring Security が管理しているセッション情報に不整合が生じることになる。このような不整合が生まれる可能性がある場合は、以下のいずれかの対応が必要になる。

- アプリケーションサーバ側のセッション状態が復元されないようにする。
- Spring Security 側のセッション情報を復元する仕組みを実装する。
- HTTP セッション以外 (データベースやキーバリューストアなど) にオブジェクトを格納する。

本節では、Spring Security のデフォルト実装を使用する方法を紹介する。Spring Security が用意しているデフォルト実装では HTTP セッションを使用するが、HTTP セッション以外 (データベースやキーバリューストアなど) にオブジェクトを格納することも可能なアーキテクチャになっている。ただし、ここで紹介する方法は **上記 Warning の制約が残っている実装方法であるため** 、適用する際は注意されたい。

セッションのライフサイクル検知の有効化

多重ログインを制御する機能は、セッションのライフサイクル *(セッションの生成と破棄)* を検知する仕組みを利用してユーザー毎のセッション状態を管理している。このため、多重ログインの制御機能を使用する際は、Spring Security から提供されている `HttpSessionEventPublisher` クラスをサーブレットコンテナに登録する必要がある。

- web.xml の定義例

```
<listener>
    <!-- (1) -->
    <listener-class>
        org.springframework.security.web.session.HttpSessionEventPublisher
```

```
    </listener-class>
</listener>
```

項番	説明
(1)	サーブレットリスナとして HttpSessionEventPublisher を登録する。

多重ログインの禁止 (先勝ち)

同じユーザー名 (ログイン ID) を使って既にログインしているユーザーがいる場合に、認証エラーを発生させて多重ログインを防ぐ場合は、以下のような bean 定義を行う。

- bean 定義ファイルの定義例

```
<sec:session-management>
    <sec:concurrency-control
            max-sessions="1"
            error-if-maximum-exceeded="true"/> <!-- (1) (2) -->
</sec:session-management>
```

項番	説明
(1)	<sec:concurrency-control>要素の max-sessions 属性に、同時にログインを許可するセッション数を指定する。多重ログインを防ぎたい場合は、通常」1」 を指定する。
(2)	<sec:concurrency-control>要素の error-if-maximum-exceeded 属性に、同時にログインできるセッション数を超えた時の動作を指定する。既にログインしているユーザーを有効なユーザーとして扱う場合は、true を指定する。

多重ログインの禁止 (後勝ち)

同じユーザー名 (ログイン ID) を使って既にログインしているユーザーがいる場合に、既にログインしているユーザーを無効化することで多重ログインを防ぐ場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<sec:session-management>
    <sec:concurrency-control
            max-sessions="1"
            error-if-maximum-exceeded="false"
            expired-url="/error/expire"/> <!-- (1) (2) -->
</sec:session-management>
```

項番	説明
(1)	<sec:concurrency-control>要素の error-if-maximum-exceeded 属性に、同時にログインできるセッション数を超えた時の動作を指定する。 新たにログインしたユーザーを有効なユーザーとして扱う場合は、false を指定する。
(2)	<sec:concurrency-control>要素の expired-url 属性に、無効化されたユーザーからのリクエストを検知した際のリダイレクト先のパスを指定する。 これは<sec:http>要素の定義順番が SecurityFilterChain の優先順位となるためである。

9.5 CSRF 対策

9.5.1 Overview

本節では、Spring Security が提供している Cross site request forgeries(以下、CSRF と略す）対策の機能について説明する。

CSRF とは、Web サイトにスクリプトや自動転送 (HTTP リダイレクト) を実装することにより、ログイン済みの別の Web サイト上で、ユーザーが意図しない何らかの操作を行わせる攻撃手法のことである。

サーバ側で CSRF を防ぐには、以下の方法が知られている。

- 秘密情報 (トークン) の埋め込み
- パスワードの再入力
- Referer のチェック

CSRF 対策機能は、攻撃者が用意した Web ページから送られてくる偽造リクエストを不正なリクエストとして扱うための機能である。CSRF 対策が行われていない Web アプリケーションを利用すると、以下のような方法で攻撃を受ける可能性がある。

- 利用者は、CSRF 対策が行われていない Web アプリケーションにログインする。
- 利用者は、攻撃者からの巧みな誘導によって、攻撃者が用意した Web ページを開いてしまう。
- 攻撃者が用意した Web ページは、フォームの自動送信などのテクニックを使用して、偽造したリクエストを CSRF 対策が行われていない Web アプリケーションに対して送信する。
- CSRF 対策が行われていない Web アプリケーションは、攻撃者が偽造したリクエストを正規のリクエストとして処理してしまう。

ちなみに： OWASP[*1] では、トークンパターンを使用する方法が推奨されている。

注釈：ログイン時における CSRF 対策

CSRF 対策はログイン中のリクエストだけではなく、ログイン処理でも行う必要がある。ログイン処理に対して CSRF 対策を怠った場合、攻撃者が用意したアカウントを使って知らぬ間にログインさせられ、ログイン中に行った操作履歴などを盗まれる可能性がある。

[*1] Open Web Application Security Project の略称であり、信頼できるアプリケーションや、セキュリティに関する 効果的なアプローチなどを検証、提唱する、国際的な非営利団体である。https://www.owasp.org/index.php/Main_Page

警告: マルチパートリクエスト (ファイルアップロード) 時における CSRF 対策

ファイルアップロード時の CSRF 対策については、ファイルアップロード *Servlet Filter* の設定を留意されたい。

Spring Security の CSRF 対策

Spring Security は、セッション単位にランダムに生成される固定トークン値 (CSRF トークン) を払い出し、払い出された CSRF トークンをリクエストパラメータ (HTML フォームの hidden 項目) として送信する。これにより正規の Web ページからのリクエストなのか、攻撃者が用意した Web ページからのリクエストなのかを判断する仕組みを採用している。

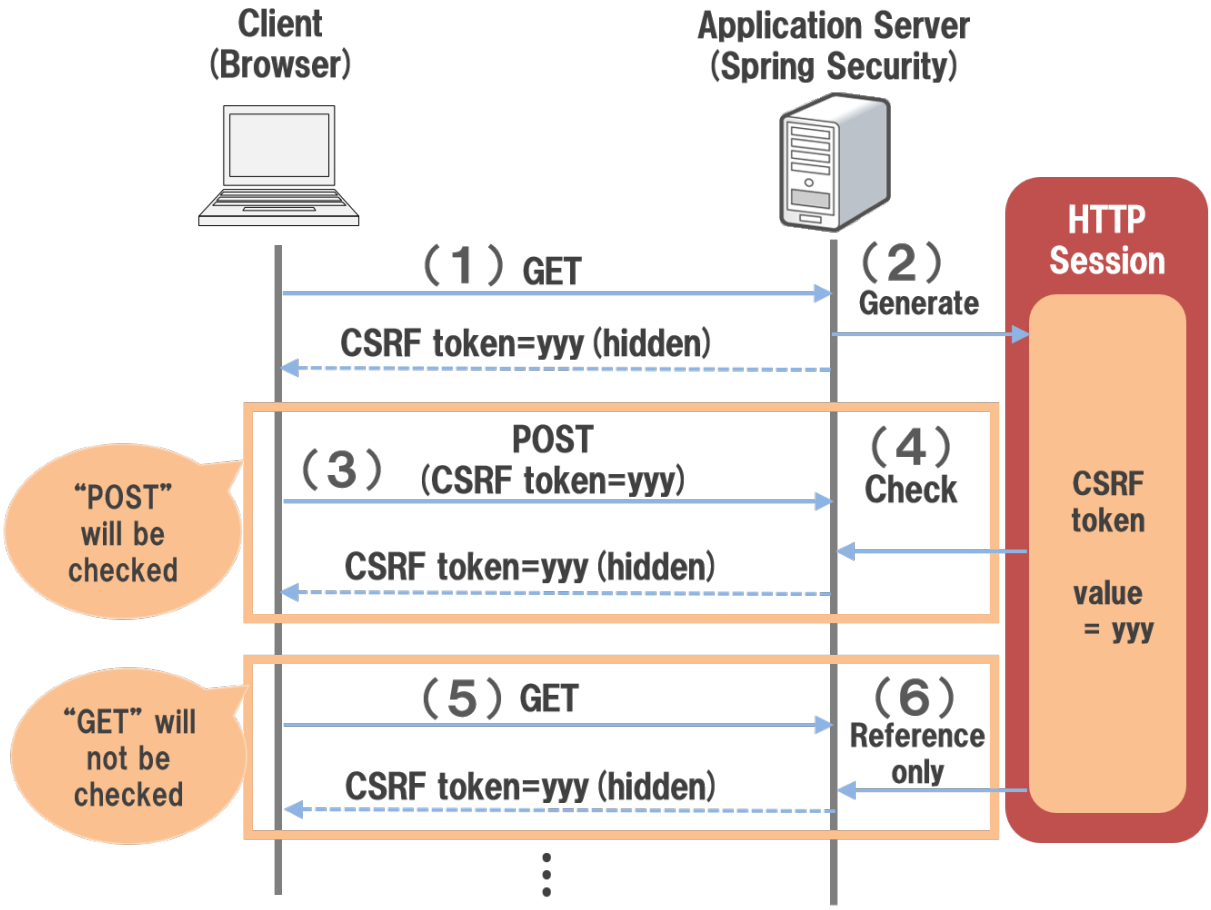


図 9.9 Spring Security の CSRF 対策の仕組み

項番	説明
(1)	クライアントは、HTTP の GET メソッドを使用してアプリケーションサーバにアクセスする。
(2)	Spring Security は、CSRF トークンを生成し HTTP セッションに格納する。 生成した CSRF トークンは、HTML フォームの hidden タグを使ってクライアントと連携する。
(3)	クライアントは、HTML フォーム内のボタンを押下してアプリケーションサーバーにリクエストを送信する。 HTML フォーム内の hidden 項目に CSRF トークンが埋め込まれているため、CSRF トークン値はリクエストパラメータとして送信される。
(4)	Spring Security は、HTTP の POST メソッドを使ってアクセスされた際は、リクエストパラメータに指定された CSRF トークン値と HTTP セッション内に保持している CSRF トークン値が同じ値であることをチェックする。 トークン値が一致しない場合は、不正なリクエスト (攻撃者からのリクエスト) としてエラーを発生させる。
(5)	クライアントは、HTTP の GET メソッドを使用してアプリケーションサーバにアクセスする。
(6)	Spring Security は、GET メソッドを使ってアクセスされた際は、CSRF トークン値のチェックは行わない。

注釈: Ajax 使用時の CSRF トークン

Spring Security は、リクエストヘッダに CSRF トークン値を設定することができるため、Ajax 向けのリクエストなどに対して CSRF 対策を行うことが可能である。

トークンチェックの対象リクエスト

Spring Security のデフォルト実装では、以下の HTTP メソッドを使用したリクエストに対して、CSRF トークンチェックを行う。

- POST
- PUT
- DELETE
- PATCH

注釈: **CSRF** トークンチェックを行わない理由

GET, HEAD, OPTIONS, TRACE メソッドがチェック対象外となっている理由は、これらのメソッドがアプリケーションの状態を変更するようなリクエストを実行するためのメソッドではないためである。

9.5.2 How to use

CSRF 対策機能の適用

CSRF トークン用の `RequestDataValueProcessor` 実装クラスを利用し、Spring のタグライブラリの`<form:form>`タグを使うことで、自動的に CSRF トークンを、hidden 項目に埋め込むことができる。

- spring-mvc.xml の設定例

```
<bean id="requestDataValueProcessor"
    class="org.terasoluna.gfw.web.mvc.support.CompositeRequestDataValueProcessor">
↪<!-- (1)  -->
    <constructor-arg>
        <util:list>
            <bean
                class="org.springframework.security.web.servlet.support.csrf.
↪CsrfRequestDataValueProcessor" /> <!-- (2)  -->
            <bean
                class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenRequestDataValueProcessor" />
        </util:list>
    </constructor-arg>
</bean>
```

項番	説明
(1)	共通ライブラリから提供されている、`org.springframework.web.servlet.support.RequestDataValueProcessor` を複数定義可能な `org.terasoluna.gfw.web.mvc.support.CompositeRequestDataValueProcessor` を bean 定義する。
(2)	コンストラクタの第 1 引数に、`org.springframework.security.web.servlet.support.csrf.CsrfRequestDataValueProcessor` の bean 定義を設定する。

Spring Security 4.0 からは、上記設定により、デフォルトで CSRF 対策機能が有効となる。このため、CSRF 対策機能を適用したくない場合は、明示的に無効化する必要がある。

CSRF 対策機能を使用しない場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<sec:http>
    <!-- omitted -->
    <sec:csrf disabled="true"/> <!-- disabled属性にtrueを設定して無効化 -->
    <!-- omitted -->
</sec:http>
```

CSRF トークン値の連携

Spring Security は、CSRF トークン値をクライアントとサーバー間で連携する方法として、以下の 2 種類の方法を提供している。

- HTML フォームの hidden 項目として CSRF トークン値を出力し、リクエストパラメータとして連携する
- HTML の meta タグとして CSRF トークンの情報を出力し、Ajax 通信時にリクエストヘッダにトークン値を設定して連携する

Spring MVC を使用した連携

Spring Security は、Spring MVC と連携するためのコンポーネントをいくつか提供している。ここでは、CSRF 対策機能と連携するためのコンポーネントの使い方を説明する。

hidden 項目の自動出力

HTML フォームを作成する際は、以下のような JSP の実装を行う。

- JSP の実装例

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<c:url var="loginUrl" value="/login"/>
<form:form action="${loginUrl}"> <!-- (1) -->
    <!-- omitted -->
</form:form>
```

項番	説明
(1)	HTML フォームを作成する際は、Spring MVC から提供されている<form:form>要素を使用する。

Spring MVC から提供されている<form:form>要素を使うと、以下のような HTML フォームが作成される。

- HTML の出力例

```
<form id="command" action="/login" method="post">
    <!-- omitted -->
    <!-- Spring MVC の機能と連携して出力された CSRF トークン値の hidden 項目 -->
    <div>
        <input type="hidden"
               name="_csrf" value="63845086-6b57-4261-8440-97a3c6fa6b99" />
    </div>
</form>
```

ちなみに: 出力される CSRF トークンチェック値

Spring 4 上で CsrfRequestDataValueProcessor を使用すると、<form:form>タグの method 属性に指定した値が CSRF トークンチェック対象の HTTP メソッド (Spring Security のデフォルト実装では GET,HEAD,TRACE,OPTIONS 以外の HTTP メソッド) と一致する場合に限り、CSRF トークンが埋め込まれた<input type="hidden">タグが出力される。

例えば、以下の例のように method 属性に GET メソッドを指定した場合は、CSRF トークンが埋め込まれた<input type="hidden">タグは出力されない。

```
<form:form method="GET" modelAttribute="xxxForm" action="...">
    <%-- ... --%>
</form:form>
```

これは、Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet で説明されている内容に対応している事を意味しており、セキュアな Web アプリケーション構築の手助けとなる。

HTML フォーム使用時の連携

Spring MVC と連携 せずに、HTML フォームを使用して CSRF トークン値を連携することも可能である。HTML フォームを使ってリクエストを送信する場合は、HTML フォームの hidden 項目として CSRF トークン値を出力し、リクエストパラメータとして連携する。

- JSP の実装例

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<form action="<c:url value="/login" />" method="post">
    <!-- omitted -->
    <sec:csrfInput /> <!-- (1) -->
    <!-- omitted -->
</form>
```

項番	説明
(1)	HTML の<form>要素の中に<sec:csrfInput>要素を指定する。

Spring Security から提供されている<sec:csrfInput>要素を指定すると、以下のような hidden 項目が出力される。HTML フォーム内に hidden 項目を出力することで、CSRF トークン値がリクエストパラメータとして連携される。デフォルトでは、CSRF トークン値を連携するためのリクエストパラメータ名は_csrf になる。

- HTML の出力例

```
<form action="/login" method="post">
    <!-- omitted -->
    <!-- CSRF トークン値の hidden 項目 -->
    <input type="hidden"
           name="_csrf"
           value="63845086-6b57-4261-8440-97a3c6fa6b99" />
    <!-- omitted -->
</form>
```

警告: GET メソッド使用時の注意点

HTTP メソッドとして GET を使用する場合、<sec:csrfInput>要素を指定しないこと。<sec:csrfInput>要素を指定してしまうと、URL に CSRF トークン値が含まれてしまうため、CSRF トークン値が盗まれるリスクが高くなる。

Ajax 使用時の連携

Ajax を使ってリクエストを送信する場合は、HTML の meta タグとして CSRF トークンの情報を出力し、meta タグから取得したトークン値を Ajax 通信時のリクエストヘッダに設定して連携する。

まず、Spring Security から提供されている JSP タグライブラリを使用して、HTML の meta タグに CSRF トークンの情報を出力する。

- JSP の実装例

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<head>
    <!-- omitted -->
    <sec:csrfMetaTags /> <!-- (1) -->
    <!-- omitted -->
</head>
```

項番	説明
(1)	HTML の<head>要素内に<sec:csrfMetaTags>要素を指定する。

<sec:csrfMetaTags>要素を指定すると、以下のような meta タグが出力される。デフォルトでは、CSRF トークン値を連携するためのリクエストヘッダ名は X-CSRF-TOKEN となる。

- HTML の出力例

```
<head>
    <!-- omitted -->
    <meta name="_csrf_parameter" content="_csrf" />
    <meta name="_csrf_header" content="X-CSRF-TOKEN" /> <!-- ヘッダ名 -->
    <meta name="_csrf"
          content="63845086-6b57-4261-8440-97a3c6fa6b99" /> <!-- トークン値 -->
    <!-- omitted -->
</head>
```

つぎに、JavaScript を使って meta タグから CSRF トークンの情報を取得し、Ajax 通信時のリクエストヘッダに CSRF トークン値を設定する。(ここでは jQuery を使った実装例となっている)

- JavaScript の実装例

```
$(function () {
    var headerName = $("meta[name='_csrf_header']").attr("content"); // (1)
    var tokenValue = $("meta[name='_csrf']").attr("content"); // (2)
    $(document).ajaxSend(function(e, xhr, options) {
        xhr.setRequestHeader(headerName, tokenValue); // (3)
    });
});
```

項番	説明
(1)	CSRF トークン値を連携するためのリクエストヘッダ名を取得する。
(2)	CSRF トークン値を取得する。
(3)	リクエストヘッダに CSRF トークン値を設定する。

トークンチェックエラー時の遷移先の制御

トークンチェックエラー時の遷移先の制御を行うためには、CSRF トークンチェックエラーに発生する例外である `AccessDeniedException` をハンドリングして、その例外に対応した遷移先を指定する。

CSRF のトークンチェックエラー時に発生する例外は以下の通りである。

表 9.38　**CSRF トークンチェックで使用される例外クラス**

クラス名	説明
`InvalidCsrfTokenException`	クライアントから送られたトークン値と、サーバー側で保持しているトークン値が一致しない場合に使用する例外クラス（主に不正なリクエスト）。
`MissingCsrfTokenException`	サーバー側にトークン値が保存されていない場合に使用する例外クラス（主にセッション切れ）。

`DelegatingAccessDeniedHandler` クラスを使用して上記の例外をハンドリングし、それぞれに `AccessDeniedHandler` インタフェースの実装クラスを割り当てることで、例外毎の遷移先を設定することが可能である。

CSRF トークンチェックエラー時に専用のエラー画面（JSP）に遷移させたい場合は、以下のような Bean 定義を行う。(以下の定義例は、ブランクプロジェクトからの抜粋である)

- spring-security.xml の定義例

```
<sec:http>
    <!-- omitted -->
    <sec:access-denied-handler ref="accessDeniedHandler"/>  <!-- (1) -->
    <!-- omitted -->
```

```
</sec:http>

<bean id="accessDeniedHandler"
    class="org.springframework.security.web.access.DelegatingAccessDeniedHandler">
 <!-- (2) -->
    <constructor-arg index="0">  <!-- (3) -->
        <map>
            <!-- (4) -->
            <entry
                key="org.springframework.security.web.csrf.
↪InvalidCsrfTokenException">
                <bean
                    class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                    <property name="errorPage"
                        value="/WEB-INF/views/common/error/invalidCsrfTokenError.
↪jsp" />
                </bean>
            </entry>
            <!-- (5) -->
            <entry
                key="org.springframework.security.web.csrf.
↪MissingCsrfTokenException">
                <bean
                    class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                    <property name="errorPage"
                        value="/WEB-INF/views/common/error/missingCsrfTokenError.
↪jsp" />
                </bean>
            </entry>
        </map>
    </constructor-arg>
    <!-- (6) -->
    <constructor-arg index="1">
        <bean
            class="org.springframework.security.web.access.AccessDeniedHandlerImpl
↪">
            <property name="errorPage"
                value="/WEB-INF/views/common/error/accessDeniedError.jsp" />
        </bean>
    </constructor-arg>
</bean>
```

項番	説明
(1)	<sec:access-denied-handler>タグの ref 属性に、Exception 毎の制御を行うための AccessDeniedHandler の Bean 名を指定する。 エラー時遷移先が全て同じ画面である場合は error-page 属性に遷移先を指定すればよい。 <sec:access-denied-handler>でハンドリングしない場合は、認可エラー時の遷移先を参照されたい。
(2)	DelegatingAccessDeniedHandler を使用して、発生した例外（AccessDeniedException サブクラス） と例外ハンドラ（AccessDeniedHandler 実装クラス）を定義する。
(3)	コンストラクタの第 1 引数で、個別に遷移先を指定したい例外（AccessDeniedException サブクラス）と、対応する例外ハンドラ（AccessDeniedHandler 実装クラス）を Map 形式で定義する。
(4)	key に AccessDeniedException のサブクラスを指定する。 value として、AccessDeniedHandler の実装クラスである、org.springframework.security.web.access.AccessDeniedHandlerImpl を指定する。 property の name に errorPage を指定し、value に表示する view を指定する。 マッピングする Exception に関しては、トークンチェックエラー時の遷移先の制御 を参照されたい。
(5)	(4) の Exception と異なる Exception を制御したい場合に定義する。 本例では InvalidCsrfTokenException、MissingCsrfTokenException それぞれに異なる遷移先を設定している。
(6)	コンストラクタの第 2 引数で、デフォルト例外（(4)(5) で指定していない AccessDeniedException のサブクラス）時の例外ハンドラ（AccessDeniedHandler 実装クラス）と遷移先を指定する。

注釈: 無効なセッションを使ったリクエストの検知

セッション管理機能の「無効なセッションを使ったリクエストの検知」処理を有効にしている場合は、`MissingCsrfTokenException` に対して「無効なセッションを使ったリクエストの検知」処理と連動する `AccessDeniedHandler` インタフェースの実装クラスが適用される。

そのため、`MissingCsrfTokenException` が発生すると、「無効なセッションを使ったリクエストの検知」処理を有効化する際に指定したパス (`invalid-session-url`) にリダイレクトする。

注釈: ステータスコード 403 以外を返却したい場合

リクエストに含まれる CSRF トークンが一致しない場合に、ステータスコード 403 以外を返却したい場合は、`org.springframework.security.web.access.AccessDeniedHandler` インタフェースを実装した、独自の AccessDeniedHandler を作成する必要がある。

9.6 ブラウザのセキュリティ対策機能との連携

9.6.1 Overview

本節では、ブラウザが提供しているセキュリティ対策機能との連携方法について説明する。

主要な Web ブラウザは、ブラウザが提供する機能が悪用されないようにするために、いくつかのセキュリティ対策機能を提供している。ブラウザが提供するセキュリティ対策機能の一部は、サーバ側で HTTP のレスポンスヘッダを出力することで動作を制御することができる。

Spring Security は、セキュリティ関連のレスポンスヘッダを出力する機能を用意することで、Web アプリケーションのセキュリティを強化する仕組みを提供している。

注釈: セキュリティリスク

セキュリティ関連のレスポンスヘッダを出力しても、セキュリティへのリスクが 100% なくなるわけではない。あくまで、セキュリティリスクを減らすためのサポート機能と考えておくこと。

なお、セキュリティヘッダのサポート状況はブラウザによってことなる。

注釈: HTTP ヘッダの上書き

後述の設定を行ったとしても、アプリケーションにより、HTTP ヘッダが上書きされる可能性は存在する。

デフォルトでサポートしているセキュリティヘッダ

Spring Security がデフォルトでサポートしているレスポンスヘッダは以下の 8 つである。

- Cache-Control (Pragma, Expires)
- X-Frame-Options
- X-Content-Type-Options
- X-XSS-Protection
- Strict-Transport-Security
- Content-Security-Policy(Content-Security-Policy-Report-Only)
- Public-Key-Pins(Public-Key-Pins-Report-Only)
- Referrer-Policy

ちなみに：ブラウザのサポート状況

これらのヘッダに対する処理は、一部のブラウザではサポートされていない。ブラウザの公式サイトまたは以下のページを参照されたい。

- https://www.owasp.org/index.php/HTTP_Strict_Transport_Security_Cheat_Sheet (Strict-Transport-Security)
- https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet (X-Frame-Options)
- https://www.owasp.org/index.php/OWASP_Secure_Headers_Project#tab=Headers (X-Content-Type-Options, X-XSS-Protection, Content-Security-Policy, Public-Key-Pins)

注釈：**Referrer-Policy** ヘッダ

Spring Security 4.2 より、ブラウザに Referrer Policy を指示するためのヘッダである Referrer-Policy ヘッダがサポートされた。詳細については次版以降の開発ガイドラインで記載する予定である。

Cache-Control

Cache-Control ヘッダは、コンテンツのキャッシュ方法を指示するためのヘッダである。保護されたコンテンツがブラウザにキャッシュされないようにすることで、権限のないユーザーが保護されたコンテンツを閲覧できてしまうリスクを減らすことができる。

コンテンツがキャッシュされないようにするためには、以下のようなヘッダを出力する。

- レスポンスヘッダの出力例

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
```

注釈: **Cache-Control ヘッダの上書き**

Spring MVC の Controller クラスが `@SessionAttributes` のフォームクラスを定義している、もしくは、リクエストハンドラで `@SessionAttributes` 属性の Model を使用している場合は、 Cache-Control ヘッダが上書きされる。

注釈: **HTTP1.0 互換のブラウザ**

Spring Security は HTTP1.0 互換のブラウザもサポートするために、Pragma ヘッダと Expires ヘッダも出力する。

X-Frame-Options

X-Frame-Options ヘッダは、フレーム (`<frame>`または`<iframe>`要素) 内でのコンテンツの表示を許可するか否かを指示するためのヘッダである。フレーム内でコンテンツが表示されないようすることで、クリックジャッキングと呼ばれる攻撃手法を使って機密情報を盗みとられるリスクをなくすことができる。

フレーム内での表示を拒否するためには、以下のようなヘッダを出力する。

- レスポンスヘッダの出力例 (Spring Security のデフォルト出力)

```
X-Frame-Options: DENY
```

なお、X-Frame-Options ヘッダには、出力例以外のオプションを指定することができる。

X-Content-Type-Options

X-Content-Type-Options ヘッダは、コンテンツの種類の決定方法を指示するためのヘッダである。一部のブラウザでは、Content-Type ヘッダの値を無視してコンテンツの内容をみて決定する。コンテンツの種類の決定する際にコンテンツの内容を見ないようにすることで、クロスサイトスクリプティングを使った攻撃を受けるリスクを減らすことができる。

コンテンツの種類の決定する際にコンテンツの内容を見ないようにするためには、以下のヘッダを出力する。

- レスポンスヘッダの出力例

```
X-Content-Type-Options: nosniff
```

X-XSS-Protection

X-XSS-Protection ヘッダは、ブラウザの XSS フィルター機能を使って有害スクリプトを検出する方法を指示するためのヘッダである。XSS フィルター機能を有効にして有害なスクリプトを検知するとこで、クロスサイトスクリプティングを使った攻撃を受けるリスクを減らすことができる。

XSS フィルター機能を有効にして有害なスクリプトを検知するためには、以下のようなヘッダを出力する。

- レスポンスヘッダの出力例 (Spring Security のデフォルト出力)

```
X-XSS-Protection: 1; mode=block
```

なお、X-XSS-Protection ヘッダには、出力例以外のオプションを指定することができる。

Strict-Transport-Security

Strict-Transport-Security ヘッダーは、HTTPS を使ってアクセスした後に HTTP を使ってアクセスしようとした際に、HTTPS に置き換えてからアクセスすることを指示するためヘッダである。HTTPS でアクセスした後に HTTP が使われないようにすることで、中間者攻撃と呼ばれる攻撃手法を使って悪意のあるサイトに誘導されるリスクを減らすことができる。

HTTPS でアクセスした後に HTTP が使われないようにするためには、以下のようなヘッダを出力する。

- レスポンスヘッダの出力例 (Spring Security のデフォルト出力)

```
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
```

注釈: Strict-Transport-Security

Spring Security のデフォルト実装では、Strict-Transport-Security ヘッダは、アプリケーションサーバに対して HTTPS を使ってアクセスがあった場合のみ出力される。なお、Strict-Transport-Security ヘッダ値は、オプションを指定することで変更することができる。

Content-Security-Policy

Content-Security-Policy ヘッダーはブラウザに読み込みを許可するコンテンツを指示するためのヘッダーである。ブラウザは Content-Security-Policy ヘッダーに指定したホワイトリストのコンテンツのみを読み込むため、悪意のあるコンテンツを読み込むことで実行される攻撃（クロスサイトスクリプティング攻撃など）を受けるリスクを減らすことができる。

Content-Security-Policy ヘッダーを送信しない場合、ブラウザは標準の同一オリジンポリシーを適用する。

コンテンツの取得元を同一オリジンのみに制限するためには、以下のようなヘッダーを出力する。

- レスポンスヘッダの出力例

```
Content-Security-Policy: default-src 'self'
```

注釈: ポリシー違反時のレポート送信について

ポリシー違反時にレポートを送信したい場合、report-uri ディレクティブに報告先の URI を指定する。

同一オリジンポリシー違反があった場合にコンテンツをブロックして`/csp_report` にレポートを送信するためには、以下のようなヘッダーを出力する。

- レスポンスヘッダの出力例

```
Content-Security-Policy: default-src 'self'; report-uri /csp_report;
```

また、ポリシー違反があった際に、コンテンツのブロックを行わずレポートの送信のみを行いたい場合は Content-Security-Policy-Report-Only ヘッダーを使用する。Content-Security-Policy-Report-Only ヘッダーを使用してレポートを収集しながら段階的にポリシーとコンテンツを修正することで、既にサービス提供しているサイトに対してポリシーを適用した場合に正常に動作しなくなるリスクを減らすことが出来る。

同一オリジンポリシー違反があった場合にコンテンツをブロックせず/csp_report にレポートを送信するためには、以下のようなヘッダーを出力する。

- レスポンスヘッダの出力例

```
Content-Security-Policy-Report-Only: default-src 'self'; report-uri /csp_
↪report;
```

Public-Key-Pins

Public-Key-Pins ヘッダはサイトの証明書の真正性を担保するために、サイトに紐づく証明書の公開鍵をブラウザに提示するヘッダである。サイトへの再訪問時に中間者攻撃と呼ばれる攻撃手法を使って悪意のあるサイトに誘導された場合でも、ブラウザが保持する真性のサイト証明書の公開鍵と悪意あるサイトが提示する証明書の公開鍵の不一致を検知して、アクセスをブロックすることができる。

ブラウザが保持する情報と一致しない証明書を検出した場合にアクセスをブロックさせるためには、以下のようなヘッダを出力する。

- レスポンスヘッダの出力例

```
Public-Key-Pins: max-age=5184000 ; pin-sha256=
↪"d6qzRu9zOECb90Uez27xWltNsj0e1Md7GkYYkVoZWmM=" ; pin-sha256=
↪"E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g="
```

注釈: 違反レポートの送信について

アクセスブロック時にブラウザに違反レポートを送信させるためには、Content-Security-Policy と同様に report-uri ディレクティブを指定する。

また、ブラウザにアクセスをブロックさせずに違反レポートを送信させるためには、Public-Key-Pins ヘッダの代わりに Public-Key-Pins-Report-Only ヘッダを使用する。

注釈: Public-Key-Pins ヘッダの設定について

Public-Key-Pins ヘッダの設定に誤りがあった場合、ユーザが長期間サイトにアクセスできなくなる可能性があるため、Public-Key-Pins-Report-Only ヘッダで十分に試験を実施した上で Public-Key-Pins ヘッダに切り替えることを推奨する。

9.6.2 How to use

セキュリティヘッダ出力機能の適用

前述のセキュリティヘッダ出力機能を適用する方法を説明する。

セキュリティヘッダ出力機能は、Spring 3.2 から追加された機能であり、以下のセキュリティヘッダがデフォルトで適用されるようになっている。

- Cache-Control (Pragma, Expires)
- X-Frame-Options
- X-Content-Type-Options
- X-XSS-Protection
- Strict-Transport-Security

そのため、デフォルトで適用されるセキュリティヘッダ出力機能を有効にするための特別な定義は不要である。なお、デフォルトで適用されるセキュリティヘッダ出力機能を適用したくない場合は、明示的に無効化する必要がある。

セキュリティヘッダ出力機能を無効化する場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<sec:http>
    <!-- omitted -->
    <sec:headers disabled="true"/> <!-- disabled属性にtrueを設定して無効化 -->
    <!-- omitted -->
</sec:http>
```

セキュリティヘッダの選択

出力するセキュリティヘッダを選択したい場合は、以下のような bean 定義を行う。ここでは Spring Security が提供しているすべてのセキュリティヘッダを出力する例になっているが、実際には必要なものだけ指定すること。

- spring-security.xml の定義例

```
<sec:headers defaults-disabled="true"> <!-- (1) -->
    <sec:cache-control/> <!-- (2) -->
    <sec:frame-options/> <!-- (3) -->
    <sec:content-type-options/> <!-- (4) -->
    <sec:xss-protection/> <!-- (5) -->
    <sec:hsts/> <!-- (6) -->
    <sec:content-security-policy policy-directives="default-src 'self'" /> <!--␣
↪(7) -->
    <sec:hpkp report-uri="https://www.example.net/hpkp-report"> <!-- (8) -->
        <sec:pins>
            <sec:pin algorithm="sha256">
↪d6qzRu9zOECb90Uez27xWltNsj0e1Md7GkYYkVoZWmM=</sec:pin>
            <sec:pin algorithm="sha256">
↪E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=</sec:pin>
```

```
        </sec:pins>
    </sec:hpkp>
</sec:headers>
```

項番	説明
(1)	まずデフォルトで適用されるヘッダ出力を行うコンポーネント登録を無効化する。
(2)	Cache-Control(Pragma, Expires) ヘッダを出力するコンポーネントを登録する。
(3)	Frame-Options ヘッダを出力するコンポーネントを登録する。
(4)	X-Content-Type-Options ヘッダを出力するコンポーネントを登録する。
(5)	X-XSS-Protection ヘッダを出力するコンポーネントを登録する。
(6)	Strict-Transport-Security ヘッダを出力するコンポーネントを登録する。
(7)	Content-Security-Policy ヘッダまたは Content-Security-Policy-Report-Only ヘッダを出力するコンポーネントを登録する。
(8)	Public-Key-Pins ヘッダまたは Public-Key-Pins-Report-Only ヘッダを出力するコンポーネントを登録する。 • サイトの提示する証明書の公開鍵が一致しなかった場合、アクセスをブロックせず `https://www.example.net/hpkp-report` に違反レポートの送信を行う。 • 証明書の危殆化や期限切れなどの理由で証明書を更新した際に公開鍵の不一致が発生しないようにするために、バックアップ用の公開鍵の情報も設定している。

注釈: **Public-Key-Pins** ヘッダの出力について

Spring Security のデフォルトの設定では、Public-Key-Pins ヘッダではなく、Public-Key-Pins-Report-Only ヘッダが出力される。

また、Spring Security のデフォルト実装では、Public-Key-Pins ヘッダは、アプリケーションサーバに対して HTTPS を使ってアクセスがあった場合のみ出力される。

また、不要なものだけ無効化する方法も存在する。

- spring-security.xml の定義例

```
<sec:headers>
    <sec:cache-control disabled="true"/> <!-- disabled 属性に true を設定して無効化 -->
</sec:headers>
```

上記の例だと、Cache-Control 関連のヘッダだけが出力されなくなる。

セキュリティヘッダの詳細については公式リファレンスを参照されたい。

注釈: **Spring Security** によるセキュリティヘッダ付与の仕様変更

TERASOLUNA Server Framework for Java 5.4.1 の依存ライブラリである Spring Security 4.2.4 では、Spring Security によって先にセキュリティヘッダが付与されることにより Controller 等で任意に付与したヘッダが有効にならないことがあった。例えば、Controller で個別にキャッシュ制御のヘッダを付与した場合でも Spring Security が先に付与した `Pragma: no-cache` ヘッダが残ることにより意図したキャッシュ制御ができないといった問題があった。

このため、Spring Security 4.2.5 及び 5.0.2 以降ではレスポンスコミットのタイミングでセキュリティヘッダを付与するように変更 (spring-projects/spring-security/issues/#5004) されている。

警告: 個別に付与したセキュリティヘッダが Spring Security により上書き（追加）される問題

`DispatcherServlet` 内の処理で付与したセキュリティヘッダが Spring Security の `HeaderWriter` により上書き（追加）される問題 (spring-projects/spring-security/issues/#5193) が報告されている。Spring Security でデフォルトのセキュリティヘッダを付与するが、一部のユースケースのみ Controller 等で個別にセキュリティヘッダを付与したい場合は、この問題の影響を受けることになる。ただし、CacheControlHeaderWriter は既に付与されているヘッダを優先する実装となっているため、キャッシュ制御に関するヘッダ（Cache-Control, Pragma, Expires）ではこの問題は発生しない。

セキュリティヘッダのオプション指定

以下のヘッダでは、Spring Security がデフォルトで出力する内容を変更することができる。

- X-Frame-Options

- X-XSS-Protection
- Strict-Transport-Security
- Content-Security-Policy(Content-Security-Policy-Report-Only)
- Public-Key-Pins(Public-Key-Pins-Report-Only)
- Referrer-Policy

Spring Security の bean 定義を変更することで、各要素の属性にオプション[*1]を指定することができる。

- spring-security.xml の定義例

```
<sec:frame-options policy="SAMEORIGIN" />
```

カスタムヘッダの出力

Spring Security がデフォルトで用意していないヘッダを出力することもできる。

以下のヘッダを出力するケースの例を説明する。

```
X-WebKit-CSP: default-src 'self'
```

上記のヘッダを出力する場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

```
<sec:headers>
    <sec:header name="X-WebKit-CSP" value="default-src 'self'"/>
</sec:headers>
```

項番	説明
(1)	<sec:headers>要素の子要素として<sec:header> を追加し、name 属性にヘッダ名を value 属性にヘッダ値を指定する。

リクエストパターン毎のセキュリティヘッダの出力

Spring Security は、RequestMatcher インタフェースの仕組みを利用して、リクエストのパターン毎にセキュリティヘッダの出力を制御することも可能である。

例えば、保護対象のコンテンツが/secure/というパスの配下に格納されていて、保護対象のコンテンツへアクセスした時だけ Cache-Control ヘッダを出力する場合は、以下のような bean 定義を行う。

- spring-security.xml の定義例

[*1] 各要素で指定できるオプションは https://docs.spring.io/spring-security/site/docs/4.2.12.RELEASE/reference/htmlsingle/#nsa-headers を参照されたい。

```
<!-- (1) -->
<bean id="secureCacheControlHeadersWriter"
      class="org.springframework.security.web.header.writers.
↪DelegatingRequestMatcherHeaderWriter">
    <constructor-arg>
        <bean class="org.springframework.security.web.util.matcher.
↪AntPathRequestMatcher">
            <constructor-arg value="/secure/**"/>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class="org.springframework.security.web.header.writers.
↪CacheControlHeadersWriter"/>
    </constructor-arg>
</bean>

<sec:http>
    <!-- omitted -->
    <sec:headers>
        <sec:header ref="secureCacheControlHeadersWriter"/> <!-- (2) -->
    </sec:headers>
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	RequestMatcher と HeadersWriter インタフェースの実装クラスを指定して DelegatingRequestMatcherHeaderWriter クラスの bean を定義する。
(2)	<sec:headers>要素の子要素として<sec:header> を追加し、ref 属性に (1) で定義した HeaderWriter の bean を指定する。

警告: 指定したパスが意図した通りに認識されない問題

<sec:http>と DelegatingRequestMatcherHeaderWriter がパスマッチングを行うタイミングの違いにより、指定したパスが意図した通りに認識されない場合がある。具体的には、DelegatingRequestMatcherHeaderWriter に指定されたパスはセキュリティヘッダ書き込み時（レスポンスのコミット時）にリクエストパスとマッチングされる。このため、リクエストのフォワードやインクルードによりリクエストパスが変更された場合、当初リクエストのパスとマッチングが行われないため、意図したパスでセキュリティヘッダが出力されなくなる。

特に Tiles を利用している場合は、Tiles の処理によりテンプレート JSP にフォワードされるため、DelegatingRequestMatcherHeaderWriter との併用ができないことが確認されている。

詳細は https://github.com/spring-projects/spring-security/issues/6338 を参照されたい。

9.7 XSS 対策

9.7.1 Overview

クロスサイトスクリプティング (以下、XSS と略す) について説明する。クロスサイトスクリプティングとは、アプリケーションのセキュリティ上の不備を意図的に利用し、サイト間を横断して悪意のあるスクリプトを混入させることである。例えば、ウェブアプリケーションが入力したデータ（フォーム入力など）を、適切にエスケープしないまま、HTML 上に出力することにより、入力値に存在するタグなどの文字が、そのまま HTML として解釈される。悪意のある値が入力された状態で、スクリプトを起動させることにより、クッキーの改ざんや、クッキーの値を取得することによる、セッションハイジャックなどの攻撃が行えてしまう。

Stored, Reflected XSS Attacks

XSS 攻撃は、大きく二つのカテゴリに分けられる。

Stored XSS Attacks

Stored XSS Attacks とは、悪意のあるコードが、永久的にターゲットサーバ上 (データベース等) に格納されていることである。ユーザーは、格納されている情報を要求するときに、サーバから悪意のあるスクリプトを取得し、実行してしまう。

Reflected XSS Attacks

Reflected attacks とは、リクエストの一部としてサーバに送信された悪意のあるコードが、エラーメッセージ、検索結果、その他いろいろなレスポンスからリフレクションされることである。ユーザーが、悪意のあるリンクをクリックするか、特別に細工されたフォームを送信すると、挿入されたコードは、ユーザーのブラウザに、攻撃を反映した結果を返却する。その結果、信頼できるサーバからきた値のため、ブラウザは悪意のあるコードを実行してしまう。

Stored XSS Attacks、Reflected XSS Attacks ともに、出力値をエスケープすることで防ぐことができる。

How to use

ユーザーの入力を、そのまま出力している場合、XSS の脆弱性にさらされている。したがって、XSS の脆弱性に対する対抗措置として、HTML のマークアップ言語で、特定の意味を持つ文字をエスケープする必要がある。

必要に応じて、3 種類のエスケープを使い分けること。

エスケープの種類:

- Output Escaping
- JavaScript Escaping
- Event handler Escaping

Output Escaping

XSS の脆弱性への対応としては、HTML 特殊文字をエスケープすることが基本である。HTML においてエスケープが必要な特殊文字の例と、エスケープ後の例は、以下の通りである。

エスケープ前	エスケープ後
「&」	&
「<」	<
「>」	>
「"」	"
「'」	'

XSS を防ぐために、文字列として出力するすべての表示項目に、`f:h()` を使用すること。入力値を、別画面に再出力するアプリケーションを例に、説明する。

出力値をエスケープしない脆弱性のある例

本例は、あくまで参考例として載せているだけなので、以下のような実装は、決して行わないこと。

出力画面の実装

```
<!-- omitted -->
<tr>
    <td>Job</td>
    <td>${customerForm.job}</td>  <!-- (1) -->
</tr>
<!-- omitted -->
```

項番	説明
(1)	customerForm のフィールドである、job をエスケープせず出力している。

入力画面の Job フィールドに、<script>タグを入力する。

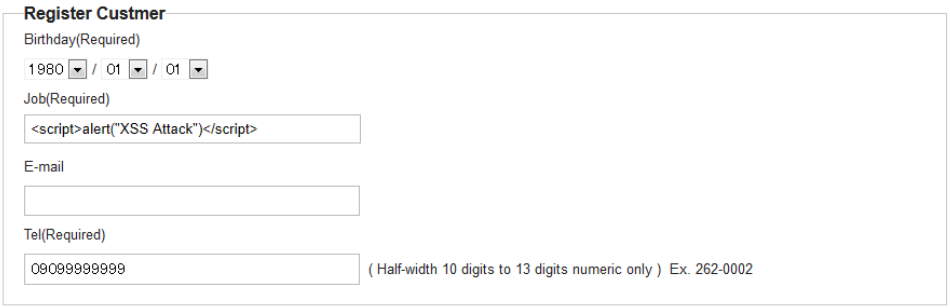


図 9.10　**Picture - Input HTML Tag**

<script>タグとして認識され、ダイアログボックスが表示されてしまう。

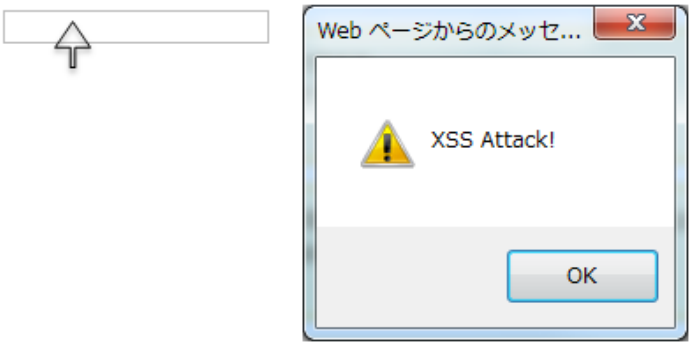


図 9.11　**Picture - No Escape Result**

出力値を **f:h()** 関数でエスケープする例

出力画面の実装

```
<!-- omitted -->
<tr>
    <td>Job</td>
    <td>${f:h(customerForm.job)}</td>  <!-- (1) -->
</tr>
.<!-- omitted -->
```

項番	説明
(1)	EL 式の `f:h()` を使用することにより、エスケープして出力している。

入力画面の Job フィールドに<script>タグを入力する。

特殊文字がエスケープされることにより、 <script>タグとして認識されず、入力値がそのまま出力される。

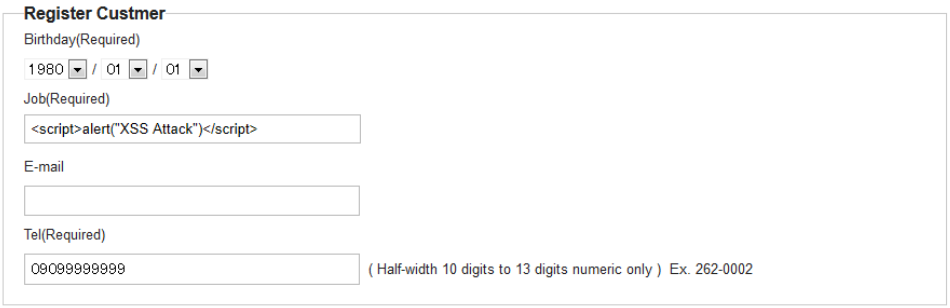


図 9.12　**Picture - Input HTML Tag**

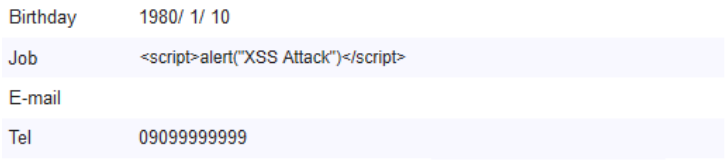


図 9.13　**Picture - Escape Result**

出力結果

```
<!-- omitted -->
<tr>
    <td>Job</td>
    <td>&lt;script&gt;alert(&quot;XSS Attack&quot;)&lt;/script&gt;</td>
</tr>
<!-- omitted -->
```

ちなみに: java.util.Date 継承クラスのフォーマット

java.util.Date 継承クラスをフォーマットして表示する場合は、JSTL の`<fmt:formatDate>`を用いることを推奨する。以下に、設定例を示す。

```
<fmt:formatDate value="${form.date}" pattern="yyyyMMdd" />
```

value の値に前述した `f:h()` を使用して値を設定すると、String になってしまい、`javax.el.ELException` がスローされるため、そのまま`${form.date}`を使用している。しかし、yyyyMMdd にフォーマットするため、XSS の心配はない。

ちなみに: java.lang.Number 継承クラス、または java.lang.Number にパースできる文字列

java.lang.Number 継承クラスまたは java.lang.Number にパースできる文字列をフォーマットして表示する場合は、`<fmt:formatNumber>`を用いることを推奨する。以下に、設定例を示す。

```
<fmt:formatNumber value="${f:h(form.price)}" pattern="###,###" />
```

上記は、String でも問題ないので、`<fmt:formatNumber>`タグを使わなくなった場合に `f:h()` を付け忘

れることを予防するため、`f:h()` を明示的に使用している。

JavaScript Escaping

XSS の脆弱性への対応としては、JavaScript 特殊文字をエスケープすることが基本である。ユーザーからの入力をもとに、JavaScript の文字列リテラルを動的に生成する場合に、エスケープが必要となる。

JavaScript においてエスケープが必要な特殊文字の例と、エスケープ後の例は、以下のとおりである。

エスケープ前	エスケープ後
「'」	\'
「"」	\"
「\」	\\
「/」	\/
「<」	\x3c
「>」	\x3e
0x0D(復帰)	\r
0x0A(改行)	\n

出力値をエスケープしない脆弱性のある例

XSS 問題が発生する例を、以下に示す。

本例は、あくまで参考例として載せているだけなので、以下のような実装は、決して行わないこと。

```
<html>
  <script  type="text/javascript">
      var aaa = '${warnCode}';
      alert(aaa);
  </script>
</html>
```

属性名	値
warnCode	';alert('XSS Attack!');aaa='message

上記例のように、ユーザーの入力を導出元としてコードを出力するなど、JavaScript の要素を動的に生成する場合、意図せず文字列リテラルが閉じられ、XSS の脆弱性が生じる。

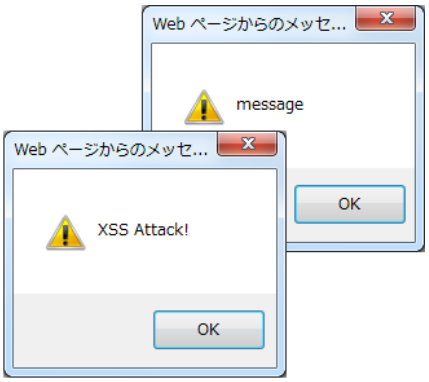


図 9.14　**Picture - No Escape Result**

出力結果

```
<script type="text/javascript">
    var aaa = '';alert('XSS Attack!');aaa='message';
    alert(aaa);
</script>
```

ちなみに： 業務要件上必要でない限り、JavaScript の要素をユーザーからの入力値に依存して動的に生成する仕様は、任意のスクリプトが埋め込まれてしまう可能性があるため、別の方式を検討する、または、極力避けるべきである。

出力値を f:js() 関数でエスケープする例

XSS を防ぐために、ユーザーの入力値、が設定される値に EL 式の関数、`f:js()` の使用を推奨する。

使用例を、下記に示す。

```
<script type="text/javascript">
    var aaa = '${f:js(warnCode)}';  // (1)
    alert(aaa);
</script>
```

項番	説明
(1)	EL 式の `f:js()` を使用することにより、エスケープして変数に設定している。

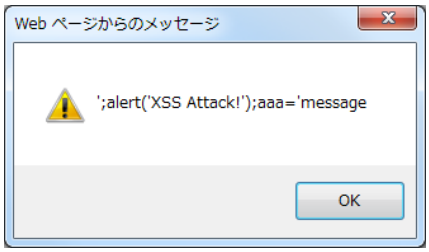


図 9.15 **Picture - Escape Result**

出力結果

```
<script  type="text/javascript">
    var aaa = '\';alert(\'XSS Attack!\');aaa=\'message';
    alert(aaa);
</script>
```

警告: スクリプトタグが含まれる値を、HTML エスケープせず `f:js()` でエスケープさせて出力する場合、document.write() を使用すると、ブラウザに HTML ソースとして解釈させるよう出力するので、XSS の脆弱性が生じる。以下に例を示すが、 このような実装は決して行わないこと。

JSP

```
<script  type="text/javascript">
   var aaa = '${f:js(warnCode)}';
   document.write(aaa);
</script>
```

属性名	値
warnCode	`<script>alert('XSS Attack!');</script>`

出力結果

```
<script  type="text/javascript">
   var aaa = '\x3cscript\x3ealert(\'XSS Attack!\');\x3c\/script\x3e';
   document.write(aaa);
</script>
```

出力結果をソースだけ確認するとエスケープできているように見える。しかし、これは<script>alert('XSS Attack!');</script> という内容の文字列を変数 aaa に格納するコードとなるため、document.write(aaa); と実装してしまうと、HTML のソースとして<script>alert('XSS Attack!');</script> を出力することになる。その結果、スクリプトが実行される。

ブラウザに値を出力させたい場合は、JavaScript を使用せず、HTML 特殊文字をエスケープする f:h() を使用することが望ましい。

JSP

```
${f:h(warnCode)}
```

出力結果

```
&lt;script&gt;alert(&#39;XSS Attack!&#39;);&lt;/script&gt;
```

あえて f:js() を使用し、document.write() で出力したい場合は、以下のいずれかのような、追加の XSS 対策が必要である。

- HTML エスケープ用の JavaScript 関数を用意し、document.write() の引数をエスケープする。
- f:h() でユーザーの入力値が設定される値を HTML エスケープした後、f:js() で JavaScript の文字列リテラル用のエスケープを行う。

Event handler Escaping

javascript のイベントハンドラの値をエスケープする場合、f:h() や、f:js() を使用するのではなく、f:hjs() を使用すること。${f:h(f:js())}と同義である。

理由としては、<input type="submit" onclick="callback('xxxx');">のようなイベントハンドラの値に');alert("XSS Attack");//を指定された場合、別のスクリプトを挿入できてしまうため、文字参照形式にエスケープ後、HTML エスケープを行う必要がある。

出力値をエスケープしない脆弱性のある例

XSS 問題が発生する例を、以下に示す。

```
<input type="text" onmouseover="alert('output is ${warnCode}') . ">
```

属性名	値
warnCode	'); alert('XSS Attack!'); // 上記の値が設定されてしまうことで、意図せず文字列リテラルが閉じられ、XSS の脆弱性が生じる。

マウスオーバ時、XSS のダイアログボックスが表示されてしまう。

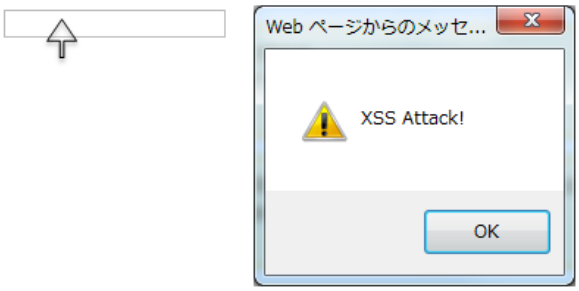


図 9.16　**Picture - No Escape Result**

出力結果

```
<!-- omitted -->
<input type="text" onmouseover="alert('output is'); alert('XSS Attack!'); // .') ">
<!-- omitted -->
```

出力値を f:hjs() 関数でエスケープする例

使用例を、下記に示す。

```
<input type="text" onmouseover="alert('output is ${f:hjs(warnCode)}') . ">  // (1)
```

項番	説明
(1)	EL 式の f:hjs() を使用することにより、エスケープして引数としている。

マウスオーバ時、XSS のダイアログは出力されない。

出力結果

```
<!-- omitted -->
<input type="text" onmouseover="alert('output is \&#39;); alert(\&#39;XSS Attack!\&
↪#39;);\&quot; \/\/ .') ">
<!-- omitted -->
```



図 9.17　**Picture - Escape Result**

9.8 暗号化

9.8.1 Overview

個人情報やパスワードなどの機密情報は、以下のようなケースで暗号化が求められる。

- インターネットなどのネットワークを介して機密情報の送受信を行う
- データベースやファイルなどの外部リソースに機密情報を保存する

Spring Security の主機能は「認証」と「認可」であるが、暗号化に関する機能も提供している。
ただし、提供される機能は限定的なものであるため、Spring Security がサポートしていない暗号化方式については、個別に実装する必要がある。

本ガイドラインでは、以下の処理について説明を行う。

- Spring Security が提供しているクラスを利用した共通鍵暗号化方式の暗号化と復号
- Spring Security が提供しているクラスを利用した疑似乱数の生成
- JCA (Java Cryptography Architecture) を利用した公開鍵暗号化方式の暗号化と復号
- JCA を利用したハイブリッド暗号化方式の暗号化と復号

Spring Security の暗号化機能の詳細については、Spring Security Reference -Spring Security Crypto Module-を参照されたい。

暗号化方式

暗号化方式について説明する。

共通鍵暗号化方式

暗号化と復号を行う際に同じ鍵を使用する方式である。
復号に使用する鍵を暗号化側へ共有しておく方式であるため、鍵を暗号化側へ安全に受け渡す経路が別途必要となる。

公開鍵暗号化方式

復号側が用意した公開鍵を使用して暗号化し、公開鍵とペアとなる秘密鍵を使用して復号する方式である。
暗号文を復号する際に使用する秘密鍵は公開されないためセキュリティの強度は高いが、暗号化と復号処理のコストは高い。

ハイブリッド暗号化方式

共通鍵暗号化方式の処理コストが低いという利点と、公開鍵暗号化方式の鍵の管理・配布が容易でセキュリティ強度が高いという利点の両方を組み合わせた方式である。
この方式は SSL/TLS などで利用されている。

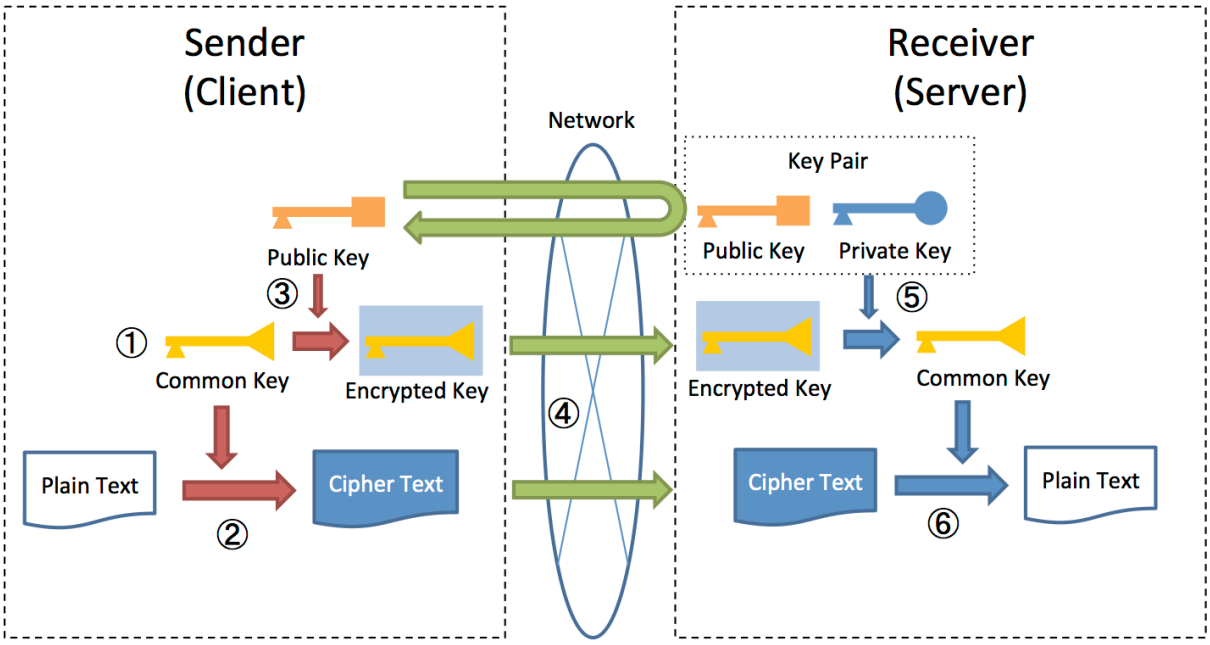
たとえば、HTTPS 通信では、クライアント側で生成した共通鍵をサーバ側の公開鍵で暗号化したうえで送信し、サーバ側は公開鍵とペアとなる秘密鍵を利用して共通鍵を復号する。その後の通信は、共有された共通鍵を使用した共通鍵暗号化方式で通信を行う。

この方式では、

- サイズが大きくなる可能性がある機密情報自体を、処理コストの低い共通鍵暗号化方式で暗号化
- サイズが小さく配布を安全に行う必要のある共通鍵を、セキュリティ強度の高い公開鍵暗号化方式で暗号化

するのがポイントである。機密情報を復号する際に使用する共通鍵は秘密鍵によって守られているため、公開鍵暗号化方式のセキュリティ強度を保ちつつ、公開鍵暗号化方式より高速な暗号化と復号処理を実現できる。

ハイブリッド暗号化方式における、暗号化から復号までの処理フローを以下の図に示す。



1. 送信側が平文を暗号化するための共通鍵を生成する。

2. 送信側が生成した共通鍵で平文を暗号化する。
3. 送信側が受信側の公開鍵で共通鍵を暗号化する。
4. 送信側が暗号化した共通鍵とともに暗号文を送信する。
5. 受信側が暗号化された共通鍵を受信側の秘密鍵で復号する。
6. 受信側が復号した共通鍵で暗号文を復号する。

暗号化アルゴリズム

暗号化アルゴリズムについて説明する。

DES / 3DES

DES (Data Encryption Standard) は共通暗号化方式のアルゴリズムとして、アメリカ合衆国の標準規格として規格化されたものである。鍵長が 56 ビットと短いため現在では推奨されていない。

3DES (トリプル DES) は、鍵を変えながら DES を繰り返す暗号化アルゴリズムである。

AES

AES (Advanced Encryption Standard) は共通鍵暗号化方式のアルゴリズムである。DES の後継として制定された暗号化規格であり、暗号化における現在のデファクトスタンダードとして利用されている。

また、ブロック長より長いメッセージを暗号化するメカニズムである暗号利用モードとして ECB (Electronic Codebook) 、CBC (Cipher Block Chaining) 、OFB (Output Feedback) など存在する。その中で、最も広く利用されているものは CBC である。

注釈: AES with GCM

GCM (Galois/Counter Mode) という、並列処理が可能であり CBC より処理効率が優れていると一般的にいわれている暗号利用モードを AES で利用することも可能である。

RSA

RSA は公開鍵暗号化方式のアルゴリズムである。素因数分解の困難性に基づいているため、計算機の能力向上により危殆化することとなる。いわゆる「暗号化アルゴリズムの 2010 年問題」として指摘されているように充分な鍵長が必要であり、現時点では 2048 ビットが標準的に利用されている。

DSA / ECDSA

DSA (Digital Signature Algorithm) は、デジタル署名のための標準規格である。離散対数問題の困難性に基づいている。

ECDSA (Elliptic Curve Digital Signature Algorithm : 楕円曲線 DSA) は、楕円曲線暗号を用いた DSA の変種である。楕円曲線暗号においては、セキュリティレベルを確保するために必要となる鍵長が短くなるというメリットがある。

疑似乱数 (生成器)

鍵の生成などで乱数が用いられる。

このとき、乱数として生成される値が予測可能だと暗号化の安全性が保てなくなるため、結果の予測が困難な乱数 (疑似乱数) を利用する必要がある。

疑似乱数の生成に用いられるのが疑似乱数生成器である。

javax.crypto.Cipher クラス

`Cipher` クラスは、暗号化および復号の機能を提供する。AES や RSA などの暗号化アルゴリズム、ECB や CBC などの暗号利用モード、PKCS1 などのパディング方式の組み合わせを指定する。

暗号利用モードとは、*AES* で説明したとおり、ブロック長より長いメッセージを暗号化するメカニズムである。

また、パディング方式とは、ブロック長に満たない暗号化対象を暗号化する場合の保管方式である。

Java アプリケーションでは、<**暗号化アルゴリズム**>/<**暗号利用モード**>/<**パディング方式**>または、<**暗号化アルゴリズム**>という形で組み合わせを指定する。たとえば、`AES/CBC/PKCS5Padding` または、`RSA` となる。詳細は、Cipher クラスの JavaDoc を参照されたい。

Spring Security における暗号化機能

Spring Security では、共通鍵暗号化方式を使用した暗号化および復号の機能を提供している。

暗号化アルゴリズムは 256-bit AES using PKCS #5’s PBKDF2 (Password-Based Key Derivation Function #2) である。

暗号利用モードは CBC、パディング方式は PKCS5Padding である。

暗号化・復号用のコンポーネント

Spring Security は、共通鍵暗号化方式での暗号化および復号の機能として以下のインターフェイスを提供している。

- `org.springframework.security.crypto.encrypt.TextEncryptor` (テキスト用)
- `org.springframework.security.crypto.encrypt.BytesEncryptor` (バイト配列用)

また、これらのインターフェイスの実装クラスとして以下のクラスを提供しており、内部では Cipher クラスを利用している。

- org.springframework.security.crypto.encrypt.HexEncodingTextEncryptor (テキスト用)
- org.springframework.security.crypto.encrypt.AesBytesEncryptor (バイト配列用)

乱数生成用のコンポーネント

Spring Security は、乱数 (鍵) 生成の機能として以下のインターフェイスを提供している。

- org.springframework.security.crypto.keygen.StringKeyGenerator (テキスト用)
- org.springframework.security.crypto.keygen.BytesKeyGenerator (バイト配列用)

また、これらのインターフェイスの実装クラスとして以下のクラスを提供している。

- org.springframework.security.crypto.keygen.HexEncodingStringKeyGenerator (テキスト用)
- org.springframework.security.crypto.keygen.SecureRandomBytesKeyGenerator (バイト配列用。generateKey メソッドで、異なる鍵長を生成して返却)
- org.springframework.security.crypto.keygen.SharedKeyGenerator (バイト配列用。generateKey メソッドで、コンストラクタで設定した同一の鍵長を返却)

注釈: Spring Security RSA

spring-security-rsa は、暗号化アルゴリズムとして RSA を使用した公開鍵暗号化方式とハイブリッド暗号化方式用の API を提供している。spring-security-rsa は現在、Spring の公式リポジトリ <https://github.com/spring-projects>_として管理されていない。今後、Spring の公式リポジトリ配下に移動した際は、本ガイドラインで利用方法を説明する予定である。

spring-security-rsa では以下 2 つのクラスを提供している。

- org.springframework.security.crypto.encrypt.RsaRawEncryptor

  公開鍵暗号化方式を使用した暗号化および復号の機能を提供するクラス。

- org.springframework.security.crypto.encrypt.RsaSecretEncryptor

  ハイブリッド暗号化方式を使用した暗号化および復号の機能を提供するクラス。

9.8.2 How to use

Oracle など、一部の Java 製品では AES の鍵長 256 ビットを扱うためには、強度が無制限の JCE 管轄ポリシーファイルを適用する必要がある。

注釈: JCE 管轄ポリシーファイル

輸入規制の関係上、一部の Java 製品ではデフォルトの暗号化アルゴリズム強度が制限されている。その為、より強力なアルゴリズムを利用する場合は、強度が無制限の JCE 管轄ポリシーファイルを入手し、JDK/JRE にインストールする必要があった。

しかし、JDK/JRE 7u161 および 8u151 以降からは、強度が無制限の JCE 管轄ポリシーファイルが予め含まれるようになったため、別途入手する必要はなくなった。また、JDK/JRE 7u171 および 8u161 以降では、当該ポリシーファイルがデフォルトで適用されているため、インストールも不要となった。

詳細については、Java Cryptography Architecture Oracle Providers Documentation を参照されたい。

JCE 管轄ポリシーファイルのダウンロード先

- Oracle Java 8 用
- Oracle Java 7 用

共通鍵暗号化方式

暗号化アルゴリズムとして AES を利用した方法について説明する。

文字列の暗号化

- テキスト（文字列）を暗号化する。

```
public static String encryptText(
    String secret, String salt, String plainText) {
    TextEncryptor encryptor = Encryptors.text(secret, salt); // (1)

    return encryptor.encrypt(plainText); // (2)
}
```

項番	説明
(1)	共通鍵とソルトを指定して `Encryptors#text` メソッドを呼び出し、`TextEncryptor` クラスのインスタンスを生成する。 生成したインスタンスの初期化ベクトルがランダムであるため、暗号化の際に異なる結果を返す。なお、暗号利用モードは CBC となる。 このときに指定した共通鍵とソルトは、復号時にも同じものを利用する。
(2)	平文を `encrypt` メソッドで暗号化する。

注釈: 暗号化の結果について

`encrypt` メソッドの返り値 (暗号化の結果) は実行毎に異なる値を返すが、鍵とソルトが同一であれば復号処理の結果は同一になる (正しく復号できる) 。

- 同一の暗号化結果を取得する。

この方法は、暗号化した結果を用いてデータベースの検索を行うようなケースで利用できる。ただし、セキュリティ強度が落ちる点を踏まえ、使用の可否を検討してほしい。

```
public static void encryptTextResult(
    String secret, String salt, String plainText) {
    TextEncryptor encryptor = Encryptors.queryableText(secret, salt); // (1)
    System.out.println(encryptor.encrypt(plainText)); // (2)
    System.out.println(encryptor.encrypt(plainText)); //
}
```

項番	説明
(1)	暗号化した結果として同じ値が必要な場合は、`Encryptors#queryableText` メソッドを利用して `TextEncryptor` クラスのインスタンスを生成する。
(2)	`Encryptors#queryableText` メソッドで生成したインスタンスは、`encrypt` メソッドでの暗号化の結果として同一の値を返す。

- GCM を用いた AES を使用してテキスト（文字列）を暗号化する。

  GCM を用いた AES は Spring Security4.0.2 以降で利用可能である。*AES* で説明したとおり、CBC より処理効率が良い。

```
public static String encryptTextByAesWithGcm(String secret, String salt,␣
↪String plainText) {
    TextEncryptor aesTextEncryptor = Encryptors.delux(secret, salt); // (1)

    return aesTextEncryptor.encrypt(plainText); // (2)
}
```

項番	説明
(1)	共通鍵とソルトを指定して `Encryptors#delux` メソッドを呼び出し、`TextEncryptor` クラスのインスタンスを生成する。 このときに指定する共通鍵とソルトは、復号時にも同じものを利用する。
(2)	平文を `encrypt` メソッドで暗号化する。

注釈: GCM を用いた AES に対する Java の対応状況

GCM を用いた AES は Java SE8 以降で使用可能である。詳細については、JDK 8 セキュリティの拡張機能を参照されたい。

文字列の復号

- テキスト（文字列）の暗号文を復号する。

```
public static String decryptText(String secret, String salt, String␣
↪cipherText) {
    TextEncryptor decryptor = Encryptors.text(secret, salt); // (1)

    return decryptor.decrypt(cipherText); // (2)
}
```

項番	説明
(1)	共通鍵とソルトを指定して Encryptors#text メソッドを呼び出し、TextEncryptor クラスのインスタンスを生成する。 共通鍵とソルトは、暗号化した際に利用したものを指定する。
(2)	暗号文を decrypt メソッドで復号する。

- GCM を用いた AES を使用してテキスト（文字列）の暗号文を復号する。

```
public static String decryptTextByAesWithGcm(String secret, String salt,␣
↪String cipherText) {
    TextEncryptor aesTextEncryptor = Encryptors.delux(secret, salt); // (1)

    return aesTextEncryptor.decrypt(cipherText); // (2)
}
```

項番	説明
(1)	共通鍵とソルトを指定して Encryptors#delux メソッドを呼び出し、TextEncryptor クラスのインスタンスを生成する。 共通鍵とソルトは、暗号化した際に利用したものを指定する。
(2)	暗号文を decrypt メソッドで復号する。

バイト配列の暗号化

- バイト配列を暗号化する。

```
public static byte[] encryptBytes(String secret, String salt, byte[]␣
↪plainBytes) {
    BytesEncryptor encryptor = Encryptors.standard(secret, salt); // (1)

    return encryptor.encrypt(plainBytes); // (2)
}
```

項番	説明
(1)	共通鍵とソルトを指定して Encryptors#standard メソッドを呼び出し、 BytesEncryptor クラスのインスタンスを生成する。 このときに指定した共通鍵とソルトは、復号時にも同じものを利用する。
(2)	バイト配列の平文を encrypt メソッドで暗号化する。

- GCM を用いた AES を使用してバイト配列を暗号化する。

```
public static byte[] encryptBytesByAesWithGcm(String secret, String salt,␣
↪byte[] plainBytes) {
    BytesEncryptor aesBytesEncryptor = Encryptors.stronger(secret, salt); //␣
↪(1)

    return aesBytesEncryptor.encrypt(plainBytes); // (2)
}
```

項番	説明
(1)	共通鍵とソルトを指定して Encryptors#stronger メソッドを呼び出し、 BytesEncryptor クラスのインスタンスを生成する。 このときに指定した共通鍵とソルトは、復号時にも同じものを利用する。
(2)	バイト配列の平文を encrypt メソッドで暗号化する。

バイト配列の復号

- バイト配列の暗号文を復号する。

```
public static byte[] decryptBytes(String secret, String salt, byte[]␣
↪cipherBytes) {
    BytesEncryptor decryptor = Encryptors.standard(secret, salt); // (1)

    return decryptor.decrypt(cipherBytes); // (2)
}
```

項番	説明
(1)	共通鍵とソルトを指定して Encryptors#standard メソッドを呼び出し、 BytesEncryptor クラスのインスタンスを生成する。 共通鍵とソルトは、暗号化した際に利用したものを指定する。
(2)	バイト配列の暗号文を decrypt メソッドで復号する。

- GCM を用いた AES によりバイト配列を復号する。

```
public static byte[] decryptBytesByAesWithGcm(String secret, String salt,␣
↪byte[] cipherBytes) {
    BytesEncryptor aesBytesEncryptor = Encryptors.stronger(secret, salt); //␣
↪(1)

    return aesBytesEncryptor.decrypt(cipherBytes); // (2)
}
```

項番	説明
(1)	共通鍵とソルトを指定して Encryptors#stronger メソッドを呼び出し、 BytesEncryptor クラスのインスタンスを生成する。 共通鍵とソルトは、暗号化した際に利用したものを指定する。
(2)	バイト配列の暗号文を decrypt メソッドで復号する。

公開鍵暗号化方式

Spring Security では公開鍵暗号化方式に関する機能は提供されていないため、JCA および OpenSSL を利用した方法についてサンプルコードを用いて説明する。

事前準備（JCA によるキーペアの生成）

- JCA でキーペア (公開鍵 / 秘密鍵の組み合わせ) を生成し、公開鍵で暗号化、秘密鍵で復号処理を行う。

```
public void generateKeysByJCA() {
    try {
        KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA"); //␣
↪(1)
        generator.initialize(2048); // (2)
        KeyPair keyPair = generator.generateKeyPair(); // (3)
        PublicKey publicKey = keyPair.getPublic();
        PrivateKey privateKey = keyPair.getPrivate();

        byte[] cipherBytes = encryptByPublicKey("Hello World!", publicKey);  /
↪/ (4)
```

```
        String plainText = decryptByPrivateKey(cipherBytes, privateKey); //␣
↪(5)
        System.out.println(plainText);
    } catch (NoSuchAlgorithmException e) {
        throw new SystemException("e.xx.xx.9002", "No Such setting error.",␣
↪e);
    }
}
```

項番	説明
(1)	RSA アルゴリズムを指定して KeyPairGenerator クラスのインスタンスを生成する。
(2)	鍵長として 2048 ビットを指定する。
(3)	キーペアを生成する。
(4)	公開鍵を利用して暗号化処理を行う。処理内容は後述する。
(5)	秘密鍵を利用して復号処理を行う。処理内容は後述する。

注釈: 暗号化したデータを文字列として扱いたい場合

外部システム連携等、暗号化したデータを文字列でやり取りしたい場合は、1 つの手段として Base64 エンコードが挙げられる。Java SE8 以降の場合は、Java 標準の `java.util.Base64` を使用する。それ以前の場合は、Spring Security の `org.springframework.security.crypto.codec.Base64` を使用する。

Base64 エンコードおよびデコードする方法を Java 標準の `java.util.Base64` を使用して説明する。

- Base64 エンコード

```
// omitted
byte[] cipherBytes = encryptByPublicKey("Hello World!", publicKey);  // 暗号
化処理
String cipherString = Base64.getEncoder().encodeToString(cipherBytes);  //␣
↪バイト配列の暗号文を文字列に変換
```

```
// omitted
```

- Base64 デコード

```
// omitted
byte[] cipherBytes = Base64.getDecoder().decode(cipherString); // 文字列の暗号文をバイト配列に変換
String plainText = decryptByPrivateKey(cipherBytes, privateKey); // 復号処理
// omitted
```

暗号化

- 公開鍵を利用して文字列を暗号化する。

```
public byte[] encryptByPublicKey(String plainText, PublicKey publicKey) {
    try {
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding"); // (1)
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);                        //␣
↪(2)
        return cipher.doFinal(plainText.getBytes(StandardCharsets.UTF_8)); //
    } catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
        throw new SystemException("e.xx.xx.9002", "No Such setting error.",␣
↪e);
    } catch (InvalidKeyException |
             IllegalBlockSizeException |
             BadPaddingException e) {
        throw new SystemException("e.xx.xx.9003", "Invalid setting error.",␣
↪e);
    }
}
```

項番	説明
(1)	暗号化アルゴリズム、暗号利用モード、パディング方式を指定して、Cipher クラスのインスタンスを生成する。
(2)	暗号化処理を実行する。

復号

- 秘密鍵を利用してバイト配列を復号する。

```
public String decryptByPrivateKey(byte[] cipherBytes, PrivateKey privateKey) {
    try {
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding"); // (1)
        cipher.init(Cipher.DECRYPT_MODE, privateKey);           // (2)
        byte[] plainBytes = cipher.doFinal(cipherBytes); //
        return new String(plainBytes, StandardCharsets.UTF_8);
    } catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
        throw new SystemException("e.xx.xx.9002", "No Such setting error.",␣
↪e);
    } catch (InvalidKeyException |
             IllegalBlockSizeException |
             BadPaddingException e) {
        throw new SystemException("e.xx.xx.9003", "Invalid setting error.",␣
↪e);
    }
}
```

項番	説明
(1)	暗号化アルゴリズム、暗号利用モード、パディング方式を指定して、Cipher クラスのインスタンスを生成する。
(2)	復号処理を実行する。

OpenSSL

Cipher が同一であれば、公開鍵暗号化方式は別の方法で暗号化および復号を行うことが可能である。

ここでは、OpenSSL を利用してあらかじめキーペアを作成しておき、その公開鍵を利用して JCA による暗号化を行う。そして、その秘密鍵を利用して OpenSSL で復号処理を行う方法を説明する。

注釈: **OpenSSL**

OpenSSL でキーペアを作成する際はソフトウェアをインストールしておく必要がある。下記サイトよりダウンロードできる。

OpenSSL のダウンロード先

- Linux 用
- Windows 用

- 事前準備として、OpenSSL でキーペアを作成する。

```
$ openssl genrsa -out private.pem 2048  # (1)

$ openssl pkcs8 -topk8 -nocrypt -in private.pem -out private.pk8 -outform DER
 # (2)

$ openssl rsa -pubout -in private.pem -out public.der -outform DER  # (3)
```

項番	説明
(1)	OpenSSL で 2048 ビットの秘密鍵 (DER 形式) を生成する。
(2)	Java アプリケーションから読み込むために、秘密鍵を PKCS #8 形式に変換する。
(3)	秘密鍵から公開鍵 (DER 形式) を生成する。

- アプリケーションでは OpenSSL で作成した公開鍵を読み込み、読み込んだ公開鍵を利用して暗号化処理を行う。

```
public void useOpenSSLDecryption() {
    try {
        KeySpec publicKeySpec = new X509EncodedKeySpec(
                Files.readAllBytes(Paths.get("public.der"))); // (1)
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        PublicKey publicKey = keyFactory.generatePublic(publicKeySpec); // (2)

        byte[] cipherBytes = encryptByPublicKey("Hello World!", publicKey); //
 (3)

        Files.write(Paths.get("encryptedByJCA.txt"), cipherBytes);
        System.out.println("Please execute the following command:");
        System.out
                .println("openssl rsautl -decrypt -inkey hoge.pem -in␣
↪encryptedByJCA.txt");
    } catch (IOException e) {
        throw new SystemException("e.xx.xx.9001", "input/output error.", e);
    } catch (NoSuchAlgorithmException e) {
        throw new SystemException("e.xx.xx.9002", "No Such setting error.",␣
↪e);
    } catch (InvalidKeySpecException e) {
        throw new SystemException("e.xx.xx.9003", "Invalid setting error.",␣
↪e);
    }
}
```

項番	説明
(1)	公開鍵ファイルからバイナリデータを読み込む。
(2)	バイナリデータから PublicKey クラスのインスタンスを生成する。
(3)	公開鍵を利用して暗号化処理を行う。

- JCA で暗号化した内容が OpenSSL で復号できることを確認する。

```
$ openssl rsautl -decrypt -inkey private.pem -in encryptedByJCA.txt  # (1)
```

項番	説明
(1)	秘密鍵を利用して OpenSSL で復号する。

続いて、OpenSSL で作成したキーペアを利用して OpenSSL で暗号化、JCA で復号する方法を説明する。

- OpenSSL のコマンドを使用して暗号化処理を行う。

```
$ echo Hello | openssl rsautl -encrypt -keyform DER -pubin -inkey public.der -
↪out encryptedByOpenSSL.txt  # (1)
```

項番	説明
(1)	公開鍵を利用して OpenSSL で暗号化する。

- アプリケーションでは OpenSSL で作成した秘密鍵を読み込み、読み込んだ秘密鍵を利用して復号処理を行う。

```
public void useOpenSSLEncryption() {
    try {
        KeySpec privateKeySpec = new PKCS8EncodedKeySpec(
                Files.readAllBytes(Paths.get("private.pk8"))); // (1)
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        PrivateKey privateKey = keyFactory.generatePrivate(privateKeySpec); //
 (2)

        String plainText = decryptByPrivateKey(
               Files.readAllBytes(Paths.get("encryptedByOpenSSL.txt")),
               privateKey); // (3)
        System.out.println(plainText);
    } catch (IOException e) {
        throw new SystemException("e.xx.xx.9001", "input/output error.", e);
    } catch (NoSuchAlgorithmException e) {
        throw new SystemException("e.xx.xx.9002", "No Such setting error.",␣
↪e);
```

```
    } catch (InvalidKeySpecException e) {
        throw new SystemException("e.xx.xx.9003", "Invalid setting error.",␣
↪e);
    }
}
```

項番	説明
(1)	PKCS #8 形式の秘密鍵ファイルからバイナリデータを読み込み PKCS8EncodedKeySpec クラスのインスタンスを生成する。
(2)	KeyFactory クラスから PrivateKey クラスのインスタンスを生成する。
(3)	秘密鍵を利用して復号処理を行う。

ハイブリッド暗号化方式

公開鍵暗号化方式と同様、Spring Security ではハイブリッド暗号化方式に関する機能は提供されていないため、サンプルコードを用いて説明する。

このサンプルコードは、spring-security-rsa の RsaSecretEncryptor クラスを参考にしている。

暗号化

```
public byte[] encrypt(byte[] plainBytes, PublicKey publicKey, String salt) {
    byte[] random = KeyGenerators.secureRandom(32).generateKey(); // (1)
    BytesEncryptor aes = Encryptors.standard(
            new String(Hex.encode(random)), salt); // (2)

    try (ByteArrayOutputStream result = new ByteArrayOutputStream()) {
        final Cipher cipher = Cipher.getInstance("RSA"); // (3)
        cipher.init(Cipher.ENCRYPT_MODE, publicKey); // (4)
        byte[] secret = cipher.doFinal(random); // (5)

        byte[] data = new byte[2]; // (6)
        data[0] = (byte) ((secret.length >> 8) & 0xFF); //
        data[1] = (byte) (secret.length & 0xFF); //
        result.write(data); //
```

```
            result.write(secret); // (7)
            result.write(aes.encrypt(plainBytes)); // (8)

            return result.toByteArray(); // (9)
        } catch (IOException e) {
            throw new SystemException("e.xx.xx.9001", "input/output error.", e);
        } catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
            throw new SystemException("e.xx.xx.9002", "No Such setting error.",␣
↪e);
        } catch (InvalidKeyException | IllegalBlockSizeException |␣
↪BadPaddingException e) {
            throw new SystemException("e.xx.xx.9003", "Invalid setting error.",␣
↪e);
        }
    }
```

項番	説明
(1)	鍵長として 32 バイトを指定して `KeyGenerators#secureRandom` メソッドを呼び出し、`BytesKeyGenerator` クラスのインスタンスを生成する。 `BytesKeyGenerator#generateKey` メソッドを呼び出し、共通鍵を生成する。 詳細については、乱数生成を参照されたい。
(2)	生成した共通鍵とソルトを指定して `BytesEncryptor` クラスのインスタンスを生成する。
(3)	暗号化アルゴリズムとして RSA を指定して、`Cipher` クラスのインスタンスを生成する。
(4)	暗号化モード定数と公開鍵を指定して `Cipher` クラスのインスタンスを初期化する。
(5)	共通鍵の暗号化処理を実行する。この暗号化処理は公開鍵暗号化方式となる。
(6)	暗号化した共通鍵の長さをバイト配列の暗号文に格納する。格納された共通鍵の長さは復号時に使用される。
(7)	暗号化した共通鍵をバイト配列の暗号文に格納する。
(8)	平文を暗号化してバイト配列の暗号文に格納する。この暗号化処理は共通鍵暗号化方式となる。
(9)	バイト配列の暗号文を返却する。

復号

```
public byte[] decrypt(byte[] cipherBytes, PrivateKey privateKey, String salt)
↪{

    try (ByteArrayInputStream input = new ByteArrayInputStream(cipherBytes);
            ByteArrayOutputStream output = new ByteArrayOutputStream()) {
        byte[] b = new byte[2]; // (1)
        input.read(b); //
        int length = ((b[0] & 0xFF) << 8) | (b[1] & 0xFF); //

        byte[] random = new byte[length]; // (2)
        input.read(random); //
        final Cipher cipher = Cipher.getInstance("RSA"); // (3)
        cipher.init(Cipher.DECRYPT_MODE, privateKey); // (4)
        String secret = new String(Hex.encode(cipher.doFinal(random))); // (5)
        byte[] buffer = new byte[cipherBytes.length - random.length - 2]; //␣
↪(6)
        input.read(buffer); //
        BytesEncryptor aes = Encryptors.standard(secret, salt); // (7)
        output.write(aes.decrypt(buffer)); // (8)

        return output.toByteArray(); // (9)
    } catch (IOException e) {
        throw new SystemException("e.xx.xx.9001", "input/output error.", e);
    } catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
        throw new SystemException("e.xx.xx.9002", "No Such setting error.",␣
↪e);
    } catch (InvalidKeyException | IllegalBlockSizeException |␣
↪BadPaddingException e) {
        throw new SystemException("e.xx.xx.9003", "Invalid setting error.",␣
↪e);
    }
}
```

項番	説明
(1)	暗号化された共通鍵の長さを取得する。
(2)	暗号化された共通鍵を取得する。
(3)	暗号化アルゴリズムとして RSA を指定して、Cipher クラスのインスタンスを生成する。
(4)	復号モード定数と秘密鍵を指定して Cipher クラスのインスタンスを初期化する。
(5)	共通鍵の復号処理を実行する。この復号処理は公開鍵暗号化方式となる。
(6)	復号対象を取得する。
(7)	復号した共通鍵とソルトを指定して BytesEncryptor クラスのインスタンスを生成する。
(8)	復号処理を実行する。この復号処理は共通鍵暗号化方式となる。
(9)	復号したバイト配列の平文を返却する。

乱数生成

文字列型の疑似乱数生成

```
public static void createStringKey() {
    StringKeyGenerator generator = KeyGenerators.string(); // (1)
```

```
    System.out.println(generator.generateKey()); // (2)
    System.out.println(generator.generateKey()); //
}
```

項番	説明
(1)	鍵 (疑似乱数) 生成器 StringKeyGenerator クラスのインスタンスを生成する。 この生成器で鍵を生成すると、毎回異なる値となる。 鍵長は指定できず、常に 8 バイトの鍵が生成される。
(2)	generateKey メソッドで鍵 (疑似乱数) を生成する。

バイト配列型の疑似乱数生成

- 異なる鍵を生成する。

```
public static void createDifferentBytesKey() {
    BytesKeyGenerator generator = KeyGenerators.secureRandom(); // (1)
    System.out.println(Arrays.toString(generator.generateKey())); // (2)
    System.out.println(Arrays.toString(generator.generateKey())); //
}
```

項番	説明
(1)	KeyGenerators#secureRandom メソッドを呼び出し、鍵 (疑似乱数) 生成器 BytesKeyGenerator クラスのインスタンスを生成する。 この生成器で鍵を生成すると、毎回異なる値となる。 鍵長を指定しない場合、デフォルトで 8 バイトの鍵が生成される。
(2)	generateKey メソッドで鍵を生成する。

- 同一の鍵を生成する。

```
public static void createSameBytesKey() {
    BytesKeyGenerator generator = KeyGenerators.shared(32); // (1)
    System.out.println(Arrays.toString(generator.generateKey())); // (2)
    System.out.println(Arrays.toString(generator.generateKey())); //
}
```

項番	説明
(1)	鍵長として 32 バイトを指定して `KeyGenerators#shared` メソッドを呼び出し、鍵 (疑似乱数) 生成器 `BytesKeyGenerator` クラスのインスタンスを生成する。 この生成器で鍵を生成すると、毎回同じ値となる。 鍵長の指定は必須である。
(2)	`generateKey` メソッドで鍵を生成する。

9.9 OAuth

9.9.1 Overview

本節では、OAuth 2.0 の概要と Spring プロジェクトの一つである Spring Security OAuth を使用して OAuth 2.0 の仕様に沿った認可制御機能を実装する方法について説明する。

ちなみに：Spring Security OAuth のリファレンス

Spring Security OAuth は、本ガイドラインで紹介していない機能も提供している。Spring Security OAuth について詳しく知りたい場合は、OAuth 2 Developers Guide を参照されたい。

OAuth 2.0 とは

OAuth 2.0 とは、サードパーティ製アプリケーションが HTTP サービスを利用する際に、サーバ上の保護されたリソースに対するアクセス範囲の指定を可能にするための認可フレームワークのことである。

OAuth 2.0 は RFC として仕様化されており、関連する複数の技術仕様から構成されている。

以下に OAuth 2.0 の主要な仕様を示す。

表 9.39: **OAuth 2.0 の主要仕様**

RFC	概要	説明
RFC 6749	The OAuth 2.0 Authorization Framework	用語や認可方式などの、OAuth 2.0 としてのもっとも基本的な内容が記載されている技術仕様。
RFC 6750	Bearer Token Usage	RFC 6749 に記載されている認可制御を実現する場合に利用する、「署名なしアクセストークン」（以降、アクセストークンと表す）のサーバ間の受け渡し方法に関する技術仕様。 アクセストークンについては後述する。
RFC 6819	Threat Model and Security Considerations	OAuth 2.0 を使用するうえで考慮が必要となるセキュリティ要件に関する技術仕様。 本ガイドラインでは検討項目の具体的な説明は割愛する。

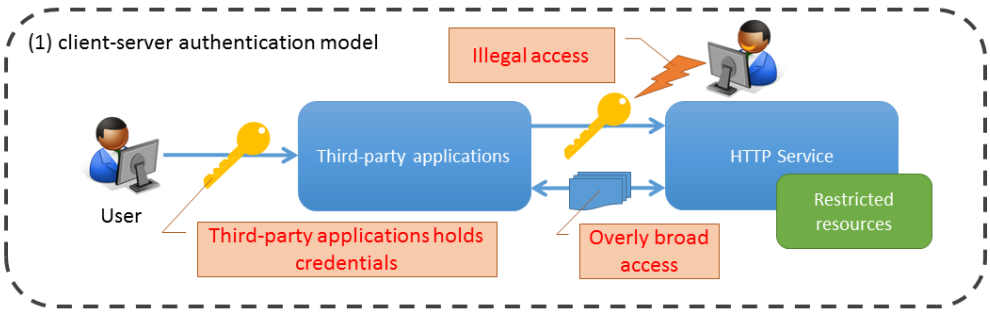
Continued on next page

表 9.39 – continued from previous page

RFC	概要	説明
RFC 7519	JSON Web Token (JWT)	署名が可能な JSON を含んだトークンである JSON Web Token (JWT) に関する技術仕様。
RFC 7523	JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants	RFC 6749 に記載されている認可制御の実現する場合に利用するアクセストークンとして、RFC 6819 で定められている JWT を利用する方法に関する技術仕様。
RFC 7009	OAuth 2.0 Token Revocation	トークンの無効化を行う追加エンドポイントに関する技術仕様。

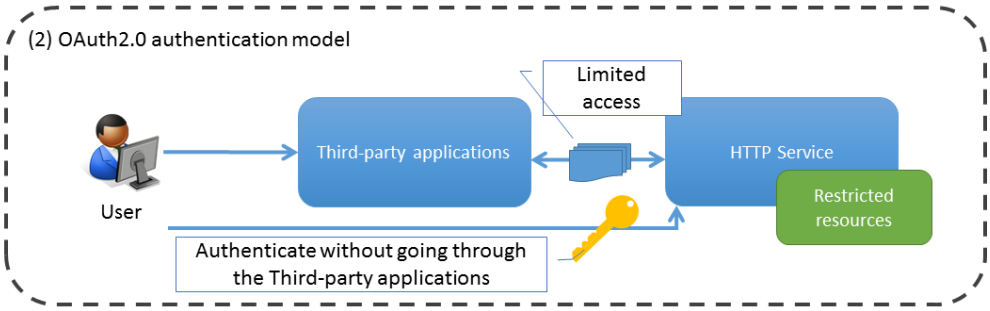
従来のクライアントサーバ型の認証モデルでは、サードパーティ製アプリケーションは HTTP サービスの保護されたリソースにアクセスするために、ユーザの認証情報（ユーザ名とパスワードなど）を利用して認証を行う。

つまり、ユーザは、サードパーティ製アプリケーションにリソースへのアクセス権を与えるために自身の認証情報をサードパーティと共有する必要があるが、これはサードパーティ製アプリケーションに不具合や悪意のある操作などが存在した場合に、ユーザの意図しないアクセスや情報漏洩等のリスクにつながる。



これに対し、OAuth 2.0 では HTTP サービスとの認証はユーザが直接行い、サードパーティ製アプリケーションには「アクセストークン」と呼ばれる認証済みリクエストを行うための情報を払い出すことで、サードパーティに認証情報を共有することなくリソースへアクセスすることが可能となる。

また、アクセストークン発行時にリソースに対するアクセス範囲（スコープ）を指定可能とすることで従来のクライアントサーバ型の認証モデルと比較してより柔軟なアクセス制御を実現している。



OAuth 2.0 のアーキテクチャ

ここでは OAuth 2.0 が定義するロール、スコープ、認可グラント、及びプロトコルフローについて説明する。OAuth 2.0 ではスコープや認可グラントという概念を定義しており、これらの概念を使用して認可の仕様を定めている。

ロール

OAuth 2.0 ではロールとして以下の 4 つを定義している。

表 9.40: **OAuth 2.0** におけるロール

ロール名	説明
リソースオーナ	保護されたリソースへのアクセスを許可するロール。人（エンドユーザ）など。
リソースサーバ	保護されたリソースを提供するサーバ。
認可サーバ	リソースオーナの認証と、アクセストークン（クライアントがリソースサーバにアクセスするときに必要な情報）の発行を行うサーバ。

Continued on next page

表 9.40 – continued from previous page

ロール名	説明
クライアント	リソースオーナの認可を得て、リソースオーナの代理として保護されたリソースに対してリクエストを行うロール。Web アプリケーションなど。クライアント情報は事前に認可サーバに登録され、認可サーバ内で一意な情報であるクライアント ID により管理される。 OAuth 2.0 ではクライアントクレデンシャル（クライアントの認証情報）の機密性を維持できる能力に基づき、クライアントタイプとして以下の 2 つを定義している。 (1) コンフィデンシャル クライアントクレデンシャルの機密性を維持することができるクライアント。 (2) パブリック リソースオーナのデバイス上で実行されるクライアントのように、クライアントクレデンシャルの機密性を維持することができず、かつ他の手段を用いたセキュアなクライアント認証が行えないクライアント。 また、OAuth 2.0 ではクライアントとして以下のような例を考慮して設計されている。 (1) コンフィデンシャル • Web アプリケーション（Web application） アプリケーションサーバ上で実行されるクライアント。 (2) パブリック • ユーザエージェントベースアプリケーション（user-agent-based application） クライアントコードがアプリケーションサーバからダウンロードされリソースオーナのユーザエージェント（ブラウザなど）内で実行されるクライアント。 JavaScript アプリケーションなど。 • ネイティブアプリケーション（native application） リソースオーナのデバイス上にインストールされ実行されるクライアント。

注釈: ユーザエージェントは、リソースオーナが使用する Web ブラウザ等を指す。本ガイドラインでは、エンドユーザの操作が発生する箇所を明確にするため、リソースオーナ（エンドユーザ）とユーザエージェントを別のものとして解説する。ガイドラインでリソースオーナと明示している場合に、エンドユーザの操作が発生する。

スコープ

OAuth 2.0 では保護されたリソースに対するアクセスを制御する方法としてスコープという概念を使用している。

認可サーバはクライアントからの要求に対し、認可サーバのポリシーまたはリソースオーナの指示に基づいてアクセストークンにスコープを含め、保護されたリソースに対するアクセス権（読み込み権限、書き込み権限など）を指定することが出来る。

プロトコルフロー

OAuth 2.0 では、以下のような流れでリソースへのアクセスを行う。

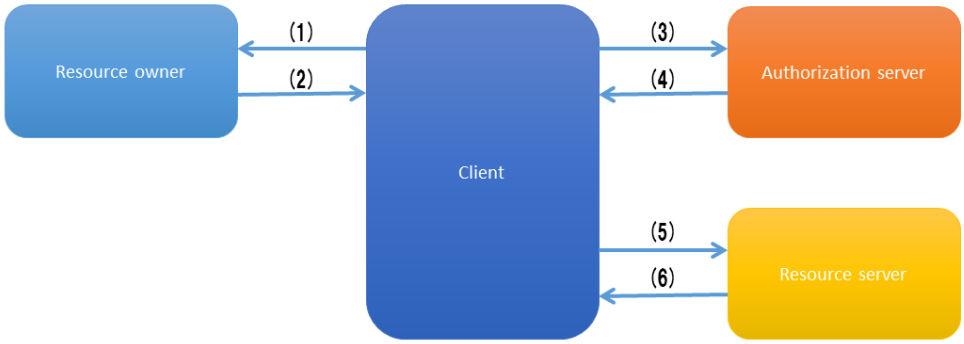


表 9.41: **OAuth 2.0** のプロトコルフロー

項番	説明
(1)	リソースオーナに対して認可を要求する。上の図ではクライアントがリソースオーナに 直接要求を行っているが、認可サーバを経由して行うほうが望ましい。 後述するグラントタイプの中では認可コードグラントとインプリシットグラントが 認可サーバを経由してリソースオーナに要求を行うフローになっている。
(2)	クライアントはリソースオーナからの認可を表すクレデンシャルとして認可グラント（後述）を受け取る。
(3)	クライアントは、認可サーバに対して自身の認証情報とリソースオーナが与えた認可グラントを提示することで、アクセス トークンを要求する。
(4)	認可サーバはクライアントを認証し、認可グラントの正当性を確認する。認可グラントが正当な場合、 アクセストークンを発行する。
(5)	クライアントはリソースサーバの保護されたリソースへリクエストを行い、発行されたアクセストークンにより認証する。
(6)	リソースサーバはアクセストークンの正当性を確認し、正当な場合、リクエストを受け入れリソースを応答する。

注釈: OAuth 1.0 で不評だった署名とトークン交換の複雑な仕組みを簡略化するために、OAuth 2.0 ではアクセストークンを扱うリクエストは HTTPS 通信で行うことを必須としている。（HTTPS 通信を使用することでアクセストークンの盗聴を防止する）

認可グラント

認可グラントは、リソースオーナからの認可を表し、クライアントがアクセストークンを取得する際に用いられる。OAuth 2.0 では、グラントタイプとして以下の 4 つを定義しているが、クレデンシャル項目を追加するなどの独自拡張を行うこともできる。

クライアントはいずれかのグラントタイプを利用して認可サーバへアクセストークンを要求し、取得したアクセストークンでリソースサーバにアクセスする。認可サーバはサポートするグラントタイプを必ず 1 つ以上定義しており、その中から使用するグラントタイプをクライアントからの認可リクエストによって決定する。

表 9.42: **OAuth 2.0** における認可グラント

グラントタイプ	説明
認可コードグラント	認可コードグラントのフローでは、認可サーバがクライアントとリソースオーナの仲介となって認可コードをクライアントへ発行し、クライアントが認可コードを認可サーバに渡すことでアクセストークンを発行する。 認可サーバが発行した認可コードを使用してアクセストークンを発行するため、クライアントへリソースオーナのクレデンシャルを共有する必要がない。 認可コードグラントは Web アプリケーションのように、コンフィデンシャルなクライアントが OAuth 2.0 を利用する際に使用する。
インプリシットグラント	インプリシットグラントのフローでは、認可コードグラントと同様に認可サーバが仲介するが、認可コードの代わりに直接アクセストークンを発行する。 これにより応答性、効率性が高いため、スクリプト言語を使用してブラウザ上で実行されるクライアントに適している。 しかし、アクセストークンが URL 中にエンコードされるため、リソースオーナや同一デバイス上の他のアプリケーションに漏えいする可能性があるほか、クライアントの認証を行わないことから、他のクライアントに対して発行されたアクセストークンを不正に用いた成りすまし攻撃のリスクがある。 セキュリティ上のリスクがあるため、応答性、効率性が求められるパブリックなクライアントでのみ使用すること。

Continued on next page

表 9.42 – continued from previous page

グラントタイプ	説明
リソースオーナパスワードクレデンシャルグラント	リソースオーナパスワードクレデンシャルグラントのフローでは、クライアントがリソースオーナの認証情報を認可グラントとして使用して、直接アクセストークンを発行する。 クライアントへリソースオーナのクレデンシャルを共有する必要があるため、クライアントの信頼性が低い場合、クレデンシャルの不正利用や漏洩のリスクがある。 リソースオーナパスワードクレデンシャルグラントはリソースオーナとクライアントの間で高い信頼があり、かつ他のグラントタイプが利用できない場合にのみ使用すること。
クライアントクレデンシャルグラント	クライアントクレデンシャルグラントのフローでは、クライアントの認証情報を認可グラントとして使用して、直接アクセストークンを発行する。 クライアントがリソースオーナであるような場合に使用する。

警告: *OAuth 2.0 における認可グラント*で解説した通り、認可コードグラント以外のグラントタイプには、セキュリティ上のリスクや、使用上の制約がある。そのため、認可コードグラントの利用を優先して検討されたい。

認可コードグラント

認可コードグラントのフローを以下に示す。

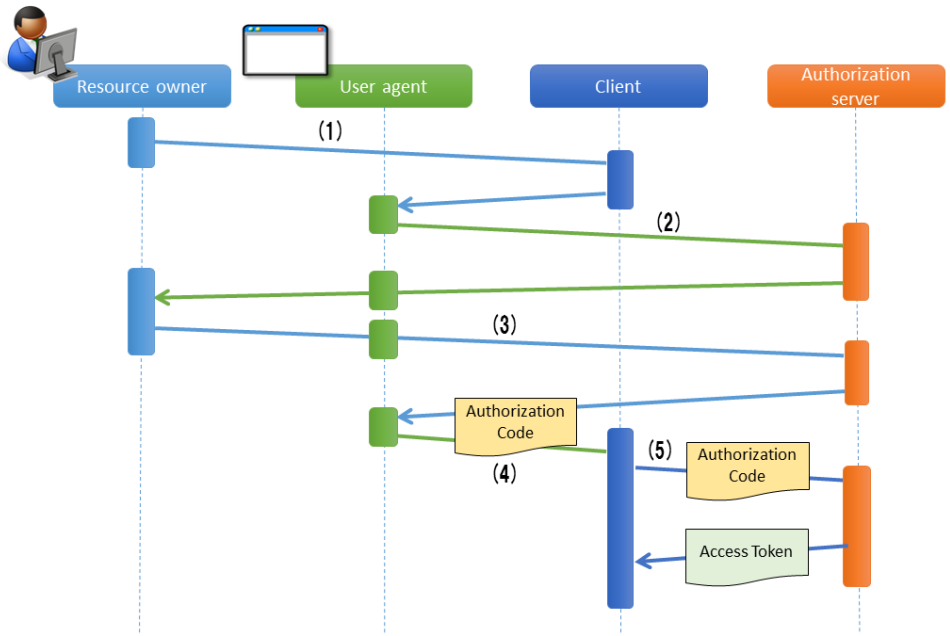


表 9.43: 認可コードグラントフロー

項番	説明
(1)	リソースオーナは、ユーザエージェント (Web ブラウザなど) を介してクライアントが提供するリソースサーバの保護されたリソースにアクセスする。 クライアントはリソースオーナから認可の取得を行うために、リソースオーナが操作するユーザエージェントを認可サーバの認可エンドポイントにリダイレクトさせる。 このとき、クライアントは自身を識別するためのクライアント ID と、オプションとしてリソースに要求するスコープ、認可サーバが認可処理後にユーザエージェントを戻すリダイレクト URI、state をリクエストパラメータに含める。 state はユーザエージェントに紐付くランダムな値であり、一連のフローが同じユーザエージェントで実行されたことを保証するために利用される (CSRF 対策)。
(2)	ユーザエージェントは、クライアントに指示された認可サーバの認可エンドポイントにアクセスする。 認可サーバはユーザエージェント経由でリソースオーナを認証し、リクエストパラメータのクライアント ID、スコープ、リダイレクト URI を元に、自身に登録済みのクライアント情報と比較しパラメータの正当性確認を行う。 確認完了後、アクセス要求の許可/拒否をリソースオーナにたずねる。

Continued on next page

表 9.43 – continued from previous page

項番	説明
(3)	リソースオーナはアクセス要求の許可/拒否を認可サーバに送信する。 リソースオーナがアクセスを許可した場合、認可サーバは、リクエストパラメータに含まれるリダイレクト URI を用いて、ユーザエージェントをクライアントにリダイレクトさせる指示を出す。 その際、認可コードをリダイレクト URI のリクエストパラメータとして付与する。
(4)	ユーザエージェントは認可コードが付与されたリダイレクト URI にアクセスする。 クライアントの処理が完了するとリソースオーナにレスポンスを返却する。
(5)	クライアントはアクセストークンを要求するために、認可コードを認可サーバのトークンエンドポイントに送信する。 認可サーバのトークンエンドポイントはクライアントの認証と認可コードの正当性の検証を行い、正当である場合アクセストークンと任意でリフレッシュトークンを発行する。 リフレッシュトークンはアクセストークンが無効化された、または期限切れの際に新しいアクセストークンを発行するために使用される。

インプリシットグラント

インプリシットグラントのフローを以下に示す。

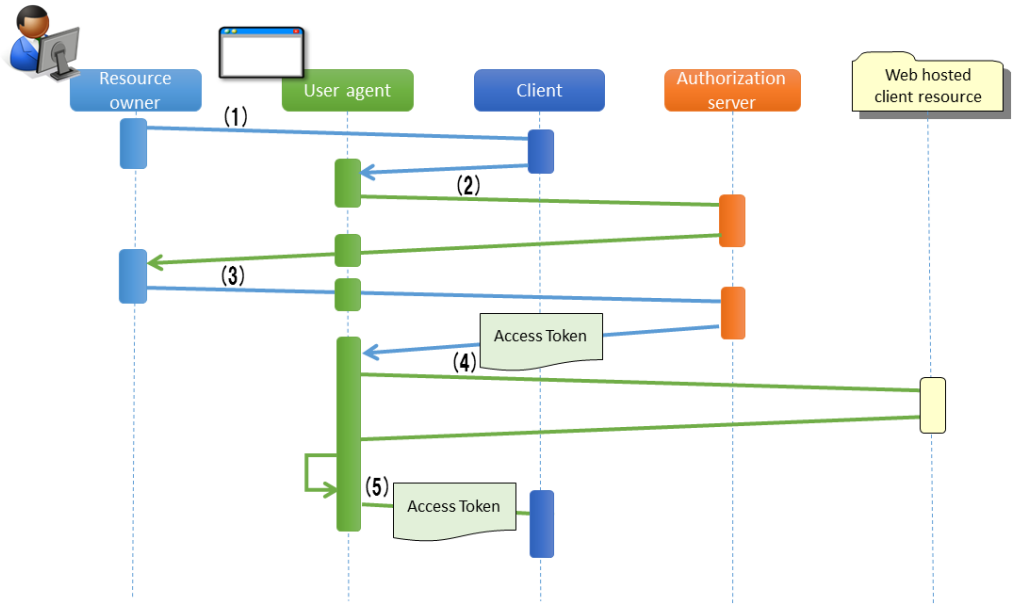


表 9.44: インプリシットグラントフロー

項番	説明
(1)	リソースオーナは、ユーザエージェントを介してクライアントが提供するリソースサーバの保護されたリソースが必要なページにアクセスする。 クライアントはリソースオーナから認可の取得とアクセストークンの発行を行うために、リソースオーナのユーザエージェントを認可サーバの認可エンドポイントにアクセスさせる。 このとき、クライアントは自身を識別するためのクライアント ID と、オプションとしてリソースに要求するスコープ、認可サーバが認可処理後にユーザエージェントを戻すリダイレクト URI、state をリクエストパラメータに含める。 state はユーザエージェントに紐付くランダムな値であり、一連のフローが同じユーザエージェントで実行されたことを保証するために利用される (CSRF 対策)。
(2)	ユーザエージェントは、クライアントに指示された認可サーバの認可エンドポイントにアクセスする。 認可サーバはユーザエージェント経由でリソースオーナを認証し、リクエストパラメータのクライアント ID、スコープ、リダイレクト URI を元に、自身に登録済みのクライアント情報と比較しパラメータの正当性確認を行う。 確認完了後、アクセス要求の許可/拒否をリソースオーナにたずねる。

Continued on next page

表 9.44 – continued from previous page

項番	説明
(3)	リソースオーナはアクセス要求の許可/拒否を認可サーバに送信する。 リソースオーナがアクセスを許可した場合、認可サーバの認可エンドポイントはリクエストパラメータのリダイレクト URI を用いてユーザエージェントをクライアントリソースにリダイレクトさせる指示を出し、アクセストークンをリダイレクト URI の URL フラグメントに付与する。 ここで「クライアントリソース」とは、クライアントアプリケーションとは別に Web サーバ等にホストしておいた静的リソースを指す。
(4)	ユーザエージェントはリダイレクトの指示に従い、クライアントリソースにリクエストを送信する。このとき、URL フラグメントの情報をローカルで保持し、リダイレクトの際には URL フラグメントを送信しない。 クライアントリソースにアクセスすると、Web ページ（通常は埋め込みスクリプトを含む HTML ドキュメント）が返却される。 ユーザエージェントは Web ページに含まれるスクリプトを実行し、ローカルで保持していた URL フラグメントからアクセストークンを抽出する。
(5)	ユーザエージェントはアクセストークンをクライアントに渡す。

リソースオーナパスワードクレデンシャルグラント

リソースオーナパスワードクレデンシャルグラントのフローを以下に示す。

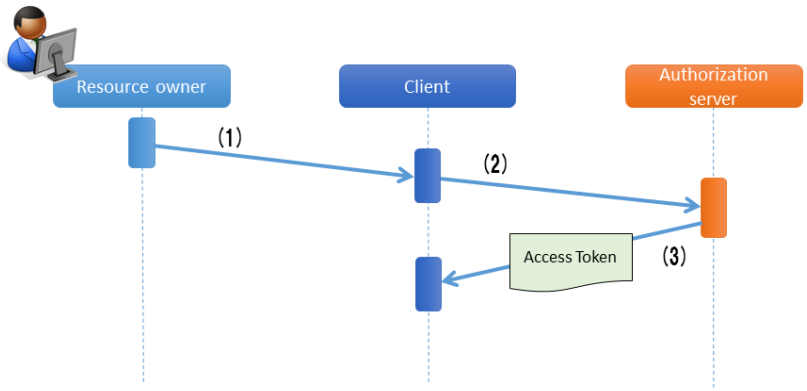


表 9.45: リソースオーナパスワードクレデンシャルグラントフロー

項番	説明
(1)	リソースオーナがクライアントにクレデンシャル（ユーザ名、パスワード）を提供する。
(2)	クライアントはアクセストークンを要求するために、認可サーバのトークンエンドポイントにアクセスする。 このとき、クライアントはリソースオーナから指定されたクレデンシャルとリソースに要求するスコープをリクエストパラメータに含める。
(3)	認可サーバのトークンエンドポイントはクライアントを認証し、リソースオーナのクレデンシャルを検証する。正当である場合アクセストークンを発行する。

クライアントクレデンシャルグラント

クライアントクレデンシャルグラントのフローを以下に示す。

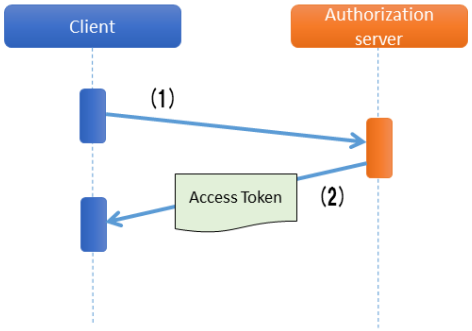


表 9.46: クライアントクレデンシャルグラントフロー

項番	説明
(1)	クライアントはアクセストークンを要求するために、認可サーバのトークンエンドポイントにアクセスする。 このとき、クライアントはクライアント自身のクレデンシャルを含めてアクセストークンを要求する。
(2)	認可サーバのトークンエンドポイントはクライアントを認証し、認証に成功した場合アクセストークンを発行する。

アクセストークンのライフサイクル

アクセストークンはクライアントが提示する認可グラントの正当性を認可サーバが確認することで発行される。発行されたアクセストークンは、認可サーバのポリシーまたはリソースオーナの指示に基づいたスコープが与えられ、保護されたリソースに対するアクセス権を保持する。アクセストークンは発行時に有効期限が設定され、有効期限切れとなると保護されたリソースに対するアクセス権を失効される。

アクセストークンの発行から失効までの流れは以下のようになる。

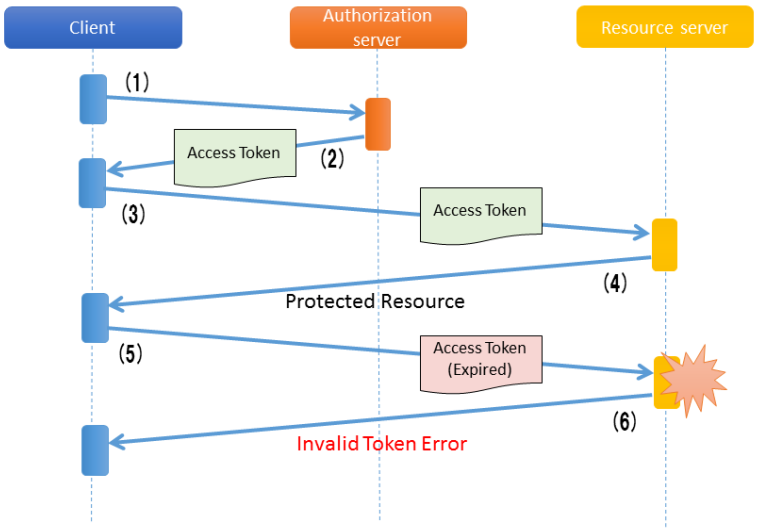


表 9.47: アクセストークンの発行から失効までのフロー

項番	説明
(1)	クライアントが認可グラントを提示し、アクセストークンを要求する。
(2)	認可サーバはクライアントが提示した認可グラントを確認し、アクセストークンを発行する。
(3)	クライアントはアクセストークンを提示し、リソースサーバの保護されたリソースを要求する。
(4)	リソースサーバはクライアントが提示したアクセストークンの正当性を検証し、正当であればリソースサーバの保護されたリソースに対して処理を行う。
(5)	クライアントはアクセストークン（有効期限切れ）を提示し、リソースサーバの保護されたリソースを要求する。
(6)	リソースサーバはクライアントが提示したアクセストークンの正当性を検証し、アクセストークンの有効期限が切れている場合はエラーを返却する。

アクセストークンが有効期限切れとなると保護されたリソースに対するアクセス権を失効されるが、アクセストークンが有効期限切れとなる前にアクセストークンを無効化し保護されたリソースに対するアクセス権を失効させることも可能である。

アクセストークンが有効期限切れとなる前に無効化する場合、クライアントより認可サーバにトークンの取り消し依頼を行う。無効化されたアクセストークンは保護されたリソースに対するアクセス権を失効される。

アクセストークンが有効期限切れとなった場合、クライアントがアクセストークンを再取得するためには認可サーバへ認可グラントの再提示を行い、認可サーバによる正当性の再確認が必要になる。そのため、アクセストークンの有効期限を短く設定した場合はユーザビリティが下がってしまう。一方で、アクセストークンの有効期限を長く設定した場合はアクセストークンの漏洩、漏洩時に悪用されるリスクが高まってしまう。

ユーザビリティを下げずに漏洩、漏洩時のリスクを下げるためにはリフレッシュトークンが用いられる。リフレッシュトークンはアクセストークンが無効化されたあるいは期限切れの際、認可グラントの再提示を行うこ

となく新しいアクセストークンを取得するために利用される。リフレッシュトークンも発行時に有効期限が設定され、リフレッシュトークンが有効期限切れとなった場合はアクセストークンの再発行ができなくなる。

アクセストークンの有効期限に短い期間を設定し、リフレッシュトークンの有効期限に長い期間を設定することで、短いサイクルでアクセストークンが再発行されユーザビリティを保ちつつアクセストークン漏洩及び漏洩時の悪用のリスクも抑えることができる。

リフレッシュトークンの発行はオプションであり、認可サーバの判断に委ねられる。

リフレッシュトークンによるアクセストークンの再発行の流れは以下のようになる。

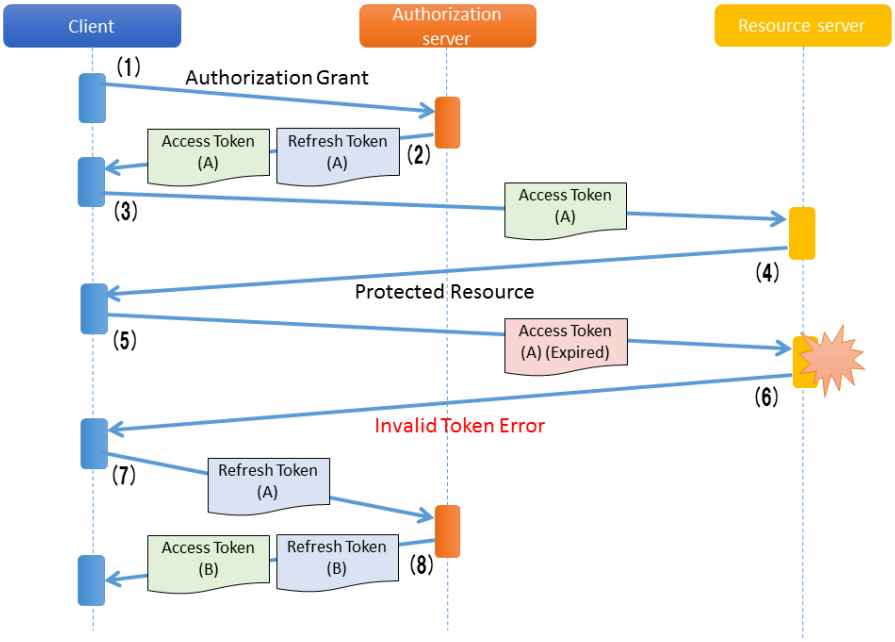


表 9.48: アクセストークンの発行から再発行までのフロー

項番	説明
(1)	クライアントが認可グラントを提示し、アクセストークンを要求する。
(2)	認可サーバはクライアントが提示した認可グラントを確認し、アクセストークンとリフレッシュトークンを発行する。
(3)	クライアントはアクセストークンを提示し、リソースサーバの保護されたリソースを要求する。

Continued on next page

表 9.48 – continued from previous page

項番	説明
(4)	リソースサーバはクライアントが提示したアクセストークンの正当性を検証し、正当であればリソースサーバの保護されたリソースに対して処理を行う。
(5)	クライアントはアクセストークン（有効期限切れ）を提示し、リソースサーバの保護されたリソースを要求する。
(6)	リソースサーバはクライアントが提示したアクセストークンの正当性を検証し、アクセストークンの有効期限が切れている場合はエラーを返却する。
(7)	リソースサーバよりアクセストークンの有効期限切れエラーが返却された場合、クライアントはリフレッシュトークン（有効期限切れ）を提示することで新しいアクセストークンを要求する。
(8)	認可サーバはクライアントが提示したリフレッシュトークンの正当性を検証し、正当であればアクセストークンとオプションでリフレッシュトークンを発行する。

リフレッシュトークンの有効期限が期限切れとなった場合は認可サーバへ認可グラントの再提示を行う。

表 9.49: リフレッシュトークンの発行から再発行までのフロー

項番	説明
(1)	クライアントが有効期限切れのアクセストークンを提示し、リソースサーバよりアクセストークンの有効期限切れエラーが返却された場合、 クライアントはリフレッシュトークン（有効期限切れ）を提示することで新しいアクセストークンを要求する。
(2)	認可サーバはクライアントが提示したリフレッシュトークンの正当性を検証し、リフレッシュトークンの有効期限が切れている場合はエラーを返却する。

Continued on next page

表 9.49 – continued from previous page

項番	説明
(3)	認可サーバよりリフレッシュトークンの有効期限切れエラーが返却された場合、クライアントは認可グラントを再提示し、アクセストークンを要求する。
(4)	認可サーバはクライアントが提示した認可グラントを確認し、アクセストークンとリフレッシュトークンを発行する。

Spring Security OAuth のアーキテクチャ

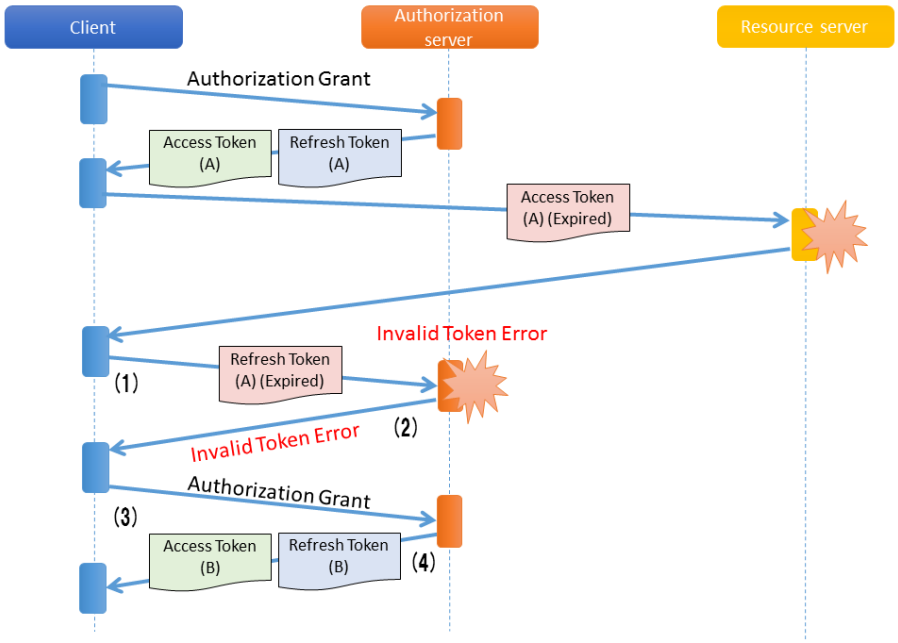
Spring Security OAuth は、OAuth 2.0 で定義されているロールのうち、認可サーバ、リソースサーバ、クライアントの 3 つのロールを Spring アプリケーションとして構築する際に必要となる機能を提供するライブラリである。Spring Security OAuth は、Spring Framework(Spring MVC) や Spring Security が提供する機能と連携して動作する仕組みになっており、Spring Security OAuth が提供するデフォルト実装を適切にコンフィギュレーション（Bean 定義）するだけで、認可サーバ、リソースサーバ、クライアントを構築することができる。また、Spring Framework や Spring Security と同様に数多くの拡張ポイントが用意されており、Spring Security OAuth が提供するデフォルト実装で実現できない要件を組み込むことができるようになっている。

なお、各ロール間のリクエストに対する認証・認可には Spring Security が提供する機能を利用するため、そちらの詳細は認証及び 認可を参照されたい。

注釈: 一般的に、OAuth 2.0 では全てのアプリケーションを 1 つのプロバイダが提供するのではなく、プロバイダが認可サーバ、リソースサーバを提供し、それらと連携するクライアントのみを実装するようなケースも多くある。そういった場合に連携するアプリケーションが Spring Security OAuth を使用して実装されているとは限らない。実装方法の解説では、Spring Security OAuth 以外のアーキテクチャで実装されたアプリケーションと連携する方法についても、適宜 Note 等で補足していく。

連携するアプリケーションの仕様に応じて、適宜本ガイドラインで紹介していている実装方法をカスタマイズして利用されたい。

Spring Security OAuth を使用して認可サーバ、リソースサーバ、クライアントを構築した場合、以下のような流れで処理が行われる。


Client
Authorization server
Resource server
Authorization Grant
Access Token (A)
Refresh Token (A)
Access Token (A) (Expired)
Invalid Token Error
Refresh Token (A) (Expired)
(1)
(2)
Invalid Token Error
Authorization Grant
(3)
Access Token (B)
Refresh Token (B)
(4)

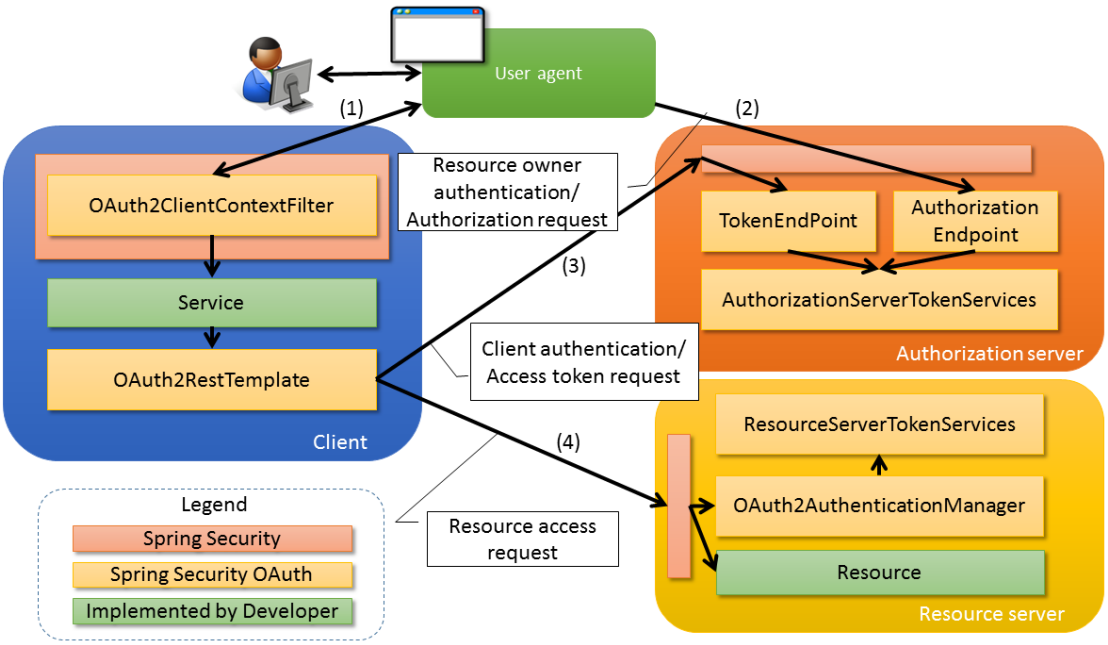

User agent
(1)
(2)
OAuth2ClientContextFilter
Service
OAuth2RestTemplate
Client
Resource owner authentication/ Authorization request
(3)
Client authentication/ Access token request
(4)
Resource access request
TokenEndPoint
Authorization Endpoint
AuthorizationServerTokenServices
Authorization server
ResourceServerTokenServices
OAuth2AuthenticationManager
Resource
Resource server
Legend
Spring Security
Spring Security OAuth
Implemented by Developer

表 9.50: **Spring Security OAuth** のフロー

項番	説明
(1)	リソースオーナはユーザエージェントを介してクライアントへアクセスする。 クライアントはサービスより `org.springframework.security.oauth2.client.OAuth2RestTemplate` の呼び出しを行う。 認可エンドポイントにアクセスする認可グラントの場合、ユーザエージェントへ認可サーバの認可エンドポイントへリダイレクトさせるよう指示する。
(2)	ユーザエージェントは認可サーバの認可エンドポイント (`org.springframework.security.oauth2.provider.endpoint.AuthorizationEndpoint`) へアクセスする。 認可エンドポイントはリソースオーナへ認可を問い合わせる画面を表示した後に、リソースオーナからの 認可リクエストを受け取り認可コードまたはアクセストークンを発行する。 発行した認可コードまたはアクセストークンは、リダイレクトを使用してユーザエージェント経由でクライアントに渡される。
(3)	クライアントは `OAuth2RestTemplate` を使用して認可サーバのトークンエンドポイント (`org.springframework.security.oauth2.provider.endpoint.TokenEndpoint`) へアクセス する。 トークンエンドポイントは `org.springframework.security.oauth2.provider.token.AuthorizationServerTokenServices` を呼び出し、クライアントが提示した認可グラントの検証及びアクセストークンの発行を行う。 発行したアクセストークンはクライアントへの応答として渡される。
(4)	クライアントは `OAuth2RestTemplate` を使用し、認可サーバから取得したアクセストークンをリクエストヘッダに設定してリソースサーバにアクセスする。 リソースサーバは `org.springframework.security.oauth2.provider.authentication.OAuth2AuthenticationManager` を呼び出し、 `org.springframework.security.oauth2.provider.token.ResourceServerTokenServices` を介してアクセストークンとアクセストークンに紐づく認証情報の検証を行う。 アクセストークンの検証に成功した場合、クライアントからのリクエストに応じたリソースを返却する。

注釈: 前述のとおり、OAuth 2.0 では各エンドポイントにおいて HTTPS 通信の使用を前提としているが、HTTPS 通信を使用するのが SSL アクセラレータや Web サーバまでの場合や、ロードバランサを使用して複数のアプリケーションサーバに分散させる場合がある。リソースオーナによって認可された後にクライアントに認可コードまたは、アクセストークンを連携するためのリダイレクト URI を組み立てる際に、SSL アクセラレータや Web サーバ、ロードバランサを指し示すリダイレクト URI を組み立てる必要がある。

Spring(Spring Security OAuth) では以下のいずれかのヘッダを使用してリダイレクト用の URL を組み立てる仕組みが提供されている。

- Forwarded ヘッダ
- X-Forwarded-Host ヘッダ、X-Forwarded-Port ヘッダ、X-Forwarded-Proto ヘッダ

SSL アクセラレータや Web サーバ、ロードバランサ側で上記ヘッダを付与するように設定し、Spring(Spring Security OAuth) が正しいリダイレクト URI を生成できるようにする必要がある。これを行わない場合、認可コードグラントやインプリシットグラントにおいて行うリクエストパラメータ（リダイレクト URI）の検証に失敗する可能性がある。

ちなみに: Spring Security OAuth が提供するエンドポイントは Spring MVC の機能を拡張して実現している。Spring Security OAuth が提供するエンドポイントには`@FrameworkEndpoint` アノテーションがクラスに設定されている。これは`@Controller` アノテーションで開発者がコンポーネントとして登録したクラスと競合させないためである。また、`@FrameworkEndpoint` アノテーションでコンポーネントとして登録されたエンドポイントは、`RequestMappingHandlerMapping` の拡張クラスである `org.springframework.security.oauth2.provider.endpoint.FrameworkEndpointHandlerMapping` がエンドポイントの`@RequestMapping` アノテーションを読み取り、URL と合致する`@FrameworkEndpoint` のメソッドを、ハンドラメソッドとして扱っている。

認可サーバ

認可サーバでは、リソースオーナの認証、リソースオーナからの認可の取得、およびアクセストークンの発行を行う機能を提供する。一部のグラントタイプでは、アクセストークンを発行するためにリソースオーナに認可を問い合わせる必要があるため、リソースオーナの認証と、リソースオーナからの認可の取得を行う機能についても提供している。

認可サーバはクライアント情報（どのクライアントに、どのリソースに対するどのスコープの認可を与えるかの情報）に基づいて、リソースにどのスコープでのアクセスを認可するかや、アクセストークンを発行してよいかの検証を行う。

なお、クライアント情報は事前に認可サーバに登録しておく必要があるが、OAuth 2.0 の仕様では認可サーバ

を利用するクライアント情報の登録手順について定められておらず、Spring Security OAuth においてもクライアント情報の登録手続きの機能は提供されていない。そのため、アプリケーションでクライアント情報を登録するためのインターフェイスを提供したい場合には、独自に実装する必要がある。

クライアントの認証

後述する各エンドポイントでは、認可サーバが管理するクライアント情報に基づき、リクエストに含まれるクライアントの認証を行う機能を提供している。

リソースオーナの認証

後述するリソースオーナからの認可の取得を行う場合、認可サーバはリソースオーナの認証も行う必要がある。この機能は Spring Security が提供している認証機能を使用して実現する。

詳細については 認証を参照のこと。

リソースオーナからの認可の取得

Spring Security OAuth は、認可エンドポイント（`AuthorizationEndpoint`）を Spring MVC の機能を利用して提供している。

クライアントは認可リクエスト送信時に利用したいスコープを指定し、リソースオーナが指定されたスコープを認可するか、認可サーバに事前に登録されている、クライアントに割り当てられたスコープと一致する場合に、認可サーバでクライアントに対してそのスコープを認可する。クライアントに対して認可を行う際には 認可の節で説明している Spring Security のロールによる認可も併用することができる。

以下に認可エンドポイントアクセス時のフローを示す。

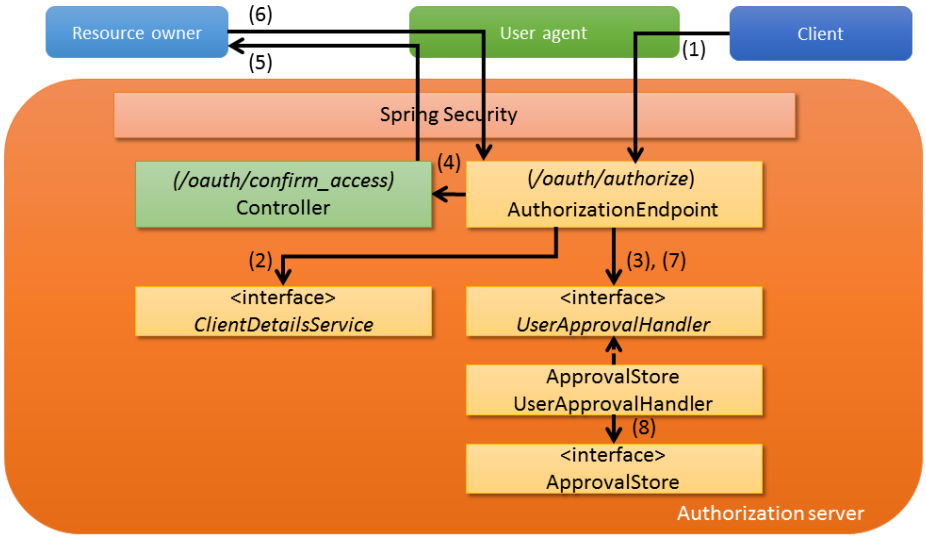


表 9.51: 認可サーバの動き（認可エンドポイントアクセス時）

項番	説明
(1)	クライアントからの指示により、ユーザエージェントが認可サーバの認可エンドポイント (/oauth/authorize) にアクセスすることで認可サーバの処理が実行される。
(2)	アクセス先の URL(/oauth/authorize) を処理する AuthorizationEndpoint では、org.springframework.security.oauth2.provider.ClientDetailsService のメソッドを呼び出し、事前に登録されているクライアント情報を取得後、リクエストパラメータを検証する。
(3)	org.springframework.security.oauth2.provider.approval.UserApprovalHandler のメソッドを呼び出し、クライアントへスコープに対する認可が登録されているかチェックする。
(4)	認可が未登録である場合、UserApprovalHandler よりリソースオーナに認可を問い合わせる画面 (認可画面) の要素を取得し、認可画面を表示させる URL(/oauth/confirm_access) へフォワードさせる。 このとき、問い合わせの対象となるスコープはリクエストパラメータと事前に登録されているクライント情報の積をとり、Spring MVC の@SessionAttributes を利用して連携される。
(5)	フォワード先の URL(/oauth/confirm_access) を処理するコントローラでは、ユーザエージェント経由でリソースオーナに認可画面を表示させる。
(6)	リソースオーナの画面操作により認可が行われた場合、再度認可エンドポイント (/oauth/authorize) にアクセスが行われる。 このとき、リクエストパラメータとして user_oauth_approval が付与される。
(7)	AuthorizationEndpoint では user_oauth_approval パラメータを付与しアクセスされた場合、UserApprovalHandler のメソッドを呼び出し認可情報を登録する。

Continued on next page

表 9.51 – continued from previous page

項番	説明
(8)	`UserApprovalHandler` の実装である `org.springframework.security.oauth2.provider.approval.ApprovalStoreUserApprovalHandler` では `org.springframework.security.oauth2.provider.approval.ApprovalStore` により認可の状態を管理する。 リソースオーナにより認可が行われた場合、`ApprovalStore` のメソッドを呼び出し、指定された情報を登録する。

注釈: 問い合わせの対象となるスコープは前述のとおり、認可サーバに事前に登録されているスコープと、クライアントが認可リクエスト時にリクエストパラメータで指定したスコープの積となる。例として、認可サーバで READ と CREATE と DELETE のスコープが割り当てられているクライアントに対して、READ と CREATE のスコープをリクエストパラメータで指定した場合は (READ,CREATE,DELETE) と (READ,CREATE) の積である、スコープ READ,CREATE が割り当てられる。認可サーバでクライアントに割り当てられていないスコープをリクエストパラメータで指定した場合はエラーとなり、アクセスが拒否される。

不正クライアントエラー

RFC 6749 の 4.1.2.1. Error Response にあるとおり、リクエストパラメータの検証でリダイレクト URI もしくはクライアント ID の正当性が確認できない場合、不正なクライアントであるとみなす。この場合、ユーザエージェントを不正なクライアントへリダイレクトさせず、ユーザエージェントを認可サーバが提供するエラー画面へと遷移させることで、リソースオーナに不正なクライアントであることを通知する。本ガイドラインでは、上記エラーを**不正クライアントエラー**と呼ぶこととする。

認可エンドポイントで不正クライアントエラーが発生した場合、クライアントにエラー通知を行わず認可サーバのエラー画面へ遷移する。認可エンドポイントで不正クライアントエラー以外のエラーが発生した場合、認可サーバからクライアントへのリダイレクトによりエラー通知が行われる。認可エンドポイントでエラーが発生した場合のフローを以下に示す。

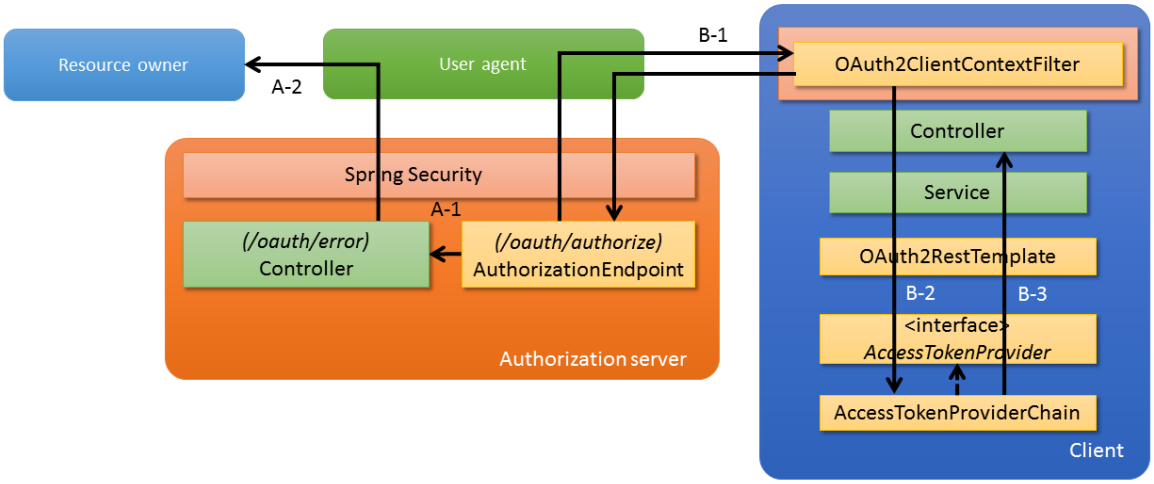


表 9.52: 認可サーバの動き（不正クライアントエラーのエラーハンドリング）

項番	説明
A-1	不正クライアントエラーが発生した場合、エラー画面を表示させる URL(`/oauth/error`) へフォワードさせる。
A-2	フォワード先の URL(`/oauth/error`) を処理するコントローラでは、 ユーザエージェント経由でリソースオーナにエラー画面を表示させる。

表 9.53: 認可サーバの動き（不正クライアントエラー以外のエラーハンドリング）

項番	説明
B-1	不正クライアントエラー以外のエラーが発生した場合、認可サーバからクライアントへのリダイレクトによりエラー通知が行われる。 リダイレクト URI にリクエストパラメータとしてエラー情報が付与される。
B-2	`OAuth2RestTemplate` を経由して `org.springframework.security.oauth2.client.token.AccessTokenProviderChain` にエラー情報が渡される。
B-3	渡されたエラー情報を例外に復元して、例外としてスローする。

アクセストークンの発行

Spring Security OAuth は、トークンエンドポイント（`TokenEndpoint`）を Spring MVC の機能を利用して提供している。

トークンエンドポイントはアクセストークンの発行を `AuthorizationServerTokenServices` のデフォルト実装である `org.springframework.security.oauth2.provider.token.DefaultTokenServices` によって行っている。アクセストークンの発行の際には、`ClientDetailsService` を介して認可サーバに登録済みのクライアント情報を取得し、アクセストークン発行の可否の検証に使用している。

以下にトークンエンドポイントアクセス時のフローを示す。

表 9.54: 認可サーバの動き（トークンエンドポイントアクセス時）

項番	説明
(1)	クライアントが認可サーバのトークンエンドポイント (`/oauth/token`) にアクセスすることでトークンエンドポイントの処理が実行される。

Continued on next page

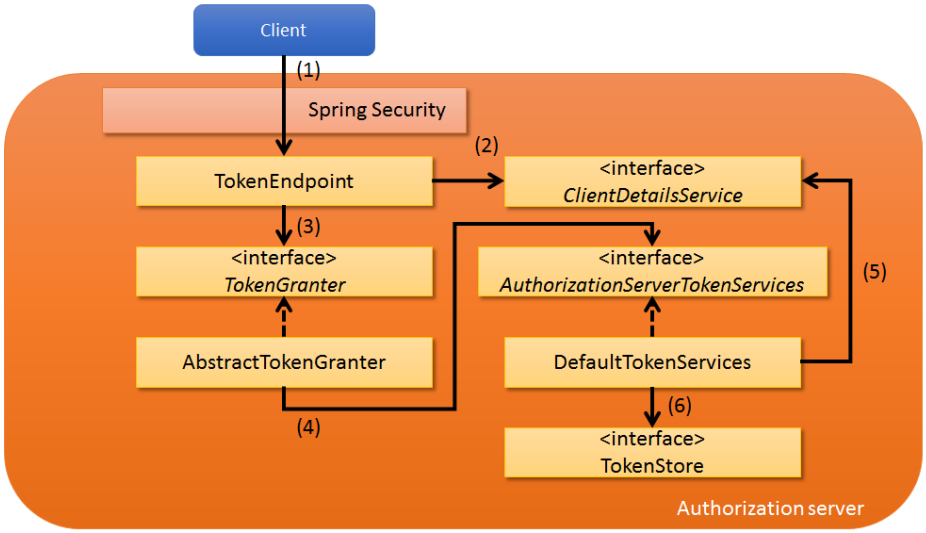
表 9.54 – continued from previous page

項番	説明
(2)	ClientDetailsService のメソッドを呼び出し、事前に登録されているクライアント情報を取得後、リクエストパラメータのスコープがクライアントに登録済みのものかチェックする。
(3)	スコープが登録済みのものであった場合、org.springframework.security.oauth2.provider.TokenGranter のメソッドを呼び出し、アクセストークンを発行する。
(4)	TokenGranter の実装である org.springframework.security.oauth2.provider.token.AbstractTokenGranter では AuthorizationServerTokenServices のアクセストークンを発行するメソッドを呼び出す。 AbstractTokenGranter はグラントタイプ別に実装されている TokenGranter の基底クラスであり、実際の処理は各クラスに委譲される。
(5)	AuthorizationServerTokenServices の実装である DefaultTokenServices で ClientDetailsService のメソッドを呼び出し、発行するアクセストークンに設定する有効期限、リフレッシュトークン発行の有無、有効な場合は有効期限を取得する。
(6)	DefaultTokenServices で ClientDetailsService から取得した情報を基にアクセストークンを発行する。 発行したアクセストークンは、アクセストークンを管理する org.springframework.security.oauth2.provider.token.TokenStore のメソッドにて登録される。

リソースサーバ

リソースサーバでは、アクセストークン自体の妥当性とアクセストークンが保持するスコープ内のリソースへのアクセスであることを検証する機能を提供する。

Spring Security OAuth では、アクセストークンの検証機能を、Spring Security の認証・認可の枠組みを利用して実現している。具体的には、Authentication Filter に Spring Security OAuth が提供する org.springframework.security.oauth2.provider.authentication.OAuth2AuthenticationProcessingFilter を、認証処理に AuthenticationManager



の実装クラスである `OAuth2AuthenticationManager` を、認証エラーの応答に `AuthenticationEntryPoint` の実装クラスである `org.springframework.security.oauth2.provider.error.OAuth2AuthenticationEntryPoint` をそれぞれ使用している。

Spring Security の詳細については 認証を参照のこと。

以下にリソースサーバの構成を示す

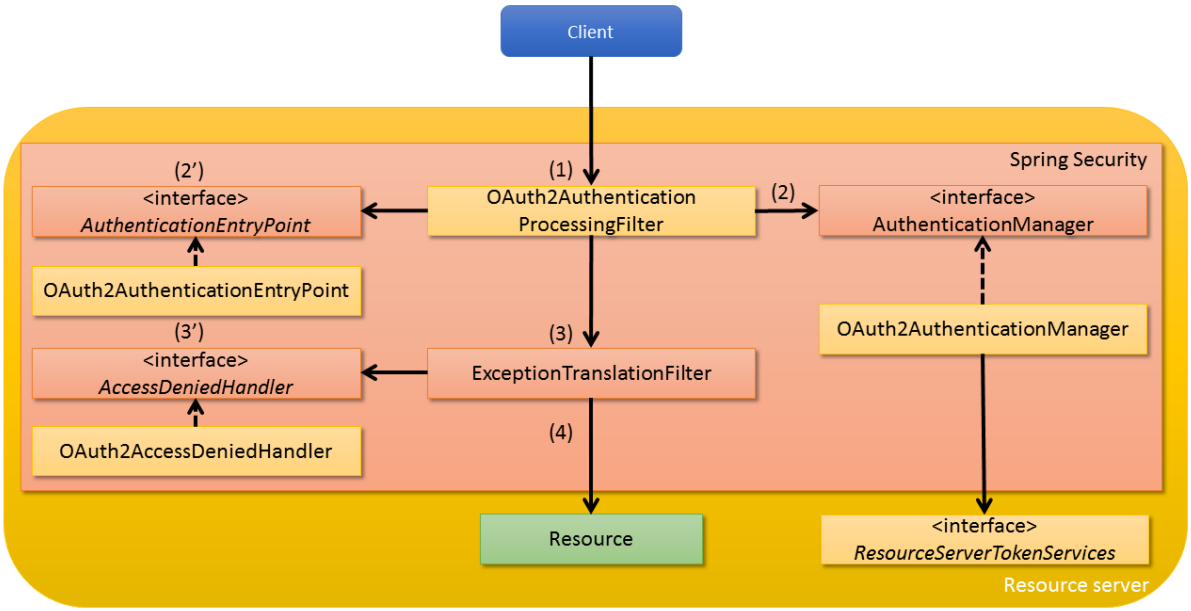


表 9.55: リソースサーバの動き

項番	説明
(1)	初めにクライアントがリソースサーバにアクセスすると `OAuth2AuthenticationProcessingFilter` の呼び出しが行われる。 `OAuth2AuthenticationProcessingFilter` ではリクエストよりアクセストークンを抽出する。
(2)	アクセストークンを抽出後、`AuthenticationManager` の実装である `OAuth2AuthenticationManager` において `ResourceServerTokenServices` のメソッドを呼び出しアクセストークンに紐づく認証情報を取得する。また、認証情報の取得時にアクセストークンを検証する。 アクセストークンに紐づく認証情報の取得方法は、認可サーバに対して HTTP による問い合わせを行うほか、認可サーバと `TokenStore` を共用して取得を行うなどの方法がある。 どのようにして認証情報の取得を行うかについては `ResourceServerTokenServices` の実装クラスに依存する。
(2』)	`OAuth2AuthenticationProcessingFilter` にて認証エラーが発生した場合、 `AuthenticationEntryPoint` の実装である `OAuth2AuthenticationEntryPoint` に処理を委譲し、エラー応答を行う。
(3)	`OAuth2AuthenticationProcessingFilter` の処理が完了した場合、次の Security Filter(`ExceptionTranslationFilter`) の呼び出しが行われる。 `ExceptionTranslationFilter` の詳細については 認可エラー時のレスポンスを参照のこと。
(3』)	`ExceptionTranslationFilter` にて例外をキャッチした場合、`AccessDeniedHandler` の実装である、`org.springframework.security.oauth2.provider.error.OAuth2AccessDeniedHandler` に処理を委譲し、エラー応答を 行う。
(4)	リクエストの認証・認可の検証に成功した場合、クライアントからのリクエストに応じたリソースを返却する。

クライアント

クライアントでは、リソースオーナからの認可を取得するために認可サーバへ誘導（リダイレクト）する機能と、アクセストークンを取得してリソースサーバへアクセスする機能を提供する。

Spring Security OAuth は、アクセストークンを取得してリソースサーバへアクセスするために利用する `OAuth2RestTemplate` や `org.springframework.security.oauth2.client.filter.OAuth2ClientContextFilter` を提供している。

`OAuth2RestTemplate` は `RestTemplate` を拡張し OAuth 2.0 向けの機能を追加したクラスで、リフレッシュトークンを使用してアクセストークンを再取得する仕組みや、アクセストークンを取得する際にリソースオーナから認可が必要な場合に例外（`org.springframework.security.oauth2.client.resource.UserRedirectRequiredException`）をスローする仕組みを備えている。なお、`OAuth2ClientContextFilter` をサーブレットフィルタに登録することで、`OAuth2RestTemplate` で発生した `UserRedirectRequiredException` をハンドリングして認可サーバへ誘導（リダイレクト）することが可能となる。

また、`OAuth2RestTemplate` では `org.springframework.security.oauth2.client.resource.OAuth2ProtectedResourceDetails` にて指定されたグラントタイプに沿って認可サーバより取得したアクセストークンを `org.springframework.security.oauth2.client.OAuth2ClientContext` の実装である `org.springframework.security.oauth2.client.DefaultOAuth2ClientContext` に保持する。`DefaultOAuth2ClientContext` はデフォルトではセッションスコープの Bean として定義され、複数のリクエスト間でアクセストークンを共有をすることが可能となる。

以下に、クライアント機能として `OAuth2RestTemplate` を使用した場合の構成を示す。

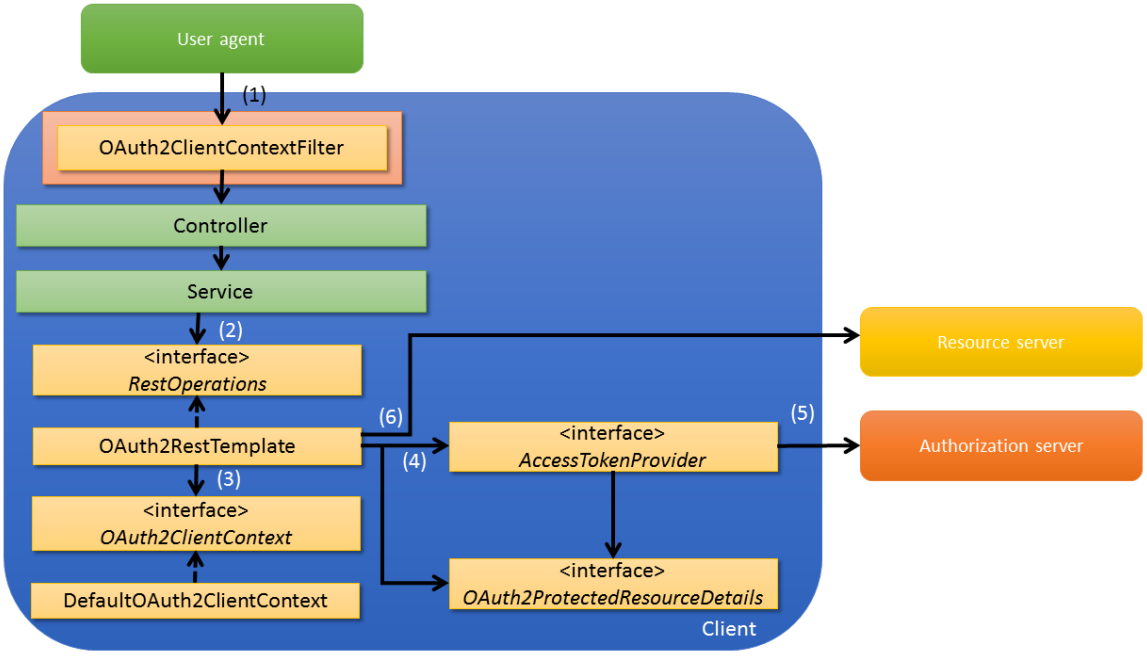


表 9.56: クライアントの動き

項番	説明
(1)	ユーザエージェントがクライアントの Service の呼び出しが行われるよう Controller へアクセスする。 リソースサーバへのアクセスを伴うアクセスに対しては、(5) で発生する可能性がある UserRedirectRequiredException を捕捉するためのサーブレットフィルタ（OAuth2ClientContextFilter）を適用する。 このサーブレットフィルタを適用することで、UserRedirectRequiredException が発生した際に、ユーザエージェントを認可サーバの認可エンドポイントへアクセスさせることができる。
(2)	Service より OAuth2RestTemplate の呼び出しを行う。
(3)	OAuth2ClientContext に保持しているアクセストークンを取得する。 有効なアクセストークンを取得できた場合は、(6) の処理に移る。
(4)	初回アクセス時などでアクセストークンを保持していなかった場合、または有効期限が超過していた場合、org.springframework.security.oauth2.client.token.AccessTokenProvider を呼び出しアクセストークンの取得を行う。
(5)	AccessTokenProvider では、リソースの詳細情報として OAuth2ProtectedResourceDetails に定義しているグラントタイプに応じてアクセストークンの取得を行う。 認可コードグラント向けの実装である org.springframework.security.oauth2.client.token.grant.code.AuthorizationCodeAccessTokenProvider では、認可コードの取得が完了していない場合、UserRedirectRequiredException をスローする。
(6)	(3) または (5) で取得したアクセストークンを指定して、リソースサーバへのアクセスを行う。 アクセス時にアクセストークンの有効期限切れ（org.springframework.security.oauth2.client.http.AccessTokenRequiredException）などの例外が発生した場合、保持しているアクセストークンを初期化した後、再度 (4) 以降の処理を行う。

注釈: インプリシットグラントは一般的に JavaScript などで実装されたクライアントで採用されるため、本ガイドラインでも JavaScript を用いて実装を行う方法を紹介する。

9.9.2 How to use

Spring Security OAuth を使用するために必要となる Bean 定義例や実装方法について説明する。

How to Use の構成

「認可グラント」で示したとおり、OAuth 2.0 ではグラントタイプにより認可サーバ、クライアント間のフローが異なる。そのため、アプリケーションがサポートするグラントタイプに沿った実装を行う必要がある。

本ガイドラインでは、グラントタイプごとに認可サーバ、リソースサーバ、クライアントの実装方法の解説を行う。

- **利用するグラントタイプに応じた読み進め方**　グラントタイプごとの実装方法については、まずは認可コードグラントで一通りの実装方法を解説し、その他のグラントタイプでは認可コードグラントからの変更点を解説する形式としている。**どのグラントタイプを利用する場合も、必ず認可コードグラントでの解説を一読した上で、利用するグラントタイプの解説を読み進められたい。**
- **作成するアプリケーションに応じた読み進め方**　認可サーバ、リソースサーバとクライアントの実装は独立しており、すべてのアプリケーションの実装方法を理解する必要はない。**いずれかのみ実装したい場合は、実装したいアプリケーションの解説のみ読み進められたい。**

Spring Security OAuth のセットアップ

Spring Security OAuth が提供している機能を使用するために、Spring Security OAuth を依存ライブラリとして追加する。

```
<!-- (1) -->
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

項番	説明
(1)	Spring Security OAuth を使用するプロジェクトの `pom.xml` に、Spring Security OAuth を依存ライブラリとして追加する。 リソースサーバ、認可サーバ、クライアントを別プロジェクトとして作成する場合、それぞれについて記述を追加すること。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

注釈: Spring Security OAuth におけるオープンリダイレクト脆弱性

Spring Security OAuth 2.3.4, 2.2.3, 2.1.3, 2.0.16 以前では、CVE-2019-3778 で報告されているオープンリダイレクト脆弱性が存在した。

具体的には、認可コードグラントを利用した認可サーバにおいて、オープンリダイレクト脆弱性により認可コードが漏洩する危険性があった。

TERASOLUNA Server Framework for Java version 5.4.2.RELEASE が利用する 2.0.17 では修正されており、脆弱性の影響を受けない。

認可コードグラントの実装

認可コードグラントを利用した認可サーバ、リソースサーバ、クライアントの実装方法について説明する。

認可サーバの実装

認可サーバの実装方法について説明する。

認可サーバでは、「リソースオーナからの認可の取得」「アクセストークンの発行」を行うためのエンドポイントを Spring Security OAuth の機能を使用して提供する。なお、上記のエンドポイントにアクセスする場合はリソースオーナまたはクライアントの認証が必要であり、本ガイドラインでは Spring Security の認証・認可の仕組みを使用して実現する。

設定ファイルの作成（認可サーバ）

認可サーバに関する定義を行うための設定ファイルとして `oauth2-auth.xml` を作成する。

`oauth2-auth.xml` では、認可サーバの機能を提供するためのエンドポイントの Bean 定義およびそれらのエンドポイントに対するセキュリティ設定、認可サーバのサポートするグラントタイプの設定を行う。

- oauth2-auth.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:sec="http://www.springframework.org/schema/security"
       xmlns:oauth2="http://www.springframework.org/schema/security/oauth2"
       xsi:schemaLocation="http://www.springframework.org/schema/security
           https://www.springframework.org/schema/security/spring-security.xsd
           http://www.springframework.org/schema/security/oauth2
           https://www.springframework.org/schema/security/spring-security-oauth2.
↪xsd
           http://www.springframework.org/schema/beans
           https://www.springframework.org/schema/beans/spring-beans.xsd">


</beans>
```

次に、作成した oauth2-auth.xml を読み込むように web.xml に設定を記述する。

- web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath*:META-INF/spring/applicationContext.xml
        classpath*:META-INF/spring/oauth2-auth.xml  <!-- (1) -->
        classpath*:META-INF/spring/spring-security.xml
    </param-value>
</context-param>
```

項番	説明
(1)	OAuth 2.0 用の Bean 定義ファイルの指定を行う。oauth2-auth.xml で設定したアクセス制御の対象の URL が spring-security.xml で設定したアクセス制御の対象の URL に含まれる場合を考慮し、spring-security.xml より先に記述すること。

認可サーバの定義

次に、認可サーバの定義を追加する。

- oauth2-auth.xml

```
<oauth2:authorization-server>  <!-- (1) -->
    <oauth2:authorization-code />  <!-- (2) -->
    <oauth2:refresh-token />  <!-- (3) -->
</oauth2:authorization-server>
```

項番	説明
(1)	<oauth2:authorization-server>タグを使用し、認可サーバの定義を行う。 <oauth2:authorization-server>タグを使用することで、認可を行うための認可エンドポイントと、アクセストークンを発行するためのトークンエンドポイントがコンポーネントとして登録される。 各コンポーネントにアクセスするため、以下のパスが設定される。 • 認可エンドポイント (/oauth/authorize) • トークンエンドポイント (/oauth/token) • リソースオーナから認可を取得する際のフォワード先 (/oauth/confirm_access) • 認可エンドポイントで不正クライアントエラーが発生した場合のフォワード先 (/oauth/error)
(2)	<oauth2:authorization-code />タグを使用して、認可コードグラントをサポートする。
(3)	<oauth2:refresh-token />タグを使用して、リフレッシュトークンをサポートする。

警告: <oauth2:authorization-code />タグと<oauth2:refresh-token />タグは上記の順番で設定する必要がある。

詳細は認可サーバで複数のグラントタイプをサポートする場合を参照されたい。

注釈: <oauth2:authorization-server>タグを使用することで登録されるエンドポイント、およびフォワード先のパスはそれぞれカスタマイズすることが可能である。

詳細はエンドポイントのカスタマイズ、フォワード先のカスタマイズを参照されたい。

注釈: 上記の設定ファイルは client-details-service-ref パラメータの設定を行っていないため、IDE によっては文法誤りによるエラーが検知されることがある。後述する設定を追加することでエラーは解消される。

注釈: 認可コードは、認可コードが発行されてからアクセストークンの発行までの短い期間しか使われないため、デフォルトではインメモリで管理される。認可サーバが複数台構成の場合は、複数サーバ間で認可コードを共有するために DB で管理する必要がある。認可コードを DB で管理する場合は、主キーとなる認可コードを保持するカラムと、認証情報を保持するカラムによって構成された以下のようなテーブルを作成する。以下の例は PostgreSQL を使用した場合の DB 定義である。

oauth_code

code	VARCHAR(256)
authentication	BYTEA

認可サーバの設定ファイルには、<oauth2:authorization-code />タグの authorization-code-services-ref に、認可コードを DB 管理する org.springframework.security.oauth2.provider.code.JdbcAuthorizationCodeServices の Bean 名を指定する。JdbcAuthorizationCodeServices のコンストラクタには、認可コード格納用のテーブルに接続するためのデータソースを指定する。認可コードを DB にて永続管理する場合の注意点についてはトランザクション制御を **必ず** 参照のこと。

- oauth2-auth.xml

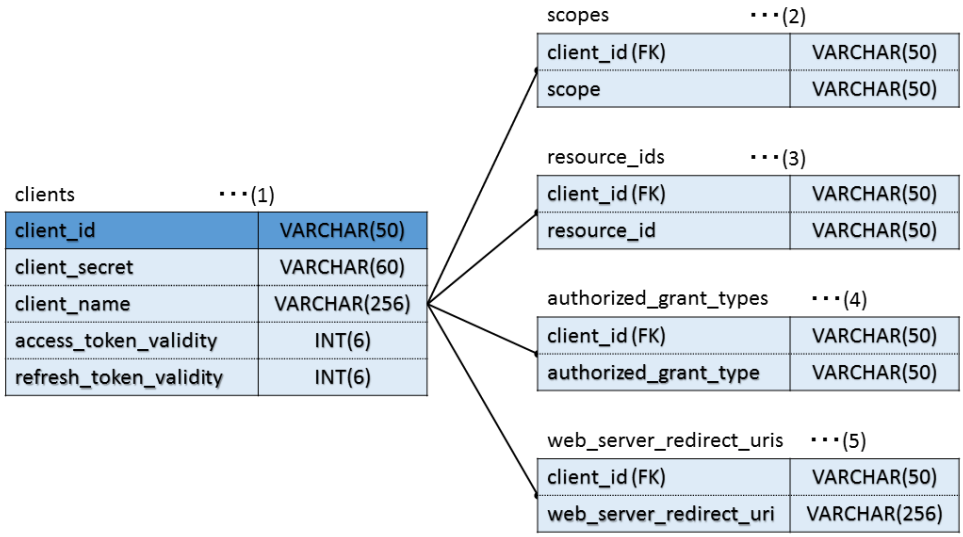
```
<oauth2:authorization-server>
    <oauth2:authorization-code authorization-code-services-ref=
↪"authorizationCodeServices"/>
    <!-- omitted -->
</oauth2:authorization-server>

<bean id="authorizationCodeServices"
      class="org.springframework.security.oauth2.provider.code.
↪JdbcAuthorizationCodeServices">
    <constructor-arg ref="dataSource"/>
</bean>
```

クライアントの認証

エンドポイントに対してアクセスしてきたクライアントについては、登録済みのクライアントか確認するために認証を行う必要がある。クライアントの認証は、クライアントよりパラメータで渡されたクライアント ID とパスワードを、認可サーバで保持しているクライアント情報をもとに検証することで行う。認証には Basic 認証を用いて行う。

Spring Security OAuth ではクライアント情報を取得するためのインタフェースである `ClientDetailsService` の実装クラスを提供している。また、クライアント情報を保持するためのクラスとして `org.springframework.security.oauth2.provider.ClientDetails` インタフェースの実装クラスである `org.springframework.security.oauth2.provider.client.BaseClientDetails` クラスを提供している。`BaseClientDetails` ではクライアント ID やサポートするグラントタイプなどの OAuth 2.0 を利用する上での基本的なパラメータを提供しており、`BaseClientDetails` を拡張することで独自のパラメータを追加することも可能である。ここでは `BaseClientDetails` の拡張と `ClientDetailsService` の実装クラス作成を行い、独自パラメータとして クライアント名 を追加したクライアント情報を DB を用いて管理、および認証を行う場合の実装方法について説明する。

まず、以下のような DB を作成する。



項番	説明
(1)	クライアント情報を保持するテーブル。client_id を主キーとする。 各カラムの役割は以下のとおりである。 • client_id：クライアントを識別する ID であるクライアント ID を保持するカラム。 • client_secret：クライアントの認証に使用するパスワードを保持するカラム。 • client_name：クライアント名を保持する独自カラム。独自カラムであるため必須ではない。 • access_token_validity：アクセストークンの有効期間 [秒] を保持するカラム。 • refresh_token_validity：リフレッシュトークンの有効期間 [秒] を保持するカラム。
(2)	スコープ情報を保持するテーブル。client_id を外部キーとし、クライアント情報と対応付けする。 scope カラムに、クライアント認可に使用するスコープを保持する。クライアントがもつスコープの数だけレコードを登録する。
(3)	リソース情報を保持するテーブル。client_id を外部キーとし、クライアント情報と対応付けする。 resource_id カラムに、クライアントのアクセス可能なリソースかどうかを、リソースサーバが識別するために使用するリソース ID を保持する。 リソースサーバが保持するリソースに対して定義しているリソース ID がここで登録されているリソース ID に含まれる場合のみ、リソースへのアクセスを許可される。 クライアントがアクセス可能なリソース ID の数だけレコードを登録する。 リソース ID を一件も登録しなかった場合は、全てのリソースに対してアクセス可能となるため、登録しない場合は注意が必要である。
(4)	グラント情報を保持するテーブル。client_id を外部キーとし、クライアント情報と対応付けする。 authorized_grant_type カラムに、クライアントの使用するグラントを保持する。 クライアントが利用するグラントの数だけレコードを登録する。

Continued on next page

表 9.60 – continued from previous page

項番	説明
(5)	リダイレクト URI 情報を保持するテーブル。client_id を外部キーとし、クライアント情報と対応付けする。 web_server_redirect_uri カラムに、リソースオーナによる認可後にユーザエージェントをリダイレクトさせる URI を保持する。 保持するリダイレクト URI は、クライアントが認可リクエスト時に申告するリダイレクト URI のホワイトリストチェックで使用される。 クライアントが申告する可能性のある URI の数だけレコードを登録する。 **警告:** Spring Security OAuth 2.0.17 より、リダイレクト URI のホワイトリストチェックの仕様が変更された。Spring Security OAuth 2.0.16 以前では、認可サーバに登録されたリダイレクト URI のパス配下であることを確認していたが、2.0.17 ではパスと完全一致することを確認するようになった。これにより、認可サーバにリダイレクトするすべてのパスを指定しなければならなくなったことに注意されたい。

クライアント情報を保持するモデルを作成する。

- Client.java

```
public class Client implements Serializable{
    private String clientId; // (1)
    private String clientSecret; // (2)
    private String clientName; // (3)
    private Integer accessTokenValidity; // (4)
    private Integer refreshTokenValidity; // (5)
    // Getters and Setters are omitted
}
```

項番	説明
(1)	クライアントを識別するクライアント ID を保持するフィールド。

Continued on next page

表 9.61 – continued from previous page

項番	説明
(2)	クライアントの認証に使用するパスワードを保持するフィールド。
(3)	Spring Security OAuth では提供されていない、クライアント名を保持する拡張フィールド。 拡張フィールドであるため必須ではない。
(4)	アクセストークンの有効期間 [秒] を保持するフィールド。
(5)	リフレッシュトークンの有効期間 [秒] を保持するフィールド。

注釈: クライアント情報を操作するための Repository クラスを作成する必要があるが、ここでは説明を割愛する。具体的な実装方法については *Repository の実装*を参照されたい。

BaseClientDetails クラスを継承させたクラスを作成することで、クライアントの詳細情報を簡単に拡張することができる。

- OAuthClientDetails.java

```
public class OAuthClientDetails extends BaseClientDetails{
    private Client client;
    // Getter and Setter are omitted
}
```

org.springframework.security.oauth2.provider.ClientDetailsService は、認可処理で必要となるクライアント詳細情報をデータストアから取得するためのインタフェースである。以下では、ClientDetailsService の実装クラスの作成について説明する。

- OAuthClientDetailsService.java

```
@Service("clientDetailsService") // (1)
@Transactional
public class OAuthClientDetailsService implements ClientDetailsService {

    @Inject
```

```
    ClientRepository clientRepository;

    @Override
    public ClientDetails loadClientByClientId(String clientId)
            throws ClientRegistrationException {

        Client client = clientRepository.findClientByClientId(clientId); // (2)

        if (client == null) { // (3)
              throw new NoSuchClientException("No client with requested id: " +␣
↪clientId);
        }

        // (4)
        Set<String> clientScopes = clientRepository.
↪findClientScopesByClientId(clientId);
        Set<String> clientResources = clientRepository.
↪findClientResourcesByClientId(clientId);
        Set<String> clientGrants = clientRepository.
↪findClientGrantsByClientId(clientId);
        Set<String> clientRedirectUris = clientRepository.
↪findClientRedirectUrisByClientId(clientId);

        // (5)
        OAuthClientDetails clientDetails = new OAuthClientDetails();
        clientDetails.setClientId(client.getClientId());
        clientDetails.setClientSecret(client.getClientSecret());
        clientDetails.setAccessTokenValiditySeconds(client.
↪getAccessTokenValidity());
        clientDetails.setRefreshTokenValiditySeconds(client.
↪getRefreshTokenValidity());
        clientDetails.setResourceIds(clientResources);
        clientDetails.setScope(clientScopes);
        clientDetails.setAuthorizedGrantTypes(clientGrants);
        clientDetails.setRegisteredRedirectUri(clientRedirectUris);
        clientDetails.setClient(client);

        return clientDetails;
    }

}
```

項番	説明
(1)	Service として component-scan の対象とするため、クラスレベルに`@Service` アノテーションをつける。 Bean 名を `clientDetailsService` として指定する。

Continued on next page

表 9.62 – continued from previous page

項番	説明
(2)	DB からクライアント情報を取得する。
(3)	クライアント情報が見つからない場合は、Spring Security OAuth の例外である `org.springframework.security.oauth2.provider.NoSuchClientException` を発生させる。 なお、認可エンドポイントでもクライアント情報の取得のため本処理が呼び出されるが、`NoSuchClientException` が発生した場合は認可エンドポイントによってハンドリングされ、`org.springframework.security.oauth2.common.exceptions.OAuth2Exception` のサブクラスである `org.springframework.security.oauth2.common.exceptions.BadClientCredentialsException` がスローされる。 認可エンドポイントで `OAuth2Exception` が発生した場合のハンドリング方法については認可リクエスト時のエラー画面のカスタマイズを参照されたい。
(4)	クライアントに紐付く情報を取得する。 複数回にわけて Repository の呼び出しを行うことにより処理性能が落ちるような場合は (2) で一括取得する。
(5)	取得した各種情報を `OAuthClientDetails` のフィールドに設定する。

`oauth2-auth.xml` にクライアント認証に必要な設定を追記する。

- `oauth2-auth.xml`

```
<oauth2:authorization-server
     client-details-service-ref="clientDetailsService">  <!-- (1) -->
    <oauth2:authorization-code />
    <oauth2:refresh-token />
</oauth2:authorization-server>

<sec:http pattern="/oauth/*token*/**"
    authentication-manager-ref="clientAuthenticationManager">  <!-- (2) -->
    <sec:http-basic entry-point-ref="oauthAuthenticationEntryPoint" />           <!
↪-- (3) -->
    <sec:csrf disabled="true"/>  <!-- (4) -->
    <sec:intercept-url pattern="/**" access="isAuthenticated()"/>  <!-- (5) -->
    <sec:access-denied-handler ref="oauth2AccessDeniedHandler"/>  <!-- (6) -->
</sec:http>
```

```
<sec:authentication-manager id="clientAuthenticationManager">  <!-- (7) -->
    <sec:authentication-provider user-service-ref="clientDetailsUserService" >  <!-
↪- (8) -->
        <sec:password-encoder ref="passwordEncoder"/>  <!-- (9) -->
    </sec:authentication-provider>
</sec:authentication-manager>

<bean id="oauthAuthenticationEntryPoint" class="org.springframework.security.
↪oauth2.provider.error.OAuth2AuthenticationEntryPoint">
    <property name="typeName" value="Basic" />  <!-- (10) -->
    <property name="realmName" value="Realm" />  <!-- (11) -->
</bean>

<bean id="oauth2AccessDeniedHandler" class="org.springframework.security.oauth2.
↪provider.error.OAuth2AccessDeniedHandler" />  <!-- (12) -->

<bean id="clientDetailsUserService"
    class="org.springframework.security.oauth2.provider.client.
↪ClientDetailsUserDetailsService">  <!-- (13) -->
    <constructor-arg ref="clientDetailsService" />  <!-- (14) -->
</bean>
```

項番	説明
(1)	client-details-service-ref 属性に OAuthClientDetailsService の Bean を指定する。 指定する Bean 名は、ClientDetailsService の実装クラスで指定した Bean 名と合わせる必要がある。
(2)	アクセストークン操作に関するエンドポイントへのセキュリティ設定を行うために、エンドポイントとして/oauth/*token*/配下をアクセス制御の対象として指定する。Spring Security OAuth により定義されているエンドポイント、およびそのデフォルト値は以下のとおりである。 • トークン払い出しに使用するトークンエンドポイント (/oauth/token) • トークンを検証するチェックトークンエンドポイント (/oauth/check_token) • JWT の署名を公開鍵暗号方式で作成した場合に、公開鍵を取得するために使用するエンドポイント (/oauth/token_key) authentication-manager-ref 属性に (7) で定義しているクライアント認証用の AuthenticationManager の Bean を指定する。

Continued on next page

表 9.63 – continued from previous page

項番	説明
(3)	クライアント認証に Basic 認証を適用する。 詳細については Basic and Digest Authentication を参照されたい。
(4)	`/oauth/*token*/**`へのアクセスに対して CSRF 対策機能を無効化する。 Spring Security OAuth では、OAuth 2.0 の CSRF 対策として推奨されている、state パラメータを使用したリクエストの正当性確認を採用している。
(5)	エンドポイント配下に対して、認証済みユーザのみがアクセスできる権限を付与する設定。 Web リソースに対してアクセスポリシーの指定方法については、認可を参照されたい。
(6)	`access-denied-handler` には `OAuth2AccessDeniedHandler` の Bean を設定する。ここでは (12) で定義している `oauth2AccessDeniedHandler` の Bean を指定する。
(7)	クライアントを認証するための `AuthenticationManager` を Bean 定義する。 リソースオーナの認証で使用する `AuthenticationManager` と別名の Bean 名を指定する必要がある。 ここでは、`sec:authentication-manager` を複数定義する必要があるため、それぞれに id 属性を用いて Bean 名を付与する。 一つのコンテキストに `sec:authentication-manager` が一つしかない場合は、alias 属性を用いて Bean 名を付与しても良い。 リソースオーナの認証についてはリソースオーナの認証を参照されたい。
(8)	`sec:authentication-provider` の `user-service-ref` 属性に (13) で定義している `org.springframework.security.oauth2.provider.client.ClientDetailsUserDetailsService` の Bean を指定する。
(9)	クライアントの認証に使用するパスワードのハッシュ化に使用する `PasswordEncoder` の Bean を指定する。 パスワードハッシュ化の詳細については パスワードのハッシュ化を参照されたい。

Continued on next page

表 9.63 – continued from previous page

項番	説明
(10)	クライアントの認証に失敗した場合に、レスポンスヘッダでクライアントに提示するクライアント認証の認証スキームを指定する。
(11)	クライアントの認証に失敗した場合に、レスポンスヘッダでクライアントに提示するクライアント認証のレルムを指定する。 クライアント認証のレルムをカスタマイズしたい場合に設定する。 指定しない場合はデフォルト値である `oauth` が設定される。
(12)	Spring Security OAuth が提供する OAuth 2.0 用の `AccessDeniedHandler` を定義する。 `OAuth2AccessDeniedHandler` は、認可エラー時に発生する例外をハンドリングしてエラー応答を行う。
(13)	`UserDetailsService` インタフェースの実装クラスである `ClientDetailsUserDetailsService` を Bean 定義する。 リソースオーナの認証で使用する `UserDetailsService` と別名の Bean 名を指定する必要がある。
(14)	コンストラクタの引数に、DB からクライアント情報を取得する `OAuthClientDetailsService` の Bean を指定する。 指定する Bean 名は、`ClientDetailsService` の実装クラスで指定した Bean 名と合わせる必要がある。

リソースオーナの認証

アクセストークンの取得に認可コードグラントを用いる場合、ログイン画面を用意する等、なんらかの方法でリソースオーナを認証する必要がある。

本ガイドラインでは、リソースオーナの認証に Spring Security を利用する前提とする。

認可の設定には、認証済みユーザのみ認可エンドポイント URL へアクセスできるよう、認可エンドポイント URL を含んだ URL をアクセスポリシーとして定義する必要がある。また、リソースオーナから認可を取得す

る際のフォワード先と、認可エンドポイントで不正クライアントエラーが発生した場合のフォワード先も同様にアクセスポリシーとして定義する必要がある。

リソースオーナから認可を取得する際のフォワードをハンドリングするコントローラについては スコープ認可画面のカスタマイズを、認可エンドポイントでの不正クライアントエラーをハンドリングするコントローラについては 認可リクエスト時のエラー画面のカスタマイズを参照されたい。

Spring Security の詳細については 認証及び 認可を参照されたい。

以下に認可エンドポイント、リソースオーナから認可を取得する際のフォワード先、認可エンドポイントで例外が発生した場合のフォワード先を含んだアクセスポリシーの定義例を示す。

- spring-security.xml

```
<sec:http authentication-manager-ref="authenticationManager"> <!-- (1) -->
    <sec:form-login login-page="/login"
        authentication-failure-url="/login?error=true"
        login-processing-url="/login" />
    <sec:logout logout-url="/logout"
        logout-success-url="/"
        delete-cookies="JSESSIONID" />
    <sec:access-denied-handler ref="accessDeniedHandler"/>
    <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
    <sec:session-management />
    <sec:intercept-url pattern="/login/**" access="permitAll" />
    <sec:intercept-url pattern="/oauth/**" access="isAuthenticated()" /> <!-- (2) -
↪->
    <!-- omitted -->
</sec:http>

 <sec:authentication-manager id="authenticationManager"> <!-- (3) -->
    <sec:authentication-provider
        user-service-ref="userDetailsService">
        <sec:password-encoder ref="passwordEncoder" />
    </sec:authentication-provider>
</sec:authentication-manager>

<bean id="userDetailsService"
    class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
    <property name="dataSource" ref="dataSource" />
</bean>
```

項番	説明
(1)	認可サーバにフォーム認証を適用し、authentication-manager-ref 属性に (3) で定義している authenticationManager を指定する。 oauth2-auth.xml でも AuthenticationManager を定義しているため、別名の Bean 名を指定する必要がある。
(2)	以下を含んだパス (/oauth/) 配下を認証済みユーザのみがアクセスできるよう指定する。 • 認可エンドポイント (/oauth/authorize) • リソースオーナから認可を取得する際のフォワード先 (/oauth/confirm_access) • 認可エンドポイントで不正クライアントエラーが発生した場合のフォワード先 (/oauth/error)
(3)	リソースオーナを認証するための authenticationManager を Bean 定義する。 ここでは、sec:authentication-manager を複数定義する必要があるため、それぞれに id 属性を用いて Bean 名を付与する。 一つのコンテキストに sec:authentication-manager が一つしかない場合は、alias 属性を用いて Bean 名を付与しても良い。

スコープごとの認可

リソースオーナからの認可の取得時に、要求されたスコープを一括で認可するのではなく、各スコープを個別に認可する場合の設定方法を説明する。

認可サーバを再起動した際に認可情報を失わないよう永続管理するために、また複数台の認可サーバで認可情報を共有するためには、認可情報を DB で管理する必要がある。スコープごとに認可情報を格納するための DB として、以下の DB を作成する。以下の例では PostgreSQL を使用した場合の DB 定義を説明する。

oauth_approvals ・・・(1)

userId	VARCHAR(50)
clientId	VARCHAR(50)
scope	VARCHAR(50)
status	VARCHAR(10)
expiresAt	TIMESTAMP
lastModifiedAt	TIMESTAMP

項番	説明
(1)	認可情報を保持するテーブル。userId、clientId、scope を主キーとする。 各カラムの役割は以下のとおりである。 • userId：認可を行ったリソースオーナのユーザ名を保持するカラム。 • clientId：リソースオーナによって認可されたクライアントのクライアント ID を保持するカラム。 • scope：リソースオーナに認可されたスコープを保持するカラム。 • status：リソースオーナに認可されたかどうかを保持するカラム。認可された場合は APPROVED、拒否された場合は DENIED が設定される。 • expiresAt：認可情報の有効期限を保持するカラム。 • lastModifiedAt：認可情報が最後に更新された日時を保持するカラム。

リソースオーナからスコープ毎の認可を取得し、DB に保存して管理するための設定を行う。

実装例は以下のとおりである。

- oauth2-auth.xml

```
<oauth2:authorization-server
     client-details-service-ref="clientDetailsService"
     user-approval-handler-ref="userApprovalHandler"> <!-- (1) -->

     <!-- omitted -->

</oauth2:authorization-server>

<!-- omitted -->

<bean id="userApprovalHandler"
      class="org.springframework.security.oauth2.provider.approval.
↪ApprovalStoreUserApprovalHandler">  <!-- (2) -->
    <property name="clientDetailsService" ref="clientDetailsService"/>
    <property name="approvalStore" ref="approvalStore"/>
    <property name="requestFactory" ref="requestFactory"/>
    <property name="approvalExpiryInSeconds" value="3200" />
</bean>

<bean id="approvalStore"
      class="org.springframework.security.oauth2.provider.approval.
↪JdbcApprovalStore">  <!-- (3) -->
    <constructor-arg ref="dataSource"/>
</bean>

<bean id="requestFactory"
      class="org.springframework.security.oauth2.provider.request.
↪DefaultOAuth2RequestFactory">
    <constructor-arg ref="clientDetailsService"/>
</bean>
```

項番	説明
(1)	スコープの認可処理を行う `UserApprovalHandler` として、 `user-approval-handler-ref` に (2) で定義している `ApprovalStoreUserApprovalHandler` の Bean を指定する。

(2)	スコープの認可処理を行う `ApprovalStoreUserApprovalHandler` を Bean 定義する。
	リソースオーナの認可結果を管理する `approvalStore` プロパティには、(3) で定義している
	`org.springframework.security.oauth2.provider.approval.`
	`JdbcApprovalStore` の Bean を指定
	する。
	スコープの認可処理に使用するクライアント情報の取得をする `clientDetailsService` プロパティには、`OAuthClientDetailsService` の Bean を指定する。
	`requestFactory` プロパティには、`org.springframework.security.oauth2.`
	`provider.request.DefaultOAuth2RequestFactory` の Bean を指定
	する。
	`requestFactory` プロパティに設定した Bean は `ApprovalStoreUserApprovalHandler` によって使用されないが、設定を行っていない場合は
	`ApprovalStoreUserApprovalHandler` の Bean 生成時にエラーとなるため、
	`requestFactory` プロパティへの設定が必要である。
	認可情報の有効期間 [秒] を指定する場合は、`approvalExpiryInSeconds` プロパティに、有効期間 [秒] を設定する。設定を行わない場合は、認可情報は認可から一ヶ月間有効となる。

(3)	認可情報を DB で管理する `JdbcApprovalStore` を Bean 定義する。 コンストラクタには、認可情報格納用のテーブルに接続するためのデータソースを指定する。 データソースの設定方法については、データソースの設定を参照されたい。 **警告:** 認可情報を DB にて永続管理する場合の注意点についてはトランザクション制御を **必ず** 参照のこと。 **注釈:** 認可情報を永続管理する必要がなく、DB ではなくインメモリで管理したい場合は、`approvalStore` として `org.springframework.security.oauth2.provider.approval.InMemoryApprovalStore` を Bean 定義すればよい。

スコープ認可画面のカスタマイズ

スコープ認可画面をカスタマイズしたい場合、コントローラと JSP を作成することでカスタマイズできる。以下ではスコープ認可画面のカスタマイズした場合の例を説明する。

リソースオーナからの認可を取得するためにエンドポイントの呼び出しを行う場合、/oauth/confirm_access にフォワードされる。/oauth/confirm_access をハンドリングするコントローラを作成する。

- OAuth2ApprovalController.java

```
@Controller
public class OAuth2ApprovalController {

    @RequestMapping("/oauth/confirm_access") // (1)
    public String confirmAccess() {
        // omitted
        return "approval/oauthConfirm";
    }

}
```

項番	説明
(1)	@RequestMapping アノテーションを使用して、/oauth/confirm_access へのアクセスに対するメソッドとしてマッピングを行う。

次に、スコープ認可画面の JSP を作成する。認可対象のスコープは scopes キーとして、リクエストパラメータは authorizationRequest として Model に登録されているため、これを利用してスコープ認可画面を表示する。

- oauthConfirm.jsp

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<body>
    <div id="wrapper">
        <h1>OAuth Approval</h1>
        <p>Do you authorize ${f:h(authorizationRequest.clientId)} to access your protected resources?</p>  <!-- (1) -->
        <form id="confirmationForm" name="confirmationForm" action="${pageContext.request.contextPath}/oauth/authorize" method="post">
            <c:forEach var="scope" items="${scopes}">  <!-- (2) -->
                <li>
                    ${f:h(scope.key)}
                    <input type="radio" name="${f:h(scope.key)}" id="${f:h(scope.key)}_approve" value="true"/><label for="${f:h(scope.key)}_approve">Approve</label>
                    <input type="radio" name="${f:h(scope.key)}" id="${f:h(scope.key)}_deny" value="false"/><label for="${f:h(scope.key)}_deny">Deny</label>
```

```
                </li>
            </c:forEach>
            <input type="hidden" name="user_oauth_approval" value="true"/>  <!--␣
↪(3) -->
            <sec:csrfInput />  <!-- (4) -->
            <label>
                <input type="submit" id="authorize" value="Authorize"/>
            </label>
        </form>
    </div>
</body>
```

項番	説明
(1)	クライアント ID を画面に表示させる。 authorizationRequest の型は org.springframework.security.oauth2.provider.AuthorizationRequest であり、クライアント ID は authorizationRequest.clientId と指定することで出力される。
(2)	スコープ毎に認可可否を指定するためのラジオボタンを出力する。対象のスコープは scopes に含まれるため、items に scopes を指定する。 scopes の型は LinkedHashMap であり、スコープ名をキーとして 、そのスコープに対する認可可否の情報を保持している。
(3)	user_oauth_approval を hidden 項目として埋め込むことで、Spring Security OAuth がリクエストパラメータに user_oauth_approval を付与する。 リクエストパラメータに付与された user_oauth_approval は、認可エンドポイントのスコープ認可を行うメソッドを実行するために用いられる。
(4)	CSRF トークン値を引き渡すために、HTML の<form>属性の中に<sec:csrfInput />属性を指定する。

認可リクエスト時のエラー画面のカスタマイズ

認可エンドポイントで不正クライアントエラー（クライアント未存在エラー等のセキュリティに関わるエラーや、リダイレクト URI チェックエラー）が発生した場合、Spring Security OAuth が提供する OAuth2Exception がスローされ、リクエストは/oauth/error にフォワードされる。そのため/oauth/error をハンドリングするコントローラを作成する必要がある。不正クライアントエラーの詳細については不正クライアントエラーを参照されたい。

以下にコントローラの実装例を示す。

- OAuth2ErrorController.java

```
@Controller
public class OAuth2ErrorController {

    @RequestMapping("/oauth/error") // (1)
    public String handleError() {
        // omitted
        return "common/error/oauthError";
    }

}
```

項番	説明
(1)	@RequestMapping アノテーションを使用して、/oauth/error へのアクセスに対するメソッドとしてマッピングを行う。

次に、表示させるエラー画面の JSP を作成する。認可エンドポイントで発生した例外が Model の属性名 error に登録されているため、例外の内容を画面に表示する。

- oauthError.jsp

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>OAuth Error!</title>
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/css/
↪styles.css">
</head>
<body>
    <div id="wrapper">
        <h1>OAuth Error!</h1>
        <c:if test="${not empty error}">
            <p>${f:h(error.oAuth2ErrorCode)}</p> <!-- (1) -->
            <p>${f:h(error.message)}</p> <!-- (2) -->
        </c:if>
    <br>
    </div>
</body>
</html>
```

項番	説明
(1)	例外に設定されている OAuth 2.0 のエラーコードを出力する。 エラーコードは、`invalid_request`、`invalid_client` といった値を持つ。
(2)	例外に設定されているエラーメッセージを出力する。

注釈: 認可エンドポイントで発生したエラーが、不正クライアントエラー以外の場合、クライアントの呼び出し元コントローラにリダイレクトすることでクライアント側にエラー通知を行う。エラー通知を行う際のエラー情報はリダイレクト URI にリクエストパラメータとして付与される。エラーハンドリングについては認可エンドポイントアクセス時のエラーハンドリングを参照されたい。

ちなみに: アプリケーションで Internet Explorer/Microsoft Edge をサポートする場合、エラー画面の応答として生成される HTML のサイズに注意する必要がある。

Internet Explorer/Microsoft Edge では、応答された HTML のサイズが規定値以下だと、アプリケーションが用意したエラー画面の代わりに、Internet Explorer/Microsoft Edge が用意した簡易メッセージが表示されるためである。

参考までに、Internet Explorer での詳細な条件は、「Friendly HTTP Error Pages」を参照されたい。

リソースサーバへのアクセストークンの連携方法

リソースサーバがアクセストークンを元にリソースへのアクセスに対する認可判定を行えるよう、認可サーバは `AuthorizationServerTokenServices` を介してアクセストークンを連携する。連携方法は以下に示すとおり複数存在する。

項番	連携方法	説明
(1)	DB を介した連携	共有 DB を利用し、アクセストークンを連携する方法。 リソースサーバと認可サーバが DB を共有している場合に利用可能。 認可サーバは `AuthorizationServerTokenServices` の実装クラスとして `DefaultTokenServices` を、`TokenStore` として `org.springframework.security.oauth2.provider.token.store.JdbcTokenStore` を指定する。

Continued on next page

表 9.71 – continued from previous page

項番	連携方法	説明
(2)	HTTP アクセスを介した連携	HTTP アクセスにより、アクセストークンを連携する方法。 リソースサーバと認可サーバが共有 DB を利用できない場合に、この方法を利用する。 リソースサーバはアクセストークンの取得及び検証を認可サーバに依頼するため、認可サーバに負荷がかかる。 認可サーバは `AuthorizationServerTokenServices` の実装クラスとして `DefaultTokenServices` を指定する。 アクセストークンを DB で管理する場合は `JdbcTokenStore` を、メモリで管理する場合は `org.springframework.security.oauth2.provider.token.store.InMemoryTokenStore` を `TokenStore` として指定する。 アクセストークンをメモリで管理する実装はサーバ再起動などでアクセストークンが失われるため、テスト用途専用の実装である。
(3)	JWT を利用した連携	JWT を利用し、アクセストークンを連携する方法。 リソースサーバと認可サーバが共有 DB を利用できない場合に、この方法を利用する。 HTTP アクセスを介した連携と比べ、認可サーバにアクセストークンの取得を依頼しないため、認可サーバへの負荷がかからない。 認可サーバは `AuthorizationServerTokenServices` の実装クラスとして `DefaultTokenServices` を、`TokenStore` として `org.springframework.security.oauth2.provider.token.store.JwtTokenStore` を指定する。 `org.springframework.security.oauth2.provider.token.store.JwtAccessTokenConverter` を利用することでアクセストークンの署名とエンコード、デコードを行う。 アクセストークンの署名とその検証には公開鍵を使用する方法と、共通鍵を使用する方法がある。

Continued on next page

表 9.71 – continued from previous page

項番	連携方法	説明
(4)	メモリを介した連携	メモリを共有することで、アクセストークンを連携する方法。 リソースサーバと認可サーバが一つのプロセスとなるようアプリケーションを設計している場合に利用可能。 認可サーバは `AuthorizationServerTokenServices` の実装クラスとして `DefaultTokenServices` を、`TokenStore` として `InMemoryTokenStore` を指定する。 メモリを介して連携させるため、共有 DB や HTTP アクセスによるアクセストークンの連携が不要となる。 メモリを介してアクセストークンを共有する実装はサーバ再起動などでアクセストークンが失われるため、テスト用途専用の実装である。

ここでは、代表的な連携方法として共有 DB を介して連携させる方法を説明する。HTTP アクセスを介した連携ついては本節の How To Extend にて説明している。

- *HTTP アクセスを介した認可サーバとリソースサーバの連携*

共有 DB を介して連携させる場合、Spring Security OAuth が提供している `JdbcTokenStore` を使用する。

実装例は以下のとおりである。

- `oauth2-auth.xml`

```
<oauth2:authorization-server
     client-details-service-ref="clientDetailsService"
     user-approval-handler-ref="userApprovalHandler"
     token-services-ref="tokenServices">  <!-- (1) -->
    <oauth2:authorization-code />
    <oauth2:refresh-token />
</oauth2:authorization-server>

<!-- omitted -->

<bean id="tokenServices"
    class="org.springframework.security.oauth2.provider.token.DefaultTokenServices
↪">  <!-- (2) -->
    <property name="tokenStore" ref="tokenStore" />
    <property name="clientDetailsService" ref="clientDetailsService" />
    <property name="supportRefreshToken" value="true" />  <!-- (3) -->
</bean>
```

```
<bean id="tokenStore"
  class="org.springframework.security.oauth2.provider.token.store.JdbcTokenStore">
↪<!-- (4) -->
  <constructor-arg ref="dataSource" />
</bean>
```

項番	説明
(1)	認可サーバが使用する TokenService として token-services-ref 属性に (2) で定義している tokenServices を指定する。
(2)	tokenServices のクラスに DefaultTokenServices を指定する。 アクセストークンを管理するトークンストアとして、tokenStore プロパティに (4) で定義している TokenStore を指定する。 共有 DB を介してリソースサーバとアクセストークンの連携を行う場合、リソースサーバでも本設定を行うこと。
(3)	リフレッシュトークンを有効にする場合は supportRefreshToken 属性に true を指定する。
(4)	トークンストアとして JdbcTokenStore を Bean 定義する。 コンストラクタには、トークン情報格納用のテーブルに接続するためのデータソースを指定する。 データソースの設定方法については、データソースの設定を参照されたい。

JdbcTokenStore がアクセストークンを連携するために、Spring Security OAuth がスキーマ定義している以下の DB を作成する。以下の例では共有 DB として PostgreSQL を使用した場合の DB 定義を説明する。

oauth_access_token ・・・(1)

authentication_id	VARCHAR(256)
token	BYTEA
token_id	VARCHAR(256)
user_name	VARCHAR(50)
client_id	VARCHAR(50)
authentication	BYTEA
refresh_token	VARCHAR(256)

oauth_refresh_token ・・・(2)

token_id	VARCHAR(256)
token	BYTEA
authentication	BYTEA

項番	説明
(1)	アクセストークンを管理するテーブル。認可サーバで発行したアクセストークンの情報をリソースサーバと共有するために使用する。 各カラムの役割は以下のとおりである。 • authentication_id：認証情報を一意に識別する認証キーを保持するカラム。主キーとする。 • token：トークンの情報をシリアル化してバイナリとして保持するカラム。 保持するトークンの情報としては、アクセストークンの有効期限、スコープ、アクセストークンのトークン ID、リフレッシュトークンのトークン ID、使用しているトークンの種類を表すトークンタイプを保持する。 • token_id：アクセストークンを一意に識別するトークン ID を保持するカラム。 • user_name：認証されたリソースオーナのユーザ名を保持するカラム。 • client_id：認証されたクライアントのクライアント ID を保持するカラム。 • authentication：リソースオーナとクライアントの認証情報をシリアル化してバイナリとして保持するカラム。 • refresh_token：アクセストークンに紐付くリフレッシュトークンのトークン ID を保持するカラム。

Continued on next page

表 9.73 – continued from previous page

項番	説明
(2)	アクセストークンに紐付くリフレッシュトークンを管理するテーブル。 各カラムの役割は以下のとおりである。 • token_id：リフレッシュトークンを一意に識別するトークン ID を保持するカラム。主キーとする。 • token：トークンの情報をシリアル化してバイナリとして保持するカラム。リフレッシュトークンの有効期限を保持する。 • authentication：リソースオーナとクライアントの認証情報をシリアル化してバイナリとして保持するカラム。 アクセストークンを管理するテーブルで保持している認証情報と同じ情報を保持する。

注釈: 共有 DB でトークンを管理する場合、有効期限切れとなったトークンはクライアントがアクセストークンを利用するタイミングに削除される。そのためトークンが有効期限切れになったとしても、クライアントがアクセストークンを利用しなければ削除されない。有効期限切れとなったトークンを DB から削除するためには、バッチ処理等で別途パージを行う必要がある。

トークンの取り消し（認可サーバ）

発行したアクセストークンの取り消しの実装方法について説明する。

アクセストークンの取り消しは、`ConsumerTokenService` インタフェースを実装したクラスの `revokeToken` メソッドを呼び出すことで実現できる。`DefaultTokenServices` クラスは `org.springframework.security.oauth2.provider.token.ConsumerTokenServices` インタフェースを実装している。

アクセストークンの取り消し時に認可情報も削除することが可能である。アクセストークンの取り消し後に認可情報を削除せずに認可リクエストを行うと、前回の認可リクエスト時の認可情報が再利用される場合がある。前回の認可リクエスト時の認可情報は、認可情報の有効期限が有効であり、認可リクエストしたスコープが全て認可されている場合に再利用される。

以下に、実装例を示す。

トークンの取り消しを行うサービスクラスのインタフェースと実装クラスを作成する。

- `RevokeTokenService.java`

```
public interface RevokeTokenService {
```

```
    ResponseEntity<Map<String,String>> revokeToken(String tokenValue, String␣
↪clientId);

}
```

- RevokeTokenServiceImpl.java

```
@Service
@Transactional
public class RevokeTokenServiceImpl implements RevokeTokenService {

    @Inject
    ConsumerTokenServices consumerService; // (1)

    @Inject
    TokenStore tokenStore; // (2)

    @Inject
    ApprovalStore approvalStore; // (3)

    @Inject
    JodaTimeDateFactory dateFactory;

    public String revokeToken(String tokenValue, String clientId){ // (4)
        // (5)
        OAuth2Authentication authentication = tokenStore.
↪readAuthentication(tokenValue);

        Map<String,String> map = new HashMap<>();

        if (authentication != null) {
            if (clientId.equals(authentication.getOAuth2Request().getClientId()))
↪{ // (6)
                // (7)
                Authentication user = authentication.getUserAuthentication();
                if (user != null) {
                    Collection<Approval> approvals = new ArrayList<>();
                    for (String scope : authentication.getOAuth2Request().
↪getScope()) {
                        approvals.add(
                                new Approval(user.getName(), clientId, scope,␣
↪dateFactory.newDate(), ApprovalStatus.APPROVED));
                    }
                    approvalStore.revokeApprovals(approvals);
                }
                consumerService.revokeToken(tokenValue); // (8)
                return ResponseEntity.ok().body(map);

            } else {
                map.put("error", "invalid_client");
                return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(map);
            }
```

```
        } else {
            map.put("error", "invalid_request");
            return ResponseEntity.badRequest().body(map);
        }
    }
}
```

項番	説明
(1)	アクセストークンの取り消しを行う `ConsumerTokenService` インタフェースの Bean をインジェクションする。
(2)	アクセストークン発行時の認証情報を取得するために使用する `TokenStore` の Bean をインジェクションする。
(3)	アクセストークンの発行時の認可情報を取得するために使用する `ApprovalStore` の Bean をインジェクションする。 アクセストークンの取り消し時に認可情報を削除しない場合は不要となる。
(4)	取り消しを行うアクセストークンの値と、クライアントのチェックを行うために使用するクライアント ID をパラメータとして受け取る。
(5)	`TokenStore` の `readAuthentication` メソッドを呼び出し、アクセストークンを発行した際の認証情報を取得する。 認証情報が正常に取得できた場合のみ、トークンの削除処理を行う。
(6)	認証情報より、アクセストークン発行時に使用したクライアント ID を取得し、リクエストパラメータのクライアント ID と一致するかを確認する。 アクセストークン発行時のクライアント ID と一致する場合のみ、アクセストークンの削除を行う。

Continued on next page

表 9.74 – continued from previous page

項番	説明
(7)	認証情報より、アクセストークン発行時のリソースオーナの認証情報を取得する。 リソースオーナの認証情報が取得できた場合、TokenStore の revokeApprovals メソッドを呼び出し、認可情報の削除を行う。 クライアントクレデンシャルグラントを使用している場合はリソースオーナの認証情報が存在しないため、revokeApprovals メソッドに渡すパラメータが生成できない。 そのため、リソースオーナの認証情報が取得できない場合は認可情報の削除処理は行わない。 アクセストークンの取り消し時に認可情報を削除しない場合、この処理は不要となる。
(8)	ConsumerTokenService の revokeToken メソッドを呼び出し、アクセストークンとアクセストークンに紐付くリフレッシュトークンの削除を行う。

トークンの取り消しリクエストを受けるコントローラを作成する。

- TokenRevocationRestController.java

```
@RestController
@RequestMapping("oauth")
public class TokenRevocationRestController {

    @Inject
    RevokeTokenService revokeTokenService;

    @RequestMapping(value = "tokens/revoke", method = RequestMethod.POST) // (1)
    public ResponseEntity<Map<String,String>> revoke(@RequestParam("token") String␣
↪tokenValue,
        @AuthenticationPrincipal UserDetails user){

        // (2)
        String clientId = user.getUsername();
        ResponseEntity<Map<String,String>> result = revokeTokenService.
↪revokeToken(tokenValue, clientId); // (3)
        return result;
    }
}
```

項番	説明
(1)	`@RequestMapping` アノテーションを使用して、`/oauth/tokens/revoke` へのアクセスに対するメソッドとしてマッピングを行う。 ここで指定するパスはクライアントの認証で行った設定と同様に、Basic 認証の適用と CSRF 対策機能の無効化を行う必要がある。
(2)	Basic 認証で生成された認証情報からトークンの取り消し時のチェックで使用するクライアント ID を取得する。
(3)	`RevokeTokenService` を呼び出し、トークンの取り消しを行う。 リクエストパラメータとして受け取ったアクセストークンの値と、認証情報から取得したクライアント ID をパラメータとして渡す。

ちなみに: RFC 7009 ではリクエストパラメータとして `token_type_hint` を任意で付与してよいことが記載されている。`token_type_hint` は削除対象のトークンがアクセストークンとリフレッシュトークンのどちらであるかを判別するためのヒントである。Spring Security OAuth が提供する `ConsumerTokenService` はアクセストークンを渡すことでアクセストークンとリフレッシュトークンの両方削除するため、上記の実装例では使用していない。

注釈: 認可サーバにトークンの取り消しをリクエストしたクライアントは、認可サーバの削除後にクライアントで保持しているトークンも取り消す必要がある。クライアントサーバのトークンの取り消しについてはトークンの取り消し（クライアントサーバ）を参照されたい。

トランザクション制御

認可サーバにおけるトランザクション制御の注意点について説明する。

Spring Security OAuth が取り扱う情報（認可コード、認可情報、トークン）を DB にて管理する場合には、トランザクション管理について考慮する必要がある。

アクセストークンやリフレッシュトークンを発行する DefaultTokenServices には@Transactional アノテーションが付与されており、トランザクション管理が行われるが、認可コードを操作する org.springframework.security.oauth2.provider.code.AuthorizationCodeServices や認可情報を操作する UserApprovalHandler には@Transactional アノテーションが付与されておらず、トランザクション管理が行われない。

そのため、DataSource から取得するコネクションの自動コミットが無効となっている場合、管理対象となる情報が DB に登録されないため、トランザクション制御の設定が必須となる。なお、自動コミットが有効な場合でも、データの一貫性を担保するためにトランザクションを管理する必要がある。

認可コード、認可情報を DB にて管理する場合のトランザクション制御設定例を以下に示す。

- oauth2-auth.xml

```
<!-- omitted -->

<tx:advice id="oauthTransactionAdvice"> <!-- (1) -->
    <tx:attributes>
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>

<aop:config>
    <aop:pointcut id="authorizationOperation"
                  expression="execution(* org.springframework.security.oauth2.
↪provider.code.AuthorizationCodeServices.*(..))"/> <!-- (2) -->
    <aop:pointcut id="approvalOperation"
                  expression="execution(* org.springframework.security.oauth2.
↪provider.approval.UserApprovalHandler.*(..))"/> <!-- (3) -->
    <aop:advisor pointcut-ref="authorizationOperation" advice-ref=
↪"oauthTransactionAdvice"/>
    <aop:advisor pointcut-ref="approvalOperation" advice-ref=
↪"oauthTransactionAdvice"/>
</aop:config>
```

項番	説明
(1)	AOP を利用したトランザクション管理を行なうため、tx:advice を設定する。 このタグを使用するために tx のネームスペースとスキーマを追加している。
(2)	AOP を使用し、認可コードを操作する各メソッドにトランザクション境界を設定する。 このタグを使用するために aop のネームスペースとスキーマを追加している。
(3)	AOP を使用し、認可情報を操作する各メソッドにトランザクション境界を設定する。

リソースサーバの実装

リソースサーバの実装方法について説明する。

リソースサーバでは、アクセストークンの検証とリソースに対しての認可制御を Spring Security OAuth の機能を使用して提供する。

ここでは TODO リソースの REST API に対して認可制御を実現する方法を説明する。

設定ファイルの作成（リソースサーバ）

リソースサーバを実装する際には新たに OAuth 2.0 用の Bean 定義ファイルを作成する。

ここでは `oauth2-resource.xml` とする。

`oauth2-resource.xml` には以下の設定を追加する。

- `oauth2-resource.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:oauth2="http://www.springframework.org/schema/security/oauth2"
       xmlns:sec="http://www.springframework.org/schema/security"
       xsi:schemaLocation="http://www.springframework.org/schema/security
           https://www.springframework.org/schema/security/spring-security.xsd
           http://www.springframework.org/schema/security/oauth2
           https://www.springframework.org/schema/security/spring-security-oauth2.
↪xsd
           http://www.springframework.org/schema/beans
           https://www.springframework.org/schema/beans/spring-beans.xsd">

    <sec:http pattern="/api/v1/todos/**" create-session="stateless"
                  entry-point-ref="oauth2AuthenticationEntryPoint"> <!-- (1) -->
        <sec:access-denied-handler ref="oauth2AccessDeniedHandler"/> <!-- (2) -->
        <sec:csrf disabled="true"/> <!-- (3) -->
        <sec:custom-filter ref="oauth2AuthenticationFilter"
                           before="PRE_AUTH_FILTER" /> <!-- (4) -->
        <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
    </sec:http>

    <bean id="oauth2AccessDeniedHandler"
            class="org.springframework.security.oauth2.provider.error.
↪OAuth2AccessDeniedHandler" /> <!-- (5) -->

    <bean id="oauth2AuthenticationEntryPoint"
            class="org.springframework.security.oauth2.provider.error.
↪OAuth2AuthenticationEntryPoint" /> <!-- (6) -->

    <oauth2:resource-server id="oauth2AuthenticationFilter" resource-id=
↪"todoResource"
```

```
            token-services-ref="tokenServices" entry-point-ref=
↪"oauth2AuthenticationEntryPoint" /> <!-- (7) -->

</beans>
```

項番	説明
(1)	pattern 属性には認可制御の対象とするパスのパターンを指定する。 entry-point-ref には OAuth2AuthenticationEntryPoint の Bean を指定する。ここでの設定は定義上必要だが指定した Bean は使用されない。実際に使用されるのは後述する OAuth2AuthenticationProcessingFilter に指定している OAuth2AuthenticationEntryPoint の Bean である。
(2)	access-denied-handler には OAuth2AccessDeniedHandler の Bean を設定する。ここでは (5) で定義している oauth2AccessDeniedHandler の Bean を指定する。
(3)	本実装例では CSRF 対策を無効化する。 CSRF 対策については *CSRF 対策*を参照されたい。
(4)	custom-filter にはリソースサーバ用の認証フィルタとして OAuth2AuthenticationProcessingFilter を設定する。 ここでは (7) で定義している oauth2AuthenticationFilter の Bean を指定する。 OAuth2AuthenticationProcessingFilter はリクエストに含まれるアクセストークンを利用して Pre-Authentication を行うためのフィルタであるため、before に PRE_AUTH_FILTER を指定し PRE_AUTH_FILTER の前に OAuth2AuthenticationProcessingFilter の処理が実行されるように設定する。 Pre-Authentication については Pre-Authentication Scenarios を参照されたい。
(5)	Spring Security OAuth が提供するリソースサーバ用の AccessDeniedHandler を定義する。 OAuth2AccessDeniedHandler は、認可エラー時に発生する例外をハンドリングしてエラー応答を行う。

Continued on next page

表 9.77 – continued from previous page

項番	説明
(6)	OAuth 用のエラー応答を行うための `OAuth2AuthenticationEntryPoint` を Bean 定義する。
(7)	Spring Security OAuth が提供するリソースサーバ用の ServletFilter を定義する。 `<oauth2:resource-server>`タグを使用することで、Authentication Filter として `OAuth2AuthenticationProcessingFilter` が登録される。 `id` 属性に指定した文字列は Bean の ID となる。ここでは `oauth2AuthenticationFilter` を指定している。 `resource-id` 属性にはサーバが提供するリソースの ID を指定する。ここでは `todoResource` を指定している。 アクセストークンに紐付くクライアント情報のリソース ID に対し、`resource-id` 属性に指定したリソース ID が含まれているか検証が行われる。 検証の結果、リソース ID が含まれている場合のみリソースに対してのアクセスを許可する。 なお、`resource-id` の定義は任意であり、定義しない場合はリソース ID の検証が行われない。 `token-services-ref` 属性には `TokenServices` の ID を指定する。`TokenServices` については後述する。 `entry-point-ref` 属性には `OAuth2AuthenticationEntryPoint` の Bean を指定する。ここでは `oauth2AuthenticationEntryPoint` を指定している。

作成した `oauth2-resource.xml` を読み込むように `web.xml` に設定を追加する。

- `web.xml`

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath*:META-INF/spring/applicationContext.xml
        classpath*:META-INF/spring/oauth2-resource.xml <!-- (1) -->
        classpath*:META-INF/spring/spring-security.xml
    </param-value>
</context-param>
```

項番	説明
(1)	oauth2-resource.xml で設定したパスのパターンを内包するようなパスが spring-security.xml にアクセス制御対象として設定されている場合を考慮し、先に oauth2-resource.xml を読み込むようにする。

リソースにアクセス可能なスコープの設定

リソースごとにアクセス可能なスコープを定義するために、OAuth 2.0 用の Bean 定義ファイルにスコープの定義と SpEL 式をサポートするための Bean 定義を追加する。

実装例は以下のとおりである。

- oauth2-resource.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:oauth2="http://www.springframework.org/schema/security/oauth2"
       xmlns:sec="http://www.springframework.org/schema/security"
       xsi:schemaLocation="http://www.springframework.org/schema/security
           https://www.springframework.org/schema/security/spring-security.xsd
           http://www.springframework.org/schema/security/oauth2
           https://www.springframework.org/schema/security/spring-security-oauth2.
↪xsd
           http://www.springframework.org/schema/beans
           https://www.springframework.org/schema/beans/spring-beans.xsd">

    <sec:http pattern="/api/v1/todos/**" create-session="stateless"
                  entry-point-ref="oauth2AuthenticationEntryPoint">
        <sec:access-denied-handler ref="oauth2AccessDeniedHandler"/>
        <sec:csrf disabled="true"/>
        <sec:expression-handler ref="oauth2ExpressionHandler"/> <!-- (1) -->
        <sec:intercept-url pattern="/**" method="GET"
                           access="#oauth2.hasScope('READ')" /> <!-- (2) -->
        <sec:intercept-url pattern="/**" method="POST"
                           access="#oauth2.hasScope('CREATE')" /> <!-- (2) -->
        <sec:custom-filter ref="oauth2AuthenticationFilter"
                           before="PRE_AUTH_FILTER" />
        <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
    </sec:http>

    <!-- omitted -->

    <oauth2:web-expression-handler id="oauth2ExpressionHandler" /> <!-- (3) -->
```

```
</beans>
```

項番	説明
(1)	expression-handler には org.springframework.security.oauth2.provider.expression.OAuth2WebSecurityExpressionHandler の Bean を指定する。
(2)	intercept-url を使用してリソースに対してスコープによるアクセスポリシーを定義する。 pattern 属性には保護したいリソースのパスのパターンを指定する。本実装例では /api/v1/todos/ 配下のリソースが保護される。 method 属性にはリソースの HTTP メソッドを指定する。 access 属性にはリソースへのアクセスを認可する scope を指定する。設定値は大文字、小文字を区別する。 ここでは SpEL 式を用いて指定を行っている。
(3)	OAuth2WebSecurityExpressionHandler を Bean 定義する。 この Bean を定義することで Spring Security OAuth が提供する認可制御を行うための SpEL がサポートされる。 なお、id 属性に指定した値がこの bean の id となる。

Spring Security OAuth が用意している主な Expression を紹介する。

詳細については org.springframework.security.oauth2.provider.expression.OAuth2SecurityExpressionMethods の JavaDoc を参照されたい。

表 9.80: **Spring Security OAuth が用意している Expression**

Expression	説明
`hasScope(String scope)`	クライアントがリソースオーナから認可されているスコープと引数のスコープが一致する場合に `true` を返却する。
`hasAnyScope(String... scopes)`	クライアントがリソースオーナから認可されているスコープと引数のスコープのいずれかが一致する場合に `true` を返却する。
`hasScopeMatching(String scopeRegex)`	クライアントがリソースオーナから認可されているスコープと引数に指定された正規表現が一致する場合に `true` を返却する。
`hasAnyScopeMatching(String... scopesRegex)`	クライアントがリソースオーナから認可されているスコープと引数に指定されたいずれかの正規表現が一致する場合に `true` を返却する。
`clientHasRole(String role)`	クライアントが引数に指定されたロールを持っている場合に `true` を返却する。
`clientHasAnyRole(String... roles)`	クライアントが引数に指定されたいずれかのロールを持っている場合に `true` を返却する。
`denyOAuthClient`	OAuth 2.0 でのリクエストを拒否する。リソースオーナのみがリソースにアクセスできるようにするために使用される。
`isOAuth`	OAuth 2.0 でのリクエストを許可する。クライアントがリソースにアクセスできるようにするために使用される。

注釈: SpEL 式については Spring Security が提供する SpEL を合わせて使用することができる。

Spring Security が提供する SpEL については *Built-In の Web Expressions* を参照されたい。

認可サーバとのアクセストークンの連携方法

認可サーバとリソースサーバはアクセストークンを TokenServices を介して連携する。

連携方法の種類についてはリソースサーバへのアクセストークンの連携方法を参照されたい。

ここでは DB を共有する方法で設定を行う。

設定の解説については認可サーバでの TokenServices の設定と同様なためリソースサーバへのアクセストークンの連携方法を参照されたい。

- oauth2-resource.xml

```
<bean id="tokenServices"
    class="org.springframework.security.oauth2.provider.token.DefaultTokenServices
↪">
    <property name="tokenStore" ref="tokenStore" />
</bean>

<bean id="tokenStore"
  class="org.springframework.security.oauth2.provider.token.store.JdbcTokenStore">
  <constructor-arg ref="dataSource" />
</bean>
```

注釈: 認可サーバとリソースサーバを連携させる方法として、Spring Security OAuth で提供されている org.springframework.security.oauth2.provider.token.RemoteTokenServices を利用する方法や、JSON Web Token を利用する方法がある。Spring Security OAuth で提供されている RemoteTokenServices を利用する方法については *HTTP アクセスを介した認可サーバとリソースサーバの連携*を参照されたい。

ユーザ情報の取得

リソースサーバでは、*認証処理と Spring MVC の連携*で説明されている認証情報の取得方法と同様に、Controller クラスのメソッド引数に UserDetails を指定し@AuthenticationPrincipal アノテーションを付与することにより認証されたリソースオーナの情報を受け取ることができる。以下に実装例を示す。

```
@RestController
@RequestMapping("api")
public class TodoRestController {

    // omitted
```

```
    @RequestMapping(value = "todos", method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public Collection<Todo> list(@AuthenticationPrincipal UserDetails user) { //␣
↪(1)

        // omitted

    }
}
```

項番	説明
(1)	引数 user にリソースオーナの認証情報が格納される。

クライアントの認証情報を取得したい場合は、Controller クラスのメソッド引数に org.springframework.security.oauth2.provider.OAuth2Authentication を指定する。以下に Controller クラスのメソッド引数に OAuth2Authentication を指定してクライアントとリソースオーナの認証情報を取得する例を示す。

```
@RestController
@RequestMapping("api")
public class TodoRestController {

    // omitted

    @RequestMapping(value = "todos", method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public Collection<Todo> list(OAuth2Authentication authentication) { // (1)

        String username = authentication.getUserAuthentication().getName(); // (2)
        String clientId = authentication.getOAuth2Request().getClientId();  // (3)

        // omitted

    }
}
```

項番	説明
(1)	引数 authentication にリソースオーナ、クライアントの認証情報が格納される。
(2)	authentication よりリソースオーナ名を取得する。
(3)	authentication よりクライアント ID を取得する。

注釈: 上記はアプリケーション層で OAuth2Authentication を使用する場合の実装例である。OAuth2Authentication に依存しない実装を行いたい場合 HandlerMethodArgumentResolver を実装することで同様の機能が実現可能である。具体的な実装方法については、*HandlerMethodArgumentResolver の実装*を参照されたい。

クライアントの実装

クライアントの実装方法について説明する。

ここでは OAuth2RestTemplate を使用して実現する方法を説明する。

OAuth2RestTemplate では、OAuth 2.0 独自の機能として、AccessTokenProvider によるグラントタイプに応じたアクセストークンの取得や、OAuth2ClientContext による複数リクエスト間でのアクセストークンの共有、リソースサーバへのアクセス時のエラーハンドリングといった機能を実装している。

クライアントでは、OAuth2RestTemplate を利用し、グラントタイプやスコープなどのアプリケーション用件に沿ったパラメータを定義することで、OAuth 2.0 機能を使用したリソースへのアクセスが可能となる。

注釈: Spring Security OAuth 以外のアーキテクチャで実装されているリソースサーバにアクセスする場合は、REST API でのアクセスを受け付けているか確認されたい。リソースサーバの API が異なる場合は *OAuth2RestTemplate* 以外の手段でアクセスする必要がある。

設定ファイルの作成（クライアント）

まず、OAuth 2.0 に関する定義を行うための設定ファイルとして oauth2-client.xml を作成する。

- oauth2-client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:sec="http://www.
↪springframework.org/schema/security"
    xmlns:oauth2="http://www.springframework.org/schema/security/oauth2"
    xsi:schemaLocation="
        http://www.springframework.org/schema/security https://www.springframework.
↪org/schema/security/spring-security.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/security/oauth2 https://www.
↪springframework.org/schema/security/spring-security-oauth2.xsd
    ">


</beans>
```

web.xml に、作成した oauth2-client.xml を読み込む設定を追加する。

- web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath*:META-INF/spring/applicationContext.xml
        classpath*:META-INF/spring/oauth2-client.xml  <!-- (1) -->
        classpath*:META-INF/spring/spring-security.xml
    </param-value>
</context-param>

<!-- omitted -->
```

項番	説明
(1)	oauth2-client.xml を読み込むように設定する。

OAuth2ClientContextFilter の適用

OAuth2ClientContextFilter をサーブレットフィルタとして登録する。

OAuth2ClientContextFilter を登録することでリソースオーナによる認可が取得できていない状態でリソースサーバへのアクセスを試みた場合に発生する UserRedirectRequiredException を捕捉し、認可サーバが提供するリソースオーナの認可を取得するためのページへリダイレクトする機能を組み込むことができる。

oauth2-client.xml には以下の設定を追加する。

- oauth2-client.xml

```
<oauth2:client id="oauth2ClientContextFilter" /> <!-- (1) -->
```

項番	説明
(1)	<oauth2:client>タグを使用することで、OAuth2ClientContextFilter の Bean が定義される。id 属性に指定した文字列は Bean の ID として使用される。

web.xml に、OAuth2ClientContextFilter の設定を追加する。

- web.xml

```
<filter> <!-- (1) -->
    <filter-name>oauth2ClientContextFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
↪class>
</filter>
<filter-mapping>
    <filter-name>oauth2ClientContextFilter</filter-name>
    <url-pattern>/oauth/*</url-pattern> <!-- (2) -->
</filter-mapping>
```

項番	説明
(1)	DelegatingFilterProxy を使用して、フィルタ名 (<filter-name>属性に指定した値) と Bean 名が一致する Bean をサーブレットフィルタとして登録する。 フィルタ名には<oauth2:client>の id 属性に指定した Bean 名と同じ値を設定する。 なお、Spring Security OAuth で発生する例外が意図しない例外ハンドリングが行われないようにするために、OAuth2ClientContextFilter はサーブレットフィルタの定義の一番最後に記述することを推奨する。
(2)	UserRedirectRequiredException が発生する可能性があるパスに対して OAuth2ClientContextFilter を適用している。 上記例では、すべてのリクエストに対して OAuth2ClientContextFilter を適用している。

注釈: OAuth2ClientContextFilter は、認可サーバが認可処理後にユーザエージェントを戻すリダイレクト URI の生成をフィルタの前処理で行っているため、/*のような広範囲な定義にすると UserRedirectRequiredException が発生しないパスに対して無駄な処理が行われることになる。

注釈: OAuth2ClientContextFilter は、認可サーバがリソースオーナの認可を取得した後にリダイレクトさせる URL をリクエストスコープに currentUri という属性名で格納する。そのため、クライアントでは currentUri という属性名を使用することはできない。

前述のとおり、org.springframework.security.oauth2.client.filter.OAuth2ClientContextFilter は UserRedirectRequiredException を捕捉し、認可サーバに対してリダイレクトさせるサーブレットフィルタである。

ただし、ブランクプロジェクトで予め設定されている SystemExceptionResolver が先に UserRedirectRequiredException をハンドリングしてしまうと OAuth2ClientContextFilter は期待した動作にならない。

そのため、spring-mvc.xml の設定を変更し、SystemExceptionResolver が UserRedirectRequiredException をハンドリングしないようにする必要がある。SystemExceptionResolver の詳しい解説については例外ハンドリングを参照されたい。

spring-mvc.xml に、UserRedirectRequiredException を SystemExceptionResolver の除外対象として追加する。

- spring-mvc.xml

```
<bean id="systemExceptionResolver"
    class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">

    <!-- omitted -->

    <property name="excludedExceptions">
        <array>
            <!-- (1) -->
            <value>org.springframework.security.oauth2.client.resource.
↪UserRedirectRequiredException
            </value>
        </array>
    </property>
</bean>
```

項番	説明
(1)	UserRedirectRequiredException を SystemExceptionResolver のハンドリング対象から除外する。

OAuth2RestTemplate の設定

OAuth2RestTemplate の設定例を示す。

- oauth2-client.xml

```
<oauth2:resource id="todoAuthCodeGrantResource" client-id="firstSec"
                 client-secret="firstSecSecret"
                 type="authorization_code"
                 scope="READ,WRITE"
                 access-token-uri="${auth.serverUrl}/oauth/token"
                 user-authorization-uri="${auth.serverUrl}/oauth/authorize"/> <!--␣
↪(1) -->

<oauth2:rest-template id="todoAuthCodeGrantResourceRestTemplate" resource=
↪"todoAuthCodeGrantResource" /> <!-- (2) -->
```

項番	説明
(1)	`OAuth2RestTemplate` が参照する、アクセス対象となるリソースに関する詳細情報を定義する。 各項目の設定値については下記表を参照のこと。
(2)	`OAuth2RestTemplate` を定義する。 `id` には `OAuth2RestTemplate` の Bean 名を指定する。 `resource` には (1) で定義した Bean の `id` を指定する。

表 9.88: リソース詳細情報

項目	説明
`id`	リソースの Bean 名。
`client-id`	認可サーバにてクライントを識別する ID。
`client-secret`	認可サーバにてクライアントの認証に用いるパスワード。
`type`	グラントタイプ。認可コードグラントの場合 `authorization_code` を指定する。
`scope`	認可を要求するスコープをカンマ区切りで列挙する。設定値は大文字、小文字を区別する。省略時は認可サーバにおいてクライアントに対して設定しているスコープを全て要求する。
`access-token-uri`	アクセストークンの発行を依頼するための認可サーバのエンドポイント。

Continued on next page

表 9.88 – continued from previous page

項目	説明
user-authorization-uri	リソースオーナの認可を得るための認可サーバのエンドポイント。

注釈: 本実装例では、実装中に設定する各サーバのコンテキストルートを以下のプレースホルダで表現している。

プレースホルダのキー	設定値
auth.serverUrl	認可サーバのコンテキストルート
resource.serverUrl	リソースサーバのコンテキストルート
client.serverUrl	クライアントのコンテキストルート

また、各サーバのエンドポイントの絶対パスを表現する場合は、パスが明示的にわかるよう、${auth.serverUrl}/oauth/token のようにプレースホルダ以下にパスを指定している。実際にアプリケーションを開発する際には、例えばエンドポイントが外部システムである場合、外部システム仕様変更時のエンドポイントパス変更に対応するためにパス全体をプレースホルダで表現するなど、要件に応じてパスの指定方法を検討することを推奨する。

注釈: <oauth2:resource>タグでは、アクセストークン取得時のクライアント認証方法を指定する方法として client-authentication-scheme 属性が用意されている。client-authentication-scheme 属性に指定可能な値は以下の通り。

- header：Authorization ヘッダを使用した Basic 認証（デフォルト値）
- query：リクエスト時の URL クエリパラメータを使用した認証
- form：リクエスト時のボディパラメータを使用した認証

本ガイドラインではクライアントの認証に Basic 認証を利用するため上記の設定例では未指定としているが、アプリケーション要件に沿ったパラメータの指定を行うこと。

注釈: Spring Security OAuth 以外のアーキテクチャで実装されている認可サーバに対してアクセストークンの発行を依頼する場合はリクエストパラメータが上記とは異なる可能性があるため注意されたい。その場合はアーキテクチャの仕様を確認し、必要なリクエストパラメータを設定されたい。

リソースサーバへのアクセス

OAuth2RestTemplate を用いてリソースサーバへアクセスする方法を説明する。

認可サーバへのアクセスは OAuth2RestTemplate と OAuth2ClientContextFilter により隠蔽されるため、開発の際には認可サーバを意識する必要がなく、リソースサーバが提供する REST API に対して行う処理を、通常の REST API へのアクセスと同様に記述する。

以下に Service クラスの実装例を示す。

```
import org.springframework.web.client.RestOperations;

@Service
public class TodoServiceImpl implements TodoService {

    @Inject
    RestOperations restOperations; // (1)

    @Value("${resource.serverUrl}/api/v1/todos")
    String url;

    @Override
    public List<Todo> getTodoList() {
        Todo[] todoArray = restOperations.
            getForObject(url, Todo[].class); // (2)
        return Arrays.asList(todoArray);
    }
}
```

項番	説明
(1)	RestOperations をインジェクションする。
(2)	指定した URL に REST でメソッド GET でアクセスし結果をリストで受け取る。

注釈: OAuth2RestTemplate を使用してリソースサーバにアクセスしようとした時点でアクセストークンが発行されていない場合、一度認可サーバにリダイレクトされる。その後、アクセストークンの発行が完了するとクライアントへリダイレクトされ、再度リソースサーバへのアクセス処理が呼び出される形となる。この時、クライアントへのリダイレクトは GET により行われるため、認可サーバからリダイレクトされる可能性のある Controller は GET を許容する必要がある。

また、リダイレクト前のリソースサーバへのアクセスが POST である場合、リダイレクト後の GET ではパラメータが失われてしまう。その場合、POST パラメータをセッションに保持する等の対処が必要となるため、

注意すること。本ガイドラインでは、具体的な対処方法の説明については割愛する。

警告: Spring Security を利用している場合、ユーザが認証されていない状態で OAuth2RestTemplate を使用してアクセストークンを取得しようとすると org.springframework.security.authentication.InsufficientAuthenticationException が発生するため、OAuth2RestTemplate を使用するパスでは必ずユーザが既に認証されていることを確認されたい。

具体的には、以下のいずれかが実施されていることを確認すれば良い。

- OAuth2RestTemplate を使用する Service メソッドに対して、@PreAuthorize により認証済みのチェックをしていること
- OAuth2RestTemplate を使用するパスに対して、<sec:intercept-url>により認証済みのチェックをしていること

詳細は *Web* リソースへの認可及びメソッドへの認可を参照されたい。

なお、クライアントがリソースオーナとなるためリソースオーナの認証を必要としないクライアントクレデンシャルグラントと、OAuth2RestTemplate を使用しないインプリシットグラントの場合はエラーが発生しない。

トークンの取り消し（クライアントサーバ）

発行したアクセストークンの取り消しの実装方法について説明する。

アクセストークンの取り消しは、認可サーバにリクエストを行い、TokenStore からアクセストークンの削除を行う。認可サーバへのリクエスト時はクライアントの Basic 認証を行うため、Basic 認証用のリクエストヘッダを設定する。

認可サーバのトークンの取り消しについては トークンの取り消し（認可サーバ）を参照されたい。

クライアントサーバはアクセストークンの取り消しを認可サーバにリクエスト後、OAuth2RestTemplate で保持しているアクセストークンを削除する必要がある。

以下に、実装例を示す。

まず、認可サーバにトークン取り消し要求を行うための RestTemplate の設定を設定ファイルに追記する。

- oauth2-client.xml

```
<!-- (1) -->
<bean id="revokeRestTemplate" class="org.springframework.web.client.RestTemplate">
    <property name="interceptors">
        <list>
            <ref bean="basicAuthInterceptor" />
        </list>
    </property>
</bean>

<bean id="basicAuthInterceptor" class="org.springframework.http.client.support.
↪BasicAuthorizationInterceptor">
    <constructor-arg index="0" value="${api.auth.username}" />
    <constructor-arg index="1" value="${api.auth.password}" />
</bean>
```

項番	説明
(1)	認可サーバにトークン取り消し要求を行うための RestTemplate を Bean 定義する。 Basic 認証用のリクエストヘッダを設定するため、interceptors プロパティに ClientHttpRequestInterceptor の実装クラスである BasicAuthorizationInterceptor を指定する。 BasicAuthorizationInterceptor の設定については *Basic 認証用のリクエストヘッダ設定処理*を参照されたい。

トークンの取り消しを行うサービスクラスのインタフェースと実装クラスを作成する。

- RevokeTokenClientService.java

```
public interface RevokeTokenClientService {

    String revokeToken();

}
```

- RevokeTokenClientServiceImpl.java

```
@Service
public class RevokeTokenClientServiceImpl implements RevokeTokenClientService {

    @Value("${auth.serverUrl}/api/v1/oauth/tokens/revoke")
    String revokeTokenUrl; // (1)

    @Inject
    @Named("todoAuthCodeGrantResourceRestTemplate")
    OAuth2RestOperations oauth2RestOperations; // (2)

    @Inject
```

```
    @Named("revokeRestTemplate")
    RestOperations revokeRestOperations; // (3)

    @Override
    public String revokeToken() {

        String tokenValue = getTokenValue(oauth2RestOperations);

        String result = "";

        if(StringUtils.hasLength(tokenValue)){
            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
            MultiValueMap<String, String> variables = new LinkedMultiValueMap<>();
            variables.add("token", tokenValue);

            try {
                revokeRestOperations.postForObject(url,
                        new HttpEntity<MultiValueMap<String, String>>(variables,␣
↪headers),
                        Void.class); // (4)
                result = "success";
                initContextToken(oauth2RestOperations); // (5)
            } catch (HttpClientErrorException e) {
                result = "invalid request";
            }
        }else{
            result ="token not exist";
        }
        return result;
    }

    // (6)
    private String getTokenValue(OAuth2RestOperations oauth2RestOperations) {
        OAuth2AccessToken token = restOperations.getOAuth2ClientContext()
                .getAccessToken();
        if (token == null) {
            return "";
        }
        return token.getValue();
    }

    // (7)
    private void initContextToken(OAuth2RestOperations oauth2RestOperations) {
        oauth2RestOperations.getOAuth2ClientContext().setAccessToken(null);
    }
}
```

項番	説明
(1)	アクセストークンの取り消しを認可サーバに依頼する際に使用する URL。
(2)	取り消しを行うアクセストークンを保持している `OAuth2RestTemplate` をインジェクションする。 `RestOperations` の実装が複数存在するため、`@Named` で Bean 名を指定してインジェクションする。
(3)	アクセストークンの取り消しを行う `RestTemplate` をインジェクションする。 `RestOperations` の実装が複数存在するため、`@Named` で Bean 名を指定してインジェクションする
(4)	認可サーバにアクセストークンの取り消しを行うために、REST でメソッド POST でアクセスする。 取り消しを行うアクセストークンの値を認可サーバに渡すためにリクエストパラメータに設定する。 アクセストークンは (6) で定義している `getTokenValue` メソッドに `org.springframework.security.oauth2.client.OAuth2RestOperations` を渡して取得する。
(5)	`OAuth2RestOperations` で保持しているアクセストークンを削除する。 アクセストークンの削除は (7) で定義している `initContextToken` メソッドにアクセストークンを保持している `OAuth2RestOperations` を渡して削除する。
(6)	`OAuth2RestOperations` で保持しているアクセストークンを取得するメソッド。 パラメータとして渡された `OAuth2RestOperations` の `getAccessToken` メソッドを呼び出すことでアクセストークンを取得し、返却する。
(7)	`OAuth2RestOperations` で保持しているアクセストークンを削除するメソッド。 パラメータとして渡された `OAuth2RestOperations` の `setAccessToken` メソッドに null を渡すことでアクセストークンを削除する。

クライアントは上記で作成したサービスをアクセストークンが不要になったタイミングで呼び出すことでトー

クンの取り消しを行う。

Spring Security OAuth のデフォルト実装ではセッションスコープでアクセストークンを保持するため、クライアントのユーザがログアウトした場合やセッションタイムアウトによってセッションが破棄されるタイミングでトークンの取り消しを行うことが考えられる。

注釈: Spring Security OAuth 以外のアーキテクチャで実装されている認可サーバでは、アクセストークンの削除を REST API で受け付けていない場合がある。

その場合は認可サーバのアーキテクチャ仕様を確認し、適切に実装する必要がある。

エラーハンドリング

`OAuth2RestTemplate` を用いた認可サーバ、リソースサーバへのアクセス時に発生するエラーのハンドリング方法について説明する。

認可エンドポイントアクセス時のエラーハンドリング

認可エンドポイントで発生するエラーは不正アクセスエラーと不正クライアントエラー以外の例外に分類される。不正クライアントエラーの詳細については、不正クライアントエラーを参照されたい。不正クライアントエラーが発生した場合のエラーハンドリングは、認可リクエスト時のエラー画面のカスタマイズを参照されたい。本節では不正クライアントエラー以外エラーが発生した場合のエラーハンドリングについて説明する。

クライアントに通知されるエラーでハンドリングが必要なものは以下となる。

表 9.93: 認可エンドポイントで発生するエラー

項番	発生するエラー	説明
(1)	リソースオーナによる認可拒否 `org.springframework.security.oauth2.common.exceptions.UserDeniedAuthorizationException`	リソースオーナがスコープ認可画面で、クライアントが指定したスコープを全て拒否した場合に発生する。 リダイレクト URI のリクエストパラメータに、`error=access_denied` と `error_description=User denied access` が設定される。

警告：エラー時のレスポンスで扱うリクエストパラメータについて

RFC 6749 の 4.1.2.1. Error Response に定められている通り、OAuth 2.0 ではエラー時のレスポンスにリクエストパラメータ `error` および `error_description` を含む。このため業務実装時にこれらの名前のパラメータを使用していると、パラメータ名が競合してしまう場合があることに注意されたい。

上記の `OAuth2RestTemplate` によってスローされる例外に対するハンドリング処理を実装する必要がある。本ガイドラインでは、以下の要件に沿ってエラーハンドリングの実装を行う。

- **リソースオーナの操作に起因する例外かどうか判断して、エラー画面を変更する**　認可サーバで発生する例外は、認可拒否などリソースオーナの操作に起因する例外と、クライアントが指定したスコープが認可サーバのクライアント情報に存在しないなどのアプリケーションの不備に起因する例外に分類できる。例外の分類に基づき、以下のように適切にハンドリングする必要がある。
  - リソースオーナの操作に起因する例外は「アクセス拒否画面」に遷移し、業務の再実施を促す
  - アプリケーションの不備に起因しない例外は「システムエラー画面」に遷移し、アプリケーションの設定を見直すよう促す
- **URL への不要なパラメータの露出を防止するため、エラー画面にリダイレクトする**　認可コードグラントやインプリシットグラントのフローでは、URL パラメータに以下のパラメータを付与するが、フローの途中でエラーが発生した場合、そのままエラー画面にフォワードするとブラウザのアドレスバーなどにパラメータが露出してしまう。これを防止するために、エラー画面にリダイレクトする必要がある。
  - **認可コード（認可コードグラントのみ）**　認可コードの発行からアクセストークン発行までの間にエラーが発生した場合
  - `state` **パラメータ**　認可リクエストからリソースオーナの認可完了までにエラーが発生した場合

  認可コードグラントやインプリシットグラント以外のグラントタイプでは、上記パラメータを使用しないためリダイレクトではなくフォワードを使用してもよい。

以下に、リソースオーナの認可拒否による `UserDeniedAuthorizationException` をハンドリングしてアクセス拒否画面に遷移する例を示す。

なお、システムエラー画面に遷移するハンドリングも同様に実装する必要がある。ハンドリング対象の例外については、クライアントから認可サーバ、リソースサーバアクセス時に発生するエラーを参照されたい。また、例では `UserDeniedAuthorizationException` をアプリケーション全体で共通的にハンドリングすることを想定して、`@ControllerAdvice` アノテーションを使用している。詳細は*@ControllerAdvice の実装*を参照されたい。

以下に実装例を示す。

```
@ControllerAdvice
public class OAuth2ExceptionHandler {

    // omitted

    @ExceptionHandler(UserDeniedAuthorizationException.class) // (1)
    public String handleUserDeniedAuthorizationException(
            UserDeniedAuthorizationException e, RedirectAttributes attributes) {
        // omitted
        attributes.addFlashAttribute("exception", e);
        return "redirect:/oauthAccessDeniedError"; // (2)
    }
}
```

項番	説明
(1)	コントローラで発生したエラーをハンドリングする。 @ExceptionHandler のパラメータに指定されている UserDeniedAuthorizationException が発生した場合に実行される。
(2)	認可コードや state パラメータを隠蔽するためリダイレクトを行う。 リダイレクト先に例外を引き渡す場合は、RedirectAttributes の addFlashAttribute メソッドを使用して引き渡す例外を設定する。

また、OAuth2ExceptionHandler の@ExceptionHandler にてリダイレクトを行うため、リダイレクト先のパス/oauthAccessDeniedError に対応するコントローラを定義する必要がある。

以下にコントローラの定義例を示す。

- OAuth2ErrorController.java

```
@Controller
public class OAuth2ErrorController {

    @RequestMapping("/oauthAccessDeniedError") // (1)
    public String handleAccessDeniedError() {
        return "common/error/accessDeniedError";
    }

}
```

項番	説明
(1)	`@RequestMapping` アノテーションを使用して、`/oauthAccessDeniedError` へのアクセスに対するメソッドとしてマッピングを行う。 `/oauthAccessDeniedError` が呼び出された場合、アクセス拒否画面を表示する。

トークンエンドポイント及びリソースサーバアクセス時のエラーハンドリング

認可コードグラントではトークンエンドポイントおよびリソースサーバで発生するエラーはすべてシステムエラーとして扱えば良い。発生するエラーについては クライアントから認可サーバ、リソースサーバアクセス時に発生するエラーを参照されたい。また、エラーハンドリングについては認可エンドポイントアクセス時のエラーハンドリングを参照されたい。

インプリシットグラントの実装

インプリシットグラントを利用した認可サーバ、リソースサーバ、クライアントの実装方法について解説する。

解説は認可コードグラントからの変更点のみを対象に行っている。変更点以外の実装、解説については認可コードグラントの実装を参照されたい。

以下に認可コードグラントの実装から各サーバを実装する際の変更点を示す。

サーバ	認可コードグラントからの変更点
認可サーバ	認可サーバの定義
リソースサーバ	なし
クライアント	クライアントの実装の全て

認可サーバの実装

認可コードグラントの実装から `oauth2-auth.xml` を変更する必要がある。

`oauth2-auth.xml` 以外の実装は認可サーバの実装を参照し作成されたい。

認可サーバの定義

認可コードグラントからの `oauth2-auth.xml` の変更点を以下に示す。

- `oauth2-auth.xml`

```
<oauth2:authorization-server
    client-details-service-ref="clientDetailsService"
    user-approval-handler-ref="userApprovalHandler"
    token-services-ref="tokenServices">
    <oauth2:implicit />  <!-- (1) -->
    <oauth2:refresh-token />
</oauth2:authorization-server>
```

項番	説明
(1)	<oauth2:implicit />タグを使用して、インプリシットグラントをサポートする。

警告: <oauth2:implicit />タグと<oauth2:refresh-token />タグは上記の順番で設定する必要がある。

詳細は認可サーバで複数のグラントタイプをサポートする場合を参照されたい。

リソースサーバの実装

認可コードグラントの実装から変更は不要である。

実装はリソースサーバの実装を参照し作成されたい。

クライアントの実装

インプリシットグラントでは一般的に JavaScript などで実装されたクライアントが採用される。

Spring Security OAuth では Java 以外のライブラリを提供していないため、本ガイドラインでは JavaScript を用いて独自にクライアントを実装する方法について解説する。

JavaScript を使用したリソースサーバへのアクセス

本ガイドラインでは、インプリシットグラント向けのクライアントの実装として、JavaScript を使用してリソースサーバから JSON 形式のデータを取得し、画面に表示させる方法を説明する。

注釈: 以降に説明する実装例では JavaScript ライブラリとして jQuery(バージョン 3.1.1) を使用する。JQuery

は src/main/webapp/resources/vendor に格納し、作成した JavaScript ファイルは、src/main/webapp/resources/app/js に格納する。

注釈: アクセストークンの格納先

HTML5 準拠のブラウザを利用する場合、アクセストークンの格納先の代表的な例として、ローカルストレージとセッションストレージが挙げられる。

以下にローカルストレージとセッションストレージの保持期間、用途を示す。

格納先	保持期間	用途
ローカルストレージ	明示的にクリアするまで	端末のブラウザを一人のユーザのみが利用する場合で、アクセストークンの発行回数を減らし、サービスの利便性を向上させたい場合
セッションストレージ	タブやブラウザを閉じるまで	端末のブラウザを複数のユーザが共用で利用する場合で、アクセストークンの不正利用を防止し、セキュリティを強化したい場合

本実装例ではローカルストレージを採用する例を紹介する。

以下に OAuth 2.0 機能を独自に実装した JavaScript と、それを利用したクライアントの実装例を示す。

- oauth2.js

```
var oauth2Func = (function(exp, $) {
    "use strict";

    var
        config = {},
        DEFAULT_LIFETIME = 3600;

    var uuid = function() {
        return "xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx".replace(/[xy]/g, function(c)
↪{
            var r = Math.random()*16|0, v = c == "x" ? r : (r&0x3|0x8);
            return v.toString(16);
        });
    };

    var encodeURL= function(url, params) {
        var res = url;
```

```
        var k, i = 0;
        for(k in params) {
            res += (i++ === 0 ? "?" : "&") + encodeURIComponent(k) + "=" +␣
↪encodeURIComponent(params[k]);
        }
        return res;
    };

    var epoch = function() {
        return Math.round(new Date().getTime()/1000.0);
    };

    var parseQueryString = function (qs) {
        var e,
            a = /\+/g,
            r = /([^&;=]+)=?([^&;]*)/g,
            d = function (s) { return decodeURIComponent(s.replace(a, " ")); },
            q = qs,
            urlParams = {};

        while (e = r.exec(q)) {
           urlParams[d(e[1])] = d(e[2]);
        }

        return urlParams;
    };

    var saveState = function(state, obj) {
        localStorage.setItem("state-" + state, JSON.stringify(obj));
    };

    var getState = function(state) {
        var obj = JSON.parse(localStorage.getItem("state-" + state));
        localStorage.removeItem("state-" + state);
        return obj;
    };

    var hasScope = function(token, scope) {
        if (!token.scopes) return false;
        var i;
        for(i = 0; i < token.scopes.length; i++) {
            if (token.scopes[i] === scope) return true;
        }
        return false;
    };

    var filterTokens = function(tokens, scopes) { // (1)
        if (!scopes) scopes = [];

        var i, j,
        result = [],
        now = epoch(),
        usethis;
```

```
        for(i = 0; i < tokens.length; i++) {
            usethis = true;

            if (tokens[i].expires && tokens[i].expires < (now+1)) usethis = false;

            for(j = 0; j < scopes.length; j++) {
                if (!hasScope(tokens[i], scopes[j])) usethis = false;
            }

            if (usethis) result.push(tokens[i]);
        }
        return result;
    };

    var saveTokens = function(providerId, tokens) {
        localStorage.setItem("tokens-" + providerId, JSON.stringify(tokens));
    };

    var getTokens = function(providerId) {
        var tokens = JSON.parse(localStorage.getItem("tokens-" + providerId));
        if (!tokens) tokens = [];

        return tokens;
    };

    var wipeTokens = function(providerId) {
        localStorage.removeItem("tokens-" + providerId);
    };

    var saveToken = function(providerId, token) { // (2)
        var tokens = getTokens(providerId);
        tokens = filterTokens(tokens);
        tokens.push(token);
        saveTokens(providerId, tokens);
    };

    var getToken = function(providerId, scopes) {
        var tokens = getTokens(providerId);
        tokens = filterTokens(tokens, scopes);
        if (tokens.length < 1) return null;
        return tokens[0];
    };

    var sendAuthRequest = function(providerId, scopes) { // (3)
        if (!config[providerId]) throw "Could not find configuration for provider
↪" + providerId;
        var co = config[providerId];

        var state = uuid();
        var request = {
            "response_type": "token"
        };
        request.state = state;
```

```
        if (co["redirectUrl"]) {
            request["redirect_uri"] = co["redirectUrl"];
        }
        if (co["clientId"]) {
            request["client_id"] = co["clientId"];
        }
        if (scopes) {
            request["scope"] = scopes.join(" ");
        }

        var authurl = encodeURL(co.authorization, request);

        if (window.location.hash) {
            request["restoreHash"] = window.location.hash;
        }
        request["providerId"] = providerId;
        if (scopes) {
            request["scopes"] = scopes;
        }

        saveState(state, request);
        redirect(authurl);

    };

    var checkForToken = function(providerId) { // (4)
        var h = window.location.hash;

        if (h.length < 2) return true;

        if (h.indexOf("error") > 0) { // (5)
            h = h.substring(1);
            var errorinfo = parseQueryString(h);
            handleError(providerId, errorinfo);
            return false;
        }

        if (h.indexOf("access_token") === -1) {
            return true;
        }
        h = h.substring(1);
        var atoken = parseQueryString(h);

        if (!atoken.state) {
            return true;
        }

        var state = getState(atoken.state);
        if (!state) throw "Could not retrieve state";
        if (!state.providerId) throw "Could not get providerId from state";
        if (!config[state.providerId]) throw "Could not retrieve config for this␣
↪provider.";
```

```
        var now = epoch();
        if (atoken["expires_in"]) {
            atoken["expires"] = now + parseInt(atoken["expires_in"]);
        } else {
            atoken["expires"] = now + DEFAULT_LIFETIME;
        }

        if (atoken["scope"]) {
            atoken["scopes"] = atoken["scope"].split(" ");
        } else if (state["scopes"]) {
            atoken["scopes"] = state["scopes"];
        }

        saveToken(state.providerId, atoken);

        if (state.restoreHash) {
            window.location.hash = state.restoreHash;
        } else {
            window.location.hash = "";
        }
        return true;
    };

    var handleError = function(providerId, cause) { // (6)
        if (!config[providerId]) throw "Could not retrieve config for this␣
↪provider.";

        var co = config[providerId];
        var errorDetail = cause["error"];
        var errorDescription = cause["error_description"];

        // redirect error page
        if(co["errRedirectUrl"]) {
            var request = {};
            if (errorDetail) {
                request["error"] = errorDetail;
            }
            if (errorDescription) {
                request["error_description"] = errorDescription;
            }
            redirect(encodeURL(co["errRedirectUrl"], request));
        } else {
            alert("Access Error. cause: " + errorDetail + "/"
                    + errorDescription);
        }
    };


    var redirect = function(url) {
        window.location = url;
    };
```

```
    var initialize = function(c) {
        config = c;
        try {
            var key, providerId;
            for(key in c) {
                    providerId = key;
            }
            return checkForToken(providerId);
        } catch(e) {
            console.log("Error when retrieving token from hash: " + e);
            window.location.hash = "";
            return false;
        }
    };

    var clearTokens = function(config) { // (7)
        var key;
        for(key in config) {
            wipeTokens(key);
        }
    };

    var oajax = function(settings) { // (8)
        var providerId = settings.providerId;
        var scopes = settings.scopes;
        var token = getToken(providerId, scopes);

        if (!token) {
            sendAuthRequest(providerId, scopes);
            return;
        }

        if (!settings.headers) settings.headers = {};
        settings.headers["Authorization"] = "Bearer " + token["access_token"];

        return $.ajax(settings);
    };

    var parseFailureJSON = function(providerId, data) { // (9)
        var res = data.responseJSON;
        var error = res.error;
        var errorDescription = res.error_description;
        var co = config[providerId];

        if (error === "invalid_token" && errorDescription) {
            var tokens = getTokens(providerId);
            for (var token of tokens) {
                if (errorDescription.indexOf(token["access_token"]) >= 0) {
                    if (errorDescription.indexOf("Invalid access token") < 0) {
                        // clear expired tokens
                        wipeTokens(providerId)

                        // redirect for get access token
```

```
                        if (co["redirectUrl"]) {
                            redirect(co["redirectUrl"]);
                            return;
                        }
                    }
                    break;
                }
            }
        }

        if (co["errRedirectUrl"]) {
            var request = {};
            if (error) {
                request["error"] = error;
            }
            if (errorDescription) {
                request["error_description"] = errorDescription;
            }
            redirect(encodeURL(co["errRedirectUrl"], request));
        } else {
            alert("Access Error. cause: " + error + "/" + errorDescription);
        }
    };

    return {
        initialize: function(config) {
            return initialize(config);
        },
        clearTokens: function() {
            return clearTokens();
        },
        oajax: function(settings) {
            return oajax(settings);
        },
        parseFailureJSON : function(providerId, data) {
            return parseFailureJSON(providerId, data);
        }
    };

})(window, jQuery);
```

項番	説明
(1)	クライアントが使用可能なアクセストークンを判別し返却する関数(`filterTokens`)。 ローカルストレージに格納されているアクセストークンより、有効期限が超過しておらず、かつパラメータで指定されたスコープと一致するものを返却する。 スコープが指定されていない場合は有効期限が超過していないアクセストークンを返却する。

Continued on next page

表 9.99 – continued from previous page

項番	説明
(2)	ローカルストレージにアクセストークンを格納する関数 (`saveToken`)。 後述するコンフィギュレーション情報の `providerId` を引数として受け取り、アクセストークンの配列をローカルストレージに格納する際のキーにする。 アクセストークンの配列は `filterTokens` から得たアクセストークンに、引数の `token` を加えたものである。
(3)	認可サーバに対して認可を要求する関数 (`sendAuthRequest`)。 コンフィギュレーション情報より必要パラメータを取得し、リクエストを作成する。
(4)	認可の応答よりアクセストークンを取得する関数 (`checkForToken`)。 URL のハッシュで返却される認可の応答を検証し、正常である場合は `saveToken` を用いてローカルストレージに情報を格納する。
(5)	認可の結果、エラーが返却された場合エラー処理として `handleError` を呼び出す。
(6)	認可時のエラーを処理する関数 (`handleError`)。 本ガイドラインでは、エラー処理の実装例としてコンフィギュレーション情報にて指定されているエラー時のリダイレクト先 URL へのリダイレクトを行っている。
(7)	ローカルストレージに格納されているアクセストークンをクリアする関数 (`clearTokens`)。
(8)	リソースサーバに対してリソースへのアクセスを要求する関数 (`oajax`)。 ローカルストレージよりアクセストークンを取得し、jQuery の `ajax` 関数を用いてリソースサーバへリクエストを行う。

Continued on next page

表 9.99 – continued from previous page

項番	説明
(9)	ajax 関数でエラーが発生した場合に返却される JSON を解析して遷移先を決定する関数(parseFailureJSON)。 アクセストークンが有効期限切れかどうか判断し、有効期限切れの場合はアクセストークンを削除して再度画面表示を行う。 例として、以下の条件で有効期限切れを判断することが可能である。 • error：invalid_token であること • error_description：アクセストークンを含み、かつ、Invalid access token を含まないこと 有効期限切れでない場合は、エラー時のリダイレクト先 URL に error と error_description を付与してリダイレクトを行う。 **注釈: 判定条件について** 上記の判定条件は Spring Security OAuth の仕様で定義されたものではない。あくまで現状の TokenServices の実装に基づいたワークアラウンド的な判定条件であり、今後の Spring Security OAuth の実装変更に合わせて変更が必要となる可能性がある点に留意されたい。 • DefaultTokenServices：有効期限切れを示す expired という文字列とアクセストークンを返却する • RemoteTokenServices：有効期限切れを明確に判断できる文字列はなくアクセストークンのみ返却する もし、リソースサーバが RemoteTokenServices を使用しない場合は、以下の条件に緩和することが出来る。 • error：invalid_token であること • error_description：expired を含むこと

クライアントの画面を表示するコントローラの実装例を示す。アクセスする認可サーバやリソースサーバの URL を可変とするため、コントローラでプロパティや環境変数から URL を取得し、画面に連携する。

- TodoController.java

```
@Controller
public class TodoListController {

    @Value("${client.serverUrl}") // (1)
```

```
    String applicationContextUrl;

    @Value("${auth.serverUrl}") // (2)
    String authServerUrl;

    @Value("${resource.serverUrl}") // (3)
    String resourceServerUrl;

    @RequestMapping(value = "todo/get", method = RequestMethod.GET)
    public String getTodo(Model model) {
        model.addAttribute("client.serverUrl", applicationContextUr);
        model.addAttribute("auth.serverUrl", authServerUrl);
        model.addAttribute("resource.serverUrl", resourceServerUrl);
        return "todo/todoList";
    }

}
```

項番	説明
(1)	クライアントのルートパスを取得する。
(2)	認可サーバのルートパスを取得する。
(3)	リソースサーバのルートパスを取得する。

また、エラー時のリダイレクト先 URL(`/oauth/error`) のハンドリングを行うコントローラを追加する。以下にコントローラの実装例を示す。

- `OAuth2ErrorController.java`

```
@Controller
@RequestMapping("/oauth/error")
public class OAuth2ErrorController {

```

```
    // omitted

    @RequestMapping(params = { "error=access_denied",
            "error_description=User denied access" }) // (1)
    public String handleAccessDeniedError(
            @RequestParam("error") String error,
            @RequestParam("error_description") String description) {

        // omitted

        return "common/error/accessDeniedError";
    }

    @RequestMapping // (2)
    public String handleError(
            @RequestParam(name = "error", required = false) String error,
            @RequestParam(name = "error_description", required = false) String␣
↪description) {

        // omitted

        return "common/error/systemError";
    }

}
```

項番	説明
(1)	リクエストパラメータがリソースオーナによる認可拒否を示す以下の条件の場合、アクセス拒否画面に遷移する。 • error：access_denied • error_description：User denied access
(2)	(1) 以外の場合、システムエラー画面に遷移する。

クライアントの画面（JSP）の実装例を示す。JSP では、前述の独自に実装した JavaScript を利用して、認可

サーバやリソースサーバにアクセスする。

- todoList.jsp

```
<script type="text/javascript" src="${pageContext.request.contextPath}/resources/
↪vendor/jquery/jquery.js"></script> <!-- (1) -->
<script type="text/javascript" src="${pageContext.request.contextPath}/resources/
↪app/js/oauth2.js"></script> <!-- (1) -->
<script type="text/javascript">
"use strict";

$(document).ready(function() {
    var result = oauth2Func.initialize({ // (2)
        "todo" : { // (3)
            clientId : "client", // (4)
            redirectUrl : "${client.serverUrl}/todo/get", // (5)
            errRedirectUrl : "${client.serverUrl}/oauth/error", // (6)
            authorization : "${auth.serverUrl}/oauth/authorize" // (7)
        }
    });

    if (result) {
        oauth2Func.oajax({ // (8)
            url : "${resource.serverUrl}/api/v1/todos", // (9)
            providerId : "todo",  // (10)
            scopes : [ "READ" ],  // (11)
            dataType : "json",    // (12)
            type : "GET"  // (13)
        }).done(function(data) {  // (14)
            $("#message").text(JSON.stringify(data));
        }).fail(function(data) {  // (15)
            oauth2Func.parseFailureJSON("todo", data);
        });
    }
});

</script>
<div id="wrapper">
    <p id="message"></p>
</div>
```

項番	説明
(1)	前述の、jQuery、独自に実装した JavaScript をそれぞれ格納したパスを指定する。
(2)	認可要求に使用するコンフィギュレーション情報を定義し、初期化する。

Continued on next page

表 9.102 – continued from previous page

項番	説明
(3)	クライアント別にコンフィギュレーション情報を区別するための識別子として一意な値を指定する。 後述するリソースへのアクセス処理では、本項目をキーにコンフィギュレーション情報を管理・取得する。
(4)	クライアントを識別する ID を指定する。
(5)	認可サーバのリソースオーナ認証後にクライアントをリダイレクトさせる URL を指定する。
(6)	認可応答として認可サーバよりエラーを受信した場合にリダイレクトさせる URL を指定する。 本ガイドラインではエラー受信時の実装例として、クライアントの画面にリダイレクトしエラー画面を表示させる方法を示す。
(7)	認可サーバの認可エンドポイントを指定する。
(8)	リソースへのアクセスを実行する。
(9)	リソースサーバのアクセス先 URL を指定する。
(10)	参照するコンフィギュレーション情報の識別子を指定する。
(11)	認可を要求するスコープを指定する。設定値は大文字、小文字を区別する。
(12)	レスポンス形式を指定する。
(13)	メソッド GET でリソースサーバへアクセスする。

Continued on next page

表 9.102 – continued from previous page

項番	説明
(14)	処理成功時に行う処理を指定する。`message` には処理成功時のレスポンスが格納される。 本ガイドラインではレスポンスをそのまま出力している。
(15)	処理失敗時に行う処理を指定する。

警告: 本実装例ではローカルストレージにアクセストークンを格納する際のキーとして固定の値を使用している。

本実装例のようにローカルストレージに固定のキーで情報を格納すると、同一端末のブラウザを複数のユーザが利用する場合に、あるユーザが格納した情報が別のユーザに意図せず参照されてしまう可能性がある。

同一端末のブラウザを複数のユーザが利用することが想定される場合は、セッションストレージを採用し、ユーザごとにユニークなキーで格納するなど、意図しない不正利用が起こらない仕組みを実装する必要がある。

注釈: Spring Security OAuth 以外のアーキテクチャで実装されている認可サーバに対してアクセストークンの発行を依頼する場合はリクエストパラメータが上記とは異なる可能性があるため注意されたい。その場合はアーキテクチャの仕様を確認し、必要なリクエストパラメータを設定されたい。

トークンの取り消し (クライアントサーバ)

発行したアクセストークンの取り消しについて説明する。

認可サーバからアクセストークンを削除する方法についてはトークンの取り消し（認可サーバ）を参照されたい。

クライアントではアクセストークンが不要になったタイミングで oauth2.js の clearTokens を呼び出すことでローカルストレージに格納したアクセストークンを削除できる。

以下に実装例を示す。

```
$(document).ready(function() {
    oauth2Func.clearTokens({ // (1)
        "todo" : {} // (2)
    });
});
```

項番	説明
(1)	ローカルストレージからアクセストークンを削除する clearTokens 関数を呼び出す。
(2)	アクセストークンを参照するキーとして initialize で設定した一意な値と同じ値を設定する。

警告: ローカルストレージからアクセストークンを削除しても、トークンの取り消し（認可サーバ）で説明しているように、リソースオーナに対する認可の要求を行わずにアクセストークンを再取得できるケースがある。クライアントからアクセストークンを削除するだけではセキュリティ面での対処としては不十分であることに注意されたい。

警告: アクセストークンの削除を行うタイミングによっては、ブラウザの終了などで JavaScript が実行されない場合がある。セキュリティ要件上アクセストークンを削除する必要がある場合は、ユーザがブラウザを終了してしまうことも考慮し、ユースケース上必ず実行されるポイントでアクセストークンの削除を行う必要があることに注意されたい。

エラーハンドリング

インプリシットグラントにおける認可サーバ、リソースサーバへのアクセス時に発生するエラーのハンドリング方法について説明する。

認可エンドポイントアクセス時のエラーハンドリング

認可エンドポイントで発生するエラーでハンドリングするエラーは認可コードグラントと同様であるため認可エンドポイントアクセス時のエラーハンドリングを参照されたい。エラーハンドリングについては *JavaScript* を使用したリソースサーバへのアクセスを参照されたい。

リソースサーバアクセス時のエラーハンドリング

リソースサーバへアクセス時のエラーでハンドリングが必要なものは以下となる。

表 9.104: リソースサーバで発生するエラー

項番	発生するエラー	説明
(1)	アクセストークン検証エラー `org.springframework.security.oauth2.common.exceptions.InvalidTokenException`	リソースサーバが受け取ったアクセストークンの正当性を確認出来ない場合（異常系）や、アクセストークンの有効期限が切れている場合（正常系）に発生する。 アクセストークン検証エラーは正常系と異常系が混在しているため、インプリシットグラント等 `OAuth2RestTemplate` を利用しない場合にこのエラーを受け取った場合、アクセストークンの有効期限切れかどうか判定する必要がある。

エラーハンドリングについては *JavaScript* を使用したリソースサーバへのアクセスを参照されたい。

注釈: 上記ではリソースオーナの操作により発生するエラーのハンドリングのみ紹介しているが、システムエラーもハンドリングする必要がある。発生するエラーについては クライアントから認可サーバ、リソースサーバアクセス時に発生するエラーを参照されたい。

リソースオーナパスワードクレデンシャルグラントの実装

リソースオーナパスワードクレデンシャルグラントを利用した認可サーバ、リソースサーバ、クライアントの実装方法について説明する。

解説は認可コードグラントからの変更点のみを対象に行っている。変更点以外の実装、解説については認可コードグラントの実装を参照されたい。

以下に認可コードグラントの実装から各サーバを実装する際の変更点を示す。

サーバ	認可コードグラントからの変更点
認可サーバ	認可サーバの定義 クライアントの認証 トークンの取り消し（認可サーバ）
リソースサーバ	なし
クライアント	*OAuth2RestTemplate* の設定

認可サーバの実装

認可コードグラントの実装から `oauth2-auth.xml` を変更し、DB 及び Service クラスから不要な定義を削除する必要がある。

`oauth2-auth.xml` 以外の実装は認可サーバの実装を参照し作成されたい。

認可サーバの定義

認可コードグラントからの `oauth2-auth.xml` の変更点を以下に示す。

- `oauth2-auth.xml`

```
<oauth2:authorization-server
    client-details-service-ref="clientDetailsService"
    user-approval-handler-ref="userApprovalHandler"
    token-services-ref="tokenServices">
    <oauth2:refresh-token />
    <oauth2:password />  <!-- (1) -->
</oauth2:authorization-server>
```

項番	説明
(1)	<oauth2:password />タグを使用して、リソースオーナパスワードクレデンシャルグラントをサポートする。

警告: <oauth2:password />タグと<oauth2:refresh-token />タグは上記の順番で設定する必要がある。

詳細は認可サーバで複数のグラントタイプをサポートする場合を参照されたい。

クライアントの認証

リソースオーナパスワードクレデンシャルグラントでは認可グラントで説明したとおりアクセストークンを直接発行する。そのため、リソースオーナに対して認可の要求は行われず、ユーザエージェントのリダイレクトは発生しない。

認可コードグラントのクライアントの認証で定義したテーブル、web_server_redirect_uris は不要となるため削除する。

トークンの取り消し（認可サーバ）

認可コードグラントの実装では、RevokeTokenServiceImpl.java にて認可サーバのトークンの取り消しに合わせて、認可情報の取り消しを行っている。

リソースオーナに対して認可の要求を行わないリソースオーナパスワードクレデンシャルグラントでは認可情報の取り消しが不要であるため、この処理を削除する。

以下に、認可情報の取り消しをコメントアウトした RevokeTokenServiceImpl.java を示す。

- RevokeTokenServiceImpl.java

```
@Service
@Transactional
public class RevokeTokenServiceImpl implements RevokeTokenService {

    @Inject
    ConsumerTokenServices consumerService;

    @Inject
    TokenStore tokenStore;

    @Inject
    ApprovalStore approvalStore;

    @Inject
    JodaTimeDateFactory dateFactory;

    public String revokeToken(String tokenValue, String clientId){

        OAuth2Authentication authentication = tokenStore.
↪readAuthentication(tokenValue);
        if (authentication != null) {
            if (clientId.equals(authentication.getOAuth2Request().getClientId())) {
                /* Authentication user = authentication.getUserAuthentication();
                if (user != null) {
                    Collection<Approval> approvals = new ArrayList<>();
                    for (String scope : authentication.getOAuth2Request().
↪getScope()) {
                        approvals.add(
                                new Approval(user.getName(), clientId, scope,␣
↪dateFactory.newDate(), ApprovalStatus.APPROVED));
                    }
                    approvalStore.revokeApprovals(approvals);
                } */
                consumerService.revokeToken(tokenValue);
                return "success";

            } else {
                return "invalid client";
            }
        } else {
            return "invalid token";
        }
    }
}
```

リソースサーバの実装

認可コードグラントの実装から変更は不要である。

実装はリソースサーバの実装を参照し作成されたい。

クライアントの実装

認可コードグラントの実装から `oauth2-client.xml` を変更する必要がある。

`oauth2-client.xml` 以外の実装はクライアントの実装を参照し作成されたい。

OAuth2RestTemplate の設定

リソースオーナパスワードクレデンシャルグラントでは、クライアントがリソースオーナのユーザ名およびパスワードを使用してアクセストークンの発行を依頼する。

`OAuth2RestTemplate` にはリソースオーナのユーザ名およびパスワードをそれぞれパラメータとして設定する必要があるが、複数のリソースオーナが同じクライアントを利用する場合、リソースオーナ毎に設定内容を切り替える考慮が必要となる。

ここでは、`OAuth2RestTemplate` のリソースを Session スコープの Bean で設定し、その Bean にリソースオーナの情報を格納することによって、リソースオーナ毎の設定内容の切り替えを実現する方法を説明する。

リソースとして通常使用する`<oauth2:resource>`タグは Singleton スコープの Bean となるため、Session スコープに変更する場合は独自に Bean 定義する必要がある。

`OAuth2RestTemplate` の設定例を以下に示す。

リソースオーナ毎に設定内容を切り替えられるよう、`org.springframework.security.oauth2.client.token.grant.password.ResourceOwnerPasswordResourceDetails` を Session スコープで定義し、`OAuth2RestTemplate` への設定を行う。

- `oauth2-client.xml`

```
<bean id="todoPasswordGrantResource" class="org.springframework.security.oauth2.
↪client.token.grant.password.ResourceOwnerPasswordResourceDetails"
    scope="session">
    <aop:scoped-proxy />
    <property name="clientId" value="firstSec" />
    <property name="clientSecret" value="firstSecSecret" />
    <property name="accessTokenUri" value="${auth.serverUrl}/oauth/token" />
    <property name="scope">
        <list>
            <value>READ</value>
            <value>WRITE</value>
        </list>
    </property>
</bean> <!-- (1) -->

<oauth2:rest-template id="todoPasswordGrantResourceRestTemplate" resource=
↪"todoPasswordGrantResource" /> <!-- (2) -->
```

項番	説明
(1)	`OAuth2RestTemplate` が参照する、アクセス対象となるリソースに関する詳細情報を定義する。 各項目の設定値については下記表を参照のこと。
(2)	`OAuth2RestTemplate` を定義する。 `id` には `OAuth2RestTemplate` の Bean 名を指定する。 `resource` には (1) で定義した Bean の `id` を指定する。

項目	説明
`class`	`OAuth2RestTemplate` のリソースとする Bean を指定する。ここでは `ResourceOwnerPasswordResourceDetails` を指定する。
`scope`	session を指定し、スコープ範囲を HTTPSession とする。
`<aop:scoped-proxy />`	Session スコープの Bean を Singleton の Bean である `OAuth2RestTemplate` にインジェクションするため設定する。 これは、Session スコープの Bean より Singleton の Bean の方がライフサイクルが長いため必要になる設定である。 このタグを使用するために `aop` のネームスペースとスキーマを追加している。
`clientId` プロパティ	Bean の `clientId` に対して認可サーバにてクライントを識別する ID を設定する。

Continued on next page

表 9.108 – continued from previous page

項目	説明
clientSecret プロパティ	Bean の clientSecret に対して認可サーバにてクライアントの認証に用いるパスワードを設定する。
accessTokenUri プロパティ	アクセストークンの発行を依頼するための認可サーバのエンドポイントを指定する。
scope プロパティ	Bean の scope に対して認可を要求するスコープの一覧を設定する。 <oauth2:resource>タグを使用する場合とは異なり、scope をリスト形式で指定する。

注釈: Spring Security OAuth 以外のアーキテクチャで実装されている認可サーバに対してアクセストークンの発行を依頼する場合はリクエストパラメータが上記とは異なる可能性があるため注意されたい。その場合はアーキテクチャの仕様を確認し、必要なリクエストパラメータを設定されたい。

注釈: リソースオーナのユーザ名、パスワードの取得は、アクセスが必要となったタイミングでクライアントの画面等でリソースオーナから入力され、Bean に格納することを想定している。本ガイドラインでは、ユーザ名、パスワードの具体的な取得方法については説明を割愛する。

警告: 本ガイドラインで説明している認可サーバでは、認証に使用するパスワードに対してハッシュ化の後比較検証を行うため、ResourceOwnerPasswordResourceDetails に設定するパスワードは平文である必要がある。クライアントがリソースオーナのパスワードを平文で扱うことによるリスクが高いため、リソースオーナパスワードクレデンシャルグラントはクライアント-リソースオーナ間で高い信頼関係があり、かつクライアントがセキュアな環境に配置されているなどの非常に限定的な状況でのみ利用

すること。認可サーバにおけるハッシュ化の設定については クライアントの認証を参照のこと。

エラーハンドリング

リソースオーナパスワードクレデンシャルグラントにおける認可サーバ、リソースサーバへのアクセス時に発生するエラーのハンドリング方法について説明する。

トークンエンドポイント及びリソースサーバアクセス時のエラーハンドリング

リソースサーバで発生するエラーでハンドリングする必要のあるエラー及びエラーハンドリングについては認可コードグラントと同様であるためトークンエンドポイント及びリソースサーバアクセス時のエラーハンドリングを参照されたい。

トークンエンドポイントで発生する例外は、クライアントの `OAuth2RestTemplate` で `org.springframework.security.oauth2.client.resource.OAuth2AccessDeniedException` にラップされる。`OAuth2AccessDeniedException` でラップされるエラーでハンドリングが必要なものは以下となる。

表 9.109: トークンエンドポイントで発生するエラー

項番	発生するエラー	説明
(1)	リソースオーナ認証エラー `org.springframework.security.oauth2.common.exceptions.InvalidGrantException`	リソースオーナパスワードクレデンシャルグラント使用時に提示したリソースオーナの認証情報に誤りがある場合に発生する。 エラー発生時には例外クラスとして `InvalidGrantException` が認可サーバでハンドリングされる。

上記の OAuth2RestTemplate によってスローされる例外に対するハンドリング処理を実装する必要がある。以下に実装例を示す。実装例で使用している@ControllerAdvice アノテーションについては *@ControllerAdvice の実装*を参照されたい。

```
@ControllerAdvice
public class OAuth2ExceptionHandler {

    // omitted

    @ExceptionHandler(OAuth2AccessDeniedException.class) // (1)
    public String handleOAuth2AccessDeniedException(OAuth2AccessDeniedException e,
            Model model) {

        // omitted

        // (2)
        Throwable cause = e.getCause();
        if (cause instanceof InvalidGrantException) {
            return handleInvalidGrantException(((InvalidGrantException) cause),
                    model);
        }

        model.addAttribute("exception", e);

        return "common/error/systemError";
    }

    private String handleInvalidGrantException(InvalidGrantException e,
            Model model) {
        model.addAttribute("exception", e);

        return "common/error/accessDeniedError";
    }

}
```

項番	説明
(1)	コントローラで発生したエラーをハンドリングする。 @ExceptionHandler のパラメータに指定されている OAuth2AccessDeniedException が発生した場合に実行される。
(2)	発生したエラーの原因により遷移先画面を決定する。 エラーの原因が InvalidGrantException の場合はアクセス拒否画面に遷移する。 上記以外の場合はシステムエラー画面に遷移する。

注釈: 上記では上記ではリソースオーナの操作により発生するエラーのハンドリングのみ紹介しているが、システムエラーもハンドリングする必要がある。発生するエラーについては クライアントから認可サーバ、リソースサーバアクセス時に発生するエラーを参照されたい。

クライアントクレデンシャルグラントの実装

クライアントクレデンシャルグラントを利用した認可サーバ、リソースサーバ、クライアントの実装方法について説明する。

解説は認可コードグラントからの変更点のみを対象に行っている。変更点以外の実装、解説については認可コードグラントの実装を参照されたい。

以下に認可コードグラントの実装から各サーバを実装する際の変更点を示す。

サーバ	認可コードグラントからの変更点
認可サーバ	認可サーバの定義 クライアントの認証 トークンの取り消し（認可サーバ）
リソースサーバ	ユーザ情報の取得
クライアント	*OAuth2RestTemplate* の設定 リソースサーバへのアクセス

認可サーバの実装

認可コードグラントの実装から `oauth2-auth.xml` を変更し、DB 及び Service クラスから不要な定義を削除する必要がある。

`oauth2-auth.xml` 以外の実装は認可サーバの実装を参照し作成されたい。

認可サーバの定義

認可コードグラントからの `oauth2-auth.xml` の変更点を以下に示す。

- oauth2-auth.xml

```
<oauth2:authorization-server
    client-details-service-ref="clientDetailsService"
    token-endpoint-url="/oauth/token"
    token-services-ref="tokenServices">
    <oauth2:refresh-token />
    <oauth2:client-credentials />  <!-- (1) -->
</oauth2:authorization-server>
```

項番	説明
(1)	<oauth2:client-credentials />タグを使用して、クライアントクレデンシャルグラントをサポートする。

警告: <oauth2:client-credentials />タグと<oauth2:refresh-token />タグは上記の順番で設定する必要がある。

詳細は認可サーバで複数のグラントタイプをサポートする場合を参照されたい。

クライアントの認証

クライアントクレデンシャルグラントでは認可グラントで説明したとおりアクセストークンを直接発行する。そのため、リソースオーナに対して認可の要求は行われず、ユーザエージェントのリダイレクトは発生しない。

認可コードグラントのクライアントの認証で定義したテーブル、web_server_redirect_uris は不要となるため削除する。

トークンの取り消し（認可サーバ）

認可コードグラントの実装では、RevokeTokenServiceImpl.java にて認可サーバのトークンの取り消しに合わせて、認可情報の取り消しを行っている。

リソースオーナに対して認可の要求を行わないクライアントクレデンシャルグラントでは認可情報の取り消しが不要であるため、この処理を削除する。

以下に、認可情報の取り消しをコメントアウトした RevokeTokenServiceImpl.java を示す。

- RevokeTokenServiceImpl.java

```
@Service
@Transactional
public class RevokeTokenServiceImpl implements RevokeTokenService {

    @Inject
    ConsumerTokenServices consumerService;

    @Inject
    TokenStore tokenStore;

    @Inject
    ApprovalStore approvalStore;

    @Inject
    JodaTimeDateFactory dateFactory;

    public String revokeToken(String tokenValue, String clientId){

        OAuth2Authentication authentication = tokenStore.
↪readAuthentication(tokenValue);
        if (authentication != null) {
            if (clientId.equals(authentication.getOAuth2Request().getClientId())) {
                /* Authentication user = authentication.getUserAuthentication();
                if (user != null) {
                    Collection<Approval> approvals = new ArrayList<>();
                    for (String scope : authentication.getOAuth2Request().
↪getScope()) {
                        approvals.add(
                                new Approval(user.getName(), clientId, scope,␣
↪dateFactory.newDate(), ApprovalStatus.APPROVED));
                    }
                    approvalStore.revokeApprovals(approvals);
                } */
                consumerService.revokeToken(tokenValue);
                return "success";

            } else {
                return "invalid client";
            }
        } else {
            return "invalid token";
        }
    }
}
```

リソースサーバの実装

認可コードグラントの実装からユーザ情報の取得について変更する必要がある。

ユーザ情報の取得以外の実装はリソースサーバの実装を参照されたい。

ユーザ情報の取得

ユーザの認証が行われないクライアントクレデンシャルグラントではリソースオーナの情報が Spring Security に保持されない。

このため認可コードグラントでの実装のように、Controller クラスの引数として`@AuthenticationPrincipal` アノテーションを付与した `UserDetails` や `OAuth2Authentication` を指定してもリソースオーナの情報を受け取ることはできない。

代わりに、クライアント情報を保持しているため、Controller のメソッド引数に `String` を指定し`@AuthenticationPrincipal` アノテーションを付与することによりクライアント ID を取得することができる。

認可コードグラントからの変更点を以下に示す。

```
@RestController
@RequestMapping("api")
public class TodoRestController {

    // omitted

    @RequestMapping(value = "todos", method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public Collection<Todo> list(@AuthenticationPrincipal String clientId) { // (1)

        // omitted

    }
}
```

項番	説明
(1)	引数 `clientId` にクライアント ID が格納される。

クライアントの実装

認可コードグラントの実装から `oauth2-client.xml` を変更する必要がある。

`oauth2-client.xml` 以外の実装はクライアントの実装を参照し作成されたい。

OAuth2RestTemplate の設定

OAuth2RestTemplate の設定例を以下に示す。

- oauth2-client.xml

```
<oauth2:resource id="todoClientGrantResource" client-id="firstSecClient"
                 client-secret="firstSecSecret"
                 type="client_credentials"
                 access-token-uri="${auth.serverUrl}/oauth/token" /> <!-- (1) -->

<oauth2:rest-template id="todoClientGrantResourceRestTemplate" resource=
↪"todoClientGrantResource" />
```

項番	説明
(1)	type 属性にはグラントタイプを指定する。クライアントクレデンシャルグラントの場合 client_credentials を指定する。

注釈: Spring Security OAuth 以外のアーキテクチャで実装されている認可サーバに対してアクセストークンの発行を依頼する場合はリクエストパラメータが上記とは異なる可能性があるため注意されたい。その場合はアーキテクチャの仕様を確認し、必要なリクエストパラメータを設定されたい。

エラーハンドリング

トークンエンドポイント及びリソースサーバアクセス時のエラーハンドリング

クライアントクレデンシャルグラントではトークンエンドポイントおよびリソースサーバで発生するエラーはすべてシステムエラーとして扱えば良い。発生するエラーについては クライアントから認可サーバ、リソースサーバアクセス時に発生するエラーを参照されたい。

リソースオーナの認証

クライアントクレデンシャルグラントではクライアントがリソースオーナとなるため、リソースオーナの認証で説明したようなリソースオーナの認証を必要としない。

9.9.3 How to extend

認可サーバで複数のグラントタイプをサポートする場合

認可グラントで紹介したように認可サーバは複数のグラントタイプをサポートすることができる。

グラントタイプを複数指定する場合、タグの順番は XML スキーマで定められているため以下の表の番号順に設定する必要がある。

番号順に設定を行わないと、アプリケーション起動時に XML 解釈の失敗により `org.springframework.beans.factory.xml.XmlBeanDefinitionStoreException` が発生する。

順番	タグ
1	*<oauth2:authorization-code />*
2	*<oauth2:implicit />*
3	*<oauth2:refresh-token />*
4	*<oauth2:client-credentials />*
5	*<oauth2:password />*

例として、認可コードグラント、リソースオーナパスワードクレデンシャルグラントおよびリフレッシュトークンをサポートする場合の設定例を以下に示す。

- `oauth2-auth.xml`

```
<oauth2:authorization-server>
    <oauth2:authorization-code />  <!-- (1) -->
    <oauth2:refresh-token />  <!-- (2) -->
    <oauth2:password />  <!-- (3) -->
</oauth2:authorization-server>
```

項番	説明
(1)	*<oauth2:authorization-code />*タグを使用して、認可コードグラントをサポートする。
(2)	*<oauth2:refresh-token />*タグを使用して、リフレッシュトークンをサポートする。

Continued on next page

表 9.116 – continued from previous page

項番	説明
(3)	*<oauth2:password />*タグを使用して、リソースオーナパスワードクレデンシャルグラントをサポートする。

HTTP アクセスを介した認可サーバとリソースサーバの連携

リソースサーバと認可サーバは、認可サーバのチェックトークンエンドポイントにリソースサーバから HTTP アクセスを行うことで連携が可能である。

チェックトークンエンドポイントは、リソースサーバからアクセストークンの値を受け取り、リソースサーバの代わりにトークンの検証を行うエンドポイントである。チェックトークンエンドポイントは `TokenServices` を用いてアクセストークンの取得、検証を行い、検証に問題がなければアクセストークンに紐づく情報をリソースサーバに渡す。

なお、チェックトークンエンドポイントは RFC に定義されていない Spring Security OAuth 独自の機能であるが、本ガイドラインでは、RFC に定義されている他のエンドポイントと同様の形式でレスポンスを行うように設定する。

リソースサーバが認可サーバのチェックトークンエンドポイントにアクセスするためには `TokenServices` の実装クラスである `RemoteTokenServices` を使用する。`RemoteTokenServices` は `RestTemplate` を用いてチェックトークンエンドポイントへ HTTP アクセスを行い、アクセストークンに紐づく情報を取得する。アクセストークンの検証はチェックトークンエンドポイントが行うため、`RemoteTokenServices` では行わない。

以下に、実装例を示す。

認可サーバの設定

まず、認可サーバにトークンを検証するための `org.springframework.security.oauth2.provider.endpoint.CheckTokenEndpoint` クラスをコンポーネントとして登録する設定を行う。

- `oauth2-auth.xml`

```
<sec:http pattern="/oauth/*token*/**"
    authentication-manager-ref="clientAuthenticationManager">  <!-- (1) -->
    <sec:http-basic entry-point-ref="oauthAuthenticationEntryPoint" />
    <sec:csrf disabled="true"/>
    <sec:intercept-url pattern="/**" access="isAuthenticated()"/>
</sec:http>

<oauth2:authorization-server
     client-details-service-ref="clientDetailsService"
     user-approval-handler-ref="userApprovalHandler"
     token-services-ref="tokenServices"
     check-token-enabled="true"
     check-token-endpoint-url="/oauth/check_token">  <!-- (2) -->
    <oauth2:authorization-code />
    <oauth2:implicit />
```

```
    <oauth2:refresh-token />
    <oauth2:client-credentials />
    <oauth2:password />
</oauth2:authorization-server>

<!-- omitted -->
```

項番	説明
(1)	チェックトークンエンドポイントへのセキュリティ設定を行うために、エンドポイントとして/oauth/*token*/配下をアクセス制御の対象として指定する。
(2)	<oauth2:authorization-server>タグの check-token-enabled 属性に true を指定することで CheckTokenEndpoint がコンポーネントとして登録される。 チェックトークンエンドポイントとして、/oauth/check_token が設定される。

警告: チェックトークンエンドポイントのセキュリティ対策

認可サーバのチェックトークンエンドポイントは、リソースサーバのみアクセス可能とし、リソースサーバ以外が利用できないように制限する必要がある。本ガイドラインではチェックトークンエンドポイントに対して Basic 認証にてアクセス制限を行っているが、可能であればネットワークにて特定 URL に対する IP 制限など、より上位のアクセス制限を行うことを推奨する。

注釈: TokenServices は、共有 DB を介して連携させる場合と同様に DefaultTokenServices を使用する。TokenServices が参照する TokenStore はアプリケーションの要件に合ったインタフェースの実装クラスを使用する。

リソースサーバの設定

リソースサーバの設定ファイルに TokenServices として RemoteTokenServices を使用する設定を行う。

- oauth2-resource.xml

```
<bean id="tokenServices"
    class="org.springframework.security.oauth2.provider.token.RemoteTokenServices">
    <property name="checkTokenEndpointUrl" value="${auth.serverUrl}/oauth/check_
↪token" />  <!-- (1) -->
    <property name="clientId" value="${resource.clientId}" />  <!-- (2) -->
    <property name="clientSecret" value="${resource.clientSecret}" />  <!-- (3) -->
</bean>
```

項番	説明
(1)	認可サーバのチェックトークンエンドポイントにアクセスしてアクセストークンに紐付く情報を取得できるよう、RemoteTokenServices を Bean 定義する。 チェックトークンエンドポイントにアクセスするための URL を checkTokenEndpointUrl プロパティに設定する。
(2)	clientId プロパティに、認可サーバにてリソースサーバを識別する ID を設定する。 設定した ID はチェックトークンエンドポイントにアクセスする際に Basic 認証のユーザ名として使用される。 設定するクライアント ID は、認可サーバのクライアント情報に登録しておく必要がある。クライアント情報の登録についてはクライアントの認証を参照されたい。
(3)	clientSecret プロパティに、認可サーバにてリソースサーバの認証に用いるパスワードを設定する。 設定したパスワードは、チェックトークンエンドポイントにアクセスする際に Basic 認証のパスワードとして使用される。 設定するパスワードは、認可サーバのクライアント情報に登録されているパスワードと一致する必要がある。

注釈: チェックトークンエンドポイントでアクセストークンの検証エラーが発生した場合、RemoteTokenServices に HTTP ステータスコード 400(Bad Request) が返却される。RestTemplate のデフォルト実装では HTTP ステータスコード 400(Bad Request) が返却された場合、エラーハンドリングを行い HttpClientErrorException を発生させる。RemoteTokenServices がデフォルトで使用する RestTemplate はアクセストークンの検証エラーをクライアントサーバに連携するために、レスポンスの HTTP ステータスコードが 400 の場合はエラーハンドリングしないよう拡張されている。

RemoteTokenServices に RestTemplate をインジェクションした場合、この拡張が適用されなくなるため注意が必要である。

RemoteTokenServices をリソースサーバで使用した場合、ハンドラメソッドで@AuthenticationPrincipal アノテーションを String に引数アノテーションとして指定することでリソースオーナのユーザ名が取得できる。

実装例は以下のようになる。

```
@RestController
@RequestMapping("api")
public class TodoRestController {

    // omitted

    @RequestMapping(value = "todos", method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public Collection<Todo> list(@AuthenticationPrincipal String userName) { // (1)

        // omitted

    }
}
```

項番	説明
(1)	引数 userName にリソースオーナのユーザ名が格納される。

リソースサーバへの独自項目連携方法

認可サーバとリソースサーバ間で DB を共有しない構成の場合でも、ユーザ情報に付随する項目を認可サーバからリソースサーバに連携したいというケースはありうるが、リソースサーバの使用する TokenServices として RemoteTokenServices を使用する場合、リソースサーバのハンドラメソッド引数の@AuthenticationPrincipal アノテーションではユーザ名以外の情報を取得することができない。

そこで、ここでは RemoteTokenServices を使用してアクセストークンを連携すると

きに使用するクラスである org.springframework.security.oauth2.provider.token.DefaultAccessTokenConverter を拡張し、ユーザ名以外の情報をリソースサーバに連携する例を示す。

DefaultAccessTokenConverter とは

RemoteTokenServices を使用したアクセストークンの連携では、RestTemplate を使用してリソースサーバから認可サーバに対してアクセストークン値に紐づくリソースオーナ、クライアントの認証情報を要求し、結果を Map として取得する。このとき、DefaultAccessTokenConverter は、認可サーバでは認証情報から Map へ、リソースサーバでは Map から認証情報へ変換するためのコンバーターとしての役割を持つ。

これを利用し、認可サーバからの返却値を Map に追加するよう DefaultAccessTokenConverter の拡張を行うことで、認可サーバ、リソースサーバ間で連携するパラメータをカスタマイズすることが出来るようになる。

以下の説明では、認可サーバ側で DefaultAccessTokenConverter と、そのプロパティである org.springframework.security.oauth2.provider.token.DefaultUserAuthenticationConverter をそれぞれカスタマイズすることで、ユーザ情報に関連した独自項目と、それ以外の独自項目を連携する例を示す。

認可サーバの実装

認可サーバ側の実装方法について説明する。

まず、ユーザ情報に関連した独自項目を追加するため、DefaultUserAuthenticationConverter を拡張する。

- CustomUserAuthenticationConverter.java

```
public class CustomUserAuthenticationConverter extends␣
↪DefaultUserAuthenticationConverter {
    @Override
    public Map<String, ?> convertUserAuthentication(
            Authentication authentication) {
        Map<String, Object> response = new LinkedHashMap<>();
        response.put(USERNAME, authentication.getName());

        if (authentication.getAuthorities() != null &&
                !authentication.getAuthorities().isEmpty()) {
            response.put(AUTHORITIES, AuthorityUtils.authorityListToSet(
                    authentication.getAuthorities()));
        }
        response.put("company_id", "COMZZZ"); // (1)
        return response;
    }
}
```

項番	説明
(1)	リソースサーバに引き渡す情報を独自項目 company_id として定義し、response に設定する。 response に設定した情報は、チェックトークンエンドポイントのトークン検証時にレスポンス BODY として JSON 形式でリソースサーバへ返却される。

次に、ユーザ情報以外の独自項目を追加するため、DefaultAccessTokenConverter を拡張する。

- CustomAccessTokenConverter.java

```
public class CustomAccessTokenConverter extends DefaultAccessTokenConverter {

    @Override
    public Map<String, ?> convertAccessToken(OAuth2AccessToken token,␣
↪OAuth2Authentication authentication) {

        @SuppressWarnings("unchecked")
        Map<String, Object> response = (Map<String, Object>) super.
↪convertAccessToken(token, authentication);
        response.put("business_id","BIDXXX"); // (1)
        // omitted

        return response;
    }
}
```

項番	説明
(1)	リソースサーバに引き渡す情報を独自項目 business_id として定義し、response に設定する。 response に設定した情報は、チェックトークンエンドポイントのトークン検証時にレスポンス BODY として JSON 形式でリソースサーバへ返却される。

認可サーバの設定ファイルに、作成した CustomUserAuthenticationConverter、CustomAccessTokenConverter の設定を行う。

- oauth2-auth.xml

```
<oauth2:authorization-server
     client-details-service-ref="clientDetailsService"
     user-approval-handler-ref="userApprovalHandler"
     token-services-ref="tokenServices">  <!-- (1) -->
    <oauth2:authorization-code />
    <oauth2:implicit />
    <oauth2:refresh-token />
    <oauth2:client-credentials />
```

```
    <oauth2:password />
</oauth2:authorization-server>

<bean id="checkTokenEndpoint"
    class="org.springframework.security.oauth2.provider.endpoint.CheckTokenEndpoint
↪">  <!-- (2) -->
    <constructor-arg ref="tokenServices" />
    <property name="accessTokenConverter" ref="accessTokenConverter" />
</bean>

<bean id="accessTokenConverter"
    class="com.example.oauth2.auth.converter.CustomAccessTokenConverter">  <!--␣
↪(3) -->
    <property name="userTokenConverter">
        <bean
            class="com.example.oauth2.auth.converter.
↪CustomUserAuthenticationConverter" />
    </property>
</bean>
```

項番	説明
(1)	CheckTokenEndpoint の Bean 定義を (2) で独自で行っているため、 <oauth2:authorization-server>タグの check-token-enabled 属性は指定しない。 トークンチェックエンドポイントとして、/oauth/check_token が設定される。
(2)	CheckTokenEndpoint を Bean 定義する。 accessTokenConverter プロパティに (3) で定義している CustomAccessTokenConverter の Bean を指定することで CustomAccessTokenConverter と CustomUserAuthenticationConverter に追加した独自項目をリソースサーバに連携するようになる。
(3)	DefaultAccessTokenConverter を拡張した CustomAccessTokenConverter を Bean 定義する。 userTokenConverter プロパティに DefaultUserAuthenticationConverter を拡張した CustomUserAuthenticationConverter の Bean を指定する。

リソースサーバの実装

リソースサーバに、認可サーバから連携された情報をハンドラメソッド引数の@AuthenticationPrincipal アノテーションで取得できるよう機能の追加を行う。まず、@AuthenticationPrincipal アノテーションで取得する情報を保持する OauthUser クラスを作成する。

- ResourceOwner.java

```
public class OauthUser implements Serializable{

    private static final long serialVersionUID = 1L;

    private String username;

    private String companyId;

    private String businessId;

    private String clientId;

    // omitted

    public OauthUser(String username, String companyId, String businessId, String␣
↪clientId){
        this.username = username;
        this.companyId = companyId;
        this.businessId = businessId;
        this.clientId = clientId;
    }

    // Getters and Setters are omitted

}
```

@AuthenticationPrincipal アノテーションでユーザ情報が取得できるように設定を行う org.springframework.security.oauth2.provider.token.DefaultAccessTokenConverter を拡張し、ユーザ名以外の情報も取得できるよう機能の追加を行う。

- CustomUserAuthenticationConverter.java

```
public class CustomUserAuthenticationConverter extends␣
↪DefaultUserAuthenticationConverter{

    private Collection<? extends GrantedAuthority> defaultAuthorities; // (1)

    public void setDefaultAuthorities(String[] defaultAuthorities) {
        this.defaultAuthorities = AuthorityUtils.
↪commaSeparatedStringToAuthorityList(StringUtils
                .arrayToCommaDelimitedString(defaultAuthorities));
    }

     // (2)
    @Override
    public Authentication extractAuthentication(Map<String, ?> map) {
        if (map.containsKey(USERNAME)) {
            Collection<? extends GrantedAuthority> authorities =␣
↪getAuthorities(map);
            OauthUser user = new OauthUser(
                    (String) map.get(USERNAME),
```

```
                    (String) map.get("company_id"),
                    (String) map.get("business_id"),
                    (String) map.get("client_id")); // (3)

            // omitted

            return new UsernamePasswordAuthenticationToken(user, "N/A",␣
↪authorities); // (4)
        }
        return null;
    }

    private Collection<? extends GrantedAuthority> getAuthorities(Map<String, ?>␣
↪map) {
        if (!map.containsKey(AUTHORITIES)) {
            return defaultAuthorities;
        }
        Object authorities = map.get(AUTHORITIES);
        if (authorities instanceof String) {
            return AuthorityUtils.commaSeparatedStringToAuthorityList((String)␣
↪authorities);
        }
        if (authorities instanceof Collection) {
            return AuthorityUtils.commaSeparatedStringToAuthorityList(StringUtils
                    .collectionToCommaDelimitedString((Collection<?>)␣
↪authorities));
        }
        throw new IllegalArgumentException("Authorities must be either a String or␣
↪a Collection");
    }
}
```

項番	説明
(1)	`DefaultUserAuthenticationConverter` に実装されている `getAuthorities` メソッドが private で定義されているため、`getAuthorities` メソッドで使用される `defaultAuthorities` と `getAuthorities` メソッドを実装する。
(2)	認可サーバから連携された情報から認証情報を抽出するメソッド。
(3)	認可サーバから連携された情報を `OauthUser` クラスに設定する。

Continued on next page

表 9.123 – continued from previous page

項番	説明
(4)	UsernamePasswordAuthenticationToken の第一引数に OauthUser を設定することで、認可サーバから連携された情報を@AuthenticationPrincipal アノテーションで取得できるようになる。

リソースサーバの設定ファイルに、CustomUserAuthenticationConverter の設定を行う。

- oauth2-resource.xml

```
<bean id="tokenServices"
    class="org.springframework.security.oauth2.provider.token.RemoteTokenServices">
    <property name="checkTokenEndpointUrl" value="${auth.serverUrl}/oauth/check_
↪token" />
    <property name="accessTokenConverter" ref="accessTokenConverter" />
</bean>

<bean id="accessTokenConverter"
    class="org.springframework.security.oauth2.provider.token.
↪DefaultAccessTokenConverter">
    <property name="userTokenConverter">
        <bean class="com.example.oauth2.resource.converter.
↪CustomUserAuthenticationConverter"/>  <!-- (1) -->
    </property>
</bean>
```

項番	説明
(1)	DefaultAccessTokenConverter を Bean 定義する。ユーザ名以外の情報を@AuthenticationPrincipal アノテーションで取得できるようにするため、userTokenConverter プロパティに CustomUserAuthenticationConverter クラスを指定する。

認可サーバにおけるパスのカスタマイズ

認可サーバではエンドポイントと、特定の状況が起きたときのフォワード先についてパスを変更することができる。本節では、認可サーバにおけるパス、および関連箇所の設定変更方法を説明する。

カスタマイズ可能なパス

How to use にて説明したとおり、認可サーバでは<oauth2:authorization-server>タグによる定義を行うことで、RFC に準拠したエンドポイントや、認可サーバ内でフォワードされたリクエストを処理するための Controller がコンポーネントとして登録される。また、認可サーバがクライアントやリソースサーバに公開するエンドポイントは、<oauth2:authorization-server>タグの属性値を変更することにより、カスタマイズが可能である。

以下に、コンポーネントとして登録されるエンドポイント、およびフォワード先のデフォルトパスと、エンドポイントのカスタマイズ時に変更する<oauth2:authorization-server>タグの属性値を示す。

表 9.125: エンドポイント

名前	属性値	デフォルトパス	説明
認可エンドポイント	authorization-endpoint-url	/oauth/authorize	クライアントがリソースオーナから認可を得るために利用するエンドポイント。
トークンエンドポイント	token-endpoint-url	/oauth/token	クライアントがアクセストークンを発行するために利用するエンドポイント。
チェックトークンエンドポイント (*HTTP アクセスを介した認可サーバとリソースサーバの連携*の実装を行っている場合のみ設定する。)	check-token-endpoint-url	/oauth/check_token	リソースサーバがアクセストークンを検証するために利用するエンドポイント。 HTTP アクセスを介してアクセストークンの連携を行う場合に利用する。なお、本エンドポイントは、 RFC の定めるエンドポイントではなく、Spring Security OAuth が独自に拡張したエンドポイントである。

表 9.126: フォワード先

名前	属性値	デフォルトパス	説明
認可取得時のフォワード先	user-approval-page	/oauth/confirm_access	リソースオーナに認可画面を返却するために利用するフォワード先。 認可サーバの処理内で利用するパスであり、クライアントやリソースサーバには公開しない。

Continued on next page

表 9.126 – continued from previous page

名前	属性値	デフォルトパス	説明
不正クライアントエラー発生時のフォワード先	error-page	/oauth/error	リソースオーナに認可エンドポイントにおけるエラーを通知するために利用するフォワード先。 認可サーバの処理内で利用するパスであり、クライアントやリソースサーバには公開しない。

これらのパスは認可サーバ側の設定を変更することにより、カスタマイズが可能である。具体的な設定方法を以降に示す。

パスのカスタマイズ

エンドポイントのカスタマイズ

前述のとおり、各エンドポイントは、<oauth2:authorization-server>タグの属性値を変更することにより、カスタマイズが可能である。

エンドポイントを変更する際の実装例を以下に示す。

- oauth2-auth.xml

```
<oauth2:authorization-server
    client-details-service-ref="clientDetailsService"
    user-approval-handler-ref="userApprovalHandler"
    token-services-ref="tokenServices"
    token-endpoint-url="/api/token"
    authorization-endpoint-url="/api/authorize"
    check-token-endpoint-url="/api/check_token"
    check-token-enabled="true">  <!-- (1) -->
    <!-- omitted -->
</oauth2:authorization-server>

<sec:http pattern="/api/*token*/**"
    authentication-manager-ref="clientAuthenticationManager" realm="Realm">  <!--
↪(2) -->
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	<oauth2:authorization-server>タグの変更したいエンドポイントに対する属性値（authorization-endpoint-url、token-endpoint-url、check-token-endpoint-url）に URL を設定する。
(2)	アクセストークン操作に関するエンドポイントの URL を含むように、アクセス制御の対象を変更する。

変更したエンドポイントに合わせて、エンドポイントを参照する設定を変更する必要がある。認可サーバでは、リソースオーナの認証の spring-security.xml とスコープ認可画面のカスタマイズの oauthConfirm.jsp のエンドポイント設定を参照されたい。

また、クライアントの実装を行っている場合は、認可リクエストの設定として認可サーバのエンドポイントを設定しているため、*OAuth2RestTemplate* の設定の oauth2-client.xml も参照されたい。

フォワード先のカスタマイズ

リソースオーナからの認可の取得時に、デフォルト設定では/oauth/confirm_access にフォワードされる。また、認可エンドポイントで不正クライアントエラーが発生した際には、デフォルト設定では/oauth/error にフォワードされる。これらのフォワード先は変更が可能である。

フォワード先を変更する際の実装例を以下に示す。

- oauth2-auth.xml

```
<oauth2:authorization-server
    client-details-service-ref="clientDetailsService"
    user-approval-handler-ref="userApprovalHandler"
    token-services-ref="tokenServices"
    error-page="forward:/api/error"
    user-approval-page="forward:/api/confirm_access">  <!-- (1) -->
    <!-- omitted -->
</oauth2:authorization-server>
```

項番	説明
(1)	<oauth2:authorization-server>タグの変更したいフォワード先に対する属性値（error-page または user-approval-page）に URL を設定する。設定する URL は先頭に forward:を付ける必要がある。

フォワード先を変更した際は対応するコントローラについても変更する必要があるため注意されたい。本実装例では以下を参照されたい。

- スコープ認可画面のカスタマイズの OAuth2ApprovalController.java
- 認可リクエスト時のエラー画面のカスタマイズの OAuth2ErrorController.java

9.9.4 Appendix

クライアントから認可サーバ、リソースサーバアクセス時に発生するエラー

本ガイドラインではクライアントから認可サーバ、リソースサーバアクセス時に発生するエラーについて、リソースオーナの操作によって発生するエラーのハンドリング方法を記載している。リソースオーナの操作起因以外のエラーも含め、発生するエラーについて以下に説明する。

認可エンドポイントで発生するエラー

認可エンドポイントで発生するエラーは不正クライアントエラーとそれ以外に分類され、不正クライアントエラーの場合は認可サーバでエラー画面にフォワードされることでエラー通知される。不正クライアントエラーの詳細については不正クライアントエラーを参照されたい。認可エンドポイントで発生する不正クライアントエラーについて以下で説明する。

表 9.129: 認可エンドポイントで発生する不正クライアントエラー

項番	発生するエラー	説明
(1)	クライアント未存在エラー NoSuchClientException	ユーザエージェントを認可サーバへアクセスさせたクライアントが、認可サーバの保持するクライアント情報に登録されていない場合に発生する。

Continued on next page

表 9.129 – continued from previous page

項番	発生するエラー	説明
(2)	リダイレクト URI 不一致エラー `org.springframework.security.oauth2.common.exceptions.RedirectMismatchException`	クライアントがパラメータとして認可サーバに渡したリダイレクト URI のホストとルートパスが、認可サーバに登録されているリダイレクト URI と一致しない場合に発生する。

発生したエラーが不正クライアントエラー以外の場合はクライアントにリクエストパラメータとしてエラー情報（`error`、`error_description`）が通知され、`OAuth2RestTemplate` にて例外に復元されスローされる。認可エンドポイントで発生する不正クライアントエラー以外のエラーについて以下で説明する。

表 9.130: 認可エンドポイントで発生する不正クライアントエラー以外のエラー

項番	発生するエラー	説明
(1)	スコープ検証エラー `org.springframework.security.oauth2.common.exceptions.InvalidScopeException`	クライアントが指定したスコープが、認可サーバの保持するクライアント情報に存在しない場合に発生する。

Continued on next page

表 9.130 – continued from previous page

項番	発生するエラー	説明
(2)	リクエスト検証エラー `org.springframework.security.oauth2.common.exceptions.InvalidRequestException`	認可後の遷移先を示すリダイレクト URI が設定されていない場合や認可画面を表示せずに認可が行われた場合に発生する。
(3)	レスポンスタイプエラー `org.springframework.security.oauth2.common.exceptions.UnsupportedResponseTypeException`	クライアントが指定した `response_type` パラメータを認可サーバがサポートしない場合に発生する。
(4)	グラントタイプエラー `InvalidGrantException`	クライアントが使用できるグラントタイプがない場合や、認可コードグラント、インプリシットグラント以外のリダイレクト URI を使用出来ないグラントタイプが指定された場合に発生する。
(5)	リソースオーナによる認可拒否 `UserDeniedAuthorizationException`	リソースオーナがスコープ認可画面で、クライアントが指定したスコープを全て拒否した場合に発生する。

Continued on next page

表 9.130 – continued from previous page

項番	発生するエラー	説明
(6)	リソースオーナ未認証エラー `org.springframework.security.authentication.InsufficientAuthenticationException`	認可エンドポイントへアクセス時にリソースオーナがユーザ認証を行っていない場合に発生する。認可サーバで適切なセキュリティ設定を行っていれば発生しない。

トークンエンドポイントで発生するエラー

トークンエンドポイントでエラーが発生した場合、クライアントの `OAuth2RestTemplate` で `OAuth2AccessDeniedException` にラップされる。トークンエンドポイントで発生するエラーについて以下で説明する。

表 9.131: トークンエンドポイントで発生するエラー

項番	発生するエラー	説明
(1)	Basic 認証エラー `HttpClientErrorException`	クライアントの Basic 認証に失敗した場合に発生する。
(2)	クライアント未認証エラー `InsufficientAuthenticationException`	トークンエンドポイントの処理前に認証が行われていない場合に発生する。認可サーバで適切なセキュリティ設定を行っていれば発生しない。
(3)	スコープ検証エラー `InvalidScopeException`	クライアントが指定したスコープが、認可サーバの保持するクライアント情報に存在しない場合に発生する。 認可コードグラント使用時は認可エンドポイントでエラーとなるため `OAuth2RestTemplate` を使用していれば発生しない。

Continued on next page

表 9.131 – continued from previous page

項番	発生するエラー	説明
(4)	リソースオーナ認証エラー `InvalidGrantException`	リソースオーナパスワードクレデンシャルグラント使用時に提示したリソースオーナの認証情報に誤りがある場合に発生する。
(5)	認可コード未存在エラー `InvalidGrantException`	認可コードグラント使用時に、クライアントから認可サーバに渡した認可コードが認可サーバに存在しない場合に発生する。

本ガイドラインでは、クライアントとして Spring Security OAuth の `OAuth2RestTemplate` を利用する前提であり、`OAuth2RestTemplate` が自動的に管理している設定に関してエラーが発生しない構造となっている。Spring Security OAuth を使用せずにクライアントを開発する場合のために、Spring Security OAuth の `OAuth2RestTemplate` を使用しない場合に発生するエラーについて以下で説明する。

表 9.132: 【参考】トークンエンドポイントで発生するエラー（**OAuth2RestTemplate** を使用しない場合）

項番	発生するエラー	説明
(1)	クライアント検証エラー `org.springframework.security.oauth2.common.exceptions.InvalidClientException`	認証済みクライアントとリクエストパラメータに設定されているクライアントが不一致の場合に発生する。 具体的にはトークンエンドポイントで認証したクライアントのクライアント ID と以下のクライアント ID が一致しない場合に発生する。 • 認可リクエストのリクエストパラメータに設定されているクライアント ID • アクセストークンリクエストのリクエストパラメータに設定されているクライアント ID

Continued on next page

表 9.132 – continued from previous page

項番	発生するエラー	説明
(2)	グラントタイプエラー `org.springframework.security.oauth2.common.exceptions.UnsupportedGrantTypeException`	トークンエンドポイントへのリクエストに指定した `grant_type` パラメータに誤りがある場合に発生する。 • クライアントが指定したグラントタイプが認可サーバがサポートしていないグラントタイプの場合 • リフレッシュトークンをサポートしていない認可サーバにリフレッシュトークンを使用した場合
(3)	リクエスト検証エラー `InvalidRequestException`	トークンエンドポイントへのリクエストに `grant_type` パラメータが存在しない場合に発生する。
(4)	リダイレクト URI 検証エラー `RedirectMismatchException`	認可エンドポイントアクセス時に使用したリダイレクト URI と、トークンエンドポイントアクセス時にパラメータに設定したリダイレクト URI が一致しない場合に発生する。

リソースサーバで発生するエラー

リソースサーバでエラーが発生した場合、クライアントの `OAuth2RestTemplate` の処理は発生したエラーによって異なる。リソースサーバで発生するエラーについて以下で説明する。

表 9.133: リソースサーバで発生するエラー

項番	発生するエラー	説明
(1)	スコープ検証エラー `org.springframework.security.oauth2.common.exceptions.InsufficientScopeException`	保護されたリソースへのアクセスに必要なスコープが、アクセストークンが保持するスコープに存在しなかった場合に発生する。
(2)	リソース ID 検証エラー `UserDeniedAuthorizationException`	保護されたリソースへのアクセスに必要なリソース ID が、アクセストークンに紐付くクライアント情報のリソース ID に存在しなかった場合に発生する。 クライアントの `OAuth2RestTemplate` で `OAuth2AccessDeniedException` にラップされる。

Continued on next page

表 9.133 – continued from previous page

項番	発生するエラー	説明
(3)	アクセストークン検証エラー `InvalidTokenException`	リソースサーバが受け取ったアクセストークンの正当性を確認出来ない場合（異常系）や、アクセストークンの有効期限が切れている場合（正常系）に発生する。 アクセストークン検証エラーは正常系と異常系が混在しているため、インプリシットグラント等 `OAuth2RestTemplate` を利用しない場合にこのエラーを受け取った場合、アクセストークンの有効期限切れかどうか判定する必要がある。 具体的な実装例については、*JavaScript を使用したリソースサーバへのアクセス*を参照されたい。 --- **注釈:** 本ガイドラインにおいてインプリシットグラント以外のグラントタイプでは `OAuth2RestTemplate` を利用している。`OAuth2RestTemplate` は、リソースサーバへのアクセス前にアクセストークンの有効期限チェックを行い、有効期限が切れていた場合にはアクセストークンの再発行を行っているため、通常はリソースサーバでアクセストークンの有効期限切れは発生しない。`OAuth2RestTemplate` はアクセストークン取得後すぐにリソースにアクセスする構造のため、アクセストークンの取得直後に `OAuth2RestTemplate` では有効期限内にも関わらず、リソースサーバで有効期限切れととなる状態は、アクセストークンの発行に何らかの問題があることになる。この場合、`OAuth2RestTemplate` で `AccessTokenRequiredException` がスローされる。 ---
(4)	チェックトークンエンドポイント Basic 認証エラー `HttpClientErrorException`	*HTTP アクセスを介した認可サーバとリソースサーバの連携*において、チェックトークンエンドポイントへアクセス時に Basic 認証に失敗した場合に発生する。 クライアントでエラーハンドリングを行う場合は、リソースサーバからクライアントに適切なエラーを返却するようリソースサーバでエラーハンドリングを行う必要がある。

Spring Security OAuth の拡張について

RFC 6749 の 8. Extensibility には OAuth 2.0 の拡張仕様が定められいる。OAuth 2.0 による認可機能を提供するアプリケーションではこれに沿ってカスタマイズすることが可能である。

Spring Security OAuth では、前述の RFC 6749 に規定された拡張ポイントに対してどのようにサポートしているかは明示的に公表されていないが、以下の拡張ポイントがサポートされていると考えられる。

- 8.1 Defining Access Token Types

- 8.2 Defining New Endpoint Parameters
- 8.3 Defining New Authorization Grant Types

本節では、特に 8.2 Defining New Endpoint Parameters に関連するアクセストークンリクエストとそのレスポンスの拡張ポイントについて主要なコンポーネントと処理の流れを解説する。

アクセストークンリクエストとそのレスポンスの拡張

アクセストークンリクエストとそのレスポンスの拡張ポイントとして、任意のパラメータやヘッダをリクエストやレスポンスに付与することができる。

ただし、認可リクエストには前述したような拡張ポイントが設けられていない。そのため、例えば認可リクエストに独自のパラメータを追加したい場合には、Spring Security OAuth の API 自体を改修する必要があり、比較的大きな改修が必要となることに注意されたい。

クライアント

クライアントでは、アクセストークンリクエストの拡張ポイントとして、`RequestEnhancer` インタフェースが用意されている。以下に拡張ポイントに関連する主要なコンポーネント、およびその関連図を示す。

表 9.134: クライアントの主要なコンポーネント

クラス・インタフェース名	説明
`OAuth2RestTemplate`	`RestTemplate` を拡張し OAuth 2.0 向けの機能を追加したクラス。 グラントタイプに応じたアクセストークンの取得など、OAuth 2.0 独自の機能を持つ。
`OAuth2ProtectedResourceDetails`	リソースサーバが保持するリソースにアクセスするための詳細情報のインタフェース。 グラントタイプに応じたパラメータを保持するために派生クラスが提供されており、認可コードグラントの場合は `AuthorizationCodeResourceDetails` が実装クラスとなる。
`AccessTokenProvider`	認可サーバよりアクセストークンを取得するための機能を提供するインタフェース。 グラントタイプに応じた派生クラスが提供されており、グラントタイプごとのプロバイダは必ず `AccessTokenProvider` を実装して `OAuth2AccessTokenSupport` を継承している。 認可コードグラントの場合は `AuthorizationCodeAccessTokenProvider` が実装クラスとなる。

Continued on next page

表 9.134 – continued from previous page

クラス・インタフェース名	説明
OAuth2AccessTokenSupport	認可サーバよりアクセストークンを取得するための機能を提供する基底クラス。 グラントタイプ共通の処理として、認可サーバに対してアクセストークンリクエストを送信する機能を提供している。 アクセストークンリクエストの生成には、後述する ClientAuthenticationHandler と RequestEnhancer を使用している。
AuthorizationCodeAccessTokenProvider	認可コードグラントにてアクセストークンを取得するための機能を提供するクラス。 認可コードグラント独自の処理として、認可リクエストを作成する機能を提供している。
ClientAuthenticationHandler	OAuth2ProtectedResourceDetails をもとにヘッダ、またはフォームにクライアント認証情報を設定するための機能を提供するインタフェース。DefaultClientAuthenticationHandler がデフォルトの実装クラスとなる。

Continued on next page

表 9.134 – continued from previous page

クラス・インタフェース名	説明
`RequestEnhancer`	アクセストークンリクエストおよび `OAuth2ProtectedResourceDetails` をもとにヘッダ、またはフォームにパラメータを設定するための機能を提供するインタフェース。`DefaultRequestEnhancer` がデフォルトの実装クラスとなる。 **アクセストークンリクエストにおける拡張ポイントの一つ。** なお、リクエストパラメータに独自の項目を追加したい場合は、デフォルトで使用される `DefaultRequestEnhancer` の Bean 定義を変更するだけで良く、独自に `RequestEnhancer` の実装クラスを作成する必要はない。 具体的には、以下のように Bean 定義すれば良い。 (1) `DefaultRequestEnhancer` `parameterIncludes` 属性に追加するパラメータのキー値を設定する (2) `AuthorizationCodeAccessTokenProvider` `tokenRequestEnhancer` 属性に (1) の `DefaultRequestEnhancer` を設定する (3) `OAuth2RestTemplate` `access-token-provider` 属性に (2) の `AuthorizationCodeAccessTokenProvider` を設定する

例として、認可コードグラントにおける、クライアントからリソースにアクセスする時のフローを以下に示す。

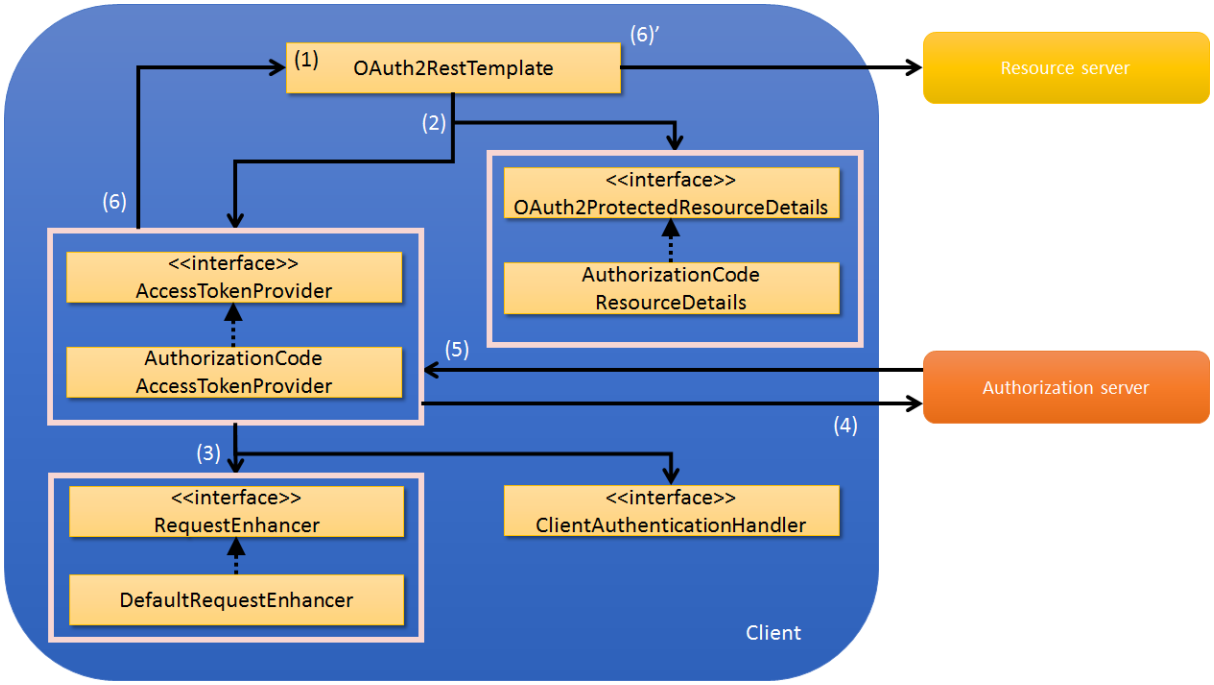


表 9.135: クライアントの動き（アクセストークンリクエスト）

項番	説明
(1)	リソースサーバが保持しているリソースにアクセスする際、`OAuth2RestTemplate` の呼び出しが行われることでクライアント側の処理が開始される。 認可コード発行までのクライアント側のフローについては、クライアントを参照されたい。
(2)	認可コード発行後、認可サーバのリダイレクトにより再度 `OAuth2RestTemplate` の呼び出しが行われる。このとき、認可コードはセッション (`OAuth2ClientContext`) に保持される。 `OAuth2RestTemplate` では、`OAuth2ProtectedResourceDetails` に定義しているグラントタイプに応じて `AccessTokenProvider` の呼び出しを行う。 `AuthorizationCodeAccessTokenProvider` が `AccessTokenProvider` の実装クラスとして呼び出される。
(3)	`AuthorizationCodeAccessTokenProvider` では、`OAuth2AccessTokenSupport` が保持する `ClientAuthenticationHandler` からはクライアント認証情報を、`RequestEnhancer` からはリクエストパラメータをヘッダ、またはフォームに設定し、アクセストークンリクエストを作成する。
(4)	`OAuth2AccessTokenSupport` は (3) にて作成したリクエストを使用し、認可サーバに対してアクセストークンリクエストを送信する。

Continued on next page

表 9.135 – continued from previous page

項番	説明
(5)	認可サーバでは、クライアントを認証し認可グラントの正当性を確認する。 認可グラントが正当な場合、アクセストークンを発行する。 アクセストークン発行のフローについては、アクセストークンの発行を参照されたい。
(6)	AccessTokenProvider は取得したアクセストークンを OAuth2RestTemplate に返却する。 OAuth2RestTemplate では、セッション (OAuth2ClientContext) にアクセストークンを保持し、それを用いてリソースサーバに対してリクエストを送信する。

認可サーバ

認可サーバでは、アクセストークンレスポンスの拡張ポイントとして、TokenEnhancer インタフェースが用意されている。以下に拡張ポイントに関連する主要なコンポーネント、およびその関連図を示す

表 9.136: 認可サーバの主要なコンポーネント

クラス・インタフェース名	説明
AbstractEndpoint	認可サーバにおいて、クライアントからのリクエストを受けるエンドポイントの共通的な機能を提供する基底クラス。TokenEndpoint が実装クラスとなる。
TokenEndpoint	クライアントからのリクエストに対し、アクセストークンを発行するためのエンドポイントの機能を提供するクラス。 クライアントや、リクエストパラメータに含まれるパラメータを検証する機能を持つ。

Continued on next page

表 9.136 – continued from previous page

クラス・インタフェース名	説明
ClientDetailsService	OAuth 2.0 機能を利用するクライアントを検証するための機能を提供するインタフェース。 クライアント情報を管理する媒体ごとに派生クラスが提供されており、JdbcClientDetailsService、InMemoryClientDetailsService が実装クラスとなる。
TokenGranter	アクセストークンを発行するための機能を提供するインタフェース。 グラントタイプに応じた派生クラスが提供されており、グラントタイプごとのプロバイダは必ず TokenGranter を実装して AbstractTokenGranter を継承している。 認可コードグラントの場合は AuthorizationCodeTokenGranter が実装クラスとなる。
AbstractTokenGranter	アクセストークンを発行するための機能を提供する基底クラス。 リクエストパラメータからクライアント情報を取得し、グラントタイプの検証と固有処理の呼び出しを行う機能を持つ。
AuthorizationCodeTokenGranter	認可コードグラントにてアクセストークンを発行するための機能を提供するクラス。 認可コードグラント独自の処理として、アクセストークンリクエストに含まれる認可コードやリダイレクト URI を検証する機能を提供している。
AuthorizationCodeServices	認可コードグラントにて認可コードの発行、および管理をするための機能を提供するインタフェース。 認可コードを管理する媒体ごとに派生クラスが提供されており、JdbcAuthorizationServerTokenServices、InMemoryAuthorizationServerTokenServices が実装クラスとなる。

Continued on next page

表 9.136 – continued from previous page

クラス・インタフェース名	説明
AuthorizationServerTokenServices	認可サーバとして必要となる、アクセストークンやリフレッシュトークンを発行するための機能を提供するインタフェース。 DefaultTokenServices がデフォルトの実装クラスとなる。
TokenEnhancer	AuthorizationServerTokenServices によって生成されたトークンに追加情報等を付与するための機能を提供するインタフェース。 本インタフェースを拡張することでアクセストークンとして管理する情報、およびクライアントに返却する情報に独自パラメータを付与することが出来る。 **アクセストークンレスポンスにおける拡張ポイントの一つ。**
TokenStore	アクセストークンを管理するための機能を提供するインタフェース。 アクセストークンを管理する媒体ごとに派生クラスが提供されており、JdbcTokenStore、InMemoryTokenStore などが実装クラスとなる。

例として、認可コードグラントにおける、認可サーバのトークンエンドポイントアクセス時のフローを以下に示す。

表 9.137: 認可サーバの動き（アクセストークンレスポンス）

項番	説明
(1)	TokenEndpoint の呼び出しが行われることで認可サーバ側の処理が開始される。
(2)	TokenEndpoint では、AbstractEndpoint が保持する ClientDetailsService の呼び出しを行いアクセストークンリクエストに含まれるクライアント情報やパラメータの検証後、TokenGranter の呼び出しを行う。 AuthorizationCodeTokenGranter が TokenGranter の実装クラスとなる。

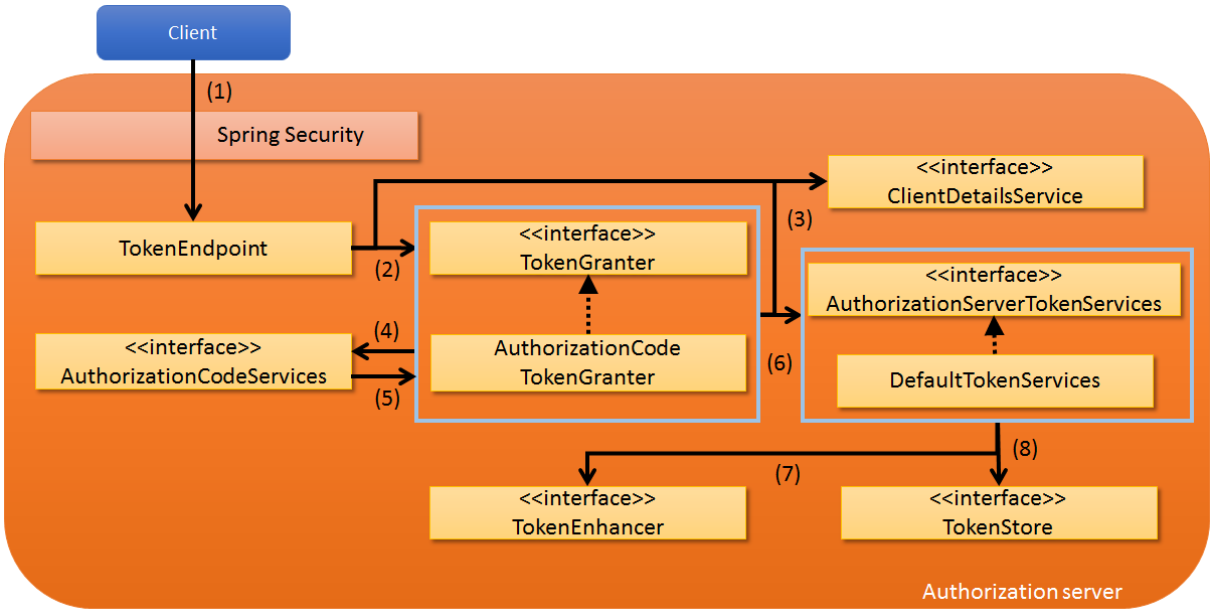
Continued on next page

表 9.137 – continued from previous page

項番	説明
(3)	TokenGranter では、まず基底クラスである AbstractTokenGranter の処理が行われる。 AbstractTokenGranter では、ClientDetailsService の呼び出しを行うことでリクエストパラメータに指定されたクライアント情報を取得し、グラントタイプの検証後、グラントタイプ固有の処理（AuthorizationCodeTokenGranter）の呼び出しを行う。
(4)	AuthorizationCodeTokenGranter では、リクエストパラメータから取得した認可コードを指定して AuthorizationCodeServices の呼び出しを行い、発行済み認可コードの削除を行うとともに認可リクエスト時の情報を取得する。
(5)	AuthorizationCodeServices では、認可コードグラントのアクセストークンリクエストパラメータとして指定されたクライアント ID、リダイレクト URI と (4) にて取得した認可リクエストの内容を比較し、アクセストークンリクエストの妥当性の検証を行う。 検証結果に問題がない場合、リクエストパラメータと認可リクエスト時に取得したリソースオーナの認証情報（ユーザ名など）より、認証情報を作成し呼び出し元に返却する。
(6)	AbstractTokenGranter では、AuthorizationServerTokenServices のアクセストークン取得処理の呼び出しを行う。 DefaultTokenServices が AuthorizationServerTokenServices の実装クラスとなる。
(7)	DefaultTokenServices ではアクセストークンを作成する。 TokenEnhancer が指定されている場合、アクセストークンを拡張し、追加パラメータなどを付与することが出来る。
(8)	TokenStore の呼び出しを行い、(5) にて作成した認証情報とともに (7) で作成したアクセストークンを登録し、アクセストークンを呼び出し元の TokenEndpoint に返却する。 TokenEndpoint では (7) で作成したアクセストークンをもとにレスポンスを作成し、リクエストの応答を行う。

9.10 代表的なセキュリティ要件の実装例

9.10.1 はじめに



この章で説明すること

- TERASOLUNA Server Framework for Java (5.x) を利用して代表的なセキュリティ要件を満たすための実装方法の例
- アプリケーションの説明 に示すサンプルアプリケーションを題材として、実装方法とソースコードの説明を行う

警告:

- この章で説明している実装方法はあくまでも一例であり、実際の開発においては個別の要件を考慮して実装する必要がある
- セキュリティ対策の網羅的な実施を保証するものではないため、必要に応じて追加の対策を検討すること

注釈: 本章は Java8 で動作することを前提に、Java8 で導入された *JSR-310 Date and Time API* を使用している。Java8 未満の環境を利用する場合は、適宜 *Joda Time* 等に読み替えられたい。

対象読者

- アプリケーション開発 の内容を理解していること
- *Spring Security* 概要, 認証, 認可 の内容を理解していること
- *Spring Security* チュートリアル を実施済みのこと

9.10.2 アプリケーションの説明

本章では、代表的なセキュリティ要件を満たすサンプルアプリケーションを題材として、セキュリティ対策の具体的な実装方法の例について説明する。

以下に本章で実装例を解説するセキュリティ要件の一覧を示し、題材となるサンプルアプリケーションの機能、認証・認可に関する仕様を示す。

以降、このサンプルアプリケーションを本アプリケーションと呼ぶ。

セキュリティ要件

本アプリケーションが満たすセキュリティ要件の一覧を以下に示す。分類ごとに、実装方法とコード解説 にて実装例の解説を行う。

項番	分類	要件	概説
(1)	パスワード変更の強制・促進	初期パスワード使用時のパスワード変更の強制	初期パスワードを使用して認証成功した際に、パスワードの変更を強制する
(2)		期限切れパスワードの変更の強制	一定期間パスワードを変更していないユーザに対して、認証成功時にパスワードの変更を強制する 本アプリケーションでは、管理ユーザのみを対象とする
(3)		パスワード変更を促すメッセージの表示	一定期間パスワードを変更していないユーザに対して、認証成功時にパスワードの変更を促すメッセージを表示する
(4)	パスワードの品質チェック	パスワードの最小文字数指定	パスワードとして設定できる文字数の最小値を指定する
(5)		パスワードの文字種別指定	パスワード中に含めなければならない文字種別（英大文字、英小文字、数字、記号）を指定する

Continued on next page

表 9.138 – continued from previous page

項番	分類	要件	概説
(6)		ユーザ名を含むパスワードの禁止	パスワード中にアカウントのユーザ名を含めることを禁止する
(7)		管理ユーザパスワードの再使用禁止	管理ユーザが、以前使用したパスワードを短期間のうちに再使用することを禁止する
(8)	アカウントのロックアウト	アカウントロックアウト	あるアカウントが短期間の間に一定回数以上認証に失敗した場合、そのアカウントを認証不能な状態（ロックアウト状態）にする
(9)		アカウントロックアウト期間の指定	アカウントのロックアウト状態の継続時間を指定する
(10)		管理ユーザによるロックアウトの解除	管理ユーザは任意のアカウントのロックアウト状態を解除できる
(11)	最終ログイン日時の表示	前回ログイン日時の表示	あるアカウントで認証成功した後、トップ画面にそのアカウントが前回認証に成功した日時を表示する
(12)	パスワード再発行のための認証情報の生成	パスワード再発行用 URL へのランダム文字列の付与	不正なアクセスを防ぐため、パスワード再発行画面にアクセスするための URL に十分に推測困難な文字列を付与する
(13)		パスワード再発行用秘密情報の発行	パスワード再発行時のユーザ確認に用いるために、事前に十分に推測困難な秘密情報（ランダム文字列）を生成する

Continued on next page

表 9.138 – continued from previous page

項番	分類	要件	概説
(14)	パスワード再発行のための認証情報の配布	パスワード再発行画面 URL のメール送付	パスワード再発行ページにアクセスするための URL は、アカウントの登録済みメールアドレスへ送付する
(15)		パスワード再発行画面の URL と秘密情報の別配布	パスワード再発行画面の URL の漏えいに備え、秘密情報はメール以外の方法でユーザに配布する
(16)	パスワード再発行実行時の検査	パスワード再発行用の認証情報に対する有効期限の設定	パスワード再発行画面の URL と秘密情報に有効期限を設定し、有効期限が切れた場合はパスワード再発行画面の URL と秘密情報を使用不能にする
(17)	パスワード再発行の失敗上限回数の設定	パスワード再発行の失敗上限回数の設定	パスワード再発行時の認証に一定回数失敗した場合、パスワード再発行画面の URL と秘密情報を使用不能にする
(18)	セキュリティ観点での入力値チェック	リクエストパラメータに対する共通的な禁止文字の設定	リクエストパラメータに含まれる文字列に対し、アプリケーション全体で共通の禁止文字を設定する
(19)		アップロードファイル名に対する共通的な禁止文字列の設定	アップロードされるファイル名に対し、アプリケーション全体で共通の禁止文字を設定する
(20)		制御文字の入力チェック	入力値に制御文字が含まれていないかをチェックする

Continued on next page

表 9.138 – continued from previous page

項番	分類	要件	概説
(21)		ファイル拡張子の入力チェック	アップロードされるファイルの拡張子がアプリケーションで許可されたものであるかをチェックする
(22)		ファイル名の入力チェック	アップロードされるファイルのファイル名がアプリケーションで許可されたパターンと一致するかをチェックする
(23)		URL のドメインに対する入力チェック	入力された URL のドメインがアプリケーションで許可されたものであるかをチェックする
(24)		メールアドレスのドメインに対する入力チェック	入力されたメールアドレスのドメインがアプリケーションで許可されたものであるかをチェックする
(25)	監査ログ出力	監査ログ出力	各リクエストに対し、日時、ユーザ名、操作内容、操作結果をログ出力する

機能

本アプリケーションは、*Spring Security チュートリアル* で作成したアプリケーションに加え、以下の機能を持つ。

機能名	説明
アカウント新規作成機能	アカウントを新規作成する機能
パスワード変更機能	ログイン済みのユーザが、自分のアカウントのパスワードを変更する機能
アカウントロックアウト機能	短期間に一定回数以上認証に失敗したアカウントを認証不能な状態にする機能
ロックアウト解除機能	アカウントロックアウト機能により認証不能な状態になったアカウントを再び認証可能な状態に戻す機能
パスワード再発行機能	ユーザがパスワードを忘れてしまった場合に、ユーザ確認を行った後、新しいパスワードを設定できる機能

注釈: 本アプリケーションはセキュリティ対策に関するサンプルであるため、本来は当然必要となるパスワード以外の登録情報の更新機能等を作成していない。

認証・認可に関する仕様

本アプリケーションにおける、認証・認可に関する仕様についてそれぞれ以下に示す。

認証

- 認証に使用するための初期パスワードはアプリケーション側から払い出されるものとする

認可

- ログイン画面、アカウント作成に使用する画面、パスワード再発行に使用する画面以外の画面へのアクセスには、認証が必要
- 「一般ユーザ」と「管理ユーザ」の二種類のロールが存在する
  - 一つのアカウントが複数のロールを持つことができる
- アカウントロックアウト解除機能は、管理ユーザの権限を持つアカウントのみが使用できる

パスワード再発行時の認証

- パスワード再発行の認証にはアプリケーションが生成する次の二つの情報を用いる
  - パスワード再発行画面の URL
  - 認証用の秘密情報
- アプリケーションが生成するパスワード再発行画面の URL は以下の形式である
  - {baseUrl}/reissue/resetpassword?form&token={token}
    - {baseUrl}：アプリケーションのベース URL

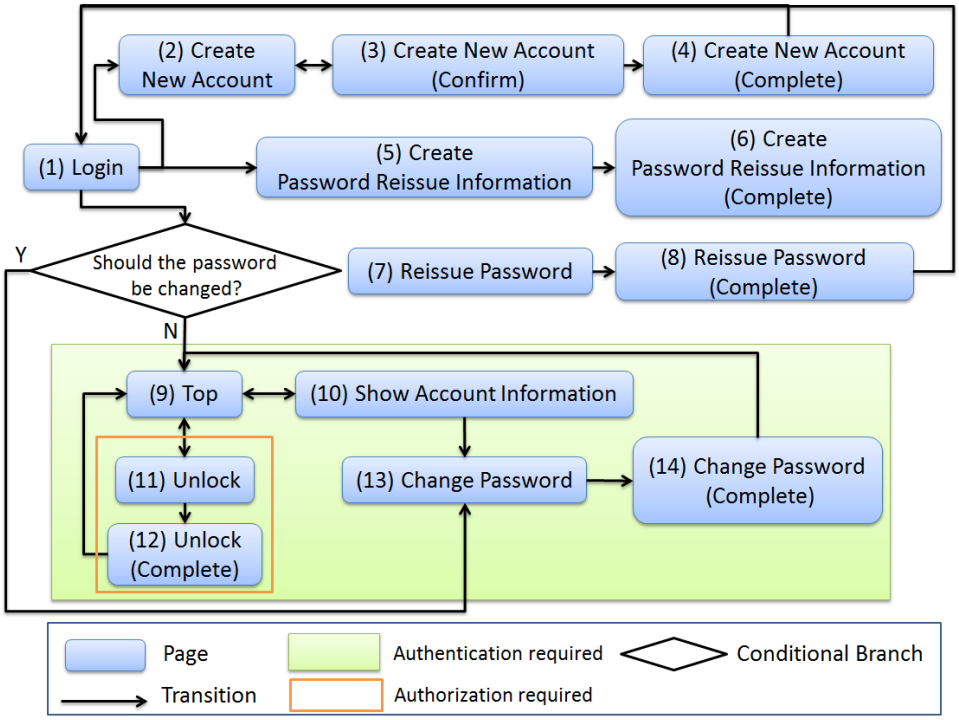
＊ {token} : UUID version4 形式の文字列（ハイフン込みで 36 文字、128bit）

- パスワード再発行画面の URL には 30 分の有効期限を設け、有効期限内のみ認証可能

設計情報

画面遷移

画面遷移図を以下に示す。エラー時の画面遷移は省略している。



項番	画面名	アクセスコントロール
(1)	ログイン画面	-
(2)	アカウント新規作成画面	-
(3)	アカウント新規作成入力確認画面	-
(4)	アカウント新規作成完了画面	-

Continued on next page

表 9.139 – continued from previous page

項番	画面名	アクセスコントロール
(5)	パスワード再発行のための認証情報生成画面	-
(6)	パスワード再発行のための認証情報生成完了画面	-
(7)	パスワード再発行画面	-
(8)	パスワード再発行完了画面	-
(9)	トップ画面	認証済みユーザのみ
(10)	アカウント情報表示画面	認証済みユーザのみ
(11)	ロックアウト解除画面	管理ユーザのみ
(12)	ロックアウト解除完了画面	管理ユーザのみ
(13)	パスワード変更画面	認証済みユーザのみ
(14)	パスワード変更完了画面	認証済みユーザのみ

URL 一覧

URL 一覧を以下に示す。

項番	プロセス名	HTTP メソッド	URL	説明
1	ログイン画面表示	GET	/login	ログイン画面を表示する
2	ログイン	POST	/login	ログイン画面から入力されたユーザー名、パスワードを使って認証する (Spring Security が行う)
3	ログアウト	POST	/logout	ログアウトする (Spring Security が行う)
4	トップ画面表示	GET	/	トップ画面を表示する
5	アカウント情報表示	GET	/accounts	ログインユーザーのアカウント情報を表示する
6	アカウント新規作成画面	GET	/accounts/create?form	アカウント新規作成画面を表示する
7	アカウント新規作成入力確認画面	POST	/accounts/create?confirm	アカウント新規作成入力確認画面を表示する
8	アカウント新規作成	POST	/accounts/create	入力された内容でアカウントを新規に作成する
9	アカウント新規作成完了画面	GET	/accounts/create?complete	アカウント新規作成完了画面を表示する
10	パスワード変更画面表示	GET	/password?form	パスワード変更画面を表示する
11	パスワード変更	POST	/password	パスワード変更画面で入力された情報を使用して、アカウントのパスワードを変更する
12	パスワード変更完了画面表示	GET	/password?complete	パスワード変更完了画面を表示する
13	ロックアウト解除画面表示	GET	/unlock?form	ロックアウト解除画面を表示する
14	ロックアウト解除	POST	/unlock	ロック解除画面に入力された情報を使用してアカウントのロックアウトを解除する
15	ロックアウト解除完了画面表示	GET	/unlock?complete	ロックアウト解除完了画面を表示する
16	パスワード再発行のための認証情報生成画面表示	GET	/reissue/create?form	パスワード再発行のための認証情報生成画面を表示する
17	パスワード再発行のための認証情報生成	POST	/reissue/create	パスワード再発行のための認証情報を生成する
18	パスワード再発行のための認証情報生成完了画面表示	GET	/reissue/create?complete	パスワード再発行のための認証情報生成完了画面を表示する

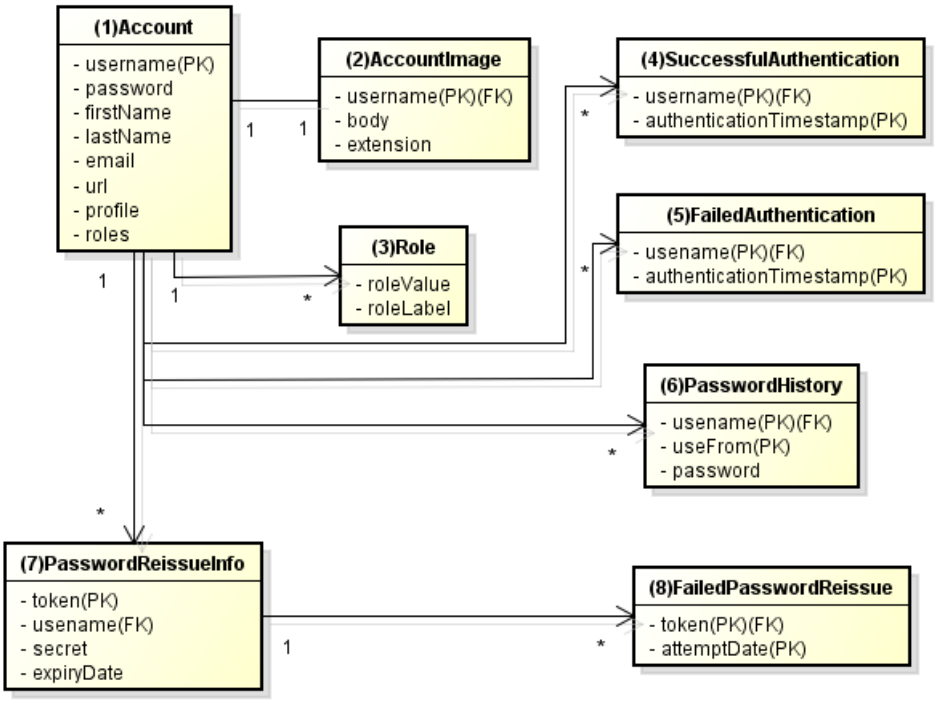
Continued on next page

表 9.140 – continued from previous page

項番	プロセス名	HTTP メソッド	URL	説明
19	パスワード再発行画面表示	GET	/reissue/resetpassword?form&token={token}	トークンパラメータを使用して、ユーザ専用のパスワード再発行画面表示を表示する
20	パスワード再発行	POST	/reissue/resetpassword	パスワード再発行画面に入力された情報を使用してパスワードを再発行する
21	パスワード再発行完了画面表示	GET	/reissue/resetpassword?complete	パスワード再発行完了画面を表示する

ER 図

本アプリケーションにおける ER 図を以下に示す。



項番	エンティティ名	説明	属性
(1)	アカウント	ユーザの登録済みアカウント情報	username : ユーザ名 password : パスワード（ハッシュ化済み） firstName : 名 lastName : 姓 email : E-mail アドレス url : 個人の Web サイトやブログの URL profile : プロフィール roles : ロール (複数可)

Continued on next page

表 9.141 – continued from previous page

項番	エンティティ名	説明	属性
(2)	アカウント画像	アカウントに対してユーザが登録する画像	username : アカウント画像に対応するアカウントのユーザ名 body : 画像ファイルのバイナリ extension : 画像ファイルの拡張子
(3)	ロール	認可に使用する権限	roleValue : ロールの識別子 roleLabel : ロールの表示名
(4)	認証成功イベント	アカウントの最終ログイン日時を取得するために、認証成功時に残す情報	username : ユーザ名 authenticationTimestamp : 認証成功日時
(5)	認証失敗イベント	アカウントのロックアウト機能で用いるために、認証失敗時に残す情報	username : ユーザ名 authenticationTimestamp : 認証失敗日時
(6)	パスワード変更履歴	パスワードの有効期限の判定等に用いるために、パスワード変更時に残す情報	username : ユーザ名 useFrom : 変更後のパスワードの使用開始日時 password : 変更後のパスワード
(7)	パスワード再発行用の認証情報	パスワード再発行時に、ユーザの確認に用いる情報	token : パスワード再発行画面の URL を一意かつ推測不能にするために用いる文字列 username : ユーザ名 secret : ユーザの確認に用いる文字列 experyDate : パスワード再発行用の認証情報の有効期限

Continued on next page

表 9.141 – continued from previous page

項番	エンティティ名	説明	属性
(8)	パスワード再発行失敗イベント	パスワード再発行用の試行回数を制限するために、パスワード再発行失敗に残す情報	token : パスワード再発行に失敗した際に使用した token attemptDate : パスワード再発行を試行した日時

ちなみに: 初期パスワードやパスワード有効期限切れの判定を行うために、アカウントエンティティにフィールドを追加してパスワードの最終変更日時等の情報を持たせるといった設計も可能である。そのような方法で実装を行う場合、アカウントのテーブルに様々な状態を判定するためのカラムが追加され、エントリが頻繁に更新されるという状況に繋がりがちである。

本アプリケーションでは、テーブルをシンプルな状態に保ち、エントリの不要な更新を避けて単純に挿入と削除を使用することで要件を実現するために、認証成功イベントエンティティ等のイベントエンティティを用いた設計を採用している。

9.10.3 実装方法とコード解説

セキュリティ要件の分類ごとに、本アプリケーションにおける実装の方法とコードの説明を行う。

ここでは分類ごとで要件の実現のために必要最小限なコード片のみを掲載している。コード全体を確認したい場合は GitHub を参照すること。

本アプリケーションを動作させるための初期データ登録用 SQL は ここ に配置されている。

注釈: 本アプリケーションでは、ボイラープレートコードの排除のために、Lombok を使用している。Lombok については、ボイラープレートコードの排除 *(Lombok)* を参照。

パスワード変更の強制・促進

実装する要件一覧

- 初期パスワード使用時のパスワード変更の強制
- 期限切れ管理ユーザパスワードの変更の強制
- パスワード変更を促すメッセージの表示

動作イメージ

実装方法

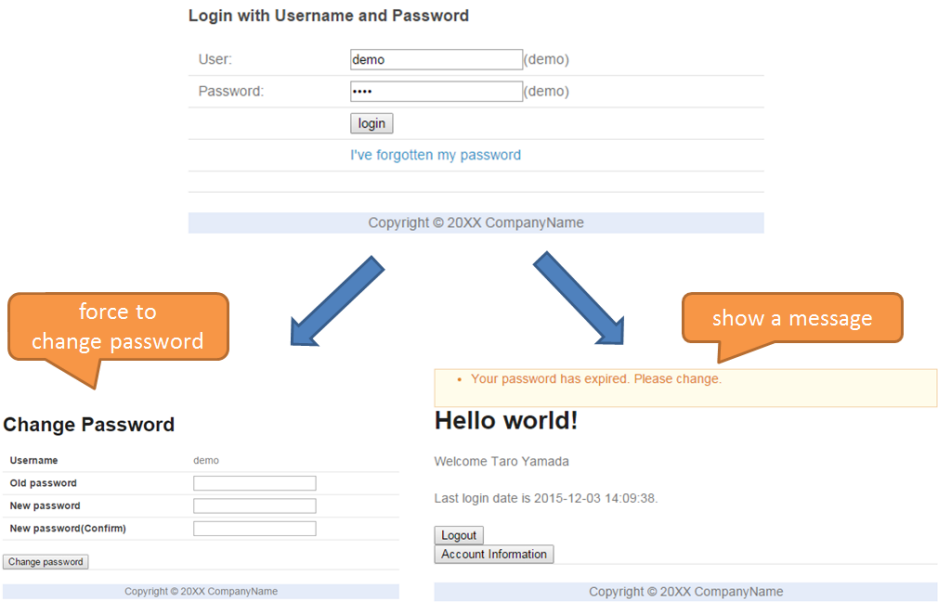
本アプリケーションでは、パスワードを変更した際の履歴を「パスワード変更履歴」エンティティとしてデータベースに保存し、このパスワード変更履歴エンティティを使用して、初期パスワードの判定およびパスワードの有効期限切れの判定を行う。

また、その判定結果に基づいてパスワード変更画面へのリダイレクトや、画面へのメッセージの表示を制御する。

具体的には以下の処理を実装して用いることで、要件を実現する。

- パスワード変更履歴エンティティの保存

  パスワードを変更した際に、以下の情報を持ったパスワード変更履歴エンティティをデータベースに登



録する。

- パスワードを変更したアカウントのユーザ名
- 変更後のパスワードの使用開始日時

- 初期パスワード、パスワード有効期限切れの判定

認証後、認証されたアカウントのパスワード変更履歴エンティティをデータベースから検索し、一件も見つからなければ初期パスワードを使用していると判断する。

そうでない場合には、最新のパスワード変更履歴エンティティを取得し、現在日時とパスワードの使用開始日時の差分を計算して、パスワードの有効期限が切れているかどうかの判定を行う。

- パスワード変更画面への強制リダイレクト

パスワードの変更を強制するために、以下のいずれかに該当する場合には、パスワード変更画面以外へのリクエストが要求された際に、パスワード変更画面へリダイレクトさせる。

- 認証済みのユーザが初回パスワードを使用している場合
- 認証済みのユーザが管理ユーザであり、かつパスワードの有効期限が切れている場合

`org.springframework.web.servlet.handler.HandlerInterceptor` を利用して、Controller のハンドラメソッド実行前に上記の条件に該当するかどうかの判定を行う。

ちなみに: 認証後にパスワード変更画面へリダイレクトさせる方法は他にもあるが、方法によってはリダイレクト後に URL を直打ちすることでパスワード変更を避けて別画面にアクセスできてしまう可能性がある。`HandlerInterceptor` を使用する方法ではハンドラメソッド実行前に処理を行うため、

URL を直打ちするなどの方法で回避することはできない。

ちなみに: `HandlerInterceptor` の代わりに Servlet Filter を用いることもできる。両者の説明については *Controller の呼び出し前後で行う共通処理の実装* を参照すること。ここでは、アプリケーションが許可したリクエストのみに対して処理を行うために、`HandlerInterceptor` を用いている。

- パスワード変更を促すメッセージの表示

  Controller の中で前述のパスワード有効期限切れ判定処理を呼び出す。判定結果を View に渡し、View でメッセージの表示・非表示を切り替える。

コード解説

上記の実装方法に従って実装されたコードについて順に解説する。

- パスワード変更履歴エンティティの保存

  パスワード変更時にパスワード変更履歴エンティティをデータベースに登録するための一連の実装を示す。

  - Entity の実装

    パスワード変更履歴エンティティの実装は以下の通り。

```
package com.example.securelogin.domain.model;

// omitted

@Data
public class PasswordHistory {

    private String username; // (1)

    private String password; // (2)

    private DateTime useFrom; // (3)

}
```

項番	説明
(1)	パスワードを変更したアカウントのユーザ名
(2)	変更後のパスワード
(3)	変更後のパスワードの使用開始日時

- Repository の実装

データベースに対するパスワード変更履歴エンティティの登録、検索を行うための Repository を以下に示す。

```
package com.example.securelogin.domain.repository.passwordhistory;

// omitted

public interface PasswordHistoryRepository {

    int create(PasswordHistory history); // (1)

    List<PasswordHistory> findByUseFrom(@Param("username") String username,
            @Param("useFrom") LocalDateTime useFrom); // (2)

    List<PasswordHistory> findLatest(@Param("username") String username,
            @Param("limit") int limit); // (3)

}
```

項番	説明
(1)	引数として与えられた PasswordHistory オブジェクトをデータベースのレコードとして登録するメソッド
(2)	引数として与えられたユーザ名をキーとして、パスワードの使用開始日時が指定された日付よりも新しい PasswordHistory オブジェクトを降順 (新しい順) に取得するメソッド
(3)	引数として与えられたユーザ名をキーとして、指定された個数の PasswordHistory オブジェクトを新しい順に取得するメソッド

マッピングファイルは以下の通り。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper
    namespace="com.example.securelogin.domain.repository.passwordhistory.
↪PasswordHistoryRepository">

    <resultMap id="PasswordHistoryResultMap" type="PasswordHistory">
        <id property="username" column="username" />
        <id property="password" column="password" />
        <id property="useFrom" column="use_from" />
    </resultMap>

    <select id="findByUseFrom" resultMap="PasswordHistoryResultMap">
    <![CDATA[
        SELECT
            username,
            password,
            use_from
        FROM
            password_history
        WHERE
            username = #{username} AND
            use_from >= #{useFrom}
        ORDER BY use_from DESC
    ]]>
    </select>

    <select id="findLatest" resultMap="PasswordHistoryResultMap">
```

```
    <![CDATA[
        SELECT
            username,
            password,
            use_from
        FROM
            password_history
        WHERE
            username = #{username}
        ORDER BY use_from DESC
        LIMIT #{limit}
    ]]>
    </select>

    <insert id="create" parameterType="PasswordHistory">
    <![CDATA[
        INSERT INTO password_history (
            username,
            password,
            use_from
        ) VALUES (
            #{username},
            #{password},
            #{useFrom}
        )
    ]]>
    </insert>
</mapper>
```

- Service の実装

パスワード変更履歴エンティティの操作は パスワードの品質チェック においても使用する。そのため、以下のように SharedService から Repository のメソッドを呼び出す。

```
package com.example.securelogin.domain.service.passwordhistory;

// omitted

@Service
@Transactional
public class PasswordHistorySharedServiceImpl implements
        PasswordHistorySharedService {

    @Inject
    PasswordHistoryRepository passwordHistoryRepository;

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public int insert(PasswordHistory history) {
        return passwordHistoryRepository.create(history);
    }

    @Transactional(readOnly = true)
```

```
    public List<PasswordHistory> findHistoriesByUseFrom(String username,
            LocalDateTime useFrom) {
        return passwordHistoryRepository.findByUseFrom(username, useFrom);
    }

    @Override
    @Transactional(readOnly = true)
    public List<PasswordHistory> findLatest(String username, int limit) {
        return passwordHistoryRepository.findLatest(username, limit);
    }

}
```

パスワード変更時にパスワード変更履歴エンティティをデータベースに保存する処理の実装を以下に示す。

```
package com.example.securelogin.domain.service.account;

// omitted

@Service
@Transactional
public class AccountSharedServiceImpl implements AccountSharedService {

    @Inject
    ClassicDateFactory dateFactory;

    @Inject
    PasswordHistorySharedService passwordHistorySharedService;

    @Inject
    AccountRepository accountRepository;

    @Inject
    PasswordEncoder passwordEncoder;

    // omitted

    public boolean updatePassword(String username, String rawPassword) { // (1)
        String password = passwordEncoder.encode(rawPassword);
        boolean result = accountRepository.updatePassword(username,␣
↪password); // (2)

        LocalDateTime passwordChangeDate = dateFactory.newTimestamp().
↪toLocalDateTime();

        PasswordHistory passwordHistory = new PasswordHistory(); // (3)
        passwordHistory.setUsername(username);
        passwordHistory.setPassword(password);
        passwordHistory.setUseFrom(passwordChangeDate);
        passwordHistorySharedService.insert(passwordHistory); // (4)
```

```
        return result;
    }

    // omitted
}
```

項番	説明
(1)	パスワードを変更する際に呼び出されるメソッド
(2)	データベース上のパスワードを更新する処理を呼び出す。
(3)	パスワード変更履歴エンティティを作成し、ユーザ名、変更後のパスワード、変更後のパスワードの使用開始日時を設定する。
(4)	作成したパスワード変更履歴エンティティをデータベースに登録する処理を呼び出す。

- 初期パスワード、パスワード有効期限切れの判定

  データベースに登録されたパスワード変更履歴エンティティを用いて、初期パスワードを使用しているかどうかの判定と、パスワードの有効期限が切れているかどうかを判定する処理の実装を以下に示す。

```
package com.example.securelogin.domain.service.account;

// omitted

@Service
@Transactional
public class AccountSharedServiceImpl implements AccountSharedService {

    @Inject
    ClassicDateFactory dateFactory;

    @Inject
    PasswordHistorySharedService passwordHistorySharedService;

    @Value("${security.passwordLifeTimeSeconds}") // (1)
    int passwordLifeTimeSeconds;

    // omitted
```

```
    @Transactional(readOnly = true)
    @Override
    @Cacheable("isInitialPassword")
    public boolean isInitialPassword(String username) { // (2)
        List<PasswordHistory> passwordHistories = passwordHistorySharedService
                .findLatest(username, 1); // (3)
        return passwordHistories.isEmpty(); // (4)
    }

    @Transactional(readOnly = true)
    @Override
    @Cacheable("isCurrentPasswordExpired")
    public boolean isCurrentPasswordExpired(String username) { // (5)
        List<PasswordHistory> passwordHistories = passwordHistorySharedService
                .findLatest(username, 1); // (6)

        if (passwordHistories.isEmpty()) { // (7)
            return true;
        }

        if (passwordHistories
                .get(0)
                .getUseFrom()
                .isBefore(
                        dateFactory.newTimestamp().toLocalDateTime()
                                .minusSeconds(passwordLifeTimeSeconds))) { //
↪(8)
            return true;
        }

        return false;
    }

}
```

項番	説明
(1)	プロパティファイルからパスワードが有効である期間の長さ（秒単位）を取得し、設定する。
(2)	初期パスワードを使用しているかどうかを判定し、使用している場合は true、そうでなければ false を返すメソッド
(3)	データベースから最新のパスワード変更履歴エンティティを一件取得する処理を呼び出す。
(4)	データベースからパスワード変更履歴エンティティが取得できなかった場合に、初期パスワードを使用していると判定し、true を返す。そうでなければ false を返す。
(5)	現在使用中のパスワードの有効期限が切れているかどうかを判定し、切れている場合は true、そうでなければ false を返すメソッド
(6)	データベースから最新のパスワード変更履歴エンティティを一件取得する処理を呼び出す。
(7)	データベースからパスワード変更履歴エンティティが取得できなかった場合には、パスワードの有効期限が切れていると判定し、true を返す。
(8)	パスワード変更履歴エンティティから取得したパスワードの使用開始日時と現在日時の差分が、(1) で設定したパスワード有効期間よりも大きい場合、パスワードの有効期限が切れていると判定し、true を返す。
(9)	(7), (8) のいずれの条件にも該当しない場合、パスワード有効期限内であると判定し、false を返す。

ちなみに： isInitialPassword および isCurrentPasswordExpired に付与されている `@Cacheable` は

Spring の Cache Abstraction 機能を使用するためのアノテーションである。@Cacheable アノテーションを付与することで、メソッドの引数に対する結果をキャッシュすることができる。ここでは、キャッシュの使用により初期パスワード判定、パスワード期限切れ判定のたびにデータベースへのアクセスが発生することを防止し、パフォーマンスの低下を防いでいる。Cache Abstraction については 公式ドキュメント - Cache を参照すること。

尚、キャッシュを使用する際には、必要なタイミングでキャッシュをクリアする必要があることに注意すること。本アプリケーションではパスワード変更時や、ログアウト時には再度初期パスワード判定、パスワード期限切れ判定を行うためにキャッシュをクリアする。

また、必要に応じてキャッシュの TTL(生存時間) を設定すること。TTL は使用するキャッシュの実装によっては設定不能であることに注意。

- パスワード変更画面への強制リダイレクト

パスワードの変更を強制するために、パスワード変更画面へリダイレクトさせる処理の実装を以下に示す。

```
package com.example.securelogin.app.common.interceptor;

// omitted

public class PasswordExpirationCheckInterceptor extends
        HandlerInterceptorAdapter { // (1)

    @Inject
    AccountSharedService accountSharedService;

    @Override
    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler) throws IOException
↪{ // (2)
        Authentication authentication = (Authentication) request
                .getUserPrincipal();

        if (authentication != null) {
            Object principal = authentication.getPrincipal();
            if (principal instanceof UserDetails) { // (3)
                LoggedInUser userDetails = (LoggedInUser) principal; // (4)
                if ((userDetails.getAccount().getRoles().contains(Role.ADMIN)␣
↪&& accountSharedService
                        .isCurrentPasswordExpired(userDetails.getUsername()))␣
↪// (5)
                        || accountSharedService.isInitialPassword(userDetails
                                .getUsername())) { // (6)
                    response.sendRedirect(request.getContextPath()
                            + "/password?form"); // (7)
                    return false; // (8)
                }
            }
        }
```

```
        return true;
    }
}
```

項番	説明
(1)	Controller のハンドラメソッド実行前に処理を挟み込むために、`org.springframework.web.servlet.handler.HandlerInterceptorAdapter` を継承する。
(2)	Controller のハンドラメソッド実行前に実行されるメソッド
(3)	取得したユーザ情報が `org.springframework.security.core.userdetails.UserDetails` のオブジェクトであるかどうかを確認する。
(4)	`UserDetails` のオブジェクトを取得する。本アプリケーションでは、`UserDetails` の実装として `LoggedInUser` というクラスを作成して用いている。
(5)	`UserDetails` オブジェクトからロールを取得してユーザが管理ユーザであるかどうかを判定する。その後、パスワード有効期限が切れているかどうかを判定する処理を呼び出す。二つの判定結果の論理積 (And) をとる。
(6)	初回パスワードを使用しているかどうかを判定する処理を呼び出す。
(7)	(5) または (6) のいずれかが真である場合、`javax.servlet.http.HttpServletResponse` の `sendRedirect` メソッドを使用して、パスワード変更画面へリダイレクトさせる。
(8)	続けて Controller のハンドラメソッドが実行されることを防ぐために、false を返す。

上記のリダイレクト処理を有効にするための設定は以下の通り。

spring-mvc.xml

```
<!-- omitted -->

<mvc:interceptors>

    <!-- omitted -->

    <mvc:interceptor>
        <mvc:mapping path="/**" /> <!-- (1) -->
        <mvc:exclude-mapping path="/password/**" /> <!-- (2) -->
        <mvc:exclude-mapping path="/reissue/**" /> <!-- (3) -->
        <mvc:exclude-mapping path="/resources/**" />
        <bean
            class="com.example.securelogin.app.common.interceptor.
↪PasswordExpirationCheckInterceptor" /> <!-- (4) -->
    </mvc:interceptor>

    <!-- omitted -->

</mvc:interceptors>

<!-- omitted -->
```

項番	説明
(1)	「/」以下のすべてのパスに対するアクセスに `HandlerInterceptor` を適用する。
(2)	パスワード変更画面からパスワード変更画面へのリダイレクトを防ぐため、「/password」以下のパスは適用対象外とする。
(3)	パスワード再発行時にはパスワード有効期限のチェックを行う必要はないため、「/reissue」以下のパスは適用対象外とする。
(4)	`HandlerInterceptor` のクラスを指定する。

- パスワード変更を促すメッセージの表示

  トップ画面にパスワード変更を促すメッセージを表示するための、Controller の実装を以下に示す。

```
package com.example.securelogin.app.welcome;

// omitted

@Controller
public class HomeController {

    @Inject
    AccountSharedService accountSharedService;

    @RequestMapping(value = "/", method = { RequestMethod.GET,
            RequestMethod.POST })
    public String home(@AuthenticationPrincipal LoggedInUser userDetails, //
↪(1)
            Model model) {

        Account account = userDetails.getAccount(); // (2)

        model.addAttribute("account", account);

        if (accountSharedService
                .isCurrentPasswordExpired(account.getUsername())) { // (3)
            ResultMessages messages = ResultMessages.warning().add(
                    "w.sl.pe.0001");
            model.addAttribute(messages);
        }

        // omitted

        return "welcome/home";

    }

}
```

項番	説明
(1)	AuthenticationPrincipal アノテーションを指定して、UserDetails を実装した LoggedInUser のオブジェクトを取得する。
(2)	LoggedInUser が保持しているアカウント情報を取得する。
(3)	アカウント情報から取得したユーザ名を引数として、パスワードの有効期限切れ判定処理を呼び出す。判定結果が true の場合、プロパティファイルからメッセージを取得して Model に設定し、View に渡す。

View の実装は以下の通り。

トップ画面 **(home.jsp)**

```
<!-- omitted -->

<body>
   <div id="wrapper">
       <span id="expiredMessage">
           <t:messagesPanel /> <!-- (1) -->
       </span>

       <!-- omitted -->

   </div>
</body>

<!-- omitted -->
```

項番	説明
(1)	messagesPanel タグを用いて、Controller から渡されたパスワード有効期限切れメッセージを表示する。

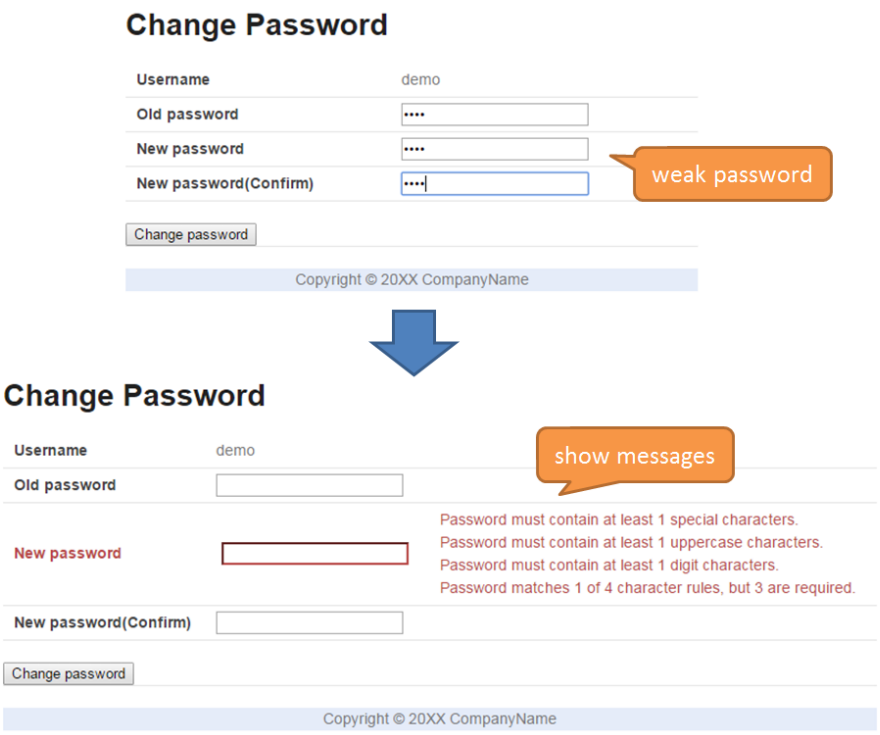
パスワードの品質チェック

実装する要件一覧

- パスワードの最小文字数指定

- パスワードの文字種別指定
- ユーザ名を含むパスワードの禁止
- 管理ユーザパスワードの再使用禁止

動作イメージ



実装方法

パスワード変更時等にユーザが指定したパスワードの品質を検査するためには、 入力チェック の機能を利用することができる。本アプリケーションでは Bean Validation を用いてパスワードの品質を検査する。

パスワードの品質として求められる要件はアプリケーションによって異なり、多岐にわたる。

そこで、パスワード入力チェック用のライブラリとして Passay を利用し、必要な Bean Validation のアノテーションを作成する。

Passay ではパスワード入力チェックで一般的に使用される機能の多くを提供しており、提供されていない機能についても標準機能を拡張することで容易に実装することができる。

Passay の概要については *Appendix* を参照。

具体的には以下の設定、処理を記述し、使用することで要件を実現する。

- Passay の検証規則の作成

  要件の実現に用いるために、以下の検証規則を作成する。

  - パスワード長の最小値を設定した検証規則
  - パスワードに含めなければならない文字種別を設定した検証規則

  - パスワードがユーザ名を含まないことをチェックするための検証規則
  - 同一のパスワードを過去に使用していないことをチェックするための検証規則

- Passay の検証器の作成

  上記で作成した検証規則を設定した、Passay の検証器を作成する。

- Bean Validation のアノテーションの作成

  Passay の検証器を使用してパスワードの入力チェックを行うためのアノテーションを作成する。一つのアノテーションですべての検証規則を検査することもできるが、多種の規則の検査を行うことで処理が複雑になり視認性が下がることを避けるため、以下の二つに分けて実装する。

  - パスワード自体の性質を検証するアノテーション

    「パスワードが最小文字列長よりも長いこと」、「指定した文字種別の文字を含むこと」、「ユーザ名を含まないこと」の三つの検証規則をチェックする

  - 過去のパスワードとの比較を行うアノテーション

    管理ユーザが、以前使用したパスワードを短期間のうちに再使用していないことをチェックする

  いずれのアノテーションも、ユーザ名と新しいパスワードを用いる相関入力チェックルールとなる。両方のルールに違反した入力を行った場合、それぞれのエラーメッセージが表示される。

- パスワードの入力チェック

  作成した Bean Validation アノテーションを用いて、パスワードの入力チェックを行う。

コード解説

上記の実装方法に従って実装されたコードについて順に解説する。Passay を用いたパスワード入力チェックについては パスワード入力チェック にて説明する。

- Passay の検証規則の作成

  本アプリケーションで使用するほとんどの検証規則は、Passay にデフォルトで用意されたクラスを利用することで定義できる。
  しかしながら、Passay が提供するクラスでは、
  `org.springframework.security.crypto.password.PasswordEncoder` でハッシュ化された過去のパスワードと比較する検証規則を定義することができない。
  そのため、Passay が提供するクラスを拡張し、独自の検証規則のクラスを以下のように作成する必要がある。

```
package com.example.securelogin.app.common.validation.rule;

// omitted
```

```
public class EncodedPasswordHistoryRule extends HistoryRule { // (1)

    private final PasswordEncoder passwordEncoder; // (2)

    public EncodedPasswordHistoryRule(PasswordEncoder passwordEncoder) {
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    protected boolean matches(final String rawPassword,
            final PasswordData.Reference reference) { // (3)
        return passwordEncoder.matches(rawPassword, reference.getPassword());␣
↪// (4)
    }
}
```

項番	説明
(1)	パスワードが過去に使用したパスワードに含まれないかをチェックするための org.passay.HistoryRule を拡張する。
(2)	パスワードのハッシュ化に用いている PasswordEncoder をインジェクションする。
(3)	過去のパスワードとの比較を行うメソッドをオーバーライドする。
(4)	PasswordEncoder の matches メソッドを使用してハッシュ化されたパスワードとの比較を行う。

Passay の検証規則を以下に示す通り Bean 定義する。

applicationContext.xml

```
<bean id="lengthRule" class="org.passay.LengthRule"> <!-- (1) -->
    <property name="minimumLength" value="${security.passwordMinimumLength}" /
↪>
</bean>
<bean id="upperCaseRule" class="org.passay.CharacterRule"> <!-- (2) -->
    <constructor-arg name="data">
        <util:constant static-field="org.passay.EnglishCharacterData.UpperCase
↪" />
    </constructor-arg>
```

```
    <constructor-arg name="num" value="1" />
</bean>
<bean id="lowerCaseRule" class="org.passay.CharacterRule"> <!-- (3) -->
    <constructor-arg name="data">
        <util:constant static-field="org.passay.EnglishCharacterData.LowerCase
↪" />
    </constructor-arg>
    <constructor-arg name="num" value="1" />
</bean>
<bean id="digitRule" class="org.passay.CharacterRule"> <!-- (4) -->
    <constructor-arg name="data">
        <util:constant static-field="org.passay.EnglishCharacterData.Digit" />
    </constructor-arg>
    <constructor-arg name="num" value="1" />
</bean>
<bean id="specialCharacterRule" class="org.passay.CharacterRule"> <!-- (5) -->
    <constructor-arg name="data">
        <util:constant static-field="org.passay.EnglishCharacterData.Special"␣
↪/>
    </constructor-arg>
    <constructor-arg name="num" value="1" />
</bean>
<bean id="characterCharacteristicsRule" class="org.passay.
↪CharacterCharacteristicsRule"> <!-- (6) -->
    <property name="rules">
        <list>
            <ref bean="upperCaseRule" />
            <ref bean="lowerCaseRule" />
            <ref bean="digitRule" />
            <ref bean="specialCharacterRule" />
        </list>
    </property>
    <property name="numberOfCharacteristics" value="3" />
</bean>
<bean id="usernameRule" class="org.passay.UsernameRule" /> <!-- (7) -->
<bean id="encodedPasswordHistoryRule"
    class="com.example.securelogin.app.common.validation.rule.
↪EncodedPasswordHistoryRule"> <!-- (8) -->
    <constructor-arg name="passwordEncoder" ref="passwordEncoder" />
</bean>
```

項番	説明
(1)	パスワードの長さをチェックするための org.passay.LengthRule のプロパティに、プロパティファイルから取得したパスワードの最短長を設定する。
(2)	半角英大文字を一文字以上含むことをチェックする検証規則。パスワードに含まれる文字種別に関するチェックを行うための org.passay.CharacterRule のコンストラクタに、org.passay.EnglishCharacterData.UpperCase と数値の 1 を設定する。
(3)	半角英小文字を一文字以上含むことをチェックする検証規則。パスワードに含まれる文字種別に関するチェックを行うための org.passay.CharacterRule のコンストラクタに、org.passay.EnglishCharacterData.LowerCase と数値の 1 を設定する。
(4)	半角数字を一文字以上含むことをチェックする検証規則。パスワードに含まれる文字種別に関するチェックを行うための org.passay.CharacterRule のコンストラクタに、org.passay.EnglishCharacterData.Digit と数値の 1 を設定する。
(5)	半角記号を一文字以上含むことをチェックする検証規則。パスワードに含まれる文字種別に関するチェックを行うための org.passay.CharacterRule のコンストラクタに、org.passay.EnglishCharacterData.Special と数値の 1 を設定する。
(6)	(2)-(5) の 4 つの検証規則のうち、3 つを満たすことをチェックするための検証規則。org.passay.CharacterCharacteristicsRule のプロパティに、(2)-(5) で定義した Bean のリストと、数値の 3 を設定する。
(7)	パスワードにユーザ名が含まれていないことをチェックするための検証規則
(8)	パスワードが過去に使用したものの中に含まれていないことをチェックするための検証規則

- Passay の検証器の作成

前述した Passay の検証規則を用いて、実際に検証を行う検証器の Bean 定義を以下に示す。

applicationContext.xml

```
<bean id="characteristicPasswordValidator" class="org.passay.PasswordValidator
↪"> <!-- (1) -->
    <constructor-arg name="rules">
        <list>
            <ref bean="lengthRule" />
            <ref bean="characterCharacteristicsRule" />
            <ref bean="usernameRule" />
        </list>
    </constructor-arg>
</bean>
<bean id="encodedPasswordHistoryValidator" class="org.passay.PasswordValidator
↪"> <!-- (2) -->
    <constructor-arg name="rules">
        <list>
            <ref bean="encodedPasswordHistoryRule" />
        </list>
    </constructor-arg>
</bean>
```

項番	説明
(1)	パスワード自体の性質を検証するための検証器。プロパティとして、`LengthRule`, `CharacterCharacteristicsRule`, `UsernameRule` の Bean を設定する。
(2)	過去に使用したパスワードの履歴を使用したチェックを行うための検証器。プロパティとして `EncodedPasswordHistoryRule` の Bean を設定する。

- Bean Validation のアノテーションの作成

要件を実現するために、前述した検証器を使用する 2 つのアノテーションを作成する。

  - パスワード自体の性質を検証するアノテーション

    パスワードが最小文字列長よりも長いこと、指定した文字種別の文字を含むこと、ユーザ名を含まないことという三つの検証規則をチェックするアノテーションの実装を以下に示す。

```
package com.example.securelogin.app.common.validation;

// omitted

@Documented
@Constraint(validatedBy = { StrongPasswordValidator.class }) // (1)
@Target({ TYPE, ANNOTATION_TYPE })
```

```
@Retention(RUNTIME)
public @interface StrongPassword {
    String message() default "{com.example.securelogin.app.common.
↪validation.StrongPassword.message}";

    Class<?>[] groups() default {};

    String usernamePropertyName(); // (2)

    String newPasswordPropertyName(); // (3)

    @Target({ TYPE, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        StrongPassword[] value();
    }

    Class<? extends Payload>[] payload() default {};
}
```

項番	説明
(1)	アノテーション付与時に使用する ConstraintValidator を指定する。
(2)	ユーザ名のプロパティ名を指定するためのプロパティ。
(3)	パスワードのプロパティ名を指定するためのプロパティ。

```
package com.example.securelogin.app.common.validation;

// omitted

public class StrongPasswordValidator implements
        ConstraintValidator<StrongPassword, Object> {

    @Inject
    @Named("characteristicPasswordValidator") // (1)
    PasswordValidator characteristicPasswordValidator;

    private String usernamePropertyName;

    private String newPasswordPropertyName;
```

```
    @Override
    public void initialize(StrongPassword constraintAnnotation) {
        usernamePropertyName = constraintAnnotation.usernamePropertyName();
        newPasswordPropertyName = constraintAnnotation.
↪newPasswordPropertyName();
    }

    @Override
    public boolean isValid(Object value, ConstraintValidatorContext␣
↪context) {
        BeanWrapper beanWrapper = new BeanWrapperImpl(value);
        String username = (String) beanWrapper.
↪getPropertyValue(usernamePropertyName);
        String newPassword = (String) beanWrapper
                .getPropertyValue(newPasswordPropertyName);

        RuleResult result = characteristicPasswordValidator
                .validate(PasswordData.newInstance(newPassword, username,␣
↪null)); // (2)

        if (result.isValid()) { // (3)
            return true;
        } else {
            context.disableDefaultConstraintViolation();
            for (String message : characteristicPasswordValidator
                    .getMessages(result)) { // (4)
                context.buildConstraintViolationWithTemplate(message)
                        .addPropertyNode(newPasswordPropertyName)
                        .addConstraintViolation();
            }
            return false;
        }
    }
}
```

項番	説明
(1)	Passay の検証器をインジェクションする。
(2)	パスワードとユーザ名を指定した `org.passay.PasswordData` のインスタンスを作成し、検証器で入力チェックを行う。
(3)	チェックの結果を確認し、OK ならば true を返し、そうでなければ false を返す。
(4)	パスワード入力チェックエラーメッセージをすべて取得し、設定する。

- 過去のパスワードとの比較を行うアノテーション

管理ユーザが、以前使用したパスワードを短期間のうちに再使用していないことをチェックするアノテーションの実装を以下に示す。

過去に使用したパスワードを取得するために、パスワード変更履歴エンティティを用いる。パスワード変更履歴エンティティについては パスワード変更の強制・促進 を参照。

注釈: 「いくつ前までのパスワードの再使用を禁止するか」のみの設定では、短時間の間にパスワード変更を繰り返すことでパスワードを再使用することが可能となってしまう。これを防ぐために、本アプリケーションでは「いつ以降使用したパスワードの再使用を禁止するか」を設定して検査を行う。

```
package com.example.securelogin.app.common.validation;

@Documented
@Constraint(validatedBy = { NotReusedPasswordValidator.class }) // (1)
@Target({ TYPE, ANNOTATION_TYPE })
@Retention(RUNTIME)
public @interface NotReusedPassword {
    String message() default "{com.example.securelogin.app.common.
↪validation.NotReusedPassword.message}";

    Class<?>[] groups() default {};
```

```
    String usernamePropertyName(); // (2)

    String newPasswordPropertyName(); // (3)

    @Target({ TYPE, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        NotReusedPassword[] value();
    }

    Class<? extends Payload>[] payload() default {};
}
```

項番	説明
(1)	アノテーション付与時に使用する ConstraintValidator を指定する。
(2)	ユーザ名のプロパティ名を指定するためのプロパティ。データベースから過去に使用したパスワードを検索するために必要となる。
(3)	パスワードのプロパティ名を指定するためのプロパティ。

```
package com.example.securelogin.app.common.validation;

// omitted

public class NotReusedPasswordValidator implements
        ConstraintValidator<NotReusedPassword, Object> {

    @Inject
    ClassicDateFactory dateFactory;

    @Inject
    AccountSharedService accountSharedService;

    @Inject
    PasswordHistorySharedService passwordHistorySharedService;

    @Inject
    PasswordEncoder passwordEncoder;

    @Inject
```

```
    @Named("encodedPasswordHistoryValidator") // (1)
    PasswordValidator encodedPasswordHistoryValidator;

    @Value("${security.passwordHistoricalCheckingCount}") // (2)
    int passwordHistoricalCheckingCount;

    @Value("${security.passwordHistoricalCheckingPeriod}") // (3)
    int passwordHistoricalCheckingPeriod;

    private String usernamePropertyName;

    private String newPasswordPropertyName;

    private String message;

    @Override
    public void initialize(NotReusedPassword constraintAnnotation) {
        usernamePropertyName = constraintAnnotation.usernamePropertyName();
        newPasswordPropertyName = constraintAnnotation.
↪newPasswordPropertyName();
        message = constraintAnnotation.message();
    }

    @Override
    public boolean isValid(Object value, ConstraintValidatorContext␣
↪context) {
        BeanWrapper beanWrapper = new BeanWrapperImpl(value);
        String username = (String) beanWrapper.
↪getPropertyValue(usernamePropertyName);
        String newPassword = (String) beanWrapper
                .getPropertyValue(newPasswordPropertyName);

        Account account = accountSharedService.findOne(username);
        String currentPassword = account.getPassword();

        boolean result = checkNewPasswordDifferentFromCurrentPassword(
                newPassword, currentPassword, context); // (4)
        if (result && account.getRoles().contains(Role.ADMIN)) { // (5)
            result = checkHistoricalPassword(username, newPassword,␣
↪context);
        }

        return result;
    }

    private boolean checkNewPasswordDifferentFromCurrentPassword(
            String newPassword, String currentPassword,
            ConstraintValidatorContext context) {
        if (!passwordEncoder.matches(newPassword, currentPassword)) {
            return true;
        } else {
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate(message)
```

```
                    .addPropertyNode(newPasswordPropertyName).
↪addConstraintViolation();
            return false;
        }
    }

    private boolean checkHistoricalPassword(String username,
            String newPassword, ConstraintValidatorContext context) {
        LocalDateTime useFrom = dateFactory.newTimestamp().
↪toLocalDateTime()
                .minusMinutes(passwordHistoricalCheckingPeriod);
        List<PasswordHistory> historyByTime = passwordHistorySharedService
                .findHistoriesByUseFrom(username, useFrom);
        List<PasswordHistory> historyByCount = passwordHistorySharedService
                .findLatest(username, passwordHistoricalCheckingCount);
        List<PasswordHistory> history = historyByCount.size() >␣
↪historyByTime
                .size() ? historyByCount : historyByTime; // (6)

        List<PasswordData.Reference> historyData = new ArrayList<>();
        for (PasswordHistory h : history) {
            historyData.add(new PasswordData.HistoricalReference(h
                    .getPassword())); // (7)
        }

        PasswordData passwordData = PasswordData.newInstance(newPassword,
                username, historyData); // (8)
        RuleResult result = encodedPasswordHistoryValidator
                .validate(passwordData); // (9)

        if (result.isValid()) { // (10)
            return true;
        } else {
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate(
                    encodedPasswordHistoryValidator.getMessages(result).
↪get(0)) // (11)
                    .addPropertyNode(newPasswordPropertyName).
↪addConstraintViolation();
            return false;
        }
    }
}
```

項番	説明
(1)	Passay の検証器をインジェクションする。

Continued on next page

表 9.142 – continued from previous page

項番	説明
(2)	いくつ前までのパスワードの再使用を禁止するかの閾値をプロパティファイルから取得し、インジェクションする。
(3)	いつ以降使用したパスワードの再使用を禁止するかの閾値（秒数）をプロパティファイルから取得し、インジェクションする。
(4)	新しいパスワードが現在使用しているものと異なるかどうかをチェックする処理を呼び出す。このチェックは一般ユーザ・管理ユーザにかかわらず行う。
(5)	管理ユーザの場合は、新しいパスワードが過去に使用したパスワードに含まれていないかをチェックする処理を呼び出す。
(6)	(2) で指定した個数分のパスワード変更履歴エンティティと、(3) で指定した期間分のパスワード変更履歴エンティティを取得し、どちらか数の多い方を以降のチェックに用いる。
(7)	Passay の検証器で過去のパスワードとの比較を行うために、パスワード変更履歴エンティティからパスワードを取得し、`org.passay.PasswordData.HistoricalReference` のリストを作成する。
(8)	パスワード、ユーザ名、過去のパスワードのリストを指定した `org.passay.PasswordData` のインスタンスを作成する。
(9)	検証器で入力チェックを行う。
(10)	チェック結果を確認し、OK ならば true を返し、そうでなければ false を返す。
(11)	パスワード入力チェックエラーメッセージを取得する。

- パスワードの入力チェック

  Bean Validation アノテーションを使用してアプリケーション層で、パスワード入力チェックを行う。Form クラスに付与されたアノテーションによって Null チェック以外の入力チェックが網羅されていることから、単項目チェックとしては`@NotNull` のみを付与している。

```
package com.example.securelogin.app.passwordchange;

// omitted

import lombok.Data;

@Data
@Compare(left = "newPassword", right = "confirmNewPassword", operator =␣
↪Compare.Operator.EQUAL) // (1)
@StrongPassword(usernamePropertyName = "username", newPasswordPropertyName =
↪"newPassword") // (2)
@NotReusedPassword(usernamePropertyName = "username", newPasswordPropertyName␣
↪= "newPassword") // (3)
@ConfirmOldPassword(usernamePropertyName = "username",␣
↪oldPasswordPropertyName = "oldPassword") // (4)
public class PasswordChangeForm implements Serializable {

    private static final long serialVersionUID = 1L;

    @NotNull
    private String username;

    @NotNull
    private String oldPassword;

    @NotNull
    private String newPassword;

    @NotNull
    private String confirmNewPassword;

}
```

項番	説明
(1)	新しいパスワードの二回の入力が一致しているかをチェックするためのアノテーション。詳細は *terasoluna-gfw-validator のチェックルール* を参照すること。
(2)	上述した、パスワード自体の性質を検証するアノテーション
(3)	過去のパスワードとの比較を行うアノテーション
(4)	入力された現在のパスワードが正しいことをチェックするアノテーション。定義は割愛する。

```
package com.example.securelogin.app.passwordchange;

// omitted

@Controller
@RequestMapping("password")
public class PasswordChangeController {

    @Inject
    PasswordChangeService passwordService;

    // omitted

    @RequestMapping(method = RequestMethod.POST)
    public String change(@AuthenticationPrincipal LoggedInUser userDetails,
            @Validated PasswordChangeForm form, BindingResult bindingResult, /
↪/ (1)
            Model model) {

        Account account = userDetails.getAccount();
        if (bindingResult.hasErrors()
                || !account.getUsername().equals(form.getUsername())) { // (2)
            model.addAttribute(account);
            return "passwordchange/changeForm";
        }

        passwordService.updatePassword(form.getUsername(),
                form.getNewPassword());

        return "redirect:/password?complete";
    }
```

```
    // omitted

}
```

項番	説明
(1)	パスワード変更時に呼び出されるハンドラメソッド。パラメータ中の Form に`@Validated`アノテーションを付与して、入力チェックを行う。
(2)	パスワード変更対象のユーザ名がログイン中のアカウントのユーザ名と一致していることを確認する。両者が異なる場合には、再度パスワード変更画面へ遷移させる。

注釈: 本アプリケーションでは Bean Valiidation でユーザ名を用いたパスワード入力チェックを行うために、ユーザ名を Form から取得している。View では`Model`に設定したユーザ名を hidden で保持することを想定しているが、改ざんされる恐れがあるため、パスワード変更前に Form から取得したユーザ名の確認を行っている。

アカウントのロックアウト

実装する要件一覧

- アカウントロックアウト
- アカウントロックアウト期間の指定
- 管理ユーザによるロックアウトの解除

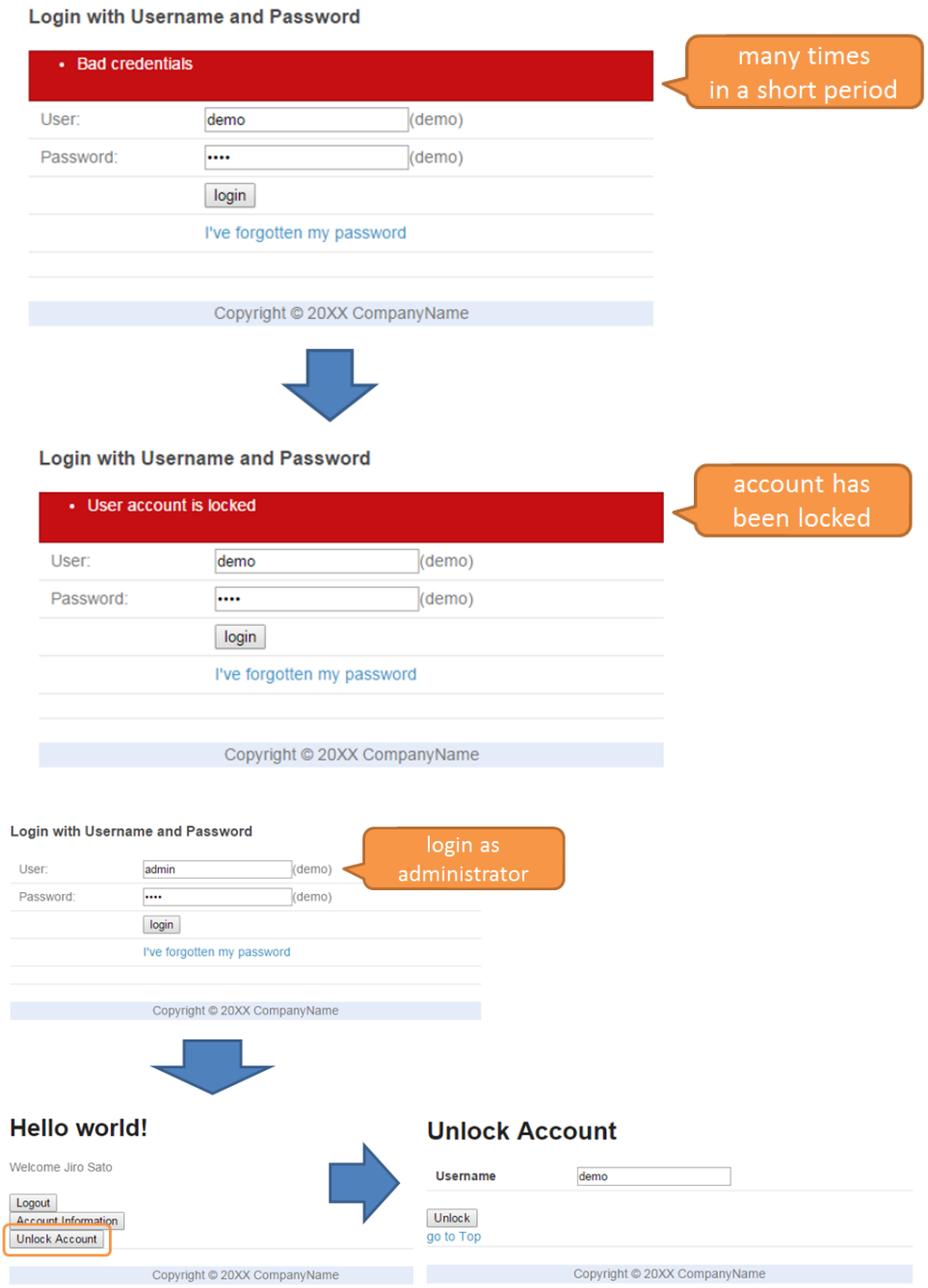
動作イメージ

- アカウントロックアウト

ログインフォームにて、あるユーザ名に対して短時間に一定回数連続して誤ったパスワードで認証を試行すると、そのユーザのアカウントはロックアウト状態となる。ロックアウト状態のアカウントは、正しいユーザ名とパスワードの組を入力した場合であっても認証されない。
ロックアウト状態は一定期間経過するか、ロックアウト解除を行うことで解消される。

- ロックアウト解除

管理権限を持つユーザでログインした場合にのみ、ロックアウト解除機能を使用することができる。ロックア



ウト状態を解消したいユーザ名を入力してロックアウト解除を実行すると、そのユーザのアカウントは再び認証可能な状態に戻る。

実装方法

Spring Security では、`org.springframework.security.core.userdetails.UserDetails` に対してアカウントのロックアウト状態を設定することができる。

「ロックアウト状態である」と設定した場合、Spring Security がその設定を読み取って `org.springframework.security.authentication.LockedException` を throw する。

この機能を用いることにより、アカウントがロックアウト状態であるか否かを判定して `UserDetails` に設定する処理のみを実装すれば、ロックアウト機能が実現できる。

本アプリケーションでは、認証に失敗した履歴を「認証失敗イベント」エンティティとしてデータベースに保存し、この認証失敗イベントエンティティを使用してアカウントのロックアウト状態の判定を行う。
具体的には以下の三つの処理を実装して用いることにより、アカウントのロックアウトに関する各要件を実現する。

- 認証失敗イベントエンティティの保存

  不正な認証情報の入力によって認証に失敗した際に、Spring Security が発生させるイベントをハンドリングし、認証に使用したユーザ名と認証を試みた日時を認証失敗イベントエンティティとしてデータベースに登録する。

- ロックアウト状態の判定

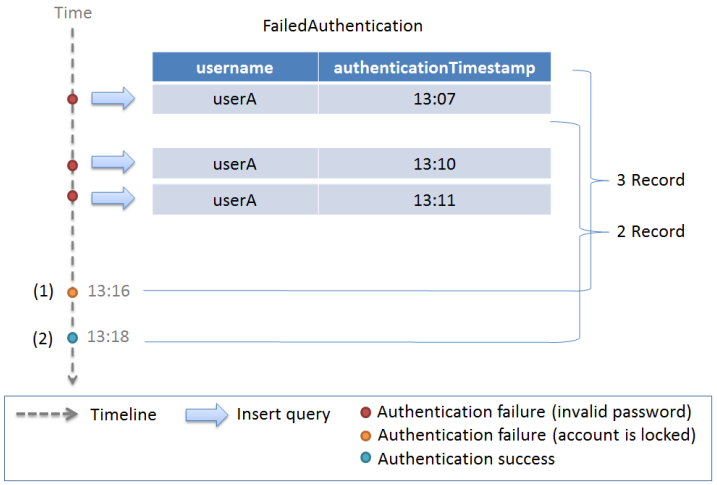
  あるアカウントについて、現在時刻から一定以上新しい認証失敗イベントエンティティが一定個数以上存在する場合、該当アカウントはロックアウト状態であると判定する。認証時にこの判定処理を呼び出し、判定結果を `UserDetails` の実装クラスに設定する。

- 認証失敗イベントエンティティの削除

  あるアカウントについて、認証失敗イベントエンティティをすべて削除する。
  ロックアウトの対象となるのは連続して認証に失敗した場合のみであるため、認証に成功した際には認証失敗イベントエンティティを削除する。
  また、アカウントのロックアウト状態は認証失敗イベントエンティティを用いて判定されるため、認証失敗イベントエンティティを消去することでロックアウト解除機能が実現できる。アカウントのロックアウトは認可機能を用いて、管理ユーザ以外実行できないようにする。

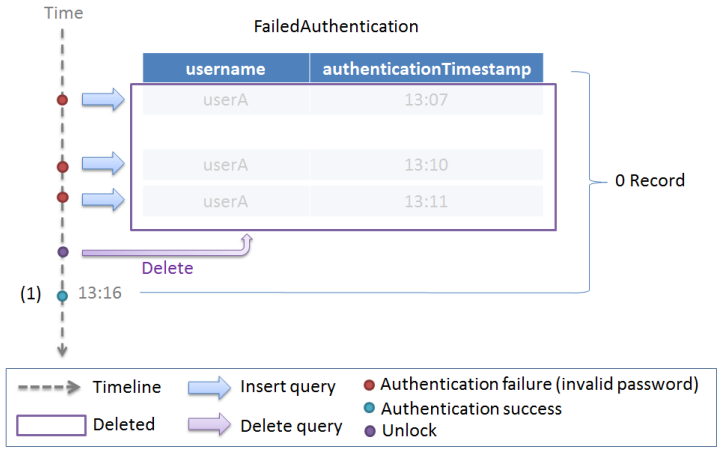
> **警告:** 認証失敗イベントエンティティはロックアウトの判定のみを目的としているため、不要になったタイミングで消去する。認証ログが必要な場合は必ず別途ログを保存しておくこと。

認証失敗イベントエンティティを用いたロックアウト機能の動作例を以下の図を用いて説明する。例として 3 回の認証失敗でロックアウトされるものとし、ロックアウト継続時間は 10 分とする。



項番	説明
(1)	過去 10 分以内に、誤ったパスワードでの認証が 3 回試行されており、データベースには 3 回分の認証失敗イベントエンティティが保存されている。 そのため、アカウントはロックアウト状態であると判定される。
(2)	データベースには 3 回分の認証失敗イベントエンティティが保存されている。 しかしながら、過去 10 分以内の認証失敗イベントエンティティは 2 回分のみであるため、ロックアウト状態ではないと判定される。

同様に、ロックアウトを解除する場合の動作例を以下の図で説明する。



項番	説明
(1)	過去 10 分以内に、誤ったパスワードでの認証が 3 回試行されている。 その後、認証失敗イベントエンティティが消去されているため、データベースには認証失敗イベントエンティティが保存されておらず、ロックアウト状態ではないと判定される。

コード解説

- 共通部分

  本アプリケーションにおいて、アカウントのロックアウトに関する機能を実現するためには、データベースに対する認証失敗イベントエンティティの登録、検索、削除が共通的に必要となる。そのため、まずは認証失敗イベントエンティティに関するドメイン層・インフラストラクチャ層の実装を示す。

  - Entity の実装

    ユーザ名と認証試行日時を持つ認証失敗イベントエンティティの実装を以下に示す。

```
package com.example.securelogin.domain.model;

// omitted

@Data
public class FailedAuthentication implements Serializable {
  private static final long serialVersionUID = 1L;

  private String username; // (1)

  private LocalDateTime authenticationTimestamp; // (2)
}
```

項番	説明
(1)	認証に使用したユーザ名
(2)	認証を試行した日時

  - Repository の実装

    認証失敗イベントエンティティの検索、登録、削除のための Repository を以下に示す。

```
package com.example.securelogin.domain.repository.authenticationevent;

// omitted

public interface FailedAuthenticationRepository {

  int create(FailedAuthentication event); // (1)

  List<FailedAuthentication> findLatest(@Param("username") String username,
          @Param("count") long count); // (2)

  int deleteByUsername(@Param("username") String username); // (3)
}
```

項番	説明
(1)	引数として与えられた FailedAuthentication オブジェクトをデータベースのレコードとして登録するメソッド
(2)	引数として与えられたユーザ名をキーとして、指定された個数の FailedAuthentication オブジェクトを新しい順に取得するメソッド
(3)	引数として与えられたユーザ名をキーとして、認証失敗イベントエンティティのレコードを一括削除するメソッド

マッピングファイルは以下の通り。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper
  namespace="com.example.securelogin.domain.repository.authenticationevent.
↪FailedAuthenticationRepository">

  <resultMap id="failedAuthenticationResultMap"
          type="FailedAuthentication">
          <id property="username" column="username" />
          <id property="authenticationTimestamp" column="authentication_
↪timestamp" />
  </resultMap>

  <insert id="create" parameterType="FailedAuthentication">
```

```
    <![CDATA[
        INSERT INTO failed_authentication (
            username,
            authentication_timestamp
        ) VALUES (
          #{username},
            #{authenticationTimestamp}
        )
    ]]>
  </insert>

  <select id="findLatest" resultMap="failedAuthenticationResultMap">
      <![CDATA[
            SELECT
                username,
                authentication_timestamp
            FROM
                failed_authentication
            WHERE
                username = #{username}
            ORDER BY authentication_timestamp DESC
            LIMIT #{count}
      ]]>
  </select>

  <delete id="deleteByUsername">
     <![CDATA[
          DELETE FROM
              failed_authentication
          WHERE
              username = #{username}
     ]]>
  </delete>
</mapper>
```

- Service の実装

作成した Repository のメソッドを呼び出す Service を以下の通り定義する。

```
package com.example.securelogin.domain.service.authenticationevent;

// omitted

@Service
@Transactional
public class AuthenticationEventSharedServiceImpl implements
        AuthenticationEventSharedService {

    // omitted

    @Inject
    ClassicDateFactory dateFactory;
```

```
    @Inject
    FailedAuthenticationRepository failedAuthenticationRepository;

    @Inject
    AccountSharedService accountSharedService;

    @Transactional(readOnly = true)
    @Override
    public List<FailedAuthentication> findLatestFailureEvents(
                    String username, int count) {
        return failedAuthenticationRepository.findLatest(username, count);
    }


    @Transactional(propagation = Propagation.REQUIRES_NEW)
    @Override
    public void authenticationFailure(String username) { // (1)
        if (accountSharedService.exists(username)) {
            FailedAuthentication failureEvents = new␣
↪FailedAuthentication();
            failureEvents.setUsername(username);
            failureEvents.setAuthenticationTimestamp(dateFactory.
↪newTimestamp()
                    .toLocalDateTime());

            failedAuthenticationRepository.create(failureEvents);
        }
    }

    @Override
    public int deleteFailureEventByUsername(String username) {
        return failedAuthenticationRepository.deleteByUsername(username);
    }

    // omitted

}
```

項番	説明
(1)	認証失敗イベントエンティティを作成してデータベースに登録するメソッド。 引数として受け取ったユーザ名のアカウントが存在しない場合、データベースの外部キー制約に違反するため、データベースへの登録処理をスキップする。 本メソッド実行後の例外により認証失敗イベントエンティティが登録されない可能性を考慮し、トランザクションの伝搬方法に REQUIRES_NEW を指定している。

以下、実装方法に従って実装されたコードについて順に解説する。

- 認証失敗イベントエンティティの保存

  認証失敗時に発生するイベントをハンドリングして処理を行うために、`@EventListener` アノテーションを使用する。`@EventListener` アノテーションによるイベントのハンドリングについては 認証イベントのハンドリング を参照すること。

```
package com.example.securelogin.domain.service.account;

// omitted

@Component
public class AccountAuthenticationFailureBadCredentialsEventListener{

    @Inject
    AuthenticationEventSharedService authenticationEventSharedService;

    @EventListener // (1)
    public void onApplicationEvent(
                    AuthenticationFailureBadCredentialsEvent event) {

        String username = (String) event.getAuthentication().getPrincipal(); //
↪(2)

        authenticationEventSharedService.authenticationFailure(username); //␣
↪(3)
    }

}
```

項番	説明
(1)	`@EventListener` アノテーションを付与することで、誤ったパスワード等の不正な認証情報によって認証が失敗した際に、`onApplicationEvent` メソッドが実行される。
(2)	`AuthenticationFailureBadCredentialsEvent` オブジェクトから、認証に使用したユーザ名を取得する。
(3)	認証失敗イベントエンティティを作成してデータベースに登録する処理を呼び出す。

- ロックアウト状態の判定

  認証失敗イベントエンティティを用いてアカウントのロックアウト状態を判定する処理を記述する。

```
package com.example.securelogin.domain.service.account;

// omitted

@Service
@Transactional
public class AccountSharedServiceImpl implements AccountSharedService {

    // omitted

    @Inject
    ClassicDateFactory dateFactory;

    @Inject
    AuthenticationEventSharedService authenticationEventSharedService;

    @Value("${security.lockingDurationSeconds}") // (1)
    int lockingDurationSeconds;

    @Value("${security.lockingThreshold}") // (2)
    int lockingThreshold;

    @Transactional(readOnly = true)
    @Override
    public boolean isLocked(String username) {
        List<FailedAuthentication> failureEvents =␣
↪authenticationEventSharedService
                .findLatestFailureEvents(username, lockingThreshold); // (3)

        if (failureEvents.size() < lockingThreshold) { // (4)
            return false;
        }

        if (failureEvents
                .get(lockingThreshold - 1) // (5)
                .getAuthenticationTimestamp()
                .isBefore(
                        dateFactory.newTimestamp().toLocalDateTime()
                                .minusSeconds(lockingDurationSeconds))) {
            return false;
        }

        return true;
    }

    // omitted
}
```

項番	説明
(1)	ロックアウトの継続時間を秒単位で指定する。プロパティファイルに定義された値をインジェクションしている。
(2)	ロックアウトの閾値を指定する。ここで指定した回数だけ認証に失敗すると、アカウントがロックアウトされる。プロパティファイルに定義された値をインジェクションしている。
(3)	認証失敗イベントエンティティを、ロックアウトの閾値と同じ数だけ新しい順に取得する。
(4)	取得した認証失敗イベントエンティティの個数がロックアウトの閾値より小さい場合、ロックアウト状態ではないと判定する。
(5)	取得した認証失敗イベントエンティティのうち最も古い認証失敗時刻と現在時刻の差分が、ロックアウト継続時間よりも大きい場合には、ロックアウト状態ではないと判定する。

`UserDetails` の実装クラスである `org.springframework.security.core.userdetails.User` では、コンストラクタにロックアウト状態を渡すことができる。

本アプリケーションでは以下のように `User` を継承したクラスと、`org.springframework.security.core.userdetails.UserDetailsService` を実装したクラスを用いる。

```
package com.example.securelogin.domain.service.userdetails;

// omitted

public class LoggedInUser extends User {

    // omitted

    private final Account account;

    public LoggedInUser(Account account, boolean isLocked,
```

```
            LocalDateTime lastLoginDate,
            List<SimpleGrantedAuthority> authorities) {
        super(account.getUsername(), account.getPassword(), true, true, true,
                !isLocked, authorities); // (1)
        this.account = account;

        // omitted
    }

     public Account getAccount() {
         return account;
     }

    // omitted
}
```

項番	説明
(1)	親クラスである User のコンストラクタに **ロックアウト状態でないかどうか** を真理値で渡す。ロックアウト状態でない場合に true を渡す必要があることに注意する。

```
package com.example.securelogin.domain.service.userdetails;

// omitted

@Service
public class LoggedInUserDetailsService implements UserDetailsService {

    @Inject
    AccountSharedService accountSharedService;

    @Transactional(readOnly = true)
    @Override
    public UserDetails loadUserByUsername(String username)
             throws UsernameNotFoundException {
        try {
            Account account = accountSharedService.findOne(username);
            List<SimpleGrantedAuthority> authorities = new ArrayList<>();
            for (Role role : account.getRoles()) {
                authorities.add(new SimpleGrantedAuthority("ROLE_"
                        + role.getRoleValue()));
            }
            return new LoggedInUser(account,
                    accountSharedService.isLocked(username), // (1)
                    accountSharedService.getLastLoginDate(username),
                    authorities);
        } catch (ResourceNotFoundException e) {
            throw new UsernameNotFoundException("user not found", e);
```

```
        }
    }

}
```

項番	説明
(1)	LoggedInUser のコンストラクタに、isLocked メソッドによるロックアウト状態の判定結果を渡す。

作成した UserDetailsService を使用するための設定は以下の通り。

spring-security.xml

```
<!-- omitted -->

<sec:authentication-manager>
    <sec:authentication-provider
        user-service-ref="loggedInUserDetailsService"> <!-- (1) -->
        <sec:password-encoder ref="passwordEncoder" />
    </sec:authentication-provider>
</sec:authentication-manager>

<!-- omitted -->
```

項番	説明
(1)	UserDetailsService の Bean の id を指定する。

- 認証失敗イベントエンティティの削除
  - 認証成功時の認証失敗イベントエンティティの削除

    連続した認証失敗のみをロックアウトの判定に使用するため、認証に成功した際にはアカウントの認証失敗イベントエンティティを削除する。共通部分として作成した Service に、認証成功時に実行するメソッドを作成する。

```
package com.example.securelogin.domain.service.authenticationevent;

// omitted

@Service
@Transactional
public class AuthenticationEventSharedServiceImpl implements
```

```
        AuthenticationEventSharedService {

    // omitted

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    @Override
    public void authenticationSuccess(String username) {

        // omitted

        deleteFailureEventByUsername(username); // (1)
    }

    // omitted

}
```

項番	説明
(1)	引数として渡されたユーザ名のアカウントに関する認証失敗イベントエンティティを削除する。

認証成功時に発生するイベントをハンドリングして処理を行うために、 `@EventListener` アノテーションを使用する。

```
package com.example.securelogin.domain.service.account;

// omitted

@Component
public class AccountAuthenticationSuccessEventListener{

    @Inject
    AuthenticationEventSharedService authenticationEventSharedService;

    @EventListener // (1)
    public void onApplicationEvent(
                    AuthenticationSuccessEvent event) {

        LoggedInUser details = (LoggedInUser) event.getAuthentication()
                .getPrincipal();

        authenticationEventSharedService.authenticationSuccess(details.
↪getUsername()); // (2)

    }

}
```

項番	説明
(1)	`@EventListener` アノテーションを付与することで、認証が成功した際に `onApplicationEvent` メソッドが実行される。
(2)	`AuthenticationSuccessEvent` からユーザ名を取得し、認証失敗イベントエンティティを削除する処理を呼び出す。

- ロックアウト状態の解除

ロックアウト状態の判定に認証失敗イベントエンティティを使用しているため、認証失敗イベントエンティティを削除することでロックアウト状態を解除することができる。ロックアウト解除機能の使用を「管理権限を持つユーザ」に限定するための認可の設定と、ドメイン層・アプリケーション層の実装を行う。

* 認可の設定

ロックアウトの解除を行うことができるユーザの権限を以下の通りに設定する。

spring-security.xml

```
<!-- omitted -->

  <sec:http pattern="/resources/**" security="none" />
  <sec:http>

      <!-- omitted -->

      <sec:intercept-url pattern="/unlock/**" access="hasRole('ADMIN')
↪" /> <!-- (1) -->

      <!-- omitted -->

  </sec:http>

<!-- omitted -->
```

項番	説明
(1)	/unlock 以下の URL へのアクセス権限を管理ユーザに限定する。

* Service の実装

```
package com.example.securelogin.domain.service.unlock;

// omitted

@Transactional
@Service
public class UnlockServiceImpl implements UnlockService {

    @Inject
    AccountSharedService accountSharedService;

    @Inject
    AuthenticationEventSharedService authenticationEventSharedService;

    @Override
    public void unlock(String username) {
        authenticationEventSharedService.
↪deleteFailureEventByUsername(username); // (1)
    }

}
```

項番	説明
(1)	認証失敗イベントエンティティを消去することによりロックアウト状態を解除する。

* Form の実装

```
package com.example.securelogin.app.unlock;

@Data
public class UnlockForm implements Serializable {

    private static final long serialVersionUID = 1L;

    @NotEmpty
    private String username;
}
```

* View の実装

トップ画面 **(home.jsp)**

```
<!-- omitted -->

<body>
    <div id="wrapper">
```

```
        <!-- omitted -->

        <sec:authorize url="/unlock"> <!-- (1) -->
        <div>
            <a id="unlock" href="${f:h(pageContext.request.
↪contextPath)}/unlock?form">
                Unlock Account
            </a>
        </div>
        </sec:authorize>

        <!-- omitted -->

    </div>
</body>

<!-- omitted -->
```

項番	説明
(1)	/unlock 以下のアクセス権限を持つユーザに対してのみ表示する。

ロックアウト解除フォーム **(unlokcForm.jsp)**

```
<!-- omitted -->

<body>
    <div id="wrapper">
        <h1>Unlock Account</h1>
        <t:messagesPanel />
        <form:form action="${f:h(pageContext.request.contextPath)}/
↪unlock"
            method="POST" modelAttribute="unlockForm">
            <table>
                <tr>
                    <th><form:label path="username" cssErrorClass=
↪"error-label">Username</form:label>
                    </th>
                    <td><form:input path="username" cssErrorClass=
↪"error-input" /></td>
                    <td><form:errors path="username" cssClass="error-
↪messages" /></td>
                </tr>
            </table>

            <input id="submit" type="submit" value="Unlock" />
        </form:form>
        <a href="${f:h(pageContext.request.contextPath)}/">go to Top</
↪a>
```

```
    </div>
</body>

<!-- omitted -->
```

ロックアウト解除完了画面 (**unlockComplete.jsp**)

```
<!-- omitted -->

<body>
    <div id="wrapper">
        <h1>${f:h(username)}'s account was successfully unlocked.</h1>
        <a href="${f:h(pageContext.request.contextPath)}/">go to Top</
↪a>
    </div>
</body>

<!-- omitted -->
```

* Controller の実装

```
package com.example.securelogin.app.unlock;

// omitted

@Controller
@RequestMapping("/unlock") // (1)
public class UnlockController {

    @Inject
    UnlockService unlockService;

    @RequestMapping(params = "form")
    public String showForm(UnlockForm form) {
        return "unlock/unlockForm";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String unlock(@Validated UnlockForm form,
            BindingResult bindingResult, Model model,
            RedirectAttributes attributes) {
        if (bindingResult.hasErrors()) {
            return showForm(form);
        }

        try {
            unlockService.unlock(form.getUsername()); // (2)
            attributes.addFlashAttribute("username", form.
↪getUsername());
            return "redirect:/unlock?complete";
        } catch (BusinessException e) {
            model.addAttribute(e.getResultMessages());
```

```
            return showForm(form);
        }
    }

    @RequestMapping(method = RequestMethod.GET, params = "complete")
    public String unlockComplete() {
        return "unlock/unlockComplete";
    }

}
```

項番	説明
(1)	/unlock 以下の URL にマッピングする。認可の設定によって、管理ユーザのみがアクセス可能となる。
(2)	Form から取得したユーザ名を引数として、アカウントのロックアウトを解除する処理を呼び出す。

最終ログイン日時の表示

実装する要件一覧

- 前回ログイン日時の表示

動作イメージ



実装方法

本アプリケーションでは、認証に成功した履歴を「認証成功イベント」エンティティとしてデータベースに保存し、この認証成功イベントエンティティを用いて、トップ画面にアカウントの前回ログイン日時を表示する。具体的には以下の二つの処理を実装することで、要件を実現する。

- 認証成功イベントエンティティの保存

  認証に成功した際に Spring Security が発生させるイベントをハンドリングし、認証に使用したユーザ名と認証に成功した日時を認証成功イベントエンティティとしてデータベースに登録する。

- 前回ログイン日時の取得と表示

  認証時に、アカウントにおける最新の認証成功イベントエンティティをデータベースから取得し、イベントエンティティから認証成功日時を取得して `org.springframework.security.core.userdetails.UserDetails` に設定する。jsp に `UserDetails` が保持している認証成功日時をフォーマットして渡し、表示する。

コード解説

- 共通部分

  本アプリケーションにおいて、前回ログイン日時を表示するためには、データベースに対する認証成功イベントエンティティの登録、検索が必要となる。そのため、まずは認証成功イベントエンティティに関するドメイン層・インフラストラクチャ層の実装から解説を行う。

  - Entity の実装

    ユーザ名と認証成功日時を持つ認証成功イベントエンティティの実装は以下の通り。

```
package com.example.securelogin.domain.model;

// omitted

@Data
public class SuccessfulAuthentication implements Serializable {

    private static final long serialVersionUID = 1L;

    private String username; // (1)

    private LocalDateTime authenticationTimestamp; // (2)

}
```

項番	説明
(1)	認証に使用したユーザ名
(2)	認証を試行した日時

  - Repository の実装

認証成功イベントエンティティの検索、登録を行うための Repository を以下に示す。

```
package com.example.securelogin.domain.repository.authenticationevent;

// omitted

public interface SuccessfulAuthenticationRepository {

    int create(SuccessfulAuthentication event); // (1)

    List<SuccessfulAuthentication> findLatest(
            @Param("username") String username, @Param("count") long count);
↪ // (2)
}
```

項番	説明
(1)	引数として与えられた SuccessfulAuthentication オブジェクトをデータベースのレコードとして登録するメソッド
(2)	引数として与えられたユーザ名をキーとして、指定された個数の SuccessfulAuthentication オブジェクトを新しい順に取得するメソッド

マッピングファイルは以下の通り。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper
    namespace="com.example.securelogin.domain.repository.
↪authenticationevent.SuccessfulAuthenticationRepository">

    <resultMap id="successfulAuthenticationResultMap"
            type="SuccessfulAuthentication">
        <id property="username" column="username" />
        <id property="authenticationTimestamp" column="authentication_
↪timestamp" />
    </resultMap>

    <insert id="create" parameterType="SuccessfulAuthentication">
    <![CDATA[
        INSERT INTO successful_authentication (
            username,
            authentication_timestamp
        ) VALUES (
```

```
            #{username},
            #{authenticationTimestamp}
        )
    ]]>
    </insert>

    <select id="findLatest" resultMap="successfulAuthenticationResultMap">
    <![CDATA[
        SELECT
            username,
            authentication_timestamp
        FROM
            successful_authentication
        WHERE
            username = #{username}
        ORDER BY authentication_timestamp DESC
        LIMIT #{count}
    ]]>
    </select>
</mapper>
```

- Service の実装

作成した Repository のメソッドを呼び出す Service を以下に示す。

```
package com.example.securelogin.domain.service.authenticationevent;

// omitted

@Service
@Transactional
public class AuthenticationEventSharedServiceImpl implements
     AuthenticationEventSharedService {

    // omitted

    @Inject
    ClassicDateFactory dateFactory;

    @Inject
    SuccessfulAuthenticationRepository successAuthenticationRepository;

    @Transactional(readOnly = true)
    @Override
    public List<SuccessfulAuthentication> findLatestSuccessEvents(
            String username, int count) {
        return successAuthenticationRepository.findLatest(username, count);
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    @Override
      public void authenticationSuccess(String username) {
          SuccessfulAuthentication successEvent = new␣
↪SuccessfulAuthentication();
```

```
            successEvent.setUsername(username);
            successEvent.setAuthenticationTimestamp(dateFactory.
↪newTimestamp()
                    .toLocalDateTime());

            successAuthenticationRepository.create(successEvent);
            deleteFailureEventByUsername(username);
        }

}
```

以下、実装方法に従って実装されたコードについて順に解説する。

- 認証成功イベントエンティティの保存

  認証成功時に発生するイベントをハンドリングして処理を行うために、`@EventListener` アノテーションを使用する。

```
package com.example.securelogin.domain.service.account;

// omitted

@Component
public class AccountAuthenticationSuccessEventListener{

    @Inject
    AuthenticationEventSharedService authenticationEventSharedService;

    @EventListener // (1)
    public void onApplicationEvent(AuthenticationSuccessEvent event) {
        LoggedInUser details = (LoggedInUser) event.getAuthentication()
                .getPrincipal(); // (2)

        authenticationEventSharedService.authenticationSuccess(details
                .getUsername()); // (3)
    }

}
```

項番	説明
(1)	`@EventListener` アノテーションを付与することで、認証が成功した際に、`onApplicationEvent` メソッドが実行される。
(2)	`AuthenticationSuccessEvent` オブジェクトから、`UserDetails` の実装クラスを取得する。このクラスについては以降で説明する。
(3)	認証成功イベントエンティティを作成し、データベースに登録する処理を呼び出す。

- 前回ログイン日時の取得と表示

  認証成功イベントエンティティから前回ログイン日時を取得するための Service を以下に示す。

```
package com.example.securelogin.domain.service.account;

// omitted

@Service
@Transactional
public class AccountSharedServiceImpl implements AccountSharedService {

    // omitted

    @Inject
    AuthenticationEventSharedService authenticationEventSharedService;

    @Transactional(readOnly = true)
    @Override
    public LocalDateTime getLastLoginDate(String username) {
        List<SuccessfulAuthentication> events =␣
↪authenticationEventSharedService
                .findLatestSuccessEvents(username, 1); // (1)

        if (events.isEmpty()) {
            return null; // (2)
        } else {
            return events.get(0).getAuthenticationTimestamp(); // (3)
        }
    }

    // omitted

}
```

項番	説明
(1)	引数として与えられたユーザ名をキーとして、最新の認証成功イベントエンティティを一件取得する。
(2)	初回ログイン時等、認証成功イベントエンティティが一件も取得できない場合には null を返す。
(3)	認証成功イベントエンティティから、認証日時を取得して返す。

ログイン時に前回ログイン日時を取得して `UserDetails` に保持させるために、以下のように `User` を継承したクラスと、`UserDetailsService` を実装したクラスを作成する。

```
package com.example.securelogin.domain.service.userdetails;

// omitted

public class LoggedInUser extends User {

    private final Account account;

    private final LocalDateTime lastLoginDate; // (1)

    public LoggedInUser(Account account, boolean isLocked,
            LocalDateTime lastLoginDate,
            List<SimpleGrantedAuthority> authorities) {

        super(account.getUsername(), account.getPassword(), true, true, true,
                !isLocked, authorities);
        this.account = account;
        this.lastLoginDate = lastLoginDate; // (2)
    }

    // omitted

    public LocalDateTime getLastLoginDate() { // (3)
        return lastLoginDate;
    }

}
```

項番	説明
(1)	前回ログイン日時を保持するためのフィールドを宣言する。
(2)	引数として与えられた前回ログイン日時をフィールドに設定する。
(3)	保持している前回ログイン日時を返すメソッド

```
package com.example.securelogin.domain.service.userdetails;

// omitted

@Service
public class LoggedInUserDetailsService implements UserDetailsService {

    @Inject
    AccountSharedService accountSharedService;

    @Transactional(readOnly = true)
    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        try {
            Account account = accountSharedService.findOne(username);
            List<SimpleGrantedAuthority> authorities = new ArrayList<>();
            for (Role role : account.getRoles()) {
                    authorities.add(new SimpleGrantedAuthority("ROLE_"
                            + role.getRoleValue()));
            }
            return new LoggedInUser(account,
                    accountSharedService.isLocked(username),
                    accountSharedService.getLastLoginDate(username), // (1)
                    authorities);
        } catch (ResourceNotFoundException e) {
            throw new UsernameNotFoundException("user not found", e);
        }
    }

}
```

項番	説明
(1)	Service のメソッドを呼び出して前回ログイン日時を取得し、`LoggedInUser` のコンストラクタに渡す。

トップ画面に前回ログイン日時を表示するためのアプリケーション層の実装を行う。

```
package com.example.securelogin.app.welcome;

// omitted

@Controller
public class HomeController {

    @Inject
    AccountSharedService accountSharedService;

    @RequestMapping(value = "/", method = { RequestMethod.GET,
            RequestMethod.POST })
    public String home(@AuthenticationPrincipal LoggedInUser userDetails, //
↪(1)
            Model model) {

        // omitted

        LocalDateTime lastLoginDate = userDetails.getLastLoginDate(); // (2)
        if (lastLoginDate != null) {
            model.addAttribute("lastLoginDate", lastLoginDate
                    .format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss
↪"))); // (3)
        }

        return "welcome/home";

    }

}
```

項番	説明
(1)	`@AuthenticationPrincipal` を使用して UserDetails オブジェクトを取得する。
(2)	`LoggedInUserDetails` から最終ログイン日時を取得する。
(3)	最終ログイン日時をフォーマットして Model に設定し、View に渡す。

トップ画面 **(home.jsp)**

```
<body>
  <div id="wrapper">

      <!-- omitted -->

      <c:if test="${!empty lastLoginDate}"> <!-- (1) -->
          <p id="lastLogin">
              Last login date is ${f:h(lastLoginDate)}. <!-- (2) -->
          </p>
      </c:if>

      <!-- omitted -->

  </div>
</body>
```

項番	説明
(1)	前回ログイン日時が null の場合は表示しない。
(2)	Controller から渡された前回ログイン日時を表示する。

パスワード再発行のための認証情報の生成

実装する要件一覧

- パスワード再発行用 *URL* へのランダム文字列の付与
- パスワード再発行用秘密情報の発行

動作イメージ



パスワード再発行のための認証情報生成画面で、パスワードを再発行するユーザ名を入力する。このとき、パスワード再発行時の認証に使用する秘密情報と、トークンが生成される。秘密情報は画面に表示され、トークンを含んだパスワード再発行画面の URL はユーザの登録済みメールアドレスに送付される。

メール送付された URL には有効期限があり、有効期限内にアクセスして秘密情報と新しいパスワードを入力することで、パスワードを変更することができる。有効期限が切れた後にメール送付された URL にアクセスした場合、エラー画面に遷移する。

ここでは、上記の流れのうち、秘密情報とトークンの生成について説明を行う。

実装方法

パスワードの再発行を行う際には、ユーザがアカウントの所有者であることを確認するためのパスワードに代わる手段が必要である。

本アプリケーションでは、ユーザを確認するための情報として、パスワード再発行画面の URL と秘密情報の二つを用いる。

パスワード再発行画面の URL を一意かつ推測困難にするために、ランダムな文字列を生成し URL に付加する。万が一 URL が漏えいした場合に備え、ランダムな文字列である秘密情報を生成し、これを用いて認証を行う。

二つのランダムな文字列は、片方からもう一方を推測することが不可能となるように、それぞれ異なる方法で生成する。

具体的には以下の処理を実装することで要件を実現する。

- パスワード再発行のための認証情報の生成と保存

以下の 4 つの情報を、パスワード再発行のための認証情報としてデータベースに保存する。

- ユーザ名：パスワードを再発行するアカウントのユーザ名
- トークン：パスワード再発行画面の URL を、一意かつ推測不能にするために生成するランダムな文字列
- 秘密情報：パスワード再発行時にユーザに入力させるために生成するランダムな文字列
- 有効期限：パスワード再発行のための認証情報の有効期限

トークンの生成には `java.util.UUID` クラスの `randomUUID` メソッドを用い、秘密情報の生成には Passay のパスワード生成機能を用いる。

秘密情報については、パスワードと同様にハッシュ化してデータベースへ保存する。有効期限の設定と確認処理については、パスワード再発行実行時の検査 に記す。パスワード再発行のための認証情報をユーザに配布する方法については、パスワード再発行のための認証情報の配布 を参照。

コード解説

- 共通部分

上記の実装方法に従って実装を進める上で、パスワード再発行のための認証情報をデータベースに登録、検索する処理が共通的に必要となる。そのため、まずはパスワード再発行のための認証情報に関連する Entity と Repository の実装から解説する。

- Entity の作成

パスワード再発行のための認証情報の Entity を作成する。

```
package com.example.securelogin.domain.model;

// omitted

@Data
public class PasswordReissueInfo {

    private String username; // (1)

    private String token; // (2)

    private String secret; // (3)

    private LocalDateTime expiryDate; // (4)

}
```

項番	説明
(1)	パスワード再発行対象のユーザ名
(2)	パスワード再発行用 URL に含めるために生成される文字列（トークン）
(3)	パスワード再発行時にユーザを確認するための文字列（秘密情報）
(4)	パスワード再発行のための認証情報の有効期限

- Repository の実装

パスワード再発行のための認証情報の検索、登録、削除を行うための Repository を以下に示す。

```
package com.example.securelogin.domain.repository.passwordreissue;

// omitted

public interface PasswordReissueInfoRepository {

    void create(PasswordReissueInfo info); // (1)

    PasswordReissueInfo findOne(@Param("token") String token); // (2)

    int delete(@Param("token") String token); // (3)

    // omitted

}
```

項番	説明
(1)	引数として与えられた PasswordReissueInfo オブジェクトをデータベースのレコードとして登録するメソッド
(2)	引数として与えられたトークンをキーとして、PasswordReissueInfo オブジェクトを検索し、取得するメソッド
(3)	引数として与えられたトークンをキーとして、PasswordReissueInfo オブジェクトを削除するメソッド

マッピングファイルは以下の通り。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper
    namespace="com.example.securelogin.domain.repository.passwordreissue.
↪PasswordReissueInfoRepository">

    <resultMap id="PasswordReissueInfoResultMap" type="PasswordReissueInfo
↪">
        <id property="username" column="username" />
        <id property="token" column="token" />
        <id property="secret" column="secret" />
        <id property="expiryDate" column="expiry_date" />
    </resultMap>

    <select id="findOne" resultMap="PasswordReissueInfoResultMap">
    <![CDATA[
        SELECT
            username,
            token,
            secret,
            expiry_date
        FROM
            password_reissue_info
        WHERE
            token = #{token}
    ]]>
    </select>

    <insert id="create" parameterType="PasswordReissueInfo">
```

```
    <![CDATA[
        INSERT INTO password_reissue_info (
            username,
            token,
            secret,
            expiry_date
        ) VALUES (
            #{username},
            #{token},
            #{secret},
            #{expiryDate}
        )
    ]]>
    </insert>

    <delete id="delete">
    <![CDATA[
        DELETE FROM
            password_reissue_info
        WHERE
            token = #{token}
    ]]>
    </delete>

    <!-- omitted -->

</mapper>
```

以下、実装方法に従って実装されたコードについて順に解説する。

- パスワード再発行のための認証情報の生成と保存
  - パスワード生成器の定義

    Passay のパスワード生成機能を使用するための、パスワード生成器と生成規則の定義を以下に示す。パスワード生成器や生成規則に関しては パスワード生成 を参照。

    applicationContext.xml

```
<bean id="passwordGenerator" class="org.passay.PasswordGenerator" /> <!--␣
↪(1) -->
<util:list id="passwordGenerationRules"> <!-- (2) -->
    <ref bean="upperCaseRule" />
    <ref bean="lowerCaseRule" />
    <ref bean="digitRule" />
</util:list>
```

項番	説明
(1)	Passay のパスワード生成機能で用いるパスワード生成器の Bean 定義
(2)	Passay のパスワード生成機能で用いるパスワード生成規則の Bean 定義。 パスワードの品質チェック で使用した検証規則を使用し、半角英大文字、半角英小文字、半角数字をそれぞれ一文字以上含むパスワードの生成規則を定義する。

- Service の実装

パスワード再発行のための認証情報を作成し、データベースへ保存するための処理の実装を以下に示す。この処理中で生成した認証情報をメール送信する。メール送信については後述するため、ここでは省略する。

```
package com.example.securelogin.domain.service.passwordreissue;

// omitted

@Service
@Transactional
public class PasswordReissueServiceImpl implements PasswordReissueService {

    @Inject
    ClassicDateFactory dateFactory;

    @Inject
    PasswordReissueInfoRepository passwordReissueInfoRepository;

    @Inject
    AccountSharedService accountSharedService;

    @Inject
    PasswordEncoder passwordEncoder;

    @Inject
    PasswordGenerator passwordGenerator; // (1)

    @Resource(name = "passwordGenerationRules")
    List<CharacterRule> passwordGenerationRules; //(2)

    @Value("${security.tokenLifeTimeSeconds}")
    int tokenLifeTimeSeconds; // (3)

    // omitted

    @Override
```

```
    public String createAndSendReissueInfo(String username) {

        String rowSecret = passwordGenerator.generatePassword(10,␣
↪passwordGenerationRules); // (4)

        if (!accountSharedService.exists(username)) { // (5)
            return rowSecret;
        }

        Account account= accountSharedService.findOne(username); // (6)

        String token = UUID.randomUUID().toString(); // (7)

        LocalDateTime expiryDate = dateFactory.newTimestamp().
↪toLocalDateTime()
                .plusSeconds(tokenLifeTimeSeconds); // (8)

        PasswordReissueInfo info = new PasswordReissueInfo(); // (9)
        info.setUsername(username);
        info.setToken(token);
        info.setSecret(passwordEncoder.encode(rowSecret)); // (10)
        info.setExpiryDate(expiryDate);

        passwordReissueInfoRepository.create(info); // (11)

        // omitted (Send E-Mail)

        return rowSecret; // (12)

    }

    // omitted

}
```

項番	説明
(1)	Passay のパスワード生成機能で用いるパスワード生成器をインジェクションする。
(2)	Passay のパスワード生成機能で用いるパスワード生成ルールをインジェクションする。
(3)	パスワード再発行用の認証情報が有効である期間の長さを秒単位で指定する。プロパティファイルに定義された値をインジェクションしている。

Continued on next page

表 9.143 – continued from previous page

項番	説明
(4)	秘密情報として用いるために、Passay のパスワード生成機能を用いて、パスワード生成規則に従った、長さ 10 のランダムな文字列を生成する。
(5)	引数として渡されてきたユーザ名のアカウントが存在するかどうか確認する。存在しなかった場合、ユーザが存在しないことを知られないためにダミーの秘密情報を返す。
(6)	パスワード再発行用の認証情報に含まれるユーザ名のアカウント情報を取得する。
(7)	トークンとして用いるために、`java.util.UUID` クラスの `randomUUID` メソッドを用いてランダムな文字列を生成する。
(8)	現在時刻に (3) の値を加えることにより、パスワード再発行用の認証情報の有効期限を計算する。
(9)	パスワード再発行用の認証情報を作成し、ユーザ名、トークン、秘密情報、有効期限を設定する。
(10)	秘密情報はハッシュ化を行ってから `PasswordReissueInfo` に設定する。
(11)	パスワード再発行用の認証情報をデータベースに登録する。
(12)	生成した秘密情報を返す。

- Form の実装

```
package com.example.securelogin.app.passwordreissue;

// omitted

@Data
public class CreateReissueInfoForm implements Serializable {

    private static final long serialVersionUID = 1L;

    @NotEmpty
    private String username;
}
```

- View の実装

パスワード再発行のための認証情報生成画面 **(createReissueInfoForm.xml)**

```
<!-- omitted -->

<body>
    <div id="wrapper">
        <h1>Reissue password</h1>
        <t:messagesPanel />
        <form:form
            action="${f:h(pageContext.request.contextPath)}/reissue/create"
            method="POST" modelAttribute="createReissueInfoForm">
            <table>
                <tr>
                    <th><form:label path="username" cssErrorClass="error-
↪label">Username</form:label>
                    </th>
                    <td><form:input path="username" cssErrorClass="error-
↪input" /></td>
                    <td><form:errors path="username" cssClass="error-
↪messages" /></td>
                </tr>
            </table>

            <input id="submit" type="submit" value="Reissue password" />
        </form:form>
    </div>
</body>

<!-- omitted -->
```

- Controller の実装

```
package com.example.securelogin.app.passwordreissue;

// omitted
```

```
@Controller
@RequestMapping("/reissue")
public class PasswordReissueController {

    @Inject
    PasswordReissueService passwordReissueService;

    @RequestMapping(value = "create", params = "form")
    public String showCreateReissueInfoForm(CreateReissueInfoForm form) {
        return "passwordreissue/createReissueInfoForm";
    }

    @RequestMapping(value = "create", method = RequestMethod.POST)
    public String createReissueInfo(@Validated CreateReissueInfoForm form,
            BindingResult bindingResult, Model model,
            RedirectAttributes attributes) {
        if (bindingResult.hasErrors()) {
            return showCreateReissueInfoForm(form);
        }

        String rawSecret = passwordReissueService.
↪createAndSendReissueInfo(form.getUsername()); // (1)
        attributes.addFlashAttribute("secret", rawSecret);
        return "redirect:/reissue/create?complete";
    }

    @RequestMapping(value = "create", params = "complete", method =␣
↪RequestMethod.GET)
    public String createReissueInfoComplete() {
        return "passwordreissue/createReissueInfoComplete";
    }

    // omitted

}
```

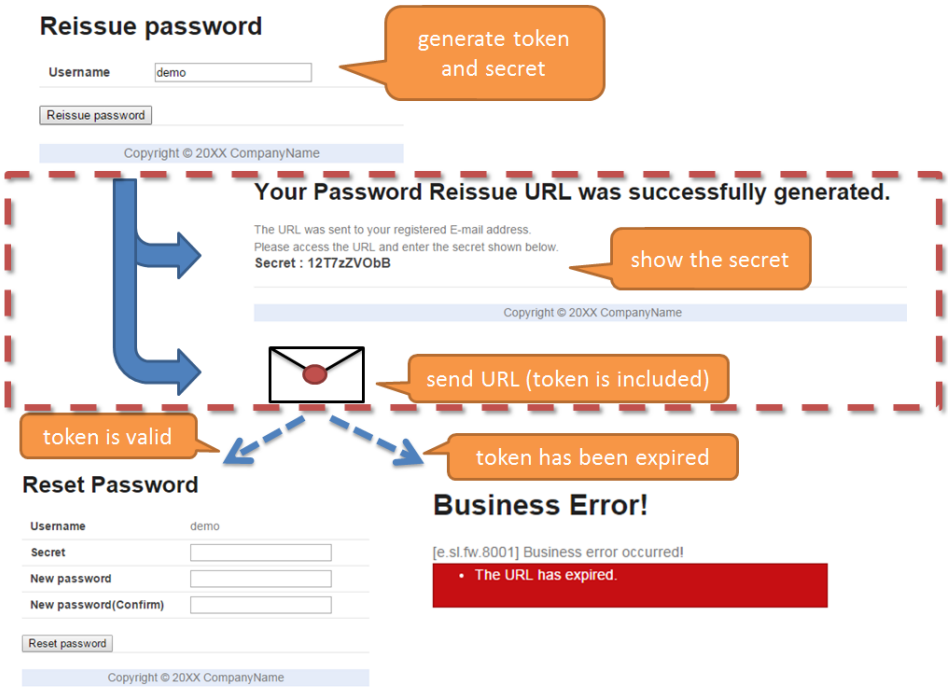
項番	説明
(1)	Form から取得したユーザ名から、パスワード再発行のための認証情報を生成し、データベースに登録する処理を呼び出す。

パスワード再発行のための認証情報の配布

実装する要件一覧

- パスワード再発行画面の *URL* と秘密情報の別配布
- パスワード再発行画面の *URL* のメール送付

動作イメージ



パスワード再発行のための認証情報の生成 では、パスワード再発行のための認証情報の生成について説明した。ここでは、生成した認証情報の配布について説明する。

パスワード再発行のための認証は、パスワード再発行画面の URL と秘密情報を用いて行う。この二つの情報が一度に漏れることを防ぐため、それぞれ別の方法でユーザに配布する。本アプリケーションでは、パスワード再発行画面の URL はユーザの登録済みメールアドレスへ送付し、秘密情報は画面に表示する。

実装方法

パスワード再発行のための認証情報の生成 で生成した認証情報を二つに分け、それぞれ別の方法でユーザに配布する。
以下の二つの処理を実装して用いることで要件を実現する。

- 秘密情報の画面表示

  パスワード再発行のための認証情報の生成 で生成したハッシュ化前の秘密情報を、画面に表示させることでユーザに配布する。

- パスワード再発行画面の URL のメール送付

  パスワード再発行のための認証情報の生成 で生成したトークンを含むパスワード再発行画面の URL を、Spring Framework の Mail 連携用コンポーネントを用いて、メールで送付する。

コード解説

上記の実装方法に従って実装されたコードについて順に解説する。

- 秘密情報の画面表示

Controller から秘密情報の生成処理を呼び出し、View に表示するための一連の実装を以下に示す。

```
package com.example.securelogin.app.passwordreissue;

// omitted

@Controller
@RequestMapping("/reissue")
public class PasswordReissueController {

    @Inject
    PasswordReissueService passwordReissueService;

    // omitted

    @RequestMapping(value = "create", method = RequestMethod.POST)
    public String createReissueInfo(@Validated CreateReissueInfoForm form,
            BindingResult bindingResult, Model model,
            RedirectAttributes attributes) {
        if (bindingResult.hasErrors()) {
            return showCreateReissueInfoForm(form);
        }

        String rawSecret = passwordReissueService.
↪createAndSendReissueInfo(form
                .getUsername()); // (1)
        attributes.addFlashAttribute("secret", rawSecret); // (2)
        return "redirect:/reissue/create?complete"; // (3)
    }

    @RequestMapping(value = "create", params = "complete", method =␣
↪RequestMethod.GET)
    public String createReissueInfoComplete() {
        return "passwordreissue/createReissueInfoComplete";
    }

    // omitted

}
```

項番	説明
(1)	秘密情報を生成する処理を呼び出す。
(2)	RedirectAttributes を利用して、リダイレクト先に秘密情報を渡す。
(3)	パスワード再発行用の認証情報完了画面にリダイレクトする。

パスワード再発行用の認証情報生成完了画面 (**createReissueInfoComplete.jsp**)

```
<!-- omitted -->

<body>
    <div id="wrapper">
        <h1>Your Password Reissue URL was successfully generated.</h1>
        The URL was sent to your registered E-mail address.<br /> Please
        access the URL and enter the secret shown below.
        <h3>Secret : <span id=secret>${f:h(secret)}</span></h3> <!-- (1) -->
    </div>
</body>

<!-- omitted -->
```

項番	説明
(1)	秘密情報を画面に表示する。

- パスワード再発行画面の URL のメール送付

  パスワード再発行用の認証情報からパスワード再発行画面の URL を作成し、メール送付する処理の実装を以下に示す。依存ライブラリの追加方法やメールセッションの取得方法等の詳細については、*E-mail 送信 (SMTP)* を参照。

```
package com.example.securelogin.domain.service.mail;

// omitted

@Service
public class PasswordReissueMailSharedServiceImpl implements
        PasswordReissueMailSharedService {
```

```
    @Inject
    JavaMailSender mailSender; // (1)

    @Inject
    @Named("passwordReissueMessage")
    SimpleMailMessage templateMessage; // (2)

    // omitted

    @Override
    public void send(String to, String text) {
        SimpleMailMessage message = new SimpleMailMessage(templateMessage); // (3)
        message.setTo(to);
        message.setText(text);
        mailSender.send(message);
    }

}
```

項番	説明
(1)	`org.springframework.mail.javamail.JavaMailSender` の Bean をインジェクションする。
(2)	送信元のメールアドレスとメールタイトルが設定された、`org.springframework.mail.SimpleMailMessage` の Bean をインジェクションする。 本アプリケーションでは `SimpleMailMessage` の Bean は一つしか定義されていないが、一般にはメールのテンプレートとして複数の Bean が定義されるため、 `@Named` で Bean 名を指定している。
(3)	テンプレートから `SimpleMailMessage` のインスタンスを生成し、引数として与えられた宛先メールアドレスと本文を設定して送信する。

```
package com.example.securelogin.domain.service.passwordreissue;

// omitted

@Service
@Transactional
```

```
public class PasswordReissueServiceImpl implements PasswordReissueService {

    @Inject
    ClassicDateFactory dateFactory;

    @Inject
    PasswordReissueMailSharedService mailSharedService;

    @Inject
    AccountSharedService accountSharedService;

    @Inject
    PasswordEncoder passwordEncoder;

    @Value("${security.tokenLifeTimeSeconds}")
    int tokenLifeTimeSeconds;

    @Value("${app.applicationBaseUrl}") // (1)
    String baseUrl;

    @Value("${app.passwordReissueProtocol}")
    String protocol;

    // omitted

    @Override
    public String createAndSendReissueInfo(String username) {

        String rowSecret = passwordGenerator.generatePassword(10,
                passwordGenerationRules);

        if (!accountSharedService.exists(username)) {
            return rowSecret;
        }

        Account account= accountSharedService.findOne(username);

        String token = UUID.randomUUID().toString();

        LocalDateTime expiryDate = dateFactory.newTimestamp().
↪toLocalDateTime()
                .plusSeconds(tokenLifeTimeSeconds);

        PasswordReissueInfo info = new PasswordReissueInfo();
        info.setUsername(username);
        info.setToken(token);
        info.setSecret(passwordEncoder.encode(rowSecret));
        info.setExpiryDate(expiryDate);

        passwordReissueInfoRepository.create(info);

        UriComponentsBuilder uriBuilder = UriComponentsBuilder.
↪fromUriString(baseUrl);
```

```
        uriBuilder.pathSegment("reissue").pathSegment("resetpassword")
                .queryParam("form").queryParam("token", info.getToken());  //␣
↪(2)
        String passwordResetUrl = uriBuilder.build().encode().toUriString();

        mailSharedService.send(account.getEmail(), passwordResetUrl); // (3)

        return rowSecret;

    }

    // omitted

}
```

項番	説明
(1)	パスワード再発行画面の URL に使用するベース URL をプロパティファイルから取得する。
(2)	(1) で取得した値と、生成したパスワード再発行用の認証情報に含まれるトークンを使用して、ユーザに配布するパスワード再発行画面の URL を作成する。 URL の作成には `org.springframework.web.util.UriComponentsBuilder` を利用する。`UriComponentsBuilder` については、ハイパーメディアリンクの実装 の中で説明されている。 上記例では、作成される URL のパス以下は」reissue/resetpassword?form&token=512f1a33-da20-4b9f-9e26-8961e9071618」のようになる。（token 部分はランダムに生成される。）
(3)	ユーザの登録メールアドレス宛てに、パスワード再発行画面の URL を本文に記したメールを送付する。

パスワード再発行実行時の検査

実装する要件一覧

- パスワード再発行用の認証情報に対する有効期限の設定

動作イメージ

パスワード再発行のための認証情報の配布 では、パスワード再発行のための認証情報の配布について説明した。ここでは、配布された認証情報を使用する際の処理について説明する。



パスワード再発行時の認証として、パスワード再発行のための認証情報の配布 でそれぞれ別配布したパスワード再発行画面の URL と秘密情報を照合する。URL に含まれるトークンと秘密情報の組が正しい場合にのみ、パスワードが再発行される。

また、一般的にはパスワードの再発行は認証情報の生成から間を置かずに行われるため、不必要に長期間有効となることが無いように、認証情報に有効期限を設定する。パスワード再発行画面の URL にアクセスした際に、認証情報が有効期限内であればパスワード再発行画面を表示し、有効期限が切れていればエラー画面に遷移する。

実装方法

メールで送付されるパスワード再発行画面の URL には、リクエストパラメータとしてトークンが含まれている。パスワード再発行画面へアクセスされた際にトークンを取得し、このトークンをキーとしてデータベースからパスワード再発行のための認証情報を検索する。

認証情報生成時にあらかじめ有効期限を設定しておき、データベースから取得した際に有効期限切れのチェックを行う。有効期限内であればパスワード変更画面を表示して秘密情報と新しいパスワードの入力を受け付ける。

データベースから取得した認証情報中の秘密情報と、ユーザが入力した秘密情報が一致すれば認証成功であり、パスワードを再発行する。

具体的には以下の三つの処理を実装することで要件を実現する。

- パスワード再発行用の認証情報に対する有効期限の設定

  パスワード再発行のための認証情報の生成 で説明した処理の中で、生成した認証情報に有効期限を設定する。

- パスワード再発行のための認証情報の有効期限の検査

パスワード再発行画面にアクセスされた際に、リクエストパラメータに含まれるトークンを取得し、トークンをキーとしてデータベースに保存されているパスワード再発行のための認証情報を検索する。認証情報に含まれる有効期限と現在時刻を比較し、有効期限が切れていればエラー画面に遷移させる。

- パスワード再発行のための認証情報を用いたユーザの確認

  パスワードの再発行を行う際に、ユーザ名、トークンとユーザが入力した秘密情報の組み合わせがデータベース内の認証情報と一致しているかどうかを確認する。一致する場合にはパスワードを再発行し、不一致の場合にはエラーメッセージを表示する。

コード解説

- パスワード再発行用の認証情報に対する有効期限の設定

  パスワード再発行用の認証情報への有効期限の設定自体は、 パスワード再発行のための認証情報の生成 で説明した処理に含まれている。ここでは、関連する実装箇所のみ再掲する。

  - Service の実装

```
package com.example.securelogin.domain.service.passwordreissue;

// omitted

@Service
@Transactional
public class PasswordReissueServiceImpl implements PasswordReissueService {

    @Inject
    ClassicDateFactory dateFactory;

    @Inject
    PasswordReissueInfoRepository passwordReissueInfoRepository;

    @Value("${security.tokenLifeTimeSeconds}")
    int tokenLifeTimeSeconds; // (1)

    // omitted

    @Override
    public String createAndSendReissueInfo(String username) {

        // omitted

        LocalDateTime expiryDate = dateFactory.newTimestamp().
↪toLocalDateTime()
                .plusSeconds(tokenLifeTimeSeconds); // (2)

        PasswordReissueInfo info = new PasswordReissueInfo(); // (3)
        info.setUsername(username);
        info.setToken(token);
        info.setSecret(passwordEncoder.encode(rowSecret));
        info.setExpiryDate(expiryDate);
```

```
        passwordReissueInfoRepository.create(info); // (4)

        // omitted (Send E-Mail)

    }

    // omitted

}
```

項番	説明
(1)	パスワード再発行用の認証情報が有効である期間の長さを秒単位で指定する。プロパティファイルに定義された値をインジェクションしている。
(2)	現在時刻に (1) の値を加えることにより、パスワード再発行用の認証情報の有効期限を計算する。
(3)	パスワード再発行用の認証情報を作成し、ユーザ名、トークン、秘密情報、有効期限を設定する。
(4)	パスワード再発行用の認証情報をデータベースに登録する。

- パスワード再発行のための認証情報の有効期限の検査

  パスワード再発行画面にアクセスされた際に、リクエストパラメータとして URL に含まれるトークンからパスワード再発行のための認証情報を取得し、有効期限内であるかどうかを検査する処理の実装を以下に示す。この処理中ではパスワード再発行の失敗上限を超過しているかどうかの検査も行うが、後述するため、ここでは省略する。

  - Service の実装

```
package com.example.securelogin.domain.service.passwordreissue;

// omitted

@Service
@Transactional
public class PasswordReissueServiceImpl implements PasswordReissueService {
```

```
    @Inject
    ClassicDateFactory dateFactory;

    @Inject
    PasswordReissueInfoRepository passwordReissueInfoRepository;

    // omitted

    @Override
    @Transactional(readOnly = true)
    public PasswordReissueInfo findOne(String token) {
        PasswordReissueInfo info = passwordReissueInfoRepository.
↪findOne(token); // (1)

        if (info == null) {
            throw new ResourceNotFoundException(ResultMessages.error().add(
                    MessageKeys.E_SL_PR_5002, token));
        }

        if (dateFactory.newTimestamp().toLocalDateTime()
                .isAfter(info.getExpiryDate())) { // (2)
            throw new BusinessException(ResultMessages.error().add(
                    MessageKeys.E_SL_PR_2001));
        }

        // omitted (attempts exceeded upper bounds)

        return info;
    }

    // omitted

}
```

項番	説明
(1)	引数として与えられたトークンをキーとして、パスワード再発行のための認証情報をデータベースから取得する。
(2)	有効期限が切れている場合は、`org.terasoluna.gfw.common.exception.BusinessException` を throw する。

- Controller の実装

```
package com.example.securelogin.app.passwordreissue;

// omitted

@Controller
@RequestMapping("/reissue")
public class PasswordReissueController {

    @Inject
    PasswordReissueService passwordReissueService;

    // omitted

    @RequestMapping(value = "resetpassword", params = "form")
    public String showPasswordResetForm(PasswordResetForm form, Model␣
↪model,
            @RequestParam("token") String token) { // (1)

        PasswordReissueInfo info = passwordReissueService.findOne(token); /
↪/ (2)

        form.setUsername(info.getUsername());
        form.setToken(token);
        model.addAttribute("passwordResetForm", form);
        return "passwordreissue/passwordResetForm";
    }

    // omitted

}
```

項番	説明
(1)	パスワード再発行画面の URL にリクエストパラメータとして含まれるトークンを取得する。
(2)	Service のメソッドにトークンを渡して呼び出す。データベースから認証情報が取得され、有効期限が検査される。

- パスワード再発行のための認証情報を用いたユーザの確認

  パスワード再発行画面においてユーザが入力した秘密情報と、パスワード再発行画面の URL に含まれるトークンの組が正しいかどうかを確認する処理の実装を以下に示す。この確認処理はパスワード再発行固有のロジックであり、かつデータベースの内容によって結果が異なるチェックであることから、Bean Validation や Spring Validator を用いず、Service に実装している。

- Service の実装

```
package com.example.securelogin.domain.service.passwordreissue;

// omitted

public interface PasswordReissueService {

    // omitted

    boolean resetPassword(String username, String token, String secret, //(1)
            String rawPassword);

    // omitted

}
```

項番	説明
(1)	引数として与えられたユーザ名、トークン、秘密情報を用いてユーザの確認を行った後、新しいパスワードを設定するメソッド

```
package com.example.securelogin.domain.service.passwordreissue;

// omitted

@Service
@Transactional
public class PasswordReissueServiceImpl implements PasswordReissueService {

    @Inject
    PasswordReissueFailureSharedService passwordReissueFailureSharedService;

    @Inject
    PasswordReissueInfoRepository passwordReissueInfoRepository;

    @Inject
    AccountSharedService accountSharedService;

    @Inject
    PasswordEncoder passwordEncoder;

    // omitted

    @Override
    public boolean resetPassword(String username, String token, String secret,
```

```
            String rawPassword) {
        PasswordReissueInfo info = this.findOne(token); // (1)
        if (!passwordEncoder.matches(secret, info.getSecret())) { // (2)
            passwordReissueFailureSharedService.resetFailure(username,␣
↪token);
            throw new BusinessException(ResultMessages.error().add(
                MessageKeys.E_SL_PR_5003));
        }
        failedPasswordReissueRepository.deleteByToken(token);
        passwordReissueInfoRepository.delete(token); // (3)

        return accountSharedService.updatePassword(username, rawPassword);␣
↪// (4)

    }

    // omitted

}
```

項番	説明
(1)	引数として与えられたトークンを用いて、データベースからパスワード再発行用の認証情報を取得する。このとき、有効期限が改めて検査される。
(2)	パスワード再発行用の認証情報に含まれるハッシュ化された秘密情報と、引数として与えられた秘密情報を比較する。異なる場合には `BusinessException` を throw する。この場合、パスワードの再発行は失敗となる。
(3)	使用された認証情報を再使用不能にするために、データベースから消去する。
(4)	引数として渡されたユーザ名を持つアカウントのパスワードを、指定された新しいパスワードに更新する。

- Form の実装

  クラスに付与されたアノテーションによって Null チェック以外の入力チェックが網羅されていることから、単項目チェックとしては`@NotNull` のみを付与している。

```
package com.example.securelogin.app.passwordreissue;

// omitted

@Data
@Compare(left = "newPassword", right = "confirmNewPassword", operator =␣
↪Compare.Operator.EQUAL)
@StrongPassword(usernamePropertyName = "username", newPasswordPropertyName␣
↪= "newPassword") // (1)
@NotReusedPassword(usernamePropertyName = "username",␣
↪newPasswordPropertyName = "newPassword") // (2)
public class PasswordResetForm implements Serializable{

    private static final long serialVersionUID = 1L;

    @NotNull
    private String username;

    @NotNull
    private String token;

    @NotNull
    private String secret;

    @NotNull
    private String newPassword;

    @NotNull
    private String confirmNewPassword;
}
```

項番	説明
(1)	パスワードの強度を検査するためのアノテーション。詳細は パスワードの品質チェック を参照。
(2)	パスワードの再利用を検査するためのアノテーション。詳細は パスワードの品質チェック を参照。

- View の実装

パスワード再発行画面 **(passwordResetForm.jsp)**

```
<body>
    <div id="wrapper">
        <h1>Reset Password</h1>
```

```
        <t:messagesPanel />
        <form:form
            action="${f:h(pageContext.request.contextPath)}/reissue/
↪resetpassword"
            method="POST" modelAttribute="passwordResetForm">
            <table>
                <tr>
                    <th><form:label path="username">Username</form:label></
↪th>
                    <td>${f:h(passwordResetForm.username)} <form:hidden
                            path="username" value="${f:h(passwordResetForm.
↪username)}" />  <!-- (1) -->
                    </td>
                    <td></td>
                </tr>
                <form:hidden path="token" value="${f:h(passwordResetForm.
↪token)}" /> <!-- (2) -->
                <tr>
                    <th><form:label path="secret" cssErrorClass="error-
↪label">Secret</form:label>
                    </th>
                    <td><form:password path="secret" cssErrorClass="error-
↪input" /></td> <!-- (3) -->
                    <td><form:errors path="secret" cssClass="error-messages
↪" /></td>
                </tr>
                <tr>
                    <th><form:label path="newPassword" cssErrorClass=
↪"error-label">New password</form:label>
                    </th>
                    <td><form:password path="newPassword"
                            cssErrorClass="error-input" /></td>
                    <td><form:errors path="newPassword" cssClass="error-
↪messages"
                            htmlEscape="false" /></td>
                </tr>
                <tr>
                    <th><form:label path="confirmNewPassword"
                            cssErrorClass="error-label">New␣
↪password(Confirm)</form:label></th>
                    <td><form:password path="confirmNewPassword"
                            cssErrorClass="error-input" /></td>
                    <td><form:errors path="confirmNewPassword"
                            cssClass="error-messages" /></td>
                </tr>
            </table>

            <input id="submit" type="submit" value="Reset password" />
        </form:form>
    </div>
</body>
```

項番	説明
(1)	ユーザ名を hidden 項目として保持する。
(2)	トークンを hidden 項目として保持する。
(3)	ユーザの確認のために、秘密情報を入力させる。

パスワード再発行画面 (**passwordResetComplete.jsp**)

```
<body>
    <div id="wrapper">
        <h1>Your password was successfully reset.</h1>
        <a href="${f:h(pageContext.request.contextPath)}/">go to Top</a>
    </div>
</body>
```

- Controller の実装

```
package com.example.securelogin.app.passwordreissue;

// omitted

@Controller
@RequestMapping("/reissue")
public class PasswordReissueController {

    @Inject
    PasswordReissueService passwordReissueService;

    // omitted

    @RequestMapping(value = "resetpassword", method = RequestMethod.POST)
    public String resetPassword(@Validated PasswordResetForm form,
            BindingResult bindingResult, Model model) {
        if (bindingResult.hasErrors()) {
            return showPasswordResetForm(form, model, form.getToken());
        }

        try {
            passwordReissueService.resetPassword(form.getUsername(),
                    form.getToken(), form.getSecret(), form.
↪getNewPassword()); // (1)
            return "redirect:/reissue/resetpassword?complete";
```

```
        } catch (BusinessException e) {
            model.addAttribute(e.getResultMessages());
            return showPasswordResetForm(form, model, form.getToken());
        }
    }

    @RequestMapping(value = "resetpassword", params = "complete", method =
↪RequestMethod.GET)
    public String resetPasswordComplete() {
        return "passwordreissue/passwordResetComplete";
    }

    // omitted

}
```

項番	説明
(1)	Service のメソッドにユーザ名、トークン、秘密情報、新しいパスワードを渡す。ユーザ名、トークン、秘密情報の組み合わせが正しい場合、新しいパスワードに更新される。

パスワード再発行の失敗上限回数の設定

実装する要件一覧

- パスワード再発行の失敗上限回数の設定

動作イメージ

パスワード再発行画面の URL が何らかの原因で漏えいした場合であっても、秘密情報が漏えいしていなければパスワードが不正に再発行されることはない。秘密情報には十分に推測困難なランダム値を用いているため簡単に破られる可能性は低いが、ブルートフォース攻撃を阻止する目的で認証失敗の回数に上限値を設定する。上限値を超えてパスワード再発行のための認証に失敗した場合、その URL（トークン）でのパスワード再発行が行えないようにする。

実装方法

本アプリケーションでは、パスワード再発行に失敗した履歴を「パスワード再発行失敗イベント」エンティティとしてデータベースに保存し、このパスワード再発行失敗イベントエンティティを用いて、パスワード再発行の失敗回数をカウントする。

失敗回数があらかじめ設定した上限値以上であれば、パスワード再発行画面へのアクセス時に例外をスローする。

具体的には、以下の二つの処理を実装して用いることにより、要件を実現する。

- パスワード再発行失敗イベントエンティティの保存



パスワード再発行実行時の検査 における、「パスワード再発行のための認証情報を用いたユーザの確認」処理の中で、ユーザの確認に失敗した場合に、使用したトークンと失敗日時の組をパスワード再発行失敗イベントエンティティとしてデータベースに登録する。

- パスワード再発行時の例外のスロー

  パスワード再発行のために認証情報をデータベースから取得した際に、パスワード再発行失敗イベントエンティティの数をカウントし、上限値以上であれば例外をスローする。

> **警告:** パスワード再発行失敗イベントエンティティはパスワード再発行の失敗回数のカウントのみを目的としているため、不要になったタイミングで消去する。パスワード再発行の失敗時のログが必要な場合は必ず別途ログを保存しておくこと。

コード解説

- 共通部分

  前提として、 パスワード再発行実行時の検査 に記した各処理が実装されているものとする。その他に共通的に必要な、データベースに対するパスワード再発行失敗イベントエンティティの登録、検索、削除に関する実装を以下に示す。

  - Entity の実装

    パスワード再発行失敗イベントエンティティの実装は以下の通り。

```
package com.example.securelogin.domain.model;

// omitted

@Data
public class FailedPasswordReissue {

    private String token; // (1)

    private LocalDateTime attemptDate; // (2)

}
```

項番	説明
(1)	パスワード再発行に使用したトークン
(2)	パスワード再発行を試行した日時

- Repository の実装

Entity の検索、登録、削除を行うための Repository を以下に示す。

```
package com.example.securelogin.domain.repository.passwordreissue;

// omitted

public interface FailedPasswordReissueRepository {

    int countByToken(@Param("token") String token); // (1)

    int create(FailedPasswordReissue event); // (2)

    int deleteByToken(@Param("token") String token); // (3)

    // omitted

}
```

項番	説明
(1)	引数として与えられたトークンをキーとして FailedPasswordReissue オブジェクトの個数を取得するメソッド
(2)	引数として与えられた FailedPasswordReissue オブジェクトをデータベースのレコードとして登録するメソッド
(3)	引数として与えられたトークンをキーとして FailedPasswordReissue オブジェクトを削除するメソッド

マッピングファイルは以下の通り。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper
 namespace="com.example.securelogin.domain.repository.passwordreissue.
↪FailedPasswordReissueRepository">

 <select id="countByToken" resultType="_int">
    <![CDATA[
        SELECT
            COUNT(*)
        FROM
            failed_password_reissue
        WHERE
            token = #{token}
    ]]>
 </select>

 <insert id="create" parameterType="FailedPasswordReissue">
    <![CDATA[
        INSERT INTO failed_password_reissue (
            token,
            attempt_date
        ) VALUES (
         #{token},
            #{attemptDate}
        )
    ]]>
 </insert>
```

```
    <delete id="deleteByToken">
        <![CDATA[
         DELETE FROM
             failed_password_reissue
         WHERE
             token = #{token}
        ]]>
    </delete>

</mapper>
```

以下、実装方法に従って実装されたコードについて順に解説する。

- パスワード再発行失敗イベントエンティティの保存

  パスワード再発行失敗時に行う処理を実装したクラスを以下に示す。

```
package com.example.securelogin.domain.service.passwordreissue;

public interface PasswordReissueFailureSharedService {

    void resetFailure(String username, String token);

}
```

```
package com.example.securelogin.domain.service.passwordreissue;

// omitted

@Service
@Transactional
public class PasswordReissueFailureSharedServiceImpl implements
        PasswordReissueFailureSharedService {

    @Inject
    ClassicDateFactory dateFactory;

    @Inject
    FailedPasswordReissueRepository failedPasswordReissueRepository;

    // omitted

    @Transactional(propagation = Propagation.REQUIRES_NEW) // (1)
    @Override
    public void resetFailure(String username, String token) {
        FailedPasswordReissue event = new FailedPasswordReissue(); // (2)
        event.setToken(token);
        event.setAttemptDate(dateFactory.newTimestamp().toLocalDateTime());
        failedPasswordReissueRepository.create(event); // (3)
    }

}
```

項番	説明
(1)	パスワード再発行に失敗した際に呼び出されるメソッドであり、呼び出し元で実行時例外を発生させる設計としている。 そのため、呼び出し元の Service とは別にトランザクション管理を行うために、伝搬方法を「REQUIRES_NEW」に指定する。
(2)	パスワード再発行失敗イベントエンティティを作成し、トークンと失敗日時を設定する。
(3)	(2) で作成したパスワード再発行失敗イベントエンティティをデータベースに登録する。

パスワード再発行実行時の検査 の「パスワード再発行のための認証情報を用いたユーザの確認」処理の中から、パスワード再発行失敗時の処理を呼び出す。

```
package com.example.securelogin.domain.service.passwordreissue;

// omitted

@Service
@Transactional
public class PasswordReissueServiceImpl implements PasswordReissueService {

    @Inject
    PasswordReissueFailureSharedService passwordReissueFailureSharedService;

    @Inject
    PasswordReissueInfoRepository passwordReissueInfoRepository;

    @Inject
    AccountSharedService accountSharedService;

    @Inject
    PasswordEncoder passwordEncoder;

    // omitted

    @Override
    public boolean resetPassword(String username, String token, String secret,
            String rawPassword) {
        PasswordReissueInfo info = this.findOne(token); // (1)
        if (!passwordEncoder.matches(secret, info.getSecret())) { // (2)
            passwordReissueFailureSharedService.resetFailure(username, token);
↪ // (3)
            throw new BusinessException(ResultMessages.error().add(  // (4)
```

```
                MessageKeys.E_SL_PR_5003));
        }

        //omitted

    }

    // omitted

}
```

項番	説明
(1)	引数として与えられたトークンを用いて、データベースからパスワード再発行用の認証情報を取得する。
(2)	パスワード再発行用の認証情報に含まれるハッシュ化された秘密情報と、引数として与えられた秘密情報を比較する。
(3)	パスワード再発行失敗時の処理を行う SharedService のメソッドを呼び出す。
(4)	実行時例外を throw するが、パスワード再発行失敗時の処理は別のトランザクションで実行されるため、影響を与えることはない。

- パスワード再発行時の例外のスロー

  パスワード再発行の失敗回数の取得と、失敗回数が上限に達した際の処理の実装を以下に示す。

```
package com.example.securelogin.domain.service.passwordreissue;

// omitted

@Service
@Transactional
public class PasswordReissueServiceImpl implements PasswordReissueService {

    @Inject
    FailedPasswordReissueRepository failedPasswordReissueRepository;

    @Inject
    PasswordReissueInfoRepository passwordReissueInfoRepository;
```

```
    @Value("${security.tokenValidityThreshold}")
    int tokenValidityThreshold; // (1)

    // omitted

    @Override
    @Transactional(readOnly = true)
    public PasswordReissueInfo findOne(String token) {

        // omitted

        int count = failedPasswordReissueRepository.countByToken(token); //
↪(2)
        if (count >= tokenValidityThreshold) { // (3)
            throw new BusinessException(ResultMessages.error().add(
                    MessageKeys.E_SL_PR_5004));
        }

        return info;
    }

    // omitted

}
```

項番	説明
(1)	パスワード再発行の失敗回数の上限値をプロパティファイルから取得して設定する。
(2)	引数として与えられたトークンをキーとして、データベースからパスワード再発行失敗イベントエンティティの数を取得。
(3)	取得したパスワード再発行の失敗イベントエンティティの数と失敗回数の上限値を比較し、上限値以上ならば例外をスローする。

セキュリティ観点での入力値チェック

実装する要件一覧

- リクエストパラメータに対する共通的な禁止文字の設定
- アップロードファイル名に対する共通的な禁止文字列の設定

- 制御文字の入力チェック
- ファイル拡張子の入力チェック
- ファイル名の入力チェック
- *URL* のドメインに対する入力チェック
- メールアドレスのドメインに対する入力チェック

動作イメージ

- 共通的な禁止文字列の設定



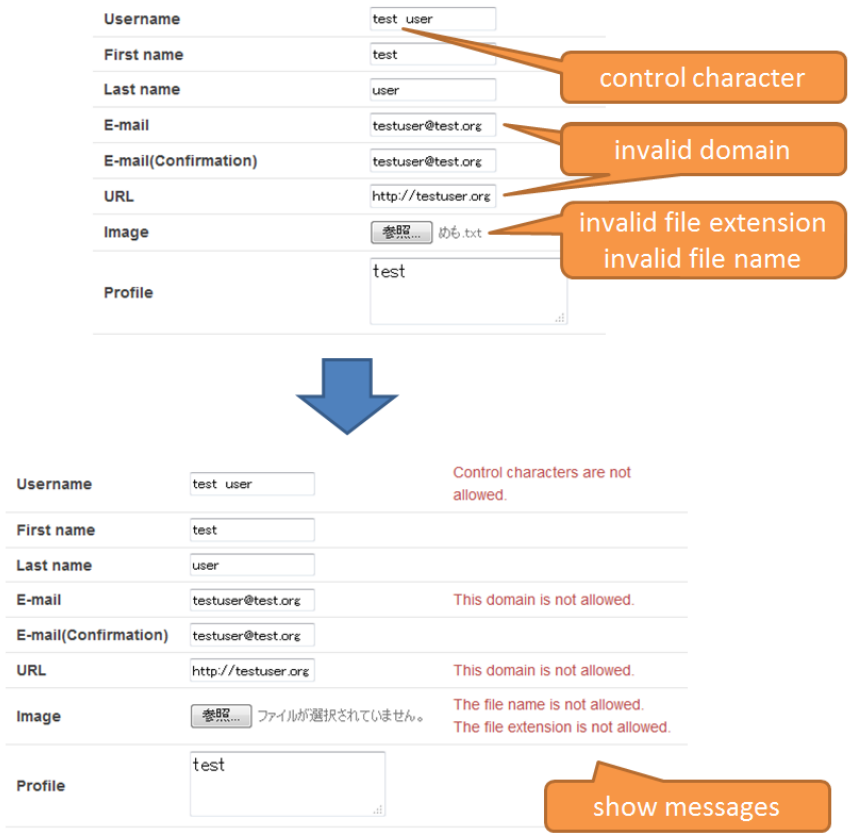
- 個別の入力チェック

実装方法

アプリケーション全体の共通的な禁止文字列の設定と個別の入力チェックとではチェックの対象範囲が大きく異なるため、それぞれ別の方法で実装を行う。

共通的な禁止文字列を設定するためには、*Controller* の呼び出し前後で行う共通処理の実装 に記した二つの方法を用いることができる。本アプリケーションでは、Controller のハンドラメソッドにマッピングされるか否かに関わらずチェックを行うために、Servlet Filter を用いて実装を行う。入力エラーが発生した場合、通常のユーザ操作の結果としては想定できない入力値が入力されたことを意味するため、ユーザビリティの低下を考慮せずに共通的なエラー画面へ遷移させるように設定ファイルへ記述する。

個別の入力チェックには 入力チェック の機能を利用することができる。本アプリケーションでは Bean Validation を用いて入力値のチェックを実現する。個別の入力エラーに対しては、Bean Validation 当該入力項目に対してエラーメッセージを表示して再入力を促すよう実装を行う。



コード解説

上記の実装方法に従って実装されたコードについて順に解説する。

- Servlet Filter の実装

ユーザからの入力をアプリケーション内で使用する場合、SQL インジェクションや XSS、ディレクトリトラバーサル（パストラバーサル）といったインジェクション攻撃の対象となる可能性がある。

これらの攻撃への対策として、アプリケーション全体でユーザからの入力を検証するための Servlet Filter の実装例を示す。

本アプリケーションでは、以下の項目に対してそれぞれ設定した禁止文字を含んでいないことを検証する。

項番	項目	説明
(1)	リクエストパラメータ	リクエストパラメータはユーザからの入力を受け付けるために一般的に利用されるため、入力チェックの対象とする パラメータ名と値の両方がユーザからの入力となる可能性があるため、両方をチェックする
(2)	アップロードファイル名	本アプリケーションではファイルアップロード機能を実装しているため、ユーザからの入力であるアップロードファイル名を入力チェックの対象とする

尚、以下のコードでは `org.springframework.web.multipart.MultipartRequest` を使用するため、`org.springframework.web.multipart.support.MultipartFilter` の使用を前提としている。`MultipartFilter` については ファイルアップロードを参照のこと。

```
package com.example.securelogin.app.common.filter;

// omitted

public class InputValidationFilter extends OncePerRequestFilter { // (1)

    private final List<Character> prohibitedChars;

    private final List<Character> prohibitedCharsForFileName;

    public InputValidationFilter(char[] prohibitedChars,
            char[] prohibitedCharsForFileName) {
        this.prohibitedChars = Chars.asList(prohibitedChars); // (2)
        this.prohibitedCharsForFileName = Chars
                .asList(prohibitedCharsForFileName); // (3)
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
            HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        if (request != null) {
            validateRequestParams(request); // (4)

            if (request instanceof MultipartRequest) {
                validateFileNames((MultipartRequest) request); // (5)
            }
```

```
        }

        filterChain.doFilter(request, response); // (6)
    }

    private void validateRequestParams(HttpServletRequest request) {
        Map<String, String[]> params = request.getParameterMap();
        for (Map.Entry<String, String[]> entry : params.entrySet()) {
            validate(entry.getKey(), prohibitedChars); // (7)
            for (String value : entry.getValue()) {
                validate(value, prohibitedChars); // (8)
            }
        }
    }

    private void validateFileNames(MultipartRequest request) {
        for (Map.Entry<String, MultipartFile> entry : request.getFileMap()
                .entrySet()) {
            String filename = new File(entry.getValue().getOriginalFilename())
                    .getName(); // (9)
            validate(filename, prohibitedCharsForFileName); // (10)
        }
    }

    private void validate(String target, List<Character> prohibited) {
        if (StringUtils.hasLength(target)) {
            List<Character> chars = Chars.asList(target.toCharArray());
            for(Character prohibitedChar : prohibited) { // (11)
                if (chars.contains(prohibitedChar)) {
                    throw new InvalidCharacterException(
                        "The request contains prohibited charcter.");
                }
            }
        }
    }

}
```

項番	説明
(1)	`org.springframework.web.filter.OncePerRequestFilter` を継承することで、リクエストに対して一度だけ処理が実行されることが保証される
(2)	リクエストパラメータの禁止文字の一覧を文字列として受け取り、文字のリストとして保持する

Continued on next page

表 9.144 – continued from previous page

項番	説明
(3)	ファイル名の禁止文字の一覧を文字列として受け取り、文字のリストとして保持する
(4)	リクエストパラメータの入力チェックを行うメソッドを呼び出す
(5)	ファイルアップロードリクエスト (Content-Type が `multipart/form-data` の POST リクエスト) の場合、`MultipartFilter` の機能によって、ここでの `request` オブジェクトは `MultipartRequest` を実装した `StandardMultipartHttpServletRequest` のインスタンスとなる `MultipartRequest` の API を用いてファイル名を取り出し、入力チェックを行うメソッドを呼び出す
(6)	入力チェックが完了した後、後続の Servlte Filter の処理を実行するためのメソッドを呼び出す
(7), (8)	`HttpServletRequest` からリクエストパラメータの一覧を取得し、各リクエストパラメータ名およびリクエストパラメータ値について、実際の入力チェックを行うメソッドを呼び出す
(9)	`MultipartRequest` からアップロードされたファイルの一覧を取得し、実際のファイル名を取得する クライアントのブラウザや OS によってはファイル名にパスが含まれたり、パス区切り文字が異なるため、ファイル名のみを取得する際にこのような処理が必要となる
(10)	アップロードされた各ファイルのファイル名について、実際の入力チェックを行うメソッドを呼び出す
(11)	入力チェック対象の文字列を一文字ずつ順に、禁止文字に含まれているかどうかをチェックし、含まれている場合は例外を投げる `InvalidCharacterException` は `RuntimeException` を継承して作成した例外である。コードは省略する

ちなみに: 本アプリケーションではリクエストパラメータおよびファイル名のチェックのみ共通的な禁止文字のチェックを行っている。必要に応じて、`HttpServletRequest` の `getHeaders` や `getCookies` を用いて HTTP ヘッダや Cookie の値を取得することにより、HTTP ヘッダや Cookie に対しても同様の方法でチェックすることができる。

- Servlet Filter の設定

作成した `InputValidationFilter` を有効にするため、web.xml に設定を行う。

禁止文字をプロパティファイルから読み込み、`InputValidationFilter` を Bean 定義した上で、`org.springframework.web.filter.DelegatingFilterProxy` を使用して設定する。

web.xml

```
<filter>
    <filter-name>MultipartFilter</filter-name>
    <filter-class>org.springframework.web.multipart.support.MultipartFilter</
↪filter-class>  <!-- (1) -->
</filter>
<filter-mapping>
    <filter-name>MultipartFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<filter>
    <filter-name>inputValidationFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</
↪filter-class>  <!-- (2) -->
</filter>
<filter-mapping>
    <filter-name>inputValidationFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- omitted -->

<error-page>
    <exception-type>com.example.securelogin.app.common.filter.exception.
↪InvalidCharacterException</exception-type>  <!-- (3) -->
    <location>/WEB-INF/views/common/error/invalidCharacterError.jsp</location>
</error-page>
```

項番	説明
(1)	InputValidationFilter の使用の前提となっている MultipartFilter を設定する MultipartFilter を InputValidationFilter よりも前に定義する必要があることに注意すること
(2)	DelegatingFilterProxy を用いて、Bean 定義した InputValidationFilter を設定する <filter-name> には Bean 名を指定すること
(3)	InvalidCharacterException がスローされた際に表示するエラー画面を設定する

注釈: ファイル名の入力チェックのために MultipartFilter を利用しているため、ここに記述した内容に加えて アプリケーションの設定 に記した Servlet 3.0 のアップロード機能を有効化するための設定が必要となる。

---

invalidCharacterError.jsp

```
<% response.setStatus(HttpServletResponse.SC_BAD_REQUEST); %>  <!-- (1) -->
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Invalid Character Error!</title>

<!-- omitted -->
```

項番	説明
(1)	InvalidCharacterException はクライアントの入力に起因して発生する例外であるため、の HTTP ステータスコードを 400 (Bad Request) に設定する

applicationContext.xml

```
<bean id="inputValidationFilter" class="com.example.securelogin.app.common.
↪filter.InputValidationFilter">
```

```
    <constructor-arg index="0" value="${app.security.prohibitedChars}"/>  <!--
↪ (1) -->
    <constructor-arg index="1" value="${app.security.
↪prohibitedCharsForFileName}"/>  <!-- (2) -->
</bean>
```

項番	説明
(1)	リクエストパラメータの禁止文字の一覧を文字列としてプロパティから取得する
(2)	ファイル名の禁止文字の一覧を文字列としてプロパティから取得する

application.properties

```
## (1)
app.security.prohibitedChars=&\\!"<>*

## (2)
app.security.prohibitedCharsForFileName=&\\!"<>*;:
```

項番	説明
(1)	本アプリケーションにおいて、リクエストパラメータに含まれることを想定していない文字のリストを文字列として指定する
(2)	本アプリケーションにおいて、アップロードファイル名に含まれることを想定していない文字のリストを文字列として指定する

- Bean Validation のアノテーションの作成

  アプリケーションの仕様で想定していない入力値を入力されることによるセキュリティ上のリスクを軽減するために、要件に合わせて入力値を検証する Bean Validation のアノテーションを作成する。

  - 制御文字が含まれないことを検証するアノテーション

    入力値に制御文字が含まれている場合、アプリケーションが予期しない問題を引き起こす可能性がある。そのため、制御文字を入力する必要がない入力項目に対して制御文字が含まれていないこと

をチェックする。

文章の入力の際には制御文字のうち改行コードのみを許容する場合も多いため、改行コードを許可するアノテーションも別途作成する。

正規表現を用いたチェックを行うことで制御文字が含まれていないことを検証することができる。

```
package com.example.securelogin.app.common.validation;

// omitted
@Documented
@Constraint(validatedBy = {})
@Target(FIELD)
@Retention(RUNTIME)
@ReportAsSingleViolation  // (1)
@Pattern(regexp = "^\\P{Cntrl}*$") // (2)
public @interface NotContainControlChars {
    String message() default "{com.example.securelogin.app.common.
↪validation.NotContainControlChars.message}";

    Class<?>[] groups() default {};

    @Target(FIELD)
    @Retention(RUNTIME)
    @Documented
    @interface List {
        NotContainControlChars[] value();
    }

    Class<? extends Payload>[] payload() default {};
}
```

項番	説明
(1)	エラーメッセージとして、このアノテーションで指定したメッセージのみを出力するために`@ReportAsSingleViolation` アノテーションを付与する
(2)	正規表現を用いた入力チェックを行うために`@Pattern` アノテーションを付与する `\P{Cntrl}` は Java の正規表現において「制御文字以外の文字」を意味するため、`^\\P{Cntrl}*$` は最初から最後まで制御文字を含まない文字列のみにマッチする

同様に、改行コードを許容する場合のアノテーションの実装例を以下に示す。

```
package com.example.securelogin.app.common.validation;

// omitted
@Documented
@Constraint(validatedBy = {})
@Target(FIELD)
@Retention(RUNTIME)
@ReportAsSingleViolation
@Pattern(regexp = "^[\\r\\n\\P{Cntrl}]*$") // (1)
public @interface NotContainControlCharsExceptNewlines {
    String message() default "{com.example.securelogin.app.common.
↪validation.NotContainControlCharsExceptNewlines.message}";

    Class<?>[] groups() default {};

    @Target(FIELD)
    @Retention(RUNTIME)
    @Documented
    @interface List {
        NotContainControlCharsExceptNewlines[] value();
    }

    Class<? extends Payload>[] payload() default {};
}
```

項番	説明
(1)	「制御文字以外の文字 (\P{Cntrl})」と改行コード (\r , \n) のみで構成される文字列にマッチするように、正規表現を指定する。

- アップロードされるファイルの拡張子が許容されているものであることを検証するアノテーション

ユーザからのファイルのアップロードを受け付ける際、受け付けるファイルの形式を制限したい場合がある。そのような場合に、許容するファイルの拡張子の一覧を設定し、アップロードされたファイルの拡張子が一覧に含まれているかをチェックする。

拡張子の大文字・小文字を区別するか否かを切り替えられる仕様とする。

> **警告:** ファイルの拡張子は容易に偽装され得るため、拡張子のチェックを行った場合であっても無条件にファイル形式を信用してはならない。

```
package com.example.securelogin.app.common.validation;

// omitted

@Documented
@Constraint(validatedBy = { FileExtensionValidator.class })
@Target(FIELD)
@Retention(RUNTIME)
public @interface FileExtension {
    String message() default "{com.example.securelogin.app.common.
↪validation.FileExtension.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    String[] extensions();  // (1)

    boolean ignoreCase() default true;  // (2)

    @Target(FIELD)
    @Retention(RUNTIME)
    @Documented
    @interface List {
        FileExtension[] value();
    }
}
```

項番	説明
(1)	許容する拡張子の一覧を設定する
(2)	大文字・小文字の違いを無視するか否か。デフォルト値は `true` （無視する）

```
package com.example.securelogin.app.common.validation;

// omitted

public class FileExtensionValidator implements
        ConstraintValidator<FileExtension, MultipartFile> {

    private Set<String> extensions;

    private boolean ignoreCase;

    @Override
```

```
    public void initialize(FileExtension constraintAnnotation) {
        this.extensions = new HashSet<String>(
                Arrays.asList(constraintAnnotation.extensions()));
        this.ignoreCase = constraintAnnotation.ignoreCase();
    }

    @Override
    public boolean isValid(MultipartFile value,
            ConstraintValidatorContext context) {
        if (value == null) {  // (1)
            return true;
        }

        String fileNameExtension = StringUtils.getFilenameExtension(value
                .getOriginalFilename());  // (2)
        if (!StringUtils.hasLength(fileNameExtension)) {  // (3)
            return false;
        }

        for (String extension : extensions) {  // (4)
            if (fileNameExtension.equals(extension) || ignoreCase
                    && fileNameExtension.equalsIgnoreCase(extension)) {
                return true;
            }
        }
        return false;
    }

}
```

項番	説明
(1)	null チェックは他のアノテーションを用いて行うため、null の場合は true を返す
(2)	org.springframework.util.StringUtils の getFilenameExtension メソッドを用いてファイル名から拡張子を取得する
(3)	拡張子の無いファイルは許容しないため、拡張子が無い場合は false を返す
(4)	許容する拡張子の一覧から拡張子を一つずつ取り出し、ファイル名から取得した文字列と比較する 一つでも一致するものがあれば true を返し、いずれとも一致しなければ false を返す。

- アップロードされるファイルのファイル名が許容されているパターンと一致することを検証するアノテーション

  ユーザからのファイルのアップロードを受け付ける際、受け付けるファイル名の形式を制限したい場合がある。そのような場合に、許容するファイル名のパターンを正規表現で設定し、アップロードされたファイルのファイル名がパターンと一致するかをチェックする。

```
package com.example.securelogin.app.common.validation;

// omitted

@Documented
@Constraint(validatedBy = { FileNamePatternValidator.class })
@Target(FIELD)
@Retention(RUNTIME)
public @interface FileNamePattern {

    String message() default "{com.example.securelogin.app.common.
↪validation.FileNamePattern.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    String pattern() default "";  // (1)

    @Target(FIELD)
```

```
    @Retention(RUNTIME)
    @Documented
    @interface List {
        FileNamePattern[] value();
    }

}
```

項番	説明
(1)	許容するファイル名のパターン

```
package com.example.securelogin.app.common.validation;

// omitted

public class FileNamePatternValidator implements
        ConstraintValidator<FileNamePattern, MultipartFile> {

    private Pattern pattern;

    @Override
    public void initialize(FileNamePattern constraintAnnotation) {
        this.pattern = Pattern.compile(constraintAnnotation.pattern());  // (1)
    }

    @Override
    public boolean isValid(MultipartFile value,
            ConstraintValidatorContext context) {
        if (value == null) {  // (2)
            return true;
        }

        String filename = new File(value.getOriginalFilename()).getName();  // (3)
        return pattern.matcher(filename).matches();  // (4)
    }

}
```

項番	説明
(1)	チェックする正規表現のパターンを `Pattern` として作成し、保持する
(2)	`null` チェックは他のアノテーションを用いて行うため、`null` の場合は `true` を返す
(3)	`MultipartRequest` から実際のファイル名を取得する。クライアントのブラウザや OS によってはファイル名にパスが含まれたり、パス区切り文字が異なるため、ファイル名のみを取得する際にこのような処理が必要となる
(4)	(1) で作成した `Pattern` とファイル名がマッチする場合は `true` を返し、マッチしない場合は `false` を返す

- 入力された URL のドメインが許容されているものであることを検証するアノテーション

ユーザから URL の入力を受け付ける際、許容するドメインを制限したい場合がある。そのような場合に、許容するドメインの一覧を設定し、入力された URL のドメインが一覧に含まれているドメインかまたはそのサブドメインであるかをチェックする。

同時に URL 形式であることをチェックするため、
`org.hibernate.validator.constraints.URL` と組み合わせて実装する。

```
package com.example.securelogin.app.common.validation;

// omitted

@Documented
@Constraint(validatedBy = { DomainRestrictedURLValidator.class })
@Target(FIELD)
@Retention(RUNTIME)
@URL  // (1)
public @interface DomainRestrictedURL {

    String message() default "{com.example.securelogin.app.common.
↪validation.DomainRestrictedURL.message}";

    Class<?>[] groups() default {};
```

```
    String[] allowedDomains() default {};  // (2)

    @Target(FIELD)
    @Retention(RUNTIME)
    @Documented
    @interface List {
        DomainRestrictedURL[] value();
    }

    Class<? extends Payload>[] payload() default {};

}
```

項番	説明
(1)	URL 形式であることをチェックするために、@URL を付与する
(2)	許容するドメインの一覧

```
package com.example.securelogin.app.common.validation;

// omitted

public class DomainRestrictedURLValidator implements
        ConstraintValidator<DomainRestrictedURL, String> {

    private static final Pattern URL_REGEX = Pattern  // (1)
        .compile( "(?i)^(?:[a-z](?:[-a-z0-9\\+\\.])*)" + // protocol
                  ":(?:\\/\\/([^\\/:]+)" + // auth+host/ip
                  "(?::([0-9]*))?" + // port
                  "(?:\\/.*)*)$"
         );

    private Set<String> allowedDomains;

    @Override
    public void initialize(DomainRestrictedURL constraintAnnotation) {
        allowedDomains = new HashSet<String>(Arrays.
↪asList(constraintAnnotation
                .allowedDomains()));  // (2)
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext␣
↪context) {
        Matcher urlMatcher = URL_REGEX.matcher(value);
```

```
        if (urlMatcher.matches()) {  // (3)
            String host = urlMatcher.group(1);
            for(String domain : allowedDomains) {  // (4)
                if (StringUtils.hasLength(host) && host.endsWith(".
↪"+domain)) {
                    return true;
                }
            }
            return false;
        } else {
            return true;
        }
    }

}
```

項番	説明
(1)	URL のドメインを取得するための正規表現のパターンを `Pattern` として作成し、保持する 正しい URL 形式であるかどうかの検証は`@URL` で行うため、ここでは必要最小限の正規表現を指定している
(2)	許容するドメインの一覧を取得し、保持する
(3)	URL 形式であるかどうかをチェックする。不正な形式の場合は、組み合わせて利用している `URL` アノテーションによって検証エラーとなるため、ここでは `true` を返す
(4)	許容されるドメインの一覧から一つずつ取り出し、URL のホスト部の末尾と一致するかをチェックする。一つでも一致すれば `true` を返し、一つも一致しなければ `false` を返す

- 入力されたメールアドレスのドメインが許容されているものであることを検証するアノテーション

ユーザからメールアドレスの入力を受け付ける際、許容するドメインを制限したい場合がある。そのような場合に、許容するドメインの一覧を設定し、入力されたメールアドレスのドメインが一覧に含まれているかをチェックする。許容するドメインのサブドメインまで許すか否かを切り替えられる仕様とする。

同時にメールアドレス形式であることをチェックするため、
`org.hibernate.validator.constraints.Email` と組み合わせて実装する。

```
package com.example.securelogin.app.common.validation;

// omitted

@Documented
@Constraint(validatedBy = { DomainRestrictedEmailValidator.class })
@Target(FIELD)
@Retention(RUNTIME)
@Email  // (1)
public @interface DomainRestrictedEmail {
    String message() default "{com.example.securelogin.app.common.
↪validation.DomainRestrictedEmail.message}";

    Class<?>[] groups() default {};

    String[] allowedDomains() default {};  // (2)

    boolean allowSubDomain() default false;  // (3)

    @Target(FIELD)
    @Retention(RUNTIME)
    @Documented
    @interface List {
        DomainRestrictedEmail[] value();
    }

    Class<? extends Payload>[] payload() default {};
}
```

項番	説明
(1)	メールアドレス形式であることをチェックするために、`@Email` を付与する
(2)	許容するドメインの一覧
(3)	サブドメインを許容するか否か。デフォルト値は `false`（許容しない）

```
package com.example.securelogin.app.common.validation;
```

```
// omitted

public class DomainRestrictedEmailValidator implements
        ConstraintValidator<DomainRestrictedEmail, CharSequence> {

    private Set<String> allowedDomains;

    private boolean allowSubDomain;

    @Override
    public void initialize(DomainRestrictedEmail constraintAnnotation) {
        allowedDomains = new HashSet<String>(Arrays.
↪asList(constraintAnnotation
                .allowedDomains()));  // (1)
        allowSubDomain = constraintAnnotation.allowSubDomain();  // (2)
    }

    @Override
    public boolean isValid(CharSequence value,
            ConstraintValidatorContext context) {
        if (value == null) {  // (3)
            return true;
        }

        for (String domain : allowedDomains) {  // (4)
            if (value.toString().endsWith("@" + domain)
                    || (allowSubDomain && value.toString().endsWith(
                            "." + domain))) {
                return true;
            }
        }
        return false;
    }

}
```

項番	説明
(1)	許容するドメインの一覧を取得し、保持する
(2)	サブドメインを許容するかどうかを表す真理値を取得し、保持する
(3)	`null` チェックは他のアノテーションを用いて行うため、`null` の場合は `true` を返す
(4)	許容されるドメインの一覧から一つずつ取り出し、メールアドレスのドメイン部と一致するかをチェックする。一つでも一致すれば `true` を返し、一つも一致しなければ `false` を返す

- Form クラスへのアノテーションの付与

  作成したアノテーションをアカウント新規作成の Form クラスのフィールドに付与する。

```
package com.example.securelogin.app.account;

// omitted

public class AccountCreateForm implements Serializable {

    // omitted

    @NotNull
    @NotContainControlChars  // (1)
    @Size(min=4, max=128)
    private String username;

    // omitted

    @NotNull
    @NotContainControlChars
    @Size(min=1, max=128)
    @DomainRestrictedEmail(allowedDomains={ "domainexample.co.jp",
            "somedomainexample.co.jp" }, allowSubDomain=true)  // (2)
    private String email;

    // omitted

    @NotNull
    @NotContainControlChars
```

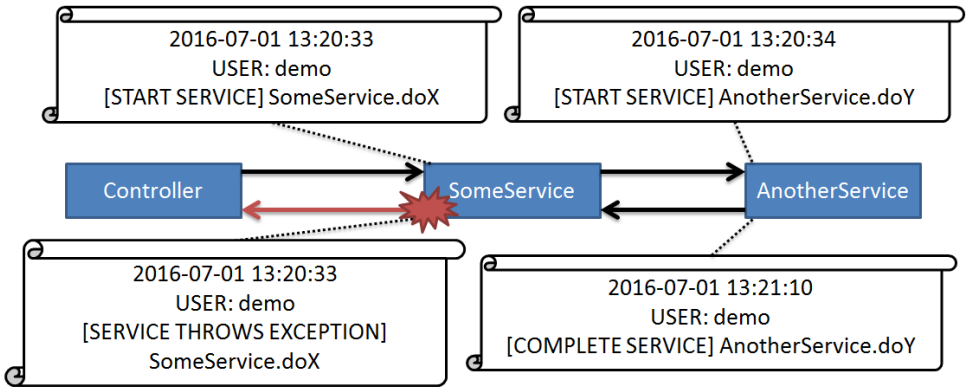
```
    @DomainRestrictedURL(allowedDomains={ "jp" })  // (3)
    private String url;

    @UploadFileRequired
    @UploadFileNotEmpty
    @UploadFileMaxSize
    @FileExtension(extensions = { "jpg", "png", "gif" })  // (4)
    @FileNamePattern(pattern = "[a-zA-Z0-9_-]+\\.[a-zA-Z]{3}")  // (5)
    private MultipartFile image;

    @NotNull
    @NotContainControlCharsExceptNewlines  // (6)
    private String profile;

}
```

項番	説明
(1)	制御文字が含まれないことをチェックする
(2)	メールアドレスのドメインが 「ntt.co.jp」, 「nttdata.co.jp」 またはそのサブドメインであることチェックする
(3)	URL のドメインが 「jp」 またはそのサブドメインであることをチェックする
(4)	ファイルの拡張子が 「jpg」, 「png」, 「gif」 のどれかであることをチェックする
(5)	ファイル名が『「半角英数字, 「_」, 「-」 の 1 文字以上の繰り返し」 + 「 「.」 」 + 「半角英字 3 文字」』というパターンとなっていることをチェックする
(6)	改行コード以外の制御文字が含まれないことをチェックする

監査ログ出力
実装する要件一覧

- 監査ログ出力

動作イメージ



監査目的で、アプリケーションに対していつ、誰が、どのような操作を行ったのかといった情報を確認できるようにするため、サービスクラスのメソッドを呼び出す際に、呼び出し日時、呼び出し元ユーザ名、メソッド名をログ出力する。また、メソッド実行の結果として例外が発生しなければ操作成功としてログを出力し、例外が発生した場合は操作失敗としてログを出力する。

ログのフォーマットは以下の通りとする。

{メソッド呼び出し日時}{スレッド}{呼び出し元ユーザ名}{`X-Track`}{ログレベル}{ロガー}{メッセージ}

項目	説明
メソッド呼び出し日時	サービスクラスのメソッドを呼び出した日時を」yyyy-MM-dd HH:mm:ss」形式で出力
スレッド	ログ出力を行ったスレッド
呼び出し元ユーザ名	サービスクラスのメソッドを呼び出した Spring Security のユーザ名を出力 未ログインの場合には空欄とする
X-Track	トレーサビリティ向上のために、リクエストごとに設定する ID 詳細はロギング を参照すること
ログレベル	出力されたログのレベル 本アプリケーションにおいては `info` レベルで出力する

Continued on next page

表 9.145 – continued from previous page

項目	説明
ロガー	ログを出力したロガー
メッセージ	メソッド呼び出し時：」[START SERVICE] ServiceClassName.methodName」 メソッド正常終了時：」[COMPLETE SERVICE] ServiceClassName.methodName」 例外発生時：」[SERVICE THROWS EXCEPTION] ExceptionClassName.methodName」

実装方法

ログにユーザ名を出力するために、Spring Security の認証情報からユーザ名を取得する。 ユーザ名の取得およびログ出力はロギング で解説している `org.terasoluna.gfw.security.web.logging.UserIdMDCPutFilter` を用いて実現する。
さらに、本アプリケーションにおいて、リクエストに対する操作内容と操作結果をそれぞれ以下の通り定義し、ログに出力する。

- 操作内容：呼び出されたサービスクラスのメソッド名
- 操作結果：メソッドの処理を実行した結果例外が発生したか否か

すべてのサービスクラスのメソッド呼び出しに対してログ出力を行うといった、横断的な機能を実現するためには、Spring が提供する AOP(Aspect Oriented Programming) の機能を利用することができる。
Spring が提供している AOP の実装方法は複数あるが、本アプリケーションでは 共通ライブラリ で提供しているロギング関連の部品の実装と合わせることを重視し、`org.aopalliance.intercept.MethodInterceptor` を実装する方式を採用する。
具体的には以下の実装・設定を行うことで要件を実現する。

- `UserIdMDCPutFilter` を設定する
- メソッド呼び出し時および実行後にログ出力を行うアドバイスを作成する
- 上記で定義したアドバイスを`@Service` の付与されたクラスに対して適用するための設定を行う

> **注釈:** アドバイスとは、AOP において指定されたタイミングで実行する処理のことを指す。また、アドバイスを織り込むことのできる箇所のことをジョインポイントと呼び、どのジョインポイントにアドバイスを織り込むかを定義したものポイントカットと呼ぶ。Spring が提供する AOP 機能に関しては、公式ドキュメント - AOP を参照すること。

コード解説

上記の実装方法に従って実装されたコードについて順に解説する。

- `UserIdMDCPutFilter` を設定する

  ログに Spring Security の認証ユーザ名を出力するための設定を以下に示す。

  spring-security.xml

```
<!-- omitted -->

<sec:http>
    <!-- omitted -->
    <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER" />
```

```
    <!-- omitted -->
</sec:http>

<!-- omitted -->
<bean id="userIdMDCPutFilter" class="org.terasoluna.gfw.security.web.logging.
↪UserIdMDCPutFilter">
</bean>
```

項番	説明
(1)	ユーザ情報が生成された後すぐにログ出力するために、Spring Security の Filter Chain に UserIdMDCPutFilter を設定する UserIdMDCPutFilter を設定することによって、MDC に USER というキーで認証ユーザ名が追加される。

logback.xml

```
<!-- omitted -->

<appender name="AUDIT_LOG_FILE"
    class="ch.qos.logback.core.rolling.RollingFileAppender">  <!-- (1) -->
    <file>log/security-audit.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>log/security-audit-%d{yyyyMMdd}.log</fileNamePattern>
        <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder>
        <charset>UTF-8</charset>
        <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tthread:%thread\tUSER:
↪%X{USER}\tX-Track:%X{X-Track}\tlevel:%-5level\tlogger:%-48logger{48}
↪\tmessage:%msg%n]]></pattern>  <!-- (2) -->
    </encoder>
</appender>

<!-- omitted -->

<logger
   name="com.example.securelogin.domain.common.interceptor.
↪ServiceCallLoggingInterceptor"
   additivity="false">  <!-- (3) -->
   <level value="info" />
   <appender-ref ref="AUDIT_LOG_FILE" />
</logger>

<!-- omitted -->
```

項番	説明
(1)	監査ログ出力用の appender を定義する
(2)	pattern 定義内に」USER:%X{USER}」を記述する
(3)	監査ログ出力用の logger を定義する `com.example.securelogin.domain.common.interceptor.ServiceCallLoggingInterceptor` の実装については以降で説明する

- メソッド呼び出し時および実行後にログ出力を行うアドバイスを作成する

  操作内容と操作結果をログ出力するための実装および設定を以下に示す。

```
package com.example.securelogin.domain.common.interceptor;

// omitted

public class ServiceCallLoggingInterceptor implements MethodInterceptor {  //
↪ (1)

  private static final Logger logger = LoggerFactory
          .getLogger(ServiceCallLoggingInterceptor.class);

  @Override
  public Object invoke(MethodInvocation invocation) throws Throwable {  //␣
↪ (2)
      String methodName = invocation.getMethod().getName();
      String className = invocation.getMethod().getDeclaringClass()
              .getSimpleName();
      logger.info("[START SERVICE]{}.{}", className, methodName);  // (3)
      try {
          Object result = invocation.proceed();  // (4)
          logger.info("[COMPLETE SERVICE]{}.{}", className, methodName);  //
 (5)
          return result;  // (6)
      } catch (Throwable e) {
          logger.info("[SERVICE THROWS EXCEPTION]{}.{}", className,  // (7)
                  methodName);
          logger.info(Exception : {}, Message : {}, e.getClass().getName(),
                  e.getMessage()); // (8)
          throw e;  // (9)
      }
```

```
    }
}
```

項番	説明
(1)	メソッド呼び出しの前後で行う処理を記述するために、`MethodInterceptor` を実装する
(2)	`MethodInterceptor` に定義されている、`invoke` メソッドをオーバーライドする 引数の `org.aopalliance.intercept.MethodInvocation` オブジェクトから、呼び出すメソッドの名前等の情報を取得することができる
(3)	メソッド呼び出しを行う前に、呼び出すメソッド名をログ出力する
(4)	実際のメソッド呼び出しを行い、結果を取得する
(5)	メソッド呼び出しの結果例外が発生しなければ、操作成功のログメッセージを出力する
(6)	メソッド呼び出しの結果のオブジェクトを返す
(7)	メソッド呼び出しの結果例外が発生した場合、操作失敗のログメッセージを出力する
(8)	発生した例外クラスと例外メッセージを出力する 監査目的であることから、ログが冗長になることを避けるためスタックトレースは出力していない
(9)	発生した例外オブジェクトを投げる

ちなみに: 本アプリケーションの例を拡張することで、メソッド呼び出し時の引数などより詳細な内容を出力することも可能である。ただし、その場合はハッシュ化されていないパスワード等がログ出力される可能性があるため、マスキングを行う等の対策が必要となることに注意すること。

- アドバイスを`@Service`の付与されたクラスに対して適用するための設定を行う

  アドバイスに対するポイントカットの設定を以下に示す。

  secure-login-domain.xml

```
<!-- omitted -->

<bean id="serviceCallLoggingInterceptor"
    class="com.example.securelogin.domain.common.interceptor.
↪ServiceCallLoggingInterceptor" />  <!-- (1) -->
<aop:config>
    <aop:advisor advice-ref="serviceCallLoggingInterceptor"
        pointcut="@within(org.springframework.stereotype.Service)" />  <!--␣
↪(2) -->
</aop:config>

<!-- omitted -->
```

項番	説明
(1)	上記で作成したアドバイスクラス (`MethodInterceptor`の実装クラス) を Bean 定義する
(2)	`aop:advisor`タグの`advice-ref`属性にアドバイスが実装されている Bean を、`pointcut`属性にポイントカットをそれぞれ設定する `@within(org.springframework.stereotype.Service)`というポイントカットの定義により、`@Service`の付与されたクラスのメソッド呼び出しがアドバイスの対象となる

注釈: Spring の AOP は、自動的に作成されたプロキシクラスがメソッド呼び出しをハンドリングする、プロキシ方式を採用している。プロキシ方式の AOP の制限として、可視性が`public`以外のメソッドの呼び出しや、同一クラス内のメソッド呼び出しの際にはアドバイスが実行されない点に注意する必要がある。詳細は 公式ドキュメント を参照すること。

ログの出力結果を以下に示す。

```
date:2016-08-18 13:45:42   thread:tomcat-http--7   USER:demo   X-
↪Track:f514cc4159324ba28d8393f2c3062d89    level:INFO  logger:o.t.s.d.c.i.
↪ServiceCallLoggingInterceptor        message:[START␣
↪SERVICE]AccountSharedService.isInitialPassword
date:2016-08-18 13:45:42   thread:tomcat-http--7   USER:demo   X-
↪Track:f514cc4159324ba28d8393f2c3062d89    level:INFO  logger:o.t.s.d.c.i.
↪ServiceCallLoggingInterceptor        message:[START␣
↪SERVICE]PasswordHistorySharedService.findLatest
date:2016-08-18 13:45:42   thread:tomcat-http--7   USER:demo   X-
↪Track:f514cc4159324ba28d8393f2c3062d89    level:INFO  logger:o.t.s.d.c.i.
↪ServiceCallLoggingInterceptor        message:[COMPLETE␣
↪SERVICE]PasswordHistorySharedService.findLatest
date:2016-08-18 13:45:42   thread:tomcat-http--7   USER:demo   X-
↪Track:f514cc4159324ba28d8393f2c3062d89    level:INFO  logger:o.t.s.d.c.i.
↪ServiceCallLoggingInterceptor        message:[COMPLETE␣
↪SERVICE]AccountSharedService.isInitialPassword
```

例外発生時のログは以下のようになる。ログイン前に行われた処理に対するログであるため、ユーザ名が表示されていない。

```
date:2016-08-18 13:52:32   thread:tomcat-http--10  USER:   X-
↪Track:1a37a9a280014216a300b61e2f4bbb66    level:INFO  logger:o.t.s.d.c.i.
↪ServiceCallLoggingInterceptor        message:[SERVICE THROWS␣
↪EXCEPTION]AccountSharedService.findOne
date:2016-08-18 13:52:32   thread:tomcat-http--10  USER:   X-
↪Track:1a37a9a280014216a300b61e2f4bbb66    level:INFO  logger:o.t.s.d.c.i.
↪ServiceCallLoggingInterceptor        message:[SERVICE THROWS␣
↪EXCEPTION]UserDetailsService.loadUserByUsername
date:2016-08-18 13:52:32   thread:tomcat-http--10  USER:   X-
↪Track:1a37a9a280014216a300b61e2f4bbb66    level:INFO  logger:o.t.s.d.c.i.
↪ServiceCallLoggingInterceptor        message:user not found
```

9.10.4 おわりに

本章では、サンプルアプリケーションを題材としてセキュリティ対策の実装方法の例を説明した。

実際の開発においては、本アプリケーションにおける実装方法をそのまま利用できないケースも考えられるため、本章の内容を参考にしつつ要件に合わせてカスタマイズしたり別の方法を考えるようにしてほしい。

9.10.5 Appendix

Passay

Passay はパスワード入力チェック機能とパスワード生成機能を提供するライブラリである。Passay の API は以下の三つの主要コンポーネントで構成される。

- 検証規則

  パスワードが満たすべき条件の定義。パスワードの長さや含まれる文字種別等の一般的によく利用される規則についてはライブラリが提供するクラスを使用して容易に作成することができる。その他、必要

な規則を自分で定義することもできる。

- 検証器

  検証規則に基づいて実際にパスワードのチェックを行うコンポーネント。複数の検証規則を一つの検証器に設定することができる。

- 生成器

  与えられた文字種別に関する検証規則に適合するパスワードを生成するコンポーネント。

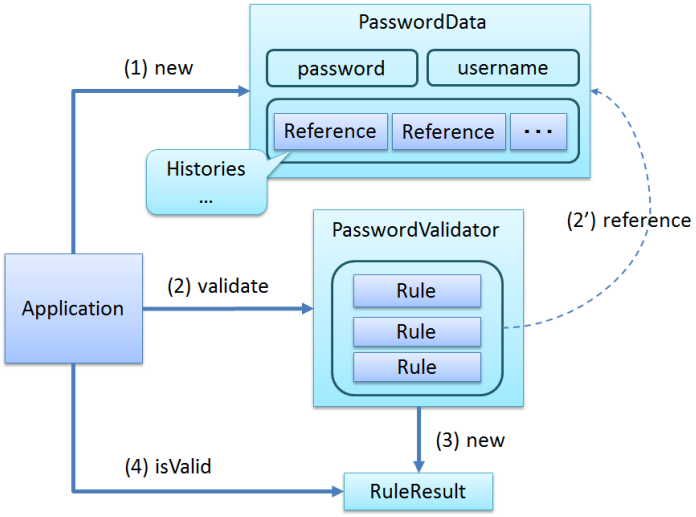
Passay の機能を使用する場合は、pom.xml に以下の定義を追加すること。

```
<dependencies>
  <dependency>
      <groupId>org.passay</groupId>
      <artifactId>passay</artifactId>
      <version>1.1.0</version>
  </dependency>
<dependencies>
```

パスワード入力チェック

Overview

Passay におけるパスワード入力チェックの流れの概略図を以下に示す。



項番	説明
(1)	org.passay.PasswordData のインスタンスを作成し、入力チェック対象のパスワードに関する情報を設定する。 PasswordData は、パスワード、ユーザ名に加え、過去に使用したパスワードのリスト等をプロパティとして持つことができる。 過去に使用したパスワード等は org.passay.PasswordData.Reference のインスタンスとして保持する。
(2)	検証規則に従い、検証器を用いて PasswordData に対する入力チェックを行う。 検証規則は org.passay.Rule の実装クラスのインスタンスとして作成する。検証器は org.passay.PasswordValidator のインスタンスであり、複数の検証規則をプロパティとして持つことができる。
(3)	検証器による入力チェックの結果として org.passay.RuleResult のインスタンスが作成される。
(4)	RuleResult からパスワード入力チェックの結果を boolean として得ることができる。また、検証器を使って RuleResult からエラーメッセージが取得できる。

Passay が提供している検証規則のクラスの一部を以下の表に示す。

クラス名	説明	主なプロパティ
`LengthRule`	パスワード長の最小値、最大値を規定するための検証規則のクラス	`minimuxLength` : パスワード長の最小値 (`int`)。コンストラクタまたは setter で設定。 `maximumLength` : パスワード長の最大値 (`int`)。コンストラクタまたは setter で設定。
`CharacterRule`	パスワードに含まれるべき文字種別と、その文字種別の最低文字数を規定するための検証規則のクラス	`characterData`: 文字種別 (`org.passay.CharacterData`)。コンストラクタで設定。 `numberOfCharacters` : 最低文字数 (`int`)。コンストラクタまたは setter で設定。
`CharacterCharacteristicsRule`	複数の `CharacterRule` のうち、いくつ以上の規則を満たす必要があるかを規定するための検証規則のクラス	`rules`: 文字種別に関する検証規則のリスト (`List<CharacterRule>`)。setter で設定。 `numberOfCharacteristics` : 満たすべき検証規則の数の最小値 (`int`)。setter で設定。
`HistoryRule`	パスワードが以前に使用したパスワードと一致していないことをチェックするための検証規則のクラス	なし
`UsernameRule`	パスワードがユーザ名を含まないことをチェックするための検証規則のクラス	`matchBackwards` : ユーザ名を逆にした文字列もチェックする (`boolean`)。コンストラクタまたは setter で設定。 `ignoreCase` : 大文字、小文字を区別しない (`boolean`)。コンストラクタまたは setter で設定。

この他にも、特定の文字を含む/含まないことのチェックや、正規表現によるチェックを行うための検証規則のクラス等が提供されている。詳細は http://www.passay.org/ を参照。

How to use

`PasswordValidator` のコンストラクタに `org.passay.Rule` のインスタンスのリストを渡すことによって、検証器を作成することができる。検証規則を設定した検証器を以下のように Bean として定義しておくことで DI が可能となる。尚、複数の検証規則を Bean 定義する場合、`@Inject` と`@Named` を併用することで Bean 名による DI を行うこと。

```
<!-- Password Rules. -->
<bean id="upperCaseRule" class="org.passay.CharacterRule"> <!-- (1) -->
    <constructor-arg name="data">
        <util:constant static-field="org.passay.EnglishCharacterData.UpperCase" />
↪<!-- (2) -->
    </constructor-arg>
    <constructor-arg name="num" value="1" /> <!-- (3) -->
</bean>
<bean id="lowerCaseRule" class="org.passay.CharacterRule"> <!-- (4) -->
    <constructor-arg name="data">
        <util:constant static-field="org.passay.EnglishCharacterData.LowerCase" />
    </constructor-arg>
    <constructor-arg name="num" value="1" />
</bean>
<bean id="digitRule" class="org.passay.CharacterRule"> <!-- (5) -->
    <constructor-arg name="data">
        <util:constant static-field="org.passay.EnglishCharacterData.Digit" />
    </constructor-arg>
    <constructor-arg name="num" value="1" />
</bean>

<!-- Password Validator. -->
<bean id="characterPasswordValidator" class="org.passay.PasswordValidator"> <!--␣
↪(6) -->
    <constructor-arg name="rules">
        <list>
            <ref bean="upperCaseRule" />
            <ref bean="lowerCaseRule" />
            <ref bean="digitRule" />
        </list>
    </constructor-arg>
</bean>
```

項番	説明
(1)	パスワードに含まれるべき文字種別と、その文字種別の最低文字数を規定するための検証規則の Bean 定義
(2)	文字種別を指定する。ここでは、`org.passay.EnglishCharacterData.UpperCase` を渡しているため、半角英大文字に関する検証規則となる。
(3)	文字数を指定する。ここでは」1」を渡しているため、半角英大文字を一文字以上含むことをチェックする検証規則となる。
(4)	(1)-(3) と同様だが、文字種別として `org.passay.EnglishCharacterData.LowerCase` を渡しているため、半角英小文字を一文字以上含むことをチェックする検証規則の Bean 定義となる。
(5)	(1)-(3) と同様だが、文字種別として `org.passay.EnglishCharacterData.Digit` を渡しているため、半角数字を一文字以上含むことをチェックする検証規則の Bean 定義となる。
(6)	検証器の Bean 定義。コンストラクタに検証規則のリストを渡す。

作成した検証器を使用してパスワード入力チェックを行う。

```
@Inject
PasswordValidator characterPasswordValidator;

// omitted

public void validatePassword(String password) {

    PasswordData pd = new PasswordData(password); // (1)
    RuleResult result = characterPasswordValidator.validate(pd); // (2)
```

```
    if (result.isValid()) { // (3)
        logger.info("Password is valid");
    } else {
        logger.error("Invalid password:");
        for (String msg : characterPasswordValidator.getMessages(result)) { // (4)
            logger.error(msg);
        }
    }

}
```

項番	説明
(1)	検証対象のパスワードを PasswordData のコンストラクタに渡し、インスタンスを作成する。
(2)	PasswordValidator の validate メソッドに PasswordData を引数として渡し、パスワード入力チェックを実行する。
(3)	RuleResult の isValid メソッドを使用して、パスワード入力チェックの結果を真理値で取得する。
(4)	PasswordValidator の getMessages メソッドに RuleResult を引数として渡し、エラーメッセージを取得する。

パスワード生成

Overview

Passay におけるパスワード生成機能では、パスワードの生成器と生成規則を用いる。生成器は org.passay.PasswordGenerator のインスタンスであり、生成規則は文字種別に関する検証規則 (org.passay.CharacterRule) のリストである。

生成器のメソッドに生成するパスワードの長さと生成規則を引数として与えることで、生成規則を満たしたパスワードが生成される。

How to use

生成規則に含まれる、文字種別に関する検証規則の作成方法は、パスワード入力チェック と同様である。生成規則と生成器を以下のように Bean として定義しておくことで DI が可能となる。

```
<!-- Password Rules. -->
<bean id="upperCaseRule" class="org.passay.CharacterRule"> <!-- (1) -->
    <constructor-arg name="data">
        <util:constant static-field="org.passay.EnglishCharacterData.UpperCase" />
↪<!-- (2) -->
    </constructor-arg>
    <constructor-arg name="num" value="1" /> <!-- (3) -->
</bean>
<bean id="lowerCaseRule" class="org.passay.CharacterRule"> <!-- (4) -->
    <constructor-arg name="data">
        <util:constant static-field="org.passay.EnglishCharacterData.LowerCase" />
    </constructor-arg>
    <constructor-arg name="num" value="1" />
</bean>
<bean id="digitRule" class="org.passay.CharacterRule"> <!-- (5) -->
    <constructor-arg name="data">
        <util:constant static-field="org.passay.EnglishCharacterData.Digit" />
    </constructor-arg>
    <constructor-arg name="num" value="1" />
</bean>

 <!-- Password Generator. -->
 <bean id="passwordGenerator" class="org.passay.PasswordGenerator" /> <!-- (6) -->
 <util:list id="passwordGenerationRules"> <!-- (7) -->
     <ref bean="upperCaseRule" />
     <ref bean="lowerCaseRule" />
     <ref bean="digitRule" />
 </util:list>
```

項番	説明
(1)	パスワードに含まれるべき文字種別と、その文字種別の最低文字数を規定するための検証規則の Bean 定義
(2)	文字種別を指定する。ここでは、`org.passay.EnglishCharacterData.UpperCase` を渡しているため、半角英大文字に関する検証規則となる。
(3)	文字数を指定する。ここでは」1」を渡しているため、半角英大文字を一文字以上含むことをチェックする検証規則となる。
(4)	(1)-(3) と同様だが、文字種別として `org.passay.EnglishCharacterData.LowerCase` を渡しているため、半角英小文字を一文字以上含むことをチェックする検証規則の Bean 定義となる。
(5)	(1)-(3) と同様だが、文字種別として `org.passay.EnglishCharacterData.Digit` を渡しているため、半角数字を一文字以上含むことをチェックする検証規則の Bean 定義となる。
(6)	生成器の Bean 定義
(7)	生成規則の Bean 定義。(1)-(5) で定義した、文字種別に関する検証規則のリストとして定義する。

作成した生成器と生成規則を使用してパスワード生成を行う。

```
@Inject
PasswordGenerator passwordGenerator;

@Resource(name = "passwordGenerationRules")
List<CharacterRule> passwordGenerationRules;

// omitted

public void generatePassword() {

    String password = passwordGenerator.generatePassword(10,␣
↪passwordGenerationRules); // (1)
```

```
}
```

項番	説明
(1)	`PasswordGenerator` の `generatePassword` メソッドに、生成するパスワードの長さと生成規則を引数として渡すと、生成規則を満たしたパスワードが生成される。

ちなみに： Bean 定義したコレクションを DI する際には、`@Inject` + `@Named` では期待した動作をしない。そのため、代わりに`@Resource` を使用して Bean 名で DI する。

第 10 章

単体テスト

10.1 単体テスト概要

10.1.1 はじめに

本章では、TERASOLUNA Server Framework (5.x) を使用したシステムにおける、JUnit を用いた単体テストについて提示する。

ここでの単体テストのスコープはレイヤまたはレイヤ間結合とし、テストに関するアクティビティのうち、テスト実装とテスト実施に関して解説する。なお、解説するテスト実装は参考例でありシステムの品質を保証するためのテスト方針については、別途検討いただきたい。

10.1.2 単体テストガイドラインが示すこと

本章では、以下の OSS ライブラリを使用したレイヤまたはレイヤ間のテスト実装方法について説明する。

単体テストで利用する OSS ライブラリ構成

テスティングフレームワーク

Java のテスティングフレームワークとして、JUnit を使用する。

アサーション

アサーションに使用するライブラリとして Hamcrest を使用する。JUnit4 が標準でサポートしているアサーションライブラリである。

モック化

テスト対象のメソッドが依存するクラスをモック化するためのライブラリとして Mockito を使用する。

DI コンテナ

テスト用の DI コンテナとして Spring Test の DI 機能を使用する。

MVC フレームワーク

テスト用の MVC フレームワークとして Spring MVC Test Framework を使用する。

トランザクション管理

テスト用のトランザクション管理として Spring Test のトランザクション管理機能を使用する。

データアクセス

テスト用のデータアクセスとして、Spring Test または DBUnit と Spring Test DBUnit を使用することを想定している。

- Spring Test
  - Spring Test は`@Sql` アノテーションや `JdbcTemplate` などを使用して SQL を発行する機能を提供している。
- DBUnit と Spring Test DBUnit
  - Spring Test DBUnit は、Spring Framework 上で DBUnit を利用する際の支援ライブラリのため、DBUnit と組み合わせて使用する。DBUnit の提供するデータベースのセットアップ、状態の検証などの機能をアノテーションベースで実装する機能を提供している。

単体テストで利用する OSS ライブラリのバージョン

単体テストで利用する OSS ライブラリの一覧を以下に示す。

なお、以下の OSS ライブラリはあくまで一例であり、実際は業務要件に合わせたライブラリを検討いただきたい。また、本章内のアプリケーション自体を動作させるために利用する OSS ライブラリ一覧については、利用する *OSS* のバージョンを参照されたい。

ちなみに： version 5.0.0.RELEASE より、Spring IO platform の`<dependencyManagement>`をインポートする構成を採用している。

Spring IO platform の`<dependencyManagement>`をインポートすることで、

- Spring Framework が提供しているライブラリ
- Spring Framework が依存している OSS ライブラリ
- Spring Framework と相性のよい OSS ライブラリ

への依存関係を解決しており、TERASOLUNA Server Framework (5.x) で使用する OSS のバージョンは、原則として、Spring IO platform の定義に準じている。

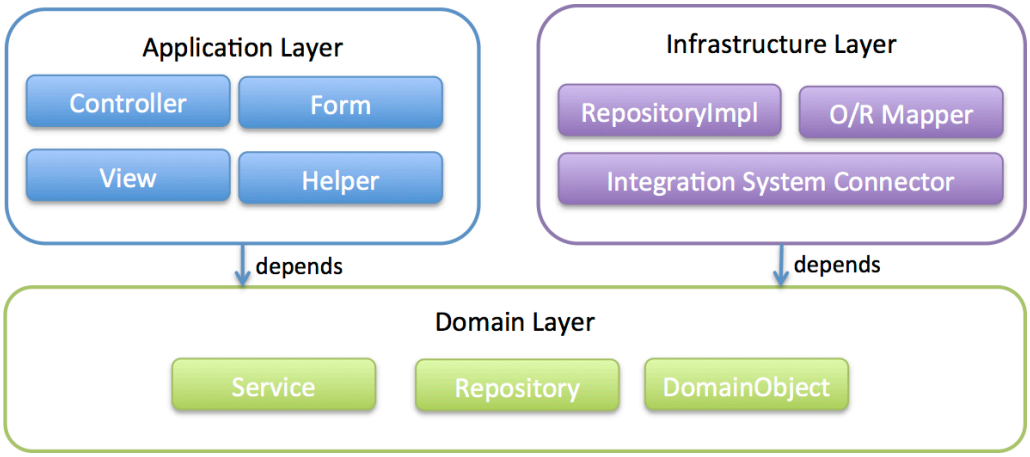
なお、version 5.4.2.RELEASE で指定している Spring IO platform のバージョンは、Brussels-SR17 である。

以下の OSS ライブラリの中で、特に Spring Test（MockMvc）、Mockito の使い方については、単体テストで利用する *OSS* ライブラリの使い方で詳細を説明する。

Type	GroupId	ArtifactId	Version	Spring IO platform
JUnit	junit	junit	4.12	*
Hamcrest	org.hamcrest	hamcrest-core	1.3	*
Hamcrest	org.hamcrest	hamcrest-library	1.3	*
Mockito	org.mockito	mockito-core	1.10.19	*
Spring Test	org.springframework	spring-test	4.3.23.RELEASE	*
DBUnit	org.dbunit	dbunit	2.5.4	
Spring Test DBUnit	com.github.springtestdbunit	spring-test-dbunit	1.3.0	

単体テストの実装

単体テストはアプリケーションのレイヤ化に沿った以下のレイヤ単位で実装している。レイヤまたはレイヤ間のテスト方法をレイヤごとのテスト実装で説明する。レイヤ単位に当てはめられない共通機能や、機能特有のテスト方法は、機能ごとのテスト実装で説明する。



10.1.3 対象読者

本章は、このドキュメントの対象読者に加えて以下の知識・経験があることを前提としている。

- JUnit を使用した単体テストを行ったことがある

10.1.4 単体テストの動作検証環境

本章は、以下の環境で動作検証をしている。他の環境で実施する際は、本章をベースに適宜読み替えること。

種別	名前
OS	Windows 10
JVM	Java 1.8.0_161
IDE	Spring Tool Suite 3.9.4.RELEASE (以降「STS」と呼ぶ)
Build Tool	Apache Maven 3.3.9 (以降「Maven」と呼ぶ)
RDBMS	PostgreSQL 9.6.10

10.2 単体テストの実装

10.2.1 テストの事前準備

本節では単体テストを実施するための事前準備として、利用する OSS ライブラリの設定方法とデータセットアップ方法およびテスト実装例で使用する設定ファイルについて説明する。

OSS ライブラリの設定

単体テストを実行するプロジェクト（domain プロジェクト、web プロジェクト）の POM ファイルにテストで利用する OSS ライブラリを設定する。

- pom.xml

```
<!-- == Begin Database == -->
<!-- <dependency> -->
<!-- <groupId>org.postgresql</groupId> -->
<!-- <artifactId>postgresql</artifactId> -->
<!-- <scope>test</scope> -->
<!-- </dependency> -->
<!-- == End Database == -->

<!-- == Begin Unit Test == -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
</dependency>
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-library</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
</dependency>
<!--  REMOVE THIS LINE IF YOU USE DBUnit
<dependency>
```

```
    <groupId>org.dbunit</groupId>
    <artifactId>dbunit</artifactId>
    <version>2.5.4</version>
    <scope>test</scope>
</dependency>
-->
<!--  REMOVE THIS LINE IF YOU USE Spring Test DBUnit
<dependency>
    <groupId>com.github.springtestdbunit</groupId>
    <artifactId>spring-test-dbunit</artifactId>
    <version>1.3.0</version>
    <scope>test</scope>
</dependency>
-->
<!-- == End Unit Test == -->
```

注釈: dbunit と spring-test-dbunit 以外の上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョン指定は不要である。また、hamcrest-core については、junit が依存関係を解決しているため、改めて定義する必要はない。

ちなみに：PostgreSQL ドライバの追加方法について

データアクセスを伴うテストで PostgreSQL のドライバを使用する場合は、POM ファイル内の PostgreSQL のドライバのコメントアウトを外すこと。なお、テストのために依存ライブラリが必要になる場合のスコープは test が適切である。

データベースのセットアップ

スキーマとテストデータのセットアップ（Spring Test 標準機能のみを利用したテストの場合）

単体テストで利用するデータベースのセットアップについて、以下の方法がある。

セットアップ方法	特徴	利用シーン
<jdbc:initialize-database>要素を使用する。	<jdbc:initialize-database>要素を定義した設定ファイルをテスト実施時に読み込みセットアップする。	インメモリデータベース (H2 Database) をセットアップする際に使用する。
initdb プロジェクトを使用する。	テスト実施と分離して事前に DB の初期化ができる。	テスト実施前にまとめてデータベースをセットアップする際に使用する。
@org.springframework.test.context.jdbc.Sql アノテーションを使用する。	@Sql アノテーションの引数で指定した SQL を発行する。@Sql アノテーションはメソッドレベル、クラスレベルで指定できる。メソッドレベルで指定した場合は指定したテストメソッドだけで、クラスレベルで指定した場合は@Sql アノテーションの指定がないすべてのテストメソッドで、実行前後に SQL を発行できる。	テストごとにテストデータをセットアップする際に使用する。

警告: <jdbc:initialize-database>タグに設定する SQL ファイルには、明示的に「COMMIT;」を記述すること。

単体テストで利用するスキーマのセットアップは、テストごとではなくテスト実施前にまとめて実施されることが想定される。そのため、本章ではテストと分離した initdb プロジェクトを使用してスキーマを作成することを前提に説明する。initdb プロジェクトについては、*initdb モジュールの構成*を参照されたい。

一方、テストデータのセットアップはテストごとに実施されることが想定される。そのため、本章ではテストクラスまたはテストメソッド毎に SQL を発行できる@Sql アノテーションを使用することを前提に説明する。

以下に、メソッドレベルに@Sql アノテーションを付与する場合のテストデータのセットアップ例を示す。

- MemberRepositoryTest.java

```
public class MemberRepositoryTest {

@Test
@Sql(scripts = "classpath:META-INF/sql/setupMemberLogin.sql" // (1)
     config = @SqlConfig(encoding = "utf-8")) // (2)
public void testUpdateMemberLogin() {
```

```
    // omitted
}
```

項番	説明
(1)	@Sql アノテーションに、テストに必要なデータを投入する SQL ファイルを指定する。
(2)	@SqlConfig アノテーションを使用して SQL ファイルのエンコードを指定する。

ちなみに: @Sql について

@Sql アノテーションの引数には、以下を指定できる。

- SQL ファイル（scripts または value）
- SQL ステートメント（statements）
- SQL 実行フェイズ（executionPhase）
- SQL 解析メタデータ (config に@SqlConfig アノテーションを指定)

また、@Sql アノテーションはデフォルトで有効になっている SqlScriptsTestExecutionListener によって実行される。詳細は、Executing SQL scripts declaratively with @Sql を参照されたい。

なお、@Sql アノテーションと@SqlConfig アノテーションによる構成は<jdbc:initialize-database>要素による構成の上位セットである。

注釈: @Sql の SQL ファイルパスの省略

@Sql アノテーションは、@ContextConfiguration アノテーション同様、SQL ファイルのパスを省略でき、省略した場合@Sql アノテーションが指定された場所に基づいて SQL ファイルの検索が行われる。

例えば、以下のようにデフォルトのパスにあるファイルがロードされる。

com.example.domain.repository.SampleRepositoryTest に指定した場合 → classpath:com/example/domain/repository/SampleRepositoryTest.sql

SampleRepositoryTest#testUpdate() に指定した場合 → classpath:com/example/domain/repository/SampleRepositoryTest.testUpdate.sql

なお、デフォルトのパスを検出できない場合は、java.lang.IllegalStateException が throw される。

注釈: @Sql の複数指定

@Sql には Java SE8 から追加された@Repeatable が付与されているため、Java SE8 以降を使う場合は同じ箇所に複数指定することができる。なお、Java SE7 以前を使用する場合は、@org.springframework.test.context.jdbc.SqlGroup を使用することで@Sql を複数指定することができる。

テストデータのセットアップ（Spring Test DBUnit を利用したテスト場合）

DBUnit とは、データベースに依存するクラスのテストを行うための JUnit 拡張フレームワークである。DBUnit と Spring Test DBUnit を使用して、テスト用データベースをセットアップする方法を説明する。

DBUnit は、表形式で記載したデータベース情報を Java オブジェクトとして抽象化して操作するための org.dbunit.dataset.IDataSet インタフェースを提供している。IDataSet インタフェースを使用することで、テストデータや期待結果データを定義したデータ定義ファイルを読み込むことができ、デフォルトでは Flat XML 形式のファイルが使用される。DBUnit は Flat XML 形式の他に、Excel 形式（.xlsx）や CSV 形式などに対応した IDataSet インタフェースの実装クラスを持つ。

Spring Test DBUnit ではデータ定義ファイルの読込機能を com.github.springtestdbunit.dataset.DataSetLoader インタフェースの実装クラスに委譲している。デフォルトでは XML 形式のデータ定義ファイルが読み込まれる。ファイル形式を変更したい場合は、変更したい形式に対応した IDataSet インタフェースの実装クラスを生成する DataSetLoader インタフェースの実装クラスを作成することで実現できる。

なお、Spring Test DBUnit を使用してデータのセットアップをする場合は、@DatabaseSetup アノテーションを使用することでテストコードにテストデータを定義したファイルを読み込ませることができる。@DatabaseSetup アノテーションはクラスレベル、メソッドレベルで指定でき、メソッドレベルに指定した場合は指定したメソッド、クラスレベルで指定した場合は各メソッドのテスト実行前に指定したファイルでデータのセットアップが行われる。

本章では、Excel 形式（.xlsx）のデータ定義ファイルを使用することを前提に説明する。Excel 形式に対応する DataSetLoader インタフェースの実装例を以下に示す。

- XlsDataSetLoader.java

```
public class XlsDataSetLoader extends AbstractDataSetLoader { // (1)

    @Override
    protected IDataSet createDataSet(Resource resource) throws IOException,␣
↪DataSetException {
        try (InputStream inputStream = resource.getInputStream()) {
            return new XlsDataSet(inputStream);
        }
    }
}
```

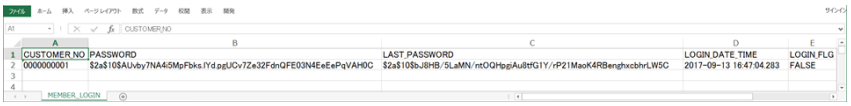
項番	説明
(1)	Spring Test DBUnit が提供する抽象基底クラスである `com.github.springtestdbunit.dataset.AbstractDataSetLoader` を利用して、Excel 形式のデータ定義ファイルの `XlsDataSetLoader` クラスを定義する。

- `MemberRepositoryDbunitTest.java`

```
// omitted
@DbUnitConfiguration(dataSetLoader = XlsDataSetLoader.class) // (1)
@DatabaseSetup("classpath:META-INF/dbunit/setup_MemberLogin.xlsx")
public class MemberRepositoryDbunitTest {
    // omitted
}
```

項番	説明
(1)	`@DbUnitConfiguration` アノテーションに `XlsDataSetLoader` クラスを指定することで、`@DatabaseSetup` アノテーションを使用した Excel 形式のデータ定義ファイル読込みができるようになる。

- Excel 形式のデータ定義ファイル（setup_MemberLogin.xlsx）



	A	B	C	D	E
1	CUSTOMER_NO	PASSWORD	LAST_PASSWORD	LOGIN_DATE_TIME	LOGIN_FLG
2	0000000001	$2a$10$AUvby7NA4i5MpFbks.IYd.pgUCv7Ze32FdnQFE03N4EeEePqVAH0C	$2a$10$bJ8HB/5LaMN/ntOQHpgiAu8tfG1Y/rP21MaoK4RBenghxcbhrLW5C	2017-09-13 16:47:04.283	FALSE
3					
4					

Excel 形式のデータ定義ファイルでは、各シートが各テーブルに対応する。シート名にはテーブル名、シートの一行目にはカラム名を設定する。 二行目以降にテーブルに挿入されるデータを記述する。

注釈: **CSV 形式のデータ定義ファイルを使用する場合**

DBUnit で CSV 形式のデータ定義ファイルを使用する場合は、`IDataSet` インタフェースの実装クラスとして `org.dbunit.dataset.csv.CsvDataSet.CsvDataSet` クラスを使用することで実現できる。

注釈: **DBUnit がデフォルトで読み込むファイル形式について**

DBUnit は、デフォルトで Flat XML 形式のデータ定義ファイルをサポートしている。

Spring Test DBUnit を使用した場合は、`@DbUnitConfiguration` に `dataSetLoader` を指定しなかった場合、Flat XML 形式のファイルに対応した `IDataSet` インタフェースの実装クラスである `org.dbunit.dataset.xml.FlatXmlDataSet` クラスが使用される。

Flat XML 形式のデータ定義ファイル例を以下に示す。

- `setup_MemberLogin.xml`

```
<!-- (1) -->
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
    <MEMBER_LOGIN CUSTOMER_NO="0000000001" PASSWORD="$2a$10$AUvby7NA4i5MpFbks.
↪lYd.pgUCv7Ze32FdnQFE03N4EeEePqVAH0C" LAST_PASSWORD="$2a$10$bJ8HB/5LaMN/
↪ntOQHpgiAu8tfG1Y/rP21MaoK4RBenghxcbhrLW5C" LOGIN_DATE_TIME="2017-09-13␣
↪16:47:04.283" LOGIN_FLG="FALSE" />
</dataset>
```

項番	説明
(1)	`dataset` 要素配下の各 XML 要素は、テーブルのレコードに対応しており、各 XML の要素名にテーブル名、属性名にカラム名、属性値に投入するデータを定義する。例では、`MEMBER_LOGIN` テーブルに値を定義している。

テスト実装例で使用する設定ファイル

Spring Test の DI 機能を使用することでテストで使用する Bean を定義した設定ファイルを読み込み、テスト時に使用することができる。詳細は *Spring Test の DI 機能*を参照されたい。

本章では、テストを行う際に必要な設定を `test-context.xml` に定義し、その設定ファイルをテスト時の共通設定としている。なお、`test-context.xml` は domain プロジェクトの `src/test/resources/test-context.xml` から`<import resource="classpath:META-INF/spring/projectName-domain.xml" />`を削除し、各層ごとにアプリケーションが保持する設定ファイル（`sample-infra.xml` など）と組み合わせて読み込む方針でテストを実装している。

注釈: 単体テストで利用する設定ファイルの作成単位

本章では上記のように設定ファイルを作成しているが、実際に設定ファイルを用意する際には、アーキテクトが業務要件を考慮して共通設定を定義し、それを元にテスト実装チームで必要な設定を追加するようにして対

応すること。

以下に本章の実装例で使用する設定ファイルを示す。

- test-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans https://www.
↪springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context https://www.springframework.
↪org/schema/context/spring-context.xsd">

    <context:property-placeholder
                    location="classpath*:/META-INF/spring/*.properties" />

    <!-- (1) -->
    <bean id="exceptionLogger" class="org.terasoluna.gfw.common.exception.
↪ExceptionLogger" />

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <!-- (2) -->
    <bean id="passwordEncoder" class="org.springframework.security.crypto.bcrypt.
↪BCryptPasswordEncoder" />

</beans>
```

項番	説明
(1)	テスト実施に必要な Bean を定義する。
(2)	ここでは、テスト例を実装するために passwordEncoder の Bean 定義を追加している。 Bean 定義については、業務に応じて適宜追加されたい。

10.2.2 レイヤごとのテスト実装

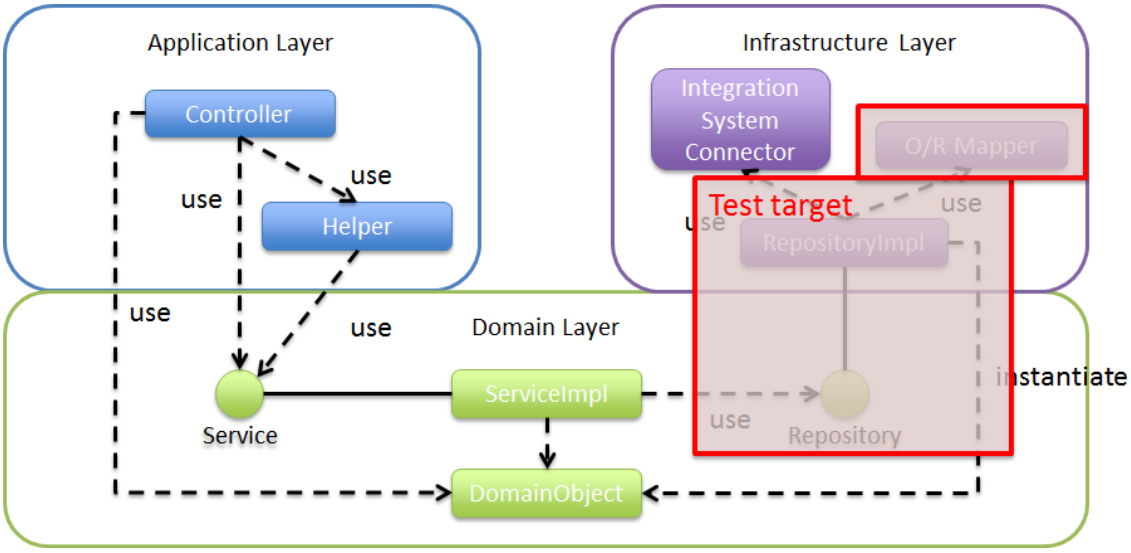
レイヤごとの単体テスト対象クラス、テスト方法およびその概要の一覧を以下に示す。

なお、本章で提示するテスト方法および実装はあくまで一例であり、実際はテスト方針に合わせたテスト方法および実装を検討いただきたい。

レイヤ	テスト方法	概要
インフラストラクチャ層	*Spring Test* 標準機能のみを利用したテスト	Spring Test の標準的な機能を使用してデータアクセスのテストを行う。
インフラストラクチャ層	*Spring Test DBUnit* を利用したテスト	DBUnit と Spring Test DBUnit の機能を使用してデータアクセスのテストを行う。
ドメイン層	依存クラスを利用したテスト	Spring Test の DI 機能を使用して `Service` をインジェクションし、インフラストラクチャ層と結合し `Service` のテストを行う。
ドメイン層	モックを利用したテスト	Mockito を使用して依存するクラスをモック化し `Service` のテストを行う。
アプリケーション層	*StandaloneSetup* を利用したテスト	Spring Test の DI 機能を使用して `Controller` をインジェクションし、ドメイン層、インフラストラクチャ層と結合した状態で `Controller` のテストを行う。
アプリケーション層	*WebAppContextSetup* を利用したテスト	Spring Test の MockMvc を使用して業務で作成した `spring-mvc.xml` と `applicationContext.xml` を適用し、`Controller` のテストを行う。
アプリケーション層	モックを利用したテスト	Mockito を使用して依存するクラスをモック化し `Controller` のテストを行う。
アプリケーション層	*Helper* の単体テスト	`Helper` のテストを行う。テスト方法の詳細は `Service` のテスト方法を参照。

インフラストラクチャ層の単体テスト

本節では、開発ガイドラインのインフラストラクチャ層の単体テストについて説明する。



インフラストラクチャ層では、Repository から MyBatis（O/R Mapper）を利用したデータアクセスのテストを行う。MyBatis3 の使用方法の詳細については、*MyBatis3 を使って Repository を実装*を参照されたい。

MyBatis により自動生成される `RepositoryImpl` は Spring の DI コンテナ上で実行されるため、テストには、本番同様の Bean 定義と、Spring の DI 機能を提供する Spring Test の `SpringJUnit4ClassRunner` を使用する。Spring Test の詳細は *Spring Test* を参照されたい。

テスト実行後のデータ検証方法には以下の 2 通りある。どちらを使用するかは別途業務要件に合わせて検討いただきたい。

- テスト実行後のデータベースの状態を SELECT 文を使用して取得し検証する。
- DBUnit と Spring Test DBUnit を使用して検証する。

本節では、SELECT 文を使用した検証方法として `JdbcTemplate` を使用した場合を例に説明する。`JdbcTemplate` とは Spring JDBC サポートのコアクラスである。JDBC API ではデータソースからコネクションの取得、`PreparedStatement` の作成、`ResultSet` の解析、コネクションの解放などを行う必要があるが、`JdbcTemplate` を使用することでこれらの処理の多くが隠蔽され、より簡単にデータアクセスを行うことができる。

注釈: アプリケーションのレイヤ化では、`Repository` インターフェイスはドメイン層の成果物であるが、インフラストラクチャ層の単体テスト対象として紹介している。`Service` とのインターフェイスが正しいことは、ドメイン層の単体テストでも確認することを推奨する。

Repository の単体テスト

本節では、以下の `Repository` の単体テスト実装方法を説明する。

テスト方法	説明
Spring Test 標準機能のみを利用したテスト	`JdbcTemplate` を使用してテスト結果の検証を行う。
Spring Test DBUnit を利用したテスト	DBUnit、Spring Test DBUnit の機能を使用してテスト結果の検証を行う。

ここでは、以下の成果物に対するテストを例に説明する。なお、Repository の実装の詳細は、*MyBatis3* を使って *Repository* を実装を参照されたい。

- `Repository` インタフェース（`MemberRepository`）の更新処理（`updateMemberLogin` メソッド）
- マッピングファイル（`MemberRepository.xml`）

以下に、テスト対象の実装例を示す。

- `MemberRepository.java`

```
public interface MemberRepository {

```

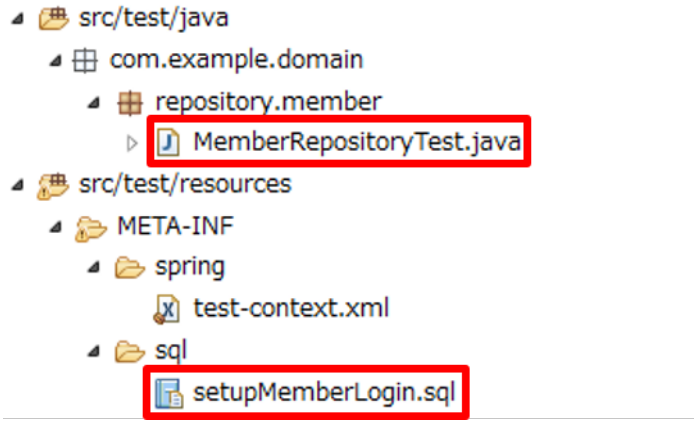
```
    int updateMemberLogin(Member member);
}
```

- MemberRepository.xml

```
<mapper namespace="com.example.domain.repository.member.MemberRepository">

  <update id="updateMemberLogin" parameterType="Member">
    UPDATE member_login SET
        last_password = password,
        password = #{memberLogin.password}
    WHERE
        customer_no = #{membershipNumber}
  </update>

</mapper>
```

Spring Test 標準機能のみを利用したテスト

Spring Test を使用した Repository の単体テストにおいて、作成するファイルを以下に示す。なお、データベースのセットアップ方法については*スキーマとテストデータのセットアップ（Spring Test 標準機能のみを利用したテストの場合）* を参照されたい。また、Spring Test を使用して単体テストを行う際に使用する設定ファイルは*テスト実装例で使用する設定ファイル*を参照されたい。



作成するファイル名	説明
MemberRepositoryTest.java	MemberRepository.java のテストクラス。
test-context.xml	Spring Test を使用して単体テストを行う際に必要な設定を補うための設定ファイル。
setupMemberLogin.sql	単体テストで利用するデータベースのデータをセットアップするための SQL ファイル。

注釈: 単体テストで利用する SQL ファイルの作成単位

ここでは、1 テストメソッドに 1 つの SQL を作成している。実際の作成単位については、テスト方針や内容に応じて適宜検討されたい。なお、`@Sql` に SQL ファイルパスを省略した場合、`@Sql` の指定場所に基づいて SQL ファイルの検索が行われる。詳細は、*@Sql の SQL ファイルパスの省略*を参照されたい。

Spring Test を使用する場合の `Repository` のテストクラス作成方法を説明する。

以下に、データアクセスを利用してテストするために使用する設定ファイルを示す。

- `sample-infra.xml`

```
<import resource="classpath:/META-INF/spring/sample-env.xml" />

<!-- define the SqlSessionFactory -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="configLocation" value="classpath:/META-INF/mybatis/mybatis-
↪config.xml" />
</bean>

<!-- scan for Mappers -->
<mybatis:scan base-package="com.example.domain.repository" />
```

- `sample-env.xml`

```
  <bean id="realDataSource" class="org.apache.commons.dbcp2.BasicDataSource"␣
↪destroy-method="close">
    <property name="driverClassName" value="org.postgresql.Driver" />
    <property name="url" value="jdbc:postgresql://localhost:5432/sample" />
    <property name="username" value="sample" />
    <property name="password" value="xxxx" />
    <property name="defaultAutoCommit" value="false" />
    <property name="maxTotal" value="96" />
    <property name="maxIdle" value="16" />
    <property name="minIdle" value="0" />
    <property name="maxWaitMillis" value="60000" />
  </bean>

  <bean id="dataSource" class="net.sf.log4jdbc.Log4jdbcProxyDataSource">
    <constructor-arg index="0" ref="realDataSource" />
  </bean>

  <bean id="transactionManager" class="org.springframework.jdbc.datasource.
↪DataSourceTransactionManager">
```

```
    <property name="dataSource" ref="dataSource" />
  </bean>

<bean id="dateFactory" class="org.terasoluna.gfw.common.date.jodatime.
↪DefaultJodaTimeDateFactory" />
```

以下に、Spring Test を使用した Repository のテスト作成方法について説明する。ここでは、テスト用のスキーマは作成済みであることを前提に、@Sql アノテーションを使用して MemberLogin テーブルをセットアップし、MemberLogin のパスワード「ABCDE」が新しいパスワード「FGHIJ」に更新されることを更新後の MemberLogin テーブルを取得して確認している。

- MemberRepositoryTest.java

```
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
        "classpath:META-INF/spring/sample-infra.xml",    //(1)
        "classpath:META-INF/spring/test-context.xml" }) //(1)
@Transactional // (2)
public class MemberRepositoryTest {

    @Inject
    MemberRepository target; // (3)

    @Inject
    JdbcTemplate jdbctemplate; // (4)

    @Test
    @Sql(scripts = "classpath:META-INF/sql/setupMemberLogin.sql", config =␣
↪@SqlConfig(encoding = "utf-8"))
    public void testUpdateMemberLogin() {

        // (5)
        // setup test data
        MemberLogin memberLogin = new MemberLogin();
        memberLogin.setPassword("FGHIJ");
        Member member = new Member();
        member.setMembershipNumber("0000000001");
        member.setMemberLogin(memberLogin);

        // (6)
        // run the test
        int updateCounts = target.updateMemberLogin(member);

        // (7)
        MemberLogin updateMemberLogin = getMemberLogin("0000000001");

        // (8)
        // assertion
        assertThat(updateCounts, is(1));
```

```
        assertThat(updateMemberLogin.getPassword(), is("FGHIJ"));
        assertThat(updateMemberLogin.getLastPassword(), is("ABCDE"));
    }

    private Member getMemberLogin(String customerNo) {

        MemberLogin memberLogin = (MemberLogin) jdbctemplate.queryForObject(
                "SELECT * FROM member_login WHERE customer_no=?",
                new Object[] {customerNo },
                new RowMapper<MemberLogin>() {

                    public MemberLogin mapRow(ResultSet rs,
                                int rowNum) throws SQLException {

                            MemberLogin mapMemberLogin = new MemberLogin();

                            mapMemberLogin.setPassword(rs.getString(
                                    "password"));
                            mapMemberLogin.setLastPassword(rs.getString(
                                    "last_password"));
                            mapMemberLogin.setLoginDateTime(rs.getDate(
                                    "login_date_time"));
                            mapMemberLogin.setLoginFlg(rs.getBoolean(
                                    "login_flg"));

                            return mapMemberLogin;
                    }
                });

        return memberLogin;
    }
```

項番	説明
(1)	`MemberRepository` クラスを動作させるために必要なアプリケーションが保持する `sample-infra.xml` と `test-context.xml` を読み込む。
(2)	`@Transactional` アノテーションを付与すると、テスト実行開始から終了まで一トランザクションとなり、デフォルトではテスト終了後にロールバックされる。クラスレベルでアノテーションを定義すると、全テストメソッドに対して`@Transactional` アノテーションが有効になる。
(3)	テスト対象である `MemberRepository` クラスをインジェクションする。
(4)	`JdbcTemplate` クラスをインジェクションする。
(5)	テスト対象メソッドを実行するためのテストデータを作成する。
(6)	テスト対象メソッドを実行する。
(7)	更新後のデータベースの情報を取得する。 `org.springframework.jdbc.core.RowMapper<T>`を使用することで、データベースから取得した `ResultSet` を特定の POJO クラスにマッピングすることができる。
(8)	更新件数、更新結果を確認する。

注釈: テスト時のトランザクションをロールバックさせない方法

`@Transactional` アノテーションをテストケースに指定した場合、デフォルトでテストメソッド実行後にロールバックされる。後続のテストでテストデータを使用するなどの目的でロールバックをさせたくない場合は、`@Transactional` アノテーションに加えて`@Rollback(false)` アノテーションまたは`@Commit` アノテーションを指定することで、テスト時のトランザクションをコミットすることができる。

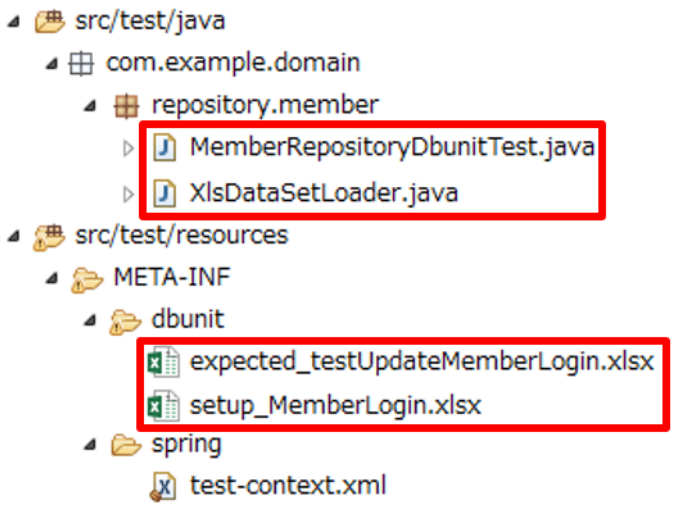
> **警告: Spring Framework 4.2 以降の@TransactionConfiguration について**
>
> Spring Framework 4.2 以降、クラスレベルで`@Rollback` または`@Commit` の設定が可能となった。これに伴い`@TransactionConfiguration` が非推奨となった。但し、Spring Framework 4.2 より前のバージョンでクラスレベルでロールバックをする場合は`@TransactionConfiguration(defaultRollback = true)` を設定すること。

Spring Test DBUnit を利用したテスト

データアクセスに DBUnit を使用する場合の `Repository` の単体テスト実装方法について説明する。なお、ここでは DBUnit のデータ定義ファイルに Excel 形式（.xlsx）のファイルを使用した場合を例に説明する。データ定義ファイルとデータベースのセットアップ方法については、テストデータのセットアップ（*Spring Test DBUnit* を利用したテスト場合）を参照されたい。

また、DBUnit に Spring Test DBUnit の機能を組み合わせて使用するには、`@TestExecutionListeners` アノテーションを使って、`com.github.springtestdbunit.TransactionDbUnitTestExecutionListener` を登録する必要がある。登録方法については、*TestExecutionListener* の登録を参照されたい。

DBUnit を利用した `Repository` の単体テストにおいて、作成するファイルを以下に示す。



作成するファイル名	説明
MemberRepositoryDbunitTest.java	MemberRepository.java のテストクラス (DBUnit と連携する場合)
XlsDataSetLoader.java	Excel 形式に対応する DataSetLoader インタフェースの実装クラス。実装方法については、テストデータのセットアップ（*Spring Test DBUnit* を利用したテスト場合）を参照されたい。
expected_testUpdateMemberLogin.xlsx	テストの期待結果検証用ファイル
setup_MemberLogin.xlsx	テストデータセットアップ用ファイル
test-context.xml	Spring Test を使用して単体テストを行う際に使用する設定ファイル。*Spring Test* 標準機能のみを利用したテストで作成した設定ファイルと同じものを使用する。

注釈: 単体テストで利用する Excel ファイルの作成単位

ここでは、1 テストメソッドにデータセットアップ用のファイルと期待結果検証用のファイルをそれぞれ 1 つずつ作成している。実際の作成単位については、テスト方針や内容に応じて適宜検討されたい。

DBUnit を使用する場合の Repository のテストクラス作成方法を説明する。

ここでは、テスト用のスキーマは作成済みであることを前提に、@DatabaseSetup アノテーションを使用して MemberLogin テーブルをセットアップし、MemberLogin のパスワード「ABCDE」が新しいパスワード「FGHIJ」に更新されることを@ExpectedDatabase アノテーションを使用して確認している。

以下に、Spring Test と DBUnit を使用した Repository のテスト作成方法を説明する。

- MemberRepositoryDbunitTest.java

```
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
        "classpath:META-INF/spring/sample-infra.xml",    // (1)
        "classpath:META-INF/spring/test-context.xml" }) // (1)
@TestExecutionListeners({
        DirtiesContextBeforeModesTestExecutionListener.class,
        DependencyInjectionTestExecutionListener.class,
        DirtiesContextTestExecutionListener.class,
        TransactionDbUnitTestExecutionListener.class})
@Transactional
```

```
@DbUnitConfiguration(dataSetLoader = XlsDataSetLoader.class)
public class MemberRepositoryDbunitTest {

    @Inject
    MemberRepository target;

    @Test
    @DatabaseSetup("classpath:META-INF/dbunit/setup_MemberLogin.xlsx")
    @ExpectedDatabase( // (2)
            value = "classpath:META-INF/dbunit/expected_testUpdateMemberLogin.xlsx
↪",
            assertionMode = DatabaseAssertionMode.NON_STRICT_UNORDERED)
    public void testUpdate() {

        // setup
        MemberLogin memberLogin = new MemberLogin();
        memberLogin.setPassword("FGHIJ");
        Member member = new Member();
        member.setMembershipNumber("0000000001");
        member.setMemberLogin(memberLogin);

        // run the test
        int updateCounts = target.updateMemberLogin(member);

        // assertion
        assertThat(updateCounts, is(1));
    }
}
```

項番	説明
(1)	`MemberRepository` クラスを動作させるために必要な設定ファイル（アプリケーションが保持する `sample-infra.xml` とそれを補う `test-context.xml`）を読み込む。
(2)	`@ExpectedDatabase` アノテーションにテストの期待結果検証用ファイルを指定することでテストメソッド実行後に DBUnit によってテーブルと期待結果データファイルが自動で比較検証される。 `@DatabaseSetup` アノテーション同様に、クラスレベルとメソッドレベルで付与できる。 ファイルフォーマットはテストセットアップ用データファイルと同じである。`assertionMode` 属性には、以下の値が設定可能である。 • `DEFAULT`（または指定なし）: 全てのテーブルとカラムの一致を比較する。 • `NON_STRICT` : 期待結果データファイルに存在しないテーブル、カラムが実際のデータベースに存在しても無視する。 • `NON_STRICT_UNORDERED` : `NON_STRICT` モードに加え、行の順序についても無視する。

警告: 外部キー制約のあるテーブル

外部キー制約のあるテーブルに対し、DBUnit を用いてデータベースを初期化すると、参照条件によってはエラーが発生するため、参照整合性を保つようにデータセットの順序を指定する必要があることに注意されたい。

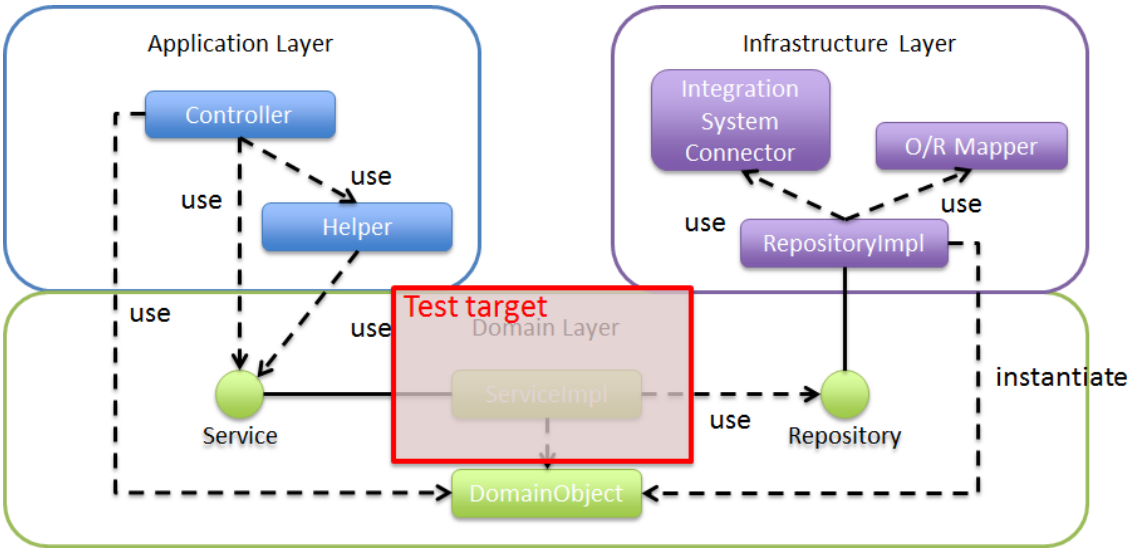
注釈: シーケンスの検証方法

シーケンスは、トランザクションをロールバックしても進んだ値は戻らないという特徴を持つ。そのため、シーケンスから採番したカラムを持つレコードを DBUnit で検証する場合、以下のいずれかの対応を行う必要がある。

- シーケンスから採番したカラムは検証対象外とする
- 明示的にシーケンスの初期化を行う SQL を実行し、テストの実施前に初期化する
- テスト実行時にシーケンスの値を確認し、確認した値を基準値として検証を行う

ドメイン層の単体テスト

本節では、開発ガイドラインのドメイン層の単体テストについて説明する。



ドメイン層では、`Service` の業務ロジックと`@Transactional` のテストを行う。`Service` をインジェクションし、インフラストラクチャ層を結合してテストを行う場合は、`Repository` のテスト実装方法と同様に Bean 定義と、Spring Test の `SpringJUnit4ClassRunner` を使用してテストを行う。Spring Test の詳細は *Spring Test* を参照されたい。

Service の単体テスト

本節では、以下の `Service` のテスト実装方法を説明する。

テスト方法	説明
依存クラスを利用したテスト	`Service` をインジェクションし、インフラストラクチャ層と結合してテストを行う。
モックを利用したテスト	`Service` の実装クラスが依存するクラスをすべてモック化してテストを行う。

ここでは、以下の成果物に対するテストを例に説明する。なお、Service の実装の詳細は、*Service の実装*を参照されたい。

- Service の実装クラス（`TicketReserveServiceImpl`）

以下に、テスト対象の実装例を示す。

- `TicketReserveServiceImpl.java`

```
@Service
@Transactional
public class TicketReserveServiceImpl implements TicketReserveService {

    @Inject
    ReservationRepository reservationRepository;

    @Override
    public TicketReserveDto registerReservation(Reservation reservation)
            throws BusinessException {

        List<ReserveFlight> reserveFlightList = reservation.getReserveFlightList();

        // repository access
        int reservationInsertCount = reservationRepository.insert(reservation);
        if (reservationInsertCount != 1) {
            throw new SystemException(LogMessages.E_AR_A0_L9002.getCode(),
                    LogMessages.E_AR_A0_L9002.getMessage(reservationInsertCount,␣
↪1));
        }

        String reserveNo = reservation.getReserveNo();

        Date paymentDate = reserveFlightList.get(0).getFlight().getDepartureDate();

        return new TicketReserveDto(reserveNo, paymentDate);
    }
}
```

以下に、テスト対象が使用するマッピングファイルを示す。

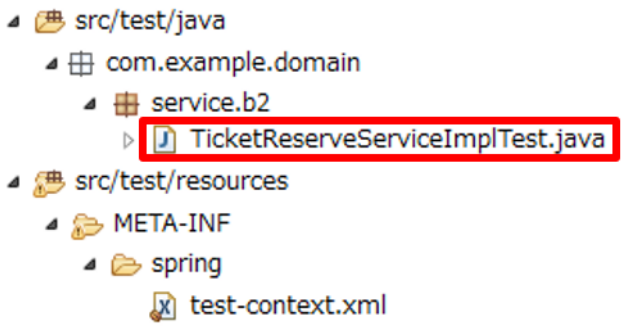
- ReservationRepository.xml

```
<mapper namespace="com.example.domain.repository.reservation.ReservationRepository
↪">

  <insert id="insert" parameterType="Reservation">
    <selectKey keyProperty="reserveNo" resultType="String" order="BEFORE">
      SELECT TO_CHAR(NEXTVAL('sq_reservation_1'), 'FM0999999999')
    </selectKey>
    INSERT INTO reservation
    (
        reserve_no,
        reserve_date,
        total_fare,
        rep_family_name,
        rep_given_name,
        rep_age,
        rep_gender,
        rep_tel,
        rep_mail,
        rep_customer_no
    )
```

```
    VALUES
    (
        #{reserveNo},
        #{reserveDate},
        #{totalFare},
        #{repFamilyName},
        #{repGivenName},
        #{repAge},
        #{repGender.code},
        #{repTel},
        #{repMail},
        NULLIF(#{repMember.membershipNumber}, '')
    )
  </insert>
```

依存クラスを利用したテスト

Service をインジェクションし、インフラストラクチャ層を結合して行う。Service のテストにおいて、作成するファイルを以下に示す。



作成するファイル名	説明
TicketReserveServiceImplTest.java	TicketReserveServiceImpl.java のテストクラス
test-context.xml	テスト実装例で使用する設定ファイルで定義した設定ファイルを使用する。

テスト対象の Service の実装クラスをインジェクションしてインフラストラクチャ層と結合してテストを行う場合のテスト作成方法を説明する。

以下に、テスト時に読み込む設定ファイルを示す。

- sample-domain.xml

```
<context:component-scan base-package="com.example.domain" />
<tx:annotation-driven />
```

```
<import resource="classpath:META-INF/spring/sample-infra.xml" />
<import resource="classpath:META-INF/spring/sample-codelist.xml" />

<bean id="resultMessagesLoggingInterceptor"
      class="org.terasoluna.gfw.common.exception.ResultMessagesLoggingInterceptor">
  <property name="exceptionLogger" ref="exceptionLogger" />
</bean>

<aop:config>
  <aop:advisor advice-ref="resultMessagesLoggingInterceptor"
    pointcut="@within(org.springframework.stereotype.Service)" />
</aop:config>
```

以下に、テスト実装例を示す。テスト対象の TicketReserveServiceImpl#registerReservation() メソッドを実行し、戻り値を確認している。なお、データベースの状態の検証方法は *Repository の単体テスト* を参照されたい。

- TicketReserveServiceImplTest.java

```
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
        "classpath:META-INF/spring/sample-domain.xml", // (1)
        "classpath:META-INF/spring/test-context.xml"}) // (1)
@Transactional
public class TicketReserveServiceImplTest {

    @Inject
    TicketReserveService target;

    @Inject
    private JdbcTemplate jdbcTemplate;

    @Test
    @Sql(statements = "ALTER SEQUENCE sq_reservation_1 RESTART WITH 1") // (2)
    public void testRegisterReservation() {

        // setup
        Reservation inputReservation = new Reservation();
        inputReservation.setTotalFare(39200);
        inputReservation.setReserveNo("0000000001");
        // omitted

        // run the test
        TicketReserveDto actTicketReserveDto = target.registerReservation(
                reservation);

        // assertion
        assertThat(actTicketReserveDto.getReserveNo(), is("0000000001"));
        // omitted
```

```
    }
}
```

項番	説明
(1)	TicketReserveServiceImpl クラスを動作させるために必要な設定ファイル（アプリケーションが保持する sample-domain.xml とそれを補う test-domain.xml）を読み込む。
(2)	@Sql の statements 属性を使用することで SQL 文を直接指定することもできる。ここではテストメソッド実行前にシーケンスの初期化を行っている。

警告：テスト時のトランザクション管理

テストケースに@Transactional アノテーションを付与すると、テスト実行開始から終了まで一トランザクションとなる。そのため、テストケースから@Transactional アノテーションを付与した Service クラスを呼び出した場合、テストケースからトランザクションが引き継がれる点に注意すること。例えば、トランザクションの伝播方法がデフォルト（REQUIRED）の場合、テストケースで開始したトランザクションでテスト対象の処理が行われ、コミット/ロールバックのタイミングもテスト終了時になる。トランザクションの伝播方法については「宣言型トランザクション管理」で必要となる情報を参照されたい。

モックを利用したテスト

Service の依存クラスをすべてモック化して行う Service の単体テストにおいて、作成するファイルを以下に示す。



作成するファイル名	説明
TicketReserveServiceImplMockTest.java	TicketReserveServiceImpl.java のテストクラス（モックを使用する場合）

テスト対象の Service の実装クラスが依存するクラスをモック化する場合のテスト作成方法を説明する。ここでは、ReservationRepository#insert() メソッドをモック化し、テスト対象の TicketReserveServiceImpl#registerReservation() メソッドでモック化したメソッドが呼び出されることとテスト対象の戻り値を確認している。

- TicketReserveServiceImplMockTest.java

```
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

public class TicketReserveServiceImplMockTest {

    @Rule // (1)
    public MockitoRule mockito = MockitoJUnit.rule();

    @Mock // (2)
    ReservationRepository reservationRepository;

    @InjectMocks // (3)
    private TicketReserveServiceImpl target;

    @Test
    public void testRegisterReservation() {

        // setup
        Reservation inputReservation = new Reservation();
        inputReservation.setTotalFare(39200);
        inputReservation.setReserveNo("0000000001");
        // omitted

        when(reservationRepository.insert(inputReservation)).thenReturn(1); // (4)

        // run the test
        TicketReserveDto ticketReserveDto = target.
↪registerReservation(inputReservation);

        // assertion
        verify(reservationRepository).insert(inputReservation); // (5)
        assertThat(ticketReserveDto.getReserveNo(), is("0000000001"));
        // omitted
    }
}
```

項番	説明
(1)	モックの初期化とインジェクションをアノテーションベースで行うための宣言。詳細はモックの生成を参照されたい。
(2)	`@Mock` アノテーションを付与することで、`TicketReserveServiceImpl` が依存している `MemberRepository` をモック化している。詳細はモックの生成を参照されたい。
(3)	`@InjectMocks` アノテーションを付与することで、自動的にモックオブジェクトが代入される。詳細はモックの生成を参照されたい。
(4)	`ReservationRepository` の `insert` メソッドについて、引数が `inputReservation` の場合、返り値として「1」 を返すように設定する。メソッドのモック化については、メソッドのモック化を参照されたい。
(5)	`ReservationRepository` の `insert` メソッドについて、引数に `inputReservation` が渡されて 1 回呼び出されたことを検証する。モック化したメソッドの検証については、モック化したメソッドの検証を参照されたい。

アプリケーション層の単体テスト
アプリケーション層の単体テスト対象

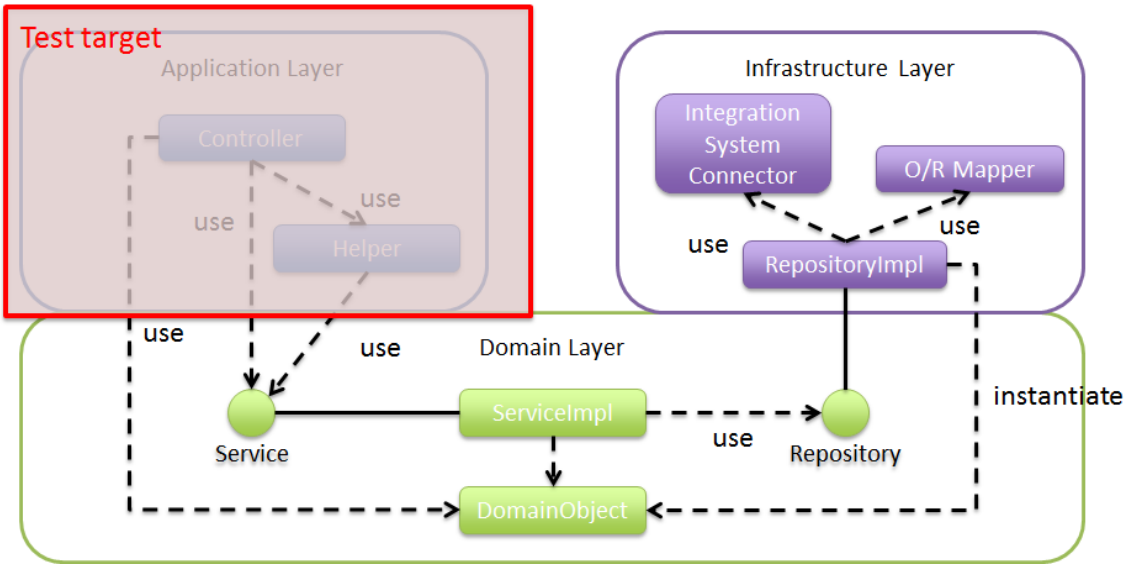
本節では、開発ガイドラインのアプリケーション層の単体テストについて説明する。

アプリケーション層では、`Controller` と `Helper` のロジックを確認するためのテストを行う。`Controller` については以下の項目を確認する。

- @RequestMapping(リクエストパス、HTTP メソッド、リクエストパラメータ)
- 返却される VIEW 名

`View` については、本来アプリケーション層に含まれるが、本ガイドラインでは対象外とする。

Spring Test は `Controller` クラスをテストするためのサポートクラス (`org.springframework.test.`



web.servlet.MockMvc など) を用意している。Controller は MockMVC を使用して疑似リクエストを送信してテストをするため、MockMVC を提供する Spring Test の SpringJUnit4ClassRunner を使用する。MockMvc は Controller に疑似リクエストを送信する仕組みを持ち、デプロイしたアプリケーションを模したテストを行うことができる。MockMVC の詳細は *MockMvc とは*を参照されたい。

注釈: **Form のバリデーションテスト**

Form のテストは、本来 Controller と組み合わせて実際の動作に近い形で行う必要があるが、Validation の全パターンを Controller と組み合わせるとテストの負担が大きくなる。そのため、単純な Validation の確認であれば、Controller と切り離して Form 単体で Validation の確認を行うこともできる。テスト方法はテスト対象の Form を使用して *Bean Validation で実装した Validator の単体テスト*を実施すればよい。

Controller の単体テスト

ここでは、以下の Controller の単体テスト実装方法を説明する。

テスト方法	説明
StandaloneSetup を利用したテスト	Spring Test が提供するデフォルトのコンテキストを使用し指定した設定ファイルを読み込むことでテストを行う。
WebAppContextSetup を利用したテスト	実際に使用する applicationContext.xml と spring-mvc.xml を使用してテストを行う。
モックを利用したテスト	Controller が依存するクラスをすべてモック化してテストを行う。

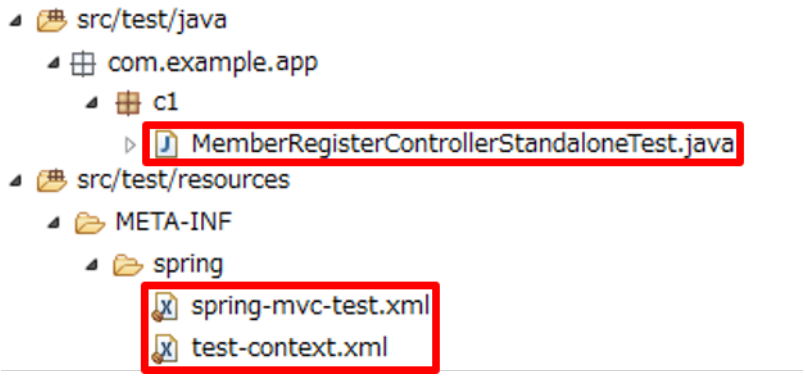
ここでは、以下の成果物に対するテストを例に説明する。Controller の実装の詳細は、*Controller の実装*を参照されたい。

- Controller クラス（TicketSearchController）
- Controller クラス（MemberRegisterController）

なお、インジェクションとモック化を組み合わせてテストを行いたい場合は、適宜以下に説明する実装方法を組み合わせて実装されたい。

StandaloneSetup を利用したテスト

Controller の依存クラスが利用できモック化する必要がない場合の Controller の単体テストにおいて、StandaloneSetup で作成するファイルを以下に示す。



作成するファイル名	説明
MemberRegisterControllerStandaloneTest.java	MemberRegisterController.java のテストクラス
spring-mvc-test.xml	アプリケーション層に依存するコンポーネントを読み込むための component-scan をテスト用に抽出した設定ファイル。
test-context.xml	Controller をドメイン層、インフラストラクチャ層と結合してテストを行う場合に使用する設定ファイル。

spring-mvc.xml を使ってテストをすることが望ましいが、Spring Test が作成したコンテキストと Spring MVC が作成したコンテキストが衝突しテスト実行ができないことがある。そのため対応策として、テストに必要な設定のみ抽出し、テスト用の設定ファイルを用意する。

以下に、必要な設定のみ抽出した設定ファイルを示す。

- spring-mvc-test.xml

```
<context:component-scan base-package="com.example.app" />
```

ServiceImpl クラスなどテスト対象の Controller クラスが依存するクラスをインジェクションする場合のテスト作成方法を説明する。なお、テストでデータアクセスする場合の検証方法は *Repository の単体テスト*を、呼び出すドメイン層のロジックを確認する方法は *Service の単体テスト*を参照されたい。

以下に、テスト対象となる Controller の実装例を示す。

- MemberRegisterController.java

```
@Controller
@RequestMapping("member/register")
@TransactionTokenCheck("member/register")
public class MemberRegisterController {

    @TransactionTokenCheck(type = TransactionTokenType.IN)
    @RequestMapping(method = RequestMethod.POST)
    public String register(@Validated MemberRegisterForm memberRegisterForm,
        BindingResult result, Model model, RedirectAttributes redirectAttributes) {

        if (result.hasErrors()) {
            throw new BadRequestException(result);
        }

        // omitted

        return "redirect:/member/register?complete";
    }
}
```

ここでは、テスト対象の MemberRegisterController クラスの register メソッドを呼び出し、リクエストマッピングと返却される VIEW およびリダイレクトされること（testRegisterConfirm01）、不正な入力値を送信したときに BadRequestException が throw されていること（testRegisterConfirm02）の確認を行う。

以下に、ServiceImpl クラスなどテスト対象の Controller クラスが依存するクラスをインジェクションする場合のテスト作成方法を説明する。なお、テストでデータアクセスする場合の検証方法は *Repository の単体テスト*を参照されたい。

- MemberRegisterControllerStandaloneTest.java

```
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.
↪*;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
        "classpath:META-INF/spring/applicationContext.xml", // (1)
```

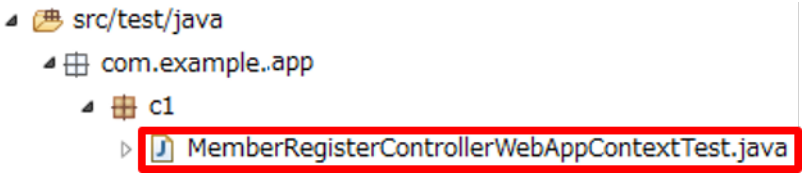
```
        "classpath:META-INF/spring/test-context.xml",       // (1)
        "classpath:META-INF/spring/spring-mvc-test.xml"})   // (1)
public class MemberRegisterControllerStandaloneTest {

    @Inject
    MemberRegisterController target;

    MockMvc mockMvc;

    @Before
    public void setUp() {

        // setup
        mockMvc = MockMvcBuilders.standaloneSetup(target).alwaysDo(log()).build();
↪// (2)
    }

    @Test
    public void testRegisterConfirm01() throws Exception {

        // setup and run the test
        mockMvc.perform(post("/member/register")
                    // omitted
                    .param("password", "testpassword")          // (3)
                    .param("reEnterPassword", "testpassword"))) // (3)
                    // assert
                    .andExpect(status().is(302))                                /
↪/ (4)
                    .andExpect(view().name("redirect:/member/register?complete")) /
↪/ (4)
                    .andExpect(model().hasNoErrors());                          /
↪/ (4)
    }

    @Test
    public void testRegisterConfirm02() throws Exception {

        try {
            // setup and run the test
            mockMvc.perform(post("/member/register")
                    // omitted
                    .param("password", "testpassword")
                    .param("reEnterPassword", "")) // (5)
                    // assert
                    .andExpect(status().is(400))
                    .andExpect(view().name("common/error/badRequest-error"))
                    .andReturn();

            fail("test failure!");
        } catch (Exception e) {

            // assert
            assertThat(e, is(instanceOf(NestedServletException.class)));         //
 (6)
```

```
            assertThat(e.getCause(), is(instanceOf(BadRequestException.class))); //
 (6)
        }
    }
}
```

項番	説明
(1)	MemberRegisterController クラスが依存する Service、Repository を動作させるために必要な設定ファイル（アプリケーションが保持する applicationContext.xml とそれを補う test-context.xml、 spring-mvc-test.xml）を読み込む。test-context.xml は、テスト実装例で使用する設定ファイルを使用している。
(2)	読み込んだ Bean 定義から生成した Controller を使用して、MockMvc をセットアップする。セットアップの詳細については *MockMvc のセットアップ*を参照されたい。
(3)	MemberRegisterController クラスの registerConfirm メソッドを呼び出すため、member/register に対して POST メソッドでリクエストを送信する。リクエストパラメータには Form の情報を設定する。リクエストデータの設定方法についてはリクエストデータの設定を、リクエスト送信の実装方法についてはリクエスト送信の実装を参照されたい。
(4)	perform メソッドから返却された ResultActions の andExpect メソッドで取得した MvcResult を使用して実行結果の妥当性を検証する。検証方法の詳細については実行結果検証の実装を参照されたい。
(5)	不正な入力値を送信する。
(6)	SystemExceptionResolver を有効にしていないため、例外ハンドリングされずに NestedServletException がサーブレットコンテナに通知される。NestedServletException の getCause メソッドにより取得された例外から、Controller で期待した例外が throw されていることを検証する。

WebAppContextSetup を利用したテスト

Controller の依存クラスが利用できモック化する必要がない場合の Controller の単体テストにおいて、WebAppContextSetup で作成するファイルを以下に示す。



作成するファイル名	説明
MemberRegisterControllerWebAppContextTest.java	MemberRegisterController.java のテストクラス

*StandaloneSetup を利用したテスト*の例では、パスへのリクエストや Controller が返す View 名などは確認できるが、TransactionTokenInterceptor や SystemExceptionResolver といった Spring に追加して利用する機能は適用されていないため、トランザクショントークンチェックが正しく設定されているか、エラーページへの遷移が正しいかを判断することはできない。そのような場合は、MockMvc を webAppContextSetup でセットアップすることにより、Spring に追加して利用する Interceptor や ExceptionResolver などをテスト時に自動で適用させることができる。

ここでは、*StandaloneSetup を利用したテスト*で説明したテストと、@TransactionTokenCheck アノテーション、SystemExceptionResolver が有効になった場合のテストとを比べた時の相違点について説明する。

以下に、テスト対象となる Controller の実装例を示す。

- MemberRegisterController.java

```
@Controller
@RequestMapping("member/register")
@TransactionTokenCheck("member/register")
public class MemberRegisterController {

    @TransactionTokenCheck(type = TransactionTokenType.BEGIN) // (1)
    @RequestMapping(method = RequestMethod.POST, params = "confirm")
    public String registerConfirm(@Validated MemberRegisterForm memberRegisterForm,
        BindingResult result, Model model) {

        // omitted

        return "C1/memberRegisterConfirm";
    }

    @TransactionTokenCheck(type = TransactionTokenType.IN) // (1)
    @RequestMapping(method = RequestMethod.POST)
    public String register(@Validated MemberRegisterForm memberRegisterForm,
        BindingResult result, Model model, RedirectAttributes redirectAttributes) {
```

```
        if (result.hasErrors()) {
            throw new BadRequestException(result); // (2)
        }

        // omitted

        return "redirect:/member/register?complete";
    }
}
```

項番	説明
(1)	@TransactionTokenCheck アノテーションを設定することで不正なリクエストを無効にする。トランザクショントークンチェックについては、トランザクショントークンチェックについてを参照されたい。
(2)	リクエスト時に検証エラーがある場合は改ざんとみなしてエラーを throw する。

初めに、@TransactionTokenCheck を有効にした場合におけるテスト作成方法の相違点について説明する。なお、テストでデータアクセスする場合の検証方法は *Repository* の単体テストを参照されたい。

- MemberRegisterControllerWebAppContextTest.java

```
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.
↪*;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextHierarchy({@ContextConfiguration(                                    // (1)
        "classpath:META-INF/spring/applicationContext.xml"),                 // (1)
        @ContextConfiguration("classpath:META-INF/spring/spring-mvc.xml")}) // (1)
@WebAppConfiguration                                                         // (1)
public class MemberRegisterControllerWebAppContextTest {

    @Inject
    WebApplicationContext webApplicationContext; // (2)

    MockMvc mockMvc;

    @Before
    public void setUp() {

        // setup
        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext) // (2)
```

```
                .alwaysDo(log()).build();
    }

    @Test
    public void testRegisterConfirm01() throws Exception {

        // setup and run the test
        MvcResult mvcResult = mockMvc.perform(post("/member/register") // (3)
                    .param("confirm", "")                              // (3)
                    // omitted
                    .param("password", "testpassword")                 // (3)
                    .param("reEnterPassword", "testpassword"))         // (3)
                    // assert
                    .andExpect(status().is(200))
                    .andExpect(view().name("C1/memberRegisterConfirm"))
                    .andReturn();

        TransactionToken actTransactionToken = (TransactionToken) mvcResult.
↪getRequest()
                .getAttribute(TransactionTokenInterceptor.NEXT_TOKEN_REQUEST_
↪ATTRIBUTE_NAME); // (4)

        MockHttpSession mockSession = (MockHttpSession) mvcResult.getRequest().
↪getSession();  // (5)

        // setup and run the test
        mockMvc.perform(post("/member/register")               // (6)
                    // omitted
                    .param("password", "testpassword")         // (6)
                    .param("reEnterPassword", "testpassword") // (6)
                    .param(TransactionTokenInterceptor.TOKEN_REQUEST_PARAMETER,
                            actTransactionToken.getTokenString()) // (6)
                    .session(mockSession)) // (6)
                    // assert
                    .andExpect(status().is(302))
↪// (7)
                    .andExpect(view().name("redirect:/member/register?complete"));
↪// (7)
    }
}
```

項番	説明
(1)	業務でカスタムした Interceptor や ExceptionResolver などを動作させるために spring-mvc.xml を読み込む。

Continued on next page

表 10.3 – continued from previous page

項番	説明
(2)	読み込んだ Bean 定義から生成した Web アプリケーションコンテキストを使用して、MockMvc をセットアップする。
(3)	トランザクショントークンを生成するために、@TransactionTokenCheck(type = TransactionTokenType.BEGIN) が設定されたメソッドに対してリクエストを送信する。
(4)	BEGIN したリクエスト（registerConfirm メソッド）から IN のリクエスト（register メソッド）にトランザクショントークンを引き継ぐため、リクエスト属性からトランザクショントークンを取得する。
(5)	サーバ側は発行したトランザクショントークンをセッションに保持するため、次のリクエストでも同じセッションを参照する必要があるが、MockMvc では 1 リクエストごとに新規セッションが使われてしまうため、明示的に同じセッションを使用するよう指定する。
(6)	再度、リクエストパス（member/register）に対して POST メソッドでリクエストを送信する。リクエストパラメータには Form の情報、(4) で取得したトランザクショントークンを設定し、セッションには (5) で取得したセッションを設定する。
(7)	トランザクショントークンチェックの設定が正しいことを確認するために、トークンチェックエラーになっていないことを検証する。

次に、SystemExceptionResolver を有効にした場合におけるテスト作成方法の相違点を説明する。

以下に、SystemExceptionResolver の定義例を示す。

- spring-mvc.xml

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">
  <property name="order" value="3" />
  <property name="exceptionMappings">
    <map>
      <entry key="InvalidTransactionTokenException" value="common/error/token-error
↪" />
```

```
      <entry key="BadRequestException" value="common/error/badRequest-error" />
      <entry key="Exception" value="common/error/system-error" />
    </map>
  </property>
  <property name="statusCodes">
    <map>
      <entry key="common/error/token-error" value="409" />
      <entry key="common/error/badRequest-error" value="400" />
    </map>
  </property>
  <property name="excludedExceptions">
      <array>
          <value>org.springframework.web.util.NestedServletException</value>
      </array>
  </property>
  <property name="defaultStatusCode" value="500" />
  <property name="exceptionCodeResolver" ref="exceptionCodeResolver" />
  <property name="preventResponseCaching" value="true" />
</bean>
```

以下に、テスト作成方法の相違点について説明する。

- MemberRegisterControllerWebAppContextTest.java

```
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.
↪*;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextHierarchy({@ContextConfiguration(
        "classpath:META-INF/spring/applicationContext.xml"),
        @ContextConfiguration("classpath:META-INF/spring/spring-mvc.xml")})
@WebAppConfiguration
public class MemberRegisterControllerWebAppContextTest {

    @Inject
    WebApplicationContext webApplicationContext;

    MockMvc mockMvc;

    @Before
    public void setUp() {

        // setup
        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext)
                .alwaysDo(log()).build();
    }

    @Test
    public void testRegisterConfirm02() throws Exception {
```

```
        // omitted

        // setup and run the test
        mvcResult = mockMvc.perform(post("/member/register")
                    .param("password", "testpassword")
                    .param("reEnterPassword", "") // (1)
                    .param(TransactionTokenInterceptor.TOKEN_REQUEST_PARAMETER,
                            actTransactionToken.getTokenString()) // (2)
                    .session(mockSession)) // (2)
                    // assert
                    .andExpect(status().is(400))                                  // (3)
                    .andExpect(view().name("common/error/badRequest-error")) // (3)
                    .andReturn();

        // assert
        Exception exception = mvcResult.getResolvedException();                   /
↪/ (4)
        assertThat(exception, is(instanceOf(BadRequestException.class)));         /
↪/ (4)
        assertThat(exception.getMessage(), is("不正リクエスト（パラメータ改竄）")); // (4)
    }
}
```

項番	説明
(1)	Form の情報を不正な値にすることで、register メソッドの内でエラーを throw させている。
(2)	前述と同様に、生成したトランザクショントークン情報を設定する。
(3)	ここでは SystemExceptionResolver が有効になっているため、定義したエラーのステータスコード、エラーページの遷移先が正しく設定されていることを検証する。
(4)	SystemExceptionResolver で例外ハンドリングされたエラーから、期待したエラーが throw されていることを検証する。

注釈: Session を利用する場合

Controller クラスが Session を利用している場合は org.springframework.mock.web.MockHttpSession を使ってテストを行う。

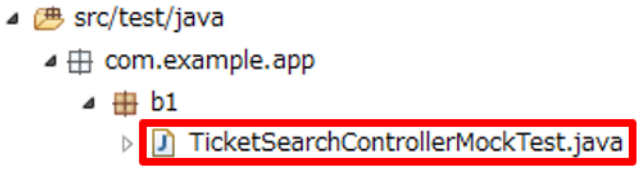
- MockHttpSession を利用したテストメソッドの例

```
public class SessionControllerTest {

    // (1)
    MockHttpSession mockSession = new MockHttpSession();

    // omitted

    @Test
    public void testSession() throws Exception {
        String formName = "todoForm";

        TodoForm form = new TodoForm();
        String todoId = "1111";
        String todoTitle = "test";

        form.setTodoId(todoId);
        form.setTodoTitle(todoTitle);

        // (2)
        mockSession.setAttribute(formName, form);

        // (3)
        ResultActions results = mockMvc.perform(post("/todo/operation")
            .param("create", "create")
            .param("todoId", todoId)
            .param("todoTitle", todoTitle)
            .session(mockSession));

        // (4)
        results.andExpect(request().sessionAttribute(formName, isA(TodoForm.
↪class)));

        // omitted

        // (5)
        results = mockMvc.perform(get("/todo/create").param("redo", "redo"));
        results.andExpect(request().sessionAttribute(formName, isA(TodoForm.
↪class)));

        // omitted
    }
}
```

項番	説明
(1)	セッションのモックオブジェクトを生成する。クラスの詳細については、MockHttpSession の Javadoc を参照されたい。
(2)	生成したセッションのモックオブジェクトに、格納したいオブジェクトをセットする。
(3)	`MockMvcRequestBuilders` の `post` メソッドでリクエストのモックを生成し、生成したリクエストに `session` メソッドでセッションのモックを登録する。
(4)	(2) でセットしたオブジェクトが、セッションスコープに格納されていることを確認する。
(5)	再度リクエストを発行し、セッションスコープに格納したオブジェクトが保持されているか確認する。

モックを利用したテスト

`Controller` の依存クラスをモック化する必要がある場合の `Controller` の単体テストにおいて、作成するファイルを以下に示す。



作成するファイル名	説明
`TicketSearchControllerMockTest.java`	`TicketSearchController.java` のテストクラス

テスト対象の Controller クラスが依存するクラスを、モック化する場合のテスト作成方法を説明する。

以下に、テスト対象となる Controller の実装例を示す。

- TicketSearchController.java

```
@Controller
@RequestMapping("ticket/search")
public class TicketSearchController {

    @Inject
    TicketSearchHelper ticketSearchHelper;

    @RequestMapping(method = RequestMethod.GET, params = "form")
    public String searchForm(Model model) {

        model.addAttribute(ticketSearchHelper.createDefaultTicketSearchForm());

        model.addAttribute(ticketSearchHelper.createFlightSearchOutputDto());

        model.addAttribute("isInitialSearchUnnecessary", true);

        return "B1/flightSearch";
    }
}
```

以下に、Controller のテスト実装例を示す。

- TicketSearchControllerMockTest.java

```
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.
↪*;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

public class TicketSearchControllerMockTest {

    @Rule // (1)
    public MockitoRule mockito = MockitoJUnit.rule();

    @InjectMocks // (2)
    TicketSearchController target;

    @Mock // (3)
    TicketSearchHelper ticketSearchHelper;

    MockMvc mockMvc;
```

```
    @Before
    public void setUp() {

        // setup
        TicketSearchForm ticketSearchForm = new TicketSearchForm();
        ticketSearchForm.setFlightType(FlightType.RT);
        ticketSearchForm.setDepAirportCd("HND");
        // omitted

        when(ticketSearchHelper.createDefaultTicketSearchForm()).
↪thenReturn(ticketSearchForm); // (4)

        mockMvc = MockMvcBuilders.standaloneSetup(target).alwaysDo(log()).build();
    }

    @Test
    public void testSearchForm() throws Exception {

        // setup and run the test
        MvcResult mvcResult = mockMvc.perform(get("/ticket/search").param("form", "
↪"))
                    // assert
                    .andExpect(status().is(200))
                    .andExpect(view().name("B1/flightSearch"))
                    .andReturn();

        // assert
        verify(ticketSearchHelper).createDefaultTicketSearchForm(); // (5)

        // omitted
    }
}
```

項番	説明
(1)	モックの初期化とインジェクションをアノテーションベースで行うための宣言。詳細はモックの生成を参照されたい。
(2)	`@Mock` アノテーションを付与することで、`TicketSearchController` が依存している `TicketSearchHelper` をモック化している。詳細はモックの生成を参照されたい。
(3)	`@InjectMocks` アノテーションを付与することで、自動的にモックオブジェクトが代入される。詳細はモックの生成を参照されたい。
(4)	すべてのテストメソッドにおいて、`ticketSearchHelper` の `createDefaultTicketSearchForm` メソッドの返り値として `createMockForm` メソッドの返り値を設定する。メソッドのモック化については、メソッドのモック化を参照されたい。
(5)	`ticketSearchHelper` の `createDefaultTicketSearchForm` メソッドについて 1 回呼び出されたことを検証する。モック化したメソッドの検証については、モック化したメソッドの検証を参照されたい。

Helper の単体テスト

`Helper` の単体テストは、`Service` と同様の実装でテストすることができる。実装方法については、*Service の単体テスト*を参照されたい。

10.2.3 機能ごとのテスト実装

本節では、レイヤ単位に当てはめられない機能のテスト方法ついて説明する。

入力チェックの単体テスト

ここでは、以下の入力チェックの単体テスト実装方法を説明する。

Validation の種類	説明
Bean Validation	Hibernate validator を使用して実装した Validator のテスト
Bean Validation	Spring の DI コンテナを使用して実装した Validator のテスト
Spring Validation	Spring Validation を使用して実装した Validator のテスト

Validator の単体テストは本来 Controller のテストとして行うが、その場合は試験パターンが多くなるため、テストの実施コストを考慮し Controller と切り分けて Validator 単体としてテストを行うこともできる。ここでは、Validator 単体としてのテスト作成方法を説明する。

本節では、Bean Validation を使用している場合と Spring Validation を使用している場合のそれぞれについて実装方法を説明する。

Bean Validation で実装した Validator の単体テスト

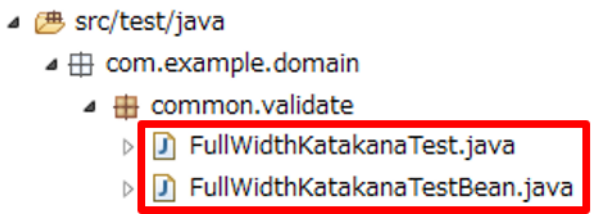
Bean Validation のテスト行う場合、アプリケーションサーバからライブラリが提供されないため、必要な依存ライブラリ追加する必要がある。追加方法については、依存ライブラリの追加を参照されたい。なお、Hibernate Validator が用意する入力チェック機能についてはテストスコープ外とする。

ここでは、以下の 2 通りの Bean Validation のテスト方法について説明する。

- Hibernate validator を使用した Bean Validation
- Spring の DI コンテナを使用した Bean Validation

Hibernate validator を使用した Bean Validation のテスト

Hibernate validator を使用した Bean Validation の単体テストにおいて、作成するファイルを以下に示す。



作成するファイル名	説明
FullWidthKatakanaTest.java	@FullWidthKatakana アノテーションのテストクラス
FullWidthKatakanaTestBean.java	@FullWidthKatakana アノテーションをフィールドに付与した Bean クラス

ここでは、テスト対象の@FullWidthKatakana を使用した Bean クラス（FullWidthKatakanaTestBean）を作成し、javax.validation.ValidatorFactory から生成した javax.validation.Validator の実装クラスによりバリデーションチェックを行っている。

以下に、@FullWidthKatakana アノテーションをフィールドに付与した Bean クラスの作成例を示す。

- FullWidthKatakanaTestBean.java

```
public class FullWidthKatakanaTestBean {

    @FullWidthKatakana
    private String testString;

    public FullWidthKatakanaTestBean() {
        // constructor
    }

    public String getTestString() {
        return testString;
    }

    public void setTestString(String testString) {
        this.testString = testString;
    }

}
```

- FullWidthKatakanaTest.java

```
public class FullWidthKatakanaTest {

    private static Validator validator;

    @BeforeClass
    public static void setUpBeforeClass() {

        // setup
        ValidatorFactory validatorFactory = Validation.
↪buildDefaultValidatorFactory();

        // (1)
        validator = validatorFactory.getValidator();
    }

    @Test
    public void testFullWidthKatakana() {

        // setup
        FullWidthKatakanaTestBean form = new FullWidthKatakanaTestBean();
        form.setTestString("テスト");

        // run the test
        Set<ConstraintViolation<FullWidthKatakanaTestBean>> violations = validator.
↪validate(form); // (2)

        // assert
        assertThat(violations, is(empty())); // (3)
```

```
    }
}
```

項番	説明
(1)	getValidator メソッドにより、Validator を取得する。 Validator を取得することで、validate メソッドを使った入力チェックが可能となる。
(2)	validate メソッドを使い、入力チェックを行う。 validate メソッドを実行することで、入力チェックエラーの数だけ ConstrainViolation の Set が返ってくる。validate メソッドの引数には FullWidthKatakanaBean クラスのオブジェクトを指定する。
(3)	(2) で取得した Set から、エラーが発生したかどうかを確認する。 今回はエラーがないため、空の Set が返ってくる。

注釈: バリデーショングループを使用したテスト

バリデーショングループを設定している場合、入力チェックを行なう際の validate メソッド引数に、グループを示す任意の java.lang.Class オブジェクトを指定することで、指定したグループの Validator のみ適用して実行できる。バリデーショングループについては、バリデーションのグループ化を参照されたい。

以下に、バリデーショングループを使用した Form 例を示す。

- テスト対象の FullWidthKatakanaTestBean.java

```
public class FullWidthKatakanaTestBean {

    public interface Search {};
    public interface Register {};

    // (1)
    @Size(min = 5, max = 10, groups = Search.class)
    @FullWidthKatakana(groups = Register.class)
    @NotNull
    private String testString;

    public FullWidthKatakanaTestBean() {
        // constructor
    }
```

```
    public String getTestString() {
        return testString;
    }

    public void setTestString(String testString) {
        this.testString = testString;
    }

}
```

項番	説明
(1)	フィールドに設定する Validator をグループ化している。

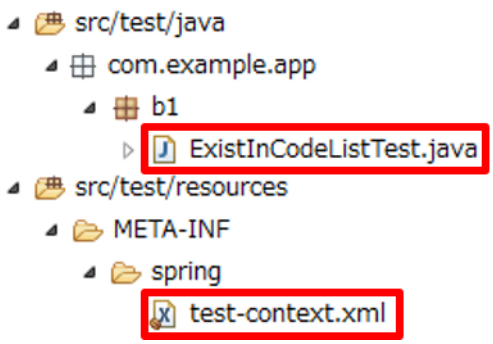
- FullWidthKatakanaTest.java

```
public class FullWidthKatakanaTest {

    // omitted

    @Test
    public void testFullWidthKatakana() {

        // setup
        FullWidthKatakanaTestBean form = new FullWidthKatakanaTestBean();
        form.setTestString("テスト");

        // run the test
        // (1)
        Set<ConstraintViolation<FullWidthKatakanaTestBean>> violations =
                validator.validate(form, Default.class, Search.class);

        // assert
        assertThat(violations, is(empty())); // (2)
    }
}
```

項番	説明
(1)	validate メソッドの引数に、java.lang.Class オブジェクトを追加することで、設定したバリデーショングループに対して入力チェックを実行できる。また、java.lang.Class オブジェクトは例のように複数指定することもできる。
(2)	エラーが発生したかどうかを確認する。

Spring の DI コンテナを使用した Bean Validation のテスト

Spring の DI コンテナを使用した Bean Validation の単体テストにおいて、作成するファイルを以下に示す。



作成するファイル名	説明
ExistInCodeListTest.java	Spring の DI コンテナを使用した Bean Validation のテストクラス
test-context.xml	Spring Test を使用して単体テストを行う際に必要な設定を補うための設定ファイル。

Spring の DI コンテナを利用した Bean Validation は、org.springframework.validation.beanvalidation.LocalValidatorFactoryBean から Validator オブジェクトを生成することでテストすることができる。

ここでは、Spring の DI コンテナを利用した入力チェックとして@ExistInCodeList を例にテストの実装方法を説明する。@ExistInCodeList についての詳細はコードリストを用いたコード値の入力チェックを参照されたい。

テストで使用する設定ファイルに、Validator オブジェクトを生成するための LocalValidatorFactoryBean を Bean 定義する。

- test-context.xml

```
<!-- (1) -->
<bean id="validator" class="org.springframework.validation.beanvalidation.
↪LocalValidatorFactoryBean" />
```

項番	説明
(1)	@ExistInCodeList で DI コンテナからコードリスト Bean を取得するため、test-context.xml で Bean 定義した LocalValidatorFactoryBean から生成した Validator を使う必要がある。

以下に、@ExistInCodeList が使われている Form クラスの実装例を示す。

- TicketSearchForm.java

```
public class TicketSearchForm implements Serializable {

    @NotNull
    @ExistInCodeList(codeListId = "CL_AIRPORT") // (1)
    private String depAirportCd;

    // omitted
}
```

項番	説明
(1)	depAirportCd フィールドに対して、コードリストに存在する値かどうか検証する。

以下に、@ExistInCodeList のテストクラス作成方法を説明する。ここでは、sample-codelist.xml に定義したコードリスト（CL_AIRPORT）に定義していない値を設定し、インジェクションした javax.validation.Validator の実装クラスによりバリデーションチェックエラーになることを確認している。

- ExistInCodeListTest.java

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
        "classpath:META-INF/spring/sample-infra.xml",
        "classpath:META-INF/spring/sample-codelist.xml",
        "classpath:META-INF/spring/test-context.xml" })
public class ExistInCodeListTest {

    // (1)
    @Inject
    private Validator validator;

    @Test
    public void testExistInCodeList() {

        // setup
        TicketSearchForm ticketSearchForm = new TicketSearchForm();
        // (2)
        ticketSearchForm.setDepAirportCd("AAA");

        // omitted

        // run the test
        Set<ConstraintViolation<TicketSearchForm>> violations = validator
                .validate(ticketSearchForm);

        // assert
        // (3)
        assertThat(violations.size(), is(1));
        ConstraintViolation<TicketSearchForm> violation = violations.iterator().
↪next();
        // (4)
        assertThat(violation.getPropertyPath().toString(), is("depAirportCd"));
        // (5)
        assertThat((String) violation.getInvalidValue(), is("AAA"));
        // (6)
        assertThat(violation.getMessage(), is("Does not exist in CL_AIRPORT"));
    }
}
```

項番	説明
(1)	`Validator` に Spring の `LocalValidatorFactoryBean` から生成した `Validator` を DI している。 `LocalValidatorFactoryBean` から生成した `Validator` は Spring の DI コンテナ上で動作し、`@ContextConfiguration` で読み込んだコードリストの Bean を取得することができる。これにより、`@ExistInCodeList` を期待通りに動作させることができる。

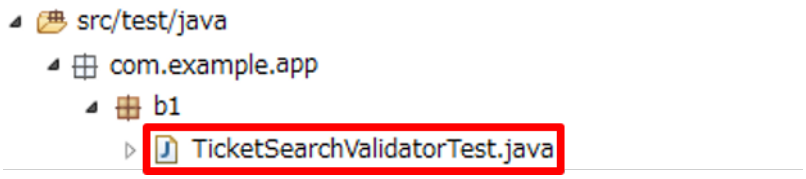
Continued on next page

表 10.5 – continued from previous page

項番	説明
(2)	コードリストに存在しないコードを入力し、`@ExistInCodeList` でエラーが発生することを期待する。
(3)	`size` メソッドを使って入力チェックエラーの数を取得し、エラーが発生したかどうかを確認する。
(4)	違反したフィールドが想定した箇所であるかを確認する。
(5)	違反した入力値が想定した値であるかを確認する。
(6)	発生したエラーのメッセージを確認する。

Spring Validator で実装した Validator の単体テスト

`Validator(Spring Validation)` の単体テストにおいて、作成するファイルを以下に示す。



作成するファイル名	説明
`TicketSearchValidatorTest.java`	`TicketSearchValidator.java` のテストクラス

以下に、テスト対象のクラスを示す。

- `TicketSearchValidator.java`

```
@Component
public class TicketSearchValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
```

```
        return (TicketSearchForm.class).isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {

        TicketSearchForm form = (TicketSearchForm) target;

        if (!errors.hasFieldErrors("depAirportCd")
            && !errors.hasFieldErrors("arrAirportCd")) {
            String depAirport = form.getDepAirportCd();
            String arrAirport = form.getArrAirportCd();
            if (depAirport.equals(arrAirport)) {
                errors.reject(TicketSearchErrorCode.E_AR_B1_5001.code());
            }
        }

        // omitted
    }
}
```

以下に、Validator(Spring Validation) のテストクラス作成方法を説明する。ここでは、テスト対象の TicketSearchValidator でエラーになる値を TicketSearchForm に設定してバリデーションエラーになることと、エラーメッセージが正しいことを確認している。

- TicketSearchValidatorTest.java

```
public class TicketSearchValidatorTest {

    private TicketSearchValidator validator;

    private TicketSearchForm ticketSearchForm;

    private BindingResult result;

    @BeforeClass
    public void setUpBeforeClass() {

        // setup
        validator = new TicketSearchValidator();
    }

    @Test
    public void testTicketSearchValidator() {

        // setup
        ticketSearchForm = new TicketSearchForm();
        result = new DirectFieldBindingResult(ticketSearchForm, "TicketSearchForm
↪");

        ticketSearchForm.setFlightType(FlightType.RT);
        ticketSearchForm.setDepAirportCd("HND");
```

```
        ticketSearchForm.setArrAirportCd("HND");
        // omitted

        // run the test
        // (1)
        validator.validate(ticketSearchForm, result);

        // (2)
        assertThat(result.hasErrors(), is(true));

        // (3)
        ObjectError error = result.getGlobalError();

        // (4)
        ResourceBundleMessageSource messageSource = new␣
↪ResourceBundleMessageSource();
        // (5)
        messageSource.setBasename("i18n/sample-messages");
        // (6)
        messageSource.setUseCodeAsDefaultMessage(true);

        String code = error.getCode();
        // assert
        // (7)
        assertThat(code, is(TicketSearchErrorCode.E_AR_B1_5001.code()));
        assertThat(messageSource.getMessage(error, Locale.JAPAN),
                is("出発空港と到着空港に同じ空港は指定できません。区間をご確認ください。"));
    }
}
```

項番	説明
(1)	validate メソッドの引数に、ticketSearchForm と、BindingResult インターフェースのオブジェクトを指定することで、ticketSearchForm に対する入力チェックの結果が、BindingResult クラスのオブジェクトに格納される。
(2)	hasErrors メソッドを使って、エラーの有無を判定する。 エラーがある場合は true が返り値として返り、エラーがない場合は false が返り値として返る。
(3)	getGlobalError メソッドで、エラー内容を取得する。

Continued on next page

表 10.6 – continued from previous page

項番	説明
(4)	エラーメッセージの内容を確認するために、`MessageSource` の実装クラスである `org.springframework.context.support.ResourceBundleMessageSource` のオブジェクトを生成する。クラスの詳細については、ResourceBundleMessageSource の Javadoc を参照されたい。
(5)	`setBasename` メソッドに、メッセージが定義されたプロパティファイルを指定して読み込ませる。
(6)	`setUseCodeAsDefaultMessage` メソッドに true を指定すると、エラーコードに対応するメッセージが定義されていない場合にエラーコードが返される。false を指定すると、エラーコードに対応するメッセージが定義されていない場合に `NoSuchMessageException` が返される。デフォルトでは false が適用されている。
(7)	エラーコード、メッセージ内容を検証する。

10.2.4 単体テストで利用する OSS ライブラリの使い方

本節では、単体テストで利用する OSS ライブラリとして、Spring Test（MockMvc）、Mockito について説明する。

Spring Test

Spring Test とは

Spring Test とは、Spring Framework 上で動作するアプリケーションのテストを支援するモジュールである。テストには、対象クラスが依存しているクラスをモックやスタブで代用して行うテストと、Spring の DI コンテナや実際の依存クラスと組み合わせて行うテストがある。

本ガイドラインでは、モックを使用してテスト対象クラス単体でテストを行う方法と、設定ファイルや実際の依存クラスを組み合わせてテストを行う方法の 2 通りの実装方法を例示する。

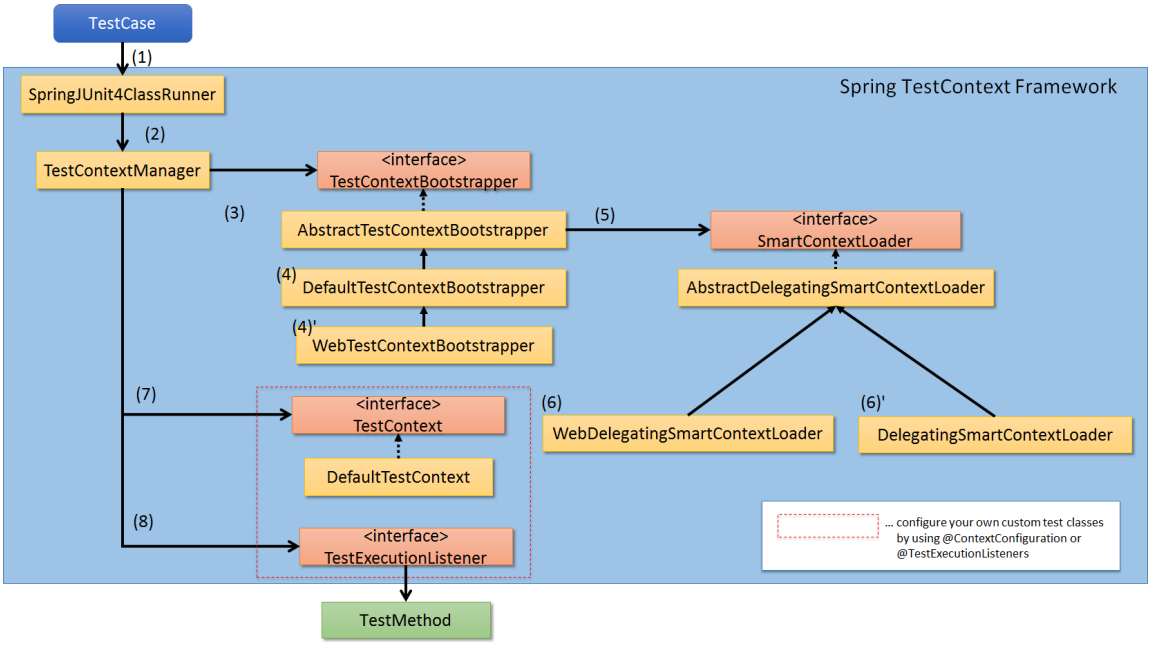
Spring Test では主に以下の機能が提供されている。

- テスティングフレームワーク（JUnit）上で Spring の DI コンテナを動かす機能
- テストデータをセットアップする機能
- アプリケーションサーバ上にデプロイせずに、Spring MVC の動作を再現する機能
- テストに最適なトランザクション管理機能

その他、様々な Spring 固有のアノテーションや、単体テストで利用する API が提供されている。

Spring Test は、テスティングフレームワーク上で動作するテスト用のフレームワーク機能として、Spring TestContext Framework を提供している。

Spring TestContext Framework の処理フロー図を以下に示す。



項番	説明
(1)	テスト実行により、`org.springframework.test.context.junit4.SpringJUnit4ClassRunner` クラスが呼び出される。
(2)	`SpringJUnit4ClassRunner` クラスは `org.springframework.test.context.TestContextManager` クラスを生成する。
(3)	`TestContextManager` クラスは `org.springframework.test.context.TestContextBootstrapper` インタフェースの `org.springframework.test.context.TestContext` インタフェースのビルド処理を呼び出す。

Continued on next page

表 10.7 – continued from previous page

項番	説明
(4)	TestContextBootstrapper クラスはテストクラスで指定された設定ファイルをマージする org.springframework.test.context.MergedContextConfiguration クラスのビルド処理を呼び出す。 この時、テストクラスに明示的にブートストラップが指定されていない場合、@WebAppConfiguration があれば org.springframework.test.context.web.WebTestContextBootstrapper クラス、指定されていなければ org.springframework.test.context.support.DefaultTestContextBootstrapper クラスが呼び出される。
(5)	MergedContextConfiguration クラスのビルド処理で org.springframework.test.context.SmartContextLoader インタフェースの実装クラスが呼び出される。
(6)	ブートストラップに WebTestContextBootstrapper クラスが使用されている場合は org.springframework.test.context.web.WebDelegatingSmartContextLoader クラス、DefaultTestContextBootstrapper クラスが使用されている場合は org.springframework.test.context.support.DelegatingSmartContextLoader クラスが SmartContextLoader インタフェースの実装クラスとして呼び出される。 SmartContextLoader インタフェースの実装クラスでテストクラスの@ContextConfiguration で指定された ApplicationContext をロードする。
(7)	ブートストラップで取得した MergedContextConfiguration クラスを使用して org.springframework.test.context.TestContext インタフェースの実装クラスである org.springframework.test.context.support.DefaultTestContext クラスを生成する。

Continued on next page

表 10.7 – continued from previous page

項番	説明
(8)	TestContextManager クラスにテストクラスの@TestExecutionListeners で指定された org.springframework.test.context.TestExecutionListener インタフェースを登録し、以下のエントリポイントで TestExecutionListener の処理を呼び出す。 • テストケースの全テストメソッド実行前（@BeforeClass） • テストインスタンスの生成後 • 各テストメソッドの実行前（@Before） • 各テストメソッドの実行後（@After） • テストケースの全テストメソッド実行後（@AfterClass） トランザクションの管理やテストデータのセットアップ処理は、TestExecutionListener の処理によって行われる。 TestExecutionListener の登録については *TestExecutionListener の登録*を参照されたい。

Spring Test の DI 機能

テストケースの@ContextConfiguration に設定ファイルを指定すると、SpringJUnit4ClassRunner にデフォルトで設定されている DependencyInjectionTestExecutionListener の処理によってテスト実行時に Spring の DI 機能を利用することができる。

以下に@ContextConfiguration を使用して設定ファイルを読み込む例を示す。ここでは、アプリケーションで使用する sample-infra.xml を使用してテスト対象の com.example.domain.repository.member.MemberRepository をインジェクションしている。

- sample-infra.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
        http://mybatis.org/schema/mybatis-spring http://mybatis.org/schema/mybatis-
↪spring.xsd
    ">

  <import resource="classpath:/META-INF/spring/sample-env.xml" />

    <!-- define the SqlSessionFactory -->
```

```
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="configLocation" value="classpath:/META-INF/mybatis/mybatis-
↪config.xml" />
    </bean>

    <!-- scan for Mappers -->
    <mybatis:scan base-package="com.example.domain.repository" />

</beans>
```

- MemberRepositoryTest.java

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
        "classpath:META-INF/spring/sample-infra.xml" }) //(1)
@Transactional
public class MemberRepositoryTest {

    @Inject
    MemberRepository target; // (2)
}
```

項番	説明
(1)	@ContextConfiguration に sample-infra.xml を指定する。
(2)	sample-infra.xml に定義された<mybatis:scan>で Bean 登録されている MemberRepository をインジェクションする。

TestExecutionListener の登録

テストケースに@TestExecutionListeners アノテーションを明示的に指定しない場合、Spring Test が提供している以下の org.springframework.test.context.TestExecutionListener インタフェースの実装クラスがデフォルトで登録される。

なお、@TestExecutionListeners アノテーションを明示的に指定しない場合、デフォルトで登録される TestExecutionListener は Order を持っており、呼び出し順は下記表の順に固定されている。TestExecutionListener が個別に指定された場合は、指定された順番通りに呼び出される。

TestExecutionListener の実装クラス	説明
ServletTestExecutionListener	WebApplicationContext のテストをサポートするモックサーブレット API を設定する機能を提供している。
DirtiesContextBeforeModesTestExecutionListener	テストで使用する DI コンテナのライフサイクル管理機能を提供している。テストクラスまたはテストメソッドの実行前に呼び出される。
DependencyInjectionTestExecutionLiLstener	テストで使用するインスタンスへの DI 機能を提供している。
DirtiesContextTestExecutionListener	テストで使用する DI コンテナのライフサイクル管理機能を提供している。テストクラスまたはテストメソッドの実行後に呼び出される。
TransactionalTestExecutionListener	テスト実行時のトランザクション管理機能を提供している。
SqlScriptsTestExecutionListener	@Sql アノテーションで指定されている SQL を実行する機能を提供している。

各 TestExecutionListener の詳細は TestExecutionListener configuration を参照されたい。

TestExecutionListener は通常、デフォルト設定から変更する必要はないが、テストライブラリが独自に提供している TestExecutionListener を使用する場合は@TestExecutionListeners アノテーションを使用して TestContextManager に登録する必要がある。

ここでは例として、Spring Test DBUnit が提供する TransactionDbUnitTestExecutionListener を登録する方法を説明する。

- MemberRepositoryDbunitTest.java

```
@TestExecutionListeners({                                    // (1)
        DirtiesContextBeforeModesTestExecutionListener.class,
        DependencyInjectionTestExecutionListener.class,
        DirtiesContextTestExecutionListener.class,
        TransactionDbUnitTestExecutionListener.class})  // (2)
@Transactional
public class MemberRepositoryDbunitTest {
```

項番	説明
(1)	クラスレベルに`@TestExecutionListeners` アノテーションを付けて `TestExecutionListener` インタフェースの実装クラスを指定することで、テスト実行時に指定した `TestExecutionListener` の処理を呼び出すことができる。詳細は`@TestExecutionListeners` の Javadoc を参照されたい。
(2)	`TransactionDbUnitTestExecutionListener` は Spring Test DBUnit が提供する `TestExecutionListener` インタフェースの実装クラスである。`@DatabaseSetup` や`@ExpectedDatabase`、`@DatabaseTearDown` などのアノテーションを使用したデータのセットアップ、検証、後処理の機能を提供している。 `TransactionDbUnitTestExecutionListener` は内部で `TransactionalTestExecutionListener` と `com.github.springtestdbunit.DbUnitTestExecutionListener` をチェインしている。

警告: DbUnitTestExecutionListener の注意点

テストケース内で`@Transactional` を指定せずに Spring Test DBUnit の提供する `DbUnitTestExecutionListener` を使用した場合、`@DatabaseSetup` などのアノテーションのトランザクションと、テスト対象クラスのトランザクションは別になるため、データのセットアップが反映されないなど正常に動作しない可能性があることに注意されたい。なお、テストケース内で`@Transactional` を指定する場合は `DbUnitTestExecutionListener` の代わりに `TransactionDbUnitTestExecutionListener` が提供されているため、そちらを使用する必要がある。

MockMvc

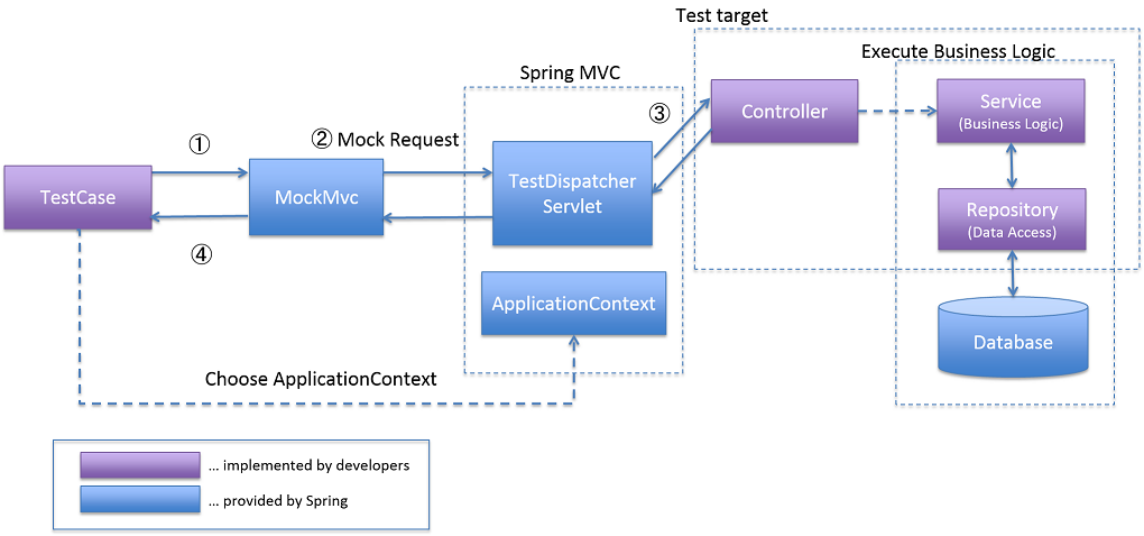
`MockMvc` は、本来 Spring Test の機能に含まれるが、本章ではアプリケーション層の単体テストにおいて使用しているため、Spring Test の説明と切り出して詳しく説明する。

MockMvc とは

Spring Test には、Spring MVC フレームワークと結合した状態でテストするための仕組みとして、`org.springframework.test.web.servlet.MockMvc` クラスを提供している。`MockMvc` を使用すると、アプリケーションサーバ上にデプロイすることなく Spring MVC の動作を再現できるため、サーバやデータ

ベースを用意する手間を省くことができる。なお、Spring MVC の詳細については *Overview of Spring MVC Processing Sequence* を参照されたい。

テスト実行時にリクエストを受けてから、レスポンスを返すまでの MockMvc の処理フローを、以下の図に示す。



項番	説明
(1)	テストメソッドは、Spring Test が用意した org.springframework.test.web.servlet.TestDispatcherServlet にリクエストするデータをセットアップする。
(2)	MockMvc は TestDispatcherServlet に疑似的なリクエストを行なう。
(3)	TestDispatcherServlet は、リクエスト内容に一致する Controller のメソッドを呼び出す。
(4)	テストメソッドは、MockMvc から実行結果を受け取り、実行結果の妥当性を検証する。

また、MockMvc には 2 つの動作オプションが実装されている。テストを行う際は、それぞれの特性を把握し、

用途毎に適したオプションを選択されたい。

以下に、2 つのオプションの概要を示す。

動作オプション	概要
webAppContextSetup	`spring-mvc.xml` などで定義した Spring MVC の設定を読み込み、`WebApplicationContext` を生成することで、デプロイ時とほぼ同じ状態でテストすることができる。
standaloneSetup	`Controller` に DI されているコンポーネントを、テストで利用する設定ファイルに定義することで、Spring Test が生成した DI コンテナを用いてテストを行うことができる。よって、Spring MVC のフレームワーク機能を利用しつつ、`Controller` のテストを単体テスト観点で行なうことができる。

以下に、2 つのオプションのメリット、デメリットを示す。

動作オプション	メリット	デメリット
webAppContextSetup	実際の稼働で使用する設定ファイルを読み込むことで、アプリケーションを動かさなければ確認できないこともデプロイなしで検証することができる。実際の設定ファイルを読み込みテストするため、設定ファイルが正しく作成されているかを確認することもできる。 また、Bean 定義にモッククラスを指定しておけば、`Controller` に DI される `Service` などをモック化することも可能である。	巨大なアプリケーションをテストする場合や、膨大な Bean 定義を読み込む場合は実行に時間がかかってしまう。そのため、デプロイする場合の設定ファイルから、必要な記述だけを抽出した設定ファイルを用意するなどの工夫が必要となる。
standaloneSetup	生成される DI コンテナに特定の `Interceptor` や `Resolver` 等を適用してテストを実施できる。そのため、Spring の設定ファイルを参照せずコントローラ単体だけ見たい場合は、`webAppContextSetup` よりも実施コストが低い。	`Interceptor` や `Resolver` などを多く適用するテストにおける設定コストが高い。 また、あくまで `Contoroller` の単体テスト観点で動作するため、Spring MVC のフレームワーク機能と合わせて `Controller` のテストを行いたい場合は、`webAppContextSetup` でのテストを検討する必要があることに留意されたい。

MockMvc のセットアップ

ここでは `MockMvc` の 2 つのオプションについて、実際にテストで使用する際のセットアップ方法を説明する。

webAppContextSetup によるセットアップ

ここでは、`webAppContextSetup` でテストを行うためのセットアップ方法について説明する。

MockMvc のセットアップ設定例を以下に示す。

- MockMvc のセットアップ設定例

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextHierarchy({ @ContextConfiguration({ // (1)
        "classpath:META-INF/spring/applicationContext.xml",
        "classpath:META-INF/spring/spring-security.xml" }),
        @ContextConfiguration("classpath:META-INF/spring/spring-mvc.xml") })
@WebAppConfiguration // (2)
public class MemberRegisterControllerWebAppContextTest {

    @Inject
    WebApplicationContext webApplicationContext; // (3)
```

```
    MockMvc mockMvc;

    @Before
    public void setUp() {

        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext) // (4)
                .alwaysDo(log()).build();
    }

    @Test
    public void testRegisterConfirm01() throws Exception {

        ResultActions results = mockMvc.perform(post("/member/register")
                // omitted
                .param("confirm", "");

        results.andExpect(status().is(200));

        // omitted
    }
}
```

項番	説明
(1)	テスト実行時に生成する DI コンテナの設定ファイルを指定する。DI コンテナの階層関係については、`@org.springframework.test.context.ContextHierarchy` を使うことで再現することができる。DI コンテナの階層関係についてはアプリケーションコンテキストの構成と *Bean* 定義ファイルの関係を参照されたい。
(2)	Web アプリケーション向けの DI コンテナ（`WebApplicationContext`）が作成できるようになる。また、`@WebAppConfiguration` を指定すると開発プロジェクト内の `src/main/webapp` が Web アプリケーションのルートディレクトリになるが、これは Maven の標準構成と同じなので特別に設定を加える必要はない。
(3)	テスト実行時に使用する DI コンテナをインジェクションする。
(4)	テスト実行時に使用する DI コンテナを指定して、MockMvc を生成する。

standaloneSetup によるセットアップ

ここでは、standaloneSetup でテストを行うためのセットアップ方法について説明する。

MockMvc のセットアップ設定例を以下に示す。

- MockMvc のセットアップ設定例

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
        "classpath:META-INF/spring/applicationContext.xml",
        "classpath:META-INF/spring/test-context.xml",
        "classpath:META-INF/spring/spring-mvc-test.xml"})
public class MemberRegisterControllerStandaloneTest {

    @Inject
    MemberRegisterController target;

    MockMvc mockMvc;

    @Before
    public void setUp() {
        mockMvc = MockMvcBuilders.standaloneSetup(target).alwaysDo(log()).build(); // (1)
    }

    @Test
    public void testRegisterConfirm01() throws Exception {

        ResultActions results = mockMvc.perform(post("/member/register")
                // omitted
                .param("password", "testpassword")
                .param("reEnterPassword", "testpassword"));

        results.andExpect(status().is(200));

        // omitted
    }
}
```

項番	説明
(1)	テスト対象の Controller を指定して、MockMvc を生成する。必要に応じて org.springframework.test.web.servlet.setup.StandaloneMockMvcBuilder のメソッドを呼び出して、Spring Test が生成する DI コンテナをカスタマイズすることができる。カスタマイズするためのメソッドについての詳細は、StandaloneMockMvcBuilder の Javadoc を参照されたい。

MockMvc によるテストの実装

ここでは MockMvc によるテスト実行の流れとして、リクエストデータの設定から、リクエスト送信の実装方法、実行結果の検証、出力まで説明する。

リクエストデータの設定

リクエストデータの設定は、org.springframework.test.web.servlet.request.MockHttpServletRequestBuilder や org.springframework.test.web.servlet.request.MockMultipartHttpServletRequestBuilder のファクトリメソッドを使用して行う。

ここでは、2 つのクラスのファクトリメソッドの中から主要なメソッドについて紹介する。詳細は、MockHttpServletRequestBuilder の Javadoc または MockMultipartHttpServletRequestBuilder の Javadoc を参照されたい。

表 10.8 MockHttpServletRequestBuilder の主なメソッド

メソッド名	説明
param/ params	テスト実行時のリクエストに、リクエストパラメータを追加するメソッド。
content	テスト実行時のリクエストに、リクエストボディを追加するメソッド。
header/ headers	テスト実行時のリクエストに、リクエストヘッダーを追加するメソッド。contentType や accept などの特定のヘッダーを指定するためのメソッドも提供されている。
requestAttr	リクエストスコープにオブジェクトを設定するメソッド。
flashAttr	フラッシュスコープにオブジェクトを設定するメソッド。
sessionAttr	セッションスコープにオブジェクトを設定するメソッド。
cookie	テスト実行時のリクエストに、指定した cookie を追加するメソッド。

表 10.9 MockMultipartHttpServletRequestBuilder の主なメソッド

メソッド名	説明
file	テスト実行時のリクエストに、アップロードするファイルを設定するメソッド。

ここでは、param メソッドを用いたリクエストデータの設定と、post メソッドを用いたリクエスト実行の例を示す。

以下に、テスト対象の Controller の実装を示す。

- テスト対象の Controller クラス

```
@Controller
@RequestMapping("member/register")
@TransactionTokenCheck("member/register")
public class MemberRegisterController {

    @RequestMapping(method = RequestMethod.POST, params = "confirm")
    public String registerConfirm(@Validated MemberRegisterForm memberRegisterForm,
        BindingResult result, Model model) {
```

```
        // omitted

        return "C1/memberRegisterConfirm";
    }
}
```

以下に、リクエスト送信の実装例を示す。

- リクエスト送信の実装例

```
@Test
public void testRegisterConfirm01() throws Exception {
    mockMvc.perform(
         post("/member/register")
        .param("confirm", "")); // (1)
}
```

項番	説明
(1)	form をリクエストパラメータに持つリクエストデータを設定している。

リクエスト送信の実装

設定したリクエストデータを MockMvc の perform メソッドの引数として渡すことで、テストで利用するリクエストデータを設定し、DispatcherServlet に疑似的なリクエストを行なう。MockMvcRequestBuilders のメソッドには、get、post、fileUpload といったメソッドが、リクエストの種類ごとに提供されている。詳細は、MockMvcRequestBuilders の Javadoc を参照されたい。

以下に、リクエスト送信の実装例を示す。

- リクエスト送信の実装例

```
@Test
public void testRegisterConfirm01() throws Exception {
    mockMvc.perform( // (1)
         post("/member/register") // (2)
        .param("confirm", ""));
}
```

項番	説明
(1)	リクエストを実行し、返り値として実行結果の検証を行うための ResultActions クラスを返す。詳細は後述の実行結果検証の実装を参照されたい。
(2)	/ticket/search へ POST リクエストを実行するように設定している。

警告: テスト時のトランザクショントークンチェック、CSRF チェック

テスト対象がトランザクショントークンチェックや CSRF チェックを利用している場合は、mockMvc のリクエストについてもチェックが適用されることに注意されたい。なお、本章では spring-security の設定は無効にしているため、CSRF チェックは行われていない。

実行結果検証の実装

実行結果の検証には、org.springframework.test.web.servlet.ResultActions の andExpect メソッドを使用する。andExpect メソッドの引数には org.springframework.test.web.servlet.ResultMatcher を指定する。Spring Test は、org.springframework.test.web.servlet.result.MockMvcResultMatchers のファクトリメソッドを介してさまざまな ResultMatcher を提供している。

ここでは、andExpect メソッドの引数として、主要となる MockMvcResultMatchers のメソッドを紹介する。ここで紹介しないメソッドについては、MockMvcResultMatchers の Javadoc を参照されたい。

表 10.10 MockMvcResultMatchers の主なメソッド

メソッド名	説明
status	HTTP ステータスコードを検証するメソッド。
view	Controller が返却した View 名を検証するメソッド。
model	Spring MVC の Model について検証するメソッド
request	リクエストスコープおよびセッションスコープの状態、Servlet 3.0 からサポートされている非同期処理の処理状態を検証するメソッド。
flash	フラッシュスコープの状態を検証するメソッド。
redirectedUrl	リダイレクト先のパスを検証するメソッド。redirectedUrlPattern メソッドを用いたパターンによる検証も提供されている。
fowardedUrl	フォワード先のパスを検証するメソッド。forwardedUrlPattern メソッドを用いたパターンによる検証も提供されている。
content	レスポンスボディの中身を検証するメソッド。jsonPath や xPath などの特定のコンテンツ向けのメソッドも提供されている。
header	レスポンスヘッダーの状態を検証するメソッド。
cookie	cookie の状態を検証するメソッド。

以下に、テストの実行結果検証の実装例を示す。

- 実行結果検証の実装例

```
@Test
public void testRegisterConfirm01() throws Exception {
    mockMvc.perform(post("/member/register")
        .param("confirm", ""));
        .andExpect(status().is(302)) // (1)
}
```

項番	説明
(1)	テスト実行時のリクエストデータを設定している。andExpect メソッドは ResultActions からチェーンして記述することができるため、IDE の補完機能によってコーディングの負担を減らすことができる。

警告：Model の検証とアサーションライブラリ

Spring Test では Model の検証として、model メソッドにチェーンする形で org.springframework.test.web.servlet.result.ModelResultMatchers の attribute メソッドを使用することができる。このメソッドを用いることで Model の中身を検証することができるが、引数として Hamcrest の org.hamcrest.Matcher を使用するため、Hamcrest 以外のアサーションライブラリを使用する場合は注意されたい。

Hamcrest 以外のアサーションライブラリを併用する場合は、MvcResult から ModelAndView オブジェクトを取得し、さらに ModelAndView オブジェクトから Model に格納されたオブジェクトを取得することで、使用しているアサーションライブラリを使って Model を検証することができる。

以下に ModelAndView オブジェクトから取得した Model の検証例を示す。

```
@Test
public void testRegisterConfirm01() throws Exception {
    MvcResult mvcResult = mockMvc.perform(post("/member/register").param(
↪"confirm", "")
                .param("kanjiFamilyName", "電電")
                .andExpect(status().is(200))
                .andReturn(); // (1)

    ModelAndView mav = mvcResult.getModelAndView(); // (2)

    MemberRegisterForm actForm = (MemberRegisterForm) mav.getModel().get(
↪"memberRegisterForm");

    assertThat(actForm.getKanjiFamilyName(), is("電電"));
    // omitted
}
```

項番	説明
(1)	ResultActions の andReturn メソッドを使用して MvcResult オブジェクトを取得する。
(2)	MvcResult から ModelAndView オブジェクトを取得し、ModelAndView オブジェクトから Model に格納されたオブジェクトを取得して Model の検証を行う。

実行結果出力の実装

テスト実行時のログ出力などを有効化する場合は、ResultActions の alwaysDo メソッドや andDo メソッドを使う。ログの出力などは共通処理になる場合が多いため、MockMvc 生成時に StandaloneMockMvcBuilder の alwaysDo メソッドを使うことを推奨する。

alwaysDo メソッドの引数には、実行結果に対して任意の処理を行なう org.springframework.test.web.servlet.ResultHandler を指定する。Spring Test では、org.springframework.test.web.servlet.result.MockMvcResultHandlers のファクトリメソッドを介してさまざまな ResultHandler を提供している。ここでは、alwaysDo メソッドの引数として主要となる MockMvcResultHandlers のメソッドを紹介する。各メソッドの詳細については、MockMvcResultHandlers の Javadoc を参照されたい。

表 10.11　MockMvcResultHandlers の主なメソッド

メソッド名	説明
log	実行結果をデバッグレベルでログ出力するメソッド。ログ出力時に使用されるロガー名は org.springframework.test.web.servlet.result である。
print	実行結果を任意の出力先に出力するメソッド。出力先を指定しない場合、標準出力が出力先になる。

以下に、テストの実行結果出力の設定例を示す。

- 実行結果出力の設定例

```
@Before
public void setUp() {
    mockMvc = MockMvcBuilders.standaloneSetup(target).alwaysDo(log()).build(); //␣
↪(1)
}
```

項番	説明
(1)	alwayDo メソッドの引数に log メソッドを指定することで、mockMvc を用いたテスト実行の際は、常に実行結果をログとして出力する。

注釈: テストケースごとの出力設定

テスト実行時のログ出力などをテストケースごとに有効化する場合は、ResultActions の andDo メソッドを使う。andDo メソッドも alwaysDo メソッドと同じく引数に ResultHandler を指定する。

以下に、ログ出力をテストケースごとに有効化する場合の設定例を示す。

- ログ出力を常に有効化する場合の設定例

```
@Test
public void testSearchForm() throws Exception {
    mockMvc.perform(get("/ticket/search").param("form", ""))
            .andDo(log()); // (1)
}
```

項番	説明
(1)	テストの実行結果をログ出力する。

Mockito

ここでは、モックの概要、Mockito の使い方について説明する。

Mockito とは

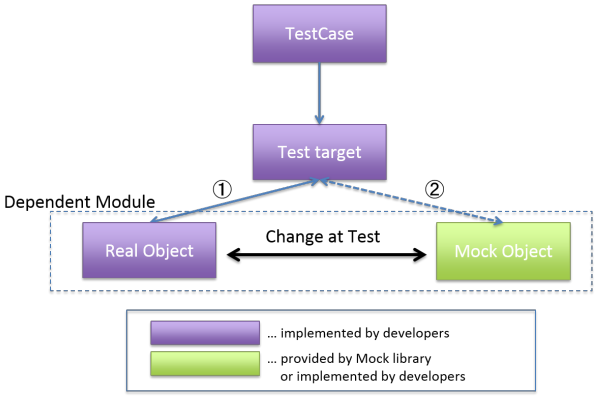
Mockito とは、単体テストにおいてテスト対象の依存クラスをモック化する際に使われるモックライブラリの一つである。モックライブラリを利用することで、モックの生成を簡単に行なうことができるため、単体テストの実装時にはよく利用されている。

ここからは、モックについての説明を行なう。

モックの概要

モックとは、テスト対象が依存するクラスの代用となる疑似クラスである。このように依存するクラスをモックに置き換えることをモック化と呼ぶ。モックは単体テストにおいて、依存クラスが正しく使用されているかを検証するために使用される。テスト対象のクラスのみ着目してテストを行いたい場合や、テスト対象の依存クラスが完成していないときは、依存クラスをモック化してテストを行なうことを考えるべきである。

以下に、モックを利用したテストのフロー図を示す。



項番	説明
(1)	依存クラスが作成され、動作も保障されている場合は、そのまま依存クラスを用いてテストすればよい。
(2)	依存クラスの動作が保障されていない場合や、作成されていない場合など、依存クラスを利用できない場合は、モッククラスを用いてテストする。

実際に依存クラスをモック化する場合、通常テスト実施者自身でモッククラスを用意する必要がある。しかし、検証のためのクラスをテストごとに一から作成していては、テスト実施に多大なコストがかかると予想される。そのような場合に利用されるのが、モック作成のためのモックライブラリである。モックライブラリを用いることで、より簡単にモックを作成することができるため、モックを利用するときはモックライブラリの使用を推奨する。

本章では、代表的なモックライブラリとして Mockito を使用し説明を行なう。

Mockito の利用

Mockito は、依存クラスのモック化、メソッドの呼び出し検証、メソッドの引数検証など、テストを行なう上で必要となる機能を提供している。しかし、テスト対象のコードによっては Mockito を利用できない場合もあるので注意されたい。

ここでは、Mockito を利用できない状況の中でも、特に注意が必要となる状況について紹介する。

モック化の制限

Mockito では、final 宣言、private 宣言されたクラス/メソッド、static 宣言されたメソッドをモック化することができない。通常のオブジェクト指向に沿った実装であれば、モック化に制限のかかるようなテストになることは考えにくいため、このような事態に直面した場合はテスト対象の設計に問題がないか一度確認したほうがよい。

その他の Mockito の制限については、What are the limitations of Mockito を参照されたい。

Mockito の機能

ここでは、Mockito の代表的な機能として、モックの作成、モック化されたメソッドの定義、検証について紹介する。

モックの生成

Mockito のモック化には 2 種類の方法が存在する。1 つは、mock メソッドを用いて依存クラスをすべてモックにする方法、もう 1 つは、spy メソッドを用いて依存クラスの一部のメソッドのみをモックにする方法である。

ここではより単純な、依存クラスをすべてモック化する方法について紹介する。依存クラスの一部のみをモックにする方法については、Mockito の Javadoc を参照されたい。

完全にモック化する場合、基本的には mock メソッドを用いてモック化する。

以下に、mock メソッドを用いたモック化の例を示す。

```
public class TicketReserveServiceImpl implements TicketReserveService {

    @Inject // (1)
    ReservationRepository reservationRepository;

    // omitted
}
```

項番	説明
(1)	テスト対象の TicketReserveServiceImpl は ReservationRepository をインジェクトしているため、ReservationRepository に依存した実装となっている。

```
public class TicketReserveServiceImplMockTest {

    ReservationRepository mockReservationRepository = mock(ReservationRepository.
↪class); // (1)

    private TicketReserveServiceImpl target;

    @Test
    public void testRegisterReservation() {
```

```
        target.reservationRepository = mockReservationRepository; // (2)

        // omitted
    }
}
```

項番	説明
(1)	mock メソッドを使うことで、TicketReserveServiceImpl が依存していた ReservationRepository をモック化している。
(2)	テスト対象のフィールドにモッククラスのオブジェクトを適用する。

mock メソッドを使用すると、テスト実施者自身が 1 つずつモックを適用していく必要があった。そのため、テスト対象の依存クラスが数多く存在する場合は、記述量も増加し実装コストもかかる。そのような場合は、@Mock アノテーションを使用したモックを自動で適用させる方法を推奨する。また、本章ではモック化の際に、mock メソッドより簡潔に記述できる@Mock アノテーションを使用している。

以下に、@Mock アノテーションを用いたモック化の例を示す。

- TicketReserveServiceImplMockTest.java

```
public class TicketReserveServiceImplMockTest {

    @Mock // (1)
    ReservationRepository mockReservationRepository;

    @InjectMocks // (2)
    private TicketReserveServiceImpl target;

    @Rule // (3)
    public MockitoRule mockito = MockitoJUnit.rule();

    // omitted
}
```

項番	説明
(1)	`@Mock` アノテーションをモック化したいクラスに付与することで、対象クラスのモックオブジェクトが Mockito によって自動的に代入される。モッククラスを別途定義する必要はない。
(2)	`@InjectMocks` アノテーションをテスト対象としたい具象クラスに付与することで、対象クラスのインスタンスが Mockito によって自動的に代入され、さらに対象クラスが依存しているクラスと、`@Mock` アノテーションが付与されたクラスが一致する場合、自動的にモックオブジェクトが設定される。
(3)	JUnit で Mockito を利用するための宣言。`@Rule` により、後述のアノテーションベースのモックオブジェクトの初期化機能が利用可能になる。

メソッドのモック化

モック化したオブジェクトの持つすべてのメソッドは、返り値がプリミティブ型の場合はそれぞれの型の初期値 (例: int 型の場合は 0) を、それ以外の場合は `null` を返すようなメソッドとして定義される。そのため、テストを行なう際は実施するテスト内容に合わせて、メソッドの返り値を改めて定義する必要がある。

引数、返り値の設定

メソッドのモック化には、`Mockito` クラスの `when` メソッドと、`when` メソッドが返す `org.mockito.stubbing.OngoingStubbing` インスタンスのメソッドを使用する。`when` メソッドの引数にはモック化するメソッドとその引数を指定し、実行時の返り値を `OngoingStubbing` のメソッドで定義する。

以下に、`OngoingStubbing` の主なメソッドを示す。`OngoingStubbing` の詳細については、OngoingStubbing の Javadoc を、また `when` メソッドについては Mockito の Javadoc を参照されたい。

表 10.12　**OngoingStubbing の主なメソッド**

メソッド名	説明
`thenReturn`	メソッドが呼び出されるときの返り値を引数に設定するメソッド。
`thenThrow`	メソッドが呼び出されるときに throw される `Throwable` オブジェクトを引数に設定するメソッド。引数には throw したい例外クラスの `java.lang.Class` オブジェクトを指定することもできる。

以下に、`thenReturn` の使用例を示す。

```
public class TicketReserveServiceImplMockTest {

    @Test
    public void testRegisterReservation() {
```

```
        Reservation testReservation = new Reservation();
        reservation.setReserveNo("0000000001");

        when(reservationRepository.insert(testReservation)).thenReturn(1); // (2)
    }
}
```

項番	説明
(1)	when メソッドの引数には、動作を定義したいメソッドとその引数を指定する。
(2)	insert メソッドの引数に testReservation を指定することで、テスト対象が insert メソッドを引数 testReservation で実行するとき、返り値は」1」 になる。

任意の引数による定義

モック化したいメソッドの引数に org.mockito.Matchers のメソッドを用いることで、任意の引数を対象に返り値を定義することもできる。

以下に、Matchers の主なメソッドを示す。詳細については、Matchers の Javadoc を参照されたい。

表 10.13　**Matchers の主なメソッド**

メソッド名	説明
any	モック化されるメソッドの引数が任意の Object であることを示すメソッド。
anyString	モック化されるメソッドの引数が null 以外の任意の String であることを示すメソッド。
anyInt	モック化されるメソッドの引数が任意の int 型、または null 以外の任意の Integer であることを示すメソッド。

以下に、any の使用例を示す。

```
public class TicketReserveServiceImplMockTest {

    @Test
    public void testRegisterReservation() {

        // omitted

        when(reservationRepository.insert((Reservation) any())).thenReturn(0); //␣
↪(1)
    }
}
```

項番	説明
(1)	insert メソッドの引数として Reservation にキャストした any メソッドを指定することで、insert メソッドを任意の Reservation 引数で実行するとき、返り値が」0」 になるように設定している。

返り値が void 型であるメソッドのモック化

モック化された返り値が void 型であるメソッドは、デフォルトでは何も動作しないメソッドとして定義される。そのため、例外をスローさせたい場合などは改めて定義する必要がある。

動作の設定

when メソッドでは定義できない返り値が void 型であるメソッドについては、Mockito クラスの doThrow メソッドなどを用いることで定義できる。

以下に、返り値が void 型であるメソッドを再定義するための Mockito の主なメソッドを示す。詳細については、Mockito の Javadoc を参照されたい。

表 10.14　返り値が void 型であるメソッドを再定義する Mockito の主なメソッド

メソッド名	説明
doThrow	返り値が void 型であるメソッドに throw させる例外を設定する場合に用いるメソッド。
doNothing	返り値が void 型であるメソッドに何もさせないよう設定する場合に用いるメソッド。

以下に、doThrow の使用例を示す。

```
doThrow(new RuntimeException()).when(mock).someVoidMethod(); // (1)
```

項番	説明
(1)	doThrow メソッドは when メソッドの前に記述し、when メソッドはその後、チェーンする形で記載する。 doThrow メソッドの引数にスローしたい例外を指定することで、モック化したメソッドが実行されるときに例外をスローするようになる。

モック化したメソッドの検証

Mockito で作成したオブジェクトをモックとして用いる場合は、Mockito クラスの verify メソッドを用いることで、モック化したメソッドの呼び出しについて検証することができる。

verify メソッドは引数にモックを指定し、チェーンする形でモック化したメソッドを続けることで、そのメソッドが正しく呼ばれているかどうかを検証できる。また、verify メソッドの引数としてモックと

org.mockito.verification.VerificationMode を指定することで、より詳しくメソッドの呼び出しについて検証できる。

以下に、VerificationMode の主なメソッドを示す。詳細については、VerificationMode の Javadoc を参照されたい。

表 10.15　**VerificationMode の主なメソッド**

メソッド名	説明
times	期待する呼び出し回数を設定するメソッド。引数に期待する呼び出し回数を設定できる。 verify メソッドの引数に VerificationMode を指定しない場合は times(1) が設定される。
never	呼び出されていないことを期待する場合に設定するメソッド。

以下に、times の使用例を示す。

```
@Test
public void testRegisterReservation() {

    // omitted

    when(reservationRepository.insert(testReservation)).thenReturn(1);

    TicketReserveDto ticketReserveDto = target.
↪registerReservation(testReservation); // (1)

    verify(reservationRepository, times(1)).insert(testReservation); // (2)
}
```

項番	説明
(1)	target の insert メソッドでは、ReservationRepository の insert メソッドが 1 回実行されるような実装になっている。
(2)	verify メソッドの引数にモックオブジェクトと、times メソッドを指定することで、insert メソッドが引数 testReservation で正しく 1 回呼ばれているかを検証することができる。 この場合は、times メソッドの引数が」1」 なので省略しても同様の検証となる。

10.3 単体テストの実行

10.3.1 テストの実行方法

本節では、JUnit の実行方法として、

- IDE 上で JUnit を実行する方法
- Maven で JUnit を実行する方法

の 2 種類を説明する。用途に応じていずれかを選択されたい。

テスト実行方法	説明
IDE	IDE 上から JUnit を実行する。
Maven	`mvn test` コマンドを使用して JUnit を実行する。

IDE 上から JUnit を実行する場合は、IDE 上で製造からテストまで行うことができるため、テスト対象への参照も容易となる。

`mvn test` コマンドを使用して JUnit を実行する場合は、コマンドベースで実行するため、CI サーバ上でのテストの自動化が可能となる。

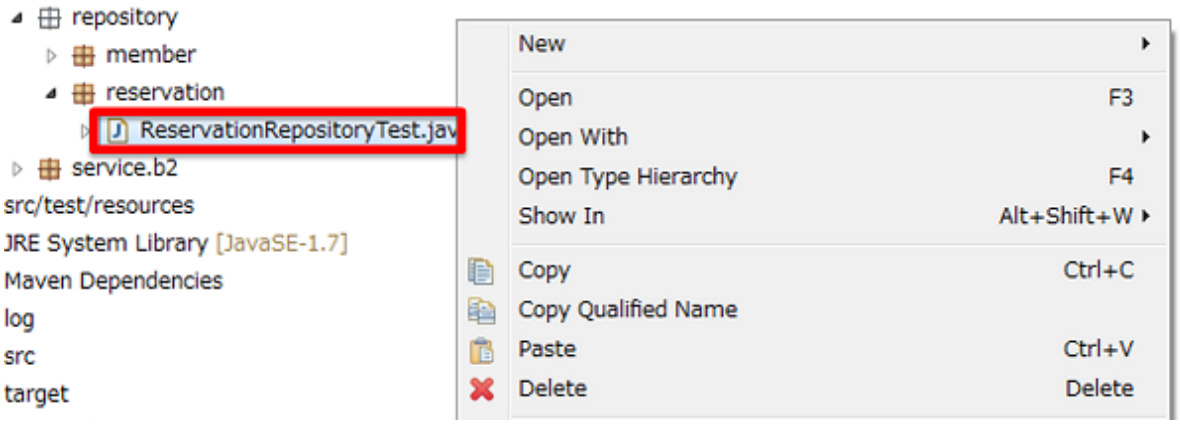
10.3.2 テストの実行

IDE 上でテストを実行

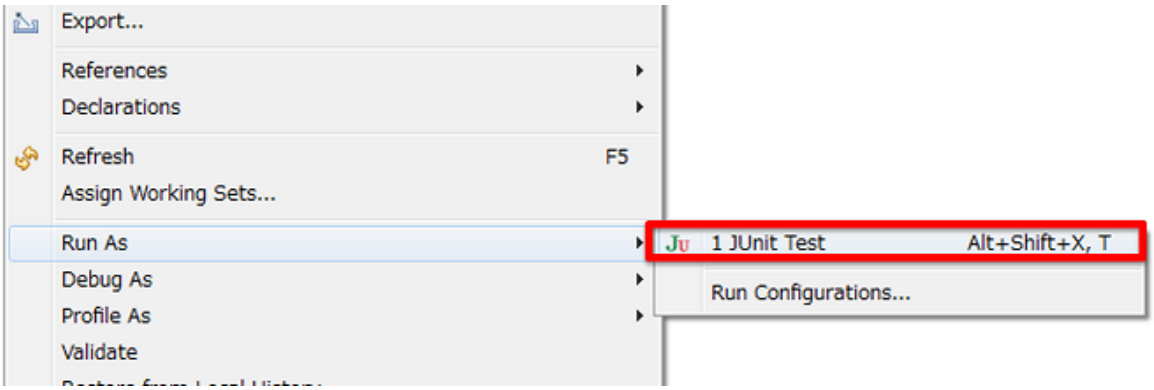
STS を用いて JUnit を実行する方法を紹介する。

テストクラスの実行

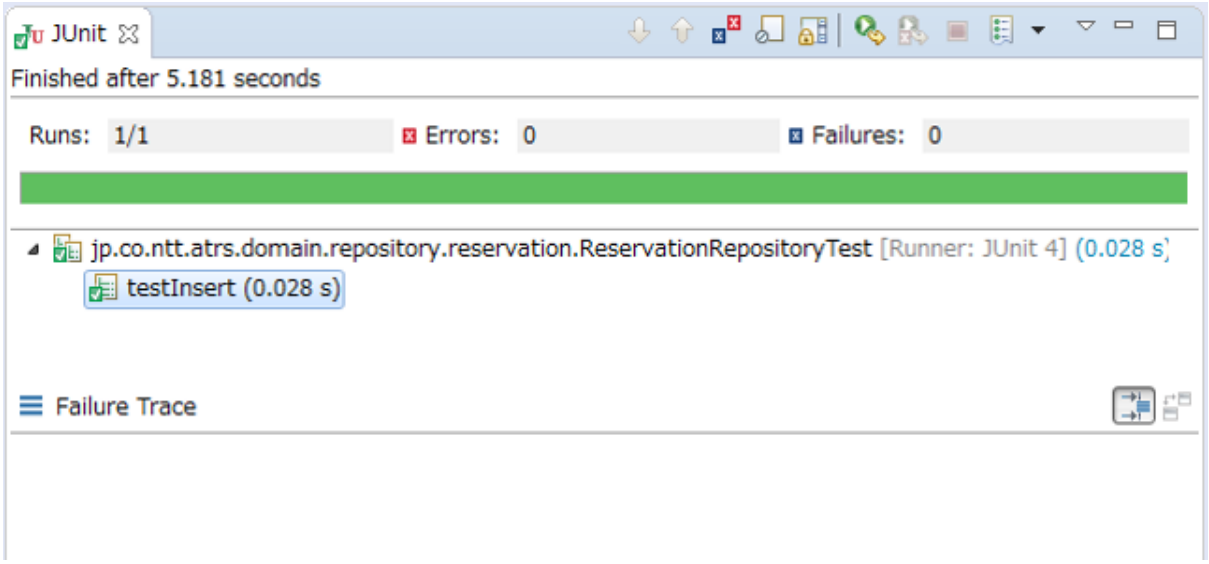
テストクラスを右クリックし、メニューを表示させる。



メニューから [Run As] -> [JUnit Test] を選択し、対象テストクラスを実行する。

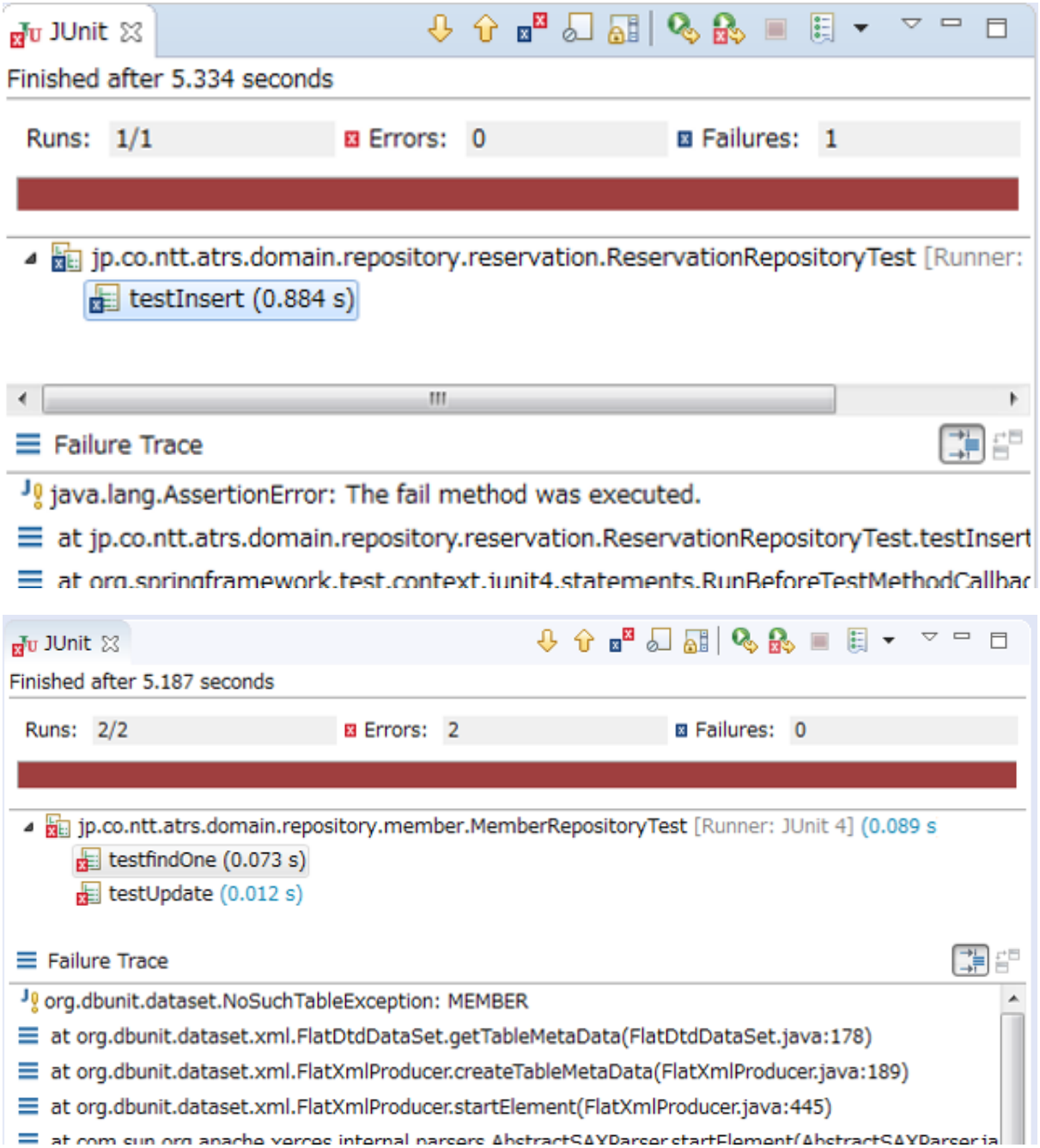


不具合なくテストが実行されれば、以下のような画面が表示される。



アサーションエラーの場合、以下のようにエラーメッセージが表示される。

テストを実行する上でエラーが発生した場合、以下のようにエラーメッセージが表示される。



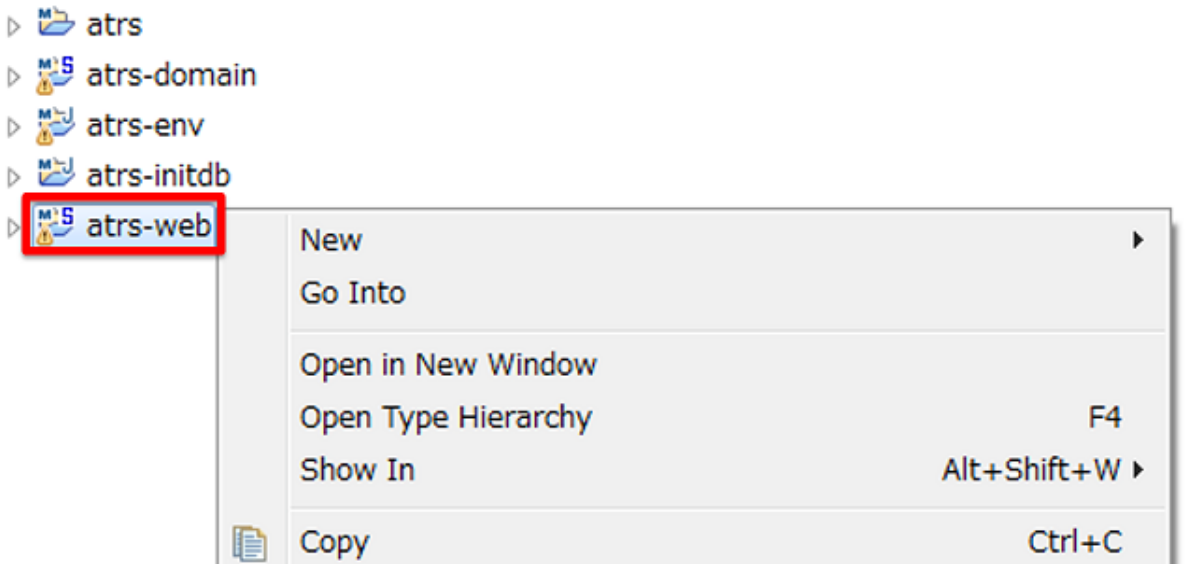
注釈: テストの失敗について

テストの失敗には大きく分けて 2 種類の要因がある。1 つはアサーション時に失敗する（Failure）場合、もう 1 つはテストの実装や、実行環境などに不備があり実行が失敗する（Error）場合である。IDE で JUnit を実行した際の実行結果画面には、これら 2 種類の結果の違いを示すだけでなく、スタックトレースも合わせて表示されるため、何が原因でテストが失敗したのかを分析する際に役立つ。

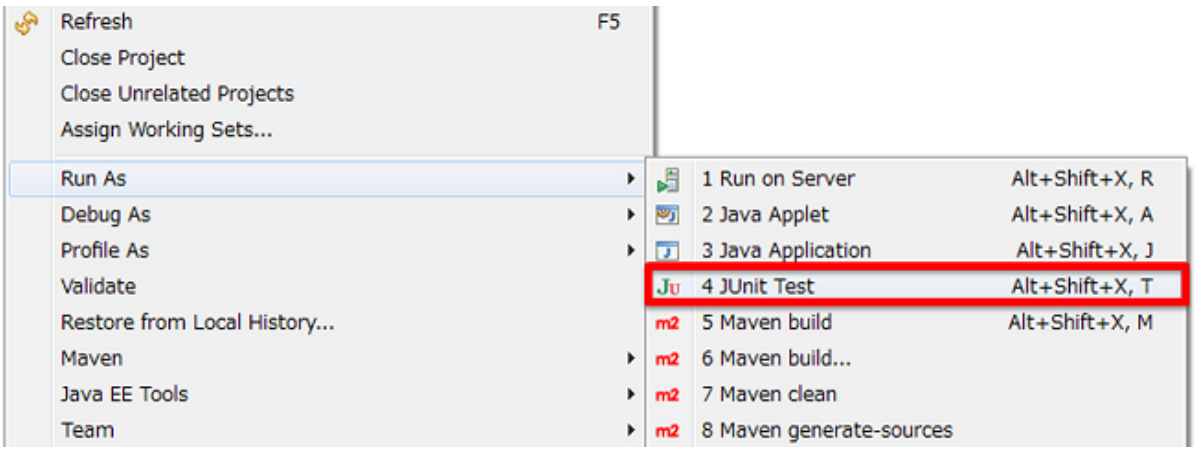
プロジェクト、メソッド単位で実行

JUnit の実行はプロジェクト単位、メソッド単位でも可能である。

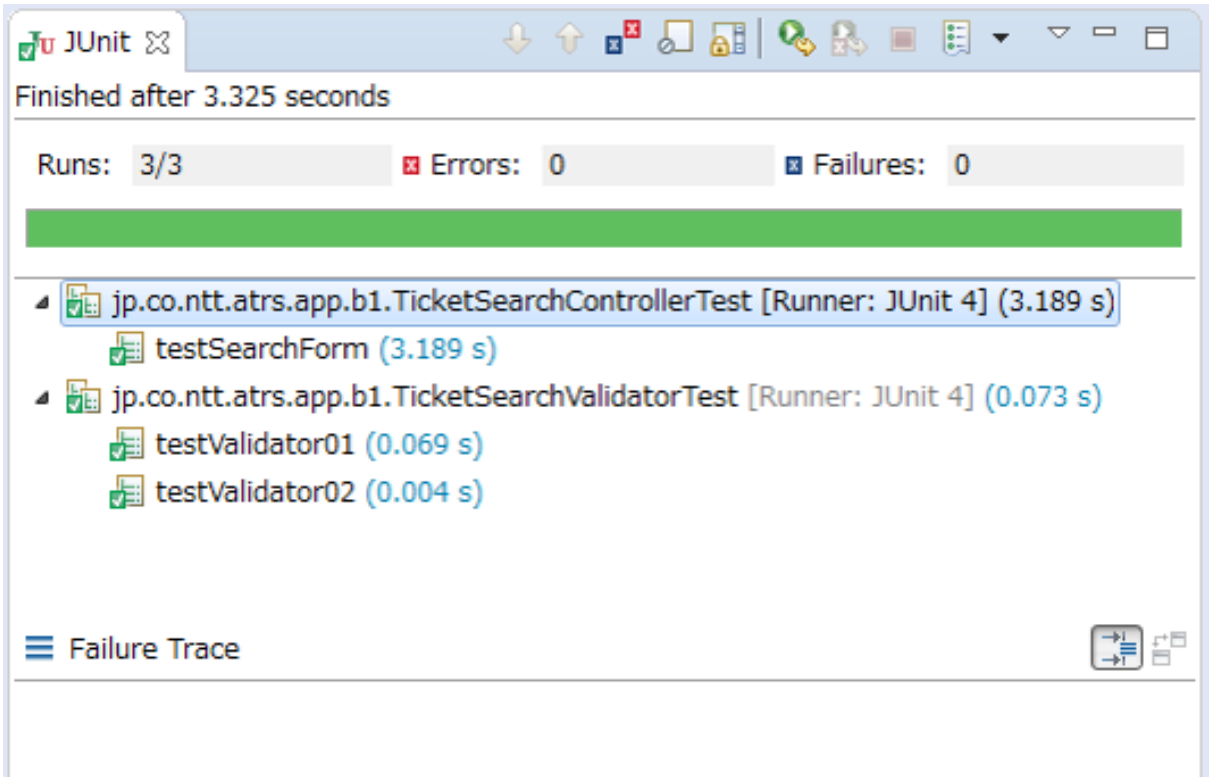
プロジェクト単位で実行する場合は、テストしたいプロジェクトを右クリックしてメニューを表示させる。



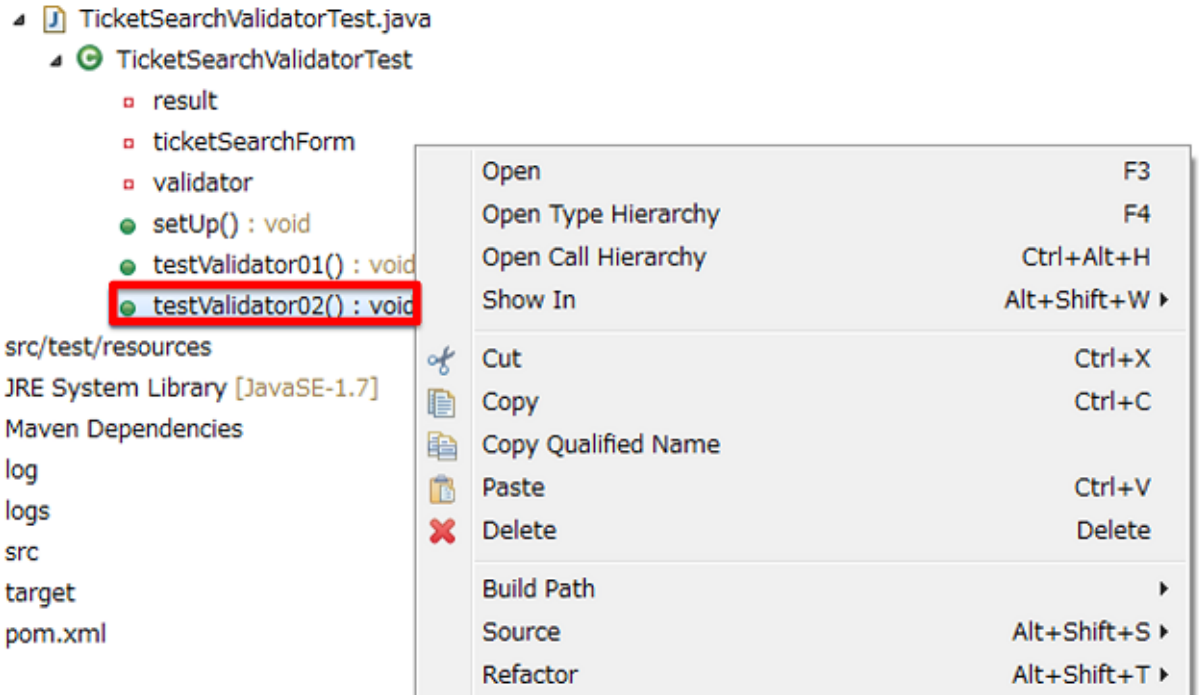
メニューから [Run As] -> [JUnit Test] を選択し、対象プロジェクトのテストを実行する。



プロジェクト単位で実行した場合は、選択したプロジェクトに含まれる全テストクラスの実行結果が表示される。



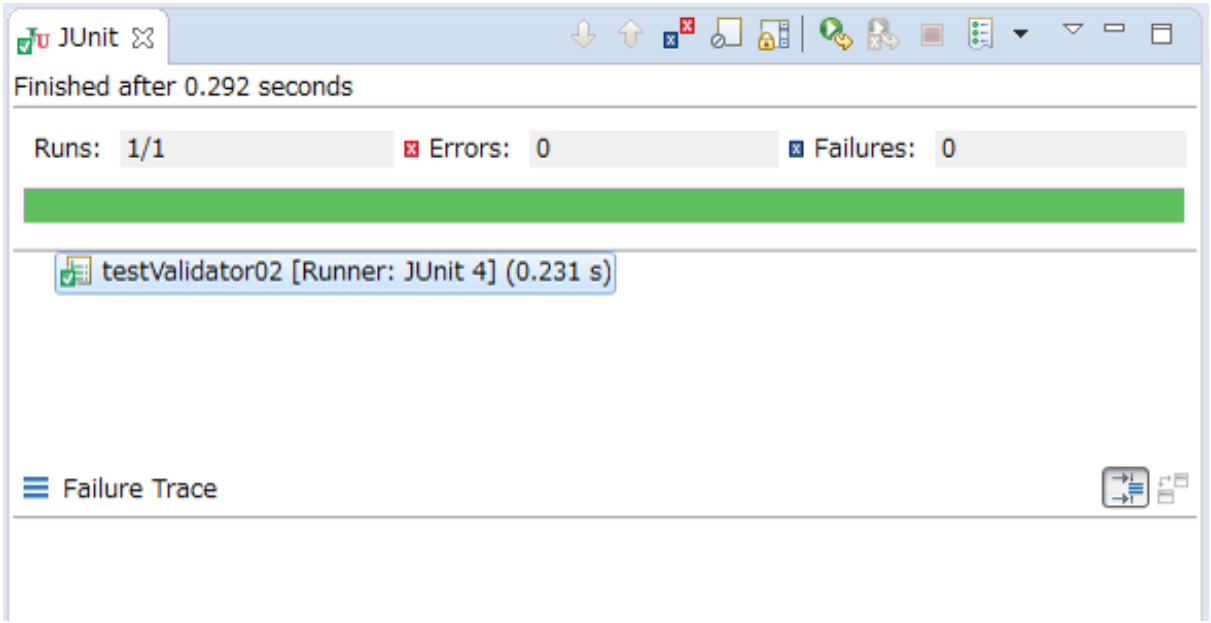
メソッド単位で実行する場合は、テストしたいメソッドを右クリックしてメニューを表示させる。



メニューから [Run As] -> [JUnit Test] を選択し、対象メソッドのテストを実行する。



メソッド単位で実行した場合は、選択したメソッドの実行結果のみが表示される。



Maven でテストを実行

Maven で JUnit を実行する方法を紹介する。

テストフェーズの実行

Maven で JUnit を実行する場合は、対象プロジェクト配下に移動し以下のコマンドを実行する。

```
mvn test
```

コマンドを実行すると、`target/classes` 配下に java コンパイルした.class ファイルを作成したのち、`target/test-classes` 配下にコンパイルしたテスト用.class ファイルを作成し、`target/surefire-reports` 配下にテスト結果が作成される。

デフォルトでは、以下のパターンにマッチするファイルが対象となりテストされる。

- `**/Test*.java`
- `**/*Test.java`
- `**/*Tests.java`
- `**/*TestCase.java`

上記パターンにマッチしないテストクラスを実行させたい場合は、`pom.xml` に設定を追加することで、テスト対象のファイルを変更することができる。また、テストファイルの除外についても設定することが可能である。

- `pom.xml`

```
<project>

  // ommited

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.20.1</version>
        <configuration>
          <includes>
            <include>*ServiceCheck.java</include> <!-- (1) -->
          </includes>
          <excludes>
            <exclude>AccountServiceCheck.java</exclude> <!-- (2) -->
          </excludes>
        </configuration>
      </plugin>
    </plugins>
  </build>

  // ommited

</project>
```

項番	説明
(1)	テスト実行時に実行対象となるファイルを設定する。
(2)	テスト実行時に除外対象となるファイルを設定する。

注釈: 設定する際には、正規表現を使って指定することもできる。詳細は maven-surefire-plugin (Regular Expression Support) を参照されたい。

コマンドオプションによる任意クラス、メソッドの指定

`mvn test` コマンドはオプションを用いることで任意のクラス、メソッドを指定し実行することもできる。

テスト対象のクラスを指定する場合は、以下のコマンドを用いて指定できる。

```
mvn test -Dtest=[クラス名]
```

「,」 区切りで複数クラスを指定することもできる。

```
mvn test -Dtest=[クラス名],[クラス名],[クラス名]...
```

テスト対象のメソッドを指定したい場合は、以下のコマンドを用いて指定できる。

```
mvn test -Dtest=[クラス名]#[メソッド名]
```

クラス名には、FQCN 指定（`com.example.domain.repository.MemberRepositoryTest` など）と単純クラス名での指定（`MemberRepositoryTest` など）のどちらで指定してもよい。また、クラス名にワイルドカード（`Member*Test` など）を用いてパターン指定することもできる。

警告: メソッド単位の指定は `maven-surefire-plugin` のバージョンが 2.7.3 以上必要となる。詳細は maven-surefire-plugin (Running a Set of Methods in a Single Test Class) を参照されたい。

注釈: オプションに `-Dmaven.test.skip=true` を指定することでテストのコンパイル・実行をスキップすることができる。実行のみスキップしたい場合は、`-DskipTests=true` を指定することでコンパイルのみ行われるようにすることもできる。

第 11 章

チュートリアル

11.1 チュートリアル (Todo アプリケーション)

11.1.1 はじめに

このチュートリアルで学ぶこと

- TERASOLUNA Server Framework for Java (5.x) による基本的なアプリケーションの開発方法
- Maven および STS(Eclipse) プロジェクトの構築方法
- TERASOLUNA Server Framework for Java (5.x) の アプリケーションのレイヤ化 に従った開発方法

対象読者

- Spring の DI や AOP に関する基礎的な知識がある
- Servlet/JSP を使用して Web アプリケーションを開発したことがある
- SQL に関する知識がある

検証環境

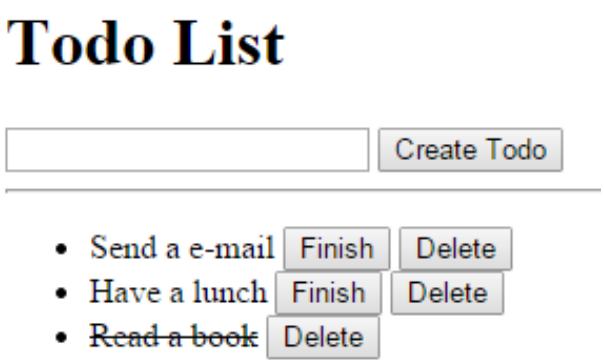
このチュートリアルは以下の環境で動作確認している。他の環境で実施する際は本書をベースに適宜読み替えて設定していくこと。

種別	名前
OS	Windows 7
JVM	Java 1.8
IDE	Spring Tool Suite 3.6.4.RELEASE (以降「STS」と呼ぶ)
Build Tool	Apache Maven 3.3.9 (以降「Maven」と呼ぶ)
Application Server	Pivotal tc Server Developer Edition v3.1 (STS に同封)
Web Browser	Google Chrome 46.0.2490.80 m

11.1.2 作成するアプリケーションの説明

アプリケーションの概要

TODO を管理するアプリケーションを作成する。TODO の一覧表示、TODO の登録、TODO の完了、TODO の削除を行える。



アプリケーションの業務要件

アプリケーションの業務要件は、以下の通りとする。

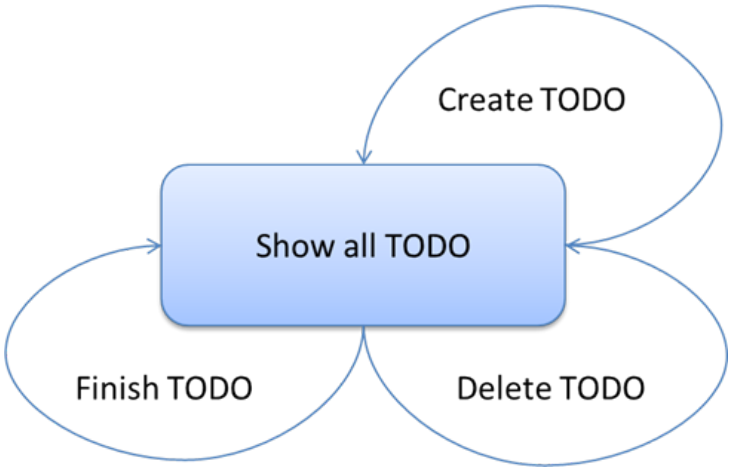
ルール ID	説明
B01	未完了の TODO は 5 件までしか登録できない
B02	完了済みの TODO は完了できない

注釈: 本要件は学習のためのもので、現実的な TODO 管理アプリケーションとしては適切ではない。

アプリケーションの処理仕様

アプリケーションの処理仕様と画面遷移は、以下の通りとする。

項番	プロセス名	HTTP メソッド	URL	備考
1	Show all TODO	-	/todo/list	
2	Create TODO	POST	/todo/create	作成処理終了後、Show all TODO へリダイレクト
3	Finish TODO	POST	/todo/finish	完了処理終了後、Show all TODO へリダイレクト
4	Delete TODO	POST	/todo/delete	削除処理終了後、Show all TODO へリダイレクト



Show all TODO

- TODO を全件表示する
- 未完了の TODO に対しては「Finish」と「Delete」用のボタンが付く
- 完了の TODO は打ち消し線で装飾する
- TODO の件名のみ表示する

Create TODO

- フォームから送信された TODO を保存する
- TODO の件名は 1 文字以上 30 文字以下であること
- アプリケーションの業務要件 の B01 を満たさない場合はエラーコード E001 でビジネス例外をスローする
- 処理が成功した場合は、遷移先の画面で「Created successfully!」を表示する

Finish TODO

- フォームから送信された `todoId` に対応する TODO を完了済みにする
- 該当する TODO が存在しない場合はエラーコード E404 でリソース未検出例外をスローする
- アプリケーションの業務要件 の B02 を満たさない場合はエラーコード E002 でビジネス例外をスローする
- 処理が成功した場合は、遷移先の画面で「Finished successfully!」を表示する

Delete TODO

- フォームから送信された `todoId` に対応する TODO を削除する
- 該当する TODO が存在しない場合はエラーコード E404 でリソース未検出例外をスローする
- 処理が成功した場合は、遷移先の画面で「Deleted successfully!」を表示する

エラーメッセージ一覧

エラーメッセージとして、以下の 3 つを定義する。

エラーコード	メッセージ	置換パラメータ
E001	[E001] The count of un-finished Todo must not be over {0}.	{0}… max unfinished count
E002	[E002] The requested Todo is already finished. (id={0})	{0}… todoId
E404	[E404] The requested Todo is not found. (id={0})	{0}… todoId

11.1.3 環境構築

本チュートリアルでは、インフラストラクチャ層の RepositoryImpl の実装として、

- データベースを使用せず `java.util.Map` を使ったインメモリ実装の RepositoryImpl
- MyBatis3 を使用してデータベースにアクセスする RepositoryImpl
- Spring Data JPA を使用してデータベースにアクセスする RepositoryImpl

の 3 種類を用意している。用途に応じていずれかを選択する。

チュートリアルの進行上、まずはインメモリ実装を試し、その後 MyBatis3 または Spring Data JPA を選ぶのが円滑である。

プロジェクトの作成

まず、`mvn archetype:generate` を利用して、実装するインフラストラクチャ層向けのブランクプロジェクトを作成する。ここでは、Windows のコマンドプロンプトを使用してブランクプロジェクトを作成する手順となっている。

注釈: インターネット接続するために、プロキシサーバーを介する必要がある場合、以下の作業を行うため、STS の Proxy 設定と、 Maven の Proxy 設定が必要である。

ちなみに: Bash 上で `mvn archetype:generate` を実行する場合は、以下のように」^」 を」\」 に置き換えて実行すればよい。

```
mvn archetype:generate -B\
 -DarchetypeGroupId=org.terasoluna.gfw.blank\
```

```
-DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype\
-DarchetypeVersion=5.4.2.RELEASE\
-DgroupId=com.example.todo\
-DartifactId=todo\
-Dversion=1.0.0-SNAPSHOT
```

O/R Mapper に依存しないブランクプロジェクトの作成

データベースを使用せず java.util.Map を使ったインメモリ実装の RepositoryImpl 用のプロジェクトを作成する場合は、以下のコマンドを実行して O/R Mapper に依存しないブランクプロジェクトを作成する。本チュートリアルを順序通り読み進める場合は、まずはこの方法でプロジェクトを作成すること。

```
mvn archetype:generate -B^
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype^
 -DarchetypeVersion=5.4.2.RELEASE^
 -DgroupId=com.example.todo^
 -DartifactId=todo^
 -Dversion=1.0.0-SNAPSHOT
```

MyBatis3 用のブランクプロジェクトの作成

MyBatis3 を使用してデータベースにアクセスする RepositoryImpl 用のプロジェクトを作成する場合は、以下のコマンドを実行して MyBatis3 用のブランクプロジェクトを作成する。このプロジェクト作成方法は *MyBatis3 を使用したインフラストラクチャ層の作成*で使用する。

```
mvn archetype:generate -B^
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-mybatis3-archetype^
 -DarchetypeVersion=5.4.2.RELEASE^
 -DgroupId=com.example.todo^
 -DartifactId=todo^
 -Dversion=1.0.0-SNAPSHOT
```

JPA 用のブランクプロジェクトの作成

Spring Data JPA を使用してデータベースへアクセスする RepositoryImpl 用のプロジェクトを作成する場合は、以下のコマンドを実行して JPA 用のブランクプロジェクトを作成する。このプロジェクト作成方法は *Spring Data JPA を使用したインフラストラクチャ層の作成*で使用する。

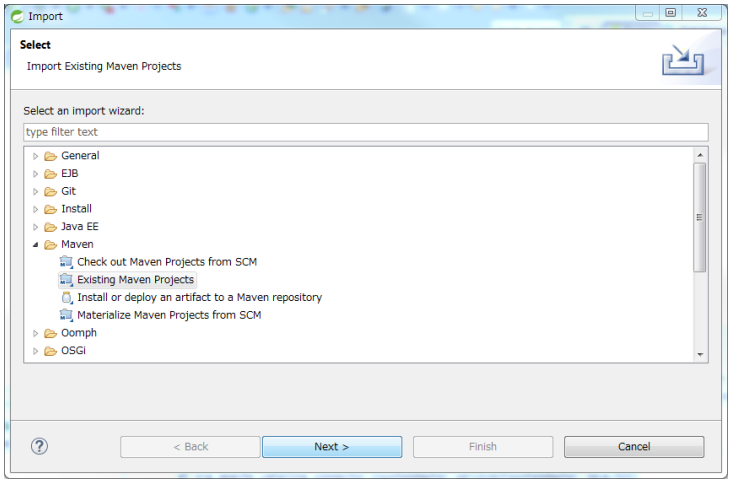
```
mvn archetype:generate -B^
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-jpa-archetype^
 -DarchetypeVersion=5.4.2.RELEASE^
```

```
-DgroupId=com.example.todo^
-DartifactId=todo^
-Dversion=1.0.0-SNAPSHOT
```

プロジェクトのインポート

作成したブランクプロジェクトを STS へインポートする。

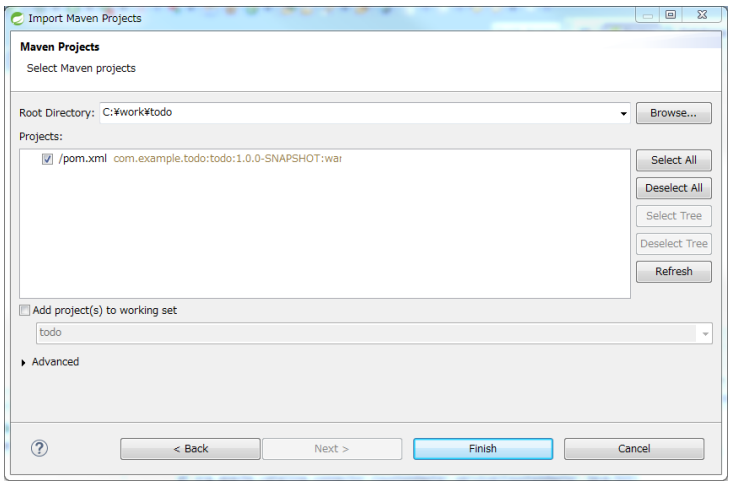
STS のメニューから、[File] -> [Import] -> [Maven] -> [Existing Maven Projects] -> [Next] を選択し、archetype で作成したプロジェクトを選択する。

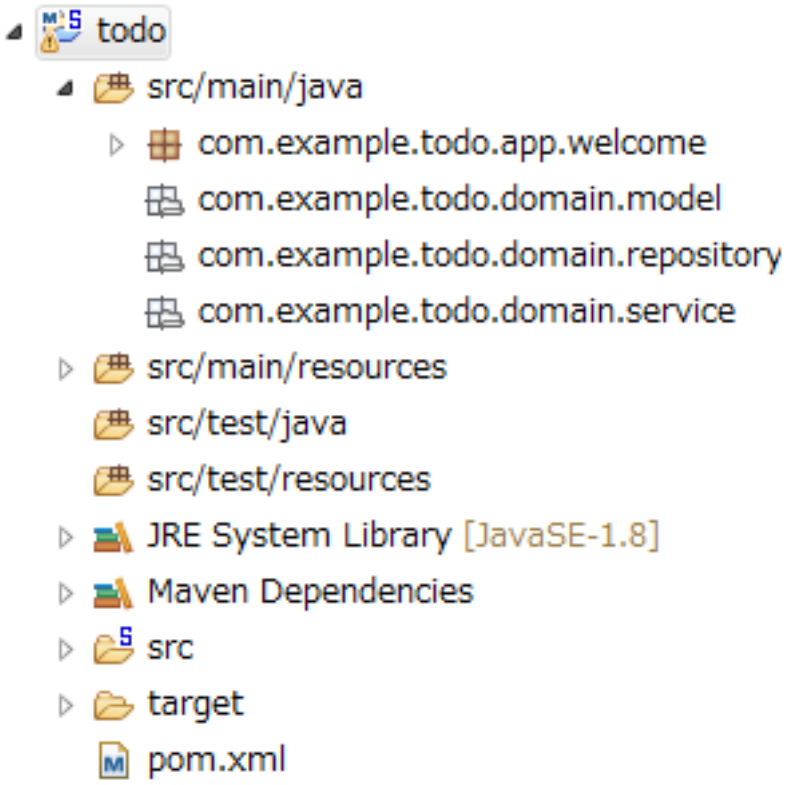


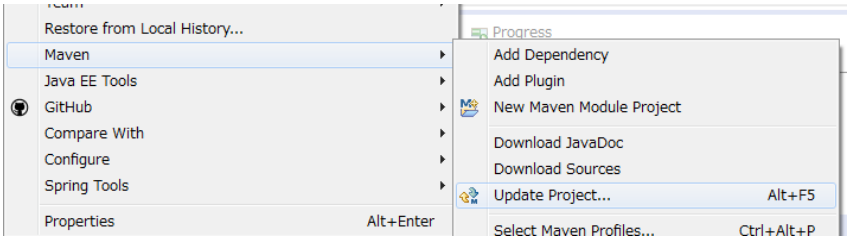
Root Directory に `C:\work\todo` を設定し、Projects に todo の pom.xml が選択された状態で、 [Finish] を押下する。

インポートが完了すると、Package Explorer に次のようなプロジェクトが表示される。

注釈: インポート後にビルドエラーが発生する場合は、プロジェクト名を右クリックし、「Maven」->「Update Project…」をクリックし、「OK」ボタンをクリックすることでエラーが解消されるケースがある。

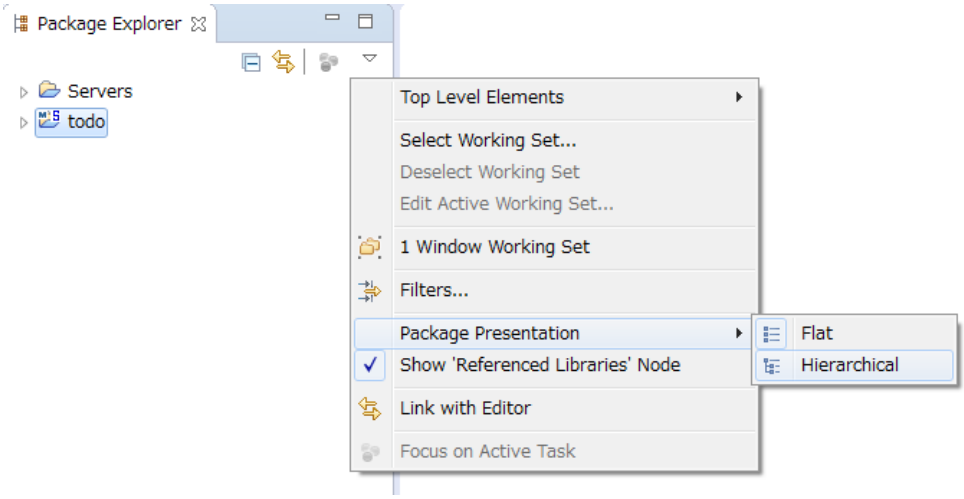

Import Maven Projects
Maven Projects
Select Maven projects
Root Directory: C:¥work¥todo
Browse...
Projects:
/pom.xml com.example.todo:todo:1.0.0-SNAPSHOT:war
Select All
Deselect All
Select Tree
Deselect Tree
Refresh
Add project(s) to working set
todo
Advanced
< Back
Next >
Finish
Cancel


todo
src/main/java
com.example.todo.app.welcome
com.example.todo.domain.model
com.example.todo.domain.repository
com.example.todo.domain.service
src/main/resources
src/test/java
src/test/resources
JRE System Library [JavaSE-1.8]
Maven Dependencies
src
target
pom.xml

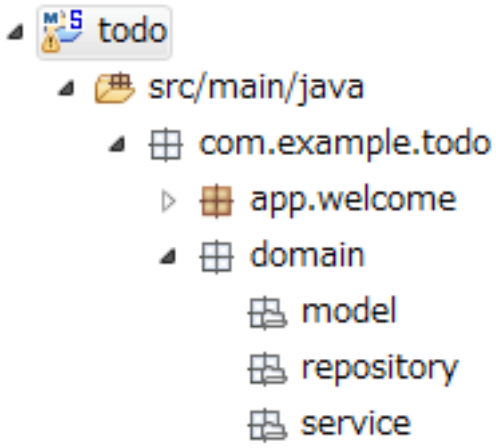

Restore from Local History...
Maven
Java EE Tools
GitHub
Compare With
Configure
Spring Tools
Properties Alt+Enter
Progress
Add Dependency
Add Plugin
New Maven Module Project
Download JavaDoc
Download Sources
Update Project... Alt+F5
Select Maven Profiles... Ctrl+Alt+P

ちなみに: パッケージの表示形式は、デフォルトは「Flat」だが、「Hierarchical」にしたほうが見通しがよい。

Package Explorer の「View Menu」 (右端の下矢印) をクリックし、「Package Presentation」->「Hierarchical」を選択する。



Package Presentation を Hierarchical にすると、以下の様な表示になる。



警告: O/R Mapper を使用するブランクプロジェクトの場合、H2 Database が dependency として定義されているが、この設定は簡易的なアプリケーションを簡単に作成するためのものであり、実際のアプリケーション開発で使用されることは想定していない。

以下の定義は、実際のアプリケーション開発を行う際は削除すること。

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

プロジェクトの構成

本チュートリアルで作成するプロジェクトの構成を以下に示す。

注釈: 前節の「プロジェクト構成」ではマルチプロジェクトにすることを推奨していたが、本チュートリアルでは、学習容易性を重視しているためシングルプロジェクト構成にしている。

ただし、実プロジェクトで適用する場合は、マルチプロジェクト構成を強く推奨する。

マルチプロジェクトの作成方法は、「*Web* アプリケーション向け開発プロジェクトの作成」を参照されたい。

[O/R Mapper に依存しないブランクプロジェクト、JPA 用のブランクプロジェクト用を作成した場合の構成]

```
src
  └main
      ├java
      |  └com
      |    └example
      |      └todo
      |        ├ app ... (1)
      |        |   └todo
      |        └domain ... (2)
      |            ├model ... (3)
      |            ├repository ... (4)
      |            |   └todo
```

```
      |             └service ... (5)
      |                 └todo
      ├resources
      |   └META-INF
      |       └spring ... (6)
      └wepapp
          └WEB-INF
              └views ... (7)
```

項番	説明
(1)	アプリケーション層のクラスを格納するパッケージ。 本チュートリアルでは、Todo 管理業務用のクラスを格納するためのパッケージを作成する。
(2)	ドメイン層のクラスを格納するパッケージ。
(3)	Domain Object を格納するパッケージ。
(4)	Repository を格納するパッケージ。 本チュートリアルでは、Todo オブジェクト (Domain Object) 用の Repository を格納するためのパッケージを作成する
(5)	Service を格納するパッケージ。 本チュートリアルでは、Todo 管理業務用の Service を格納するためのパッケージを作成する。
(6)	spring 関連の設定ファイルを格納するディレクトリ。
(7)	jsp を格納するディレクトリ。

[MyBatis3 用のブランクプロジェクトを作成した場合の構成]

```
src
  └main
      ├java
```

```
    |   └com
    |     └example
    |       └todo
    |         ├ app
    |         |   └todo
    |         └domain
    |             ├model
    |             ├repository
    |             |   └todo
    |             └service
    |                 └todo
    ├resources
    |   ├META-INF
    |   |   ├mybatis ... (8)
    |   |   └spring
    |   └com
    |     └example
    |       └todo
    |         └domain
    |             └repository ... (9)
    |                 └todo
    └wepapp
        └WEB-INF
            └views
```

項番	説明
(8)	MyBatis 関連の設定ファイルを格納するディレクトリ。
(9)	SQL を記述する MyBatis の Mapper ファイルを格納するディレクトリ。 本チュートリアルでは、Todo オブジェクト用の Repository の Mapper ファイルを格納するためのディレクトリを作成する。

設定ファイルの確認

チュートリアルを進める上で必要となる設定の多くは、作成したブランクプロジェクトに既に設定済みの状態である。

チュートリアルを実施するだけであれば、これらの設定の理解は必須ではないが、アプリケーションを動かすためにどのような設定が必要なのかを理解しておくことを推奨する。

アプリケーションを動かすために必要な設定 (設定ファイル) の解説については、「設定ファイルの解説」を参照されたい。

注釈: まず、手を動かして Todo アプリケーションを作成したい場合は、設定ファイルの確認は読み飛ばしてもよいが、Todo アプリケーションを作成した後に一読して頂きたい。

プロジェクトの動作確認

Todo アプリケーションの開発を始める前に、プロジェクトの動作確認を行う。

ブランクプロジェクトでは、トップページを表示するための Controller と JSP の実装が用意されているため、トップページを表示する事で動作確認を行う事ができる。

ブランクプロジェクトから提供されている Controller(`src/main/java/com/example/todo/app/welcome/HelloController.java`) は、以下のような実装となっている。

```
package com.example.todo.app.welcome;

import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 * Handles requests for the application home page.
 */
// (1)
@Controller
public class HelloController {

    // (2)
    private static final Logger logger = LoggerFactory
            .getLogger(HelloController.class);

    /**
     * Simply selects the home view to render by returning its name.
     */
    // (3)
    @RequestMapping(value = "/", method = {RequestMethod.GET, RequestMethod.POST})
    public String home(Locale locale, Model model) {
        // (4)
        logger.info("Welcome home! The client locale is {}.", locale);
```

```
        Date date = new Date();
        DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG,
                DateFormat.LONG, locale);

        String formattedDate = dateFormat.format(date);

        // (5)
        model.addAttribute("serverTime", formattedDate);

        // (6)
        return "welcome/home";
    }

}
```

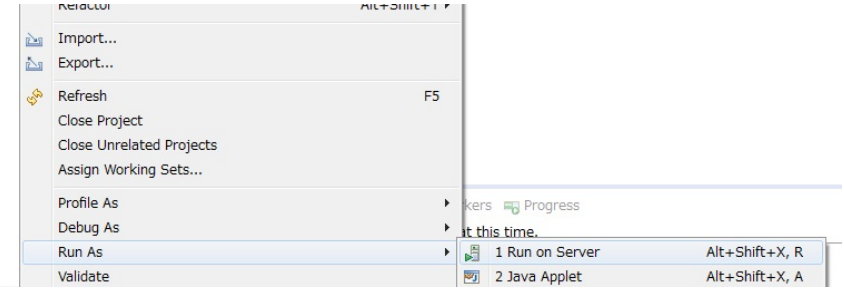
項番	説明
(1)	Controller として component-scan の対象とするため、クラスレベルに`@Controller` アノテーションが付与している。
(2)	(4) でログ出力するためのロガーを生成している。 ロガーの実装は logback のものであるが、API は SLF4J の `org.slf4j.Logger` を使用している。
(3)	`@RequestMapping` アノテーションを使用して、」/」 (ルート) へのアクセスに対するメソッドとしてマッピングを行っている。
(4)	メソッドが呼ばれたことを通知するためのログを info レベルで出力している。
(5)	画面に表示するための日付文字列を、`serverTime` という属性名で Model に設定している。
(6)	view 名として `welcome/home` を返す。`ViewResolver` の設定により、`WEB-INF/views/welcome/home.jsp` が呼び出される。

ブランクプロジェクトから提供されている JSP(src/main/webapp/WEB-INF/views/welcome/home.jsp) は、以下のような実装となっている。

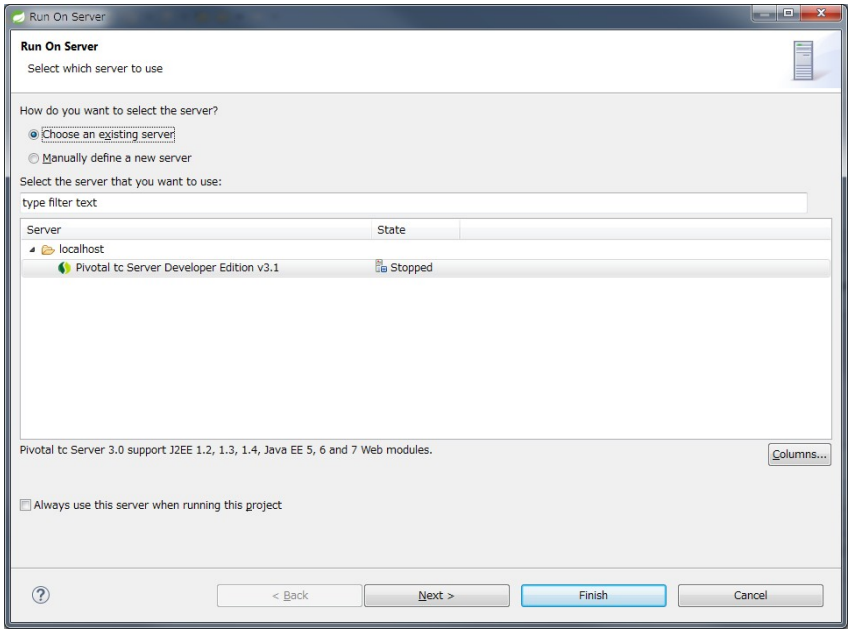
```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Home</title>
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/css/
↪styles.css">
</head>
<body>
    <div id="wrapper">
        <h1>Hello world!</h1>
        <!-- (7) -->
        <p>The time on the server is ${serverTime}.</p>
    </div>
</body>
</html>
```

項番	説明
(7)	Controller で Model に設定した serverTime を表示する。 ここでは、XSS 対策を行っていないが、ユーザの入力値を表示する場合は、f:h() 関数を用いて、必ず XSS 対策を行うこと。

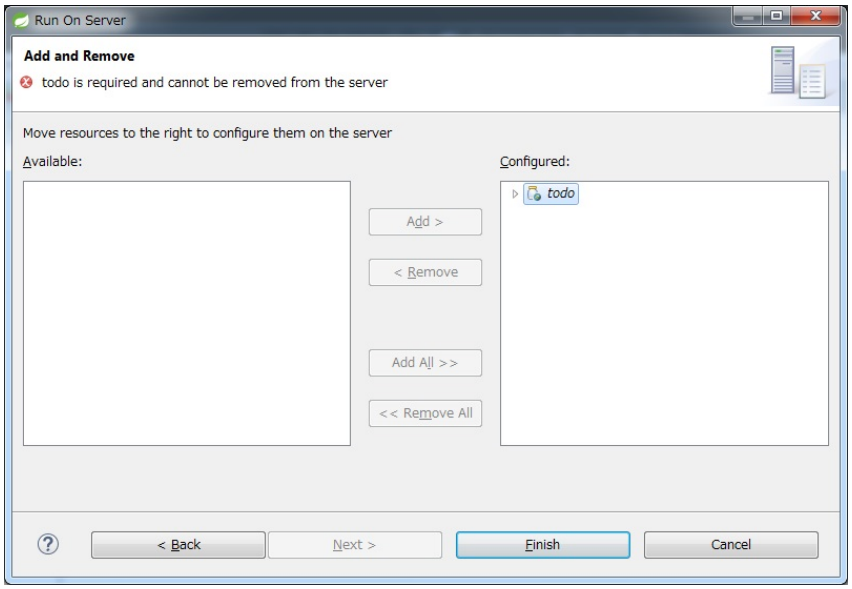
プロジェクトを右クリックして「Run As」->「Run on Server」を選択する。



AP サーバー (Pivotal tc Server Developer Edition v3.1) を選択し、「Next」をクリックする。



todo が「Configured」に含まれていることを確認して「Finish」をクリックしてサーバーを起動する。



起動すると以下のようなログが出力される。/というパスに対して `com.example.todo.app.welcome.HelloController` の hello メソッドがマッピングされていることが分かる。

```
 date:2016-02-17 11:25:30    thread:localhost-startStop-1    X-Track:
↪level:INFO      logger:o.springframework.web.servlet.DispatcherServlet
↪message:FrameworkServlet 'appServlet': initialization started
 date:2016-02-17 11:25:31    thread:localhost-startStop-1    X-Track:
↪level:DEBUG     logger:o.t.gfw.web.codelist.CodeListInterceptor
↪message:registered codeList : []
 date:2016-02-17 11:25:31    thread:localhost-startStop-1    X-Track:
↪level:INFO      logger:o.s.w.s.m.m.a.RequestMappingHandlerMapping
↪message:Mapped "{[/],methods=[GET || POST],params=[],headers=[],consumes=[],
↪produces=[],custom=[]}" onto public java.lang.String com.example.todo.app.
↪welcome.HelloController.home(java.util.Locale,org.springframework.ui.Model)
 date:2016-02-17 11:25:31    thread:localhost-startStop-1    X-Track:
↪level:INFO      logger:o.s.w.s.m.m.a.RequestMappingHandlerAdapter
↪message:Looking for @ControllerAdvice: WebApplicationContext for namespace
↪'appServlet-servlet': startup date [Wed Feb 17 11:25:30 JST 2016]; parent: Root
↪WebApplicationContext
 date:2016-02-17 11:25:32    thread:localhost-startStop-1    X-Track:
↪level:INFO      logger:o.s.w.s.m.m.a.RequestMappingHandlerAdapter
↪message:Looking for @ControllerAdvice: WebApplicationContext for namespace
↪'appServlet-servlet': startup date [Wed Feb 17 11:25:30 JST 2016]; parent: Root
↪WebApplicationContext
 date:2016-02-17 11:25:32    thread:localhost-startStop-1    X-Track:
↪level:INFO      logger:o.s.web.servlet.handler.SimpleUrlHandlerMapping
↪message:Mapped URL path [/**] onto handler 'org.springframework.web.servlet.
↪resource.DefaultServletHttpRequestHandler#0'
 date:2016-02-17 11:25:32    thread:localhost-startStop-1    X-Track:
↪level:INFO      logger:o.s.web.servlet.handler.SimpleUrlHandlerMapping
↪message:Mapped URL path [/resources/**] onto handler 'org.springframework.web.
↪servlet.resource.ResourceHttpRequestHandler#0'
 date:2016-02-17 11:25:33    thread:localhost-startStop-1    X-Track:
↪level:INFO      logger:o.springframework.web.servlet.DispatcherServlet
↪message:FrameworkServlet 'appServlet': initialization completed in 2826 ms
```

ブラウザで http://localhost:8080/todo にアクセスすると、以下のように表示される。



コンソールを見ると、

- 共通ライブラリから提供している `TraceLoggingInterceptor` の TRACE ログ

- Controller で実装した INFO ログ

が出力されていることがわかる。

```
date:2016-02-17 11:25:35    thread:tomcat-http--11  X-
↪Track:b49b630274974bffbcd9e8d13261f6a7        level:TRACE     logger:o.t.gfw.web.
↪logging.TraceLoggingInterceptor      message:[START CONTROLLER] HelloController.
↪home(Locale,Model)
date:2016-02-17 11:25:35    thread:tomcat-http--11  X-
↪Track:b49b630274974bffbcd9e8d13261f6a7        level:INFO      logger:com.example.
↪todo.app.welcome.HelloController                      message:Welcome home! The␣
↪client locale is ja_JP.
date:2016-02-17 11:25:35    thread:tomcat-http--11  X-
↪Track:b49b630274974bffbcd9e8d13261f6a7        level:TRACE     logger:o.t.gfw.web.
↪logging.TraceLoggingInterceptor      message:[END CONTROLLER  ] HelloController.
↪home(Locale,Model)-> view=welcome/home, model={serverTime=2016/02/17 11:25:35␣
↪JST}
date:2016-02-17 11:25:35    thread:tomcat-http--11  X-
↪Track:b49b630274974bffbcd9e8d13261f6a7        level:TRACE     logger:o.t.gfw.web.
↪logging.TraceLoggingInterceptor      message:[HANDLING TIME   ] HelloController.
↪home(Locale,Model)-> 97,346,576 ns
```

注釈: `TraceLoggingInterceptor` は Controller の開始、終了でログを出力する。終了時には `View` と `Model` の情報および処理時間が出力される。

11.1.4 Todo アプリケーションの作成

Todo アプリケーションを作成する。作成する順は、以下の通りである。

- ドメイン層 (+ インフラストラクチャ層)
- Domain Object 作成
- Repository 作成
- RepositoryImpl 作成
- Service 作成
- アプリケーション層
- Controller 作成
- Form 作成

- View 作成

RepositoryImpl の作成は、選択したインフラストラクチャ層の種類に応じて実装方法が異なる。

ここでは、データベースを使用せず `java.util.Map` を使ったインメモリ実装の RepositoryImpl を作成する方法について説明を行う。データベースを使用する場合は、「データベースアクセスを伴うインフラストラクチャ層の作成」に記載されている内容で読み替えて、Todo アプリケーションを作成して頂きたい。

ドメイン層の作成

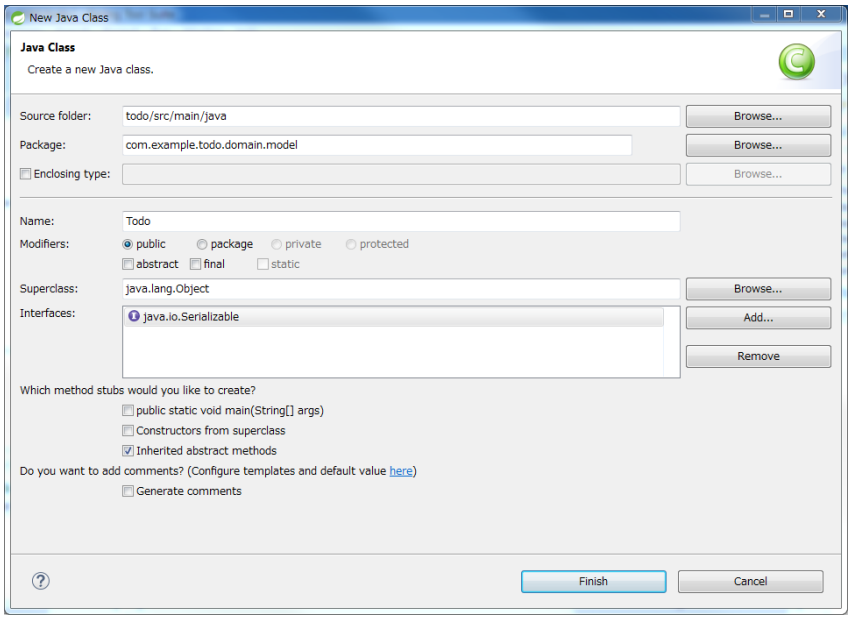
Domain Object の作成

Domain オブジェクトを作成する。

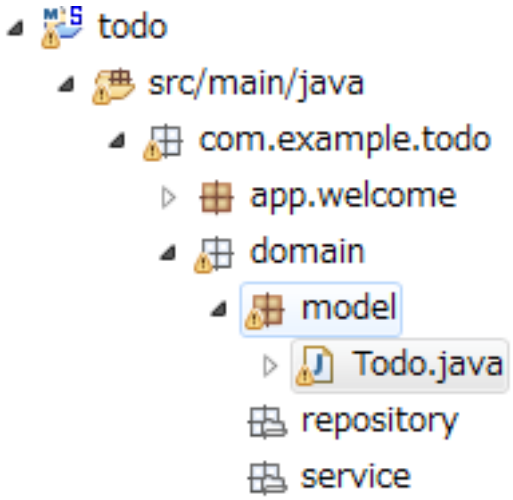
Package Explorer 上で右クリック -> New -> Class を選択し、「New Java Class」ダイアログを表示し、

項番	項目	入力値
1	Package	`com.example.todo.domain.model`
2	Name	`Todo`
3	Interfaces	`java.io.Serializable`

を入力して「Finish」する。



作成したクラスは以下のディレクトリに格納される。



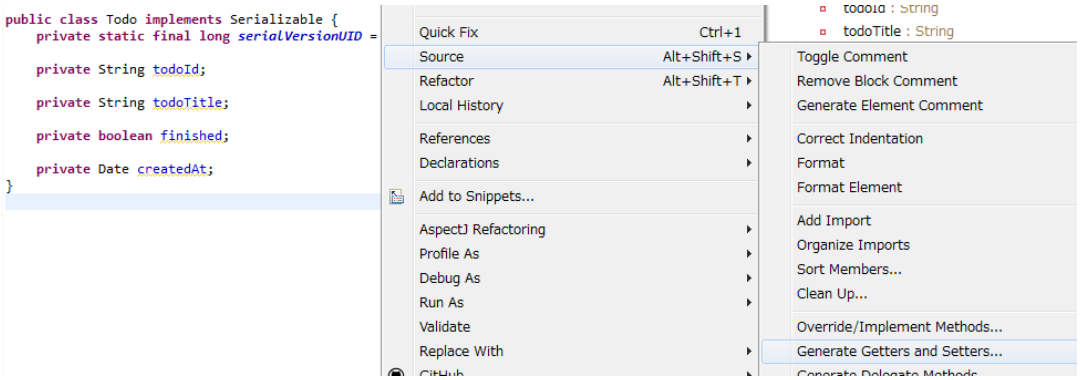
作成したクラスに以下のプロパティを追加する。

- ID → todoId
- タイトル → todoTitle
- 完了フラグ → finished
- 作成日 → createdAt

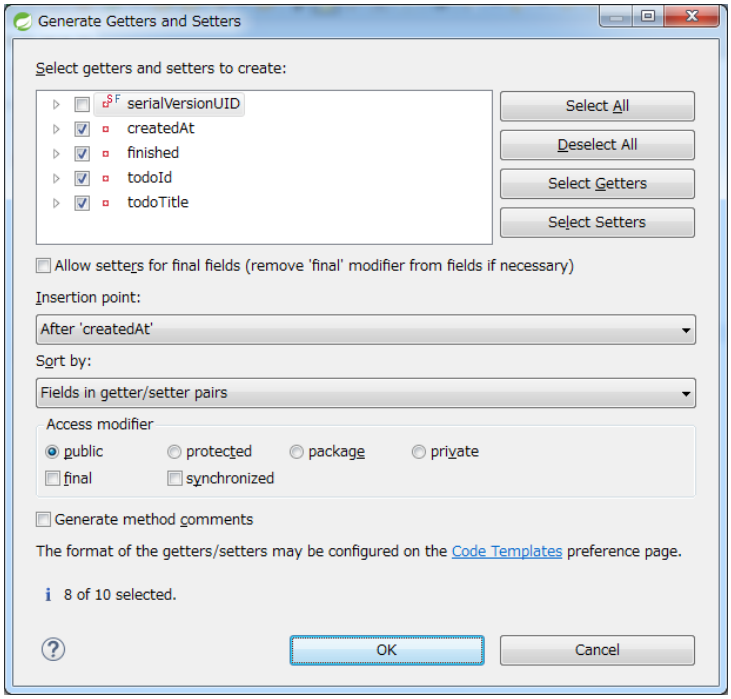
```
package com.example.todo.domain.model;

import java.io.Serializable;
import java.util.Date;

public class Todo implements Serializable {

    private static final long serialVersionUID = 1L;

    private String todoId;

    private String todoTitle;

    private boolean finished;

    private Date createdAt;

    public String getTodoId() {
        return todoId;
    }
```

```
    public void setTodoId(String todoId) {
        this.todoId = todoId;
    }

    public String getTodoTitle() {
        return todoTitle;
    }

    public void setTodoTitle(String todoTitle) {
        this.todoTitle = todoTitle;
    }

    public boolean isFinished() {
        return finished;
    }

    public void setFinished(boolean finished) {
        this.finished = finished;
    }

    public Date getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Date createdAt) {
        this.createdAt = createdAt;
    }
}
```

ちなみに: Getter/Setter メソッドは STS の機能を使って自動生成することができる。フィールドを定義した後、エディタ上で右クリックし、「Source」->「Generate Getter and Setters…」を選択する。



serialVersionUID 以外を選択して「OK」



Repository の作成

`TodoRepository` インタフェースを作成する。データベースを使用する場合は、「データベースアクセスを伴うインフラストラクチャ層の作成」に記載されている内容で読み替えて、Repository を作成する。

Package Explorer 上で右クリック -> New -> Interface を選択し、「New Java Interface」ダイアログを表示し、

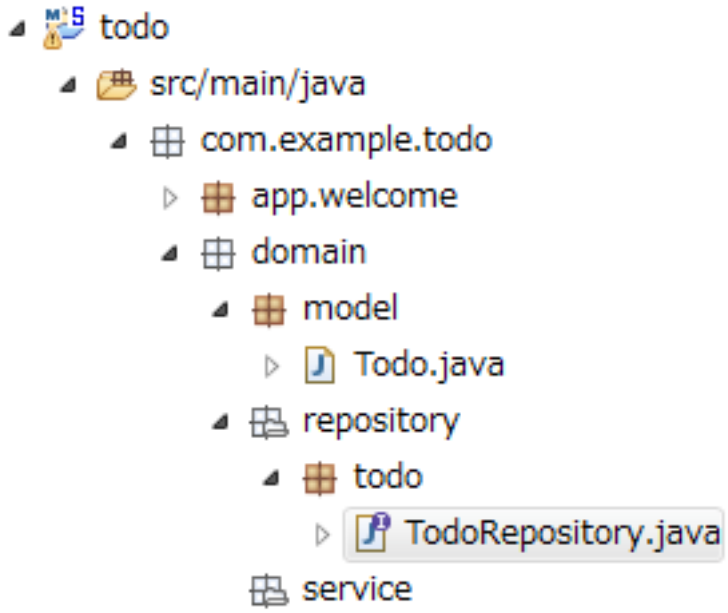
項番	項目	入力値
1	Package	`com.example.todo.domain.repository.todo`
2	Name	`TodoRepository`

を入力して「Finish」する。

作成したインタフェースは以下のディレクトリに格納される。

作成したインタフェースに、今回のアプリケーションで必要となる以下の CRUD 操作を行うメソッドを定義する。

- TODO の 1 件取得 → findOne
- TODO の全件取得 → findAll
- TODO の 1 件作成 → create
- TODO の 1 件更新 → update



- TODO の 1 件削除 → delete
- 完了済み TODO 件数の取得 → countByFinished

```
package com.example.todo.domain.repository.todo;

import java.util.Collection;

import com.example.todo.domain.model.Todo;

public interface TodoRepository {
    Todo findOne(String todoId);

    Collection<Todo> findAll();

    void create(Todo todo);

    boolean update(Todo todo);

    void delete(Todo todo);

    long countByFinished(boolean finished);
}
```

注釈: ここでは、TodoRepository の汎用性を上げるため、「完了済み件数を取得する」メソッド (long countFinished()) ではなく、「完了状態が xx である件数を取得する」メソッド (long countByFinished(boolean)) として定義している。

long countByFinished(boolean) の引数として true を渡すと「完了済みの件数」、false を渡すと「未完了の件数」が取得できる仕様としている。

RepositoryImpl の作成 (インフラストラクチャ層)

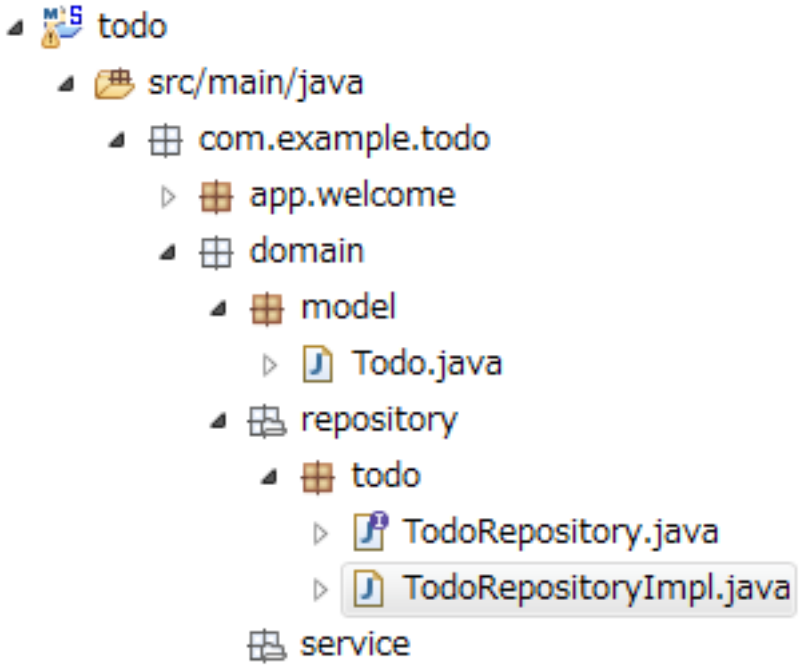
ここでは、説明を単純化するため、`java.util.Map` を使ったインメモリ実装の RepositoryImpl を作成する。データベースを使用する場合は、「データベースアクセスを伴うインフラストラクチャ層の作成」に記載されている内容で読み替えて、RepositoryImpl を作成する。

Package Explorer 上で右クリック -> New -> Class を選択し、「New Java Class」ダイアログを表示し、

項番	項目	入力値
1	Package	`com.example.todo.domain.repository.todo`
2	Name	`TodoRepositoryImpl`
3	Interfaces	`com.example.todo.domain.repository.todo.TodoRepository`

を入力して「Finish」する。

作成したクラスは以下のディレクトリに格納される。



作成したクラスに CRUD 操作を実装する。

注釈: RepositoryImpl には、業務ロジックは含めず、Domain オブジェクトの保存先への出し入れ (CRUD 操作) に終始することが実装ポイントである。

```
package com.example.todo.domain.repository.todo;

import java.util.Collection;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import org.springframework.stereotype.Repository;

import com.example.todo.domain.model.Todo;

@Repository // (1)
public class TodoRepositoryImpl implements TodoRepository {
    private static final Map<String, Todo> TODO_MAP = new ConcurrentHashMap<String,
↪ Todo>();

    @Override
    public Todo findOne(String todoId) {
        return TODO_MAP.get(todoId);
    }

    @Override
    public Collection<Todo> findAll() {
        return TODO_MAP.values();
    }

    @Override
    public void create(Todo todo) {
        TODO_MAP.put(todo.getTodoId(), todo);
    }

    @Override
    public boolean update(Todo todo) {
        TODO_MAP.put(todo.getTodoId(), todo);
        return true;
    }

    @Override
    public void delete(Todo todo) {
        TODO_MAP.remove(todo.getTodoId());
    }

    @Override
    public long countByFinished(boolean finished) {
        long count = 0;
        for (Todo todo : TODO_MAP.values()) {
            if (finished == todo.isFinished()) {
                count++;
            }
        }
        return count;
    }
}
```

項番	説明
(1)	Repository として component-scan 対象とするため、クラスレベルに`@Repository` アノテーションをつける。

注釈: 本チュートリアルでは、インフラストラクチャ層に属するクラス (RepositoryImpl) をドメイン層のパッケージ (`com.example.todo.domain`) に格納しているが、完全に層別にパッケージを分けるのであれば、インフラストラクチャ層のクラスは、`com.example.todo.infra` 以下に作成した方が良い。

ただし、通常のプロジェクトでは、インフラストラクチャ層が変更されることを前提としていない (そのような前提で進めるプロジェクトは、少ない)。そこで、作業効率向上のために、ドメイン層の Repository インタフェースと同じ階層に、RepositoryImpl を作成しても良い。

Service の作成

まず、`TodoService` インタフェースを作成する。

Package Explorer 上で右クリック -> New -> Interface を選択し、「New Java Interface」ダイアログを表示し、

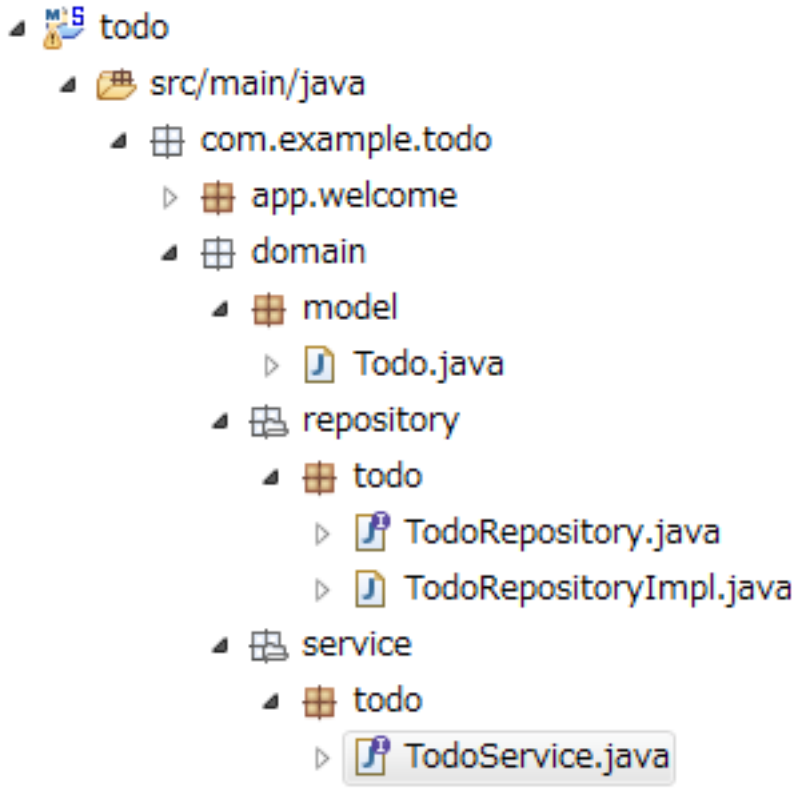
項番	項目	入力値
1	Package	`com.example.todo.domain.service.todo`
2	Name	`TodoService`

を入力して「Finish」する。

作成したインタフェースは以下のディレクトリに格納される。

作成したインタフェースに以下の業務処理を行うメソッドを定義する。

- Todo の全件取得 → findAll
- Todo の新規作成 → create
- Todo の完了 → finish
- Todo の削除 → delete




```
package com.example.todo.domain.service.todo;

import java.util.Collection;

import com.example.todo.domain.model.Todo;

public interface TodoService {
    Collection<Todo> findAll();

    Todo create(Todo todo);

    Todo finish(String todoId);

    void delete(String todoId);
}
```

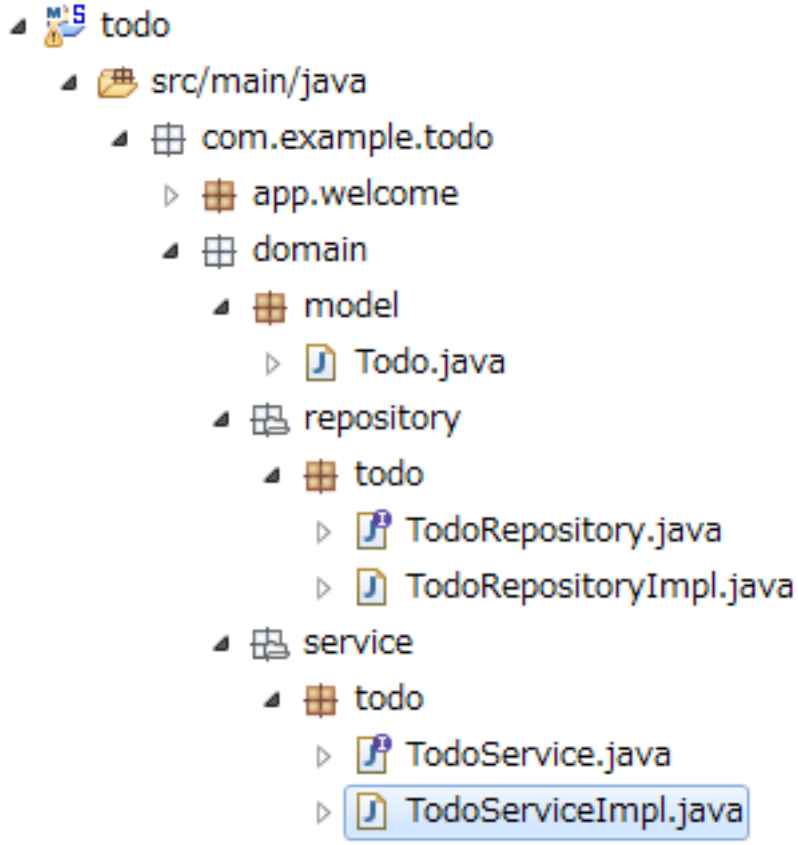
次に、TodoService インタフェースに定義したメソッドを実装する TodoServiceImpl クラスを作成する。

Package Explorer 上で右クリック -> New -> Class を選択し、「New Java Class」ダイアログを表示し、

項番	項目	入力値
1	Package	`com.example.todo.domain.service.todo`
2	Name	`TodoServiceImpl`
3	Interfaces	`com.example.todo.domain.service.todo.TodoService`

を入力して「Finish」する。

作成したクラスは以下のディレクトリに格納される。




```
package com.example.todo.domain.service.todo;

import java.util.Collection;
import java.util.Date;
import java.util.UUID;

import javax.inject.Inject;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.message.ResultMessage;
import org.terasoluna.gfw.common.message.ResultMessages;
```

```
import com.example.todo.domain.model.Todo;
import com.example.todo.domain.repository.todo.TodoRepository;

@Service// (1)
@Transactional // (2)
public class TodoServiceImpl implements TodoService {

    private static final long MAX_UNFINISHED_COUNT = 5;

    @Inject// (3)
    TodoRepository todoRepository;

    // (4)
    private Todo findOne(String todoId) {
        Todo todo = todoRepository.findOne(todoId);
        if (todo == null) {
            // (5)
            ResultMessages messages = ResultMessages.error();
            messages.add(ResultMessage
                    .fromText("[E404] The requested Todo is not found. (id="
                            + todoId + ")"));
            // (6)
            throw new ResourceNotFoundException(messages);
        }
        return todo;
    }

    @Override
    @Transactional(readOnly = true) // (7)
    public Collection<Todo> findAll() {
        return todoRepository.findAll();
    }

    @Override
    public Todo create(Todo todo) {
        long unfinishedCount = todoRepository.countByFinished(false);
        if (unfinishedCount >= MAX_UNFINISHED_COUNT) {
            ResultMessages messages = ResultMessages.error();
            messages.add(ResultMessage
                    .fromText("[E001] The count of un-finished Todo must not be␣
↪over "
                            + MAX_UNFINISHED_COUNT + "."));
            // (8)
            throw new BusinessException(messages);
        }

        // (9)
        String todoId = UUID.randomUUID().toString();
        Date createdAt = new Date();

        todo.setTodoId(todoId);
        todo.setCreatedAt(createdAt);
```

```
        todo.setFinished(false);

        todoRepository.create(todo);
        /* REMOVE THIS LINE IF YOU USE JPA
            todoRepository.save(todo); // 10
           REMOVE THIS LINE IF YOU USE JPA */

        return todo;
    }

    @Override
    public Todo finish(String todoId) {
        Todo todo = findOne(todoId);
        if (todo.isFinished()) {
            ResultMessages messages = ResultMessages.error();
            messages.add(ResultMessage
                    .fromText("[E002] The requested Todo is already finished. (id="
                            + todoId + ")"));
            throw new BusinessException(messages);
        }
        todo.setFinished(true);
        todoRepository.update(todo);
        /* REMOVE THIS LINE IF YOU USE JPA
            todoRepository.save(todo); // (11)
           REMOVE THIS LINE IF YOU USE JPA */
        return todo;
    }

    @Override
    public void delete(String todoId) {
        Todo todo = findOne(todoId);
        todoRepository.delete(todo);
    }
}
```

項番	説明
(1)	Service として component-scan の対象とするため、クラスレベルに`@Service` アノテーションをつける。

Continued on next page

表 11.1 – continued from previous page

項番	説明
(2)	クラスレベルに、`@Transactional` アノテーションをつけることで、公開メソッドをすべてトランザクション管理する。 アノテーションを付与することで、メソッド開始時にトランザクションを開始、メソッド正常終了時にトランザクションのコミットが行われる。 また、途中で非検査例外が発生した場合は、トランザクションがロールバックされる。 データベースを使用しない場合は、`@Transactional` アノテーションは不要である。
(3)	`@Inject` アノテーションで、`TodoRepository` の実装をインジェクションする。
(4)	1 件取得は、`finish` メソッドでも `delete` メソッドでも使用するため、メソッドとして用意しておく (interface に公開しても良い)。
(5)	結果メッセージを格納するクラスとして、共通ライブラリで用意されている `org.terasoluna.gfw.common.message.ResultMessage` を用いる。 今回は、エラーメッセージを例外に追加する際に、`ResultMessages.error()` でメッセージ種別を指定して、`ResultMessage` を追加している。
(6)	対象のデータが存在しない場合、共通ライブラリで用意されている `org.terasoluna.gfw.common.exception.ResourceNotFoundException` をスローする。
(7)	参照のみ行う処理に関しては、`readOnly=true` をつける。 O/R Mapper によっては、この設定により、参照時のトランザクション制御の最適化が行われる (JPA を使用する場合、効果はない)。 データベースを使用しない場合は、`@Transactional` アノテーションは不要である。
(8)	業務エラーが発生した場合、共通ライブラリで用意されている `org.terasoluna.gfw.common.exception.BusinessException` をスローする。

Continued on next page

表 11.1 – continued from previous page

項番	説明
(9)	一意性のある値を生成するために、UUID を使用している。データベースのシーケンスを用いてもよい。
(10)	Spring Data JPA を使用してデータベースにアクセスする場合は、`create` メソッドではなく、`save` メソッドを呼び出す。
(11)	Spring Data JPA を使用してデータベースにアクセスする場合は、`update` メソッドではなく、`save` メソッドを呼び出す。

注釈: 本節では、説明を単純化するため、エラーメッセージをハードコードしているが、メンテナンスの観点で本来は好ましくない。通常、メッセージは、プロパティファイルに外部化することが推奨される。プロパティファイルに外部化する方法は、プロパティ管理を参照されたい。

アプリケーション層の作成

ドメイン層の実装が完了したので、次はドメイン層を利用して、アプリケーション層の作成に取り掛かる。

Controller の作成

まずは、todo 管理業務にかかわる画面遷移を、制御する Controller を作成する。

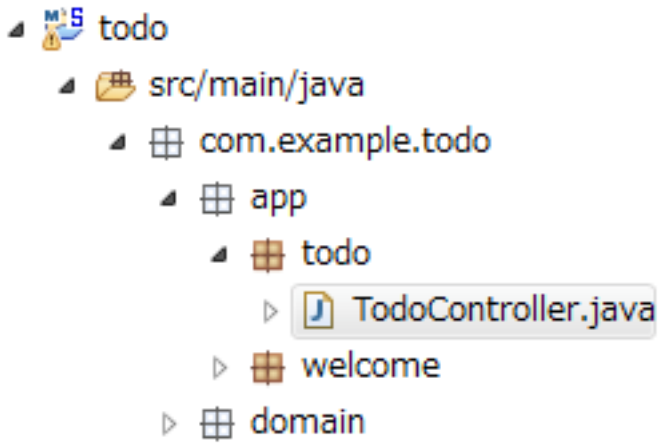
Package Explorer 上で右クリック -> New -> Class を選択し、「New Java Class」ダイアログを表示し、

項番	項目	入力値
1	Package	`com.example.todo.app.todo`
2	Name	`TodoController`

を入力して「Finish」する。

注釈: 上位パッケージがドメイン層と異なるので注意すること。

作成したクラスは以下のディレクトリに格納される。




```
package com.example.todo.app.todo;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
@Controller // (1)
@RequestMapping("todo") // (2)
public class TodoController {

}
```

項番	説明
(1)	Controller として component-scan の対象とするため、クラスレベルに、`@Controller` アノテーションをつける。
(2)	`TodoController` が扱う画面遷移のパスを、すべて`<contextPath>/todo` 配下にするため、クラスレベルに`@RequestMapping(“todo”)` を設定する。

Show all TODO の実装

本チュートリアルで作成する画面では、

- 新規作成フォームの表示
- TODO の全件表示

を行う。

はじめに、TODO の全件表示を行うための処理を実装する。

Form の作成

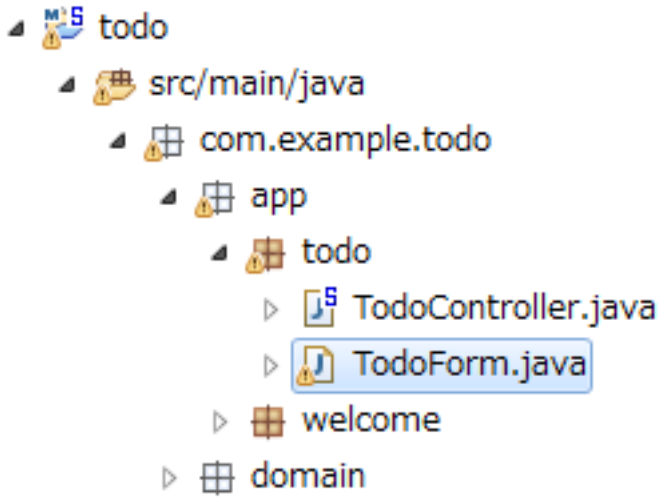
Form クラス (JavaBean) を作成する。

Package Explorer 上で右クリック -> New -> Class を選択し、「New Java Class」ダイアログを表示し、

項番	項目	入力値
1	Package	`com.example.todo.app.todo`
2	Name	`TodoForm`
3	Interfaces	`java.io.Serializable`

を入力して「Finish」する。

作成したクラスは以下のディレクトリに格納される。



作成したクラスに以下のプロパティを追加する。

- タイトル → todoTitle

```
package com.example.todo.app.todo;

import java.io.Serializable;

public class TodoForm implements Serializable {
    private static final long serialVersionUID = 1L;

    private String todoTitle;

    public String getTodoTitle() {
        return todoTitle;
    }

    public void setTodoTitle(String todoTitle) {
        this.todoTitle = todoTitle;
    }

}
```

Controller の実装

一覧画面表示処理を `TodoController` に追加する。

```
package com.example.todo.app.todo;

import java.util.Collection;

import javax.inject.Inject;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
```

```
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

import com.example.todo.domain.model.Todo;
import com.example.todo.domain.service.todo.TodoService;

@Controller
@RequestMapping("todo")
public class TodoController {
    @Inject // (1)
    TodoService todoService;

    @ModelAttribute // (2)
    public TodoForm setUpForm() {
        TodoForm form = new TodoForm();
        return form;
    }

    @GetMapping("list") // (3)
    public String list(Model model) {
        Collection<Todo> todos = todoService.findAll();
        model.addAttribute("todos", todos); // (4)
        return "todo/list"; // (5)
    }
}
```

項番	説明
(1)	TodoService を、DI コンテナによってインジェクションさせるために、@Inject アノテーションをつける。 DI コンテナの管理する TodoService 型のインスタンス (TodoServiceImpl のインスタンス) がインジェクションされる。
(2)	Form を初期化する。 @ModelAttribute アノテーションをつけることで、このメソッドの返り値の form オブジェクトが、todoForm という名前で Model に追加される。 これは、TodoController の各処理で、model.addAttribute("todoForm", form) を実装するのと同義である。
(3)	/todo/list というパスに GET メソッドを使用してリクエストされた際に、一覧画面表示処理用のメソッド (list メソッド) が実行されるように@GetMapping アノテーションを設定する。 クラスレベルに@RequestMapping(“todo”) が設定されているため、ここでは@GetMapping("list") のみで良い。
(4)	Model に Todo のリストを追加して、View に渡す。
(5)	View 名として todo/list を返すと、spring-mvc.xml に定義した ViewResolver によって、WEB-INF/views/todo/list.jsp がレンダリングされることになる。

注釈: @GetMapping や以降に登場する@PostMapping は、対応する HTTP メソッドにマッピングする。

詳細は、 リクエストとハンドラメソッドのマッピング方法 を参照されたい。

JSP の作成

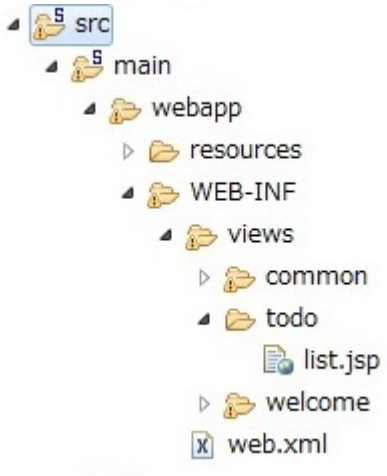
JSP を作成し、Controller から渡された Model を表示する。

Package Explorer 上で右クリック -> New -> File を選択し、「New File」ダイアログを表示し、

項番	項目	入力値
1	Enter or select the parent folder	`todo/src/main/webapp/WEB-INF/views/todo`
2	File name	`list.jsp`

を入力して「Finish」する。

作成したファイルは以下のディレクトリに格納される。



まず、TODO の全件表示を行うために必要な JSP の実装を行う。

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Todo List</title>
<style type="text/css">
.strike {
    text-decoration: line-through;
}
</style>
</head>
<body>
    <h1>Todo List</h1>
    <hr />
    <div id="todoList">
        <ul>
            <!-- (1) -->
            <c:forEach items="${todos}" var="todo">
                <li><c:choose>
                        <c:when test="${todo.finished}"><!-- (2) -->
                            <span class="strike">
                            <!-- (3) -->
                            ${f:h(todo.todoTitle)}
                            </span>
```

```
                        </c:when>
                        <c:otherwise>
                            ${f:h(todo.todoTitle)}
                         </c:otherwise>
                    </c:choose></li>
            </c:forEach>
        </ul>
    </div>
</body>
</html>
```

項番	説明
(1)	<c:forEach>タグを用いて、Todo のリストを全て表示する。
(2)	完了かどうか (finished) で、打ち消し線 (text-decoration: line-through;) を装飾するかどうかを判断する。
(3)	文字列値を出力する際は、**XSS** 対策のため、必ず **f:h()** 関数を使用して **HTML** エスケープを行うこと。 XSS 対策についての詳細は、*XSS* 対策を参照されたい。

STS で「todo」プロジェクトを右クリックし、「Run As」→「Run on Server」で Web アプリケーションを起動する。ブラウザで http://localhost:8080/todo/todo/list にアクセスすると、以下のような画面が表示される。



注釈: 上記で表示されている画面には、TODO が 1 件も登録されていないため、TODO の一覧は出力され

ない。

以下のように、ドメイン層の作成で作成した TodoRepositoryImpl を一時的に修正し初期データを登録することで、TODO の一覧が出力されることを確認できる。

なお、次節「*Create TODO の実装*」で実際に TODO を登録できるようになるため、一覧の出力が確認できたら削除して構わない。

- `TodoRepositoryImpl.java`

```
package com.example.todo.domain.repository.todo;

import java.util.Collection;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import org.springframework.stereotype.Repository;

import com.example.todo.domain.model.Todo;

@Repository
public class TodoRepositoryImpl implements TodoRepository {
    private static final Map<String, Todo> TODO_MAP = new ConcurrentHashMap
↪<String, Todo>();

    static {
        Todo todo1 = new Todo();
        todo1.setTodoId("1");
        todo1.setTodoTitle("Send a e-mail");
        Todo todo2 = new Todo();
        todo2.setTodoId("2");
        todo2.setTodoTitle("Have a lunch");
        Todo todo3 = new Todo();
        todo3.setTodoId("3");
        todo3.setTodoTitle("Read a book");
        todo3.setFinished(true);
        TODO_MAP.put(todo1.getTodoId(), todo1);
        TODO_MAP.put(todo2.getTodoId(), todo2);
        TODO_MAP.put(todo3.getTodoId(), todo3);
    }

      // omitted
```

以下のように画面に出力される。



Create TODO の実装

次に、一覧表示画面から「Create TODO」ボタンを押した後の、新規作成処理を実装する。

Controller の修正

新規作成処理を `TodoController` に追加する。

```
package com.example.todo.app.todo;

import java.util.Collection;

import javax.inject.Inject;
import javax.validation.Valid;

import org.dozer.Mapper;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.message.ResultMessage;
import org.terasoluna.gfw.common.message.ResultMessages;

import com.example.todo.domain.model.Todo;
import com.example.todo.domain.service.todo.TodoService;

@Controller
@RequestMapping("todo")
public class TodoController {
    @Inject
    TodoService todoService;

    // (1)
    @Inject
    Mapper beanMapper;

    @ModelAttribute
    public TodoForm setUpForm() {
        TodoForm form = new TodoForm();
        return form;
    }

    @GetMapping("list")
    public String list(Model model) {
        Collection<Todo> todos = todoService.findAll();
```

```
        model.addAttribute("todos", todos);
        return "todo/list";
    }

    @PostMapping("create") // (2)
    public String create(@Valid TodoForm todoForm, BindingResult bindingResult, //␣
↪(3)
            Model model, RedirectAttributes attributes) { // (4)

        // (5)
        if (bindingResult.hasErrors()) {
            return list(model);
        }

        // (6)
        Todo todo = beanMapper.map(todoForm, Todo.class);

        try {
            todoService.create(todo);
        } catch (BusinessException e) {
            // (7)
            model.addAttribute(e.getResultMessages());
            return list(model);
        }

        // (8)
        attributes.addFlashAttribute(ResultMessages.success().add(
                ResultMessage.fromText("Created successfully!")));
        return "redirect:/todo/list";
    }

}
```

項番	説明
(1)	Form オブジェクトを DomainObject に変換するために、Dozer の `Mapper` インタフェースをインジェクションする。
(2)	`/todo/create` というパスに `POST` メソッドを使用してリクエストされた際に、新規作成処理用のメソッド (`create` メソッド) が実行されるように`@PostMapping` アノテーションを設定する。
(3)	フォームの入力チェックを行うため、Form の引数に`@Valid` アノテーションをつける。入力チェック結果は、その直後の引数 `BindingResult` に格納される。

Continued on next page

表 11.2 – continued from previous page

項番	説明
(4)	正常に作成が完了した後にリダイレクトし、一覧画面を表示する。 リダイレクト先への情報を格納するために、引数に `RedirectAttributes` を加える。
(5)	入力エラーがあった場合、一覧画面に戻る。 Todo 全件取得を再度行う必要があるので、`list` メソッドを再実行する。
(6)	Dozer の `Mapper` インタフェースを用いて、`TodoForm` オブジェクトから `Todo` オブジェクトを作成する。 変換元と変換先のプロパティ名が同じ場合は、設定不要である。 今回は、`todoTitle` プロパティのみ変換するため、Dozer の `Mapper` インタフェースを使用するメリットはほとんどない。プロパティの数が多い場合には、非常に便利である。
(7)	業務処理を実行して、`BusinessException` が発生した場合、結果メッセージを `Model` に追加して、一覧画面に戻る。
(8)	正常に作成が完了したので、結果メッセージを flash スコープに追加して、一覧画面でリダイレクトする。 リダイレクトすることにより、ブラウザを再読み込みして、再び新規登録処理が `POST` されることがなくなる。（詳しくは、「*PRG(Post-Redirect-Get) パターンについて*」を参照されたい） なお、今回は成功メッセージであるため、`ResultMessages.success()` を使用している。

Form の修正

入力チェックのルールを定義するため、Form オブジェクトにアノテーションを追加する。

```
package com.example.todo.app.todo;

import java.io.Serializable;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class TodoForm implements Serializable {
    private static final long serialVersionUID = 1L;

    @NotNull // (1)
    @Size(min = 1, max = 30) // (2)
    private String todoTitle;

    public String getTodoTitle() {
        return todoTitle;
    }

    public void setTodoTitle(String todoTitle) {
        this.todoTitle = todoTitle;
    }
}
```

項番	説明
(1)	`@NotNull` アノテーションを使用して必須チェックを有効化する。
(2)	`@Size` アノテーションを使用して文字数チェックを有効化する。

JSP の修正

以下を表示するために必要な JSP の実装を追加する。

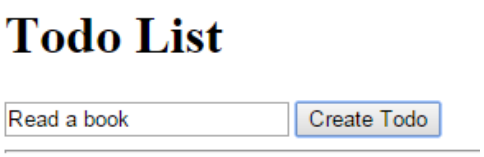
- TODO の入力フォーム
- 「Create Todo」ボタン
- 入力チェックエラーを表示するエリア
- 結果メッセージを表示するエリア

```
<!DOCTYPE html>
<html>
<head>
```

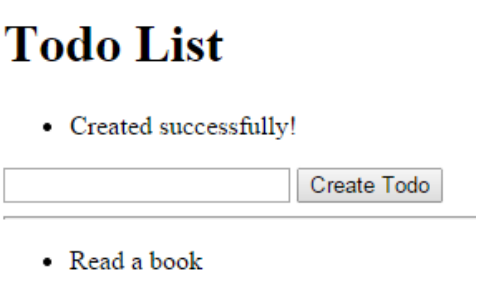
```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Todo List</title>
<style type="text/css">
.strike {
    text-decoration: line-through;
}
</style>
</head>
<body>
    <h1>Todo List</h1>
    <div id="todoForm">
        <!-- (1) -->
        <t:messagesPanel />

        <!-- (2) -->
        <form:form
           action="${pageContext.request.contextPath}/todo/create"
            method="post" modelAttribute="todoForm">
            <form:input path="todoTitle" /><!-- (3) -->
            <form:errors path="todoTitle" /><!-- (4) -->
            <form:button>Create Todo</form:button>
        </form:form>
    </div>
    <hr />
    <div id="todoList">
        <ul>
            <c:forEach items="${todos}" var="todo">
                <li><c:choose>
                        <c:when test="${todo.finished}">
                            <span class="strike">
                            ${f:h(todo.todoTitle)}
                            </span>
                        </c:when>
                        <c:otherwise>
                            ${f:h(todo.todoTitle)}
                         </c:otherwise>
                    </c:choose></li>
            </c:forEach>
        </ul>
    </div>
</body>
</html>
```

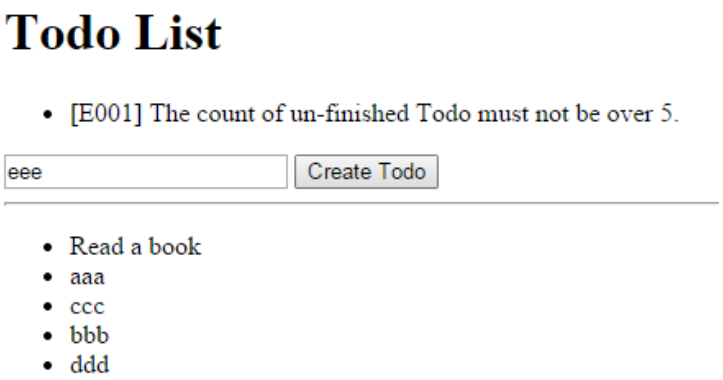
項番	説明
(1)	<t:messagesPanel>タグで、結果メッセージを表示する。
(2)	新規作成処理用の form を表示する。 form を表示するために、<form:form>タグを使用する。 modelAttribute 属性には、Controller で Model に追加した Form の名前を指定する。 action 属性には新規作成処理を実行するための URL(<contextPath>/todo/create) を指定する。 新規作成処理は更新系の処理なので、method 属性には POST メソッドを指定する。 action 属性に指定する<contextPath>は、 ${pageContext.request.contextPath}で取得することができる。
(3)	<form:input>タグでフォームのプロパティをバインドする。 modelAttribute 属性に指定した Form のプロパティ名と、path 属性の値が一致している必要がある。
(4)	<form:errors>タグで、入力エラーがあった場合に表示する。path 属性の値は、<form:input>タグと合わせる。

フォームに適切な値を入力して submit すると、以下のように、成功メッセージが表示される。

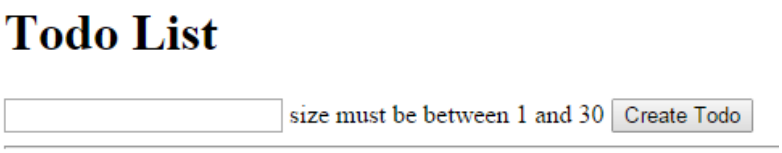


未完了の TODO が 5 件登録済みの場合は、業務エラーとなり、エラーメッセージが表示される。





入力フォームを、空文字にして submit すると、以下のように、エラーメッセージが表示される。



メッセージ表示のカスタマイズ

<t:messagesPanel>を使用した場合、以下のような HTML が出力される。

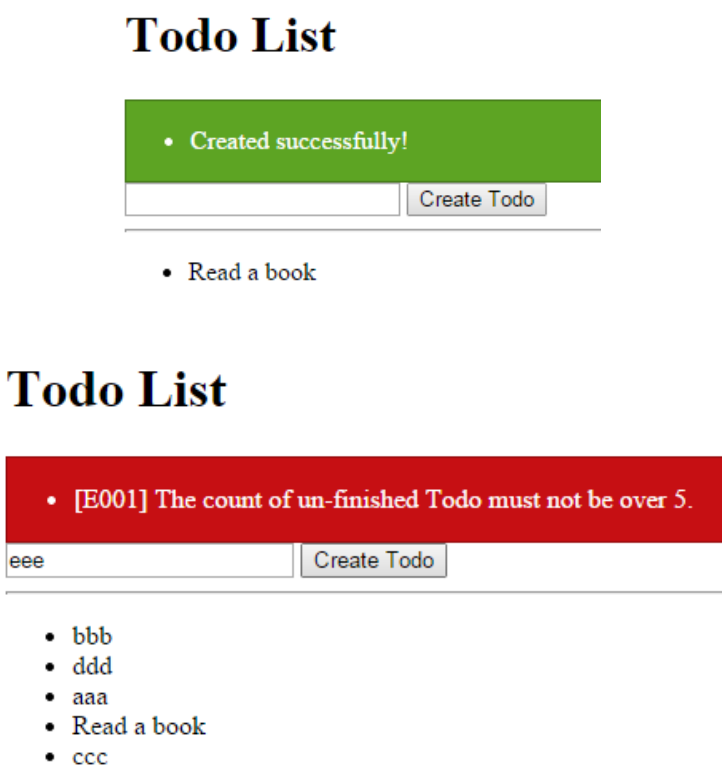
```
<div class="alert alert-success"><ul><li>Created successfully!</li></ul></div>
```

スタイルシート (list.jsp の<style>タグ内) に、以下の修正を加えて、結果メッセージの見た目をカスタマイズする。

```
.alert {
    border: 1px solid;
}

.alert-error {
    background-color: #c60f13;
    border-color: #970b0e;
    color: white;
}

.alert-success {
    background-color: #5da423;
    border-color: #457a1a;
```

```
    color: white;
}
```

メッセージは、以下のように装飾される。



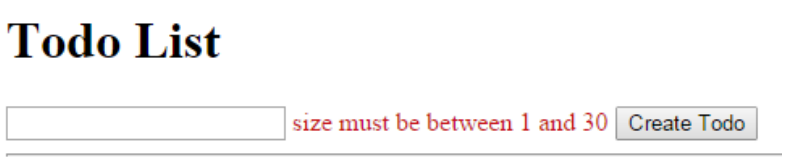
また、<form:errors>タグの cssClass 属性で、入力エラーメッセージの class を指定できる。

JSP を次のように修正し、

```
<form:errors path="todoTitle" cssClass="text-error" />
```

スタイルシートに、以下を追加する。

```
.text-error {
    color: #c60f13;
}
```



入力エラー時のメッセージは、以下のように装飾される。

Finish TODO の実装

一覧画面に「Finish」ボタンを追加し、TODO を完了させるための処理を追加する。

Form の修正

完了処理用の Form についても、`TodoForm` を使用する。

`TodoForm` に `todoId` プロパティを追加する必要があるが、単純に追加してしまうと、新規作成処理でも `todoId` プロパティのチェックが実行されてしまう。一つの Form クラスを使用して複数の form から送信されるリクエストパラメータをバインドする場合は、`groups` 属性を使用して、入力チェックルールをグループ化する。

Form クラスに以下のプロパティを追加する。

- ID → todoId

```
package com.example.todo.app.todo;

import java.io.Serializable;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class TodoForm implements Serializable {
    // (1)
    public static interface TodoCreate {
    };

    public static interface TodoFinish {
    };

    private static final long serialVersionUID = 1L;

    // (2)
    @NotNull(groups = { TodoFinish.class })
    private String todoId;

    // (3)
    @NotNull(groups = { TodoCreate.class })
```

```
    @Size(min = 1, max = 30, groups = { TodoCreate.class })
    private String todoTitle;

    public String getTodoId() {
        return todoId;
    }

    public void setTodoId(String todoId) {
        this.todoId = todoId;
    }

    public String getTodoTitle() {
        return todoTitle;
    }

    public void setTodoTitle(String todoTitle) {
        this.todoTitle = todoTitle;
    }

}
```

項番	説明
(1)	入力チェックルールをグループ化するためのインタフェースを作成する。 入力チェックルールのグループ化については、入力チェックを参照されたい。 ここでは、新規作成処理用のインタフェースとして TodoCreate を、完了処理用のインタフェースとして TodoFinish を作成している。
(2)	todoId は完了処理で使用するプロパティである。 そのため、@NotNull アノテーションの groups 属性には、完了処理用の入力チェックルールである事を示す TodoFinish インタフェースを指定する。
(3)	todoTitle は新規作成処理で使用するプロパティである。 そのため、@NotNull アノテーションと@Size アノテーションの groups 属性には、新規作成処理用の入力チェックルールである事を示す TodoCreate インタフェースを指定する。

Controller の修正

完了処理を TodoController に追加する。

グループ化した入力チェックルールを適用するためには、**@Valid** アノテーションの代わりに、**@Validated**

アノテーションを使用することに注意する。

```
package com.example.todo.app.todo;

import java.util.Collection;

import javax.inject.Inject;
import javax.validation.groups.Default;

import org.dozer.Mapper;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.message.ResultMessage;
import org.terasoluna.gfw.common.message.ResultMessages;

import com.example.todo.app.todo.TodoForm.TodoCreate;
import com.example.todo.app.todo.TodoForm.TodoFinish;
import com.example.todo.domain.model.Todo;
import com.example.todo.domain.service.todo.TodoService;

@Controller
@RequestMapping("todo")
public class TodoController {
    @Inject
    TodoService todoService;

    @Inject
    Mapper beanMapper;

    @ModelAttribute
    public TodoForm setUpForm() {
        TodoForm form = new TodoForm();
        return form;
    }

    @GetMapping("list")
    public String list(Model model) {
        Collection<Todo> todos = todoService.findAll();
        model.addAttribute("todos", todos);
        return "todo/list";
    }

    @PostMapping("create")
    public String create(
            @Validated({ Default.class, TodoCreate.class }) TodoForm todoForm, //␣
↪(1)
```

```
            BindingResult bindingResult, Model model,
            RedirectAttributes attributes) {

        if (bindingResult.hasErrors()) {
            return list(model);
        }

        Todo todo = beanMapper.map(todoForm, Todo.class);

        try {
            todoService.create(todo);
        } catch (BusinessException e) {
            model.addAttribute(e.getResultMessages());
            return list(model);
        }

        attributes.addFlashAttribute(ResultMessages.success().add(
                ResultMessage.fromText("Created successfully!")));
        return "redirect:/todo/list";
    }

    @PostMapping("finish") // (2)
    public String finish(
            @Validated({ Default.class, TodoFinish.class }) TodoForm form, // (3)
            BindingResult bindingResult, Model model,
            RedirectAttributes attributes) {
        // (4)
        if (bindingResult.hasErrors()) {
            return list(model);
        }

        try {
            todoService.finish(form.getTodoId());
        } catch (BusinessException e) {
            // (5)
            model.addAttribute(e.getResultMessages());
            return list(model);
        }

        // (6)
        attributes.addFlashAttribute(ResultMessages.success().add(
                ResultMessage.fromText("Finished successfully!")));
        return "redirect:/todo/list";
    }
}
```

項番	説明
(1)	グループ化した入力チェックルールを適用するために、`@Valid` アノテーションを`@Validated` アノテーションに変更する。 `value` 属性には、適用する入力チェックルールのグループ (グループインタフェース) を指定する。`Default.class` は、グループ化されていない入力チェックルールを適用するために用意されているグループインタフェースである。
(2)	`/todo/finish` というパスに `POST` メソッドを使用してリクエストされた際に、完了処理用のメソッド (`finish` メソッド) が実行されるように`@PostMapping` アノテーションを設定する。
(3)	適用する入力チェックのグループとして、完了処理用のグループインタフェース (`TodoFinish` インタフェース) を指定する。
(4)	入力エラーがあった場合、一覧画面に戻る。
(5)	業務処理を実行して、`BusinessException` が発生した場合は、結果メッセージを `Model` に追加して、一覧画面に戻る。
(6)	正常に作成が完了した場合は、結果メッセージを flash スコープに追加して、一覧画面でリダイレクトする。

注釈: 新規作成処理用と完了処理用を別々の Form クラスとして作成しても良い。別々の Form クラスにした場合、入力チェックルールをグループ化する必要がないため、入力チェックルールの定義はシンプルになる。

ただし、処理毎に Form クラスを作成した場合、

- クラス数が増える
- プロパティが重複するため入力チェックルールを一元管理できない

ため、仕様変更が発生した場合に修正コストが高くなる可能性があるという点に注意してほしい。

また、`@ModelAttribute` メソッドを使用して複数の Form を初期化した場合、毎回すべての Form が初期

化されるため、不要なインスタンスが生成されることになる。

JSP の修正

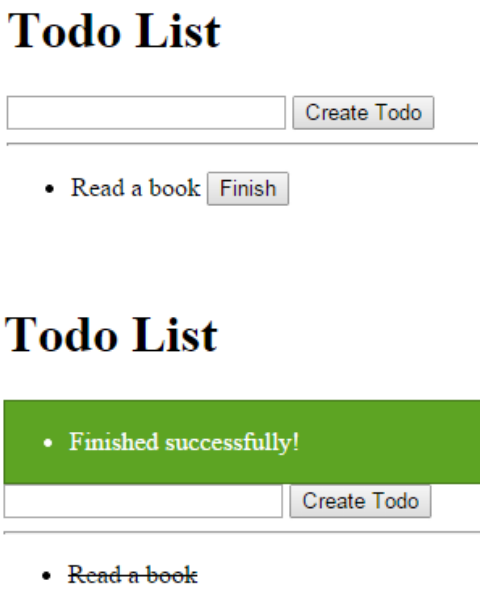
完了処理用の form を追加する。

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Todo List</title>
</head>
<style type="text/css">
.strike {
    text-decoration: line-through;
}

.inline {
    display: inline-block;
}

.alert {
    border: 1px solid;
}

.alert-error {
    background-color: #c60f13;
    border-color: #970b0e;
    color: white;
}

.alert-success {
    background-color: #5da423;
    border-color: #457a1a;
    color: white;
}

.text-error {
    color: #c60f13;
}
</style>
<body>
    <h1>Todo List</h1>

    <div id="todoForm">
        <t:messagesPanel />

        <form:form
            action="${pageContext.request.contextPath}/todo/create"
            method="post" modelAttribute="todoForm">
            <form:input path="todoTitle" />
            <form:errors path="todoTitle" cssClass="text-error" />
            <form:button>Create Todo</form:button>
```

```
        </form:form>
    </div>
    <hr />
    <div id="todoList">
        <ul>
            <c:forEach items="${todos}" var="todo">
                <li><c:choose>
                        <c:when test="${todo.finished}">
                            <span class="strike">${f:h(todo.todoTitle)}</span>
                        </c:when>
                        <c:otherwise>
                            ${f:h(todo.todoTitle)}
                            <!-- (1) -->
                            <form:form
                                action="${pageContext.request.contextPath}/todo/
↪finish"
                                method="post"
                                modelAttribute="todoForm"
                                cssClass="inline">
                                <!-- (2) -->
                                <form:hidden path="todoId"
                                    value="${f:h(todo.todoId)}" />
                                <form:button>Finish</form:button>
                            </form:form>
                        </c:otherwise>
                    </c:choose></li>
            </c:forEach>
        </ul>
    </div>
</body>
</html>
```

項番	説明
(1)	TODO が未完了の場合は、TODO を完了させるためのリクエストを送信する form を表示する。 action 属性には完了処理を実行するための URL(<contextPath>/todo/finish) を指定する。 完了処理は更新系の処理なので、method 属性には POST メソッドを指定する。 なお、「Finish」ボタンをインラインブロック要素（display: inline-block;）として TODO の横に表示させている。
(2)	<form:hidden>タグを使用して、リクエストパラメータとして todoId を送信する。 value 属性に値を設定する場合も、 **必ず f:h() 関数で HTML エスケープすること。**

Todo を新規作成した後に、「Finish」ボタン押下すると、以下のように打ち消し線が入り、完了したことがわかる。



Delete TODO の実装

一覧表示画面に「Delete」ボタンを追加して、TODO を削除するための処理を追加する。

Form の修正

削除処理用の Form についても、`TodoForm` を使用する。

```
package com.example.todo.app.todo;

import java.io.Serializable;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class TodoForm implements Serializable {
    public static interface TodoCreate {
    };

    public static interface TodoFinish {
    };

    // (1)
    public static interface TodoDelete {
    }

    private static final long serialVersionUID = 1L;
```

```
    // (2)
    @NotNull(groups = { TodoFinish.class, TodoDelete.class })
    private String todoId;

    @NotNull(groups = { TodoCreate.class })
    @Size(min = 1, max = 30, groups = { TodoCreate.class })
    private String todoTitle;

    public String getTodoId() {
        return todoId;
    }

    public void setTodoId(String todoId) {
        this.todoId = todoId;
    }

    public String getTodoTitle() {
        return todoTitle;
    }

    public void setTodoTitle(String todoTitle) {
        this.todoTitle = todoTitle;
    }

}
```

項番	説明
(1)	削除処理用の入力チェックルールをグループ化するためのインタフェースとして TodoDelete を作成する。
(2)	削除処理では todoId プロパティを使用する。 そのため、todoId の@NotNull アノテーションの groups 属性には、削除処理用の入力チェックルールである事を示す TodoDelete インタフェースを指定する。

Controller の修正

削除処理を TodoController に追加する。完了処理とほぼ同じである。

```
package com.example.todo.app.todo;

import java.util.Collection;

import javax.inject.Inject;
import javax.validation.groups.Default;
```

```
import org.dozer.Mapper;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.message.ResultMessage;
import org.terasoluna.gfw.common.message.ResultMessages;

import com.example.todo.app.todo.TodoForm.TodoDelete;
import com.example.todo.app.todo.TodoForm.TodoCreate;
import com.example.todo.app.todo.TodoForm.TodoFinish;
import com.example.todo.domain.model.Todo;
import com.example.todo.domain.service.todo.TodoService;

@Controller
@RequestMapping("todo")
public class TodoController {
    @Inject
    TodoService todoService;

    @Inject
    Mapper beanMapper;

    @ModelAttribute
    public TodoForm setUpForm() {
        TodoForm form = new TodoForm();
        return form;
    }

    @GetMapping("list")
    public String list(Model model) {
        Collection<Todo> todos = todoService.findAll();
        model.addAttribute("todos", todos);
        return "todo/list";
    }

    @PostMapping("create")
    public String create(
            @Validated({ Default.class, TodoCreate.class }) TodoForm todoForm,
            BindingResult bindingResult, Model model,
            RedirectAttributes attributes) {

        if (bindingResult.hasErrors()) {
            return list(model);
        }

        Todo todo = beanMapper.map(todoForm, Todo.class);
```

```
        try {
            todoService.create(todo);
        } catch (BusinessException e) {
            model.addAttribute(e.getResultMessages());
            return list(model);
        }

        attributes.addFlashAttribute(ResultMessages.success().add(
                ResultMessage.fromText("Created successfully!")));
        return "redirect:/todo/list";
    }

    @PostMapping("finish")
    public String finish(
            @Validated({ Default.class, TodoFinish.class }) TodoForm form,
            BindingResult bindingResult, Model model,
            RedirectAttributes attributes) {
        if (bindingResult.hasErrors()) {
            return list(model);
        }

        try {
            todoService.finish(form.getTodoId());
        } catch (BusinessException e) {
            model.addAttribute(e.getResultMessages());
            return list(model);
        }

        attributes.addFlashAttribute(ResultMessages.success().add(
                ResultMessage.fromText("Finished successfully!")));
        return "redirect:/todo/list";
    }

    @PostMapping("delete") // (1)
    public String delete(
            @Validated({ Default.class, TodoDelete.class }) TodoForm form,
            BindingResult bindingResult, Model model,
            RedirectAttributes attributes) {

        if (bindingResult.hasErrors()) {
            return list(model);
        }

        try {
            todoService.delete(form.getTodoId());
        } catch (BusinessException e) {
            model.addAttribute(e.getResultMessages());
            return list(model);
        }

        attributes.addFlashAttribute(ResultMessages.success().add(
                ResultMessage.fromText("Deleted successfully!")));
```

```
        return "redirect:/todo/list";
    }

}
```

項番	説明
(1)	/todo/delete というパスに POST メソッドを使用してリクエストされた際に、削除処理用のメソッド (delete メソッド) が実行されるように@PostMapping アノテーションを設定する。

JSP の修正

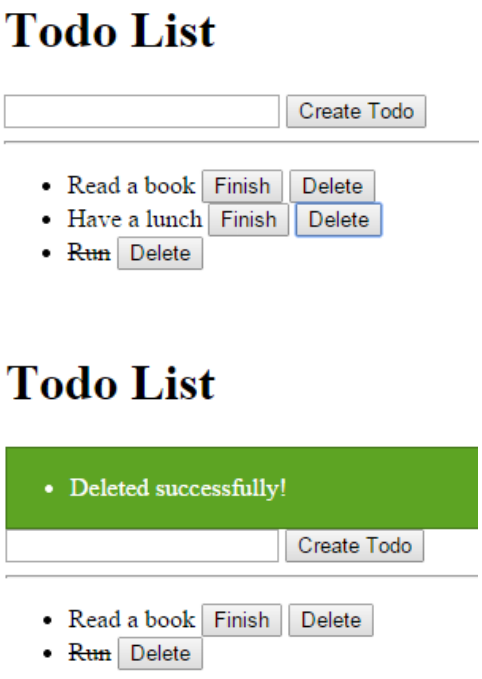
削除処理用の form を追加する。

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Todo List</title>
</head>
<style type="text/css">
.strike {
    text-decoration: line-through;
}

.inline {
    display: inline-block;
}

.alert {
    border: 1px solid;
}

.alert-error {
    background-color: #c60f13;
    border-color: #970b0e;
    color: white;
}

.alert-success {
    background-color: #5da423;
    border-color: #457a1a;
    color: white;
}

.text-error {
    color: #c60f13;
}
</style>
<body>
```

```
    <h1>Todo List</h1>

    <div id="todoForm">
        <t:messagesPanel />

        <form:form
            action="${pageContext.request.contextPath}/todo/create"
            method="post" modelAttribute="todoForm">
            <form:input path="todoTitle" />
            <form:errors path="todoTitle" cssClass="text-error" />
            <form:button>Create Todo</form:button>
        </form:form>
    </div>
    <hr />
    <div id="todoList">
        <ul>
            <c:forEach items="${todos}" var="todo">
                <li><c:choose>
                        <c:when test="${todo.finished}">
                            <span class="strike">${f:h(todo.todoTitle)}</span>
                        </c:when>
                        <c:otherwise>
                            ${f:h(todo.todoTitle)}
                            <form:form
                                action="${pageContext.request.contextPath}/todo/
↪finish"
                                method="post"
                                modelAttribute="todoForm"
                                cssClass="inline">
                                <form:hidden path="todoId"
                                    value="${f:h(todo.todoId)}" />
                                <form:button>Finish</form:button>
                            </form:form>
                        </c:otherwise>
                    </c:choose>
                    <!-- (1) -->
                    <form:form
                        action="${pageContext.request.contextPath}/todo/delete"
                        method="post" modelAttribute="todoForm"
                        cssClass="inline">
                        <!-- (2) -->
                        <form:hidden path="todoId"
                            value="${f:h(todo.todoId)}" />
                        <form:button>Delete</form:button>
                    </form:form>
                </li>
            </c:forEach>
        </ul>
    </div>
</body>
</html>
```

項番	説明
(1)	削除処理用の form を表示する。 `action` 属性には削除処理を実行するための URL(`<contextPath>/todo/delete`) を指定する。 削除処理は更新系の処理なので、`method` 属性には `POST` メソッドを指定する。
(2)	`<form:hidden>`タグを使用して、リクエストパラメータとして `todoId` を送信する。 `value` 属性に値を設定する場合も、**必ず f:h() 関数で HTML エスケープすること。**

未完了状態の TODO の「Delete」ボタンを押下すると、以下のように TODO が削除される。



CSS ファイルの使用

これまでスタイルシートを JSP ファイルの中で直接定義していたが、実際のアプリケーションを開発する場合は、CSS ファイルに定義するのが一般的である。

ここでは、スタイルシートを CSS ファイルに定義する方法について説明する。

ブランクプロジェクトから提供している CSS ファイル (src/main/webapp/resources/app/css/styles.css) にスタイルシートの定義を追加する。

```
/* ... */

.strike {
    text-decoration: line-through;
}

.inline {
    display: inline-block;
}

.alert {
    border: 1px solid;
    margin-bottom: 5px;
}

.alert-error {
    background-color: #c60f13;
    border-color: #970b0e;
    color: white;
}

.alert-success {
    background-color: #5da423;
    border-color: #457a1a;
    color: white;
}

.text-error {
    color: #c60f13;
}

.alert ul {
    margin: 15px 0px 15px 0px;
}

#todoList li {
    margin-top: 5px;
}
```

JSP から CSS ファイルを読み込む。

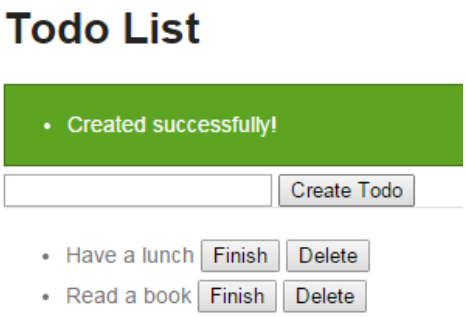
```
<!DOCTYPE html>
<html>
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Todo List</title>
<!-- (1) -->
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/css/
↪styles.css" type="text/css">
</head>
<body>
    <h1>Todo List</h1>

    <div id="todoForm">
        <t:messagesPanel />

        <form:form
            action="${pageContext.request.contextPath}/todo/create"
            method="post" modelAttribute="todoForm">
            <form:input path="todoTitle" />
            <form:errors path="todoTitle" cssClass="text-error" />
            <form:button>Create Todo</form:button>
        </form:form>
    </div>
    <hr />
    <div id="todoList">
        <ul>
            <c:forEach items="${todos}" var="todo">
                <li><c:choose>
                        <c:when test="${todo.finished}">
                            <span class="strike">${f:h(todo.todoTitle)}</span>
                        </c:when>
                        <c:otherwise>
                            ${f:h(todo.todoTitle)}
                            <form:form
                                action="${pageContext.request.contextPath}/todo/
↪finish"
                                method="post"
                                modelAttribute="todoForm"
                                cssClass="inline">
                                <form:hidden path="todoId"
                                    value="${f:h(todo.todoId)}" />
                                <form:button>Finish</form:button>
                            </form:form>
                        </c:otherwise>
                    </c:choose>
                    <form:form
                        action="${pageContext.request.contextPath}/todo/delete"
                        method="post" modelAttribute="todoForm"
                        cssClass="inline">
                        <form:hidden path="todoId"
                            value="${f:h(todo.todoId)}" />
                        <form:button>Delete</form:button>
                    </form:form>
                </li>
            </c:forEach>
        </ul>
```

```
    </div>
</body>
</html>
```

項番	説明
(1)	JSP ファイルからスタイルシートの定義を削除し、代わりにスタイルシートを定義した CSS ファイルを読み込む。

CSS ファイルを適用すると、以下のようなレイアウトになる。



11.1.5 データベースアクセスを伴うインフラストラクチャ層の作成

ここでは、Domain オブジェクトをデータベースに永続化するためのインフラストラクチャ層の実装方法について説明する。

本チュートリアルでは、以下の 2 つの O/R Mapper を使用したインフラストラクチャ層の実装方法について説明する。

- MyBatis3
- Spring Data JPA

O/R Mapper に依存したブランクプロジェクトの作成

ここでは、O/R Mapper に依存したブランクプロジェクトの作成を行う。

まず、使用する O/R Mapper に応じてプロジェクトを作成し直す。

- *MyBatis3* 用のブランクプロジェクトの作成
- *JPA* 用のブランクプロジェクトの作成

次に、データベースアクセスを伴うインフラストラクチャ層の作成までで作成した `src` フォルダ以下のうち、**TodoRepositoryImpl** クラス以外のファイルを新規作成したプロジェクトにコピーする。

ただし、コピーするファイルは新規作成したファイル・変更を加えたファイルに限り、修正を加えていないファイルはコピーしないこと。

データベースのセットアップ

ここでは、データベースのセットアップを行う。

本チュートリアルでは、データベースのセットアップの手間を省くため、H2 Database を使用する。

todo-infra.properties の修正

AP サーバ起動時に H2 Database 上にテーブルが作成されるようにするために、`src/main/resources/META-INF/spring/todo-infra.properties` の設定を変更する。

```
database=H2
# (1)
database.url=jdbc:h2:mem:todo;DB_CLOSE_DELAY=-1;INIT=create table if not exists␣
↪todo(todo_id varchar(36) primary key, todo_title varchar(30), finished boolean,␣
↪created_at timestamp)
database.username=sa
database.password=
database.driverClassName=org.h2.Driver
# connection pool
cp.maxActive=96
cp.maxIdle=16
cp.minIdle=0
cp.maxWait=60000
```

項番	説明
(1)	接続 URL の INIT パラメータに、テーブルを作成する DDL 文を指定する。

注釈: INIT パラメータに設定している DDL 文をフォーマットすると、以下の様な SQL となる。

```
create table if not exists todo (
    todo_id varchar(36) primary key,
    todo_title varchar(30),
    finished boolean,
    created_at timestamp
)
```

MyBatis3 を使用したインフラストラクチャ層の作成

ここでは、MyBatis3 を使用してインフラストラクチャ層の RepositoryImpl を作成する方法について説明する。

Spring Data JPA を使用する場合は、本節を読み飛ばして、*Spring Data JPA を使用したインフラストラクチャ層の作成*に進んでよい。

TodoRepository の作成

`TodoRepository` は、O/R Mapper を使用しない場合と同じ方法で作成する。作成方法は、「*Repository の作成*」を参照されたい。

TodoRepositoryImpl の作成

MyBatis3 を使用する場合、RepositoryImpl は Repository インタフェース (Mapper インタフェース) から自動生成される。そのため、`TodoRepositoryImpl` の作成は不要である。作成した場合は削除すること。

Mapper ファイルの作成

`TodoRepository` インタフェースのメソッドが呼び出された際に実行する SQL を定義するための Mapper ファイルを作成する。

Package Explorer 上で右クリック -> New -> File を選択し、「New File」ダイアログを表示し、

項番	項目	入力値
1	Enter or select the parent folder	`todo/src/main/resources/com/example/todo/domain/repository/todo`
2	File name	`TodoRepository.xml`

を入力して「Finish」する。

作成したファイルは以下のディレクトリに格納される。

`TodoRepository` インタフェースに定義したメソッドが呼び出された際に実行する SQL を記述する。




```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- (1) -->
<mapper namespace="com.example.todo.domain.repository.todo.TodoRepository">

    <!-- (2) -->
    <resultMap id="todoResultMap" type="Todo">
        <id property="todoId" column="todo_id" />
        <result property="todoTitle" column="todo_title" />
        <result property="finished" column="finished" />
        <result property="createdAt" column="created_at" />
    </resultMap>

    <!-- (3) -->
    <select id="findOne" parameterType="String" resultMap="todoResultMap">
    <![CDATA[
        SELECT
            todo_id,
            todo_title,
            finished,
            created_at
        FROM
            todo
        WHERE
            todo_id = #{todoId}
    ]]>
    </select>

    <!-- (4) -->
    <select id="findAll" resultMap="todoResultMap">
    <![CDATA[
```

```
        SELECT
            todo_id,
            todo_title,
            finished,
            created_at
        FROM
            todo
    ]]>
    </select>

    <!-- (5) -->
    <insert id="create" parameterType="Todo">
    <![CDATA[
        INSERT INTO todo
        (
            todo_id,
            todo_title,
            finished,
            created_at
        )
        VALUES
        (
            #{todoId},
            #{todoTitle},
            #{finished},
            #{createdAt}
        )
    ]]>
    </insert>

    <!-- (6) -->
    <update id="update" parameterType="Todo">
    <![CDATA[
        UPDATE todo
        SET
            todo_title = #{todoTitle},
            finished = #{finished},
            created_at = #{createdAt}
        WHERE
            todo_id = #{todoId}
    ]]>
    </update>

    <!-- (7) -->
    <delete id="delete" parameterType="Todo">
    <![CDATA[
        DELETE FROM
            todo
        WHERE
            todo_id = #{todoId}
    ]]>
    </delete>
```

```
    <!-- (8) -->
    <select id="countByFinished" parameterType="Boolean"
        resultType="Long">
    <![CDATA[
        SELECT
            COUNT(*)
        FROM
            todo
        WHERE
            finished = #{finished}
    ]]>
    </select>

</mapper>
```

項番	説明
(1)	mapper 要素の namespace 属性に、Repository インタフェースの完全修飾クラス名 (FQCN) を指定する。
(2)	<resultMap>要素に、検索結果 (ResultSet) と JavaBean のマッピング定義を行う。 マッピングファイルの詳細はデータベースアクセス（*MyBatis3* 編）を参照されたい。
(3)	todoId(PK) が一致するレコードを 1 件取得する SQL を実装する。 <select>要素の resultMap 属性には、適用するマッピング定義の ID を指定する。
(4)	全レコードを取得する SQL を実装している。 <select>要素の resultMap 属性に、適用するマッピング定義の ID を指定する。 アプリケーションの要件には記載がないが、最新の TODO が先頭に表示されるようにレコードを並び替えている。
(5)	引数に指定された Todo オブジェクトを挿入する SQL を実装する。 <insert>要素の parameterType 属性に、パラメータのクラス名 (FQCN 又はエイリアス名) を指定する。
(6)	引数に指定された Todo オブジェクトを更新する SQL を実装する。 <update>要素の parameterType 属性に、パラメータのクラス名 (FQCN 又はエイリアス名) を指定する。
(7)	引数に指定された Todo オブジェクトを削除する SQL を実装する。 <delete>要素の parameterType 属性に、パラメータのクラス名 (FQCN 又はエイリアス名) を指定する。
(8)	引数に指定された finished に一致する Todo の件数を取得する SQL を実装する。

以上で、MyBatis3 を使用したインフラストラクチャ層の作成が完了したので、Service 及びアプリケーション層の作成を行う。

Service 及びアプリケーション層を作成後に AP サーバーを起動し、Todo の表示を行うと、以下のような SQL ログやトランザクションログが出力される。

```
 date:2016-02-17 13:18:54    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:TRACE     logger:o.t.gfw.web.
↪logging.TraceLoggingInterceptor      message:[START CONTROLLER] TodoController.
↪list(Model)
 date:2016-02-17 13:18:54    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:DEBUG     logger:o.s.jdbc.
↪datasource.DataSourceTransactionManager message:Creating new transaction with␣
↪name [com.example.todo.domain.service.todo.TodoServiceImpl.findAll]: PROPAGATION_
↪REQUIRED,ISOLATION_DEFAULT,readOnly; ''
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:DEBUG     logger:o.s.jdbc.
↪datasource.DataSourceTransactionManager message:Acquired Connection [net.sf.
↪log4jdbc.ConnectionSpy@4e53de7c] for JDBC transaction
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:DEBUG     logger:t.domain.
↪repository.todo.TodoRepository.findAll  message:==>  Preparing: SELECT todo_id,␣
↪todo_title, finished, created_at FROM todo
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:DEBUG     logger:t.domain.
↪repository.todo.TodoRepository.findAll  message:==> Parameters:
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:DEBUG     logger:jdbc.
↪sqltiming                                  message: sun.reflect.
↪NativeMethodAccessorImpl.invoke0(NativeMethodAccessorImpl.java:-2)
 1. SELECT
            todo_id,
            todo_title,
            finished,
            created_at
        FROM
            todo {executed in 0 msec}
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:INFO      logger:jdbc.
↪resultsettable                             message:|--------|-----------|-------
↪--|-----------|
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:INFO      logger:jdbc.
↪resultsettable                             message:|TODO_ID |TODO_TITLE␣
↪|FINISHED |CREATED_AT |
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:INFO      logger:jdbc.
↪resultsettable                             message:|--------|-----------|-------
↪--|-----------|
```

```
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:INFO      logger:jdbc.
↪resultsettable                              message:|--------|-----------|-------
↪--|-----------|
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:DEBUG     logger:t.domain.
↪repository.todo.TodoRepository.findAll  message:<==      Total: 0
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:DEBUG     logger:o.s.jdbc.
↪datasource.DataSourceTransactionManager message:Initiating transaction commit
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:DEBUG     logger:o.s.jdbc.
↪datasource.DataSourceTransactionManager message:Committing JDBC transaction on␣
↪Connection [net.sf.log4jdbc.ConnectionSpy@4e53de7c]
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:DEBUG     logger:o.s.jdbc.
↪datasource.DataSourceTransactionManager message:Releasing JDBC Connection [net.
↪sf.log4jdbc.ConnectionSpy@4e53de7c] after transaction
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:TRACE     logger:o.t.gfw.web.
↪logging.TraceLoggingInterceptor      message:[END CONTROLLER  ] TodoController.
↪list(Model)-> view=todo/list, model={todoForm=com.example.todo.app.todo.
↪TodoForm@2a075f1d, todos=[], org.springframework.validation.BindingResult.
↪todoForm=org.springframework.validation.BeanPropertyBindingResult: 0 errors}
 date:2016-02-17 13:18:55    thread:tomcat-http--5   X-
↪Track:390066c43aa94b6588e5bac6a54812b2        level:TRACE     logger:o.t.gfw.web.
↪logging.TraceLoggingInterceptor      message:[HANDLING TIME   ] TodoController.
↪list(Model)-> 756,709,153 ns
```

Spring Data JPA を使用したインフラストラクチャ層の作成

ここでは、Spring Data JPA を使用してインフラストラクチャ層の RepositoryImpl を作成する方法について説明する。

Entity の修正

Todo クラスとデータベースの TODO テーブルをマッピングするために、JPA のアノテーションを設定する。

```
package com.example.todo.domain.model;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
```

```
import javax.persistence.TemporalType;

// (1)
@Entity
@Table(name = "todo")
public class Todo implements Serializable {
    private static final long serialVersionUID = 1L;

    // (2)
    @Id
    @Column(name = "todo_id")
    private String todoId;

    @Column(name = "todo_title")
    private String todoTitle;

    @Column(name = "finished")
    private boolean finished;

    // (3)
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_at")
    private Date createdAt;

    public String getTodoId() {
        return todoId;
    }

    public void setTodoId(String todoId) {
        this.todoId = todoId;
    }

    public String getTodoTitle() {
        return todoTitle;
    }

    public void setTodoTitle(String todoTitle) {
        this.todoTitle = todoTitle;
    }

    public boolean isFinished() {
        return finished;
    }

    public void setFinished(boolean finished) {
        this.finished = finished;
    }

    public Date getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Date createdAt) {
```

```
        this.createdAt = createdAt;
    }
}
```

項番	説明
(1)	JPA のエンティティであることを示す`@Entity` アノテーションを付け、対応するテーブル名を`@Table` アノテーションで設定する。
(2)	主キーとなるカラムに対応するフィールドに、`@Id` アノテーションをつける。
(3)	`java.util.Date` 型は、`java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp` のインスタンスを格納できるため、明示的にどの型のインスタンスを設定するか指定する必要がある。 `createdAt` プロパティには、`Timestamp` を指定する。

TodoRepository の作成

Spring Data JPA の Repository 機能を使用して `TodoRepository` の作成を行う。

Package Explorer 上で右クリック -> New -> Interface を選択し、「New Java Interface」ダイアログを表示し、

項番	項目	入力値
1	Package	`com.example.todo.domain.repository.todo`
2	Name	`TodoRepository`
3	Extended interfaces	`org.springframework.data.jpa.repository.JpaRepository<T, ID>`

を入力して「Finish」する。

```
package com.example.todo.domain.repository.todo;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import com.example.todo.domain.model.Todo;

// (1)
public interface TodoRepository extends JpaRepository<Todo, String> {
```

```
    @Query("SELECT COUNT(t) FROM Todo t WHERE t.finished = :finished") // (2)
    long countByFinished(@Param("finished") boolean finished); // (3)

}
```

項番	説明
(1)	JpaRepository の Generics のパラメータを指定する。 左から順に、Entity のクラス (Todo)、主キーのクラス (String) を指定する。 基本的な CRUD 操作 (findOne, findAll, save, delete など) は、JpaRepository インタフェースに定義済みであるため、TodoRepository には countByFinished メソッドのみ定義すればよい。
(2)	countByFinished メソッドを呼び出した際に実行する JPQL を、@Query アノテーションで指定する。
(3)	(2) で指定した JPQL 内のバインド変数に対応するメソッド引数に、@Param アノテーションを指定する。 ここでは、JPQL 中の”:finished”に値を埋め込むために、メソッド引数の finished に@Param(“finished”) を付けている。

TodoRepositoryImpl の作成

Spring Data JPA を使用する場合、RepositoryImpl は Repository インタフェースから自動生成される。そのため、TodoRepositoryImpl の作成は不要である。作成した場合は削除すること。

以上で、Spring Data JPA を使用したインフラストラクチャ層の作成が完了したので、Service 及びアプリケーション層の作成を行う。

Service 及びアプリケーション層を作成後に AP サーバーを起動し、Todo の表示を行うと、以下のような SQL ログや、トランザクションログが出力される。

```
 date:2016-02-17 13:32:44    thread:tomcat-http--5    X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:TRACE     logger:o.t.gfw.web.
↪logging.TraceLoggingInterceptor      message:[START CONTROLLER] TodoController.
↪list(Model)
```

```
date:2016-02-17 13:32:44    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:DEBUG     logger:o.h.e.
↪transaction.spi.AbstractTransactionImpl    message:begin
date:2016-02-17 13:32:44    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:DEBUG     logger:o.h.e.
↪transaction.internal.jdbc.JdbcTransaction  message:initial autocommit status:␣
↪false
date:2016-02-17 13:32:45    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:DEBUG     logger:jdbc.
↪sqltiming                                   message: org.hibernate.engine.jdbc.
↪internal.ResultSetReturnImpl.extract(ResultSetReturnImpl.java:82)
5. /* select generatedAlias0 from Todo as generatedAlias0 */ select todo0_.todo_
↪id as todo_id1_0_, todo0_.created_at as created_2_0_, todo0_.finished as␣
↪finished3_0_, todo0_.todo_title as todo_tit4_0_ from todo todo0_ {executed in 1␣
↪msec}
date:2016-02-17 13:32:45    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:INFO      logger:jdbc.
↪resultsettable                              message:|--------|-----------|-------
↪--|-----------|
date:2016-02-17 13:32:45    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:INFO      logger:jdbc.
↪resultsettable                              message:|TODO_ID |CREATED_AT␣
↪|FINISHED |TODO_TITLE |
date:2016-02-17 13:32:45    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:INFO      logger:jdbc.
↪resultsettable                              message:|--------|-----------|-------
↪--|-----------|
date:2016-02-17 13:32:45    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:INFO      logger:jdbc.
↪resultsettable                              message:|--------|-----------|-------
↪--|-----------|
date:2016-02-17 13:32:45    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:DEBUG     logger:o.h.e.
↪transaction.spi.AbstractTransactionImpl    message:committing
date:2016-02-17 13:32:45    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:DEBUG     logger:o.h.e.
↪transaction.internal.jdbc.JdbcTransaction  message:committed JDBC Connection
date:2016-02-17 13:32:45    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:TRACE     logger:o.t.gfw.web.
↪logging.TraceLoggingInterceptor      message:[END CONTROLLER  ] TodoController.
↪list(Model)-> view=todo/list, model={todoForm=com.example.todo.app.todo.
↪TodoForm@5dac2c75, todos=[], org.springframework.validation.BindingResult.
↪todoForm=org.springframework.validation.BeanPropertyBindingResult: 0 errors}
date:2016-02-17 13:32:45    thread:tomcat-http--5   X-
↪Track:7c34263e0a2143639f3ffd191b35c135        level:TRACE     logger:o.t.gfw.web.
↪logging.TraceLoggingInterceptor      message:[HANDLING TIME   ] TodoController.
↪list(Model)-> 320,129,237 ns
```

11.1.6 おわりに

このチュートリアルでは、以下の内容を学習した。

- TERASOLUNA Server Framework for Java (5.x) による基本的なアプリケーションの開発方法
- Maven および STS(Eclipse) プロジェクトの構築方法
- TERASOLUNA Server Framework for Java (5.x) のアプリケーションのレイヤ化に従った開発方法
- POJO(+ Spring) を使用したドメイン層の実装
- POJO(+ Spring MVC) と JSP タグライブラリを使用したアプリケーション層の実装
- MyBatis3 を使用したインフラストラクチャ層の実装
- Spring Data JPA を使用したインフラストラクチャ層の実装
- O/R Mapper を使用しないインフラストラクチャ層の実装

本チュートリアルで作成した TODO 管理アプリケーションには、以下の改善点がある。アプリケーションの修正を学習課題として、ガイドライン中の該当する説明を参照されたい。

- プロパティ (未完了 TODO の上限数) を外部化する → プロパティ管理
- メッセージを外部化する → メッセージ管理
- ページネーション機能を追加する → ページネーション
- 例外ハンドリングを加える → 例外ハンドリング
- 二重送信を防止する (トランザクショントークンチェックを追加する) → 二重送信防止
- システム日時の取得元を変更する → システム時刻

11.1.7 Appendix

設定ファイルの解説

アプリケーションを動かすためにどのような設定が必要なのかを理解するために、設定ファイルの解説を行う。ここでは、チュートリアルで作成する Todo アプリケーションで使用しない設定については、解説を割愛している箇所がある。

web.xml

`web.xml` には、Web アプリケーションとして Todo アプリをデプロイするための設定を行う。

作成したブランクプロジェクトの `src/main/webapp/WEB-INF/web.xml` は、以下のような設定となっている。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (1) -->
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/
↪ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <!-- (2) -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</
↪listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <!-- Root ApplicationContext -->
        <param-value>
            classpath*:META-INF/spring/applicationContext.xml
            classpath*:META-INF/spring/spring-security.xml
        </param-value>
    </context-param>

    <listener>
        <listener-class>org.terasoluna.gfw.web.logging.
↪HttpSessionEventLoggingListener</listener-class>
    </listener>

    <!-- (3) -->
    <filter>
        <filter-name>MDCClearFilter</filter-name>
        <filter-class>org.terasoluna.gfw.web.logging.mdc.MDCClearFilter</filter-
↪class>
    </filter>
    <filter-mapping>
        <filter-name>MDCClearFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>exceptionLoggingFilter</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
↪class>
    </filter>
    <filter-mapping>
        <filter-name>exceptionLoggingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>XTrackMDCPutFilter</filter-name>
        <filter-class>org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter</
↪filter-class>
    </filter>
```

```
    <filter-mapping>
        <filter-name>XTrackMDCPutFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>CharacterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</
↪filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
↪class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- (4) -->
    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
↪class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <!-- ApplicationContext for Spring MVC -->
            <param-value>classpath*:META-INF/spring/spring-mvc.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!-- (5) -->
    <jsp-config>
        <jsp-property-group>
```

```
            <url-pattern>*.jsp</url-pattern>
            <el-ignored>false</el-ignored>
            <page-encoding>UTF-8</page-encoding>
            <scripting-invalid>false</scripting-invalid>
            <include-prelude>/WEB-INF/views/common/include.jsp</include-prelude>
        </jsp-property-group>
    </jsp-config>

    <!-- (6) -->
    <error-page>
        <error-code>500</error-code>
        <location>/WEB-INF/views/common/error/systemError.jsp</location>
    </error-page>

    <error-page>
        <error-code>404</error-code>
        <location>/WEB-INF/views/common/error/resourceNotFoundError.jsp</location>
    </error-page>

    <error-page>
        <exception-type>java.lang.Exception</exception-type>
        <location>/WEB-INF/views/common/error/unhandledSystemError.html</location>
    </error-page>

    <!-- (7) -->
    <session-config>
        <!-- 30min -->
        <session-timeout>30</session-timeout>
        <cookie-config>
            <http-only>true</http-only>
            <!-- <secure>true</secure> -->
        </cookie-config>
        <tracking-mode>COOKIE</tracking-mode>
    </session-config>

</web-app>
```

項番	説明
(1)	Servlet3.0 を使用するための宣言。

Continued on next page

表 11.3 – continued from previous page

項番	説明
(2)	サーブレットコンテキストリスナーの定義。 ブランクプロジェクトでは、 • アプリケーション全体で使用される ApplicationContext を作成するための ContextLoaderListener • HttpSession に対する操作をログ出力するための HttpSessionEventLoggingListener が設定済みである。
(3)	サーブレットフィルタの定義。 ブランクプロジェクトでは、 • 共通ライブラリから提供しているサーブレットフィルタ • Spring Framework から提供されている文字エンコーディングを指定するための CharacterEncodingFilter • Spring Security から提供されている認証・認可用のサーブレットフィルタ が設定済みである。
(4)	Spring MVC のエントリポイントとなる DispatcherServlet の定義。 DispatcherServlet の中で使用する ApplicationContext を、(2) で作成した ApplicationContext の子として作成する。 (2) で作成した ApplicationContext を親にすることで、(2) で読み込まれたコンポーネントも使用することができる。
(5)	JSP の共通定義。 ブランクプロジェクトでは、 • JSP 内で EL 式が使用可能な状態 • JSP のページエンコーディングとして UTF-8 • JSP 内でスクリプティングが使用可能な状態 • 各 JSP の先頭でインクルードする JSP として、/WEB-INF/views/common/include.jsp が設定済みである。
(6)	エラーページの定義。 ブランクプロジェクトでは、 • サーブレットコンテナに HTTP ステータスコードとして、404 又は 500 が応答 • サーブレットコンテナに例外が通知 された際の遷移先が定義済みである。

Continued on next page

表 11.3 – continued from previous page

項番	説明
(7)	セッション管理の定義。 ブランクプロジェクトでは、 • セッションタイムアウトとして、30 分 が定義済みである。

インクルード JSP

インクルード JSP には、全ての JSP に適用する JSP の設定や、タグライブラリの設定を行う。

作成したブランクプロジェクトの `src/main/webapp/WEB-INF/views/common/include.jsp` は、以下のような設定となっている。

```
<!-- (1) -->
<%@ page session="false"%>
<!-- (2) -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<!-- (3) -->
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<!-- (4) -->
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec"%>
<!-- (5) -->
<%@ taglib uri="http://terasoluna.org/tags" prefix="t"%>
<%@ taglib uri="http://terasoluna.org/functions" prefix="f"%>
```

項番	説明
(1)	JSP 実行時にセッションを作成しないようにするための定義。
(2)	標準タグライブラリの定義。
(3)	Spring MVC 用タグライブラリの定義。
(4)	Spring Security 用タグライブラリの定義 (本チュートリアルでは使用しない。)
(5)	共通ライブラリで提供されている、EL 関数、タグライブラリの定義。

Bean 定義ファイル

作成したブランクプロジェクトには、以下の Bean 定義ファイルとプロパティファイルが作成される。

- src/main/resources/META-INF/spring/applicationContext.xml
- src/main/resources/META-INF/spring/todo-domain.xml
- src/main/resources/META-INF/spring/todo-infra.xml
- src/main/resources/META-INF/spring/todo-infra.properties
- src/main/resources/META-INF/spring/todo-env.xml
- src/main/resources/META-INF/spring/spring-mvc.xml
- src/main/resources/META-INF/spring/spring-security.xml

注釈: O/R Mapper に依存しないブランクプロジェクトを作成した場合は、todo-infra.properties と todo-env.xml は作成されない。

注釈: 本ガイドラインでは、Bean 定義ファイルを役割 (層) ごとにファイルを分割することを推奨している。

これは、どこに何が定義されているか想像しやすく、メンテナンス性が向上するからである。今回のチュートリアルのような小さなアプリケーションでは効果はないが、アプリケーションの規模が大きくなるにつれ、効果が大きくなる。

---

applicationContext.xml

applicationContext.xml には、Todo アプリ全体に関わる設定を行う。

作成したブランクプロジェクトの
src/main/resources/META-INF/spring/applicationContext.xml は、以下のような設定となっている。
なお、チュートリアルで使用しないコンポーネントについての説明は割愛する。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
```

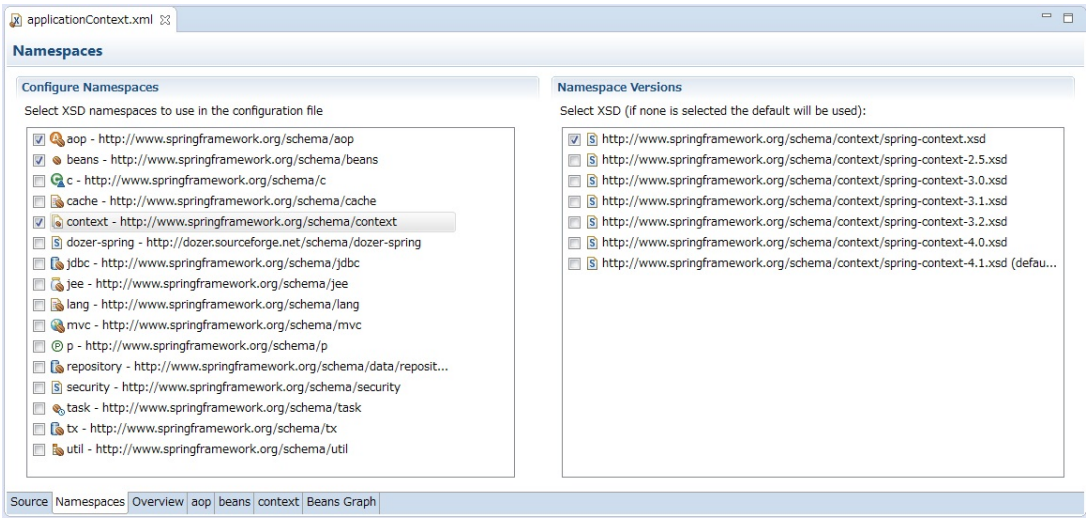
```
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context https://www.springframework.
↪org/schema/context/spring-context.xsd
    ">

    <!-- (1) -->
    <import resource="classpath:/META-INF/spring/todo-domain.xml" />

    <bean id="passwordEncoder" class="org.springframework.security.crypto.bcrypt.
↪BCryptPasswordEncoder" />

    <!-- (2) -->
    <context:property-placeholder
        location="classpath*:/META-INF/spring/*.properties" />

    <!-- (3) -->
    <bean id="beanMapper" class="org.dozer.spring.DozerBeanMapperFactoryBean">
        <property name="mappingFiles"
            value="classpath*:/META-INF/dozer/**/*-mapping.xml" />
    </bean>

    <!-- Message -->
    <bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>i18n/application-messages</value>
            </list>
        </property>
    </bean>

    <!-- Exception Code Resolver. -->
    <bean id="exceptionCodeResolver"
        class="org.terasoluna.gfw.common.exception.
↪SimpleMappingExceptionCodeResolver">
        <!-- Setting and Customization by project. -->
        <property name="exceptionMappings">
            <map>
                <entry key="ResourceNotFoundException" value="e.xx.fw.5001" />
                <entry key="InvalidTransactionTokenException" value="e.xx.fw.7001"␣
↪/>
                <entry key="BusinessException" value="e.xx.fw.8001" />
                <entry key=".DataAccessException" value="e.xx.fw.9002" />
            </map>
        </property>
        <property name="defaultExceptionCode" value="e.xx.fw.9001" />
    </bean>

    <!-- Exception Logger. -->
    <bean id="exceptionLogger"
        class="org.terasoluna.gfw.common.exception.ExceptionLogger">
```

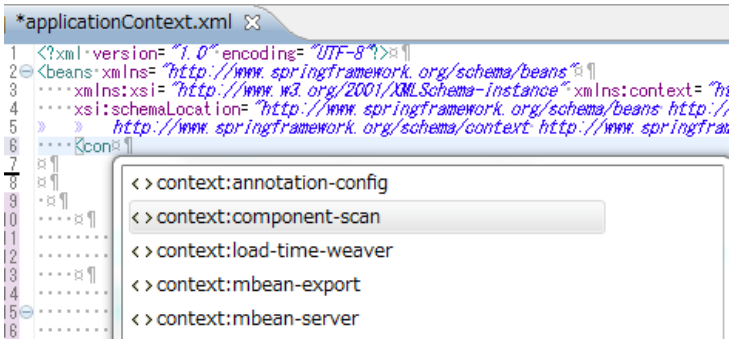
```
        <property name="exceptionCodeResolver" ref="exceptionCodeResolver" />
    </bean>

    <!-- Filter. -->
    <bean id="exceptionLoggingFilter"
        class="org.terasoluna.gfw.web.exception.ExceptionLoggingFilter" >
        <property name="exceptionLogger" ref="exceptionLogger" />
    </bean>

</beans>
```

項番	説明
(1)	ドメイン層に関する Bean 定義ファイルを import する。
(2)	プロパティファイルの読み込み設定を行う。 `src/main/resources/META-INF/spring` 直下の任意のプロパティファイルを読み込む。 この設定により、プロパティファイルの値を Bean 定義ファイル内で`${propertyName}`形式で埋め込んだり、Java クラスに`@Value("${propertyName}")` でインジェクションすることができる。
(3)	Bean 変換用ライブラリ Dozer の Mapper を定義する。 (マッピングファイルに関しては Dozer マニュアル を参照されたい。)

ちなみに: エディタの「Configure Namespaces」タブにて、以下のようにチェックを入れると、チェックした XML スキーマが有効になり、XML 編集時に Ctrl+Space を使用して入力を補完することができる。

「Namespace Versions」にはバージョンなしの xsd ファイルを選択することを推奨する。バージョンなしの xsd ファイルを選択することで、常に jar に含まれる最新の xsd が使用されるため、Spring のバージョンアップを意識する必要がなくなる。





todo-domain.xml

`todo-domain.xml` には、Todo アプリのドメイン層に関わる設定を行う。

作成したブランクプロジェクトの `src/main/resources/META-INF/spring/todo-domain.xml` は、以下のような設定となっている。

なお、チュートリアルで使用しないコンポーネントについての説明は割愛する。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
        http://www.springframework.org/schema/aop https://www.springframework.org/
↪schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context https://www.springframework.
↪org/schema/context/spring-context.xsd
    ">
```

```
    <!-- (1) -->
    <import resource="classpath:META-INF/spring/todo-infra.xml" />
    <import resource="classpath*:META-INF/spring/**/*-codelist.xml" />

    <!-- (2) -->
    <context:component-scan base-package="com.example.todo.domain" />

    <!-- AOP. -->
    <bean id="resultMessagesLoggingInterceptor"
        class="org.terasoluna.gfw.common.exception.ResultMessagesLoggingInterceptor
↪">
        <property name="exceptionLogger" ref="exceptionLogger" />
    </bean>
    <aop:config>
        <aop:advisor advice-ref="resultMessagesLoggingInterceptor"
            pointcut="@within(org.springframework.stereotype.Service)" />
    </aop:config>

</beans>
```

項番	説明
(1)	インフラストラクチャ層に関する Bean 定義ファイルを import する。
(2)	ドメイン層のクラスを管理する com.example.todo.domain パッケージ配下を component-scan 対象とする。 これにより、com.example.todo.domain パッケージ配下のクラスに `@Repository` , `@Service` などのアノテーションを付けることで、Spring Framerowk が管理する Bean として登録される。 登録されたクラス (Bean) は、Controller や Service クラスに DI する事で、利用する事が出来る。

注釈: O/R Mapper に依存するブランクプロジェクトを作成した場合は、`@Transactional` アノテーションによるトランザクション管理を有効にするために、`<tx:annotation-driven>`タグが設定されている。

```
<tx:annotation-driven />
```

---

todo-infra.xml

todo-infra.xml には、Todo アプリのインフラストラクチャ層に関わる設定を行う。

作成したブランクプロジェクトの src/main/resources/META-INF/spring/todo-infra.xml は、以下のような設定となっている。

todo-infra.xml は、インフラストラクチャ層によって設定が大きく異なるため、ブランクプロジェクト毎に説明を行う。作成したブランクプロジェクト以外の説明は読み飛ばしてもよい。

O/R Mapper に依存しないブランクプロジェクトを作成した場合の todo-infra.xml

O/R Mapper に依存しないブランクプロジェクトを作成した場合、以下のように空定義のファイルが作成される。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
    ">

</beans>
```

MyBatis3 用のブランクプロジェクトを作成した場合の todo-infra.xml

MyBatis3 用のブランクプロジェクトを作成した場合、以下のような設定となっている。

```
 <?xml version="1.0" encoding="UTF-8"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="
         http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
         http://mybatis.org/schema/mybatis-spring http://mybatis.org/schema/
↪mybatis-spring.xsd
     ">

      <!-- (1) -->
     <import resource="classpath:/META-INF/spring/todo-env.xml" />

      <!-- (2) -->
     <!-- define the SqlSessionFactory -->
     <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
          <!-- (3) -->
         <property name="dataSource" ref="dataSource" />
          <!-- (4) -->
         <property name="configLocation" value="classpath:/META-INF/mybatis/
↪mybatis-config.xml" />
     </bean>

      <!-- (5) -->
```

```
    <!-- scan for Mappers -->
    <mybatis:scan base-package="com.example.todo.domain.repository" />

</beans>
```

項番	説明
(1)	環境依存するコンポーネント (データソースやトランザクションマネージャなど) を定義する Bean 定義ファイルを import する。
(2)	`SqlSessionFactory` を生成するためのコンポーネントとして、`SqlSessionFactoryBean` を bean 定義する。
(3)	`dataSource` プロパティに、設定済みのデータソースの bean を指定する。 MyBatis3 の処理の中で SQL を発行する際は、ここで指定したデータソースからコネクションが取得される。
(4)	`configLocation` プロパティに、MyBatis 設定ファイルのパスを指定する。 ここで指定したファイルは `SqlSessionFactory` を生成する時に読み込まれる。
(5)	Mapper インタフェースをスキャンするために`<mybatis:scan>`を定義し、`base-package` 属性には、 Mapper インタフェースが格納されている基底パッケージを指定する。 指定されたパッケージ配下に格納されている Mapper インタフェースがスキャンされ、 スレッドセーフな Mapper オブジェクト (Mapper インタフェースの Proxy オブジェクト) が自動的に生成される。

注釈: `mybatis-config.xml` は、MyBatis3 自体の動作設定を行う設定ファイルである。

ブランクプロジェクトでは、デフォルトで以下の設定が行われている。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <!-- See http://mybatis.github.io/mybatis-3/configuration.html#settings --
↪>
    <settings>
        <setting name="mapUnderscoreToCamelCase" value="true" />
        <setting name="lazyLoadingEnabled" value="true" />
        <setting name="defaultFetchSize" value="100" />
<!--
        <setting name="defaultExecutorType" value="REUSE" />
        <setting name="jdbcTypeForNull" value="NULL" />
        <setting name="localCacheScope" value="STATEMENT" />
-->
    </settings>

    <typeAliases>
        <package name="com.example.todo.domain.model" />
        <package name="com.example.todo.domain.repository" />
<!--
        <package name="com.example.todo.infra.mybatis.typehandler" />
-->
    </typeAliases>

    <typeHandlers>
<!--
        <package name="com.example.todo.infra.mybatis.typehandler" />
-->
    </typeHandlers>

</configuration>
```

JPA 用のブランクプロジェクトを作成した場合の todo-infra.xml

JPA 用のブランクプロジェクトを作成した場合、以下のような設定となっている。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jpa="http://www.springframework.org/schema/data/jpa"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util https://www.springframework.org/
↪schema/util/spring-util.xsd
        http://www.springframework.org/schema/data/jpa https://www.springframework.
↪org/schema/data/jpa/spring-jpa.xsd
    ">
```

```
    <!-- (1) -->
    <import resource="classpath:/META-INF/spring/todo-env.xml" />

    <!-- (2) -->
    <jpa:repositories base-package="com.example.todo.domain.repository"></
↪jpa:repositories>

    <!-- (3) -->
    <bean id="jpaVendorAdapter"
        class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <property name="showSql" value="false" />
        <property name="database" value="${database}" />
    </bean>

    <!-- (4) -->
    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <!-- (5) -->
        <property name="packagesToScan" value="com.example.todo.domain.model" />
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
        <!-- (6) -->
        <property name="jpaPropertyMap">
            <util:map>
                <entry key="hibernate.hbm2ddl.auto" value="" />
                <entry key="hibernate.ejb.naming_strategy"
                    value="org.hibernate.cfg.ImprovedNamingStrategy" />
                <entry key="hibernate.connection.charSet" value="UTF-8" />
                <entry key="hibernate.show_sql" value="false" />
                <entry key="hibernate.format_sql" value="false" />
                <entry key="hibernate.use_sql_comments" value="true" />
                <entry key="hibernate.jdbc.batch_size" value="30" />
                <entry key="hibernate.jdbc.fetch_size" value="100" />
            </util:map>
        </property>
    </bean>

</beans>
```

項番	説明
(1)	環境依存するコンポーネント (データソースやトランザクションマネージャなど) を定義する Bean 定義ファイルを import する。
(2)	Spring Data JPA を使用して、Repository インタフェースから実装クラスを自動生成する。 <jpa:repository>タグの base-package 属性に、Repository を格納するパッケージを指定する。
(3)	JPA の実装ベンダの設定を行う。 JPA 実装として、Hibernate を使うため、HibernateJpaVendorAdapter を定義している。
(4)	EntityManager の定義を行う。
(5)	JPA のエンティティとして扱うクラスが格納されているパッケージ名を指定する。
(6)	Hibernate に関する詳細な設定を行う。

todo-infra.properties

todo-infra.properties には、Todo アプリのインフラストラクチャ層における環境依存値の設定を行う。

O/R Mapper に依存しないブランクプロジェクトを作成した際は、todo-infra.properties は作成されない。

作成したブランクプロジェクトの src/main/resources/META-INF/spring/todo-infra.properties は、以下のような設定となっている。

```
# (1)
database=H2
database.url=jdbc:h2:mem:todo;DB_CLOSE_DELAY=-1
```

```
database.username=sa
database.password=
database.driverClassName=org.h2.Driver
# (2)
# connection pool
cp.maxActive=96
cp.maxIdle=16
cp.minIdle=0
cp.maxWait=60000
```

項番	説明
(1)	データベースに関する設定を行う。 本チュートリアルでは、データベースのセットアップの手間を省くため、H2 Database を使用する。
(2)	コネクションプールに関する設定。

注釈: これらの設定値は、todo-env.xml から参照されている。

todo-env.xml

todo-env.xml には、デプロイする環境によって設定が異なるコンポーネントの設定を行う。

作成したブランクプロジェクトの src/main/resources/META-INF/spring/todo-env.xml は、以下のような設定となっている。

ここでは、MyBatis3 用のブランクプロジェクトに格納されるファイルを例に説明する。なお、データベースにアクセスしないブランクプロジェクトを作成した際は、todo-env.xml は作成されない。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/jdbc https://www.springframework.org/
↪schema/jdbc/spring-jdbc.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
```

```
    ">

    <bean id="dateFactory" class="org.terasoluna.gfw.common.date.jodatime.
↪DefaultJodaTimeDateFactory" />

    <!-- (1) -->
    <bean id="realDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName" value="${database.driverClassName}" />
        <property name="url" value="${database.url}" />
        <property name="username" value="${database.username}" />
        <property name="password" value="${database.password}" />
        <property name="defaultAutoCommit" value="false" />
        <property name="maxTotal" value="${cp.maxActive}" />
        <property name="maxIdle" value="${cp.maxIdle}" />
        <property name="minIdle" value="${cp.minIdle}" />
        <property name="maxWaitMillis" value="${cp.maxWait}" />
    </bean>


    <!-- (2) -->
    <bean id="dataSource" class="net.sf.log4jdbc.Log4jdbcProxyDataSource">
        <constructor-arg index="0" ref="realDataSource" />
    </bean>

    <!-- (3) -->
    <jdbc:initialize-database data-source="dataSource"
        ignore-failures="ALL">
        <!-- (4) -->
        <jdbc:script location="classpath:/database/${database}-schema.sql"␣
↪encoding="UTF-8" />
        <jdbc:script location="classpath:/database/${database}-dataload.sql"␣
↪encoding="UTF-8" />
    </jdbc:initialize-database>

    <!--  REMOVE THIS LINE IF YOU USE JPA
    <bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
          REMOVE THIS LINE IF YOU USE JPA  -->
    <!-- (5) -->
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
        <property name="rollbackOnCommitFailure" value="true" />
    </bean>
</beans>
```

項番	説明
(1)	実データソースの設定。
(2)	データソースの設定。 JDBC 関連のログを出力する機能をもったデータソースを指定している。 `net.sf.log4jdbc.Log4jdbcProxyDataSource` を使用すると、SQL などの JDBC 関連のログを出力できるため、デバッグに役立つ情報を出力することができる。
(3)	データベース初期化の設定。 データベースを初期化する SQL ファイルを実行するための設定を行っている。 この設定は通常、開発中のみでしか使用しない (環境に依存する設定) ため、`todo-env.xml` に定義されている。
(4)	データベースを初期化する SQL ファイルの設定。 データベースを初期化するための、DDL 文が記載されている SQL ファイルと DML 文が記載されている SQL ファイルを指定している。 ブランクプロジェクトの設定では `todo-infra.properties` に `database=H2` と定義されているため、`H2-schema.sql` 及び `H2-dataload.sql` が実行される。
(5)	トランザクションマネージャの設定。 id 属性には、`transactionManager` を指定する。 別の名前を指定する場合は、`<tx:annotation-driven>`タグにもトランザクションマネージャ名を指定する必要がある。 ブランクプロジェクトでは、JDBC の API を使用してトランザクションを制御するクラス (`org.springframework.jdbc.datasource.DataSourceTransactionManager`) が設定されている。

注釈: JPA 用のブランクプロジェクトを作成した場合は、トランザクションマネージャには、

JPA の API を使用してトランザクションを制御するクラス (org.springframework.orm.jpa.JpaTransactionManager) が設定されている。

```
<bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

spring-mvc.xml

spring-mvc.xml には、Spring MVC に関する定義を行う。

作成したブランクプロジェクトの src/main/resources/META-INF/spring/spring-mvc.xml は、以下のような設定となっている。

なお、チュートリアルで使用しないコンポーネントについての説明は割愛する。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc https://www.
↪springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util https://www.springframework.org/
↪schema/util/spring-util.xsd
        http://www.springframework.org/schema/context https://www.springframework.
↪org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop https://www.springframework.org/
↪schema/aop/spring-aop.xsd
    ">

    <!-- (1) -->
    <context:property-placeholder
        location="classpath*:/META-INF/spring/*.properties" />

    <!-- (2) -->
    <mvc:annotation-driven>
        <mvc:argument-resolvers>
```

```
            <bean
                class="org.springframework.data.web.
↪PageableHandlerMethodArgumentResolver" />
            <bean
                class="org.springframework.security.web.method.annotation.
↪AuthenticationPrincipalArgumentResolver" />
        </mvc:argument-resolvers>
    </mvc:annotation-driven>

    <mvc:default-servlet-handler />

    <!-- (3) -->
    <context:component-scan base-package="com.example.todo.app" />

    <!-- (4) -->
    <mvc:resources mapping="/resources/**"
        location="/resources/,classpath:META-INF/resources/"
        cache-period="#{60 * 60}" />

    <mvc:interceptors>
        <!-- (5) -->
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
                class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor" />
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
                class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenInterceptor" />
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean class="org.terasoluna.gfw.web.codelist.CodeListInterceptor">
                <property name="codeListIdPattern" value="CL_.+" />
            </bean>
        </mvc:interceptor>
        <!--  REMOVE THIS LINE IF YOU USE JPA
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
                class="org.springframework.orm.jpa.support.
↪OpenEntityManagerInViewInterceptor" />
        </mvc:interceptor>
            REMOVE THIS LINE IF YOU USE JPA  -->
    </mvc:interceptors>

    <!-- (6) -->
```

```
    <!-- Settings View Resolver. -->
    <mvc:view-resolvers>
        <mvc:jsp prefix="/WEB-INF/views/" />
    </mvc:view-resolvers>

    <bean id="requestDataValueProcessor"
        class="org.terasoluna.gfw.web.mvc.support.
↪CompositeRequestDataValueProcessor">
        <constructor-arg>
            <util:list>
                <bean
                    class="org.springframework.security.web.servlet.support.csrf.
↪CsrfRequestDataValueProcessor" />
                <bean
                    class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenRequestDataValueProcessor" />
            </util:list>
        </constructor-arg>
    </bean>

    <!-- Setting Exception Handling. -->
    <!-- Exception Resolver. -->
    <bean id="systemExceptionResolver"
        class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">
        <property name="exceptionCodeResolver" ref="exceptionCodeResolver" />
        <!-- Setting and Customization by project. -->
        <property name="order" value="3" />
        <property name="exceptionMappings">
            <map>
                <entry key="ResourceNotFoundException" value="common/error/
↪resourceNotFoundError" />
                <entry key="BusinessException" value="common/error/businessError" /
↪>
                <entry key="InvalidTransactionTokenException" value="common/error/
↪transactionTokenError" />
                <entry key=".DataAccessException" value="common/error/
↪dataAccessError" />
            </map>
        </property>
        <property name="statusCodes">
            <map>
                <entry key="common/error/resourceNotFoundError" value="404" />
                <entry key="common/error/businessError" value="409" />
                <entry key="common/error/transactionTokenError" value="409" />
                <entry key="common/error/dataAccessError" value="500" />
            </map>
        </property>
        <property name="excludedExceptions">
            <array>
                <value>org.springframework.web.util.NestedServletException</value>
            </array>
        </property>
        <property name="defaultErrorView" value="common/error/systemError" />
```

```
        <property name="defaultStatusCode" value="500" />
    </bean>
    <!-- Setting AOP. -->
    <bean id="handlerExceptionResolverLoggingInterceptor"
        class="org.terasoluna.gfw.web.exception.
↪HandlerExceptionResolverLoggingInterceptor">
        <property name="exceptionLogger" ref="exceptionLogger" />
    </bean>
    <aop:config>
        <aop:advisor advice-ref="handlerExceptionResolverLoggingInterceptor"
            pointcut="execution(* org.springframework.web.servlet.
↪HandlerExceptionResolver.resolveException(..))" />
    </aop:config>

</beans>
```

項番	説明
(1)	プロパティファイルの読み込み設定を行う。 src/main/resources/META-INF/spring 直下の任意のプロパティファイルを読み込む。 この設定により、プロパティファイルの値を Bean 定義ファイル内で`${propertyName}`形式で埋め込んだり、Java クラスに`@Value("${propertyName}")` でインジェクションすることができる。
(2)	Spring MVC のアノテーションベースのデフォルト設定を行う。
(3)	アプリケーション層のクラスを管理する com.example.todo.app パッケージ配下を component-scan 対象とする。

Continued on next page

表 11.4 – continued from previous page

項番	説明
(4)	静的リソース (css, images, js など) アクセスのための設定を行う。 mapping 属性に URL のパスを、location 属性に物理的なパスの設定を行う。 この設定の場合<contextPath>/resources/app/css/styles.css に対してリクエストが来た場合、WEB-INF/resources/app/css/styles.css を探し、見つからなければクラスパス上 (src/main/resources や jar 内) の META-INF/resources/app/css/styles.css を探す。 どこにも styles.css が格納されていない場合は、404 エラーを返す。 ここでは cache-period 属性で静的リソースのキャッシュ時間 (3600 秒=60 分) も設定している。 cache-period="3600"と設定しても良いが、60 分であることを明示するために SpEL を使用して cache-period="#{60 * 60}" と書く方が分かりやすい。
(5)	コントローラ処理の Trace ログを出力するインターセプタを設定する。 /resources 配下を除く任意のパスに適用されるように設定する。
(6)	ViewResolver の設定を行う。 この設定により、例えばコントローラから view 名として hello が返却された場合には/WEB-INF/views/hello.jsp が実行される。

注釈: JPA 用のブランクプロジェクトを作成した場合は、<mvc:interceptors>の定義として、OpenEntityManagerInViewInterceptor の定義が有効な状態となっている。

```
<mvc:interceptor>
    <mvc:mapping path="/**" />
    <mvc:exclude-mapping path="/resources/**" />
    <bean
        class="org.springframework.orm.jpa.support.
↪OpenEntityManagerInViewInterceptor" />
</mvc:interceptor>
```

OpenEntityManagerInViewInterceptor は、EntityManager のライフサイクルの開始と終了を行う Interceptor である。この設定を追加することで、アプリケーション層 (Controller や、View クラス) での Lazy Load が、サポートされる。

spring-security.xml

spring-security.xml には、Spring Security に関する定義を行う。

作成したブランクプロジェクトの src/main/resources/META-INF/spring/spring-security.xml は、以下のような設定となっている。

なお、本チュートリアルでは Spring Security の設定ファイルの説明は割愛する。Spring Security の設定ファイルについては、「*Spring Security チュートリアル*」を参照されたい。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://www.springframework.org/schema/security"
    xsi:schemaLocation="
        http://www.springframework.org/schema/security https://www.springframework.
↪org/schema/security/spring-security.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
    ">

    <sec:http pattern="/resources/**" security="none"/>
    <sec:http>
        <sec:form-login/>
        <sec:logout/>
```

```
        <sec:access-denied-handler ref="accessDeniedHandler"/>
        <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
        <sec:session-management />
    </sec:http>

    <sec:authentication-manager />

    <!-- CSRF Protection -->
    <bean id="accessDeniedHandler"
        class="org.springframework.security.web.access.
↪DelegatingAccessDeniedHandler">
        <constructor-arg index="0">
            <map>
                <entry
                    key="org.springframework.security.web.csrf.
↪InvalidCsrfTokenException">
                    <bean
                        class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                        <property name="errorPage"
                            value="/WEB-INF/views/common/error/
↪invalidCsrfTokenError.jsp" />
                    </bean>
                </entry>
                <entry
                    key="org.springframework.security.web.csrf.
↪MissingCsrfTokenException">
                    <bean
                        class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                        <property name="errorPage"
                            value="/WEB-INF/views/common/error/
↪missingCsrfTokenError.jsp" />
                    </bean>
                </entry>
            </map>
        </constructor-arg>
        <constructor-arg index="1">
            <bean
                class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                <property name="errorPage"
                    value="/WEB-INF/views/common/error/accessDeniedError.jsp" />
            </bean>
        </constructor-arg>
    </bean>

    <!-- Put UserID into MDC -->
    <bean id="userIdMDCPutFilter" class="org.terasoluna.gfw.security.web.logging.
↪UserIdMDCPutFilter">
    </bean>

</beans>
```

logback.xml

logback.xml には、ログ出力に関する定義を行う。

作成したブランクプロジェクトの src/main/resources/logback.xml は、以下のような設定となっている。
なお、チュートリアルで使用しないログ設定についての説明は割愛する。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration>
<configuration>

    <!-- (1) -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tthread:%thread\tX-
↪Track:%X{X-Track}\tlevel:%-5level\tlogger:%-48logger{48}\tmessage:%msg%n]]></
↪pattern>
        </encoder>
    </appender>

    <appender name="APPLICATION_LOG_FILE" class="ch.qos.logback.core.rolling.
↪RollingFileAppender">
        <file>${app.log.dir:-log}/todo-application.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${app.log.dir:-log}/todo-application-%d{yyyyMMdd}.log
↪</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder>
            <charset>UTF-8</charset>
            <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tthread:%thread\tX-
↪Track:%X{X-Track}\tlevel:%-5level\tlogger:%-48logger{48}\tmessage:%msg%n]]></
↪pattern>
        </encoder>
    </appender>

    <appender name="MONITORING_LOG_FILE" class="ch.qos.logback.core.rolling.
↪RollingFileAppender">
        <file>${app.log.dir:-log}/todo-monitoring.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${app.log.dir:-log}/todo-monitoring-%d{yyyyMMdd}.log</
↪fileNamePattern>
```

```
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder>
            <charset>UTF-8</charset>
            <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tX-Track:%X{X-Track}
↪\tlevel:%-5level\tmessage:%msg%n]]></pattern>
        </encoder>
    </appender>

    <!-- Application Loggers -->
    <!-- (2) -->
    <logger name="com.example.todo">
        <level value="debug" />
    </logger>

    <!-- TERASOLUNA -->
    <logger name="org.terasoluna.gfw">
        <level value="info" />
    </logger>
    <!-- (3) -->
    <logger name="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor">
        <level value="trace" />
    </logger>
    <logger name="org.terasoluna.gfw.common.exception.ExceptionLogger">
        <level value="info" />
    </logger>
    <logger name="org.terasoluna.gfw.common.exception.ExceptionLogger.Monitoring"␣
↪additivity="false">
        <level value="error" />
        <appender-ref ref="MONITORING_LOG_FILE" />
    </logger>

    <!-- 3rdparty Loggers -->
    <logger name="org.springframework">
        <level value="warn" />
    </logger>

    <logger name="org.springframework.web.servlet">
        <level value="info" />
    </logger>

    <!--  REMOVE THIS LINE IF YOU USE JPA
    <logger name="org.hibernate.engine.transaction">
        <level value="debug" />
    </logger>
          REMOVE THIS LINE IF YOU USE JPA  -->
    <!--  REMOVE THIS LINE IF YOU USE MyBatis3
    <logger name="org.springframework.jdbc.datasource.DataSourceTransactionManager
↪">
        <level value="debug" />
    </logger>
          REMOVE THIS LINE IF YOU USE MyBatis3  -->
```

```
    <logger name="jdbc.sqltiming">
        <level value="debug" />
    </logger>

    <!-- only for development -->
    <logger name="jdbc.resultsettable">
        <level value="debug" />
    </logger>

    <root level="warn">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="APPLICATION_LOG_FILE" />
    </root>

</configuration>
```

項番	説明
(1)	標準出力でログを出力するアペンダを設定。
(2)	com.example.todo パッケージ以下は debug レベル以上を出力するように設定。
(3)	spring-mvc.xml に設定した `TraceLoggingInterceptor` に出力されるように trace レベルで設定。

注釈: O/R Mapper を使用するブランクプロジェクトを作成した場合は、トランザクション制御関連のログを出力するロガーが有効な状態となっている。

- JPA 用のブランクプロジェクト

```
<logger name="org.hibernate.engine.transaction">
    <level value="debug" />
</logger>
```

- MyBatis3 用のブランクプロジェクト

```
<logger name="org.springframework.jdbc.datasource.DataSourceTransactionManager
↪">
    <level value="debug" />
</logger>
```

11.2 チュートリアル (Todo アプリケーション REST 編)

11.2.1 はじめに

このチュートリアルで学ぶこと

- TERASOLUNA Server Framework for Java (5.x) による基本的な RESTful Web サービスの構築方法

対象読者

- チュートリアル *(Todo* アプリケーション*)* を実施している。

検証環境

本チュートリアルは以下の環境で動作確認している。

REST Client として、Google Chrome の拡張機能を使用するため、Web Browser は Google Chrome を使用する。

種別	プロダクト
REST Client	DHC REST Client 1.2.3
上記以外のプロダクト	チュートリアル *(Todo* アプリケーション*)* と同様

11.2.2 環境構築

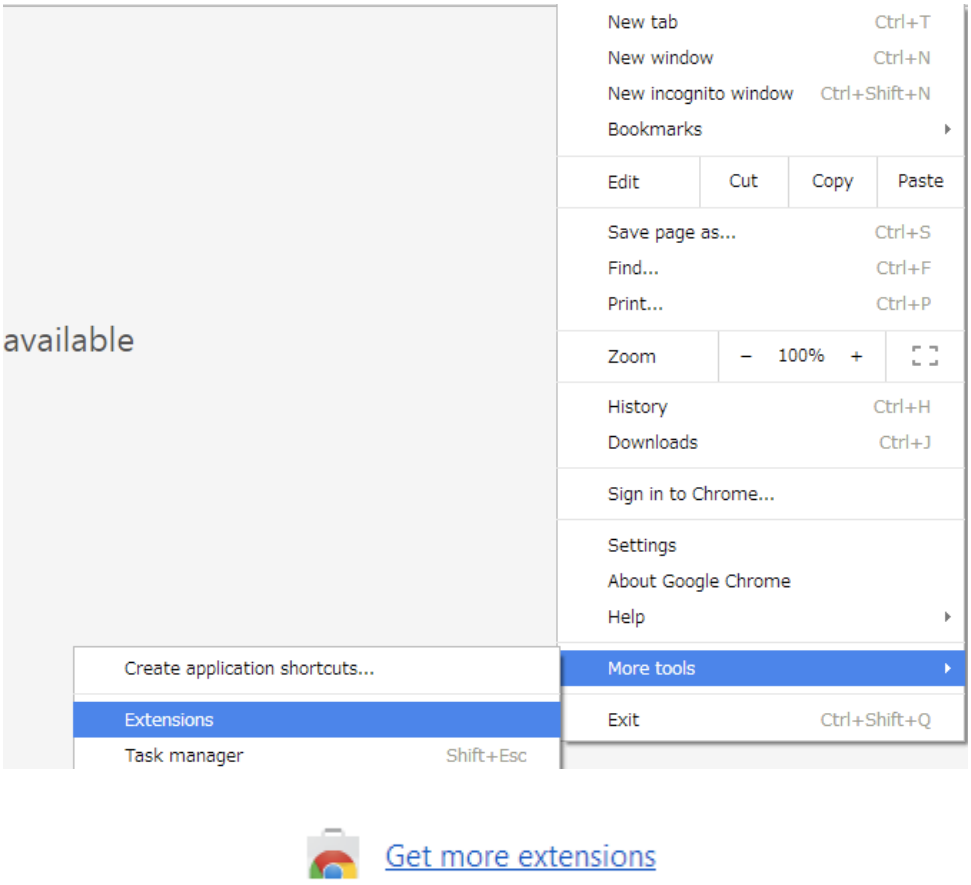
Java, STS, Maven, Google Chrome については、チュートリアル *(Todo* アプリケーション*)* を実施する事でインストール済みの状態である事を前提とする。

DHC のインストール

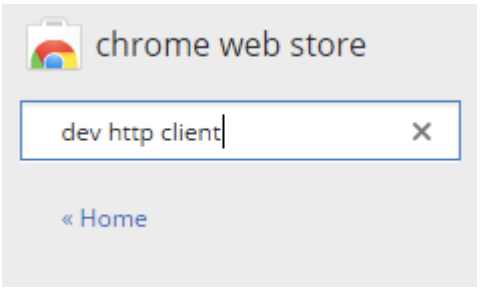
REST クライアントして、Chrome の拡張機能である「DHC」をインストールする。

Chrome の「Tools」→「Extensions」を選択する。

「Get more extensions」のリンクを押下する。



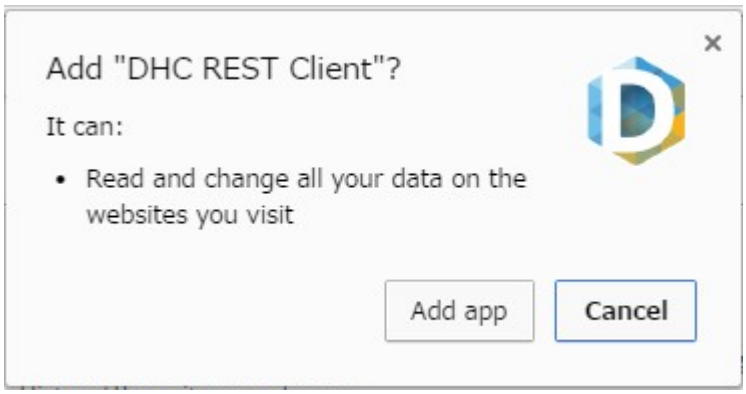
検索フォームに「dev http client」を入力して検索する。



DHC REST Client の「+ ADD TO CHROME」ボタンを押下する。



「Add app」ボタンを押下する。



Chrome のアプリケーション一覧を開く (ブラウザのアドレスバーに「chrome://apps/」を指定して開く) と、DHC が追加されている。



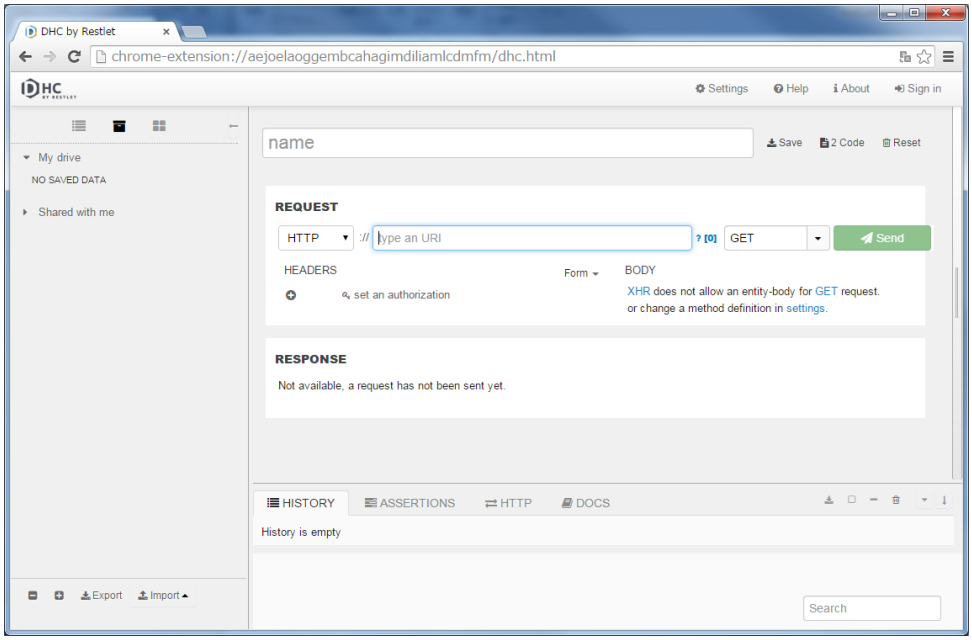
DHC をクリックする。

以下の画面が表示されれば、インストール完了となる。

この画面は、ブラウザのアドレスバーに「chrome-extension://aejoelaoggembcahagimdiliamlcdmfm/dhc.html」を入力する事で開く事もできる。

プロジェクト作成

本チュートリアルでは、「チュートリアル *(Todo* アプリケーション*)*」で作成したプロジェクトに対して、RESTful Web サービスを追加する手順となっている。



そのため、「チュートリアル *(Todo* アプリケーション*)*」で作成したプロジェクトが残っていない場合は、再度「チュートリアル *(Todo* アプリケーション*)*」を実施してプロジェクトを作成してほしい。

注釈: 再度「チュートリアル *(Todo* アプリケーション*)*」を実施する場合は、ドメイン層の作成まで行えば本チュートリアルを進める事ができる。

11.2.3 REST API の作成

本チュートリアルでは、todo テーブルで管理しているデータ (以降、「Todo リソース」呼ぶ) を Web 上に公開するための REST API を作成する。

API 名	HTTP メソッド	パス	ステータスコード	説明
GET Todos	GET	`/api/v1/todos`	200 (OK)	Todo リソースを全件取得する。
POST Todos	POST	`/api/v1/todos`	201 (Created)	Todo リソースを新規作成する。
GET Todo	GET	`/api/v1/todos/{todoId}`	200 (OK)	Todo リソースを一件取得する。
PUT Todo	PUT	`/api/v1/todos/{todoId}`	200 (OK)	Todo リソースを完了状態に更新する。
DELETE Todo	DELETE	`/api/v1/todos/{todoId}`	204 (No Content)	Todo リソースを削除する。

ちなみに: パス内に含まれている`{todoId}`は、パス変数と呼ばれ、任意の可変値を扱う事ができる。パス変数を使用する事で、`GET /api/v1/todos/123` と `GET /api/v1/todos/456` を同じ API で扱う事ができる。

本チュートリアルでは、Todo を一意に識別するための ID(Todo ID) をパス変数として扱っている。

API 仕様

HTTP リクエストとレスポンスの具体例を用いて、本チュートリアルで作成する REST API のインタフェース仕様を示す。

本質的ではない HTTP ヘッダー等は例から除いている。

GET Todos

[リクエスト]

```
> GET /todo/api/v1/todos HTTP/1.1
```

[レスポンス]

作成済みの Todo リソースのリストを JSON 形式で返却する。

```
< HTTP/1.1 200 OK
< Content-Type: application/json;charset=UTF-8
<
[{"todoId":"9aef3ee3-30d4-4a7c-be4a-bc184ca1d558","todoTitle":"Hello World!",
↪"finished":false,"createdAt":"2014-02-25T02:21:48.493+0000"}]
```

POST Todos

[リクエスト]

新規作成する Todo リソースの内容 (タイトル) を JSON 形式で指定する。

```
> POST /todo/api/v1/todos HTTP/1.1
> Content-Type: application/json
> Content-Length: 29
>
{"todoTitle": "Study Spring"}
```

[レスポンス]

作成した Todo リソースを JSON 形式で返却する。

```
< HTTP/1.1 201 Created
< Content-Type: application/json;charset=UTF-8
<
{"todoId":"d6101d61-b22c-48ee-9110-e106af6a1404","todoTitle":"Study Spring",
↪"finished":false,"createdAt":"2014-02-25T04:05:58.752+0000"}
```

GET Todo

[リクエスト]

パス変数「todoId」に、取得対象の Todo リソースの ID を指定する。

下記例では、パス変数「todoId」に 9aef3ee3-30d4-4a7c-be4a-bc184ca1d558 を指定している。

```
> GET /todo/api/v1/todos/9aef3ee3-30d4-4a7c-be4a-bc184ca1d558 HTTP/1.1
```

[レスポンス]

パス変数「todoId」に一致する Todo リソースを JSON 形式で返却する。

```
< HTTP/1.1 200 OK
< Content-Type: application/json;charset=UTF-8
<
{"todoId":"9aef3ee3-30d4-4a7c-be4a-bc184ca1d558","todoTitle":"Hello World!",
↪"finished":false,"createdAt":"2014-02-25T02:21:48.493+0000"}
```

PUT Todo

[リクエスト]

パス変数「todoId」に、更新対象の Todo の ID を指定する。

PUT Todo では、Todo リソースを完了状態に更新するだけなので、リクエスト BODY を受け取らないインタフェース仕様にしている。

```
> PUT /todo/api/v1/todos/9aef3ee3-30d4-4a7c-be4a-bc184ca1d558 HTTP/1.1
```

[レスポンス]

パス変数「todoId」に一致する Todo リソースを完了状態 (finished フィールドを true) に更新し、JSON 形式で返却する。

```
< HTTP/1.1 200 OK
< Content-Type: application/json;charset=UTF-8
<
{"todoId":"9aef3ee3-30d4-4a7c-be4a-bc184ca1d558","todoTitle":"Hello World!",
↪"finished":true,"createdAt":"2014-02-25T02:21:48.493+0000"}
```

DELETE Todo

[リクエスト]

パス変数「todoId」に、削除対象の Todo リソースの ID を指定する。

```
> DELETE /todo/api/v1/todos/9aef3ee3-30d4-4a7c-be4a-bc184ca1d558 HTTP/1.1
```

[レスポンス]

DELETE Todo では、Todo リソースの削除が完了した事で返却するリソースが存在しなくなった事を示すために、レスポンス BODY を返却しないインタフェース仕様にしている。

```
< HTTP/1.1 204 No Content
```

エラー応答

REST API でエラーが発生した場合は、JSON 形式でエラー内容を返却する。

以下に代表的なエラー発生時のレスポンス仕様について記載する。

下記以外のエラーパターンもあるが、本チュートリアルでは説明は割愛する。

チュートリアル *(Todo* アプリケーション*)* では、エラーメッセージはプログラムの中でハードコーディングしていたが、本チュートリアルでは、エラーメッセージはエラーコードをキーにプロパティファイルから取得するように修正する。

[入力チェックエラー発生時のレスポンス仕様]

```
< HTTP/1.1 400 Bad Request
< Content-Type: application/json;charset=UTF-8
<
{"code":"E400","message":"[E400] The requested Todo contains invalid values.",
↪"details":[{"code":"NotNull","message":"todoTitle may not be null.",target:
↪"todoTitle"}]}
```

[業務エラー発生時のレスポンス仕様]

```
< HTTP/1.1 409 Conflict
< Content-Type: application/json;charset=UTF-8
<
{"code":"E002","message":"[E002] The requested Todo is already finished.␣
↪(id=353fb5db-151a-4696-9b4a-b958358a5ab3)"}
```

[リソース未検出時のレスポンス仕様]

```
< HTTP/1.1 404 Not Found
< Content-Type: application/json;charset=UTF-8
<
{"code":"E404","message":"[E404] The requested Todo is not found. (id=353fb5db-
↪151a-4696-9b4a-b958358a5ab2)"}
```

[システムエラー発生時のレスポンス仕様]

```
< HTTP/1.1 500 Internal Server Error
< Content-Type: application/json;charset=UTF-8
<
{"code":"E500","message":"[E500] System error occurred."}
```

REST API 用の DispatcherServlet を用意

まず、REST API 用のリクエストを処理するための `DispatcherServlet` の定義を追加する。

web.xml の修正

REST API 用の設定を追加する。

`src/main/webapp/WEB-INF/web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/
↪ns/javaee/web-app_3_0.xsd"
```

```
    version="3.0">

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</
↪listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <!-- Root ApplicationContext -->
        <param-value>
            classpath*:META-INF/spring/applicationContext.xml
            classpath*:META-INF/spring/spring-security.xml
        </param-value>
    </context-param>

    <listener>
        <listener-class>org.terasoluna.gfw.web.logging.
↪HttpSessionEventLoggingListener</listener-class>
    </listener>

    <filter>
        <filter-name>MDCClearFilter</filter-name>
        <filter-class>org.terasoluna.gfw.web.logging.mdc.MDCClearFilter</filter-
↪class>
    </filter>
    <filter-mapping>
        <filter-name>MDCClearFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>exceptionLoggingFilter</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
↪class>
    </filter>
    <filter-mapping>
        <filter-name>exceptionLoggingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>XTrackMDCPutFilter</filter-name>
        <filter-class>org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter</
↪filter-class>
    </filter>
    <filter-mapping>
        <filter-name>XTrackMDCPutFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>CharacterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</
↪filter-class>
```

```
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
↪class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- (1) -->
    <servlet>
        <servlet-name>restApiServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
↪class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <!-- ApplicationContext for Spring MVC (REST) -->
            <param-value>classpath*:META-INF/spring/spring-mvc-rest.xml</param-
↪value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- (2) -->
    <servlet-mapping>
        <servlet-name>restApiServlet</servlet-name>
        <url-pattern>/api/v1/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
↪class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <!-- ApplicationContext for Spring MVC -->
            <param-value>classpath*:META-INF/spring/spring-mvc.xml</param-value>
        </init-param>
```

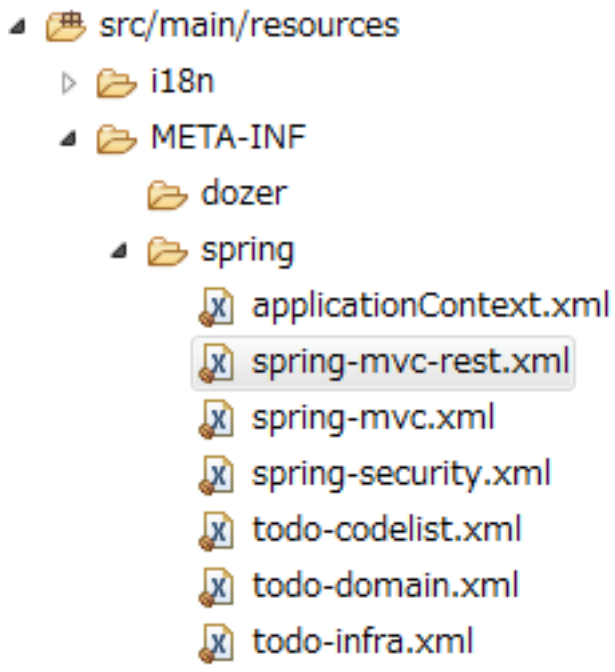
```
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <jsp-config>
        <jsp-property-group>
            <url-pattern>*.jsp</url-pattern>
            <el-ignored>false</el-ignored>
            <page-encoding>UTF-8</page-encoding>
            <scripting-invalid>false</scripting-invalid>
            <include-prelude>/WEB-INF/views/common/include.jsp</include-prelude>
        </jsp-property-group>
    </jsp-config>

    <error-page>
        <error-code>500</error-code>
        <location>/WEB-INF/views/common/error/systemError.jsp</location>
    </error-page>

    <error-page>
        <error-code>404</error-code>
        <location>/WEB-INF/views/common/error/resourceNotFoundError.jsp</location>
    </error-page>

    <error-page>
        <exception-type>java.lang.Exception</exception-type>
        <location>/WEB-INF/views/common/error/unhandledSystemError.html</location>
    </error-page>

    <session-config>
        <!-- 30min -->
        <session-timeout>30</session-timeout>
        <cookie-config>
            <http-only>true</http-only>
            <!-- <secure>true</secure> -->
        </cookie-config>
        <tracking-mode>COOKIE</tracking-mode>
    </session-config>

</web-app>
```

項番	説明
(1)	初期化パラメータ「contextConfigLocation」に、REST 用の Spring MVC 設定ファイルを指定する。 本チュートリアルでは、クラスパス上にある「META-INF/spring/spring-mvc-rest.xml」を指定している。
(2)	<url-pattern>要素に、REST API 用の DispatcherServlet にマッピングする URL のパターンを指定する。 本チュートリアルでは、/api/v1/から始まる場合はリクエストを REST API へのリクエストとして REST API 用の DispatcherServlet へマッピングしている。

spring-mvc-rest.xml の作成

REST 用の Spring MVC 設定ファイルを作成する。

REST 用の Spring MVC 設定ファイルは以下のような定義となる。



src/main/resources/META-INF/spring/spring-mvc-rest.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc https://www.
↪springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util https://www.springframework.org/
↪schema/util/spring-util.xsd
        http://www.springframework.org/schema/context https://www.springframework.
↪org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop https://www.springframework.org/
↪schema/aop/spring-aop.xsd
    ">

    <!-- (1) -->
    <context:property-placeholder
        location="classpath*:/META-INF/spring/*.properties" />

    <mvc:annotation-driven>
        <mvc:message-converters register-defaults="false">
            <!-- (2) -->
            <bean
                class="org.springframework.http.converter.json.
↪MappingJackson2HttpMessageConverter">
                <!-- (3) -->
                <property name="objectMapper">
                    <bean class="com.fasterxml.jackson.databind.ObjectMapper">
                        <property name="dateFormat">
                            <!-- (4) -->
                            <bean class="com.fasterxml.jackson.databind.util.
↪StdDateFormat"/>
                        </property>
                    </bean>
                </property>
            </bean>
        </mvc:message-converters>
    </mvc:annotation-driven>

    <context:component-scan base-package="com.example.todo.api" /> <!-- (5) -->

    <!-- (6) -->
    <mvc:interceptors>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
                class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor" />
        </mvc:interceptor>
```

```
    </mvc:interceptors>

    <!-- (7) -->
    <!-- Setting AOP. -->
    <bean id="handlerExceptionResolverLoggingInterceptor"
        class="org.terasoluna.gfw.web.exception.
↪HandlerExceptionResolverLoggingInterceptor">
        <property name="exceptionLogger" ref="exceptionLogger" />
    </bean>
    <aop:config>
        <aop:advisor advice-ref="handlerExceptionResolverLoggingInterceptor"
            pointcut="execution(* org.springframework.web.servlet.
↪HandlerExceptionResolver.resolveException(..))" />
    </aop:config>

</beans>
```

項番	説明
(1)	アプリケーション層のコンポーネントでプロパティファイルに定義されている値を参照する必要がある場合は、<context:property-placeholder>要素を使用してプロパティファイルを読み込む必要がある。
(2)	<mvc:message-converters>に、Controller の引数と返り値で扱う JavaBean をシリアライズ/デシリアライズするためのクラス (org.springframework.http.converter.HttpMessageConverter) を設定する。 HttpMessageConverter は複数設定する事ができるが、本チュートリアルでは JSON しか使用しないため、MappingJackson2HttpMessageConverter のみ指定している。
(3)	MappingJackson2HttpMessageConverter の objectMapper プロパティに、Jackson より提供されている ObjectMapper(「JSON <-> JavaBean」の変換を行うためのコンポーネント) を指定する。 本チュートリアルでは、日時型のフォーマットをカスタマイズした ObjectMapper を指定している。カスタマイズする必要がない場合は objectMapper プロパティは省略可能である。
(4)	ObjectMapper の dateFormat プロパティに、日時型フィールドの形式を指定する。 本チュートリアルでは、java.util.Date オブジェクトをシリアライズする際に ISO-8601 形式とする。Date オブジェクトをシリアライズする際に ISO-8601 形式にする場合は、com.fasterxml.jackson.databind.util.StdDateFormat を設定する事で実現する事ができる。
(5)	REST API 用のパッケージ配下のコンポーネントをスキャンする。 本チュートリアルでは、REST API 用のパッケージを com.example.todo.api にしている。画面遷移用の Controller は、app パッケージ配下に格納していたが、REST API 用の Controller は、api パッケージ配下に格納する事を推奨する。
(6)	Controller の処理開始、終了時の情報をログに出力するために、共通ライブラリから提供されている TraceLoggingInterceptor を定義する。
(7)	Spring MVC のフレームワークでハンドリングされた例外を、ログ出力するための AOP 定義を指定する。

REST API 用の Spring Security の定義追加

ブランクプロジェクトでは、CSRF 対策といった、Spring Security のセキュリティ対策機能が有効になっている。

REST API を使って構築する Web アプリケーションでも、セキュリティ対策機能は必要である。ただし、本チュートリアルの目的として、

セキュリティ対策の話題は本質的ではないため、機能を無効化し、説明も割愛する。

以下の設定を追加する事で、Spring Security のセキュリティ対策機能を無効化することができる。

src/main/resources/META-INF/spring/spring-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://www.springframework.org/schema/security"
    xsi:schemaLocation="
        http://www.springframework.org/schema/security https://www.springframework.
↪org/schema/security/spring-security.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
    ">

    <sec:http pattern="/resources/**" security="none"/>

    <!-- (1) -->
    <sec:http pattern="/api/v1/**" security="none"/>

    <sec:http>
        <sec:form-login/>
        <sec:logout/>
        <sec:access-denied-handler ref="accessDeniedHandler"/>
        <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
        <sec:session-management />
    </sec:http>

    <sec:authentication-manager />

    <!-- CSRF Protection -->
    <bean id="accessDeniedHandler"
        class="org.springframework.security.web.access.
↪DelegatingAccessDeniedHandler">
        <constructor-arg index="0">
            <map>
                <entry
                    key="org.springframework.security.web.csrf.
↪InvalidCsrfTokenException">
                    <bean
                        class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                        <property name="errorPage"
                            value="/WEB-INF/views/common/error/
↪invalidCsrfTokenError.jsp" />
                    </bean>
                </entry>
                <entry
                    key="org.springframework.security.web.csrf.
↪MissingCsrfTokenException">
                    <bean
                        class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
```

```
                        <property name="errorPage"
                            value="/WEB-INF/views/common/error/
↪missingCsrfTokenError.jsp" />
                    </bean>
                </entry>
            </map>
        </constructor-arg>
        <constructor-arg index="1">
            <bean
                class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                <property name="errorPage"
                    value="/WEB-INF/views/common/error/accessDeniedError.jsp" />
            </bean>
        </constructor-arg>
    </bean>

    <!-- Put UserID into MDC -->
    <bean id="userIdMDCPutFilter" class="org.terasoluna.gfw.security.web.logging.
↪UserIdMDCPutFilter">
    </bean>

</beans>
```

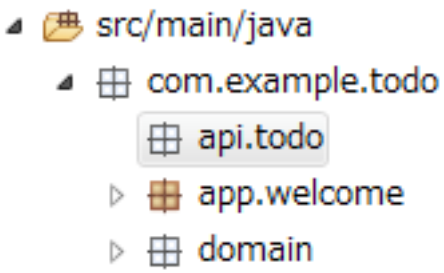
項番	説明
(1)	REST API 用の Spring Security のセキュリティ機能を無効にする定義を追加する。 <sec:http>要素の pattern 属性に、REST API 用のリクエストパスの URL パターンを指定している。 本チュートリアルでは/api/v1/で始まるリクエストパスを REST API 用のリクエストパスとして扱う。

REST API 用パッケージの作成

REST API 用のクラスを格納するパッケージを作成する。

REST API 用のクラスを格納するルートパッケージのパッケージ名は api として、配下にリソース毎のパッケージ (リソース名の小文字) を作成する事を推奨する。

本チュートリアルで扱うリソースのリソース名は Todo なので、com.example.todo.api.todo パッケージを作成する。



注釈: 作成したパッケージに格納するクラスは、通常以下の 3 種類となる。作成するクラスのクラス名は、以下のネーミングルールとする事を推奨する。

- [リソース名]Resource
- [リソース名]RestController
- [リソース名]Helper (必要に応じて)

本チュートリアルで扱うリソースのリソース名が Todo なので、

- TodoResource
- TodoRestController

を作成する。

本チュートリアルでは、TodoRestHelper は作成しない。

Resource クラスの作成

Todo リソースを表現する TodoResource クラスを作成する。

本ガイドラインでは、REST API の入出力となる JSON(または XML) を表現する Java Bean を **Resource** クラスと呼ぶ。

src/main/java/com/example/todo/api/todo/TodoResource.java

```
package com.example.todo.api.todo;

import java.io.Serializable;
import java.util.Date;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class TodoResource implements Serializable {
```

```
    private static final long serialVersionUID = 1L;

    private String todoId;

    @NotNull
    @Size(min = 1, max = 30)
    private String todoTitle;

    private boolean finished;

    private Date createdAt;

    public String getTodoId() {
        return todoId;
    }

    public void setTodoId(String todoId) {
        this.todoId = todoId;
    }

    public String getTodoTitle() {
        return todoTitle;
    }

    public void setTodoTitle(String todoTitle) {
        this.todoTitle = todoTitle;
    }

    public boolean isFinished() {
        return finished;
    }

    public void setFinished(boolean finished) {
        this.finished = finished;
    }

    public Date getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Date createdAt) {
        this.createdAt = createdAt;
    }
}
```

注釈: DomainObject クラス (本チュートリアルでは `Todo` クラス) があるにも関わらず、Resource クラスを作成する理由は、クライアントとの入出力で使用するインタフェース上の情報と、業務処理で扱う情報は必ずしも一致しないためである。

これらを混同して使用すると、アプリケーション層の影響がドメイン層におよび、保守性を低下させる。

DomainObject と Resource クラスは別々に作成し、Dozer 等の BeanMapper を利用してデータ変換を行うことを推奨する。

Resource クラスは Form クラスと役割が似ているが、Form クラスは HTML の`<form>` タグを JavaBean で表現したもの、Resource クラスは REST API の入出力を JavaBean で表現したものであり、本質的には異なるものである。

ただし、実体としては Bean Validation のアノテーションを付与した JavaBean であり、Controller クラスと同じパッケージに格納することから、Form クラスとほぼ同じである。

Controller クラスの作成

`TodoResource` の REST API を提供する `TodoRestController` クラスを作成する。

`src/main/java/com/example/todo/api/todo/TodoRestController.java`

```
package com.example.todo.api.todo;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController // (1)
@RequestMapping("todos") // (2)
public class TodoRestController {

}
```

項番	説明
(1)	`@RestController` を指定する。 `@RestController` の詳細については、 *RestController クラスの作成* を参照されたい。
(2)	リソースのパスを指定する。 `/api/v1/`の部分は web.xml に定義しているため、この設定を行うことで`/<contextPath>/api/v1/todos` というパスにマッピングされる。

GET Todos の実装

作成済みの Todo リソースを全件取得する API(GET Todos) の処理を、TodoRestController の getTodos メソッドに実装する。

src/main/java/com/example/todo/api/todo/TodoRestController.java

```
package com.example.todo.api.todo;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import javax.inject.Inject;

import org.dozer.Mapper;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.todo.domain.model.Todo;
import com.example.todo.domain.service.todo.TodoService;

@RestController
@RequestMapping("todos")
public class TodoRestController {

    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    @GetMapping // (1)
    @ResponseStatus(HttpStatus.OK) // (2)
    public List<TodoResource> getTodos() {
        Collection<Todo> todos = todoService.findAll();
        List<TodoResource> todoResources = new ArrayList<>();
        for (Todo todo : todos) {
            todoResources.add(beanMapper.map(todo, TodoResource.class)); // (3)
        }
        return todoResources; // (4)
    }

}
```

項番	説明
(1)	メソッドが GET のリクエストを処理するために、`@GetMapping` アノテーションを設定する。
(2)	応答する HTTP ステータスコードを`@ResponseStatus` アノテーションに指定する。 HTTP ステータスとして、」200 OK」を設定するため、`value` 属性には `HttpStatus.OK` を設定する。
(3)	`TodoService` の `findAll` メソッドから返却された `Todo` オブジェクトを、応答する JSON を表現する `TodoResource` 型のオブジェクトに変換する。 `Todo` と `TodoResource` の変換処理は、Dozer の `org.dozer.Mapper` インタフェースを使うと便利である。
(4)	`List<TodoResource>`オブジェクトを返却することで、`spring-mvc-rest.xml` に定義した `MappingJackson2HttpMessageConverter` によって JSON にシリアライズされる。

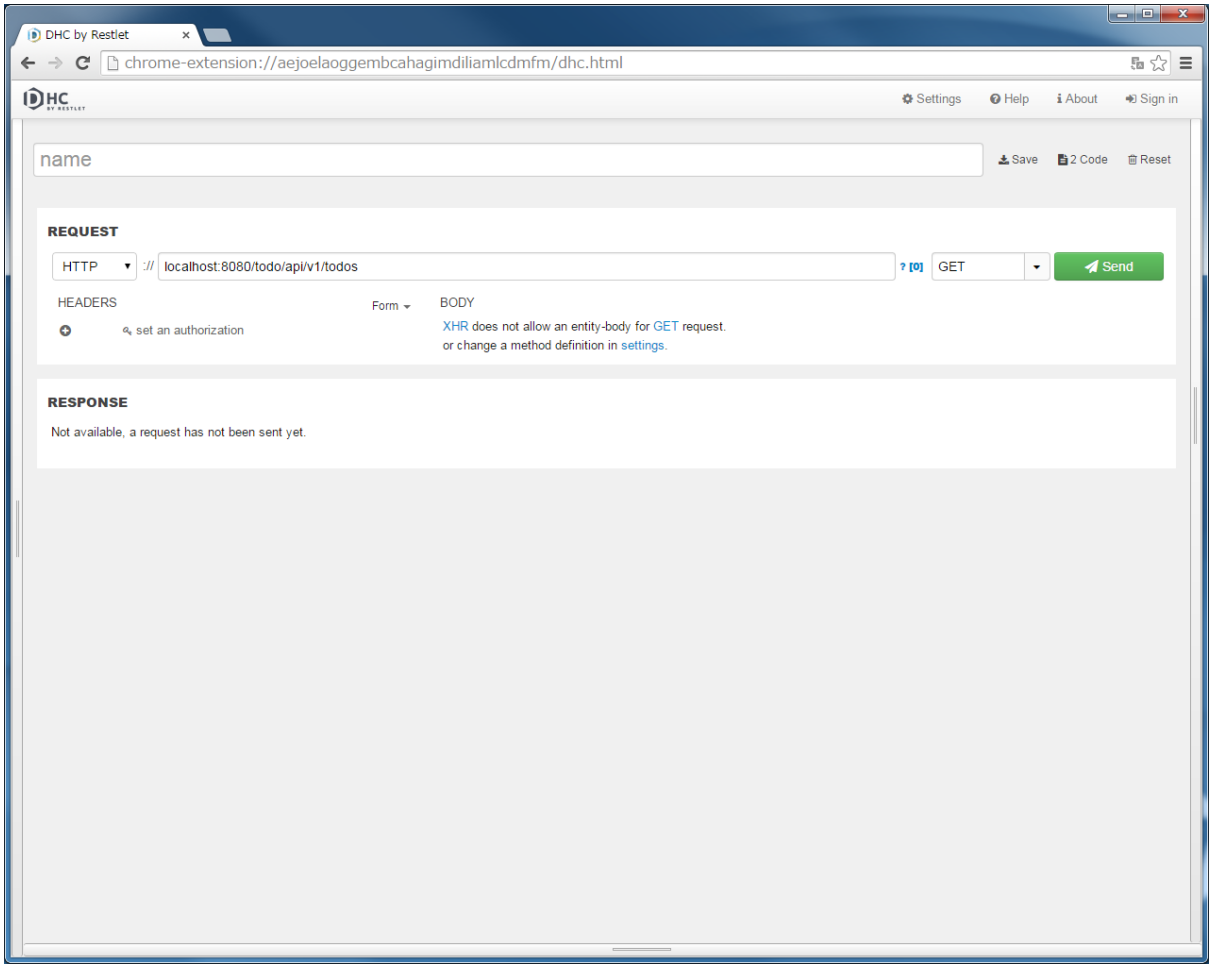
Application Server を起動し、実装した API の動作確認を行う。

REST API(Get Todos) にアクセスする。

DHC を開いて URL に `localhost:8080/todo/api/v1/todos` を入力し、メソッドに GET を指定して、」Send」ボタンをクリックする。

以下のように「RESPONSE」の「BODY」に実行結果の JSON が表示される。

現時点ではデータが何も登録されていないため、空配列である `[]` が返却される。

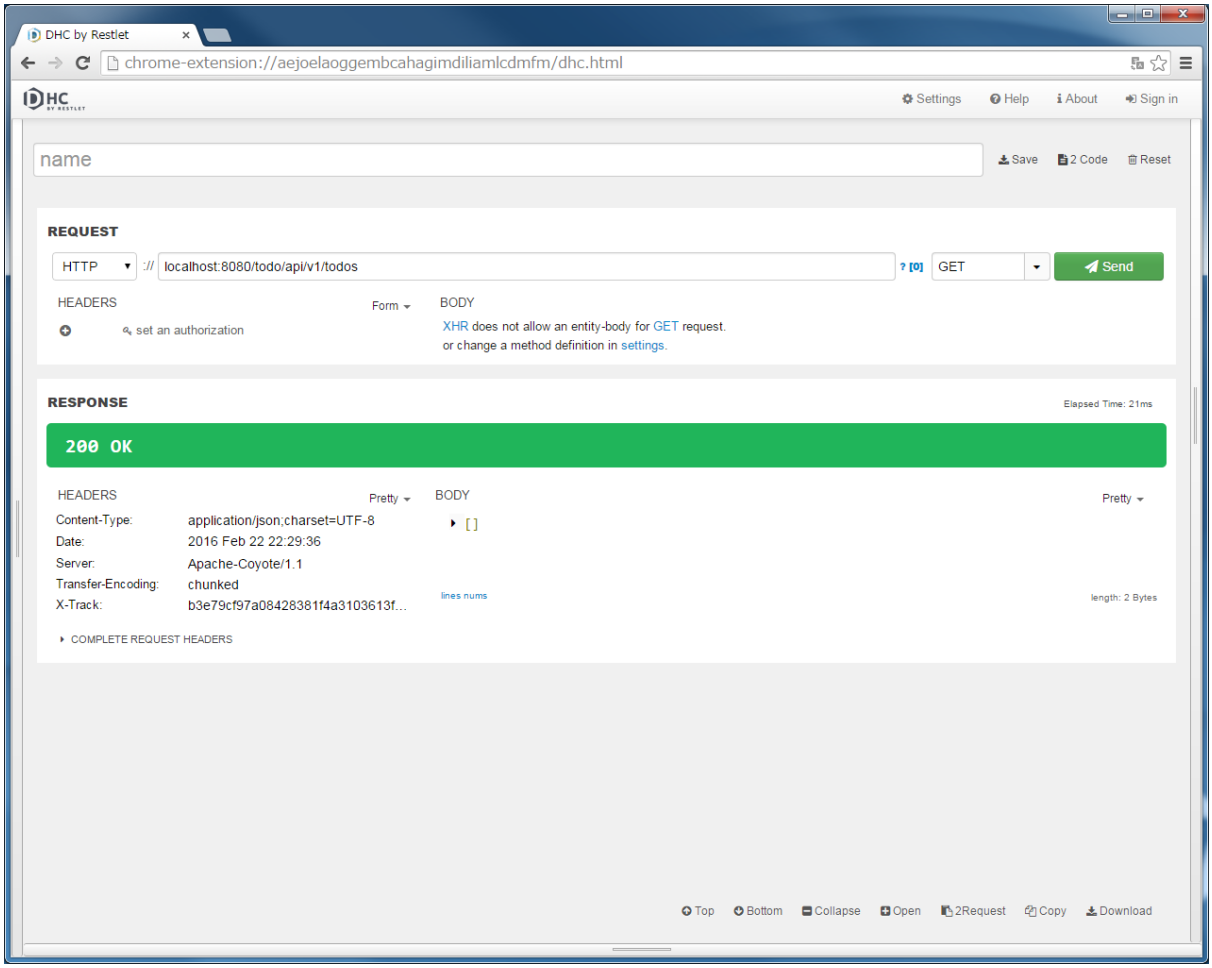


POST Todos の実装

Todo リソースを新規作成する API(POST Todos) の処理を、`TodoRestController` の `postTodos` メソッドに実装する。

`src/main/java/com/example/todo/api/todo/TodoRestController.java`

```
package com.example.todo.api.todo;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import javax.inject.Inject;

import org.dozer.Mapper;
import org.springframework.http.HttpStatus;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
```




```
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.todo.domain.model.Todo;
import com.example.todo.domain.service.todo.TodoService;

@RestController
@RequestMapping("todos")
public class TodoRestController {

    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    public List<TodoResource> getTodos() {
        Collection<Todo> todos = todoService.findAll();
        List<TodoResource> todoResources = new ArrayList<>();
        for (Todo todo : todos) {
```

```
            todoResources.add(beanMapper.map(todo, TodoResource.class));
        }
        return todoResources;
    }

    @PostMapping // (1)
    @ResponseStatus(HttpStatus.CREATED) // (2)
    public TodoResource postTodos(@RequestBody @Validated TodoResource␣
↪todoResource) { // (3)
        Todo createdTodo = todoService.create(beanMapper.map(todoResource, Todo.
↪class)); // (4)
        TodoResource createdTodoResponse = beanMapper.map(createdTodo,␣
↪TodoResource.class); // (5)
        return createdTodoResponse; // (6)
    }

}
```

項番	説明
(1)	メソッドが POST のリクエストを処理するために、`@PostMapping` アノテーションを設定する。
(2)	応答する HTTP ステータスコードを`@ResponseStatus` アノテーションに指定する。 HTTP ステータスとして、」201 Created」を設定するため、`value` 属性には `HttpStatus.CREATED` を設定する。
(3)	HTTP リクエストの Body(JSON) を JavaBean にマッピングするために、`@RequestBody` アノテーションをマッピング対象の `TodoResource` クラスに付与する。 また、入力チェックするために`@Validated` も付与する。例外ハンドリングは別途行う必要がある。
(4)	`TodoResource` を `Todo` クラスに変換後、`TodoService` の `create` メソッドを実行し、Todo リソースを新規作成する。
(5)	`TodoService` の `create` メソッドによって新規作成された `Todo` オブジェクトを、応答する JSON を表現する `TodoResource` 型に変換する。
(6)	`TodoResource` オブジェクトを返却することで、`spring-mvc-rest.xml` に定義した `MappingJackson2HttpMessageConverter` によって JSON にシリアライズされる。

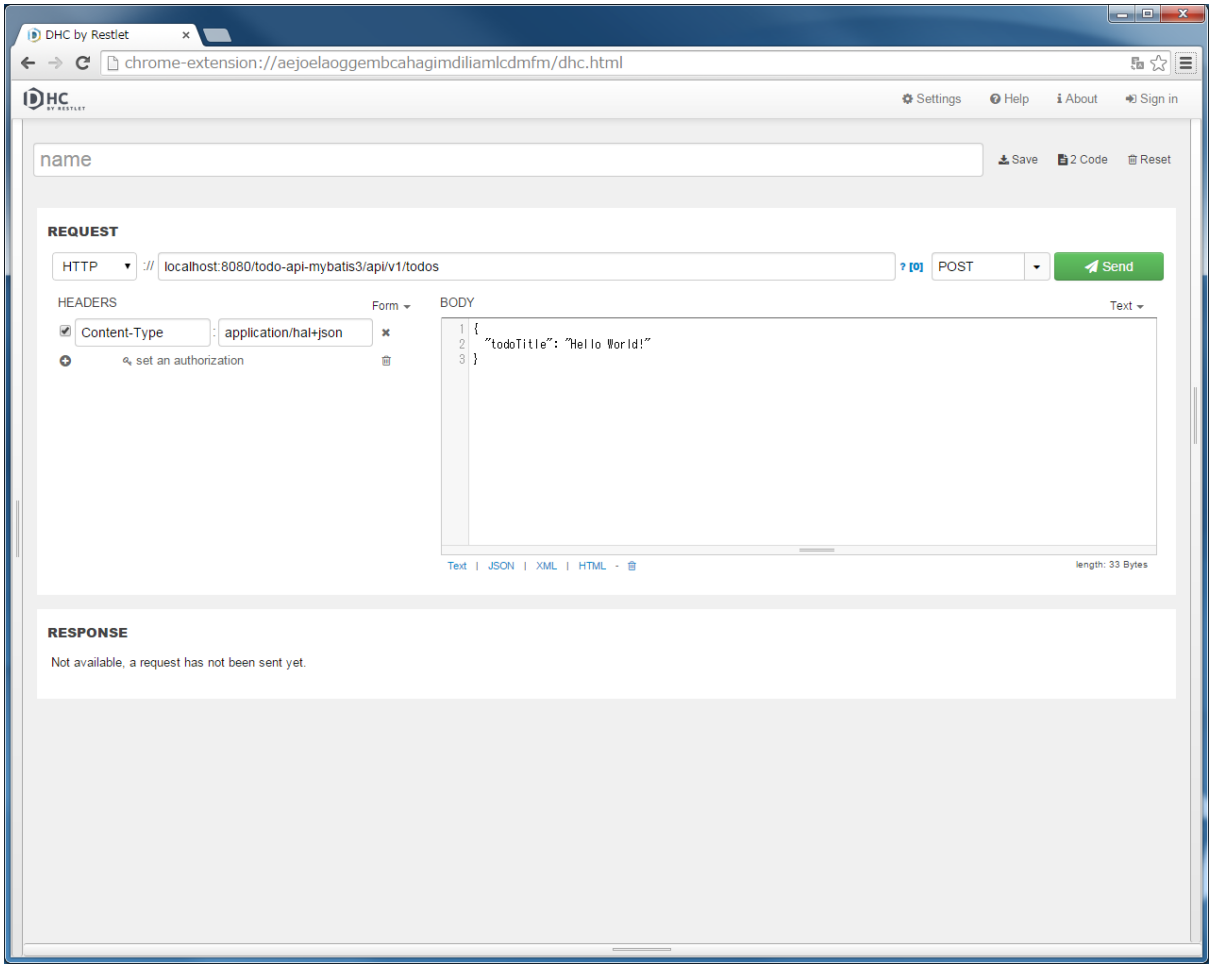
DHC を使用して、実装した API の動作確認を行う。

DHC を開いて URL に `localhost:8080/todo/api/v1/todos` を入力し、メソッドに POST を指定する。

「REQUEST」の「BODY」に以下の JSON を入力する。

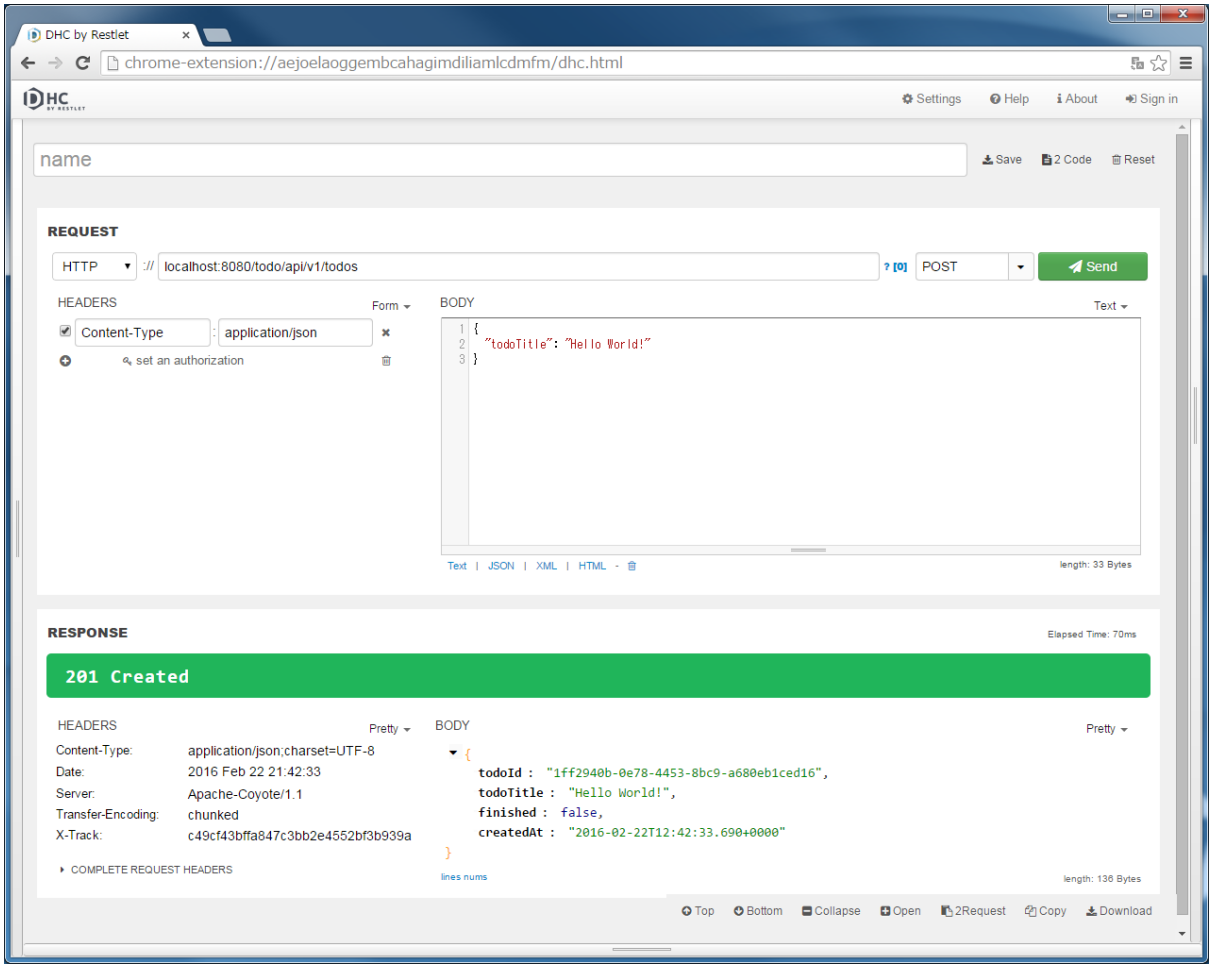
```
{
  "todoTitle": "Hello World!"
}
```

また、「REQUEST」の「HEADERS」の「+」ボタンで HTTP ヘッダーを追加し、「Content-Type」に「application/json」を設定後、」Send」ボタンをクリックする。



「201 Created」の HTTP ステータスが返却され、「RESPONSE」の「Body」に新規作成された Todo リソースの JSON が表示される。

この状態で再び GET Todos を実行すると、作成した Todo リソースを含む配列が返却される。



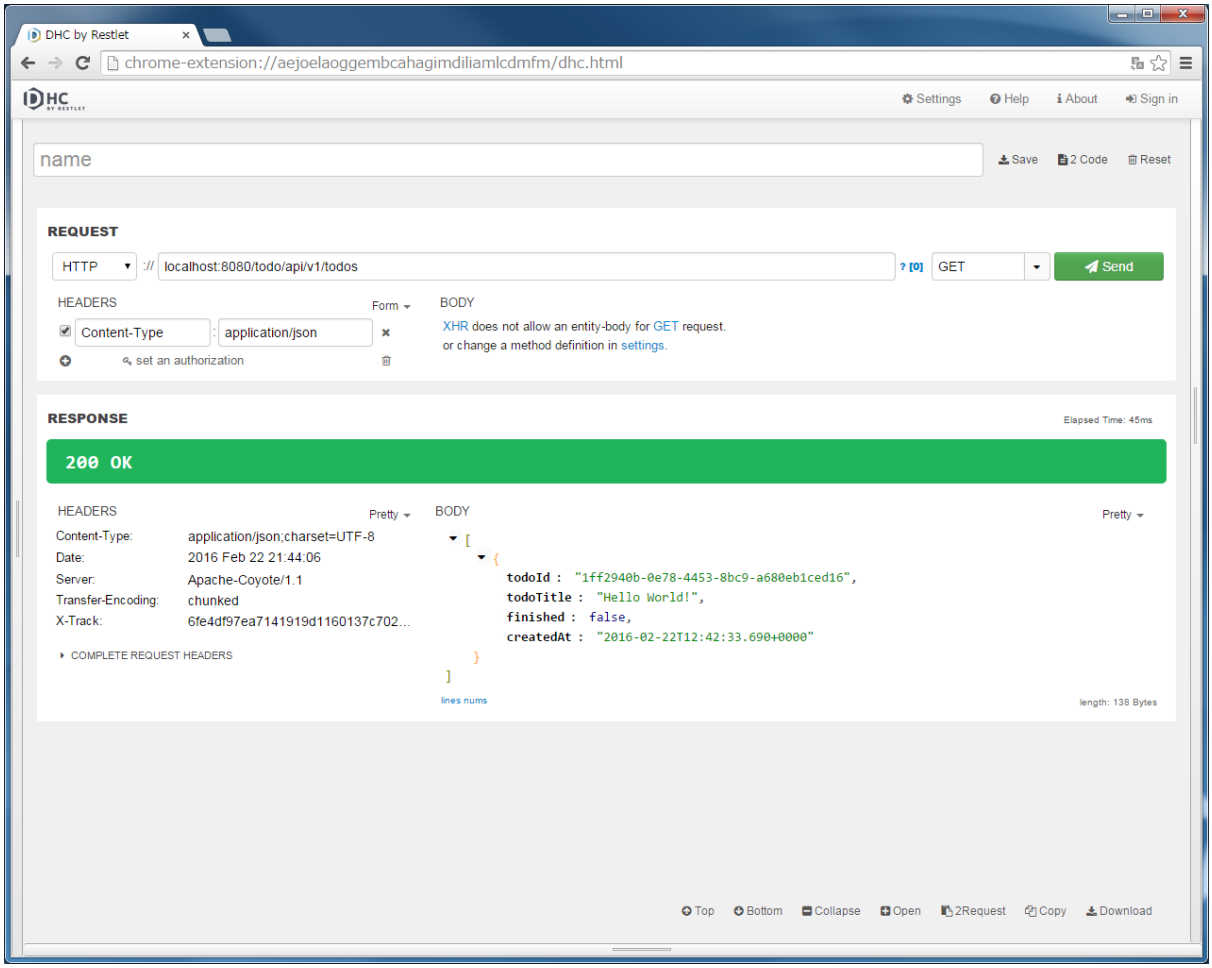
GET Todo の実装

チュートリアル *(Todo* アプリケーション*)* では、TodoService に一件取得用のメソッド (findOne) を作成しなかったため、TodoService と TodoServiceImpl に以下のハイライト部を修正・追加する。

findOne メソッドの定義を追加する。

src/main/java/com/example/todo/domain/service/todo/TodoService.java

```
package com.example.todo.domain.service.todo;

import java.util.Collection;

import com.example.todo.domain.model.Todo;

public interface TodoService {
    Collection<Todo> findAll();
```




```
    Todo findOne(String todoId);

    Todo create(Todo todo);

    Todo finish(String todoId);

    void delete(String todoId);
}
```

findOne メソッド呼び出し時に開始されるトランザクションを読み取り専用に設定し、アクセス修飾子を public に変更する。

src/main/java/com/example/todo/domain/service/todo/TodoServiceImpl.java

```
package com.example.todo.domain.service.todo;

import java.util.Collection;
import java.util.Date;
import java.util.UUID;

import javax.inject.Inject;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.message.ResultMessage;
import org.terasoluna.gfw.common.message.ResultMessages;

import com.example.todo.domain.model.Todo;
import com.example.todo.domain.repository.todo.TodoRepository;

@Service
@Transactional
public class TodoServiceImpl implements TodoService {

    private static final long MAX_UNFINISHED_COUNT = 5;

    @Inject
    TodoRepository todoRepository;

    @Override
    @Transactional(readOnly = true)
    public Todo findOne(String todoId) {
        Todo todo = todoRepository.findOne(todoId);
        if (todo == null) {
            ResultMessages messages = ResultMessages.error();
            messages.add(ResultMessage
                    .fromText("[E404] The requested Todo is not found. (id="
                            + todoId + ")"));
            throw new ResourceNotFoundException(messages);
        }
        return todo;
    }

    @Override
    @Transactional(readOnly = true)
    public Collection<Todo> findAll() {
        return todoRepository.findAll();
    }

    @Override
    public Todo create(Todo todo) {
        long unfinishedCount = todoRepository.countByFinished(false);
        if (unfinishedCount >= MAX_UNFINISHED_COUNT) {
            ResultMessages messages = ResultMessages.error();
            messages.add(ResultMessage
```

```
                    .fromText("[E001] The count of un-finished Todo must not be␣
↪over "
                            + MAX_UNFINISHED_COUNT + "."));
            throw new BusinessException(messages);
        }

        String todoId = UUID.randomUUID().toString();
        Date createdAt = new Date();

        todo.setTodoId(todoId);
        todo.setCreatedAt(createdAt);
        todo.setFinished(false);

        todoRepository.create(todo);
        /* REMOVE THIS LINE IF YOU USE JPA
            todoRepository.save(todo);
           REMOVE THIS LINE IF YOU USE JPA */

        return todo;
    }

    @Override
    public Todo finish(String todoId) {
        Todo todo = findOne(todoId);
        if (todo.isFinished()) {
            ResultMessages messages = ResultMessages.error();
            messages.add(ResultMessage
                    .fromText("[E002] The requested Todo is already finished. (id="
                            + todoId + ")"));
            throw new BusinessException(messages);
        }
        todo.setFinished(true);
        todoRepository.update(todo);
        /* REMOVE THIS LINE IF YOU USE JPA
            todoRepository.save(todo);
           REMOVE THIS LINE IF YOU USE JPA */
        return todo;
    }

    @Override
    public void delete(String todoId) {
        Todo todo = findOne(todoId);
        todoRepository.delete(todo);
    }
}
```

Todo リソースを一件取得する API(GET Todo) の処理を、`TodoRestController` の `getTodo` メソッドに

実装する。

src/main/java/com/example/todo/api/todo/TodoRestController.java

```
package com.example.todo.api.todo;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import javax.inject.Inject;

import org.dozer.Mapper;
import org.springframework.http.HttpStatus;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.todo.domain.model.Todo;
import com.example.todo.domain.service.todo.TodoService;

@RestController
@RequestMapping("todos")
public class TodoRestController {

    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    public List<TodoResource> getTodos() {
        Collection<Todo> todos = todoService.findAll();
        List<TodoResource> todoResources = new ArrayList<>();
        for (Todo todo : todos) {
            todoResources.add(beanMapper.map(todo, TodoResource.class));
        }
        return todoResources;
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public TodoResource postTodos(@RequestBody @Validated TodoResource␣
↪todoResource) {
        Todo createdTodo = todoService.create(beanMapper.map(todoResource, Todo.
↪class));
```

```
        TodoResource createdTodoResponse = beanMapper.map(createdTodo,␣
↪TodoResource.class);
        return createdTodoResponse;
    }

    @GetMapping("{todoId}") // (1)
    @ResponseStatus(HttpStatus.OK)
    public TodoResource getTodo(@PathVariable("todoId") String todoId) { // (2)
        Todo todo = todoService.findOne(todoId); // (3)
        TodoResource todoResource = beanMapper.map(todo, TodoResource.class);
        return todoResource;
    }

}
```

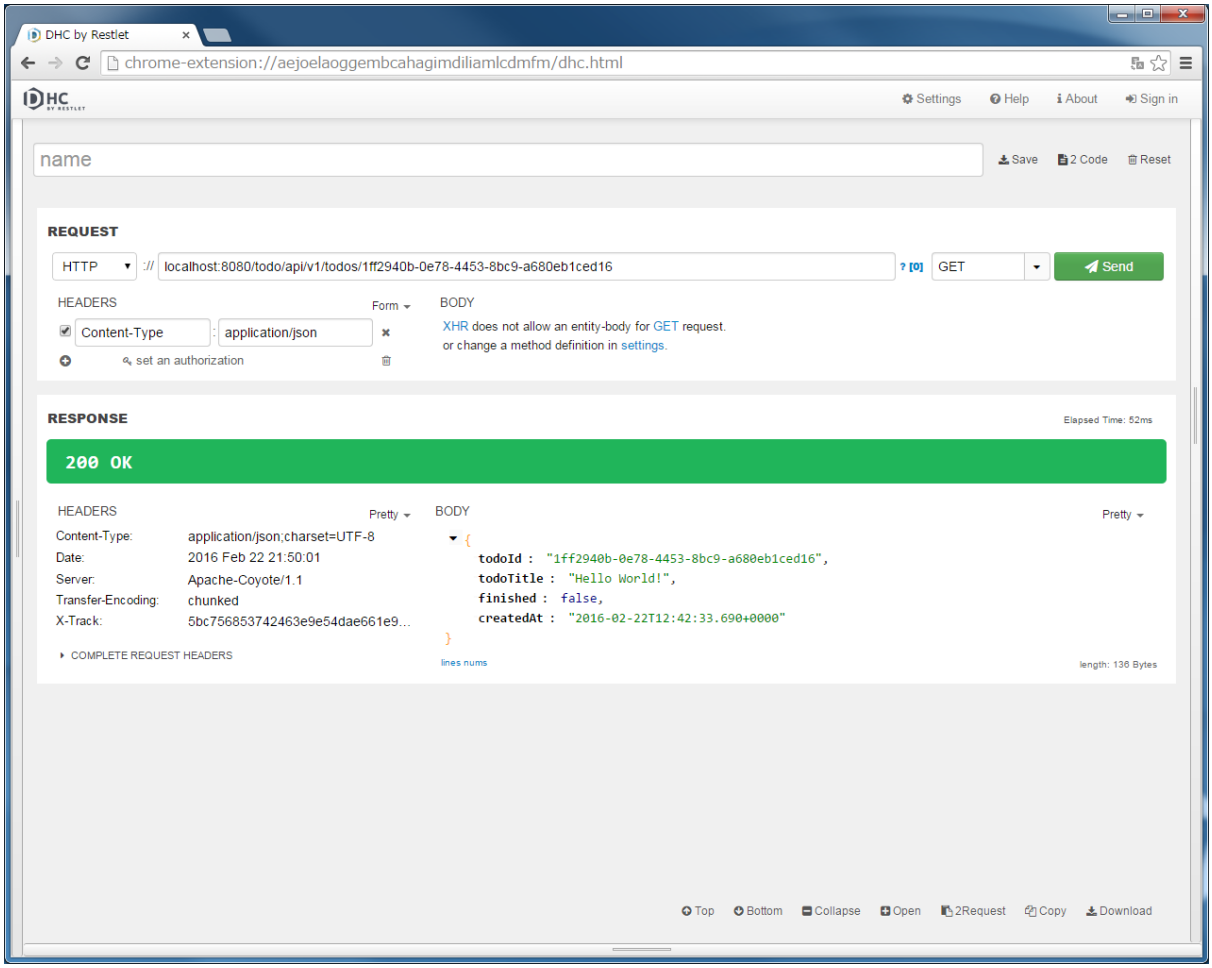
項番	説明
(1)	メソッドが GET のリクエストを処理するために、`@GetMapping` アノテーションを設定する。 パスから `todoId` を取得するために、`value` 属性にパス変数を指定する。
(2)	`@PathVariable` アノテーションの `value` 属性に、`todoId` を取得するためのパス変数名を指定する。
(3)	パス変数から取得した `todoId` を使用して、Todo リソースを一件取得する。

DHC を使用して、実装した API の動作確認を行う。

DHC を開いて URL に `localhost:8080/todo/api/v1/todos/{todoId}`を入力し、メソッドに GET を指定する。

`{todoId}`の部分は実際の ID を入れる必要があるので、POST Todos または GET Todos を実行して Response 中の `todoId` をコピーして貼り付けてから、」Send」ボタンをクリックする。

「200 OK」の HTTP ステータスが返却され、「RESPONSE」の「Body」に指定した Todo リソースの JSON が表示される。



PUT Todo の実装

Todo リソースを一件更新 (完了状態へ更新) する API(PUT Todo) の処理を、`TodoRestController` の `putTodo` メソッドに実装する。

`src/main/java/com/example/todo/api/todo/TodoRestController.java`

```
package com.example.todo.api.todo;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import javax.inject.Inject;

import org.dozer.Mapper;
import org.springframework.http.HttpStatus;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.todo.domain.model.Todo;
import com.example.todo.domain.service.todo.TodoService;

@RestController
@RequestMapping("todos")
public class TodoRestController {

    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    public List<TodoResource> getTodos() {
        Collection<Todo> todos = todoService.findAll();
        List<TodoResource> todoResources = new ArrayList<>();
        for (Todo todo : todos) {
            todoResources.add(beanMapper.map(todo, TodoResource.class));
        }
        return todoResources;
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public TodoResource postTodos(@RequestBody @Validated TodoResource␣
↪todoResource) {
        Todo createdTodo = todoService.create(beanMapper.map(todoResource, Todo.
↪class));
        TodoResource createdTodoResponse = beanMapper.map(createdTodo,␣
↪TodoResource.class);
        return createdTodoResponse;
    }

    @GetMapping("{todoId}")
    @ResponseStatus(HttpStatus.OK)
    public TodoResource getTodo(@PathVariable("todoId") String todoId) {
        Todo todo = todoService.findOne(todoId);
        TodoResource todoResource = beanMapper.map(todo, TodoResource.class);
        return todoResource;
    }

    @PutMapping("{todoId}") // (1)
    @ResponseStatus(HttpStatus.OK)
    public TodoResource putTodo(@PathVariable("todoId") String todoId) { // (2)
        Todo finishedTodo = todoService.finish(todoId); // (3)
```

```
        TodoResource finishedTodoResource = beanMapper.map(finishedTodo,␣
↪TodoResource.class);
        return finishedTodoResource;
    }

}
```

項番	説明
(1)	メソッドが PUT のリクエストを処理するために、`@PutMapping` アノテーションを設定する。 パスから `todoId` を取得するために、`value` 属性にパス変数を指定する。
(2)	`@PathVariable` アノテーションの `value` 属性に、`todoId` を取得するためのパス変数名を指定する。
(3)	パス変数から取得した `todoId` を使用して、Todo リソースを完了状態へ更新する。

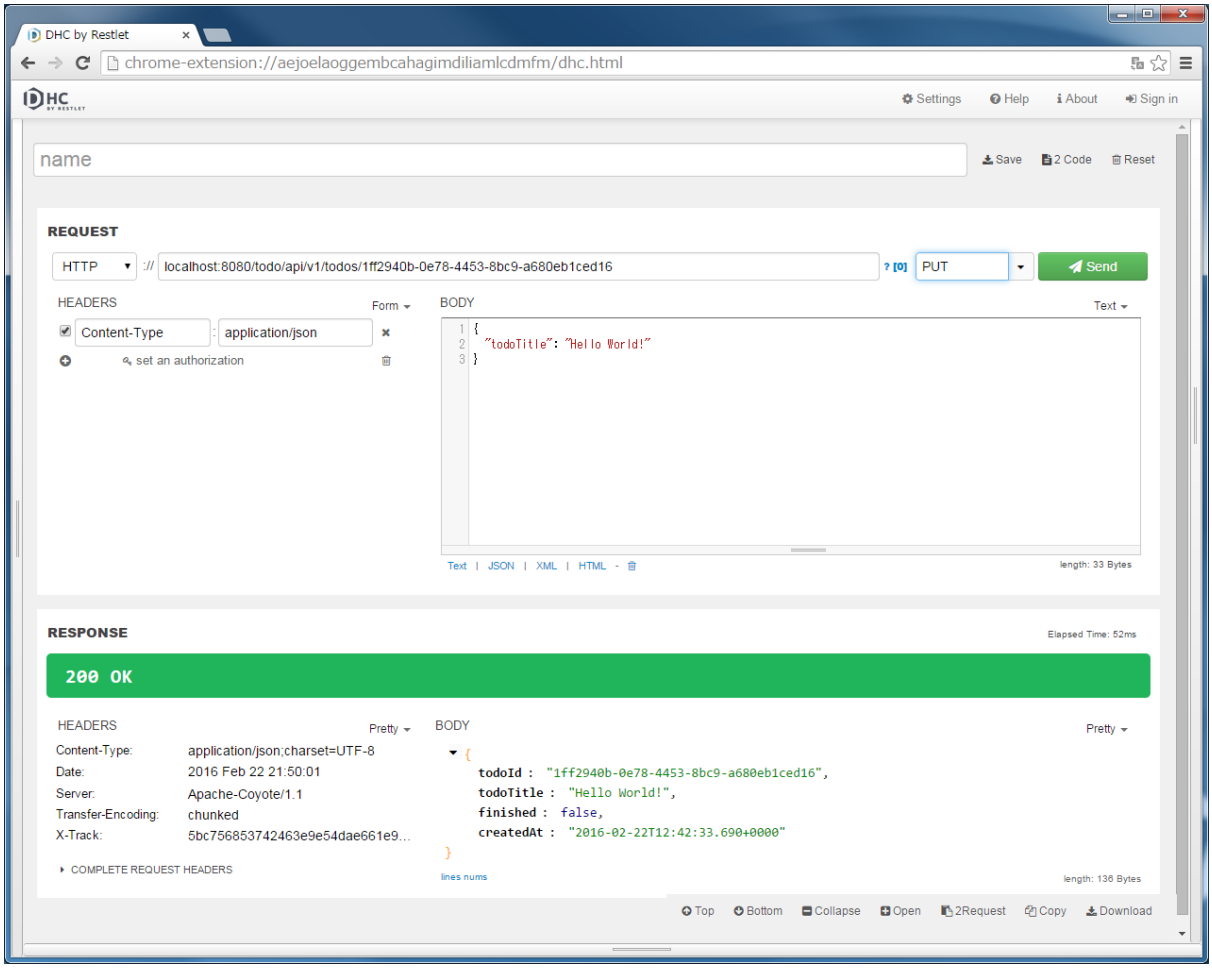
DHC を使用して、実装した API の動作確認を行う。

DHC を開いて URL に `localhost:8080/todo/api/v1/todos/{todoId}`を入力し、メソッドに PUT を指定する。

`{todoId}`の部分は実際の ID を入れる必要があるので、POST Todos または GET Todos を実行して Response 中の `todoId` をコピーして貼り付けてから、」Send」ボタンをクリックする。

「200 OK」の HTTP ステータスが返却され、「RESPONSE」の「Body」に更新された Todo リソースの JSON が表示される。

`finished` が `true` に更新されている。

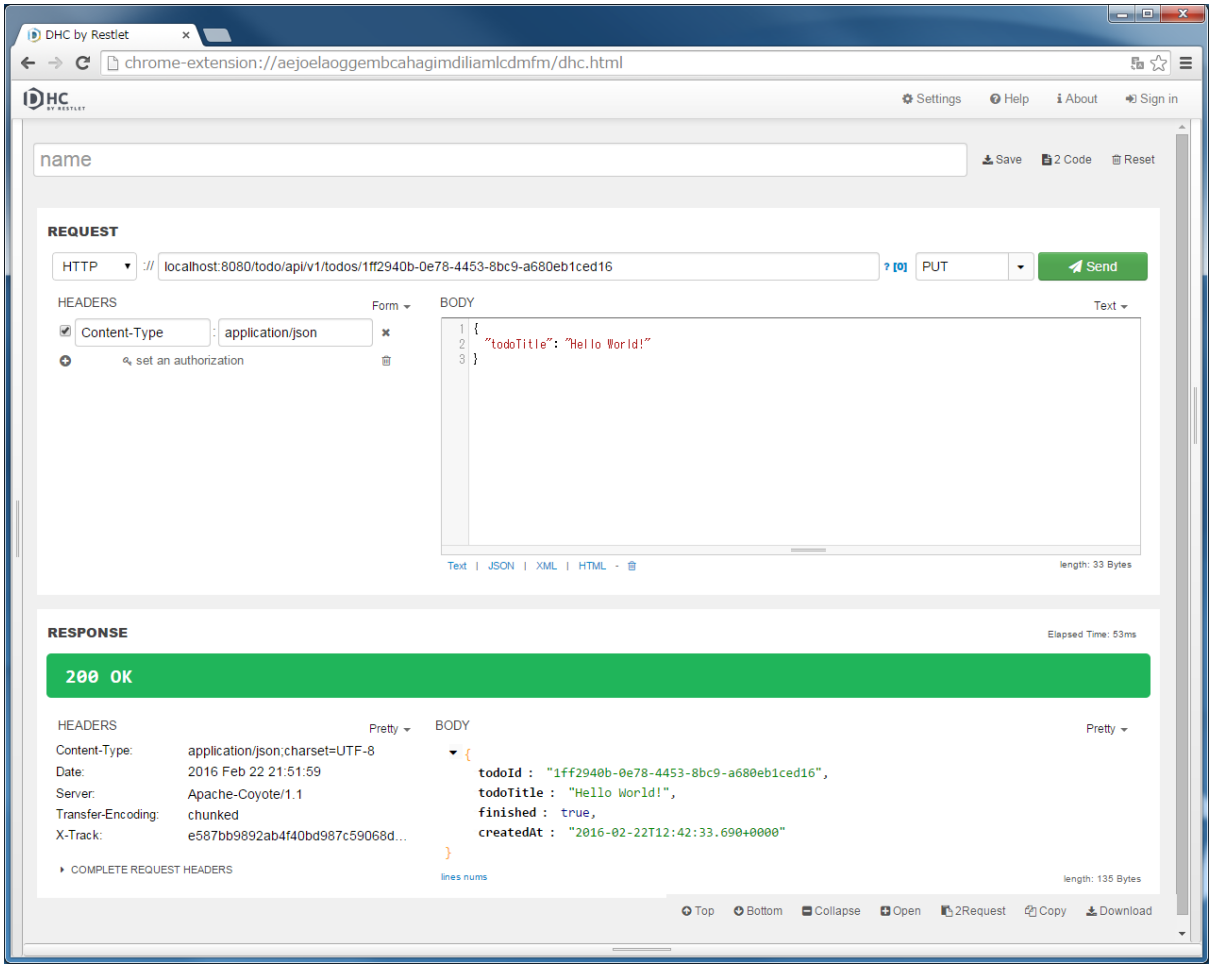


DELETE Todo の実装

最後に、Todo リソースを一件削除する API(DELETE Todo) の処理を、`TodoRestController` の `deleteTodo` メソッドに実装する。

`src/main/java/com/example/todo/api/todo/TodoRestController.java`

```
package com.example.todo.api.todo;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import javax.inject.Inject;

import org.dozer.Mapper;
import org.springframework.http.HttpStatus;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
```




```
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.todo.domain.model.Todo;
import com.example.todo.domain.service.todo.TodoService;

@RestController
@RequestMapping("todos")
public class TodoRestController {

    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    public List<TodoResource> getTodos() {
```

```
        Collection<Todo> todos = todoService.findAll();
        List<TodoResource> todoResources = new ArrayList<>();
        for (Todo todo : todos) {
            todoResources.add(beanMapper.map(todo, TodoResource.class));
        }
        return todoResources;
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public TodoResource postTodos(@RequestBody @Validated TodoResource␣
↪todoResource) {
        Todo createdTodo = todoService.create(beanMapper.map(todoResource, Todo.
↪class));
        TodoResource createdTodoResponse = beanMapper.map(createdTodo,␣
↪TodoResource.class);
        return createdTodoResponse;
    }

    @GetMapping("{todoId}")
    @ResponseStatus(HttpStatus.OK)
    public TodoResource getTodo(@PathVariable("todoId") String todoId) {
        Todo todo = todoService.findOne(todoId);
        TodoResource todoResource = beanMapper.map(todo, TodoResource.class);
        return todoResource;
    }

    @PutMapping("{todoId}")
    @ResponseStatus(HttpStatus.OK)
    public TodoResource putTodo(@PathVariable("todoId") String todoId) {
        Todo finishedTodo = todoService.finish(todoId);
        TodoResource finishedTodoResource = beanMapper.map(finishedTodo,␣
↪TodoResource.class);
        return finishedTodoResource;
    }

    @DeleteMapping("{todoId}") // (1)
    @ResponseStatus(HttpStatus.NO_CONTENT) // (2)
    public void deleteTodo(@PathVariable("todoId") String todoId) { // (3)
        todoService.delete(todoId); // (4)
    }

}
```

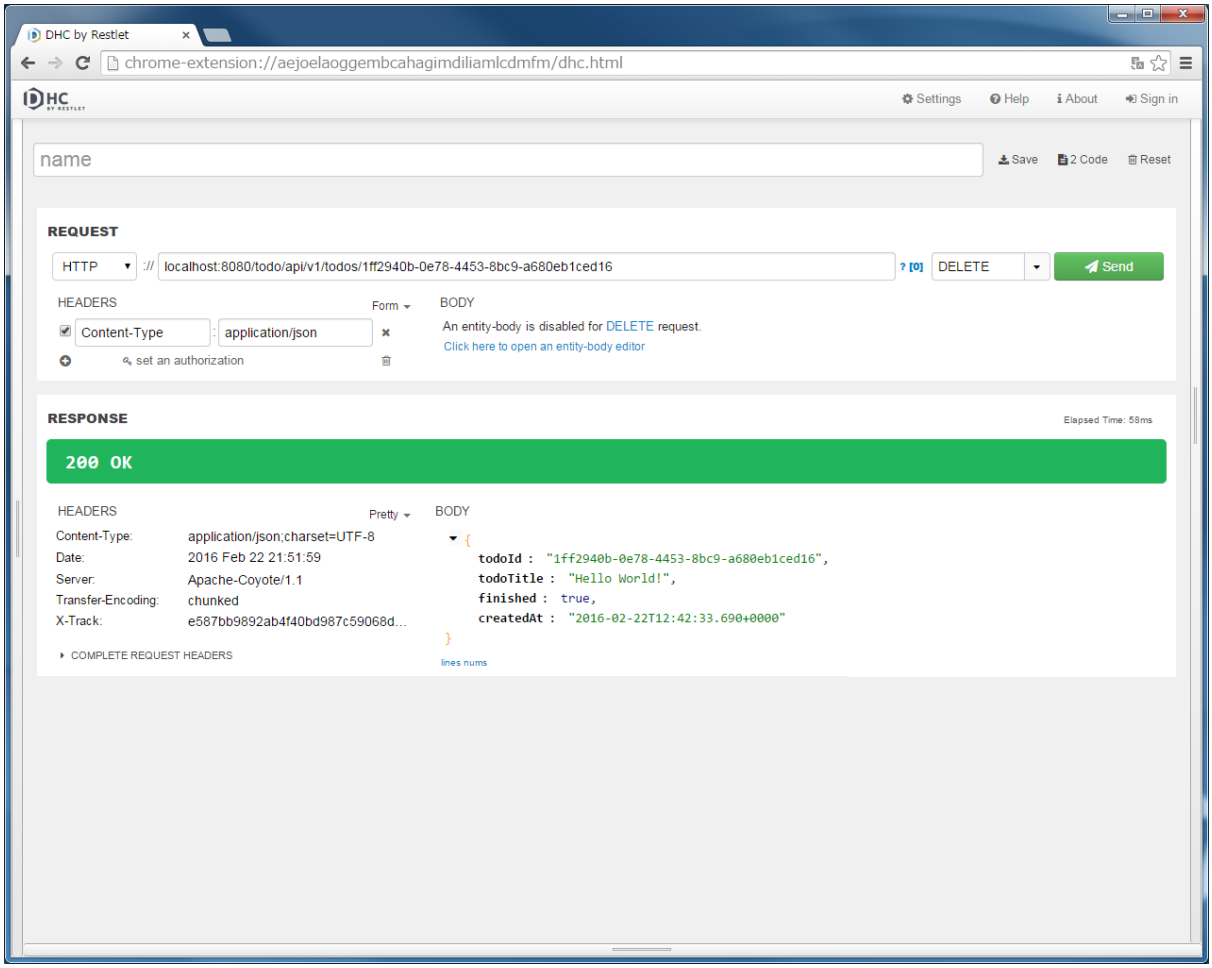
項番	説明
(1)	メソッドが DELETE のリクエストを処理するために、`@DeleteMapping` アノテーションを設定する。 パスから `todoId` を取得するために、`value` 属性にパス変数を指定する。
(2)	応答する HTTP ステータスコードを`@ResponseStatus` アノテーションに指定する。 HTTP ステータスとして、」204 No Content」を設定するため、`value` 属性には `HttpStatus.NO_CONTENT` を設定する。
(3)	DELETE の場合は返却するコンテンツがないため、返り値の型を `void` とする。
(4)	パス変数から取得した `todoId` を使用して、Todo リソースを削除する。

DHC を使用して、実装した API の動作確認を行う。

DHC を開いて URL に `localhost:8080/todo/api/v1/todos/{todoId}`を入力し、メソッドに DELETE を指定する。

`{todoId}`の部分は実際の ID を入れる必要があるので、POST Todos または GET Todos を実行して Response 中の `todoId` をコピーして貼り付けてから、」Send」ボタンをクリックする。

「204 No Content」の HTTP ステータスが返却され、「RESPONSE」の「Body」は空である。



DHC の URL に `localhost:8080/todo/api/v1/todos` を入力し、メソッドに GET を指定してから」Send」ボタンをクリックする。

Todo リソースが削除されている事が確認できる。

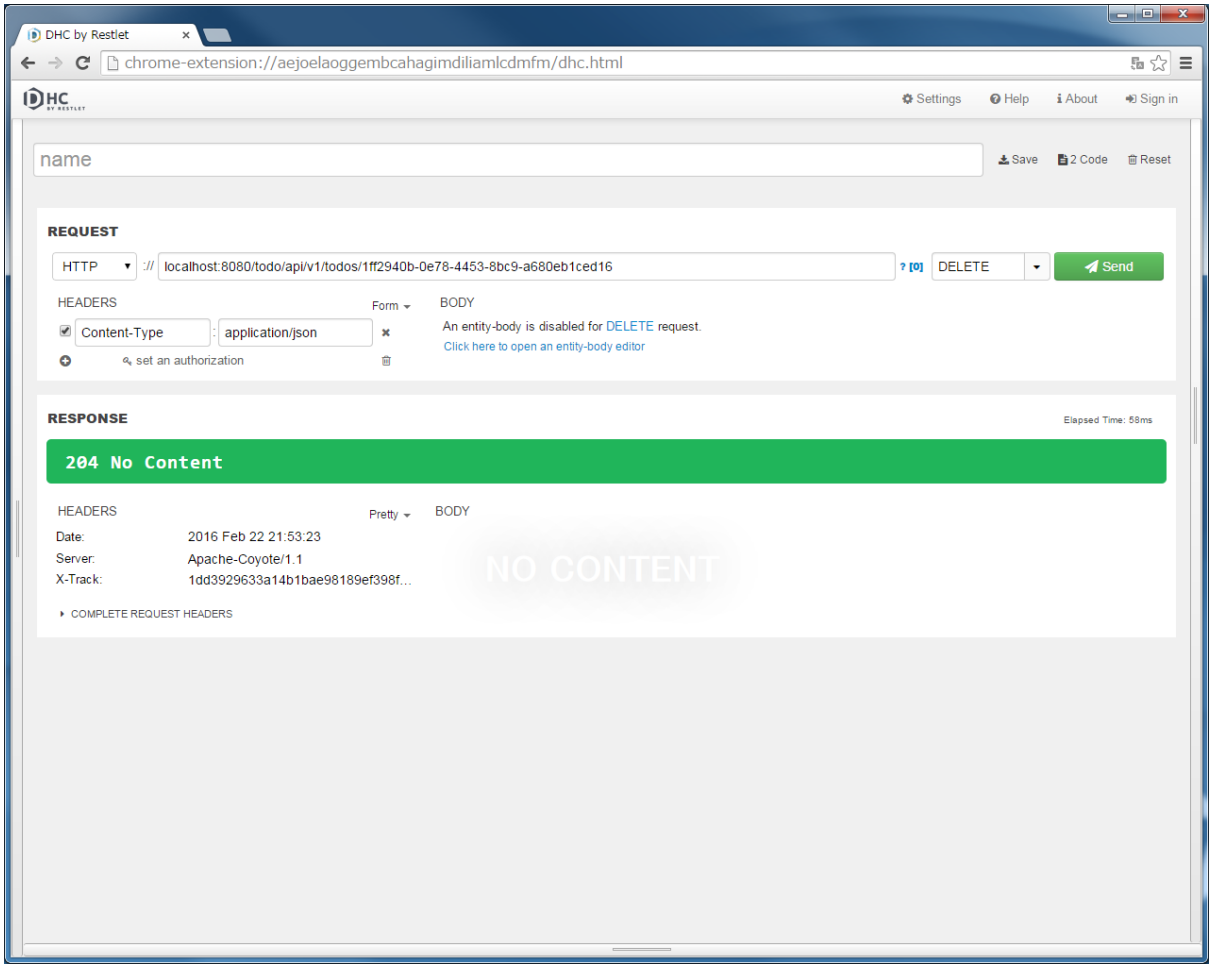
例外ハンドリングの実装

本チュートリアルでは、例外ハンドリングの実装方法をイメージしやすくするため、本ガイドラインで推奨している実装よりシンプルな実装にしてある。

実際の例外ハンドリングは、*RESTful Web Service* で説明されている方法でハンドリングを行うことを強く推奨する。

ドメイン層の実装を変更

本チュートリアルでは、エラーコードをキーにプロパティファイルからエラーメッセージを取得する。

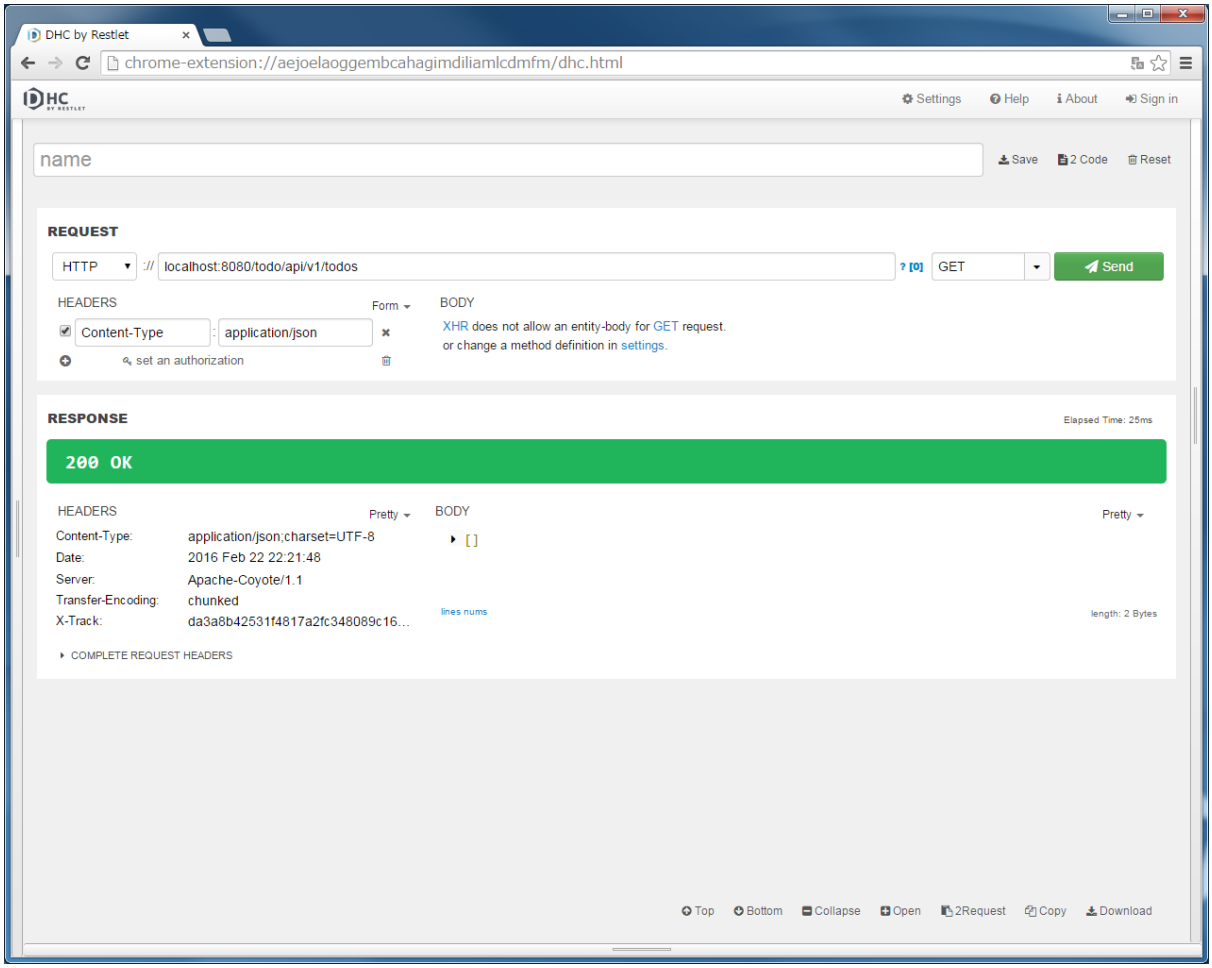


そのため、例外ハンドリングの実装を行う前に、チュートリアル *(Todo* アプリケーション*)* で作成した Service クラスの実装を以下のように変更する。

ハードコーディングされていたエラーメッセージの代わりに、エラーコードを指定するように変更する。

`src/main/java/com/example/todo/domain/service/todo/TodoServiceImpl.java`

```
package com.example.todo.domain.service.todo;

import java.util.Collection;
import java.util.Date;
import java.util.UUID;

import javax.inject.Inject;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.message.ResultMessages;
```




```
import com.example.todo.domain.model.Todo;
import com.example.todo.domain.repository.todo.TodoRepository;

@Service
@Transactional
public class TodoServiceImpl implements TodoService {

    private static final long MAX_UNFINISHED_COUNT = 5;

    @Inject
    TodoRepository todoRepository;

    @Override
    @Transactional(readOnly = true)
    public Todo findOne(String todoId) {
        Todo todo = todoRepository.findOne(todoId);
        if (todo == null) {
            ResultMessages messages = ResultMessages.error();
            messages.add("E404", todoId);
            throw new ResourceNotFoundException(messages);
        }
        return todo;
    }
```

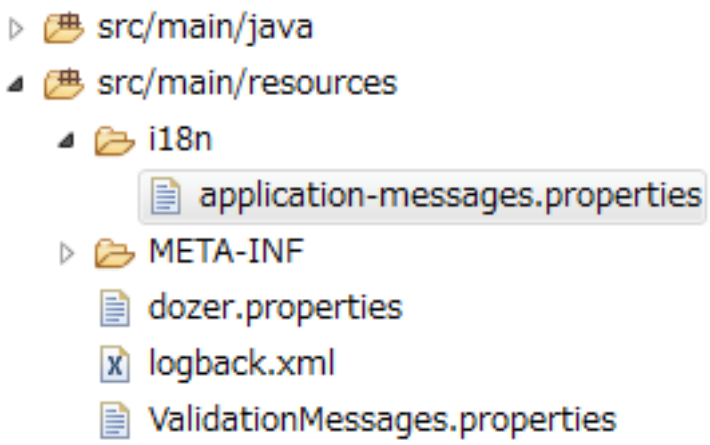
```
    @Override
    @Transactional(readOnly = true)
    public Collection<Todo> findAll() {
        return todoRepository.findAll();
    }

    @Override
    public Todo create(Todo todo) {
        long unfinishedCount = todoRepository.countByFinished(false);
        if (unfinishedCount >= MAX_UNFINISHED_COUNT) {
            ResultMessages messages = ResultMessages.error();
            messages.add("E001", MAX_UNFINISHED_COUNT);
            throw new BusinessException(messages);
        }

        String todoId = UUID.randomUUID().toString();
        Date createdAt = new Date();

        todo.setTodoId(todoId);
        todo.setCreatedAt(createdAt);
        todo.setFinished(false);

        todoRepository.create(todo);
        /* REMOVE THIS LINE IF YOU USE JPA
            todoRepository.save(todo);
           REMOVE THIS LINE IF YOU USE JPA */

        return todo;
    }

    @Override
    public Todo finish(String todoId) {
        Todo todo = findOne(todoId);
        if (todo.isFinished()) {
            ResultMessages messages = ResultMessages.error();
            messages.add("E002", todoId);
            throw new BusinessException(messages);
        }
        todo.setFinished(true);
        todoRepository.update(todo);
        /* REMOVE THIS LINE IF YOU USE JPA
            todoRepository.save(todo);
           REMOVE THIS LINE IF YOU USE JPA */
        return todo;
    }

    @Override
    public void delete(String todoId) {
        Todo todo = findOne(todoId);
        todoRepository.delete(todo);
    }
}
```

エラーメッセージの定義

本チュートリアルでは、エラーコードをキーにプロパティファイルからエラーメッセージを取得する。

そのため、例外ハンドリングの実装を行う前に、エラーコードに対応するエラーメッセージを、メッセージ用のプロパティファイルに定義する。

処理結果用のエラーコードに対応するエラーメッセージを、メッセージ用のプロパティファイルに定義する。



src/main/resources/i18n/application-messages.properties

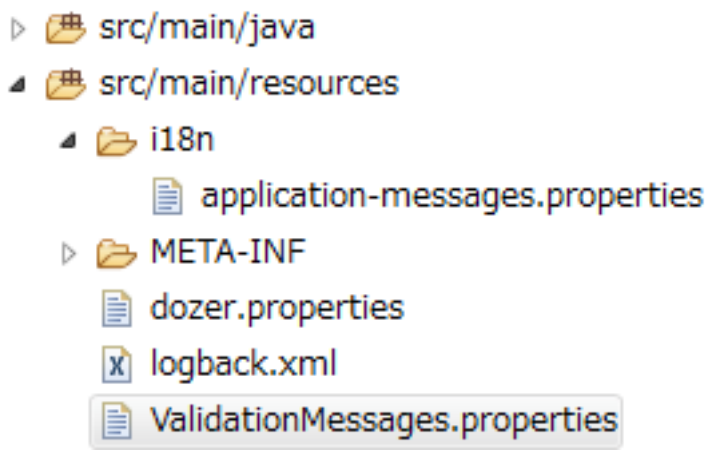
```
e.xx.fw.5001 = Resource not found.

e.xx.fw.7001 = Illegal screen flow detected!
e.xx.fw.7002 = CSRF attack detected!
e.xx.fw.7003 = Access Denied detected!
e.xx.fw.7004 = Missing CSRF detected!

e.xx.fw.8001 = Business error occurred!

e.xx.fw.9001 = System error occurred!
e.xx.fw.9002 = Data Access error!

# typemismatch
typeMismatch="{0}" is invalid.
typeMismatch.int="{0}" must be an integer.
typeMismatch.double="{0}" must be a double.
typeMismatch.float="{0}" must be a float.
typeMismatch.long="{0}" must be a long.
typeMismatch.short="{0}" must be a short.
typeMismatch.boolean="{0}" must be a boolean.
```

```
typeMismatch.java.lang.Integer="{0}" must be an integer.
typeMismatch.java.lang.Double="{0}" must be a double.
typeMismatch.java.lang.Float="{0}" must be a float.
typeMismatch.java.lang.Long="{0}" must be a long.
typeMismatch.java.lang.Short="{0}" must be a short.
typeMismatch.java.lang.Boolean="{0}" is not a boolean.
typeMismatch.java.util.Date="{0}" is not a date.
typeMismatch.java.lang.Enum="{0}" is not a valid value.

# For this tutorial
E001 = [E001] The count of un-finished Todo must not be over {0}.
E002 = [E002] The requested Todo is already finished. (id={0})
E400 = [E400] The requested Todo contains invalid values.
E404 = [E404] The requested Todo is not found. (id={0})
E500 = [E500] System error occurred.
E999 = [E999] Error occurred. Caused by : {0}
```

入力チェック用のエラーコードに対応するエラーメッセージを、Bean Validation のメッセージ用のプロパティファイルに定義する。

デフォルトのメッセージは、メッセージの中に項目名が含まれないため、デフォルトのメッセージ定義を変更する。

本チュートリアルでは、`TodoResource` クラスで使用しているルール (`@NotNull` と`@Size`) に対応するメッセージのみ定義する。



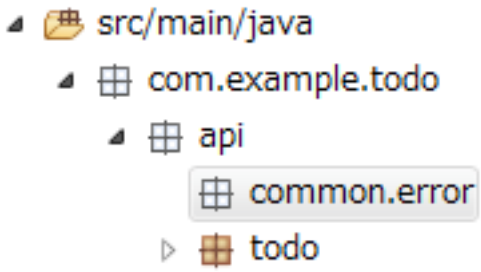
`src/main/resources/ValidationMessages.properties`

```
javax.validation.constraints.NotNull.message = {0} may not be null.
javax.validation.constraints.Size.message    = {0} size must be between {min} and
↪{max}.
```

エラーハンドリング用のクラスを格納するパッケージの作成

エラーハンドリング用のクラスを格納するためのパッケージを作成する。

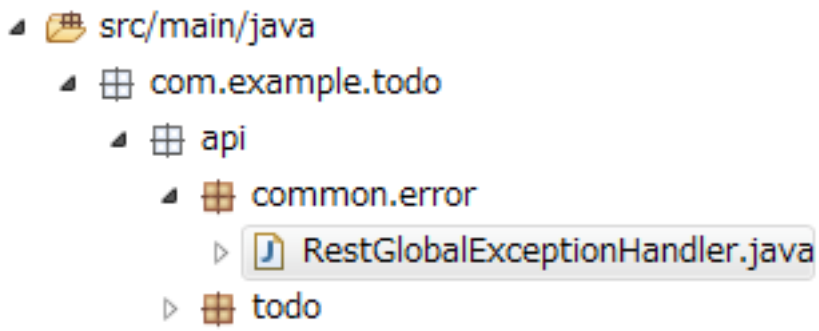
本チュートリアルでは、`com.example.todo.api.common.error` をエラーハンドリング用のクラスを格納するためのパッケージとする。



REST API のエラーハンドリングを行うクラスの作成

REST API のエラーハンドリングは、Spring MVC から提供されている `org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler` を継承したクラスを作成し、`@ControllerAdvice` アノテーションを付与する方法でハンドリングする。

以下に、`ResponseEntityExceptionHandler` を継承した `com.example.todo.api.common.error.RestGlobalExceptionHandler` クラスを作成する。



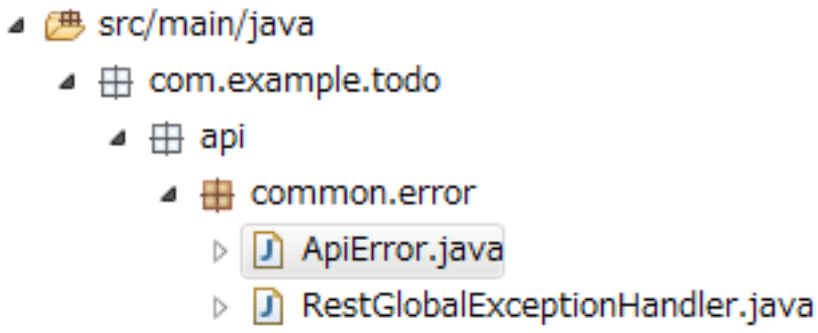
src/main/java/com/example/todo/api/common/error/RestGlobalExceptionHandler.java

```
package com.example.todo.api.common.error;

import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.servlet.mvc.method.annotation.
↪ResponseEntityExceptionHandler;

@ControllerAdvice
public class RestGlobalExceptionHandler extends ResponseEntityExceptionHandler {

}
```

REST API のエラー情報を保持する JavaBean の作成

REST API で発生したエラー情報を保持するクラスとして、ApiError クラスを com.example.todo.api.common.error パッケージに作成する。

ApiError クラスが JSON に変換されて、クライアントに応答される。



src/main/java/com/example/todo/api/common/error/ApiError.java

```
package com.example.todo.api.common.error;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import com.fasterxml.jackson.annotation.JsonInclude;

public class ApiError implements Serializable {

    private static final long serialVersionUID = 1L;
```

```
    private final String code;

    private final String message;

    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private final String target;

    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private final List<ApiError> details = new ArrayList<>();

    public ApiError(String code, String message) {
        this(code, message, null);
    }

    public ApiError(String code, String message, String target) {
        this.code = code;
        this.message = message;
        this.target = target;
    }

    public String getCode() {
        return code;
    }

    public String getMessage() {
        return message;
    }

    public String getTarget() {
        return target;
    }

    public List<ApiError> getDetails() {
        return details;
    }

    public void addDetail(ApiError detail) {
        details.add(detail);
    }

}
```

HTTP レスポンス BODY にエラー情報を出力するための実装

`ResponseEntityExceptionHandler` はデフォルトでは HTTP ステータス (400 や 500 など) の設定のみを行い、HTTP レスポンスの BODY は設定しない。そのため、`handleExceptionInternal` メソッドを以下のようにオーバーライドして、BODY を出力するように実装する。

src/main/java/com/example/todo/api/common/error/RestGlobalExceptionHandler.java

```
package com.example.todo.api.common.error;

import javax.inject.Inject;

import org.springframework.context.MessageSource;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.
↪ResponseEntityExceptionHandler;

@ControllerAdvice
public class RestGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @Inject
    MessageSource messageSource;

    @Override
    protected ResponseEntity<Object> handleExceptionInternal(Exception ex,
            Object body, HttpHeaders headers, HttpStatus status,
            WebRequest request) {
        Object responseBody = body;
        if (body == null) {
            responseBody = createApiError(request, "E999", ex.getMessage());
        }
        return ResponseEntity.status(status).headers(headers).body(responseBody);
    }

    private ApiError createApiError(WebRequest request, String errorCode,
            Object... args) {
        return new ApiError(errorCode, messageSource.getMessage(errorCode,
                args, request.getLocale()));
    }

}
```

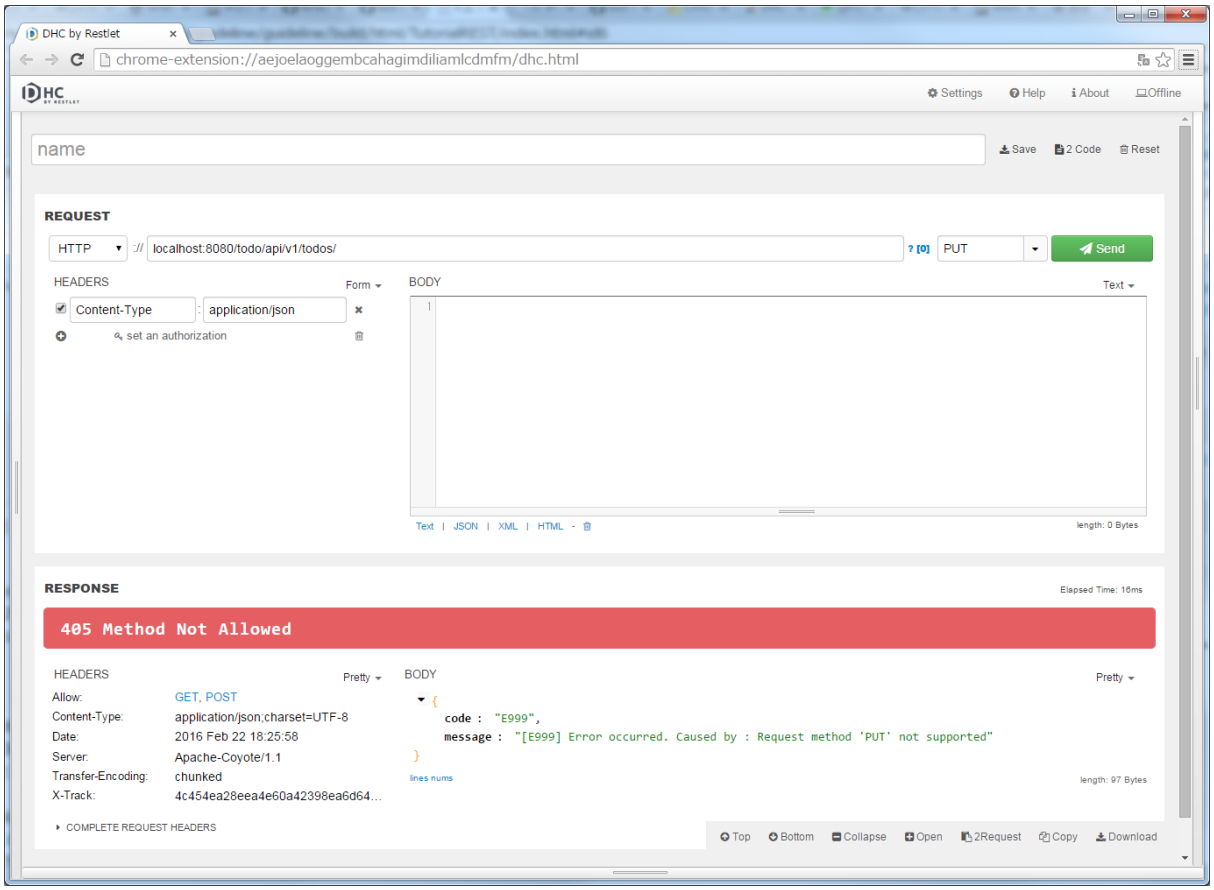
上記実装を行う事で、ResponseEntityExceptionHandler でハンドリングされる例外については、HTTP レスポンス BODY にエラー情報が出力される。

ResponseEntityExceptionHandler でハンドリングされる例外については、*DefaultHandlerExceptionResolver で設定される HTTP レスポンスコードについて*を参照されたい。

DHC を使用して、実装したエラーハンドリングの動作確認を行う。

DHC を開いて URL に `localhost:8080/todo/api/v1/todos` を入力し、メソッドに PUT を指定してから、」Send」ボタンをクリックする。

「405 Method Not Allowed」の HTTP ステータスが返却され、「RESPONSE」の「Body」には、エラー情報の JSON が表示される。



入力エラーのエラーハンドリングの実装

入力エラーの種類は、

- `org.springframework.web.bind.MethodArgumentNotValidException`
- `org.springframework.validation.BindException`
- `org.springframework.http.converter.HttpMessageNotReadableException`
- `org.springframework.beans.TypeMismatchException`

となる。

本チュートリアルでは、MethodArgumentNotValidException のエラーハンドリングの実装を行う。

MethodArgumentNotValidException は、HTTP リクエスト BODY に格納されているデータに入力エラーがあった場合に発生する例外である。

src/main/java/com/example/todo/api/common/error/RestGlobalExceptionHandler.java

```
package com.example.todo.api.common.error;

import javax.inject.Inject;

import org.springframework.context.MessageSource;
import org.springframework.context.support.DefaultMessageSourceResolvable;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.FieldError;
import org.springframework.validation.ObjectError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.
↪ResponseEntityExceptionHandler;

@ControllerAdvice
public class RestGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @Inject
    MessageSource messageSource;

    @Override
    protected ResponseEntity<Object> handleExceptionInternal(Exception ex,
            Object body, HttpHeaders headers, HttpStatus status,
            WebRequest request) {
        Object responseBody = body;
        if (body == null) {
            responseBody = createApiError(request, "E999", ex.getMessage());
        }
        return ResponseEntity.status(status).headers(headers).body(responseBody);
    }

    private ApiError createApiError(WebRequest request, String errorCode,
            Object... args) {
        return new ApiError(errorCode, messageSource.getMessage(errorCode,
                args, request.getLocale()));
    }

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
```

```
        ApiError apiError = createApiError(request, "E400");
        for (FieldError fieldError : ex.getBindingResult().getFieldErrors()) {
            apiError.addDetail(createApiError(request, fieldError, fieldError
                    .getField()));
        }
        for (ObjectError objectError : ex.getBindingResult().getGlobalErrors()) {
            apiError.addDetail(createApiError(request, objectError, objectError
                    .getObjectName()));
        }
        return handleExceptionInternal(ex, apiError, headers, status, request);
    }

    private ApiError createApiError(WebRequest request,
            DefaultMessageSourceResolvable messageSourceResolvable,
            String target) {
        return new ApiError(messageSourceResolvable.getCode(), messageSource
                .getMessage(messageSourceResolvable, request.getLocale()), target);
    }

}
```

DHC を使用して、実装したエラーハンドリングの動作確認を行う。

DHC を開いて URL に `localhost:8080/todo/api/v1/todos` を入力し、メソッドに POST を指定する。

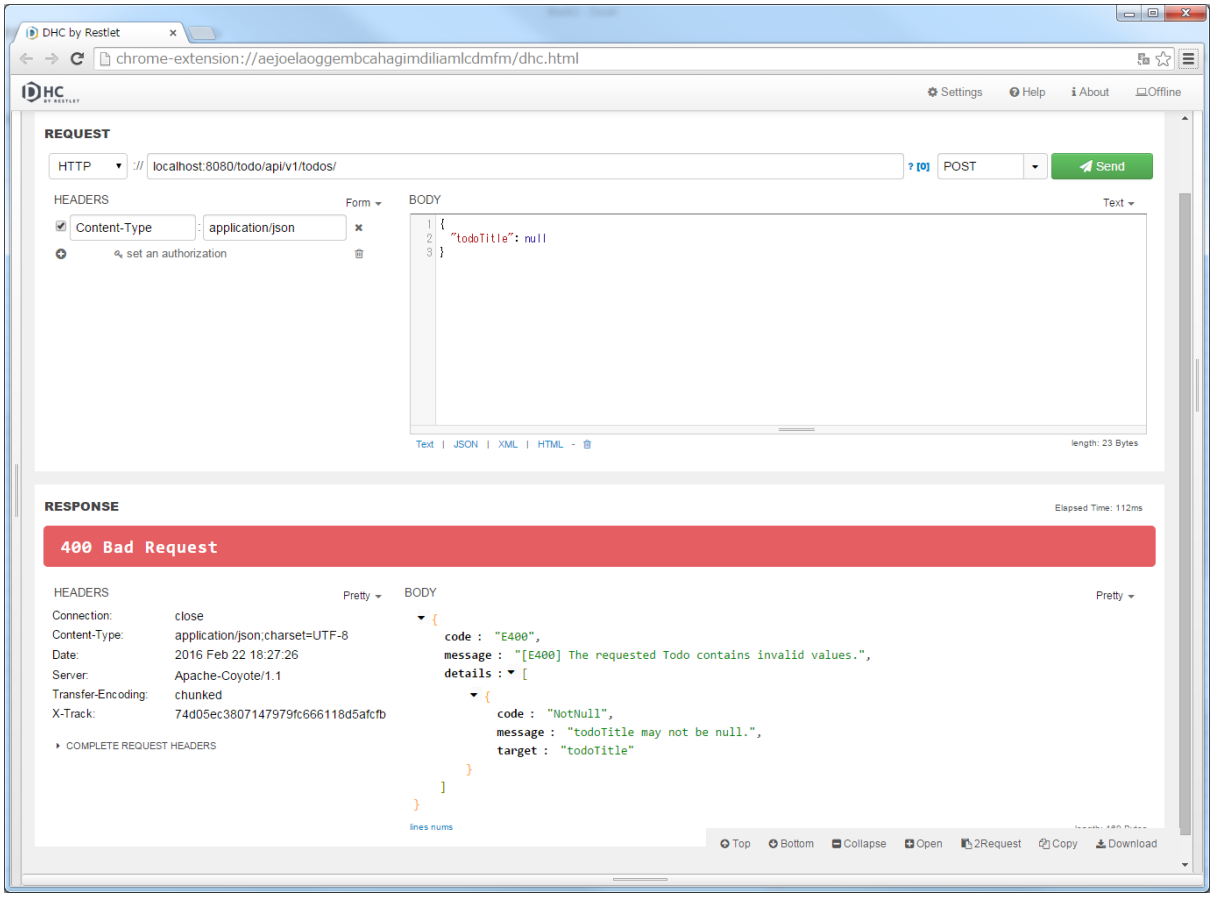
「REQUEST」の「BODY」に以下の JSON を入力する。

```
{
  "todoTitle": null
}
```

また、「REQUEST」の「HEADERS」の「+」ボタンで HTTP ヘッダーを追加し、「`Content-Type`」に「`application/json`」を設定後、”Send”ボタンをクリックする。

「400 Bad Request」の HTTP ステータスが返却され、「RESPONSE」の「Body」には、エラー情報の JSON が表示される。

`todoTitle` は必須項目なので、必須エラーが発生している。



業務例外のエラーハンドリングの実装

`RestGlobalExceptionHandler` に `org.terasoluna.gfw.common.exception.BusinessException` をハンドリングするメソッドを追加して、業務例外をハンドリングする。

業務例外が発生した場合は、」409 Conflict」の HTTP ステータスを設定する。

`src/main/java/com/example/todo/api/common/error/RestGlobalExceptionHandler.java`

```
package com.example.todo.api.common.error;

import javax.inject.Inject;

import org.springframework.context.MessageSource;
import org.springframework.context.support.DefaultMessageSourceResolvable;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.FieldError;
import org.springframework.validation.ObjectError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.
↪ResponseEntityExceptionHandler;
```

```
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ResultMessagesNotificationException;
import org.terasoluna.gfw.common.message.ResultMessage;

@ControllerAdvice
public class RestGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @Inject
    MessageSource messageSource;

    @Override
    protected ResponseEntity<Object> handleExceptionInternal(Exception ex,
            Object body, HttpHeaders headers, HttpStatus status,
            WebRequest request) {
        Object responseBody = body;
        if (body == null) {
            responseBody = createApiError(request, "E999", ex.getMessage());
        }
        return ResponseEntity.status(status).headers(headers).body(responseBody);
    }

    private ApiError createApiError(WebRequest request, String errorCode,
            Object... args) {
        return new ApiError(errorCode, messageSource.getMessage(errorCode,
                args, request.getLocale()));
    }

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        ApiError apiError = createApiError(request, "E400");
        for (FieldError fieldError : ex.getBindingResult().getFieldErrors()) {
            apiError.addDetail(createApiError(request, fieldError, fieldError
                    .getField()));
        }
        for (ObjectError objectError : ex.getBindingResult().getGlobalErrors()) {
            apiError.addDetail(createApiError(request, objectError, objectError
                    .getObjectName()));
        }
        return handleExceptionInternal(ex, apiError, headers, status, request);
    }

    private ApiError createApiError(WebRequest request,
            DefaultMessageSourceResolvable messageSourceResolvable,
            String target) {
        return new ApiError(messageSourceResolvable.getCode(), messageSource
                .getMessage(messageSourceResolvable, request.getLocale()), target);
    }

    @ExceptionHandler(BusinessException.class)
    public ResponseEntity<Object> handleBusinessException(BusinessException ex,
            WebRequest request) {
```

```
        return handleResultMessagesNotificationException(ex, new HttpHeaders(),
                HttpStatus.CONFLICT, request);
    }

    private ResponseEntity<Object> handleResultMessagesNotificationException(
            ResultMessagesNotificationException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        ResultMessage message = ex.getResultMessages().iterator().next();
        ApiError apiError = createApiError(request, message.getCode(), message
                .getArgs());
        return handleExceptionInternal(ex, apiError, headers, status, request);
    }

}
```

DHC を使用して、実装したエラーハンドリングの動作確認を行う。

DHC を開いて URL に `localhost:8080/todo/api/v1/todos/{todoId}`を入力し、メソッドに PUT を指定する。

{todoId}の部分は実際の ID を入れる必要があるので、POST Todos または GET Todos を実行して Response 中の `todoId` をコピーして貼り付けてから、”Send”ボタンを 2 回クリックする。

未完了状態の Todo の `todoId` を指定すること。

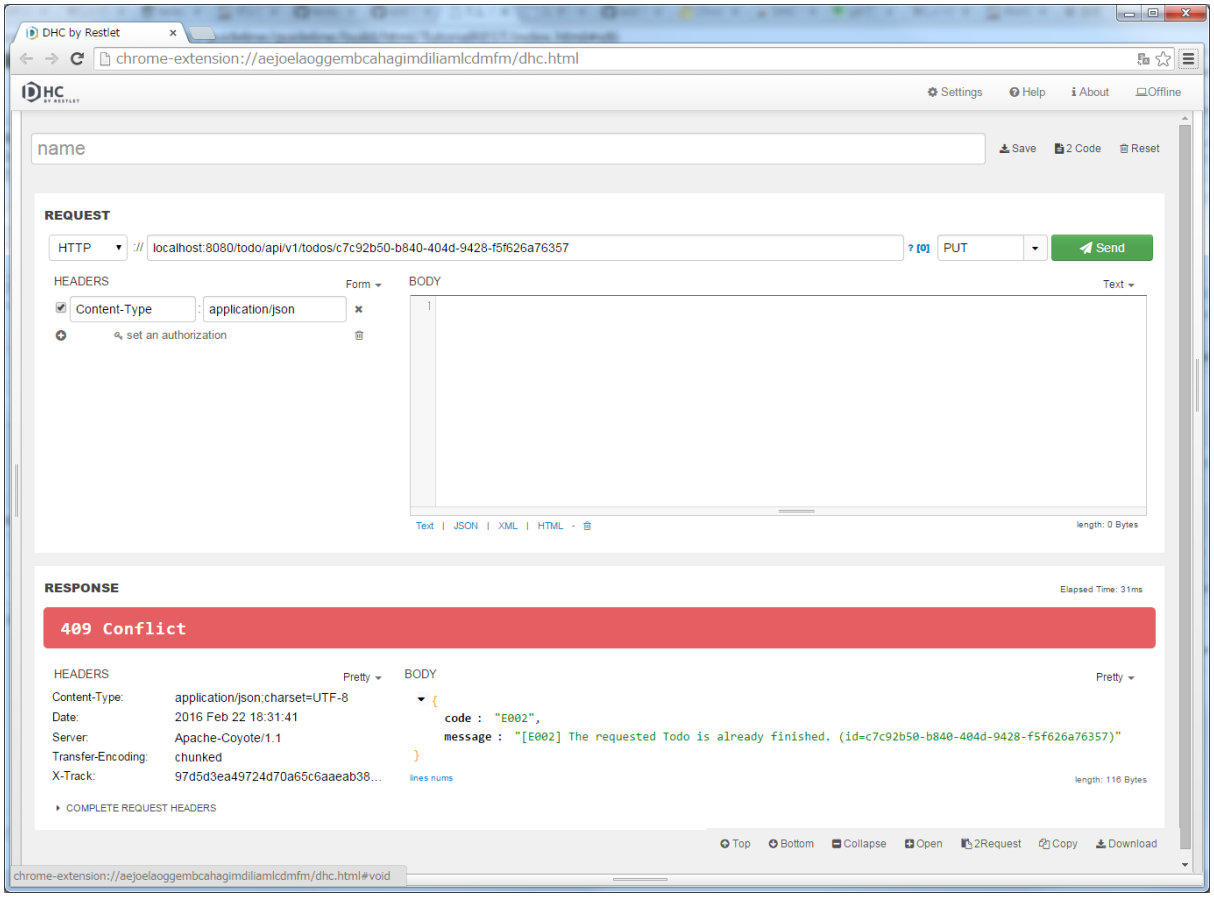
2 回目のリクエストに対するレスポンスとして、」409 Conflict」の HTTP ステータスが返却され、「RESPONSE」の「Body」には、エラー情報の JSON が表示される。

リソース未検出例外のエラーハンドリングの実装

`RestGlobalExceptionHandler` に `org.terasoluna.gfw.common.exception.ResourceNotFoundException` をハンドリングするメソッドを追加して、リソース未検出例外をハンドリングする。

リソース未検出例外が発生した場合、」404 NotFound」の HTTP ステータスを設定する。

`src/main/java/com/example/todo/api/common/error/RestGlobalExceptionHandler.java`




```
package com.example.todo.api.common.error;

import javax.inject.Inject;

import org.springframework.context.MessageSource;
import org.springframework.context.support.DefaultMessageSourceResolvable;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.FieldError;
import org.springframework.validation.ObjectError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.
 ↪ResponseEntityExceptionHandler;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.exception.ResultMessagesNotificationException;
import org.terasoluna.gfw.common.message.ResultMessage;

@ControllerAdvice
public class RestGlobalExceptionHandler extends ResponseEntityExceptionHandler {
```

```
    @Inject
    MessageSource messageSource;

    @Override
    protected ResponseEntity<Object> handleExceptionInternal(Exception ex,
            Object body, HttpHeaders headers, HttpStatus status,
            WebRequest request) {
        Object responseBody = body;
        if (body == null) {
            responseBody = createApiError(request, "E999", ex.getMessage());
        }
        return ResponseEntity.status(status).headers(headers).body(responseBody);
    }

    private ApiError createApiError(WebRequest request, String errorCode,
            Object... args) {
        return new ApiError(errorCode, messageSource.getMessage(errorCode,
                args, request.getLocale()));
    }

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        ApiError apiError = createApiError(request, "E400");
        for (FieldError fieldError : ex.getBindingResult().getFieldErrors()) {
            apiError.addDetail(createApiError(request, fieldError, fieldError
                    .getField()));
        }
        for (ObjectError objectError : ex.getBindingResult().getGlobalErrors()) {
            apiError.addDetail(createApiError(request, objectError, objectError
                    .getObjectName()));
        }
        return handleExceptionInternal(ex, apiError, headers, status, request);
    }

    private ApiError createApiError(WebRequest request,
            DefaultMessageSourceResolvable messageSourceResolvable,
            String target) {
        return new ApiError(messageSourceResolvable.getCode(), messageSource
                .getMessage(messageSourceResolvable, request.getLocale()), target);
    }

    @ExceptionHandler(BusinessException.class)
    public ResponseEntity<Object> handleBusinessException(BusinessException ex,
            WebRequest request) {
        return handleResultMessagesNotificationException(ex, new HttpHeaders(),
                HttpStatus.CONFLICT, request);
    }

    private ResponseEntity<Object> handleResultMessagesNotificationException(
            ResultMessagesNotificationException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
```

```
        ResultMessage message = ex.getResultMessages().iterator().next();
        ApiError apiError = createApiError(request, message.getCode(), message
                .getArgs());
        return handleExceptionInternal(ex, apiError, headers, status, request);
    }

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Object> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {
        return handleResultMessagesNotificationException(ex, new HttpHeaders(),
                HttpStatus.NOT_FOUND, request);
    }

}
```

DHC を使用して、実装したエラーハンドリングの動作確認を行う。

DHC を開いて URL に `localhost:8080/todo/api/v1/todos/{todoId}`を入力し、メソッドに GET を指定する。

{todoId}の部分には存在しない ID を指定して、”Send”ボタンをクリックする。

「404 Not Found」の HTTP ステータスが返却され、「RESPONSE」の「Body」には、エラー情報の JSON が表示される。

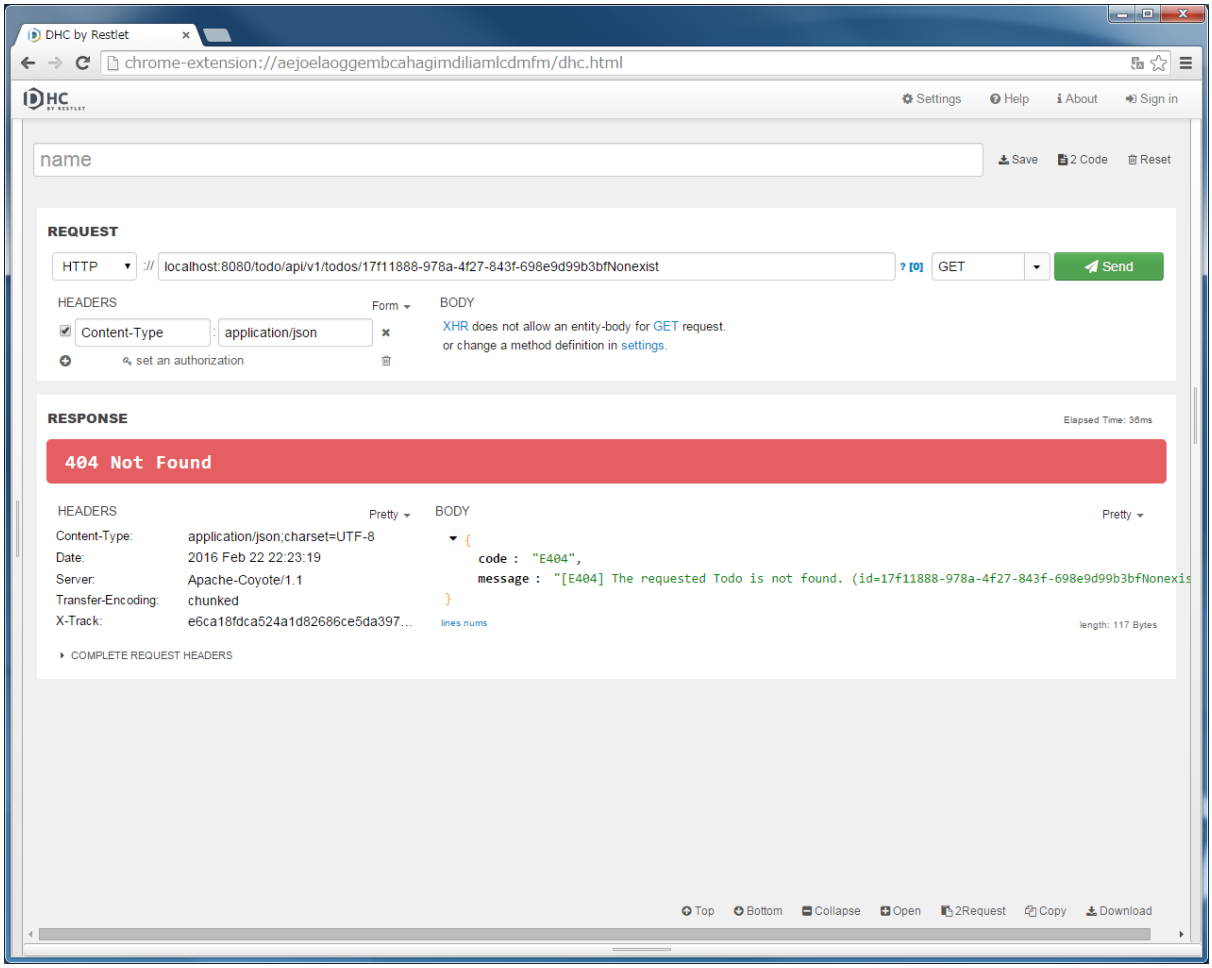
システム例外のエラーハンドリングの実装

最後に、`RestGlobalExceptionHandler` に `java.lang.Exception` をハンドリングするメソッドを追加して、システム例外をハンドリングする。

システム例外が発生した場合、」500 InternalServerError」の HTTP ステータスを設定する。

`src/main/java/com/example/todo/api/common/error/RestGlobalExceptionHandler.java`

```
package com.example.todo.api.common.error;

import javax.inject.Inject;

import org.springframework.context.MessageSource;
```




```
import org.springframework.context.support.DefaultMessageSourceResolvable;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.FieldError;
import org.springframework.validation.ObjectError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.
↪ResponseEntityExceptionHandler;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.exception.ResultMessagesNotificationException;
import org.terasoluna.gfw.common.message.ResultMessage;

@ControllerAdvice
public class RestGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @Inject
    MessageSource messageSource;
```

```
    @Override
    protected ResponseEntity<Object> handleExceptionInternal(Exception ex,
            Object body, HttpHeaders headers, HttpStatus status,
            WebRequest request) {
        Object responseBody = body;
        if (body == null) {
            responseBody = createApiError(request, "E999", ex.getMessage());
        }
        return ResponseEntity.status(status).headers(headers).body(responseBody);
    }

    private ApiError createApiError(WebRequest request, String errorCode,
            Object... args) {
        return new ApiError(errorCode, messageSource.getMessage(errorCode,
                args, request.getLocale()));
    }

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        ApiError apiError = createApiError(request, "E400");
        for (FieldError fieldError : ex.getBindingResult().getFieldErrors()) {
            apiError.addDetail(createApiError(request, fieldError, fieldError
                    .getField()));
        }
        for (ObjectError objectError : ex.getBindingResult().getGlobalErrors()) {
            apiError.addDetail(createApiError(request, objectError, objectError
                    .getObjectName()));
        }
        return handleExceptionInternal(ex, apiError, headers, status, request);
    }

    private ApiError createApiError(WebRequest request,
            DefaultMessageSourceResolvable messageSourceResolvable,
            String target) {
        return new ApiError(messageSourceResolvable.getCode(), messageSource
                .getMessage(messageSourceResolvable, request.getLocale()), target);
    }

    @ExceptionHandler(BusinessException.class)
    public ResponseEntity<Object> handleBusinessException(BusinessException ex,
            WebRequest request) {
        return handleResultMessagesNotificationException(ex, new HttpHeaders(),
                HttpStatus.CONFLICT, request);
    }

    private ResponseEntity<Object> handleResultMessagesNotificationException(
            ResultMessagesNotificationException ex, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        ResultMessage message = ex.getResultMessages().iterator().next();
        ApiError apiError = createApiError(request, message.getCode(), message
                .getArgs());
```

```
        return handleExceptionInternal(ex, apiError, headers, status, request);
    }

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Object> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {
        return handleResultMessagesNotificationException(ex, new HttpHeaders(),
                HttpStatus.NOT_FOUND, request);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<Object> handleSystemError(Exception ex,
            WebRequest request) {
        ApiError apiError = createApiError(request, "E500");
        return handleExceptionInternal(ex, apiError, new HttpHeaders(),
                HttpStatus.INTERNAL_SERVER_ERROR, request);
    }

}
```

DHC を使用して、実装したエラーハンドリングの動作確認を行う。

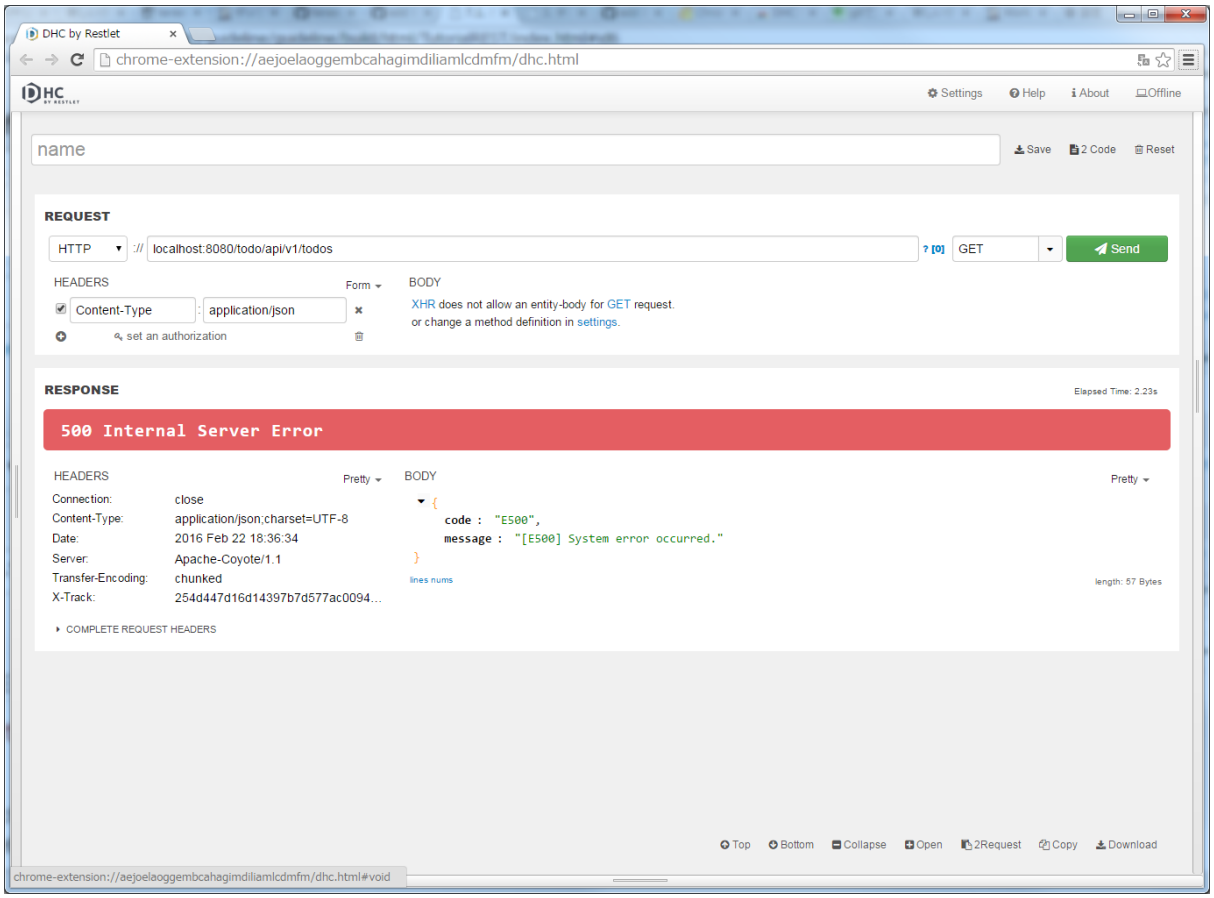
システムエラーを発生させるために、テーブルを未作成の状態でアプリケーションを起動させる。

src/main/resources/META-INF/spring/todo-infra.properties

```
database=H2
#database.url=jdbc:h2:mem:todo;DB_CLOSE_DELAY=-1;INIT=create table if not exists␣
↪todo(todo_id varchar(36) primary key, todo_title varchar(30), finished boolean,␣
↪created_at timestamp)
database.url=jdbc:h2:mem:todo;DB_CLOSE_DELAY=-1
database.username=sa
database.password=
database.driverClassName=org.h2.Driver
# connection pool
cp.maxActive=96
cp.maxIdle=16
cp.minIdle=0
cp.maxWait=60000
```

DHC を開いて URL に `localhost:8080/todo/api/v1/todos` を入力し、メソッドに GET を指定して、”Send”ボタンをクリックする。

「500 Internal Server Error」の HTTP ステータスが返却され、「RESPONSE」の「Body」には、エラー情報の JSON が表示される。



注釈: システムエラーが発生した場合、クライアントへ返却するメッセージは、エラー原因が特定されないシンプルなエラーメッセージを設定することを推奨する。エラー原因が特定できるメッセージを設定してしまうと、システムの脆弱性をクライアントに公開する可能性があり、セキュリティー上問題がある。

エラー原因は、エラー解析用にログに出力すればよい。Blank プロジェクトのデフォルトの設定では、共通ライブラリから提供している `ExceptionLogger` によってログが出力されるようなっているため、ログを出力するための設定や実装は不要である。

`ExceptionLogger` によって出力されるログは以下の通りである。Todo テーブルが存在しない事が原因でシステムエラーが発生している事がわかる。

```
date:2015-01-19 02:08:47        thread:tomcat-http--4   X-
↪Track:aadf5822205d423c95a6531f2f76036f        level:ERROR     logger:o.t.
↪gfw.common.exception.ExceptionLogger         message:[e.xx.fw.9002]
### Error querying database.  Cause: org.h2.jdbc.JdbcSQLException: Table "TODO
↪" not found; SQL statement:
SELECT
            todo_id,
```

```
            todo_title,
            finished,
            created_at
        FROM
            todo [42102-182]
### The error may exist in com/example/todo/domain/repository/todo/
↪TodoRepository.xml
### The error may involve com.example.todo.domain.repository.todo.
↪TodoRepository.findAll
### The error occurred while executing a query

... (omitted)
```

11.2.4 おわりに

このチュートリアルでは、以下の内容を学習した。

- TERASOLUNA Server Framework for Java (5.x) による基本的な RESTful Web サービスの構築方法
- REST API(GET, POST, PUT, DELETE) を提供する Controller クラスの実装
- JavaBean と JSON の相互変換方法
- エラーメッセージの定義方法
- Spring MVC を使用した各種例外のハンドリング方法

ここでは、基本的な RESTful Web サービスの実装法について示した。考え方の元となるアーキテクチャ・設計指針等について理解を深める為には、「*RESTful Web Service*」 を参照されたい。

11.3 セッションチュートリアル

11.3.1 始めに

学習の流れ

このチュートリアルでは、簡易 web アプリケーションの作成を通じてセッション管理対象となるデータの設計方法やセッションを利用するための具体的な実装方法を学習する。本チュートリアルは以下の流れで実施する。

1. 作成する web アプリケーションの要件を確認する
2. 要件を満たすような Controller の実装方法とデータの設計を行う手順を確認する
3. 設計情報をもとに実装する

このチュートリアルで学ぶこと

- セッション管理対象となるデータの設計方法
  - セッションに格納するデータの選択
  - セッション中のデータの破棄
- 本 FW におけるセッションの具体的な利用方法
  - `@SessionAttributes` を使用する方法
  - セッションスコープの Bean を使用する方法

対象読者

- チュートリアル：Todo アプリケーションを実施している
- チュートリアル：Spring Security を実施している

検証環境

本チュートリアルは以下の環境で動作確認している。

種別	プロダクト
OS	Windows 7
JVM	Java 1.8
IDE	Spring Tool Suite 3.6.4.RELEASE (以降「STS」と呼ぶ)
Build Tool	Apache Maven 3.3.3 (以降「Maven」と呼ぶ)
Application Server	Pivotal tc Server Developer Edition v3.1 (STS に同封)
Web Browser	Google Chrome 42.0.2311.90 m

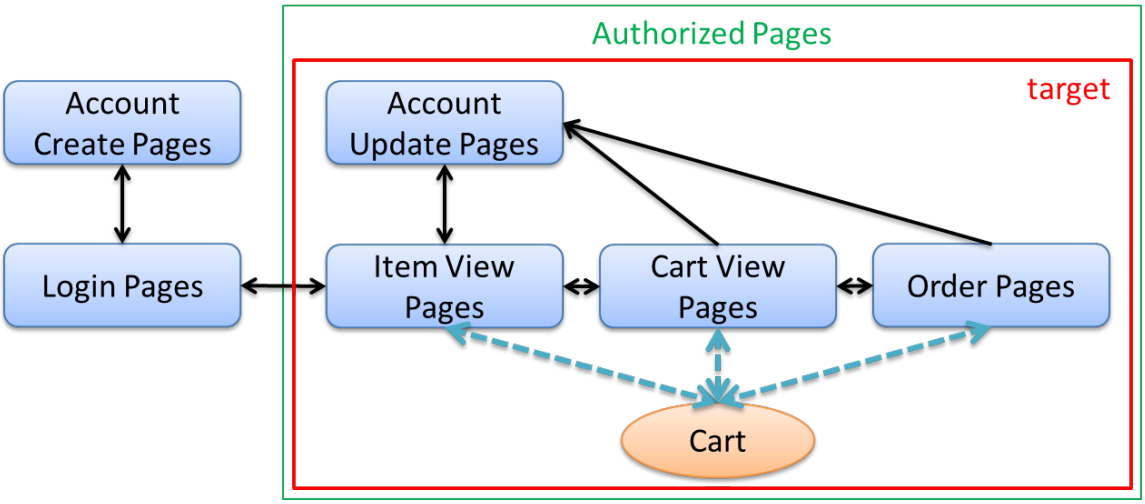
11.3.2 アプリケーションの概要と要件

概要

簡易 EC サイトを作成する。EC サイトにおいて、ユーザは以下が行える。

- アカウントでログインできる
- アカウントを作成する
- 作成したアカウント情報を変更する
- EC サイトで扱っている商品一覧を見る
- 商品の詳細を見る
- 購入したい商品をカートに登録する
- カートに登録した商品をカートから削除する
- カート内の商品を注文する

アプリケーションの概要を以下の図に示す。図中の XxxPages は画面群を表している。本チュートリアルでは、1 つの画面群で行われるシステムとユーザとのやり取りを 1 つのユースケースとして扱う。



要件

機能要件

本アプリケーションでは、前述の各画面 (ユースケース) に対して以下の機能を実装する。

画面 (ユースケース)	機能
Login Pages	ログイン機能 (作成済み)
Account Create Pages	アカウント作成機能 (作成済み)
Account Update Pages	アカウント情報変更機能
Item View Pages	商品一覧表示機能 (作成済み) 商品詳細表示機能 (作成済み) カートアイテム登録機能
Cart View Pages	カートアイテム削除機能
Order Pages	商品注文機能

本チュートリアルの初期資材として提供されるプロジェクトでは、あらかじめ一部の機能が作成されている。これは、セッション管理に直接関連しない部分を作成するコストを削減することを目的としている。

本チュートリアルでは、未完成の機能を作成する。また、未完成の機能においても、ドメイン層・インフラストラクチャ層の実装は作成済みである。したがって、本チュートリアルでは、未完成機能の画面とアプリケーション層の作成を行う。

非機能要件

実際のアプリケーションを作成する際には、そのシステムに求められている非機能要件を考慮して設計、実装する必要がある。本チュートリアルでは以下のような非機能要件があることを仮定して設計・作成を行う。以下で示されている各要件の具体的な数値は学習のための仮想的な値である。本チュートリアルで作成したアプリケーションが実際に要件を満たすことを保証できないので注意されたい。

可用性

- 運用期間：24 時間
- 年に数日の計画停止日あり
- 1 時間ほどの停止は許容

- 障害復帰は 1 営業日以内を目標とする
- 稼働率：99%

使用性

- 複数ブラウザ及びタブ上での動作保証はしない

性能

- ユーザ数：10,000 人
- 同時アクセス数：200 人
- オンライン処理件数：10,000 件 / 月
- ユーザ数・同時アクセス数・オンライン処理件数ともに 1 年で 1.2 倍の増大が見込まれる

セッション管理の設計をするうえで、以下の項目を検討する際に上記要件を考慮する必要がある。

要件	検討項目
可用性	• 複数サーバ運用におけるレプリケーションの有無
使用性	• データの整合性の保持
性能	• 複数サーバ運用におけるレプリケーションの有無 • メモリ使用量

また、上記以外にも個人情報・クレジットカード情報といった重要情報の持ち回りもセッション管理の設計の中で考慮すべきである。

基盤構成

本チュートリアルで作成するアプリケーションは以下の基盤上で動作させるものとする。以下で示されている構成の具体的な数値は学習のための仮想的な値である。

- Web・AP・DB の各サーバは 2 台構成とする。
- AP サーバのメモリ搭載量は 8GB、2 つ空きスロットあり

セッション管理の設計をするうえで、メモリ使用量やレプリケーションの有無を検討する際に上記構成を考慮する必要がある。

11.3.3 アプリケーションの設計

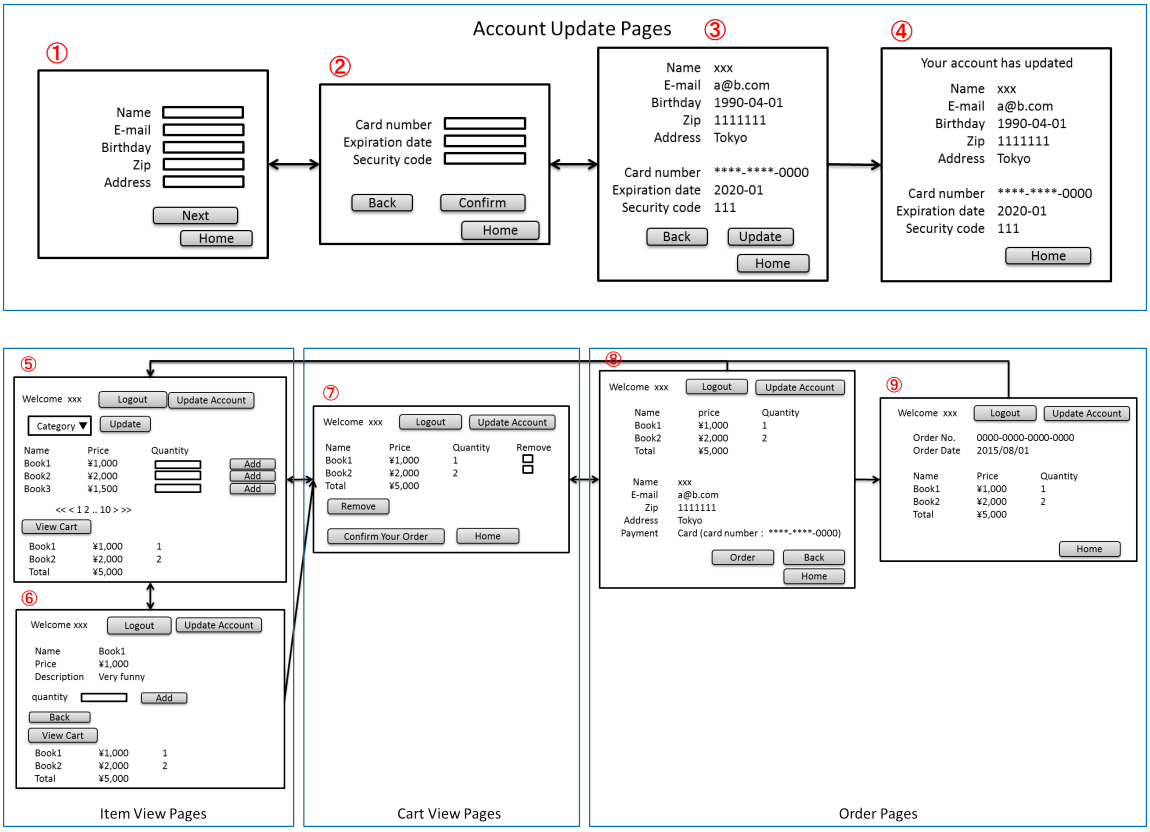
前述の要件をもとに、アプリケーションの作成方針を決定する。本チュートリアルではドメイン層・インフラストラクチャ層は作成済みであるため、アプリケーション層に関連する項目のみを対象とする。また、本チュートリアルはセッションの利用方法を学習することを目的としているため、セッション管理に直接関連しない項目は記載を省略する。

> **警告:** 本章では、セッションを利用するプロセスの一例を示しているという点に留意する。実際の開発では、案件ごとにある作業要領・作業手順に従う必要がある。

画面定義

要件をもとにアプリケーションが表示する画面を定義する。画面定義プロセスの詳細は省略する。

最終的に定義した本チュートリアルで作成する画面のイメージは以下のとおりである。



上記の図では省略されているが、他に以下の遷移が存在する。

- ログイン画面からログインすると、⑤の画面に遷移する
- Account Update Pages の各画面で「Home」ボタンを押すと、⑤の画面に遷移する
- Item View Pages、Cart View Pages、Order Pages の各画面で「Update Account」ボタンを押すと、①の画面に遷移する
- Item View Pages、Cart View Pages、Order Pages の各画面で「Logout」ボタンを押すと、ログイン画面に遷移する

URL の抽出

画面イメージをもとに、アプリケーションが処理をする URL を決定する。

画面から発生するイベントごとに URL とパラメータを設定する。それぞれ、次の規約通りに名称を付与する。

- URL：/<ユースケース名>
- パラメータ：?<処理名>

本アプリケーションではアカウント作成と更新でユースケースが分かれるため、それぞれ /account/create, /account/update という URL とする。

また、各 URL を処理する Controller も決定する。基本的に 1 つのユースケースを 1 つの Controller で処理させる。

最終的に、抽出された URL は以下のように整理できる。作成済みと書かれている Controller は、初期資材として提供されるプロジェクトに存在している。また、作成済みと書かれているパスは、そのパスにアクセスした際の処理が前述の作成済み Controller 内に既に書かれている。

項番	処理名	HTTP メソッド	パス	Controller 名	画面
(1)	アカウント情報変更画面 1 表示処理	GET	/account/update?form1	AccountUpdateController	/account/updateForm1
(2)	アカウント情報変更画面 2 表示処理	POST	/account/update?form2	AccountUpdateController	/account/updateForm2
(3)	アカウント情報変更確認画面表示処理	POST	/account/update?confirm	AccountUpdateController	/account/updateConfirm
(4)	アカウント情報変更処理	POST	/account/update	AccountUpdateController	アカウント情報変更完了画面表示処理へリダイレクト
(5)	アカウント情報変更完了画面表示処理	GET	/account/update?finish	AccountUpdateController	/account/updateFinish

Continued on next page

表 11.5 – continued from previous page

項番	処理名	HTTP メソッド	パス	Controller 名	画面
(6)	アカウント情報変更画面 1 に戻る処理	POST	/account/update?redoform1	AccountUpdateController	/account/updateForm1
(7)	アカウント情報変更画面 2 に戻る処理	POST	/account/update?redoform2	AccountUpdateController	/account/updateForm2
(8)	ホームに戻る処理	GET	/account/update?home	AccountUpdateController	商品一覧画面表示処理にリダイレクト
(9)	商品一覧画面表示処理 (デフォルト)	GET	/goods (作成済み)	GoodsController (作成済み)	/goods/showGoods
(10)	商品一覧画面表示処理 (カテゴリ選択時)	GET	/goods?categoryId (作成済み)	GoodsController (作成済み)	/goods/showGoods
(11)	商品一覧画面表示処理 (ページ選択時)	GET	/goods?page (作成済み)	GoodsController (作成済み)	/goods/showGoods
(12)	商品詳細画面表示処理	GET	/goods?{goodsId} (作成済み)	GoodsController (作成済み)	/goods/showGoodsDetail
(13)	商品をカートへ追加処理	GET	/addToCart	GoodsController (作成済み)	商品一覧画面表示処理へリダイレクト
(14)	カート画面表示処理	GET	/cart	CartController	cart/viewCart
(15)	商品をカートから削除処理	POST	/cart	CartController	カート画面表示処理へリダイレクト

Continued on next page

表 11.5 – continued from previous page

項番	処理名	HTTP メソッド	パス	Controller 名	画面
(16)	注文確認画面表示処理	GET	/order?confirm	OrderController	order/confirm
(17)	注文処理	POST	/order	OrderController	注文完了画面表示処理へリダイレクト
(18)	注文完了画面表示処理	GET	/order?finish	OrderController	order/finish

入出力データの設計

画面イメージをもとに、アプリケーションが扱う入出力データを設計する。

データの抽出

アプリケーションの画面で扱う入出データを抽出する。前述の画面イメージをもとに以下のデータが抽出できる。

項番	データ項目名	データの要素
(1)	アカウント更新情報	アカウント名、メールアドレス、誕生日、郵便番号、住所、カード番号、有効期限、セキュリティコード
(2)	アカウント情報	アカウント名、メールアドレス、パスワード、誕生日、郵便番号、住所、カード番号、有効期限、セキュリティコード
(3)	商品検索情報	選択カテゴリ、ページ番号
(4)	商品情報	商品名、単価、説明、(商品 ID)
(5)	カート登録情報	数量、(商品 ID)
(6)	カート情報	商品名、単価、数量、(商品 ID)
(7)	カート削除情報	商品 ID リスト
(8)	注文情報	注文 ID、注文日時、(アカウント ID)、商品名、単価、数量

ライフサイクルの定義

前項で抽出したデータのライフサイクルを定義する。ライフサイクルの定義では、データがいつ生成されていつ破棄されるかを決定する。

複数画面にわたって保持する必要があるデータは、以下のように破棄のタイミングが複数あるので注意する必要がある。

- 業務が通常のフローで終了する
- 業務の途中でその業務を中止する

上記注意事項を考慮すると、前項で抽出したデータのライフサイクルを以下のように定義できる。

項番	データ項目名	ライフサイクル
(1)	アカウント更新情報	画面①からの入力によって生成し、①～③を遷移する間は保持する。画面①～③以外に遷移した場合に破棄する。
(2)	アカウント情報	ログイン時に生成し、ログアウト時に破棄する。
(3)	商品検索情報	画面⑤に遷移した際に生成し、①～⑧を遷移する間は保持する。画面⑨に遷移した場合に破棄する。
(4)	商品情報	画面⑤または⑥に遷移する際に生成し、そのリクエスト間のみ保持する。
(5)	カート登録情報	画面⑤または⑥からの入力によって生成し、そのリクエスト間のみ保持する。
(6)	カート情報	画面⑤に遷移する際に空のオブジェクトを生成し、①～⑧を遷移する間は保持する。画面⑨に遷移した場合に破棄する。
(7)	カート削除情報	画面⑦からの入力によって生成し、そのリクエスト間のみ保持する。
(8)	注文情報	画面⑨に遷移する際に生成し、そのリクエスト間のみ保持する。

セッション利用有無の判断

複数画面にわたって情報を保持する必要がある場合、セッションを利用することで実現が容易となる。一方で、セッションを利用する場合、そのデメリットも考慮する必要がある。本チュートリアルでは、ガイドラインの セッション管理を参考にセッションを利用するか否かを判断する。

ガイドラインには、まずセッションを使わない方針で検討して本当に必要なデータのみセッションに格納することを推奨するとの記述がある。本チュートリアルでもセッションを使わない方針で検討を行う。

データ項目	検討内容
アカウント更新情報	アカウント更新情報は 3 画面にまたがって保持されるため、hidden を用いたデータの持ち回りが必要となる。しかし、アカウント更新情報にはカード番号等の重要情報が含まれる。hidden を用いた持ち回りでは、重要情報がマスクされず HTML のソースに書かれてしまうため、セキュリティ上問題となる。そのため、本チュートリアルではセッションを利用することを検討する。
アカウント情報	ログイン後のすべての画面で保持されるため、hidden を用いたデータの持ち回りが必要となる。この場合、作成するほぼすべての画面でデータ持ち回りの処理を記述しなければならない。そのため、画面の実装コストを抑えるためにも、本チュートリアルではセッションを利用することを検討する。
商品検索情報	商品検索情報は 8 画面にまたがって保持されるため、hidden を用いたデータの持ち回りが必要となる。この場合、作成するほぼすべての画面でデータ持ち回りの処理を記述しなければならない。そのため、画面の実装コストを抑えるためにも、本チュートリアルではセッションを利用することを検討する。
商品情報	商品情報は 1 画面でのみ利用されるため、リクエストスコープでデータを扱えばよい。
カート登録情報	カート登録情報は 1 画面でのみ利用されるため、リクエストスコープでデータを扱えばよい。
カート情報	カート情報は 8 画面にまたがって保持されるため、hidden を用いたデータの持ち回りが必要となる。この場合、作成するほぼすべての画面でデータ持ち回りの処理を記述しなければならない。そのため、画面の実装コストを抑えるためにも、本チュートリアルではセッションを利用することを検討する。

Continued on next page

表 11.6 – continued from previous page

データ項目	検討内容
カート削除情報	カート削除情報は 1 画面でのみ利用されるため、リクエストスコープでデータを扱えばよい。
注文情報	注文情報は 1 画面でのみ利用されるため、リクエストスコープでデータを扱えばよい。

以上から、アカウント更新情報、アカウント情報、カート情報、商品検索情報の 4 つについて、セッションを利用することを検討する。

次に、セッションを利用することのデメリットを検証する。この検証によって、デメリットの影響が無視できないと判断される場合はセッションを利用しない。

セッション利用によるデメリットとして大きく以下の 3 点が挙げられる。

- 複数タブ、複数ブラウザで利用した場合、互いの操作によってデータの整合性が失われる可能性がある(ことを考慮する必要がある)。
- メモリ上で管理されるため、管理するデータのサイズによってはメモリ枯渇の恐れがある。
- スケールアウトの実施や高い可用性の獲得を目的として AP サーバを多重化した際に、セッションのレプリケーションを考慮する必要がある。その際、大量のデータをセッションで扱っていると、性能等に影響する可能性がある。

上記の観点について、それぞれ該当するリスクにどう対処するかやリスクを許容するかを検討する。

観点	検討内容
データの整合性	本アプリケーションでは、複数ブラウザ及びタブ上での動作保証はしない。そのため、データの整合性を担保する対策は不要である。
メモリ使用量	セッションの利用を検討しているデータのサイズを見積もる。文字列要素は最大 100 文字 240 バイト (4 文字 8 バイト + 初期 40 バイト)、日付要素は 24 バイト、数値要素は 16 バイトとして推定する。また、ログイン認証時にセッションへ格納される認証情報 `UserDetails` のサイズも含める。`UserDetails` には大きく、ID、パスワード、ユーザの権限が含まれる。ユーザの権限は複数指定できるが、ここでは 1 つとして推定を行う。各項目の推定結果は、以下のようになる。 • アカウント情報 (文字列：7 項目、日付：2 項目)： 最大 1.7K バイト • アカウント変更情報 (文字列：8 項目、日付：2 項目)： 最大 2.0K バイト • カート情報 (最大 19 商品× (文字列:3 項目、数値：3 項目))： 最大 14.6K バイト • 商品検索情報 (数値：2 項目)：32 バイト • `UserDetails`：(文字列：3 項目)：0.7K バイト 1 ユーザで最大合計 19KB 使用する。安全率を 10% と考慮すると 1 ユーザ約 21KB 使用する。同時接続人数 1 万人を考慮しても使用量は約 210MB であり、その他のメモリ使用量を考えてもメモリ搭載量 8GB を大幅に下回るため、メモリ枯渇が発生する可能性は小さい。
AP サーバの多重化	本アプリケーションでは高い可用性は求められていないため、障害発生時におけるユースケースの継続は不要で、再ログインによるユースケースのやり直しを許容している。そのため、同一セッション内で発生するリクエストを全て同じ AP サーバに振り分けるようにロードバランサを設定する対処のみとし、セッションの AP サーバ間でのレプリケーションを実現しない。

警告: オブジェクトのサイズを推定するには、オブジェクトのサイズを計測するためのツール (例えば SizeOf など) を用いる必要がある。本チュートリアルの計算式は SizeOf での実測値の傾向を参考にしているが、あくまで仮の値であることに注意する。実際のシステム開発でのサイジングの際にはどのように算出するかを個別に検討すること。

警告: メモリ枯渇を防ぐために、セッションに格納するデータは基本的に入力データに限る。検索結果等の出力データはサイズが大きくなりやすい一方、画面操作で編集することができない読み取り専用であることが多いため、セッションに格納するには向いていない。

上記以外にも、セッションキーの管理コストの増加も考慮点の 1 つではある。しかし、今回作成するアプリケーションではセッションに格納するデータ数が多くないため、セッションキーの管理コストは限定的なものであるといえる。

この結果から、セッションを利用することで発生するデメリットの影響は大きくないといえる。最終的にセッションに格納するデータは以下のとおりである。

- アカウント変更情報
- アカウント情報
- 商品検索情報
- カート情報

本チュートリアルでは、セッションを利用してデータの持ち回りを実現するという判断を下した。しかし、検討の結果、セッションを利用しないという判断を下すことも考えられる。セッションを利用しない場合は、一例として hidden を利用してデータの持ち回りを実現する。

また、セッションを利用する際にデータの整合性を保つ方式やレプリケーションの設定が必要になることがある。

ガイドラインではトランザクショントークンチェックを使用して回避する方法を挙げている。ただし、この場合ユーザビリティの低いアプリケーションとなることに注意する。具体的な実現方法は 二重送信防止 を参照されたい。

レプリケーションの設定は AP サーバに依存するため、レプリケーションを考慮する必要がある場合は、AP サーバの設定を確認する必要がある。

警告: ここで判断したデータ以外にもセッションに格納されるデータが存在する場合がある。ガイドラインにある項目のうち、以下の項目を利用する場合にセッションが使用される。

- Spring Security を利用した認証・認可・CSRF 対策を利用している
- 二重送信防止のためのトランザクショントークンチェックを利用している

セッション中のデータを利用するための実装方法

本項では、各データに対してセッション中のデータを利用するための実装方法を決定する。

ガイドラインでは、データの利用場所に応じて 2 種類の実装方法を提供している。セッション管理 では、1 つの Controller 内で完結するデータかどうかによって利用方法を区別している。したがって、セッションに格納するデータのライフサイクルと URL マッピングを考慮して実装方法を決める必要がある。また、認証情報に紐づくデータである場合は、Spring Security の機能によってセッション管理を実現することが望ましい。

これらを考慮して、セッションで扱うデータを整理した最終的な結果が以下である。

データ	特性	セッション中のデータ利用方法
アカウント変更情報	1 つの Controller 内でのみ利用される	`@SessionAttributes` アノテーションを用いた方法
アカウント情報	複数の Controller 間で利用される 認証処理で使用される	Spring Security の機能を用いた方法
商品検索情報	複数の Controller 間で利用される	Spring のセッションスコープの Bean を用いた方法
カート情報	複数の Controller 間で利用される	Spring のセッションスコープの Bean を用いた方法

アカウント情報は初期資材として提供されるプロジェクトですでに作成済みであり、Spring Security の機能を利用して管理されている。そのため、本チュートリアルでは具体的な利用方法の説明は行わない。具体的な利用方法については 認証 を参照されたい。

セッションを利用する際の考慮事項

セッションを利用することが決まった場合、以降に挙げる項目を考慮する必要がある。それぞれの項目を検討する。

セッションの同期化

同一ユーザの複数のリクエストによって、セッションに格納されているオブジェクトに同時にアクセスする可能性がある。そのため、セッションの同期化を行わない場合、想定外のエラーや、動作を引き起こす原因になりうる。

ガイドラインでは、 セッション管理 にて BeanProcessor を利用した同期化の実現方法が挙げられているため、本チュートリアルではこれを利用する。

セッションタイムアウト

セッションを利用する場合、セッションのタイムアウト時間を設定する必要がある。タイムアウト時間が長すぎれば、不要なリソースをメモリ上に持ち続けることになり、タイムアウト時間が短すぎれば、ユーザの利便性が低下する。そのため、要件に合わせて適切な時間を設定する必要がある。

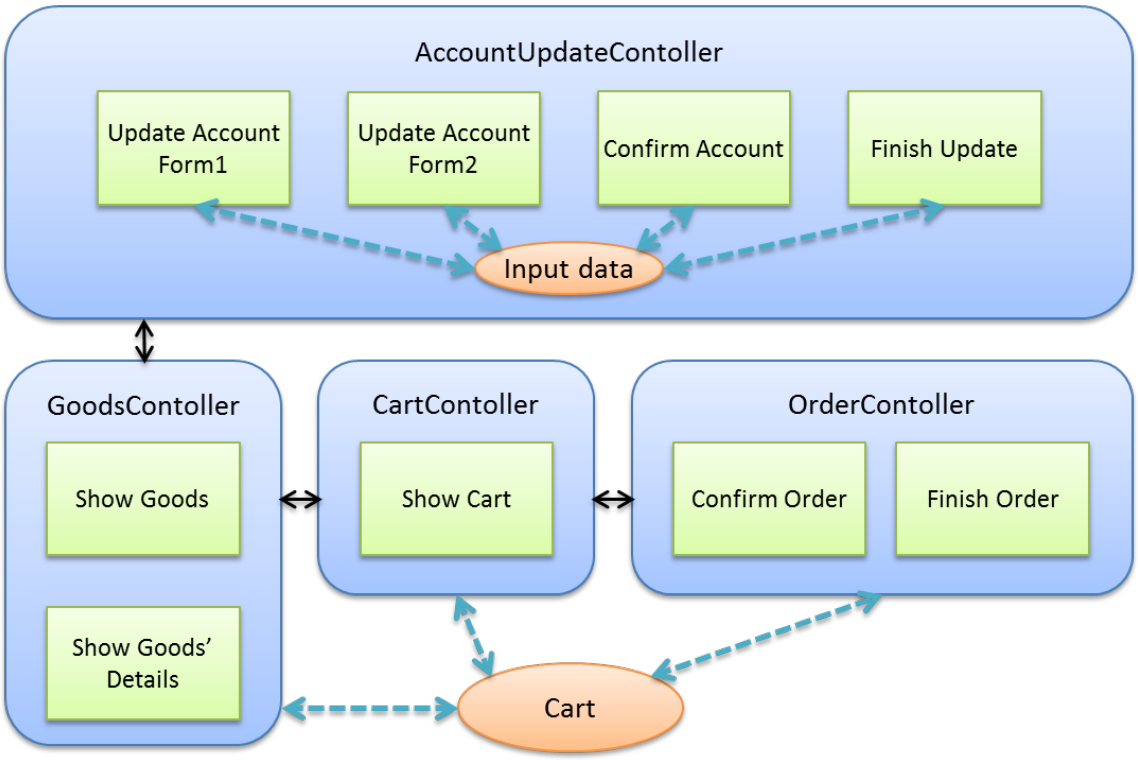
本チュートリアルでは、メモリリソースが十分に用意されていることもあり、AP サーバのデフォルト値 30 分に設定する。

また、セッションタイムアウト後のリクエストに対する処理も検討する必要がある。ガイドラインでは、 セッション管理 にてセッションタイムアウト後のリクエストを処理する方法が挙げられている。

本チュートリアルでは、タイムアウト後はログイン画面に遷移するように設定する。

アプリケーション設計の全体

最終的なアプリケーション設計の全体イメージ図を以下に示す。



11.3.4 プロジェクトの構成

プロジェクトの作成

すでに述べているように、本チュートリアルは一部機能が作成された状態からスタートする。そのため、すでに作成済みのプロジェクトを用いて開発を進める。

作成済みのプロジェクトは次の手順で取得することができる。

1. tutorial-apps にアクセスする。
2. 「Branch」ボタン押下して必要なバージョンの Branch を選択し、「Download ZIP」ボタンを押下して zip ファイルをダウンロードする
3. zip ファイルを展開し、中のプロジェクトをインポートする。

なお、プロジェクトのインポート方法は、 チュートリアル *(Todo* アプリケーション*)* で説明済みのため、本チュートリアルでは説明を割愛する。

プロジェクトの構成

git で取得した初期プロジェクトの構成について述べる。取得したプロジェクトとブランクプロジェクトとの差分のみを以下に示す。

```
session-tutorial-init-domain
    └── src
        └── main
            ├── java
            │   └── com
            │       └── example
            │           └── session
            │               └── domain
            │                   ├── model  ... (1)
            │                   │   ├── Account.java  ... (2)
            │                   │   ├── Cart.java  ... (3)
            │                   │   ├── CartItem.java  ... (3)
            │                   │   ├── Goods.java
            │                   │   ├── Order.java  ... (4)
            │                   │   └── OrderLine.java  ... (4)
            │                   ├── repository  ... (5)
            │                   │   ├── account
            │                   │   │   └── AccountRepository.java
            │                   │   ├── goods
            │                   │   │   └── GoodsRepository.java
            │                   │   └── order
            │                   │       └── OrderRepository.java
            │                   └── service  ... (6)
            │                       ├── account
            │                       │   └── AccountService.java
            │                       ├── goods
            │                       │   └── GoodsService.java
            │                       ├── order
            │                       │   ├── EmptyCartOrderException.java
            │                       │   ├── InvalidCartOrderException.java
            │                       │   └── OrderService.java
            │                       └── userdetails
            │                           ├── AccountDetails.java
            │                           └── AccountDetailsService.java
            │
            └── resources
                ├── com
                │   └── example
                │       └── session
                │           └── domain
                │               └── repository  ... (7)
                │                   ├── account
                │                   │   └── AccountRepository.xml
                │                   ├── goods
                │                   │   └── GoodsRepository.xml
                │                   └── order
                │                       └── OrderRepository.xml
                └── META-INF
                    ├── dozer
                    │   └── order-mapping.xml  ... (8)
```

```
                    └──── spring
                        └──── session-tutorial-init-codelist.xml  ... (9)
```

項番	説明
(1)	本アプリケーションで使用する model を扱うパッケージ。 チュートリアルを進める上で理解しておく必要がある model は以下で詳しく説明する。
(2)	ユーザアカウント情報を保持するクラス。
(3)	ユーザがカートに登録した商品の情報を保持するクラス。 全体を *Cart* が管理し、個別の商品を *CartItem* が管理する。
(4)	ユーザが注文した商品の情報を保持するクラス。 全体を *Order* が管理し、個別の商品を *OrderLine* が管理する。
(5)	本アプリケーションで使用する repository を扱うパッケージ。
(6)	本アプリケーションで使用する service を扱うパッケージ。
(7)	repository で使用するマッピングファイルを格納するディレクトリ。
(8)	Dozer(Bean Mapper) のマッピング定義ファイル。 *Cart* から *Order* への変換が定義されている。
(9)	本アプリケーションで使用するコードリストを定義した Bean 定義ファイル。

```
session-tutorial-init-env
    └── src
        └── main
            └── resources
                └── database  ... (1)
                    ├── H2-dataload.sql
                    └── H2-schema.sql
```

ファイル名	説明
(1)	本アプリケーションでインメモリデータベース (H2 Database) をセットアップするための SQL を格納するディレクトリ。

```
session-tutorial-init-web
    └── src
        └── main
            ├── java
            │   └── com
            │       └── example
            │           └── session
            │               └── app  ... (1)
            │                   ├── account
            │                   │   ├── AccountCreateController.java
            │                   │   ├── AccountCreateForm.java
            │                   │   ├── IlleagalOperationException.java
            │                   │   └── IlleagalOperationExceptionHandler.java
            │                   ├── goods
            │                   │   ├── GoodsController.java
            │                   │   └── GoodsViewForm.java
            │                   ├── login
            │                   │   └── LoginController.java
            │                   └── validation
            │                       ├── Confirm.java
            │                       └── ConfirmValidator.java
            ├── resources
            │   ├── i18n
            │   │   └── application-messages.properties  ... (2)
            │   ├── META-INF
            │   │   └── spring  ... (3)
            │   │       ├── spring-mvc.xml
            │   │       └── spring-security.xml
            │   └── ValidationMessages.properties  ... (2)
            └── webapp
                ├── resources  ... (4)
                │   ├── app
                │   │   └── css
                │   │       └── styles.css
                │   └── vendor
```

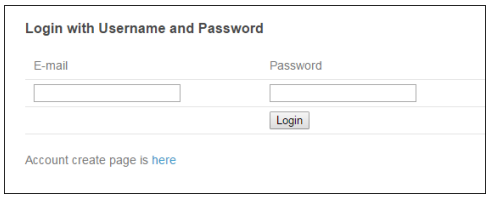
```
                    |       └── bootstrap-3.0.0
                    |           └── css
                    |               └── bootstrap.css
                    └── WEB-INF
                        └── views  ... (5)
                            ├── account
                            |   ├── createConfirm.jsp
                            |   ├── createFinish.jsp
                            |   └── createForm.jsp
                            ├── common
                            |   ├── error
                            |   |   └── illegalOperationError.jsp
                            |   └── include.jsp
                            ├── layout
                            |   └── template.jsp
                            ├── goods
                            |   ├── showGoods.jsp
                            |   └── showGoodsDetails.jsp
                            └── login
                                └── loginForm.jsp
```

項番	説明
(1)	本アプリケーションで使用するアプリケーション層のクラスを格納するためのパッケージ。
(2)	本アプリケーションで使用するメッセージが定義されているプロパティファイル
(3)	本アプリケーションで使用するコンポーネントが定義されている Bean 定義ファイル
(4)	本アプリケーションで使用する静的リソースファイル
(5)	本アプリケーションで使用する jsp が格納されているディレクトリ

動作確認

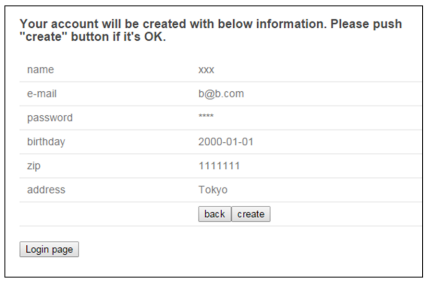
アプリケーション開発を行う前に、取得したプロジェクトの動作確認を行う。STS にインポートしたプロジェクトを対象として、アプリケーションサーバを起動するアプリケーションサーバの起動方法は、 チュートリアル *(Todo* アプリケーション*)* で説明済みのため、本チュートリアルでは説明を割愛する。

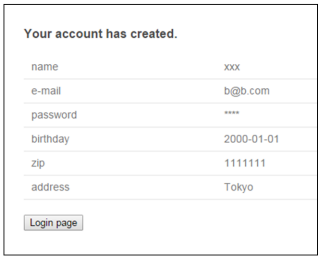
アプリケーションサーバ起動後、 http://localhost:8080/session-tutorial-init-web/loginForm にアクセスすると以下の画面が表示される。



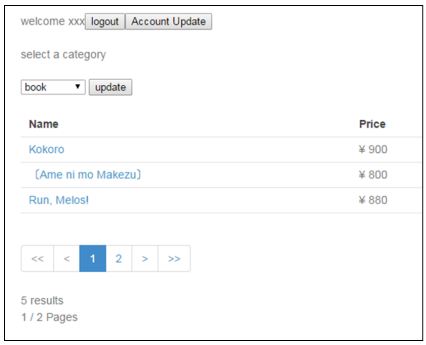
ログイン画面上にある」here」のリンクを選択すると、アカウント作成を行うことができる。

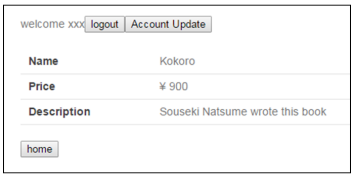






ログイン画面にて、(E-mail=」a@b.com」、Password=」demo」) をフォーム入力するとログインすることができる。ログイン後は商品一覧が表示される。商品名を選択すると商品詳細を表示できる。





11.3.5 簡易 EC サイトアプリケーションの作成

アカウント情報変更機能を作成する

ユーザに情報を入力させてアカウント情報を更新する機能を作成する。

アプリケーションの設計 で説明したとおり、アカウント変更情報は `@SessionAttributes` アノテーションを利用して管理する。

以下にアカウント情報変更機能で実装する画面の情報を示す。

処理名	HTTP メソッド	パス	画面
アカウント情報変更画面 1 表示処理	GET	/account/update?form1	/account/updateForm1
アカウント情報変更画面 2 表示処理	GET	/account/update?form2	/account/updateForm2
アカウント情報変更確認画面表示処理	GET	/account/update?confirm	/account/updateConfirm
アカウント情報変更処理	POST	/account/update	アカウント情報変更完了画面表示処理へリダイレクト
アカウント情報変更完了画面表示処理	GET	/account/update?finish	/account/updateFinish
アカウント情報変更画面 1 に戻る処理	GET	/account/update?redoform1	/account/updateForm1
アカウント情報変更画面 2 に戻る処理	GET	/account/update?redoform2	/account/updateForm2
ホームに戻る処理	GET	/account/update?home	ホーム画面表示処理にリダイレクト

フォームオブジェクトの作成

アカウント変更情報を保持するクラスを作成する。

/session-tutorial-init-web/src/main/java/com/example/session/app/account/AccountUpdateForm.java

```
package com.example.session.app.account;

import java.io.Serializable;
import java.util.Date;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.Email;
import org.springframework.format.annotation.DateTimeFormat;

public class AccountUpdateForm implements Serializable {  // (1)

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    private String id;

    // (2)
    @NotNull(groups = { Wizard1.class })
    @Size(min = 1, max = 255, groups = { Wizard1.class })
    private String name;

    @NotNull(groups = { Wizard1.class })
    @Size(min = 1, max = 255, groups = { Wizard1.class })
    @Email(groups = { Wizard1.class })
    private String email;

    @NotNull(groups = { Wizard1.class })
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    private Date birthday;

    @NotNull(groups = { Wizard1.class })
    @Size(min = 7, max = 7, groups = { Wizard1.class })
    private String zip;

    @NotNull(groups = { Wizard1.class })
    @Size(min = 1, max = 255, groups = { Wizard1.class })
    private String address;

    @Size(min = 16, max = 16, groups = { Wizard2.class })
    private String cardNumber;

    @DateTimeFormat(pattern = "yyyy-MM")
    private Date cardExpirationDate;

    @Size(min = 1, max = 255, groups = { Wizard2.class })
    private String cardSecurityCode;
```

```
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public String getZip() {
        return zip;
    }

    public void setZip(String zip) {
        this.zip = zip;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String getCardNumber() {
        return cardNumber;
    }
```

```
    public void setCardNumber(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    public Date getCardExpirationDate() {
        return cardExpirationDate;
    }

    public void setCardExpirationDate(Date cardExpirationDate) {
        this.cardExpirationDate = cardExpirationDate;
    }

    public String getCardSecurityCode() {
        return cardSecurityCode;
    }

    public void setCardSecurityCode(String cardSecurityCode) {
        this.cardSecurityCode = cardSecurityCode;
    }

    public String getLastFourOfCardNumber() {
        if (cardNumber == null) {
            return "";
        }
        return cardNumber.substring(cardNumber.length() - 4);
    }

    public static interface Wizard1 {

    }

    public static interface Wizard2 {

    }
}
```

項番	説明
(1)	このクラスのインスタンスをセッションに格納するため、 Serializable を実装しておく。
(2)	画面遷移ごとに入力チェックの対象を指定するために、バリデーションのグループ化を行う。 上記例では、1 ページ目の入力項目と 2 ページ目の入力項目にそれぞれに対応した入力チェックを実現するために、2 つのグループを作成している。

Controller の作成

Controller を作成する。Controller では、入力情報を受け取るフォームを `@SessionAttributes` アノテーションで管理させる記述が必要である。

`/session-tutorial-init-web/src/main/java/com/example/session/app/account/AccountUpdateController.java`

```
package com.example.session.app.account;

import javax.inject.Inject;

import org.dozer.Mapper;
import org.springframework.beans.propertyeditors.StringTrimmerEditor;
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.terasoluna.gfw.common.message.ResultMessages;

import com.example.session.app.account.AccountUpdateForm.Wizard1;
import com.example.session.app.account.AccountUpdateForm.Wizard2;
import com.example.session.domain.model.Account;
import com.example.session.domain.service.account.AccountService;
import com.example.session.domain.service.userdetails.AccountDetails;

@Controller
@RequestMapping("account/update")
@SessionAttributes(value = { "accountUpdateForm" }) // (1)
public class AccountUpdateController {

    @Inject
    AccountService accountService;

    @Inject
    Mapper beanMapper;

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        binder.registerCustomEditor(String.class, new StringTrimmerEditor(true));
    }

    @ModelAttribute(value = "accountUpdateForm") // (2)
    public AccountUpdateForm setUpAccountForm() {
```

```
        return new AccountUpdateForm();
    }

    @GetMapping(params = "form1")
    public String showUpdateForm1(
            @AuthenticationPrincipal AccountDetails userDetails,
            AccountUpdateForm form) { // (3)

        Account account = accountService.findOne(userDetails.getAccount()
                .getEmail());
        beanMapper.map(account, form);

        return "account/updateForm1";
    }

    @PostMapping(params = "form2")
    public String showUpdateForm2(
            @Validated(Wizard1.class) AccountUpdateForm form,
            BindingResult result) {

        if (result.hasErrors()) {
            return "account/updateForm1";
        }

        return "account/updateForm2";
    }

    @PostMapping(params = "redoForm1")
    public String redoUpdateForm1() {
        return "account/updateForm1";
    }

    @PostMapping(params = "confirm")
    public String confirmUpdate(
            @Validated(Wizard2.class) AccountUpdateForm form,
            BindingResult result) {

        if (result.hasErrors()) {
            return "account/updateForm2";
        }

        return "account/updateConfirm";
    }

    @PostMapping(params = "redoForm2")
    public String redoUpdateForm2() {
        return "account/updateForm2";
    }

    @PostMapping
    public String update(
            @AuthenticationPrincipal AccountDetails userDetails,
            @Validated({ Wizard1.class, Wizard2.class }) AccountUpdateForm form,
```

```
            BindingResult result, RedirectAttributes attributes, SessionStatus␣
↪sessionStatus) {

        if (result.hasErrors()) {
            ResultMessages messages = ResultMessages.error();
            messages.add("e.st.ac.5001");
            throw new IllegalOperationException(messages);
        }

        Account account = beanMapper.map(form, Account.class);
        accountService.update(account);
        userDetails.setAccount(account);
        attributes.addFlashAttribute("account", account);
        sessionStatus.setComplete();  // (4)

        return "redirect:/account/update?finish";
    }

    @GetMapping(params = "finish")
    public String finishUpdate() {
        return "account/updateFinish";
    }

    @GetMapping(params = "home")
    public String home(SessionStatus sessionStatus) {
        sessionStatus.setComplete();
        return "redirect:/goods";
    }

}
```

項番	説明
(1)	`@SessionAttributes` アノテーションの `value` 属性に、セッションに格納するオブジェクトの属性名を指定する。 上記例は、属性名が `accountUpdateForm` のオブジェクトが、セッションに格納される。
(2)	`Model` オブジェクトに格納する属性名を、 `value` 属性に指定する。 上記例では、返却したオブジェクトが、 `accountUpdateForm` という属性名でセッションに格納される。 `value` 属性を指定した場合、セッションにオブジェクトを格納した後のリクエストで、`@ModelAttribute` アノテーションの付与されたメソッドが呼び出されなくなるため、無駄なオブジェクトの生成が行われないというメリットがある。
(3)	`@SessionAttributes` アノテーションによって管理されたオブジェクトを利用するには、そのオブジェクトを受け取れるようメソッドに引数を追加する。 入力チェックが必要であれば `@Validated` アノテーションを利用する。 上記例では、 `AccountUpdateForm` のデフォルトの属性名である `accountUpdateForm` を属性名にもつオブジェクトが引数として渡される。
(4)	`SessionStatus` オブジェクトの `setComplete` メソッドを呼び出し、オブジェクトをセッションから削除する。

警告: `@SessionAttributes` アノテーションで管理しているオブジェクトは、明示的に削除を行わない限りセッション中に残り続ける。そのため、Controller が扱う画面外に遷移して再度戻ってきた場合にも保持していたデータを参照できる。メモリの枯渇を防ぐために、不要になったデータは必ず削除すること。

警告: ブラウザのボタンでバックされたり、URL を直接入力して画面遷移した場合は、`setComplete` メソッドが呼ばれず、セッションがクリアされずに残ってしまう点に留意する必要がある。

JSP の作成

`@SessionAttributes` アノテーションで管理しているフォームオブジェクトにデータの受け渡しをする画

面を作成する。

1 ページ目の入力画面

/session-tutorial-init-web/src/main/webapp/WEB-INF/views/account/updateForm1.jsp

```
<div>
    <%-- (1) --%>
    <form:form action="${pageContext.request.contextPath}/account/update"
        method="post" modelAttribute="accountUpdateForm">

        <h2>Account Update Page 1/2</h2>
        <table>
            <tr>
                <td><form:label path="name" cssErrorClass="error-label">name</
↪form:label></td>
                <%-- (2) --%>
                <td><form:input path="name" cssErrorClass="error-input" />
                    <form:errors path="name" cssClass="error-messages" />
                </td>
            </tr>
            <tr>
                <td><form:label path="email" cssErrorClass="error-label">e-mail</
↪form:label></td>
                <td><form:input path="email" cssErrorClass="error-input" />
                    <form:errors path="email" cssClass="error-messages" />
                </td>
            </tr>
            <tr>
                <td><form:label path="birthday" cssErrorClass="error-label">
↪birthday</form:label></td>
                <td><fmt:formatDate value="${accountUpdateForm.birthday}" pattern=
↪"yyyy-MM-dd" var="formattedBirthday" />
                    <input type="date" id="birthday" name="birthday" value="$
↪{formattedBirthday}">
                    <form:errors path="birthday" cssClass="error-messages" />
                </td>
            </tr>
            <tr>
                <td><form:label path="zip" cssErrorClass="error-label">zip</
↪form:label></td>
                <td><form:input path="zip" cssErrorClass="error-input" />
                    <form:errors path="zip" cssClass="error-messages" />
                </td>
            </tr>
            <tr>
                <td><form:label path="address" cssErrorClass="error-label">address
↪</form:label></td>
                <td><form:input path="address" cssErrorClass="error-input" />
                    <form:errors path="address" cssClass="error-messages" />
                </td>
            </tr>
```

```
            <tr>
                <td> </td>
                <td><input type="submit" name="form2" id="next" value="next" /></
↪td>
            </tr>
        </table>
    </form:form>

    <form method="get" action="${pageContext.request.contextPath}/account/update">
        <input type="submit" name="home" id="home" value="home" />
    </form>
</div>
```

項番	説明
(1)	入力データを受け取るフォームオブジェクトの属性名を modelAttribute 属性に指定する。 上記例は、属性名が accountUpdateForm のオブジェクトが入力データを受け取る。
(2)	form:input タグの path 属性に入力データを格納するオブジェクトの要素名を指定する。 この方法を利用すると、指定したオブジェクトの要素名にすでにデータがある場合、その値が入力フォームのデフォルト値となる。

2 ページ目の入力画面

/session-tutorial-init-web/src/main/webapp/WEB-INF/views/account/updateForm2.jsp

```
<div>

    <form:form action="${pageContext.request.contextPath}/account/update" method=
↪"post" modelAttribute="accountUpdateForm">

        <h2>Account Update Page 2/2</h2>
        <table>
            <tr>
                <td><form:label path="cardNumber" cssErrorClass="error-label">your␣
↪card number</form:label></td>
                <td><form:input path="cardNumber" cssErrorClass="error-input" />
                    <form:errors path="cardNumber" cssClass="error-messages" />
                </td>
            </tr>
            <tr>
                <td><form:label path="cardExpirationDate" cssErrorClass="error-
↪label">expiration date of your card</form:label></td>
                <td><fmt:formatDate value="${accountUpdateForm.cardExpirationDate}
↪" pattern="yyyy MM" var="formattedCardExpirationDate" />
```

```
                    <input type="month" name="cardExpirationDate" id=
↪"cardExpirationDate" value="${formattedCardExpirationDate}">
                    <form:errors path="cardExpirationDate" cssClass="error-messages
↪" />
                </td>
            </tr>
            <tr>
                <td><form:label path="cardSecurityCode" cssErrorClass="error-label
↪">security code of your card</form:label></td>
                <td><form:input path="cardSecurityCode" cssErrorClass="error-input
↪" />
                    <form:errors path="cardSecurityCode" cssClass="error-messages"␣
↪/>
                </td>
            </tr>
            <tr>
                <td> </td>
                <td><input type="submit" name="redoForm1" id="back" value="back" />
                    <input type="submit" name="confirm" id="confirm" value="confirm
↪" />
                </td>
            </tr>
        </table>
    </form:form>

    <form method="get" action="${pageContext.request.contextPath}/account/update">
        <input type="submit" name="home" id="home" value="home" />
    </form>
</div>
```

確認画面

/session-tutorial-init-web/src/main/webapp/WEB-INF/views/account/updateConfirm.jsp

```
<div>

    <form:form action="${pageContext.request.contextPath}/account/update" method=
↪"post">

        <h3>Your account will be updated with below information. Please push
↪"update" button if it's OK.</h3>
        <table>
            <tr>
                <td><label for="name">name</label></td>
                <td id="name">${f:h(accountUpdateForm.name)}</td>
            </tr>
            <tr>
                <td><label for="email">e-mail</label></td>
                <td id="email">${f:h(accountUpdateForm.email)}</td>
            </tr>
            <tr>
```

```
                <td><label for="birthday">birthday</label></td>
                <td id="birthday"><fmt:formatDate value="${accountUpdateForm.
↪birthday}" pattern="yyyy-MM-dd" /></td>
            </tr>
            <tr>
                <td><label for="zip">zip</label></td>
                <td id="zip">${f:h(accountUpdateForm.zip)}</td>
            </tr>
            <tr>
                <td><label for="address">address</label></td>
                <td id="address">${f:h(accountUpdateForm.address)}</td>
            </tr>
            <tr>
                <td><label for="cardNumber">your card number</label></td>
                <td id="cardNumber">****-****-****-${f:h(accountUpdateForm.
↪lastFourOfCardNumber)}</td>
            </tr>
            <tr>
                <td><label for="cardExpirationDate">expiration date of your card</
↪label></td>
                <td id="cardExpirationDate"><fmt:formatDate value="$
↪{accountUpdateForm.cardExpirationDate}" pattern="yyyy-MM" /></td>
            </tr>
            <tr>
                <td><label for="cardSecurityCode">security code of your card</
↪label></td>
                <td id="cardSecurityCode">${f:h(accountUpdateForm.
↪cardSecurityCode)}</td>
            </tr>
            <tr>
                <td> </td>
                <td><input type="submit" name="redoForm2" id="back" value="back" />
                    <input type="submit" id="update" value="update" />
                </td>
            </tr>
        </table>
    </form:form>

    <form method="get" action="${pageContext.request.contextPath}/account/update">
        <input type="submit" name="home" id="home" value="home" />
    </form>
</div>
```

完了画面

/session-tutorial-init-web/src/main/webapp/WEB-INF/views/account/updateFinish.jsp

```
<div>

    <h3>Your account has updated.</h3>
    <table>
```

```
        <tr>
            <td><label for="name">name</label></td>
            <td id="name">${f:h(account.name)}</td>
        </tr>
        <tr>
            <td><label for="email">e-mail</label></td>
            <td id="email">${f:h(account.email)}</td>
        </tr>
        <tr>
            <td><label for="birthday">birthday</label></td>
            <td id="birthday"><fmt:formatDate value="${account.birthday}" pattern=
↪"yyyy-MM-dd" /></td>
        </tr>
        <tr>
            <td><label for="zip">zip</label></td>
            <td id="zip">${f:h(account.zip)}</td>
        </tr>
        <tr>
            <td><label for="address">address</label></td>
            <td id="address">${f:h(account.address)}</td>
        </tr>
        <tr>
            <td><label for="cardNumber">your card number</label></td>
            <td id="cardNumber">****-****-****-${f:h(account.lastFourOfCardNumber)}
↪</td>
        </tr>
        <tr>
            <td><label for="cardExpirationDate">expiration date of your card</
↪label></td>
            <td id="cardExpirationDate"><fmt:formatDate value="${account.
↪cardExpirationDate}" pattern="yyyy-MM" /></td>
        </tr>
        <tr>
            <td><label for="cardSecurityCode">security code of your card</label></
↪td>
            <td id="cardSecurityCode">${f:h(account.cardSecurityCode)}</td>
        </tr>
    </table>

    <form method="get" action="${pageContext.request.contextPath}/account/update">
        <input type="submit" name="home" id="home" value="home" />
    </form>

</div>
```

動作確認

ここまでの実装でアカウント情報更新を行うことができるようになっている。商品一覧表示画面の上部にある「Account Update」のボタンを押下することでアカウント情報更新画面に遷移する。現在、ログインしているアカウントの情報が初期値としてフォームに表示される。フォームの値を変更して次の画面に進んでいくことで、最終的にアカウントの情報が更新される。

ここまでの実装で入力値を受け取るフォームをセッションに格納しているため、データの持ち回りが簡単に実現できる。また、「home」ボタンを押した際にセッションが破棄されるため、「home」ボタンを押した後にアカウント情報更新画面に遷移すると、変更情報がリセットされる。

カートアイテム登録機能を作成する

指定した数量で商品をカートに登録する機能を作成する。

アプリケーションの設計 で説明したとおり、カート情報はセッションスコープの Bean として管理する。

以下にカートアイテム登録機能で実装する画面の情報を示す。

処理名	HTTP メソッド	パス	画面
商品をカートへ追加処理	POST	/addToCart	商品一覧画面表示処理へリダイレクト

セッションスコープ Bean を定義

カート情報を保持するオブジェクトは、`Cart.java` としてすでに作成済みである。そのため、このオブジェクトをセッションスコープの Bean として扱えるように設定を加える。

セッションスコープの Bean を使用する方法として、 セッション管理 に 2 種類の設定方法が記載されている。本チュートリアルでは、component-scan を使用して bean を定義する。

> **警告:** セッションスコープの Bean として登録するためには対象のオブジェクトが *Serializable* である必要がある

component-scan を用いてセッションスコープの Bean を定義するには、Bean として登録したいクラスに以下のアノテーションを追加すればよい。

`/session-tutorial-init-domain/src/main/java/com/example/session/domain/model/Cart.java`

```
package com.example.session.domain.model;

import java.io.Serializable;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Collection;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;

import org.springframework.context.annotation.Scope;
```

```
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.security.crypto.codec.Base64;
import org.springframework.stereotype.Component;
import org.springframework.util.SerializationUtils;

@Component // (1)
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS) // (2)
public class Cart implements Serializable {

    // omitted

}
```

項番	説明
(1)	component-scan の対象となるように`@Component` アノテーションを指定する
(2)	Bean のスコープを `session` にする。また、 `proxyMode` 属性で `ScopedProxyMode.TARGET_CLASS` を指定し、scoped-proxy を有効にする。

また、component-scan の対象となる base-package を Bean 定義ファイルに指定する必要がある。しかし、本チュートリアルでは作成済みの Bean 定義ファイルにすでに以下の記述があるため、新たに記述を追加する必要はない。

`/session-tutorial-init-domain/src/main/resources/META-INF/spring/session-tutorial-init-domain.xml`

```
<!-- (1) -->
<context:component-scan base-package="com.example.session.domain" />
```

項番	説明
(1)	component-scan の対象となるパッケージを指定する。

フォームオブジェクトの作成

注文する商品の情報を保持するクラスを作成する。

`/session-tutorial-init-web/src/main/java/com/example/session/app/goods/GoodAddForm.java`

```
package com.example.session.app.goods;

import java.io.Serializable;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public class GoodAddForm implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @NotNull
    private String goodsId;

    @NotNull
    @Min(1)
    private int quantity;

    public String getGoodsId() {
        return goodsId;
    }

    public void setGoodsId(String goodsId) {
        this.goodsId = goodsId;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

Controller の作成

Controller を作成する。

一部リクエストを処理するためにすでに作成されているため、以下のコードを追加する。

`/session-tutorial-init-web/src/main/java/com/example/session/app/goods/GoodsController.java`

```
package com.example.session.app.goods;

import javax.inject.Inject;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.terasoluna.gfw.common.message.ResultMessages;

import com.example.session.domain.model.Cart;
import com.example.session.domain.model.CartItem;
import com.example.session.domain.model.Goods;
import com.example.session.domain.service.goods.GoodsService;

@Controller
@RequestMapping("goods")
public class GoodsController {

    @Inject
    GoodsService goodsService;

    // (1)
    @Inject
    Cart cart;

    @ModelAttribute(value = "goodViewForm")
    public GoodViewForm setUpCategoryId() {
        return new GoodViewForm();
    }

    @GetMapping
    String showGoods(GoodViewForm form, Pageable pageable, Model model) {

        Page<Goods> page = goodsService.findByCategoryId(form.getCategoryId(),
                pageable);
        model.addAttribute("page", page);
        return "goods/showGoods";
    }

    @GetMapping("/{goodsId}")
    public String showGoodsDetail(@PathVariable String goodsId, Model model) {

        Goods goods = goodsService.findOne(goodsId);
        model.addAttribute(goods);

        return "goods/showGoodsDetail";
    }

    @PostMapping("/addToCart")
    public String addToCart(@Validated GoodAddForm form, BindingResult result,
```

```
            RedirectAttributes attributes) {

        if (result.hasErrors()) {
            ResultMessages messages = ResultMessages.error()
                    .add("e.st.go.5001");
            attributes.addFlashAttribute(messages);
            return "redirect:/goods";
        }

        Goods goods = goodsService.findOne(form.getGoodsId());
        CartItem cartItem = new CartItem();
        cartItem.setGoods(goods);
        cartItem.setQuantity(form.getQuantity());
        cart.add(cartItem); // (2)

        return "redirect:/goods";
    }
}
```

項番	説明
(1)	セッションスコープの Bean を DI コンテナから取得する。
(2)	セッションスコープの Bean にデータを追加する。 画面に情報を表示させるために、オブジェクトを Model に追加する必要はない。

JSP の作成

カートの中身を表示するための JSP を作成する。

JSP もすでに作成されているため、以下に示すコードを body タグの最後に追加する。

/session-tutorial-init-web/src/main/webapp/WEB-INF/views/goods/showGoods.jsp

```
<sec:authentication property="principal" var="userDetails" />
<div style="display: inline-flex">
    welcome   <span id="userName">${f:h(userDetails.account.name)}</span>
    <form method="post" action="${pageContext.request.contextPath}/logout">
        <input type="submit" id="logout" value="logout" />
        <sec:csrfInput />
    </form>
    <form method="get" action="${pageContext.request.contextPath}/account/update">
        <input type="submit" name="form1" id="updateAccount" value="Account Update
↪" />
    </form>
```

```
</div>
<br>
<br>

<div>
    <p>select a category</p>

    <form:form method="get" action="${pageContext.request.contextPath}/goods/"␣
↪modelAttribute="goodViewForm">
        <form:select path="categoryId" items="${CL_CATEGORIES}" />
        <input type="submit" id="update" value="update" />
    </form:form>
    <br />
    <t:messagesPanel />
    <table>
        <tr>
            <th>Name</th>
            <th>Price</th>
            <th>Quantity</th>
        </tr>
        <c:forEach items="${page.content}" var="goods" varStatus="status">
            <tr>
                <td><a id="${f:h(goods.name)}" href="${pageContext.request.
↪contextPath}/goods/${f:h(goods.id)}">${f:h(goods.name)}</a></td>
                <td><fmt:formatNumber value="${f:h(goods.price)}" type="CURRENCY"␣
↪currencySymbol="&yen;" maxFractionDigits="0" /></td>
                <td><form:form method="post"
                        action="${pageContext.request.contextPath}/goods/addToCart"
                        modelAttribute="goodAddForm">
                        <input type="text" name="quantity" id="quantity${status.
↪index}" value="1" />
                        <input type="hidden" name="goodsId" value="${f:h(goods.id)}
↪" />
                        <input type="submit" id="add${status.index}" value="add" />
                    </form:form>
                </td>
            </tr>
        </c:forEach>
    </table>
    <t:pagination page="${page}" outerElementClass="pagination" />
</div>
<div>
    <p>
        <fmt:formatNumber value="${page.totalElements}" /> results <br>
        ${f:h(page.number + 1) } / ${f:h(page.totalPages)} Pages
    </p>
</div>
<div>
    <%-- (1) --%>
    <spring:eval var="cart" expression="@cart" />
    <form method="get" action="${pageContext.request.contextPath}/cart">
        <input type="submit" id="viewCart" value="view cart" />
    </form>
```

```
    <table>
        <%-- (2) --%>
        <c:forEach items="${cart.cartItems}" var="cartItem" varStatus="status">
            <tr>
                <td id="itemName${status.index}">${f:h(cartItem.goods.name)}</td>
                <td id="itemPrice${status.index}"><fmt:formatNumber value="$
↪{cartItem.goods.price}" type="CURRENCY" currencySymbol="&yen;" maxFractionDigits=
↪"0" /></td>
                <td id="itemQuantity${status.index}">${f:h(cartItem.quantity)}</td>
            </tr>
        </c:forEach>
        <tr>
            <td>Total</td>
            <td id="totalPrice"><fmt:formatNumber value="${f:h(cart.totalAmount)}"␣
↪type="CURRENCY" currencySymbol="&yen;" maxFractionDigits="0" /></td>
            <td></td>
        </tr>
    </table>
</div>
```

項番	説明
(1)	セッションスコープの Bean の中身を画面に表示させるために、Bean を変数に格納する。 上記例では、セッションスコープにある Cart オブジェクトを変数 cart に格納している。
(2)	(1) で作成した変数を通して、セッションスコープの Bean の中身を参照する。 上記例では、変数 cart を通してセッションスコープの Bean の中身を参照している。

注釈: 変数に格納せず単に Bean の中身を表示させるだけであれば var 属性は不要である。上記例では、`<spring:eval expression="@cart" />` で表示できる。

`/session-tutorial-init-web/src/main/webapp/WEB-INF/views/goods/showGoodsDetail.jsp`

```
<sec:authentication property="principal" var="userDetails" />
<div style="display: inline-flex">
    welcome   <span id="userName">${f:h(userDetails.account.name)}</span>
    <form:form method="post" action="${pageContext.request.contextPath}/logout">
        <input type="submit" id="logout" value="logout" />
    </form:form>
    <form method="get" action="${pageContext.request.contextPath}/account/update">
        <input type="submit" name="form1" id="updateAccount" value="Account Update
↪" />
```

```
    </form>
</div>
<br>
<br>

<div>
    <table>
        <tr>
            <th>Name</th>
            <td id="name">${f:h(goods.name)}</td>
        </tr>
        <tr>
            <th>Price</th>
            <td id="price"><fmt:formatNumber value="${f:h(goods.price)}" type=
↪"CURRENCY" currencySymbol="&yen;" maxFractionDigits="0" /></td>
        </tr>
        <tr>
            <th>Description</th>
            <td id="description">${f:h(goods.description)}</td>
        </tr>
    </table>
    <form:form method="post"
        action="${pageContext.request.contextPath}/goods/addToCart"
        modelAttribute="AddToCartForm">
        Quantity<input type="text" id="quantity" name="quantity" value="1" />
        <input type="hidden" name="goodsId" value="${f:h(goods.id)}" />
        <input type="submit" id="add" value="add" />
    </form:form>

    <form method="get" action="${pageContext.request.contextPath}/goods">
        <input type="submit" id="home" value="home" />
    </form>
</div>
<div>
    <spring:eval var="cart" expression="@cart" />
    <form method="get" action="${pageContext.request.contextPath}/cart">
        <input type="submit" value="view cart" />
    </form>
    <table>
        <c:forEach items="${cart.cartItems}" var="cartItem" varStatus="status">
            <tr>
                <td id="itemName${status.index}">${f:h(cartItem.goods.name)}</td>
                <td id="itemPrice${status.index}"><fmt:formatNumber value="$
↪{cartItem.goods.price}" type="CURRENCY" currencySymbol="&yen;" maxFractionDigits=
↪"0" /></td>
                <td id="itemQuantity${status.index}">${f:h(cartItem.quantity)}</td>
            </tr>
        </c:forEach>
        <tr>
            <td>Total</td>
            <td id="totalPrice"><fmt:formatNumber value="${f:h(cart.totalAmount)}"␣
↪type="CURRENCY" currencySymbol="&yen;" maxFractionDigits="0" /></td>
            <td></td>
```

```
        </tr>
    </table>
</div>
```

動作確認

ここまでの実装でカートに商品を登録することができるようになっている。商品一覧表示画面で、ある商品の「add」のボタンを押下することで、同ページカートの中身が表示されるようになる。

ここまでの実装でカートオブジェクトをセッションに格納しているため、アカウント情報更新画面に遷移して戻ってきてもカートの情報は保存されている。

商品検索情報を保持する仕組みを作成する

ここまでの実装で商品をカートに追加することはできるようになった。しかし、商品追加後に遷移する画面は、常に「book」カテゴリの 1 ページ目となっている。

本チュートリアルでは、選択カテゴリやページ番号といった商品検索情報は注文が完了するまで保持する仕様となっている。そのため、商品追加後やアカウント更新画面から戻ってきたときに前の状態に遷移するように実装を修正する。

アプリケーションの設計 で説明したとおり、商品検索情報はセッションスコープの Bean として管理する。

以下に修正する画面の情報を示す。

処理名	HTTP メソッド	パス	画面
商品一覧画面表示処理 (デフォルト)	GET	/goods (作成済み)	/goods/showGoods
商品一覧画面表示処理 (カテゴリ選択時)	GET	/goods?categoryId (作成済み)	/goods/showGoods
商品一覧画面表示処理 (ページ選択時)	GET	/goods?page (作成済み)	/goods/showGoods

セッションスコープ Bean を作成

商品検索情報を保持するセッションスコープ Bean を作成する。カート情報と同様に component-scan を使用して bean を定義する。

/session-tutorial-init-web/src/main/java/com/example/session/app/goods/GoodsSearchCriteria.java

```
package com.example.session.app.goods;

import java.io.Serializable;

import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;

@Component // (1)
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS) // (2)
public class GoodsSearchCriteria implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    private int categoryId = 1;

    private int page = 0;

    public int getCategoryId() {
        return categoryId;
    }

    public void setCategoryId(int categoryId) {
        this.categoryId = categoryId;
    }

    public int getPage() {
        return page;
    }

    public void setPage(int page) {
        this.page = page;
    }

    public void clear() {
        categoryId = 1;
        page = 0;
    }

}
```

項番	説明
(1)	component-scan の対象となるように`@Component` アノテーションを指定する
(2)	Bean のスコープを `session` にする。また、proxyMode 属性で `ScopedProxyMode.TARGET_CLASS` を指定し、scoped-proxy を有効にする。

また、component-scan の対象となる base-package を Bean 定義ファイルに指定する必要がある。 しかし、本チュートリアルでは作成済みの Bean 定義ファイルにすでに以下の記述があるため、新たに記述を追加する必要はない。

`/session-tutorial-init-web/src/main/resources/META-INF/spring/spring-mvc.xml`

```
<!-- (1) -->
<context:component-scan base-package="com.example.session.app" />
```

項番	説明
(1)	component-scan の対象となるパッケージを指定する。

Controller の修正

商品検索情報をセッションで保持する、また、セッションで保持されている商品検索情報を利用するように Controller を修正する。

`/session-tutorial-init-web/src/main/java/com/example/session/app/goods/GoodsController.java`

```
package com.example.session.app.goods;

import javax.inject.Inject;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ModelAttribute;
```

```
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.terasoluna.gfw.common.message.ResultMessages;

import com.example.session.domain.model.Cart;
import com.example.session.domain.model.CartItem;
import com.example.session.domain.model.Goods;
import com.example.session.domain.service.goods.GoodsService;

@Controller
@RequestMapping("goods")
public class GoodsController {

    @Inject
    GoodsService goodsService;

    @Inject
    Cart cart;

    // (1)
    @Inject
    GoodsSearchCriteria criteria;

    @ModelAttribute(value = "goodViewForm")
    public GoodViewForm setUpCategoryId() {
        return new GoodViewForm();
    }

    // (2)
    @GetMapping
    String showGoods(GoodViewForm form, Model model) {
        Pageable pageable = new PageRequest(criteria.getPage(), 3);
        form.setCategoryId(criteria.getCategoryId());
        return showGoods(pageable, model);
    }

    // (3)
    @GetMapping(params = "categoryId")
    String changeCategoryId(GoodViewForm form, Pageable pageable, Model model) {
        criteria.setPage(pageable.getPageNumber());
        criteria.setCategoryId(form.getCategoryId());
        return showGoods(pageable, model);
    }

    // (4)
    @GetMapping(params = "page")
    String changePage(GoodViewForm form, Pageable pageable, Model model) {
        criteria.setPage(pageable.getPageNumber());
        form.setCategoryId(criteria.getCategoryId());
        return showGoods(pageable, model);
    }
```

```
    // (5)
    String showGoods(Pageable pageable, Model model) {
        Page<Goods> page = goodsService.findByCategoryId(
                criteria.getCategoryId(), pageable);
        model.addAttribute("page", page);
        return "goods/showGoods";
    }

    @GetMapping("/{goodsId}")
    public String showGoodsDetail(@PathVariable String goodsId, Model model) {

        Goods goods = goodsService.findOne(goodsId);
        model.addAttribute(goods);

        return "/goods/showGoodsDetail";
    }

    @PostMapping("/addToCart")
    public String addToCart(@Validated GoodAddForm form, BindingResult result,
            RedirectAttributes attributes) {

        if (result.hasErrors()) {
            ResultMessages messages = ResultMessages.error()
                    .add("e.st.go.5001");
            attributes.addFlashAttribute(messages);
            return "redirect:/goods";
        }

        Goods goods = goodsService.findOne(form.getGoodsId());
        CartItem cartItem = new CartItem();
        cartItem.setGoods(goods);
        cartItem.setQuantity(form.getQuantity());
        cart.add(cartItem);

        return "redirect:/goods";
    }
}
```

項番	説明
(1)	セッションスコープの Bean を DI コンテナから取得する。
(2)	通常の商品一覧画面表示処理の前処理を行う。セッションに格納されている商品カテゴリをフォームに、ページ番号を `pageable` に設定する。商品カテゴリをフォームに設定するのは、セレクトボックスで表示される商品カテゴリを指定するためである。
(3)	カテゴリが変更された時の商品一覧画面表示処理の前処理を行う。入力された商品カテゴリをセッションに格納する。ページ番号はデフォルトの 1 ページ目を `pageable` に指定する。
(4)	ページが変更された時の商品一覧画面表示処理の前処理を行う。入力されたページ番号をセッションに格納する。セッションに格納されている商品カテゴリをフォームに設定する。
(5)	共通部分を扱う。セッションで管理されている商品カテゴリ、前処理で取得した `pageable` をもとに商品を検索する。

動作確認

ここまでの実装で、商品検索情報を保持することができるようになっている。例えば、「music」カテゴリの 2 ページ目で商品をカートに追加した際の遷移先がもとの「music」カテゴリの 2 ページ目のままとなる。また、同画面から「Account Update」ボタンを押してアカウント更新画面に遷移し、アカウント更新画面の「home」ボタンを押して戻ってきた際の遷移先がもとの「music」カテゴリの 2 ページ目のままとなる。

カートアイテム削除機能を作成する

指定した商品をカートから削除する機能を作成する。

削除する商品を指定するために、チェックボックスを利用する。

以下にカートアイテム削除機能で実装する画面の情報を示す。

処理名	HTTP メソッド	パス	画面
カート画面表示処理	GET	/cart	cart/viewCart
商品をカートから削除処理	POST	/cart	カート画面表示処理へリダイレクト

フォームオブジェクトの作成

削除対象となる商品の ID を保持するクラスを作成する。

`/session-tutorial-init-web/src/main/java/com/example/session/app/cart/CartForm.java`

```
package com.example.session.app.cart;

import java.util.Set;

import org.hibernate.validator.constraints.NotEmpty;

public class CartForm {

    @NotEmpty
    private Set<String> removedItemsIds;

    public Set<String> getRemovedItemsIds() {
        return removedItemsIds;
    }

    public void setRemovedItemsIds(Set<String> removedItemsIds) {
        this.removedItemsIds = removedItemsIds;
    }
}
```

Controller の作成

Controller を作成する。

`/session-tutorial-init-web/src/main/java/com/example/session/app/cart/CartController.java`

```
package com.example.session.app.cart;

import javax.inject.Inject;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.terasoluna.gfw.common.message.ResultMessages;

import com.example.session.domain.model.Cart;

@Controller
@RequestMapping("cart")
public class CartController {

    // (1)
    @Inject
    Cart cart;

    @ModelAttribute
    CartForm setUpForm() {
        return new CartForm();
    }

    @GetMapping
    String viewCart(Model model) {
        return "cart/viewCart";
    }

    @PostMapping
    String removeFromCart(@Validated CartForm cartForm,
            BindingResult bindingResult, Model model) {
        if (bindingResult.hasErrors()) {
            ResultMessages messages = ResultMessages.error()
                    .add("e.st.ca.5001");
            model.addAttribute(messages);
            return viewCart(model);
        }
        cart.remove(cartForm.getRemovedItemsIds()); // (2)
        return "redirect:/cart";
    }
}
```

項番	説明
(1)	セッションスコープの Bean を DI コンテナから取得する。
(2)	セッションスコープの Bean のデータを削除する。

JSP の作成

カート一覧を表示し、削除したい商品を選択するための JSP を作成する。この画面から商品注文が行える。

/session-tutorial-init-web/src/main/webapp/WEB-INF/views/cart/viewCart.jsp

```
<sec:authentication property="principal" var="userDetails" />

<div style="display: inline-flex">
    welcome   <span id="userName">${f:h(userDetails.account.name)}</span>
    <form:form method="post" action="${pageContext.request.contextPath}/logout">
        <input type="submit" id="logout" value="logout" />
    </form:form>
    <form method="get" action="${pageContext.request.contextPath}/account/update">
        <input type="submit" name="form1" id="updateAccount" value="Account Update
↪" />
    </form>
</div>
<br>
<br>

<div>
    <spring:eval var="cart" expression="@cart" />
    <form:form method="post"
        action="${pageContext.request.contextPath}/cart"
        modelAttribute="cartForm">
        <form:errors path="removedItemsIds" cssClass="error-messages" />
        <t:messagesPanel />
        <table>
            <tr>
                <th>Name</th>
                <th>Price</th>
                <th>Quantity</th>
                <th>Remove</th>
            </tr>
            <c:forEach items="${cart.cartItems}" var="cartItem" varStatus="status">
                <tr>
                    <td id="itemName${status.index}">${f:h(cartItem.goods.name)}</
↪td>
                    <td id="itemPrice${status.index}"><fmt:formatNumber value="$
↪{cartItem.goods.price}" type="CURRENCY" currencySymbol="&yen;" maxFractionDigits=
↪"0" /></td>
```

```
                    <td id="itemQuantity${status.index}">${f:h(cartItem.quantity)}
↪</td>
                    <%-- (1) --%>
                    <td><input type="checkbox" name="removedItemsIds" id=
↪"removedItemsIds${status.index}" value="${f:h(cartItem.goods.id)}" /></td>
                </tr>
            </c:forEach>
            <tr>
                <td>Total</td>
                <td id="totalPrice"><fmt:formatNumber value="${f:h(cart.
↪totalAmount)}" type="CURRENCY" currencySymbol="&yen;" maxFractionDigits="0" /></
↪td>
                <td></td>
                <td></td>
            </tr>
        </table>
        <input type="submit" id="remove" value="remove" />
    </form:form>
</div>

<div style="display: inline-flex">
    <form method="get" action="${pageContext.request.contextPath}/order">
        <input type="submit" id="confirm" name="confirm" value="confirm your order
↪" />
    </form>
    <form method="get" action="${pageContext.request.contextPath}/goods">
        <input type="submit" id="home" value="home" />
    </form>
</div>
```

項番	説明
(1)	チェックボックスを利用して、削除する商品を指定する。 チェックボックスが選択された状態で削除ボタンが押されると、該当商品の ID がサーバに送信される。

動作確認

ここまでの実装でカートに登録された商品を削除することができるようになっている。商品一覧表示画面で「viewCart」ボタンを押下することでカート表示画面に遷移する。カート表示画面で削除したい商品をチェックして「remove」ボタンを押すことで、商品をカートから削除できる。

商品注文機能を作成する

カートに登録されている商品を注文する機能を作成する。

注文完了後カートの中身は空になる。

以下に商品注文機能で実装する画面の情報を示す。

処理名	HTTP メソッド	パス	画面
注文確認画面表示処理	GET	/order?confirm	order/confirm
注文処理	POST	/order	注文完了画面表示処理へリダイレクト
注文完了画面表示処理	GET	/order?finish	order/finish

Controller の作成

Controller を作成する。

`/session-tutorial-init-web/src/main/java/com/example/session/app/order/OrderController.java`

```
package com.example.session.app.order;

import javax.inject.Inject;

import org.springframework.http.HttpStatus;
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.message.ResultMessages;

import com.example.session.app.goods.GoodsSearchCriteria;
import com.example.session.domain.model.Cart;
import com.example.session.domain.model.Order;
import com.example.session.domain.service.order.EmptyCartOrderException;
import com.example.session.domain.service.order.InvalidCartOrderException;
import com.example.session.domain.service.order.OrderService;
```

```
import com.example.session.domain.service.userdetails.AccountDetails;

@Controller
@RequestMapping("order")
public class OrderController {

    @Inject
    OrderService orderService;

    // (1)
    @Inject
    Cart cart;

    @Inject
    GoodsSearchCriteria criteria;

    @GetMapping(params = "confirm")
    String confirm(@AuthenticationPrincipal AccountDetails userDetails,
            Model model) {
        if (cart.isEmpty()) {
            ResultMessages messages = ResultMessages.error()
                    .add("e.st.od.5001");
            model.addAttribute(messages);
            return "cart/viewCart";
        }
        model.addAttribute("account", userDetails.getAccount());
        model.addAttribute("signature", cart.calcSignature());
        return "order/confirm";
    }

    @PostMapping
    String order(@AuthenticationPrincipal AccountDetails userDetails,
            @RequestParam String signature, RedirectAttributes attributes) {
        Order order = orderService.purchase(userDetails.getAccount(), cart,
                signature); // (2)
        attributes.addFlashAttribute(order);
        criteria.clear(); // (3)
        return "redirect:/order?finish";
    }

    @GetMapping(params = "finish")
    String finish() {
        return "order/finish";
    }

    // (4)
    @ExceptionHandler({ EmptyCartOrderException.class,
            InvalidCartOrderException.class })
    @ResponseStatus(HttpStatus.CONFLICT)
    ModelAndView handleOrderException(BusinessException e) {
        return new ModelAndView("common/error/businessError").addObject(e
                .getResultMessages());
    }
```

```
}
```

項番	説明
(1)	セッションスコープの Bean を DI コンテナから取得する。
(2)	ドメイン層にある Service のメソッドにて、セッションスコープの Bean の中身を空にしている。 これによりセッションスコープの Bean の破棄が行われたことになる。 また、今回のアプリケーションでは、セッションスコープの Bean にある情報を Bean 破棄後に遷移する画面で使用する。 そのため、セッションスコープの Bean にあった情報を別のオブジェクトに入れなおしてフラッシュスコープに追加している。
(3)	商品検索情報をデフォルト状態に戻している。
(4)	Service のメソッドで Business 例外が発生する可能性があるため、このメソッドでエラーハンドリングを行っている。 これにより、Business 例外が発生した場合、指定したエラー画面に遷移することになる。

警告: セッションスコープの Bean の破棄を行う方法は `@SessionAttributes` で管理させるオブジェクトの破棄方法とは異なる。セッションスコープ Bean の破棄は DI コンテナに任せるべきであり、アプリケーションから破棄すべきでない。そのため、セッションスコープの Bean の破棄を行うには、セッションスコープ Bean のフィールドをリセットするだけで良い。セッションタイムアウト時またはログアウト時に Bean 自体が破棄される。

JSP の作成

注文内容と支払情報を表示する JSP を作成する。

`/session-tutorial-init-web/src/main/webapp/WEB-INF/views/order/confirm.jsp`

```
<sec:authentication property="principal" var="userDetails" />

<div style="display: inline-flex">
    welcome   <span id="userName">${f:h(userDetails.account.name)}</span>
    <form:form method="post"
```

```
        action="${pageContext.request.contextPath}/logout">
        <input type="submit" id="logout" value="logout" />
    </form:form>
    <form method="get"
        action="${pageContext.request.contextPath}/account/update">
        <input type="submit" name="form1" id="updateAccount"
            value="Account Update" />
    </form>
</div>
<br>
<br>

<div>
    <spring:eval var="cart" expression="@cart" />

    <h3>Below items will be ordered. Please push "order" button if it's OK.</h3>
    <table>
        <tr>
            <th>Name</th>
            <th>Price</th>
            <th>Quantity</th>
        </tr>
        <c:forEach items="${cart.cartItems}" var="cartItem" varStatus="status">
            <tr>
                <td id="itemName${status.index}">${f:h(cartItem.goods.name)}</td>
                <td id="itemPrice${status.index}"><fmt:formatNumber value="$
↪{cartItem.goods.price}" type="CURRENCY" currencySymbol="&yen;" maxFractionDigits=
↪"0" /></td>
                <td id="itemQuantity${status.index}">${f:h(cartItem.quantity)}</td>
            </tr>
        </c:forEach>
        <tr>
            <td>Total</td>
            <td id="totalPrice"><fmt:formatNumber value="${f:h(cart.totalAmount)}"␣
↪type="CURRENCY" currencySymbol="&yen;" maxFractionDigits="0" /></td>
            <td></td>
        </tr>
    </table>

    <table>
        <tr>
            <td><label for="name">name</label></td>
            <td id="name">${f:h(account.name)}</td>
        </tr>
        <tr>
            <td><label for="email">e-mail</label></td>
            <td id="email">${f:h(account.email)}</td>
        </tr>
        <tr>
            <td><label for="zip">zip</label></td>
            <td id="zip">${f:h(account.zip)}</td>
        </tr>
        <tr>
```

```
            <td><label for="address">address</label></td>
            <td id="address">${f:h(account.address)}</td>
        </tr>
        <tr>
            <%-- (1) --%>
            <td>payment</td>
            <td id="payment">
                <c:choose>
                    <c:when test="${empty account.cardNumber}">cash</c:when>
                    <c:otherwise>card (card number : ****-****-****-${f:h(account.
↪lastFourOfCardNumber)})</c:otherwise>
                </c:choose>
            </td>
        </tr>
    </table>
</div>
<div style="display: inline-flex">
    <form:form method="post" action="${pageContext.request.contextPath}/order">
        <input type="hidden" name="signature" value="${f:h(signature)}" />
        <input type="submit" id="order" value="order" />
    </form:form>
    <form method="get" action="${pageContext.request.contextPath}/cart">
        <input type="submit" id="back" value="back" />
    </form>
</div>
<div>
    <form method="get" action="${pageContext.request.contextPath}/goods">
        <input type="submit" id="home" value="home" />
    </form>
</div>
```

項番	説明
(1)	アカウント情報としてカード番号が登録されている場合支払方法がカード払いとなる。 登録されていない場合は現金払いとなる。

注文確定後の情報を表示する JSP を作成する。

/session-tutorial-init-web/src/main/webapp/WEB-INF/views/order/finish.jsp

```
<sec:authentication property="principal" var="userDetails" />

<div style="display: inline-flex">
    welcome   <span id="userName">${f:h(userDetails.account.name)}</span>
    <form:form method="post" action="${pageContext.request.contextPath}/logout">
        <input type="submit" id="logout" value="logout" />
    </form:form>
    <form method="get" action="${pageContext.request.contextPath}/account/update">
```

```
        <input type="submit" name="form1" id="updateAccount" value="Account Update
↪" />
    </form>
</div>
<br>
<br>

<div>

    <h3>Your order has been accepted</h3>
    <table>
        <tr>
            <td><label for="orderNumber">order number</label></td>
            <td id="orderNumber">${f:h(order.id)}</td>
        </tr>
        <tr>
            <td><label for="orderDate">order date</label></td>
            <td id="orderDate"><fmt:formatDate value="${order.orderDate}" pattern=
↪"yyyy-MM-dd  hh:mm:ss" /></td>
        </tr>
    </table>
    <table>
        <tr>
            <th>Name</th>
            <th>Price</th>
            <th>Quantity</th>
        </tr>
        <c:forEach items="${order.orderLines}" var="orderLine" varStatus="status">
            <tr>
                <td id="itemName${status.index}">${f:h(orderLine.goods.name)}</td>
                <td id="itemPrice${status.index}"><fmt:formatNumber value="$
↪{orderLine.goods.price}" type="CURRENCY" currencySymbol="&yen;"␣
↪maxFractionDigits="0" /></td>
                <td id="itemQuantity${status.index}">${f:h(orderLine.quantity)}</
↪td>
            </tr>
        </c:forEach>
        <tr>
            <td>Total</td>
            <td id="totalPrice"><fmt:formatNumber value="${f:h(order.totalAmount)}
↪" type="CURRENCY" currencySymbol="&yen;" maxFractionDigits="0" /></td>
            <td></td>
        </tr>
    </table>
</div>
<div>
    <form method="get" action="${pageContext.request.contextPath}/goods">
        <input type="submit" id="home" value="home" />
    </form>
</div>
```

動作確認

ここまでの実装でカートに登録された商品を注文することができるようになっている。カート表示画面で「confirm your order」ボタンを押下することで注文確認画面に遷移する。注文確認画面で「order」ボタンを押下することで、注文が完了する。

ここまでの実装で、注文完了時にセッションにあるカートオブジェクトが削除される。そのため、注文完了後に商品一覧画面に戻るとカートの中身がクリアされている。

セッションの同期化とタイムアウトの設定

最後にセッション同期化とタイムアウトの設定を行う。

セッションの同期化は BeanProcessor を利用して実現する。

`/session-tutorial-init-web/src/main/java/com/example/session/app/config/EnableSynchronizeOnSessionPostProcessor.java`

```
package com.example.session.app.config;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.web.servlet.mvc.method.annotation.
↪RequestMappingHandlerAdapter;

public class EnableSynchronizeOnSessionPostProcessor implements
        BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
            throws BeansException {
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
            throws BeansException {
        if (bean instanceof RequestMappingHandlerAdapter) {
            RequestMappingHandlerAdapter adapter = (RequestMappingHandlerAdapter)␣
↪bean;
            adapter.setSynchronizeOnSession(true); // (1)
        }
        return bean;
    }
}
```

項番	説明
(1)	setSynchronizeOnSession メソッドの引数に true を指定することで、同一セッション内でのリクエストが同期化される。

/session-tutorial-init-web/src/main/resources/META-INF/spring/spring-mvc.xml

```
<!-- Bean Processor -->
<bean class="com.example.session.app.config.EnableSynchronizeOnSessionPostProcessor
↪" />
```

タイムアウト時間は web.xml で設定する。デフォルト値の 30 分を採用する。

/session-tutorial-init-web/src/main/webapp/WEB-INF/web.xml (デフォルトで設定済み)

```
<session-config>
    <!-- 30min -->
    <session-timeout>30</session-timeout>
    <cookie-config>
        <http-only>true</http-only>
        <!-- <secure>true</secure> -->
    </cookie-config>
    <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

タイムアウト後のリクエスト検知は Spring Security の機能を利用する。

/session-tutorial-init-web/src/main/resources/META-INF/spring/spring-security.xml

```
<!-- (1) -->
<sec:session-management invalid-session-url="/loginForm" />
```

項番	説明
(1)	sec:session-management タグの invalid-session-url 属性にタイムアウト後のリクエストを検知した際の遷移先を記述する。

11.3.6 終わりに

本チュートリアルでは以下の内容を学習した。

- セッション管理対象となるデータの設計方法
  - セッションに格納するデータの選択
  - セッションを利用するか否かの判断フローの一例
  - セッション中のデータの破棄
- 本 FW におけるセッションの具体的な利用方法
  - @SessionAttributes を使用する方法

  - セッションスコープの Bean を使用する方法
  - 各利用方法におけるセッション内データの参照方法
  - 各利用方法におけるセッションの破棄方法

11.4 Spring Security チュートリアル

11.4.1 はじめに

このチュートリアルで学ぶこと

- Spring Security による基本的な認証・認可
- データベース上のアカウント情報を使用したログイン
- 認証済みアカウントオブジェクトの取得方法

対象読者

- チュートリアル *(Todo* アプリケーション*)* を実施ずみ (インフラストラクチャ層の実装として MyBatis3 を使用して実施していること)
- Maven の基本的な操作を理解している

検証環境

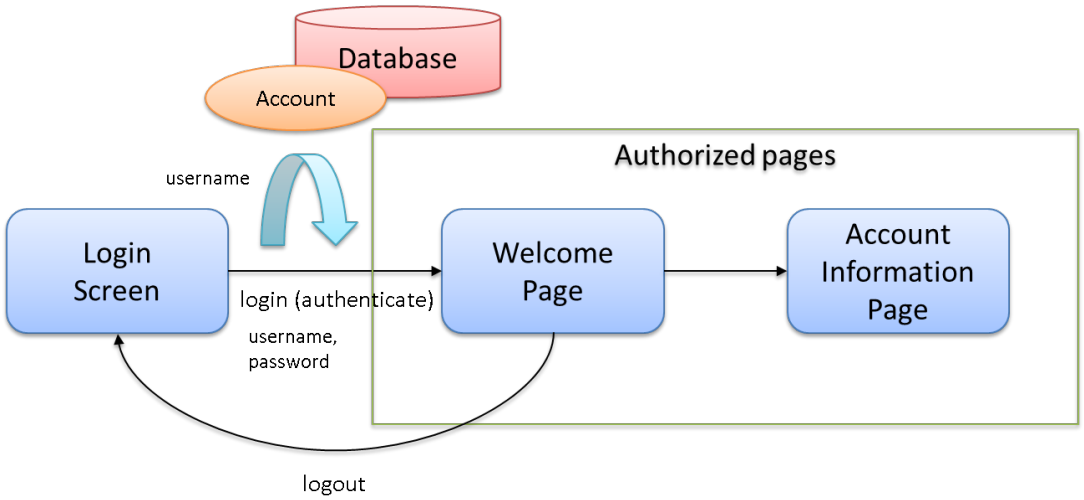
- チュートリアル *(Todo* アプリケーション*)* と同様。

11.4.2 作成するアプリケーションの概要

- ログインページで ID とパスワード指定して、アプリケーションにログインする事ができる。
- ログイン処理で必要となるアカウント情報はデータベース上に格納する。
- ウェルカムページとアカウント情報表示ページがあり、これらのページはログインしないと閲覧する事ができない。
- アプリケーションからログアウトする事ができる。

アプリケーションの概要を以下の図で示す。

URL 一覧を以下に示す。



項番	プロセス名	HTTP メソッド	URL	説明
1	ログインフォーム表示	GET	/login/loginForm	ログインフォームを表示する
2	ログイン	POST	/authentication	ログインフォームから入力されたユーザー名、パスワードを使って認証する (Spring Security が行う)
3	ウェルカムページ表示	GET	/	ウェルカムページを表示する
4	アカウント情報表示	GET	/account	ログインユーザーのアカウント情報を表示する
5	ログアウト	POST	/logout	ログアウトする (Spring Security が行う)

11.4.3 環境構築

プロジェクトの作成

Maven のアーキタイプを利用し、TERASOLUNA Server Framework for Java (5.x) のブランクプロジェクトを作成する。

本チュートリアルでは、MyBatis3 用のブランクプロジェクトを作成する。

なお、Spring Tool Suite(STS) へのインポート方法やアプリケーションサーバの起動方法など基本知識については、チュートリアル (*Todo* アプリケーション) で説明済みのため、本チュートリアルでは説明を割愛する。

```
mvn archetype:generate -B^
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-mybatis3-archetype^
 -DarchetypeVersion=5.4.2.RELEASE^
```

```
-DgroupId=com.example.security^
-DartifactId=first-springsecurity^
-Dversion=1.0.0-SNAPSHOT
```

チュートリアルを進める上で必要となる設定の多くは、作成したブランクプロジェクトに既に設定済みの状態である。チュートリアルを実施するだけであれば、これらの設定の理解は必須ではないが、アプリケーションを動かすためにどのような設定が必要なのかを理解しておくことを推奨する。

アプリケーションを動かすために必要な設定 (設定ファイル) の解説については、「設定ファイルの解説」を参照されたい。

11.4.4 アプリケーションの作成

ドメイン層の実装

Spring Security の認証処理は基本的に以下の流れになる。

1. 入力された `username` からユーザー情報を検索する。
2. ユーザー情報が存在する場合、そのユーザー情報がもつパスワードと入力されたパスワードをハッシュ化したものを比較する。
3. 比較結果が一致する場合、認証成功とみなす。

ユーザー情報が見つからない場合やパスワードの比較結果が一致しない場合は認証失敗である。

ドメイン層ではユーザー名から Account オブジェクトを取得する処理が必要となる。実装は、以下の順に進める。

1. Domain Object(`Account`) の作成
2. `AccountRepository` の作成
3. `AccountSharedService` の作成

Domain Object の作成

認証情報 (ユーザー名とパスワード) を保持する Account クラスを作成する。

src/main/java/com/example/security/domain/model/Account.java

```
package com.example.security.domain.model;

import java.io.Serializable;

public class Account implements Serializable {
    private static final long serialVersionUID = 1L;

    private String username;

    private String password;

    private String firstName;

    private String lastName;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
```

```
    @Override
    public String toString() {
        return "Account [username=" + username + ", password=" + password
                + ", firstName=" + firstName + ", lastName=" + lastName + "]";
    }
}
```

AccountRepository の作成

Account オブジェクトをデータベースから取得する処理を実装する。

AccountRepository インタフェースを作成する。

src/main/java/com/example/security/domain/repository/account/AccountRepository.java

```
package com.example.security.domain.repository.account;

import com.example.security.domain.model.Account;

public interface AccountRepository {
    Account findOne(String username);
}
```

Account を 1 件取得するための SQL を Mapper ファイルに定義する。

src/main/resources/com/example/security/domain/repository/account/AccountRepository.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.security.domain.repository.account.AccountRepository
↪">
```

```
    <resultMap id="accountResultMap" type="Account">
        <id property="username" column="username" />
        <result property="password" column="password" />
        <result property="firstName" column="first_name" />
        <result property="lastName" column="last_name" />
    </resultMap>

    <select id="findOne" parameterType="String" resultMap="accountResultMap">
        SELECT
            username,
            password,
            first_name,
            last_name
        FROM
            account
        WHERE
            username = #{username}
    </select>
</mapper>
```

AccountSharedService の作成

ユーザー名から Account オブジェクトを取得する業務処理を実装する。

この処理は、Spring Security の認証サービスから利用するため、インタフェース名は AccountSharedService、クラス名は AccountSharedServiceImpl とする。

注釈: 本ガイドラインでは、Service から別の Service を呼び出す事を推奨していない。

ドメイン層の処理 (Service) を共通化したい場合は、XxxService という名前ではなく、Service の処理を共通化するための Service であることを示すために、XxxSharedService という名前にすることを推奨している。

本チュートリアルで作成するアプリケーションでは共通化は必須ではないが、通常のアプリケーションであればアカウント情報を管理する業務の Service と処理を共通化することが想定される。そのため、本チュートリアルではアカウント情報の取得処理を SharedService として実装する。

AccountSharedService インタフェースを作成する。

src/main/java/com/example/security/domain/service/account/AccountSharedService.java

```
package com.example.security.domain.service.account;

import com.example.security.domain.model.Account;

public interface AccountSharedService {
    Account findOne(String username);
}
```

AccountSharedServiceImpl クラスを作成する。

src/main/java/com/example/security/domain/service/account/AccountSharedServiceImpl.java

```
package com.example.security.domain.service.account;

import javax.inject.Inject;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;

import com.example.security.domain.model.Account;
import com.example.security.domain.repository.account.AccountRepository;

@Service
public class AccountSharedServiceImpl implements AccountSharedService {
    @Inject
    AccountRepository accountRepository;

    @Transactional(readOnly=true)
    @Override
    public Account findOne(String username) {
        // (1)
        Account account = accountRepository.findOne(username);
        // (2)
        if (account == null) {
            throw new ResourceNotFoundException("The given account is not found!␣
↪username="
                    + username);
        }
        return account;
```

```
    }

}
```

項番	説明
(1)	ユーザー名に一致する Account オブジェクトを 1 件取得する。
(2)	ユーザー名に一致する Account が存在しない場合は、共通ライブラリから提供している ResourceNotFoundException をスローする。

認証サービスの作成

Spring Security で使用する認証ユーザー情報を保持するクラスを作成する。

src/main/java/com/example/security/domain/service/userdetails/SampleUserDetails.java

```
package com.example.security.domain.service.userdetails;

import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.userdetails.User;

import com.example.security.domain.model.Account;

public class SampleUserDetails extends User { // (1)
    private static final long serialVersionUID = 1L;

    private final Account account; // (2)

    public SampleUserDetails(Account account) {
        // (3)
        super(account.getUsername(), account.getPassword(), AuthorityUtils
                .createAuthorityList("ROLE_USER")); // (4)
        this.account = account;
    }

    public Account getAccount() { // (5)
        return account;
```

```
    }

}
```

項番	説明
(1)	org.springframework.security.core.userdetails.UserDetails インタフェースを実装する。 ここでは UserDetails を実装した org.springframework.security.core.userdetails.User クラスを継承し、本プロジェクト用の UserDetails クラスを実装する。
(2)	Spring の認証ユーザークラスに、本プロジェクトのアカウント情報を保持させる。
(3)	User クラスのコンストラクタを呼び出す。第 1 引数はユーザー名、第 2 引数はパスワード、第 3 引数は権限リストである。
(4)	簡易実装として、ROLE_USER というロールのみ持つ権限を作成する。
(5)	アカウント情報の getter を用意する。これにより、ログインユーザーの Account オブジェクトを取得することができる。

Spring Security で使用する認証ユーザー情報を取得するサービスを作成する。

src/main/java/com/example/security/domain/service/userdetails/SampleUserDetailsService.java

```
package com.example.security.domain.service.userdetails;

import javax.inject.Inject;
```

```
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;

import com.example.security.domain.model.Account;
import com.example.security.domain.service.account.AccountSharedService;

@Service
public class SampleUserDetailsService implements UserDetailsService { // (1)
    @Inject
    AccountSharedService accountSharedService; // (2)

    @Transactional(readOnly=true)
    @Override
    public UserDetails loadUserByUsername(String username) throws␣
↪UsernameNotFoundException {
        try {
            Account account = accountSharedService.findOne(username); // (3)
            return new SampleUserDetails(account); // (4)
        } catch (ResourceNotFoundException e) {
            throw new UsernameNotFoundException("user not found", e); // (5)
        }
    }

}
```

項番	説明
(1)	`org.springframework.security.core.userdetails.UserDetailsService` インタフェースを実装する。
(2)	`AccountSharedService` をインジェクションする。
(3)	`username` から `Account` オブジェクトを取得する処理を `AccountSharedService` に委譲する。
(4)	取得した `Account` オブジェクトを使用して、本プロジェクト用の `UserDetails` オブジェクトを作成し、メソッドの返り値として返却する。
(5)	対象のユーザーが見つからない場合は、`UsernameNotFoundException` がスローする。

データベースの初期化スクリプトの設定

本チュートリアルでは、アカウント情報を保持するデータベースとして H2 Database(インメモリデータベース) を使用する。そのため、アプリケーション起動時に SQL を実行してデータベースを初期化する必要がある。

ブランクプロジェクトには以下のように `jdbc:initialize-database` が設定済みであり、`${database}-schema.sql` に DDL 文、`${database}-dataload.sql` に DML 文を追加するだけでアプリケーション起動時に SQL を実行してデータベースを初期化することができる。なお、ブランクプロジェクトの設定では `first-springsecurity-infra.properties` に `database=H2` と定義されているため、`H2-schema.sql` 及び `H2-dataload.sql` が実行される。

`src/main/resources/META-INF/spring/first-springsecurity-env.xml`

```
<jdbc:initialize-database data-source="dataSource"
    ignore-failures="ALL">
    <jdbc:script location="classpath:/database/${database}-schema.sql" encoding=
↪"UTF-8" />
    <jdbc:script location="classpath:/database/${database}-dataload.sql" encoding=
↪"UTF-8" />
</jdbc:initialize-database>
```

アカウント情報を保持するテーブルを作成するための DDL 文を作成する。

src/main/resources/database/H2-schema.sql

```
CREATE TABLE account(
    username varchar(128),
    password varchar(60),
    first_name varchar(128),
    last_name varchar(128),
    constraint pk_tbl_account primary key (username)
);
```

デモユーザー (username=demo、password=demo) を登録するための DML 文を作成する。

src/main/resources/database/H2-dataload.sql

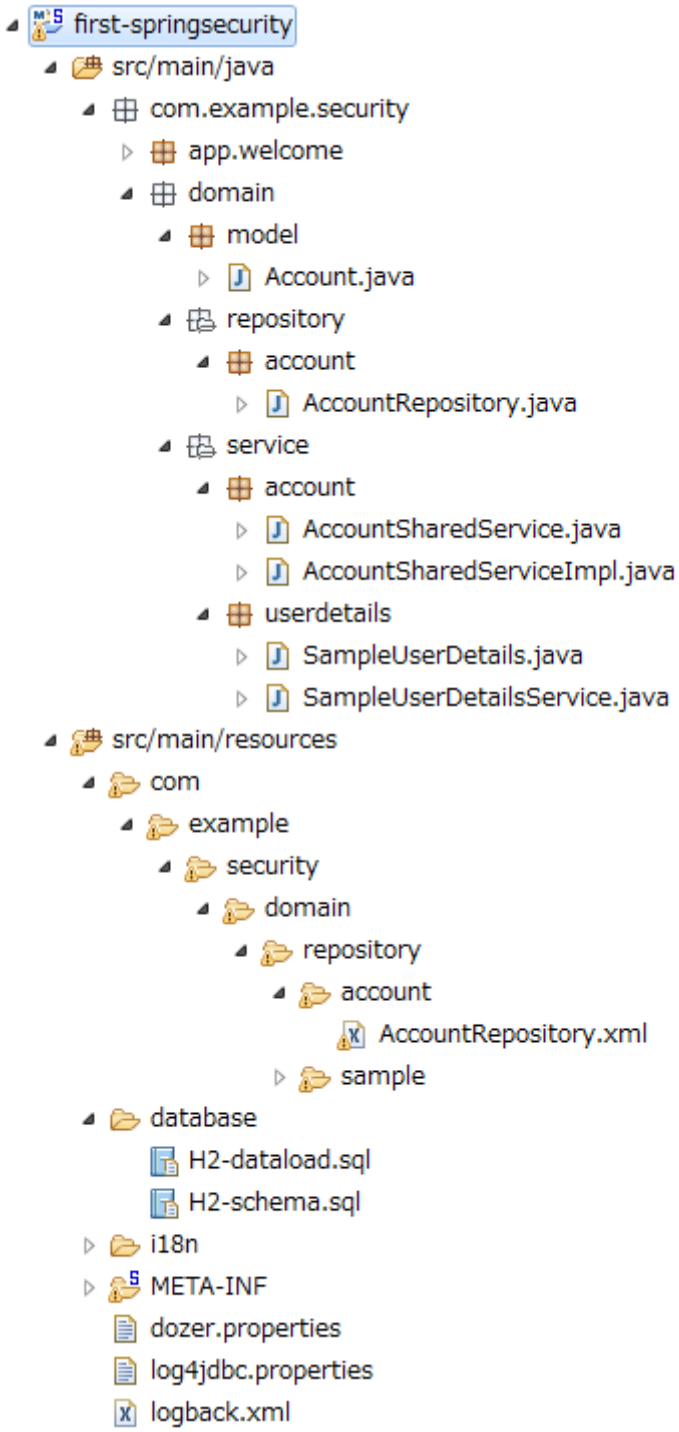
```
INSERT INTO account(username, password, first_name, last_name) VALUES('demo', '$2a
↪$10$oxSJl.keBwxmsMLkcT9lPeAIxfNTPNQxpeywMrF7A3kVszwUTqfTK', 'Taro', 'Yamada'); --
 (1)
COMMIT;
```

項番	説明
(1)	ブランクプロジェクトの設定では、applicationContext.xml にパスワードをハッシュ化するためのクラスとして org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder が設定されている。 本チュートリアルでは、BCryptPasswordEncoder を使用してパスワードのハッシュ化を行うため、パスワードには demo という文字列を BCrypt アルゴリズムでハッシュ化した文字列を投入する。

ドメイン層の作成後のパッケージエクスプローラー

ドメイン層に作成したファイルを確認する。

Package Explorer の Package Presentation は Hierarchical を使用している。



アプリケーション層の実装

Spring Security の設定

`spring-security.xml` に Spring Security による認証・認可の設定を行う。

本チュートリアルで作成するアプリケーションで扱う URL のパターンを以下に示す。

URL	説明
/login/loginForm	ログインフォームを表示するための URL
/login/loginForm?error=true	認証エラー時に遷移するページ (ログインページ) を表示するための URL
/login	認証処理を行うための URL
/logout	ログアウト処理を行うための URL
/	ウェルカムページを表示するための URL
/account	ログインユーザーのアカウント情報を表示するための URL

ブランクプロジェクトから提供されている設定に加えて、以下の設定を追加する。

`src/main/resources/META-INF/spring/spring-security.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://www.springframework.org/schema/security"
    xsi:schemaLocation="
        http://www.springframework.org/schema/security https://www.springframework.
↪org/schema/security/spring-security.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
    ">

    <sec:http pattern="/resources/**" security="none"/>
    <sec:http>
        <!-- (1) -->
        <sec:form-login
            login-page="/login/loginForm"
            authentication-failure-url="/login/loginForm?error=true" />
        <!-- (2) -->
        <sec:logout
            logout-success-url="/"
            delete-cookies="JSESSIONID" />
        <sec:access-denied-handler ref="accessDeniedHandler"/>
        <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
        <sec:session-management />
        <!-- (3) -->
        <sec:intercept-url pattern="/login/**" access="permitAll" />
        <sec:intercept-url pattern="/**" access="isAuthenticated()" />
    </sec:http>

    <sec:authentication-manager>
        <!-- com.example.security.domain.service.userdetails.
↪SampleUserDetailsService
          is scanned by component scan with @Service -->
        <!-- (4) -->
        <sec:authentication-provider
            user-service-ref="sampleUserDetailsService">
            <!-- (5) -->
            <sec:password-encoder ref="passwordEncoder" />
        </sec:authentication-provider>
    </sec:authentication-manager>

    <!-- CSRF Protection -->
    <bean id="accessDeniedHandler"
        class="org.springframework.security.web.access.
↪DelegatingAccessDeniedHandler">
        <constructor-arg index="0">
            <map>
                <entry
                    key="org.springframework.security.web.csrf.
↪InvalidCsrfTokenException">
                    <bean
                        class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
```

```
                        <property name="errorPage"
                            value="/WEB-INF/views/common/error/
↪invalidCsrfTokenError.jsp" />
                    </bean>
                </entry>
                <entry
                    key="org.springframework.security.web.csrf.
↪MissingCsrfTokenException">
                    <bean
                        class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                        <property name="errorPage"
                            value="/WEB-INF/views/common/error/
↪missingCsrfTokenError.jsp" />
                    </bean>
                </entry>
            </map>
        </constructor-arg>
        <constructor-arg index="1">
            <bean
                class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                <property name="errorPage"
                    value="/WEB-INF/views/common/error/accessDeniedError.jsp" />
            </bean>
        </constructor-arg>
    </bean>

    <!-- Put UserID into MDC -->
    <bean id="userIdMDCPutFilter" class="org.terasoluna.gfw.security.web.logging.
↪UserIdMDCPutFilter">
    </bean>

</beans>
```

項番	説明
(1)	<sec:form-login>タグでログインフォームに関する設定を行う。 <sec:form-login>タグには、 • login-page 属性にログインフォームを表示するための URL • authentication-failure-url 属性に認証エラー時に遷移するページを表示するための URL を設定する。
(2)	<sec:logout>タグでログアウトに関する設定を行う。 <sec:logout>タグには、 • logout-success-url 属性にログアウト後に遷移するページを表示するための URL(本チュートリアルではウェルカムページを表示するための URL) • delete-cookies 属性にログアウト時に削除する Cookie 名 (本チュートリアルではセッション ID の Cookie 名) を設定する。
(3)	<sec:intercept-url>タグを使用して URL 毎の認可設定を行う。 <sec:intercept-url>タグには、 • ログインフォームを表示するための URL には、全てのユーザーのアクセスを許可する permitAll • 上記以外の URL には、認証済みユーザーのみアクセスを許可する isAuthenticated() を設定する。 ただし、/resources/配下の URL については、Spring Security による認証・認可処理を行わない設定 (<sec:http pattern="/resources/**" security="none"/>) が行われているため、全てのユーザーがアクセスすることができる。
(4)	<sec:authentication-provider>タグを使用して、認証処理を行う org.springframework.security.authentication.AuthenticationProvider の設定を行う。 デフォルトでは、UserDetailsService を使用して UserDetails を取得し、その UserDetails が持つハッシュ化済みパスワードと、ログインフォームで指定されたパスワードを比較してユーザー認証を行うクラス (org.springframework.security.authentication.dao.DaoAuthenticationProvider) が使用される。 user-service-ref 属性に UserDetailsService インタフェースを実装しているコンポーネントの bean 名を指定する。本チュートリアルでは、ドメイン層に作成した SampleUserDetailsService クラスを設定する。
(5)	<sec:password-encoder>タグを使用して、ログインフォームで指定されたパスワードをハッシュ化するためのクラス (PasswordEncoder) の設定を行う。 本チュートリアルでは、applicationContext.xml に定義されている org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder を利用する。

ログインページを返す Controller の作成

ログインページを返す Controller を作成する。

src/main/java/com/example/security/app/login/LoginController.java

```
package com.example.security.app.login;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/login")
public class LoginController {

    @GetMapping("/loginForm") // (1)
    public String view() {
        return "login/loginForm";
    }
}
```

項番	説明
(1)	ログインページである、login/loginForm を返す。

ログインページの作成

ログインページにログインフォームを作成する。

src/main/webapp/WEB-INF/views/login/loginForm.jsp

```
<!DOCTYPE html>
<html>
<head>
<title>Login Page</title>
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/css/
↪styles.css">
</head>
<body>
    <div id="wrapper">
        <h3>Login with Username and Password</h3>
```

```
        <!-- (1) -->
        <c:if test="${param.containsKey('error')}">
            <!-- (2) -->
            <t:messagesPanel messagesType="error"
                messagesAttributeName="SPRING_SECURITY_LAST_EXCEPTION" />
        </c:if>

        <!-- (3) -->
        <form:form action="${pageContext.request.contextPath}/login">
            <table>
                <tr>
                    <td><label for="username">User:</label></td>
                    <td><input type="text" id="username"
                        name="username" value="demo">(demo)</td><!-- (4) -->
                </tr>
                <tr>
                    <td><label for="password">Password:</label></td>
                    <td><input type="password" id="password"
                        name="password" value="demo" />(demo)</td><!-- (5) -->
                </tr>
                <tr>
                    <td> </td>
                    <td><input name="submit" type="submit" value="Login" /></td>
                </tr>
            </table>
        </form:form>
    </div>
</body>
</html>
```

項番	説明
(1)	認証が失敗した場合、/login/loginForm?error=true が呼び出され、ログインページを表示する。そのため、認証エラー後の表示の時のみエラーメッセージが表示されるように<c:if>タグを使用する。
(2)	共通ライブラリから提供されている<t:messagesPanel>タグを使用してエラーメッセージを表示する。 認証が失敗した場合、Spring Security のデフォルトの設定で使用される、org.springframework.security.web.authentication.SimpleUrlAuthenticationFailureHandler では、認証エラー時に発生した例外オブジェクトを SPRING_SECURITY_LAST_EXCEPTION という属性名で、リダイレクト時はセッション、フォワード時はリクエストスコープに格納する。 ここでは、認証エラー時にはリダイレクトするため、認証エラー時に発生した例外オブジェクトは、セッションスコープに格納される。
(3)	<form:form>タグの action 属性に、認証処理用の URL(/login) を設定する。この URL は Spring Security のデフォルトである。 認証処理に必要なパラメータ (ユーザー名とパスワード) を POST メソッドで送信する。
(4)	ユーザー名を指定するテキストボックスを作成する。 Spring Security のデフォルトのパラメータ名は username である。
(5)	パスワードを指定するテキストボックス (パスワード用のテキストボックス) を作成する。 Spring Security のデフォルトのパラメータ名は password である。

セッションスコープに格納される認証エラーの例外オブジェクトを JSP から取得できるようにする。

src/main/webapp/WEB-INF/views/common/include.jsp

```
<%@ page session="true"%> <!-- (6) -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec"%>
<%@ taglib uri="http://terasoluna.org/tags" prefix="t"%>
<%@ taglib uri="http://terasoluna.org/functions" prefix="f"%>
```

項番	説明
(6)	page ディレクティブの session 属性を true にする。

注釈: ブランクプロジェクトのデフォルト設定では、JSP からセッションスコープにアクセスできないようになっている。これは、安易にセッションが使用されないようにするためであるが、認証エラーの例外オブジェクトを JSP から取得する場合は、JSP からセッションスコープにアクセスできるようにする必要がある。

ブラウザのアドレスバーに http://localhost:8080/first-springsecurity/ を入力し、ウェルカムページを表示しようとする。

未ログイン状態のため、<sec:form-login>タグの login-page 属性の設定値 (http://localhost:8080/first-springsecurity/login/loginForm) に遷移し、以下のような画面が表示される。



JSP からログインユーザーのアカウント情報へアクセス

JSP からログインユーザーのアカウント情報にアクセスし、氏名を表示する。

src/main/webapp/WEB-INF/views/welcome/home.jsp

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Home</title>
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/css/
↪styles.css">
```

```
</head>

<!-- (1) -->
<sec:authentication property="principal.account" var="account" />

<body>
    <div id="wrapper">
        <h1>Hello world!</h1>
        <p>The time on the server is ${serverTime}.</p>
        <!-- (2) -->
        <p>Welcome ${f:h(account.firstName)} ${f:h(account.lastName)} !!</p>
        <ul>
            <li><a href="${pageContext.request.contextPath}/account">view account</
↪a></li>
        </ul>
    </div>
</body>
</html>
```

項番	説明
(1)	<sec:authentication>タグを使用して、ログインユーザーの org.springframework.security.core.Authentication オブジェクトにアクセスする。 property 属性を使用すると Authentication オブジェクトが保持する任意のプロパティにアクセスする事ができ、アクセスしたプロパティ値は var 属性を使用して任意のスコープに格納することできる。デフォルトでは page スコープが設定され、この JSP 内のみで参照可能となる。 チュートリアルでは、ログインユーザーの Account オブジェクトを account という属性名で page スコープに格納する。
(2)	ログインユーザーの Account オブジェクトにアクセスして、firstName と lastName を表示する。

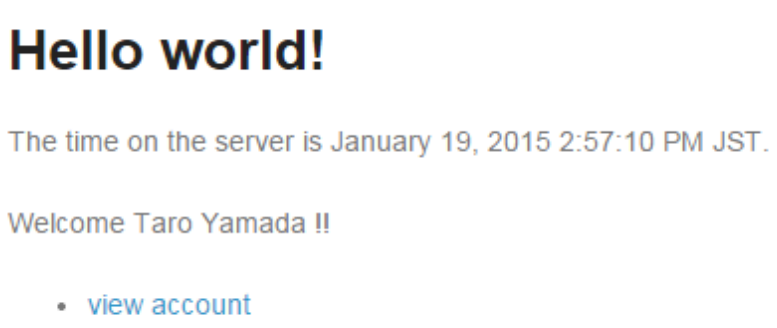
ログインページの Login ボタンを押下し、ウェルカムページを表示する。

ログアウトボタンの追加

ログアウトするためのボタンを追加する。

src/main/webapp/WEB-INF/views/welcome/home.jsp

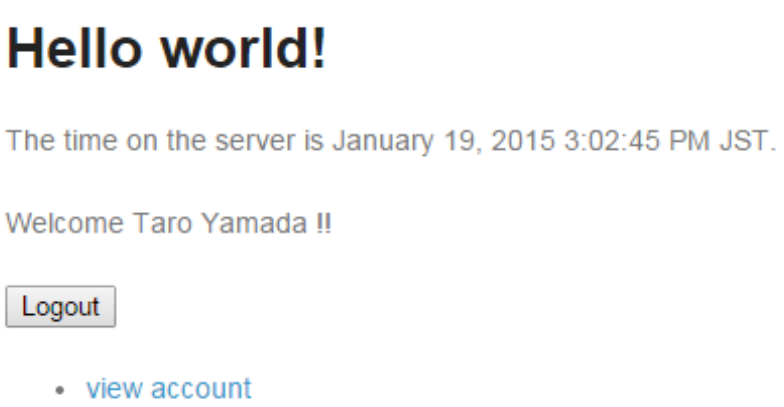
```
<!DOCTYPE html>
<html>
<head>
```




```
<meta charset="utf-8">
<title>Home</title>
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/css/
↪styles.css">
</head>

<sec:authentication property="principal.account" var="account" />

<body>
    <div id="wrapper">
        <h1>Hello world!</h1>
        <p>The time on the server is ${serverTime}.</p>
        <p>Welcome ${f:h(account.firstName)} ${f:h(account.lastName)} !!</p>
        <p>
            <!-- (1) -->
            <form:form action="${pageContext.request.contextPath}/logout">
                <button type="submit">Logout</button>
            </form:form>
        </p>
        <ul>
            <li><a href="${pageContext.request.contextPath}/account">view account</
↪a></li>
        </ul>
    </div>
</body>
</html>
```

項番	説明
(1)	<form:form>タグを使用して、ログアウト用のフォームを追加する。 action 属性には、ログアウト処理用の URL(/logout) を指定して、Logout ボタンを追加する。 この URL は Spring Security のデフォルトである。

ウェルカムページに Logout ボタンが表示される。



ウェルカムページで Logout ボタンを押下すると、アプリケーションからログアウトする (ログインページが表示される)。



Controller からログインユーザーのアカウント情報へアクセス

Controller からログインユーザーのアカウント情報にアクセスし、アカウント情報を View に引き渡す。

`src/main/java/com/example/security/app/account/AccountController.java`

```
package com.example.security.app.account;

import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import com.example.security.domain.model.Account;
import com.example.security.domain.service.userdetails.SampleUserDetails;

@Controller
@RequestMapping("account")
public class AccountController {

    @GetMapping
```

```
    public String view(
            @AuthenticationPrincipal SampleUserDetails userDetails, // (1)
            Model model) {
        // (2)
        Account account = userDetails.getAccount();
        model.addAttribute(account);
        return "account/view";
    }
}
```

項番	説明
(1)	@AuthenticationPrincipal アノテーションを指定して、ログインユーザーの UserDetails オブジェクトを受け取る。
(2)	SampleUserDetails オブジェクトが保持している Account オブジェクトを取得し、View に引き渡すために Model に格納する。

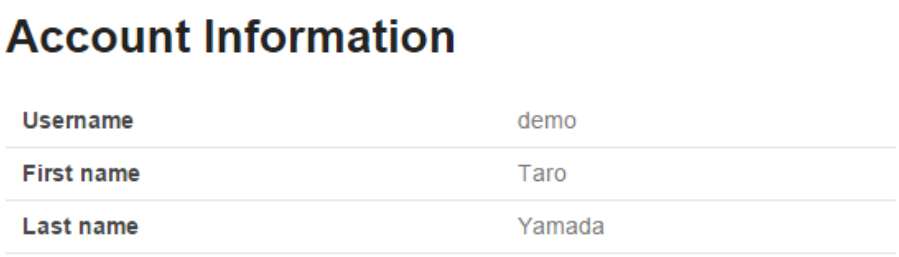
Controller から引き渡されたアカウント情報にアクセスし、アカウント情報を表示する。

src/main/webapp/WEB-INF/views/account/view.jsp

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Home</title>
<link rel="stylesheet" href="${pageContext.request.contextPath}/resources/app/css/
↪styles.css">
</head>
<body>
    <div id="wrapper">
        <h1>Account Information</h1>
        <table>
            <tr>
                <th>Username</th>
                <td>${f:h(account.username)}</td>
            </tr>
            <tr>
```

```
                <th>First name</th>
                <td>${f:h(account.firstName)}</td>
            </tr>
            <tr>
                <th>Last name</th>
                <td>${f:h(account.lastName)}</td>
            </tr>
        </table>
    </div>
</body>
</html>
```

ウェルカムページの view account リンクを押下して、ログインユーザーのアカウント情報表示ページを表示する。



アプリケーション層の作成後のパッケージエクスプローラー

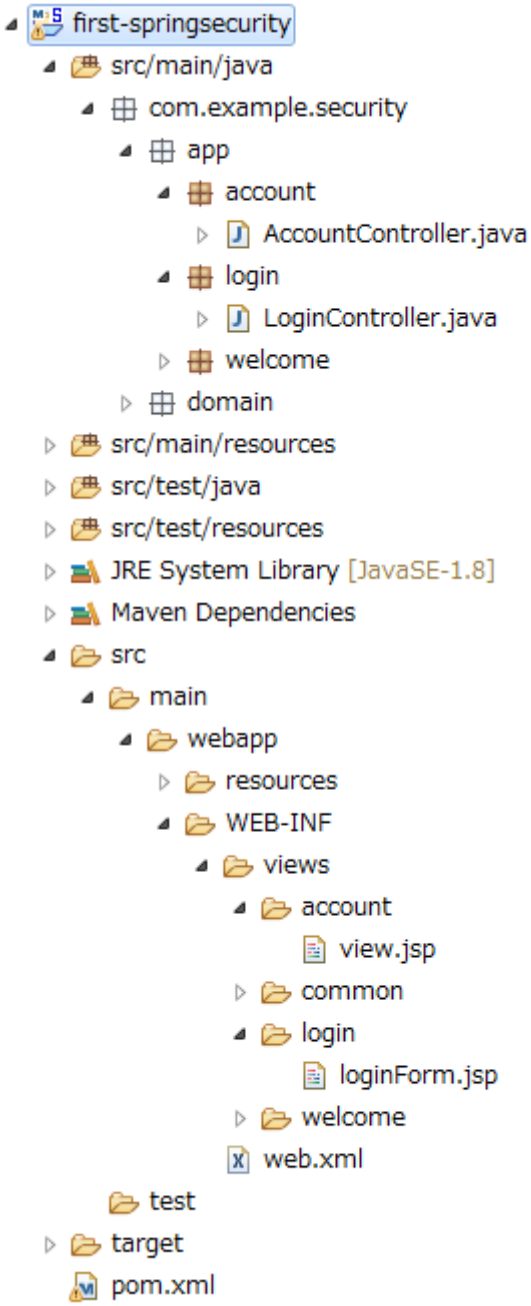
アプリケーション層に作成したファイルを確認する。

Package Explorer の Package Presentation は Hierarchical を使用している。

11.4.5 おわりに

本チュートリアルでは以下の内容を学習した。

- Spring Security による基本的な認証・認可
- 認証ユーザーオブジェクトのカスタマイズ方法
- Repository および Service クラスを用いた認証処理の設定



- JSP でログイン済みアカウント情報にアクセスする方法
- Controller でログイン済みアカウント情報にアクセスする方法

11.4.6 Appendix

設定ファイルの解説

Spring Security を利用するためにどのような設定が必要なのかを理解するために、設定ファイルの解説を行う。

spring-security.xml

spring-security.xml には、Spring Security に関する定義を行う。

作成したブランクプロジェクトの src/main/resources/META-INF/spring/spring-security.xml は、以下のような設定となっている。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://www.springframework.org/schema/security"
    xsi:schemaLocation="
        http://www.springframework.org/schema/security https://www.springframework.
↪org/schema/security/spring-security.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
    ">

    <!-- (1) -->
    <sec:http pattern="/resources/**" security="none"/>
    <sec:http>
        <!-- (2) -->
        <sec:form-login/>
        <!-- (3) -->
        <sec:logout/>
        <!-- (4) -->
        <sec:access-denied-handler ref="accessDeniedHandler"/>
        <!-- (5) -->
        <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
        <!-- (6) -->
        <sec:session-management />
    </sec:http>

    <!-- (7) -->
    <sec:authentication-manager />

    <!-- (4) -->
    <!-- CSRF Protection -->
    <bean id="accessDeniedHandler"
        class="org.springframework.security.web.access.
↪DelegatingAccessDeniedHandler">
        <constructor-arg index="0">
            <map>
                <entry
                    key="org.springframework.security.web.csrf.
↪InvalidCsrfTokenException">
                    <bean
                        class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                        <property name="errorPage"
                            value="/WEB-INF/views/common/error/
↪invalidCsrfTokenError.jsp" />
                    </bean>
```

```
                </entry>
                <entry
                    key="org.springframework.security.web.csrf.
↪MissingCsrfTokenException">
                    <bean
                        class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                        <property name="errorPage"
                            value="/WEB-INF/views/common/error/
↪missingCsrfTokenError.jsp" />
                    </bean>
                </entry>
            </map>
        </constructor-arg>
        <constructor-arg index="1">
            <bean
                class="org.springframework.security.web.access.
↪AccessDeniedHandlerImpl">
                <property name="errorPage"
                    value="/WEB-INF/views/common/error/accessDeniedError.jsp" />
            </bean>
        </constructor-arg>
    </bean>

    <!-- (5) -->
    <!-- Put UserID into MDC -->
    <bean id="userIdMDCPutFilter" class="org.terasoluna.gfw.security.web.logging.
↪UserIdMDCPutFilter">
    </bean>

</beans>
```

項番	説明
(1)	<sec:http>タグを使用して HTTP アクセスに対して認証・認可を制御する。 ブランクプロジェクトのデフォルトの設定では、静的リソース (js, css, image ファイルなど) にアクセスするための URL を認証・認可の対象外にしている。
(2)	<sec:form-login>タグを使用して、フォーム認証を使用したログインに関する動作を制御する。使用方法については、「フォーム認証」 を参照されたい。
(3)	<sec:logout>タグ を使用して、ログアウトに関する動作を制御する。使用方法については、「ログアウト」 を参照されたい。
(4)	<sec:access-denied-handler>タグを使用して、アクセスを拒否した後の動作を制御する。 ブランクプロジェクトのデフォルトの設定では、 • 不正な CSRF トークンを検知した場合 (InvalidCsrfTokenException が発生した場合) の遷移先 • トークンストアから CSRF トークンが取得できない場合 (MissingCsrfTokenException が発生した場合) の遷移先 • 認可処理でアクセスが拒否された場合 (上記以外の AccessDeniedException が発生した場合) の遷移先 が設定済みである。
(5)	Spring Security の認証ユーザ名をロガーの MDC に格納するためのサーブレットフィルタを有効化する。この設定を有効化すると、ログに認証ユーザ名が出力されるため、トレーサビリティを向上することができる。
(6)	<sec:session-management>タグを使用して、Spring Security のセッション管理方法を制御する。 使用方法については、「セッション管理機能の適用」を参照されたい。
(7)	<sec:authentication-manager>タグを使用して、認証処理を制御する。 使用方法については、「*DB* 認証の適用」を参照されたい。

spring-mvc.xml

spring-mvc.xml には、Spring Security と Spring MVC を連携するための設定を行う。

作成したブランクプロジェクトの src/main/resources/META-INF/spring/spring-mvc.xml は、以下のような設定となっている。Spring Security と関係のない設定については、説明を割愛する。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
```

```
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc https://www.
↪springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans https://www.springframework.
↪org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util https://www.springframework.org/
↪schema/util/spring-util.xsd
        http://www.springframework.org/schema/context https://www.springframework.
↪org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop https://www.springframework.org/
↪schema/aop/spring-aop.xsd
    ">

    <context:property-placeholder
        location="classpath*:/META-INF/spring/*.properties" />

    <mvc:annotation-driven>
        <mvc:argument-resolvers>
            <bean
                class="org.springframework.data.web.
↪PageableHandlerMethodArgumentResolver" />
            <!-- (1) -->
            <bean
                class="org.springframework.security.web.method.annotation.
↪AuthenticationPrincipalArgumentResolver" />
        </mvc:argument-resolvers>
    </mvc:annotation-driven>

    <mvc:default-servlet-handler />

    <context:component-scan base-package="com.example.security.app" />

    <mvc:resources mapping="/resources/**"
        location="/resources/,classpath:META-INF/resources/"
        cache-period="#{60 * 60}" />

    <mvc:interceptors>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
                class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor" />
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
                class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenInterceptor" />
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
```

```
            <mvc:exclude-mapping path="/resources/**" />
            <bean class="org.terasoluna.gfw.web.codelist.CodeListInterceptor">
                <property name="codeListIdPattern" value="CL_.+" />
            </bean>
        </mvc:interceptor>
        <!--  REMOVE THIS LINE IF YOU USE JPA
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
                class="org.springframework.orm.jpa.support.
↪OpenEntityManagerInViewInterceptor" />
        </mvc:interceptor>
            REMOVE THIS LINE IF YOU USE JPA  -->
    </mvc:interceptors>

    <!-- Settings View Resolver. -->
    <mvc:view-resolvers>
        <mvc:jsp prefix="/WEB-INF/views/" />
    </mvc:view-resolvers>

    <bean id="requestDataValueProcessor"
        class="org.terasoluna.gfw.web.mvc.support.
↪CompositeRequestDataValueProcessor">
        <constructor-arg>
            <util:list>
                <!-- (2) -->
                <bean
                    class="org.springframework.security.web.servlet.support.csrf.
↪CsrfRequestDataValueProcessor" />
                <bean
                    class="org.terasoluna.gfw.web.token.transaction.
↪TransactionTokenRequestDataValueProcessor" />
            </util:list>
        </constructor-arg>
    </bean>

    <!-- Setting Exception Handling. -->
    <!-- Exception Resolver. -->
    <bean id="systemExceptionResolver"
        class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">
        <property name="exceptionCodeResolver" ref="exceptionCodeResolver" />
        <!-- Setting and Customization by project. -->
        <property name="order" value="3" />
        <property name="exceptionMappings">
            <map>
                <entry key="ResourceNotFoundException" value="common/error/
↪resourceNotFoundError" />
                <entry key="BusinessException" value="common/error/businessError" /
↪>
                <entry key="InvalidTransactionTokenException" value="common/error/
↪transactionTokenError" />
                <entry key=".DataAccessException" value="common/error/
↪dataAccessError" />
```

```
            </map>
        </property>
        <property name="statusCodes">
            <map>
                <entry key="common/error/resourceNotFoundError" value="404" />
                <entry key="common/error/businessError" value="409" />
                <entry key="common/error/transactionTokenError" value="409" />
                <entry key="common/error/dataAccessError" value="500" />
            </map>
        </property>
        <property name="excludedExceptions">
            <array>
                <value>org.springframework.web.util.NestedServletException</value>
            </array>
        </property>
        <property name="defaultErrorView" value="common/error/systemError" />
        <property name="defaultStatusCode" value="500" />
    </bean>
    <!-- Setting AOP. -->
    <bean id="handlerExceptionResolverLoggingInterceptor"
        class="org.terasoluna.gfw.web.exception.
↪HandlerExceptionResolverLoggingInterceptor">
        <property name="exceptionLogger" ref="exceptionLogger" />
    </bean>
    <aop:config>
        <aop:advisor advice-ref="handlerExceptionResolverLoggingInterceptor"
            pointcut="execution(* org.springframework.web.servlet.
↪HandlerExceptionResolver.resolveException(..))" />
    </aop:config>

</beans>
```

項番	説明
(1)	`@AuthenticationPrincipal` アノテーションを指定して、ログインユーザーの `UserDetails` オブジェクトを Controller の引数として受け取れるようにするための設定。 `<mvc:argument-resolvers>`タグに `AuthenticationPrincipalArgumentResolver` を指定する。
(2)	`<form:form>`タグ (JSP タグライブラリ) を使用して、CSRF トークン値を HTML フォームに埋め込むための設定。 `CompositeRequestDataValueProcessor` のコンストラクタに `CsrfRequestDataValueProcessor` を指定する。

第 12 章

Appendix(Know How)

12.1 NEXUS による Maven リポジトリの管理

Sonatype NEXUS はパッケージリポジトリマネージャソフトウェアである。OSS 版と商用版があるが、OSS 版でも十分な機能がある。

本章では OSS 版の NEXUS の役割と設定方法などについて解決する。

12.1.1 Why NEXUS ?

開発者が一人しかいない場合には、インターネット上のセントラルリポジトリと、その開発者の PC 内のローカルリポジトリだけでも、maven や ant+ivy を使って開発することは可能である。

しかし、Java アプリケーションを複数のサブプロジェクトに分けてチームで開発する場合にはライブラリの依存性解決が複雑になるため、ライブラリの依存性解決の自動化が必要となる。そのためにはパッケージリポジトリサーバの存在が不可欠である。

Java アプリケーション開発プロジェクトにおいて必要となるパッケージリポジトリは次のようなものがある。

- セントラルリポジトリをはじめとする外部のリポジトリサーバへのアクセスをプロキシする **プロキシリポジトリ**
- インターネット上のリポジトリでは公開されていない、他者から提供された artifact を組織内部で配布するための **サードパーティリポジトリ**
- そのプロジェクト自体で開発された artifact を格納するための **プライベートリポジトリ**
- 複数の異なるリポジトリの artifact へのアクセスを一つのリポジトリ URL に集約するための **グループリポジトリ**

NEXUS ならこうした複数のリポジトリを楽に運用管理できる。

12.1.2 Install and Start up

NEXUS をインストールするマシンは次の条件を満たしている必要がある。

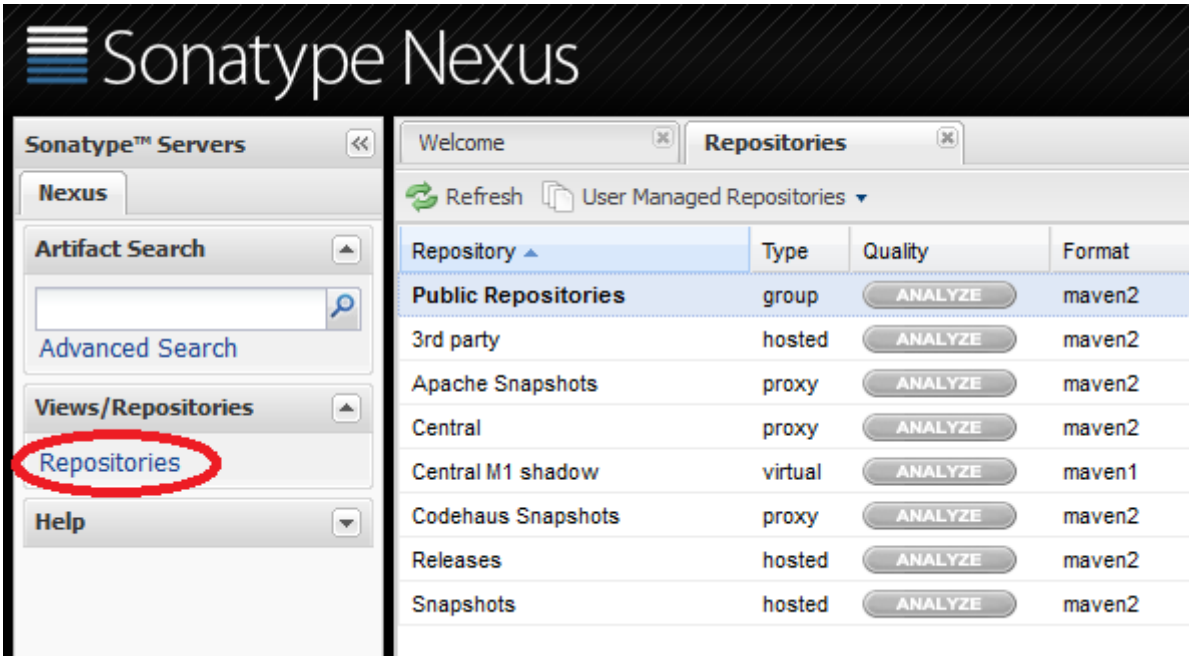
- JRE6 以上がインストール済みであること

- インターネット上のセントラルリポジトリ（先頭が http://repo1.maven.org/ で始まる URL）に http アクセス可能であること

インストール手順は次の通り。

1. NEXUS OSS をダウンロードし、アーカイブを展開する。
2. bin/nexus または bin/nexus.bat を実行すると NEXUS が起動する。
3. http://{[}IP or FQDN]:8081/nexus/ へアクセスし、NEXUS の初期画面が見えることを確認する。

いくつかのリポジトリがデフォルトで用意されている。特別な場合を除いて、デフォルトのままでも十分に開発に使える。画面左のメニュー部の Repositories をクリックするとリポジトリ一覧が表示される。



- **Central** = インターネット上のセントラルリポジトリ (http://repo1.maven.org/maven2/) への proxy の役割を果たすリポジトリ。
- **3rd party** = インターネット上で公開されているリポジトリにはないが、開発で必要となるサードパーティ製ライブラリを保管するリポジトリ。
- **Releases** = 自分たちで開発したアプリケーションのリリースバージョンの成果物を格納するリポジトリ。
- **Snapshots** = 自分たちで開発したアプリケーションの SNAPSHOT バージョンの成果物を格納するリポジトリ。
- **Public Repositories** = 上記 4 つのリポジトリへ、一つの URL でアクセスできるようにするためのグループリポジトリ。

12.1.3 settings.xml

構築した NEXUS を maven コマンドから使用するには、ローカル開発環境のユーザーホームディレクトリに settings.xml ファイルを作成しておく必要がある。

- Windows: C:/Users/[OSaccount]/.m2/settings.xml
- Unix: $HOME/.m2/settings.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>

  <mirrors>
    <mirror>
      <id>myteam-nexus</id>
      <mirrorOf>*</mirrorOf>
      <!-- CHANGE HERE by your team own nexus server -->
      <url>http:// IP or FQDN /nexus/content/groups/public </url>
    </mirror>
  </mirrors>

  <activeProfiles>
    <activeProfile>myteam-nexus</activeProfile>
  </activeProfiles>

  <profiles>
    <profile>
      <id>myteam-nexus</id>
      <repositories>
        <repository>
          <id>central</id>
          <url>http://central</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>central</id>
          <url>http://central</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>

</settings>
```

注釈: see also: Configuring Maven to Use a Single Repository Group / Documentation Sonatype.com

12.1.4 mvn deploy how to

jar/war ファイルを artifact としてパッケージリポジトリ（NEXUS）にアップロードするには、mvn deploy コマンドを使用する。

パッケージリポジトリに誰でもデプロイ可能な状態は混乱を招くので避けるべきである。そこで、Jenkins だけがパッケージリポジトリに対して mvn deploy 可能とする運用を推奨する。

Jenkins サーバ内の Jenkins の実行ユーザーのホームディレクトリ配下の.m2/settings.xml に、前述と同じ内容に加えて、さらに下記を追加しておく。

```
<servers>
  <server>
   <id>releases</id>
   <username>deployment</username>
   <password>deployment123</password>
  </server>
  <server>
    <id>snapshots</id>
    <username>deployment</username>
    <password>deployment123</password>
  </server>
</servers>
```

deployment はデプロイ権限を持つアカウント（NEXUS にデフォルトで設定済みの）であり、deployment123 はそのパスワードである。もちろん、NEXUS の GUI 画面上であらかじめパスワードを変更しておくことを推奨する。

注釈: settings.xml 上に plain text でパスワードを保存することを避けたい場合には、maven のパスワード暗号化機能を利用するとよい。詳しくは Maven - Password Encryption を参照のこと。

Jenkins のビルドジョブでは次のようにして mvn deploy 手順を設定する。

12.1.5 pom.xml

maven で管理されたプロジェクトでは、artifact となった自分自身をどのパッケージリポジトリに格納されるべきかを pom.xml 上の<distributionManagement>タグで表明する必要がある。

```
<distributionManagement>
  <repository>
    <id>releases</id>
    <!-- CHANGE HERE by your team nexus server -->
    <url>http://192.168.0.1:8081/nexus/content/repositories/releases/</url>
  </repository>
  <snapshotRepository>
    <id>snapshots</id>
    <!-- CHANGE HERE by your team nexus server -->
    <url>http://192.168.0.1:8081/nexus/content/repositories/snapshots/</url>
  </snapshotRepository>
</distributionManagement>
```

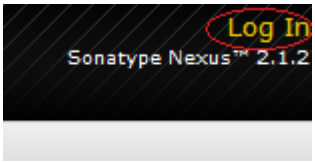
前述の mvn deploy コマンドは、<distributionManagement>タグで指定された URL に対して HTTP PUT で artifact をアップロードする。

12.1.6 Upload 3rd party artifact (ex. ojdbc6.jar)

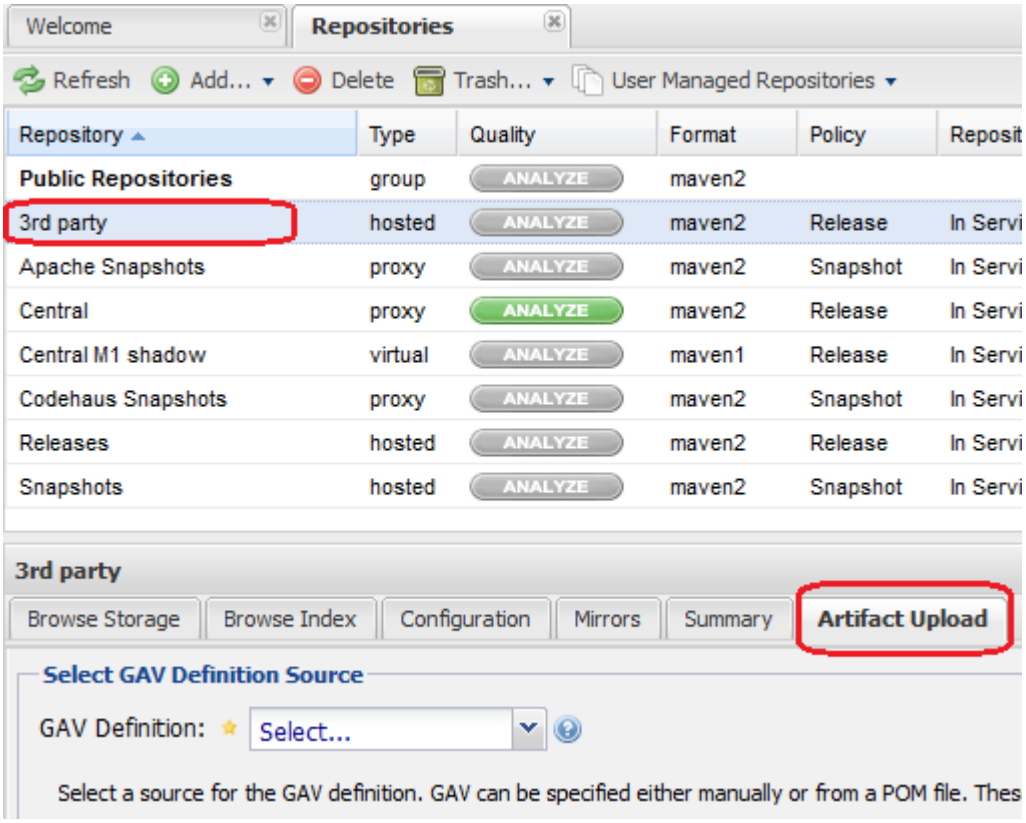
サードパーティ用リポジトリには、外部のリモートリポジトリでは公開されていない artifact を格納する。

典型的な例が、oracle の JDBC ドライバ (ojdbc*.jar) である。RDBMS として oracle を使用する場合に必須だが、セントラルリポジトリはもちろん、インターネット上の公開リポジトリに格納されていることはほとんどない。そのため、組織内のパッケージリポジトリに格納しておく必要がある。

1. admin ユーザーでログインする。（デフォルトのパスワードは admin123）



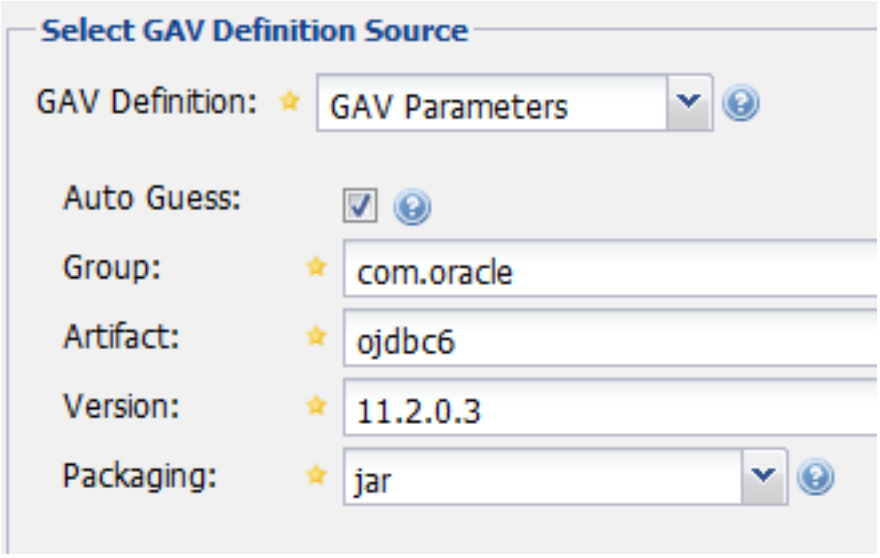
2. 3rdParty リポジトリを選択し、 **Artifact Upload** タブを選択する。

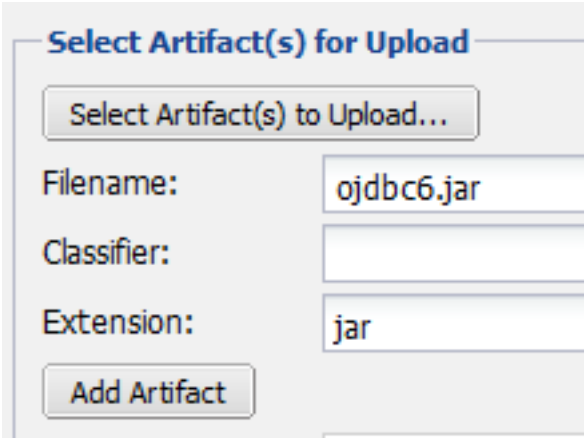


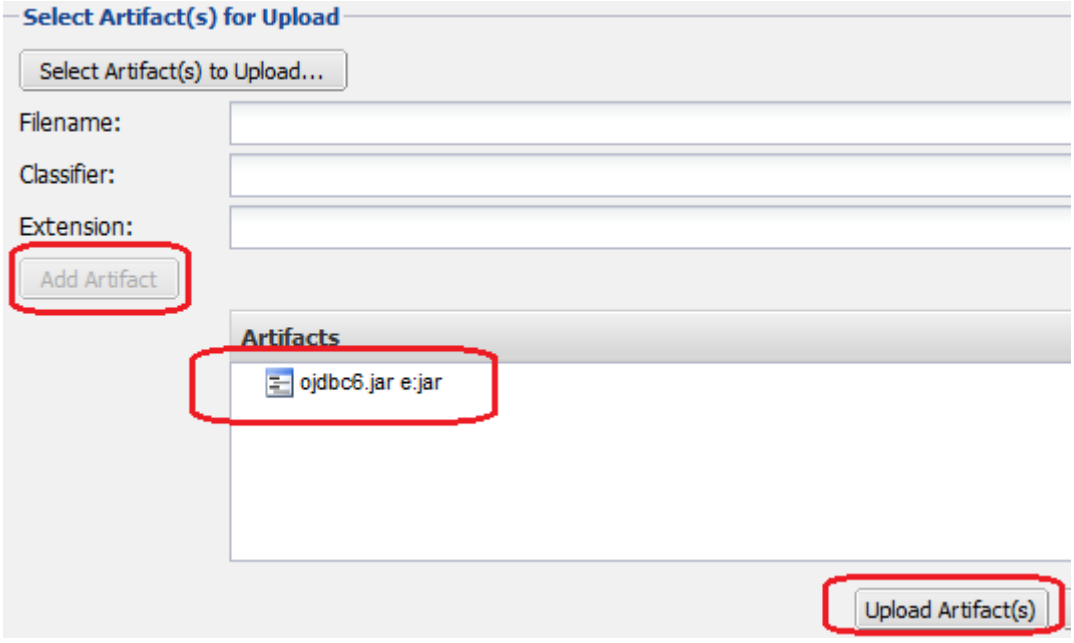
3. GAV 情報を入力する。(GAV = groupId, artifactId, version)

4. ローカル PC 上の ojdbc6.jar ファイルを選択し、 **Add Artifact** ボタンを押す。

5. 最後に **Upload Artifact(s)** ボタンを押すと、リポジトリに jar ファイルが格納される。


Select GAV Definition Source
GAV Definition:
GAV Parameters
Auto Guess:
Group:
com.oracle
Artifact:
ojdbc6
Version:
11.2.0.3
Packaging:
jar


Select Artifact(s) for Upload
Select Artifact(s) to Upload...
Filename:
ojdbc6.jar
Classifier:
Extension:
jar
Add Artifact


Select Artifact(s) for Upload
Select Artifact(s) to Upload...
Filename:
Classifier:
Extension:
Add Artifact
Artifacts
ojdbc6.jar e:jar
Upload Artifact(s)

以上でアップロード作業は完了。

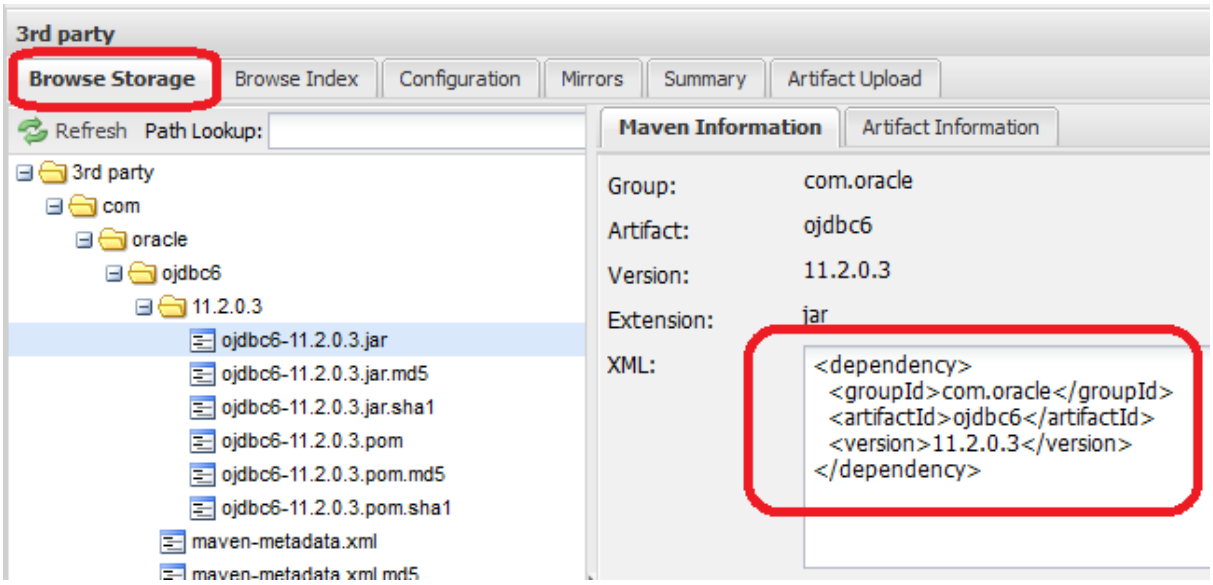
注釈: NEXUS の GUI 画面を使って artifact をアップロードする作業は完全に手作業でありオペレーションミスを誘発しやすいため、推奨しない。odjbc6.jar のような、サードパーティ製で、しかも 1 個または数個程度のファイルで構成可能な単純なライブラリに対してのみ、ここで説明している方法を用いるべきである。それ以外のケースでは **mvn deploy** コマンドを使うべきである。

---

use artifact

3rd party リポジトリ上の ojdbc6 をプロジェクトの依存性管理に追加するには、そのプロジェクトの pom.xml に dependency タグを追加するだけである。

Browse Storage タブから目的の artifact を選択すると、画面右側に dependency タグのサンプルが表示される。それを pom.xml にコピー＆ペーストすればよい。



12.2 ボイラープレートコードの排除 (Lombok)

12.2.1 Lombok とは

Lombok は、Java 言語におけるボイラープレートコードをソースコードから排除するために使用するライブラリである。

ボイラープレートコードとは、言語仕様上省く事ができない定型的なコードの事である。ボイラープレートコードは本質的なロジックでないため、アプリケーションを実装する上で冗長なコードとなる。

Java 言語における代表的なボイラープレートコードには、

- メンバー変数にアクセスするための getter / setter メソッド
- `equals/hashCode` メソッド
- `toString` メソッド
- コンストラクタ
- リソース (入出力ストリーム等) のクローズ処理
- ロガーインスタンスの生成

等がある。

Lombok は、これらのボイラープレートコードをコンパイル時に生成することで、開発者が実装するソースコード上から冗長なコードを取り除く仕組みを提供している。

ちなみに: リソース (入出力ストリーム等) のクローズ処理については、Java SE7 から追加された try-with-resources 文を使う事で、ボイラープレートコードにならないように言語仕様が改善されている。

Java 言語自体もバージョンアップする毎に、冗長なコードを記載しなくて済むように改善されている。Java SE8 からサポートされたラムダ式は、代表的な言語仕様の改善と言える。

12.2.2 Lombok の効果

以下に、Lombok を使用して作成した JavaBean のソースコードを示す。

```
package com.example.domain.model;

@lombok.Data
public class User {

    private String userId;
    private String password;
```

```
}
```

クラスレベルに`@lombok.Data`アノテーションを付与するだけで、JavaBean として必要なメソッドが Lombok によって生成される。

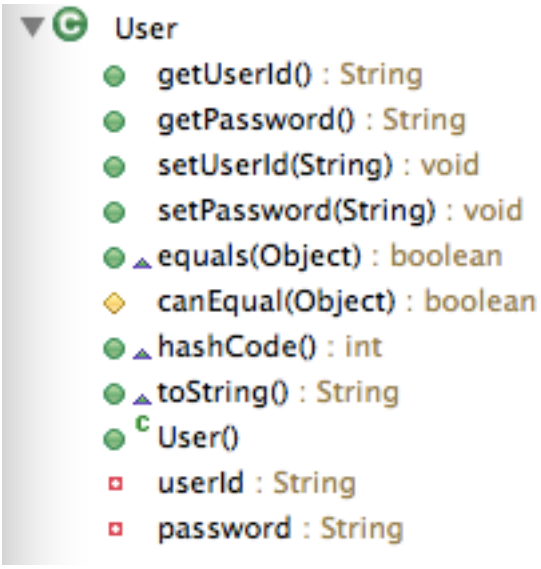


図 12.1　**Lombok** によって生成されたクラス構造

これは、Lombok の`@Data`アノテーションを付与しただけで、約 10 行のソースコードから、約 60 行ある下記のソースコード (Eclipse の自動生成機能を使用して出力したソースコード) によって生成されるクラスと同じ効果を得る事ができる事を意味している。

```
package com.example.domain.model;

public class User {

    private String userId;
    private String password;

    public User() {
    }

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
```

```
        this.password = password;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result
                + ((password == null) ? 0 : password.hashCode());
        result = prime * result + ((userId == null) ? 0 : userId.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        User other = (User) obj;
        if (password == null) {
            if (other.password != null)
                return false;
        } else if (!password.equals(other.password))
            return false;
        if (userId == null) {
            if (other.userId != null)
                return false;
        } else if (!userId.equals(other.userId))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return "User [userId=" + userId + ", password=" + password + "]";
    }

}
```

12.2.3 Lombok のセットアップ

依存ライブラリの追加

Lombok が提供しているクラスを使用するために、Lombok を依存ライブラリとして追加する。

```
<!-- (1) -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope> <!-- (2) -->
</dependency>
```

項番	説明
(1)	Lombok を使用するプロジェクトの pom.xml に、Lombok を依存ライブラリとして追加する。
(2)	Lombok はアプリケーション実行時には必要ないライブラリなので、スコープは provided が適切である。

注釈: 上記設定例は、依存ライブラリのバージョンを親プロジェクトである terasoluna-gfw-parent で管理する前提であるため、pom.xml でのバージョンの指定は不要である。上記の依存ライブラリは terasoluna-gfw-parent が利用している Spring IO Platform で定義済みである。

IDE 連携

Lombok を IDE 上で使用する場合は、IDE が提供するコンパイル (ビルド) 機能と連携するために、Lombok を IDE にインストールする必要がある。

本ガイドラインでは、Spring Tool Suite(以降、「STS」と呼ぶ) にインストールする方法を紹介する。使用する IDE によってインストール方法は異なるため、STS 以外の IDE を使用する場合は、 こちらのページ を参考にされたい。

Lombok のダウンロード

Lombok の jar ファイルをダウンロードする。

Lombok の jar ファイルは、

- Lombok のダウンロードページ

- Maven のローカルリポジトリ (通常は、*$HOME*/.m2/repository/org/projectlombok/lombok/<version>/lombok-<version>.jar)

から取得する。

Lombok のインストール

ダウンロードした Lombok の jar ファイルを実行 (ダブルクリック) し、インストーラーを立ち上げる。

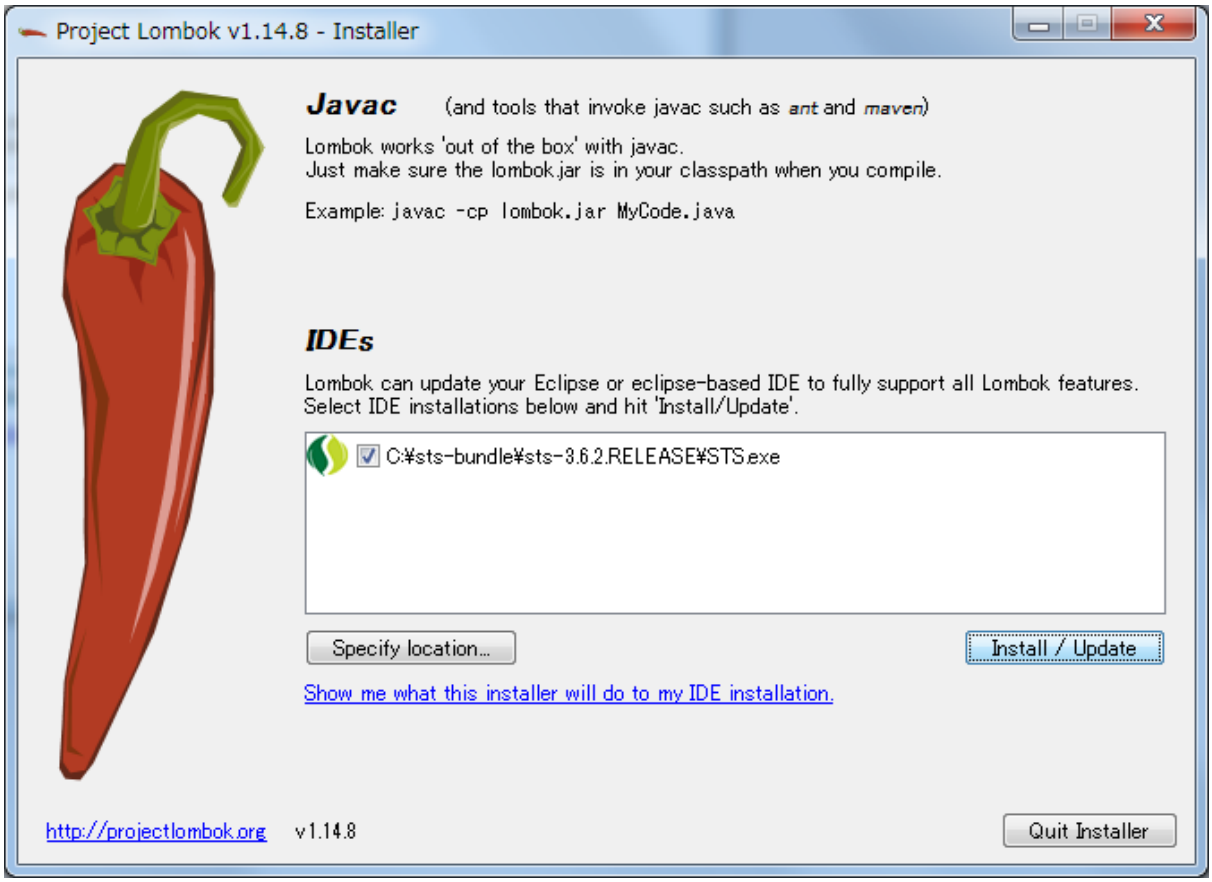


図 12.2 **Lombok のインストーラー**

インストール対象の STS を選択後、」Install / Update」 ボタンを押下してインストールを実行する。インストール候補の STS は、インストーラーによって自動検出される仕組みになっているが、自動で検出されない場合は、」Specify location …」を押下して IDE を指定する必要がある。

Lombok をインストールした後に STS を起動 (又は再起動) すると、STS 上で Lombok を使用して開発する事ができる。

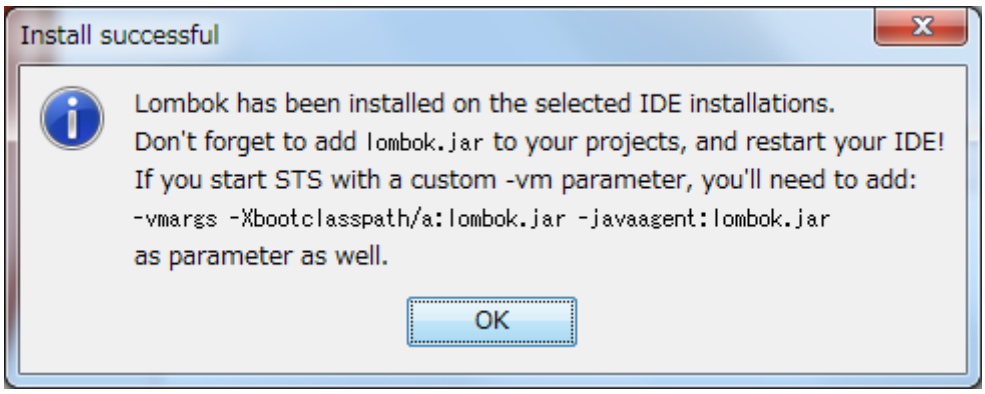


図 12.3　インストール成功時のダイアログ

12.2.4 Lombok の使用方法

ここからは、Lombok の具体的な使い方について説明していく。

Lombok を初めて使用する場合は、まず、Lombok の「Demo Video」を参照するとよい。Demo Video は 4 分弱で構成されており、最も基本的な使い方が説明されている。

Lombok が提供しているアノテーション

まず、Lombok が提供する代表的なアノテーションを紹介する。

各アノテーションの詳細な使用方法や、本ガイドラインで紹介していないアノテーションの使い方については、

- Lombok features

を参照されたい。

項番	アノテーション	説明
1.	@lombok.Getter	getter メソッドを生成するためのアノテーション。 クラスレベルにアノテーションを指定すると、全てのフィールドに getter メソッドを生成する事ができる。
2.	@lombok.Setter	setter メソッドを生成するためのアノテーション。 クラスレベルにアノテーションを指定すると、全ての非 final フィールドに setter メソッドを生成する事ができる。
3.	@lombok.ToString	`toString` メソッドを生成するためのアノテーション。
4.	@lombok.EqualsAndHashCode	`equals` と `hashCode` メソッドを生成するためのアノテーション。
5.	@lombok.RequiredArgsConstructor	初期化が必要なフィールド (final フィールドなど) の初期化パラメータを引数に持つコンストラクタを生成するためのアノテーション。 全てのフィールドが任意のフィールドの場合は、デフォルトコンストラクタ (引数なしのコンストラクタ) が生成される。
6.	@lombok.AllArgsConstructor	全てのフィールドの初期化パラメータを引数に持つコンストラクタを生成するためのアノテーション。
7.	@lombok.NoArgsConstructor	デフォルトコンストラクタを生成するためのアノテーション。
8.	@lombok.Data	`@Getter`、`@Setter`、`@ToString`、`@EqualsAndHashCode`、`@RequiredArgsConstructor` へのショートカットアノテーション。 `@Data` アノテーションを指定すると、上記 5 つのアノテーションを指定したのと同じ意味となる。
9.	@lombok.extern.slf4j.Slf4j	SLF4J のロガーインスタンスを生成するためのアノテーション。

JavaBean の作成

本ガイドラインが推奨する方法でアプリケーションを構築した場合、

- Form クラス
- Resource クラス (REST API 構築時)
- Entity クラス
- DTO クラス

などの JavaBean を作成する必要がある。

以下に、JavaBean の作成例を示す。

```
package com.example.domain.model;

import lombok.Data;

@Data // (1)
public class User {

    private String userId;
    private String password;

}
```

項番	説明
(1)	クラスレベルに、`@Data` アノテーションを指定し、 • getter/setter メソッド • `equals`/ `hashCode` メソッド • `toString` メソッド • デフォルトコンストラクタ を生成する。

toString の対象から特定のフィールドを除外する方法

オブジェクトの状態を文字列に変換する際は、

- 相互参照関係をもつオブジェクトを保持するフィールド
- 個人情報やパスワードなどの機密情報を保持するフィールド

などを文字列変換の対象から除外する事が必要になるケースがある。これらのフィールドを変換対象から除外しない場合、

- 前者は、循環参照となり `StackOverflowError` や `OutOfMemoryError` などが発生する
- 後者は、変換後の文字列の使用方法によっては、個人情報の漏洩に繋がる

可能性があるので、注意が必要である。

> **警告:** JPA の Entity クラスに`@Data` や`@ToString` アノテーションを使用する場合は、循環参照になりやすいので特に注意が必要である。

以下に、特定のフィールドを文字列変換の対象から除外する方法を示す。

```
package com.example.domain.model;

import lombok.Data;
import lombok.ToString;

@Data
@ToString(exclude = "password") // (1)
public class User {

    private String userId;
    private String password;

}
```

項番	説明
(1)	クラスレベルに@ToStringアノテーションを指定し、exclude属性に除外したいフィールド名を列挙する。 上記例のソースコードから生成されたクラスのtoStringメソッドを呼び出すと、 • User(userId=U00001) という文字列に変換される。

equals と hashCode の対象から特定のフィールドを除外する方法

Lombokのアノテーションを使用してequalsメソッドとhashCodeメソッドを作成する場合は、相互参照関係をもつオブジェクトを保持するフィールドを除外して生成する必要がある。

これらのフィールドを除外せずに生成した場合、循環参照となりStackOverflowErrorやOutOfMemoryErrorなどが発生するので、注意が必要である。

警告: JPAのEntityクラスにDataアノテーション、Valueアノテーション、@EqualsAndHashを使用する場合は、循環参照になりやすいので特に注意が必要である。

以下に、特定のフィールドを除外する方法を示す。

```
package com.example.domain.model;

import java.util.List;

import lombok.Data;

@Data
public class Order {

    private String orderId;
    private List<OrderLine> orderLines;

}
```

```
package com.example.domain.model;

import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.ToString;

@Data
@ToString(exclude = "order")
@EqualsAndHashCode(exclude = "order") // (1)
public class OrderLine {

    private Order order;
    private String itemCode;
    private int quantity;

}
```

項番	説明
(1)	クラスレベルに`@EqualsAndHashCode` アノテーションを指定し、`exclude` 属性に除外したいフィールド名を列挙する。

ちなみに: 除外するフィールドを指定するのではなく、特定のフィールドのみを使用するように指定することもできる。

```
@Data
@ToString(exclude = "order")
@EqualsAndHashCode(of = "itemCode") // (2)
public class OrderLine {

    private final Order order;
    private final String itemCode;
```

```
    private final int quantity;

}
```

項番	説明
(2)	特定のフィールドのみを使用する場合は、@EqualsAndHashCode アノテーションの of 属性に対象のフィールド名を列挙する。 上記例では、itemCode フィールドのみを参照して処理を行う equals メソッドと hashCode メソッドが生成される。

フィールド初期化用のコンストラクタを生成する方法

アプリケーションの実装コードから JavaBean のインスタンスを生成する場合は、フィールドの初期値を引数に渡す事ができるコンストラクタがあった方が便利であり、冗長なコードを排除することもできる。

デフォルトコンストラクタを使用してインスタンスを生成した場合は、以下のようなコードとなる。

```
public void login(String userId, String password) {
    User user = new User();
    user.setUserId(userId);
    user.setPassword(password);
    // ...
}
```

以下に、フィールドの初期値を指定するコンストラクタを生成する方法を示す。

```
package com.example.domain.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;

@Data
@AllArgsConstructor // (1)
@NoArgsConstructor  // (2)
@ToString(exclude = "password")
```

```
public class User {

    private String userId;
    private String password;

}
```

```
public void login(String userId, String password) {
    User user = new User(userId, password); // (3)
    // ...
}
```

項番	説明
(1)	クラスレベルに@AllArgsConstructor アノテーションを指定し、全てのフィールドの初期値を引数にとるコンストラクタを生成する。
(2)	クラスレベルに@NoArgsConstructor アノテーションを指定し、デフォルトコンストラクタを生成する。 JavaBean として使用する場合は、デフォルトコンストラクタも生成しておく必要がある。
(3)	フィールドの初期値を指定するコンストラクタを呼び出し、JavaBean のインスタンスを生成する。 デフォルトコンストラクタを使用した場合は 3 ステップ必要だったものが、1 ステップでインスタンスの生成が出来るようになった。

警告: @Data と@NoArgsConstructor を付与する順序について

TERASOLUNA Framework 5.4.2.RELEASE が利用する Lombok 1.16.22 では、@Data と@NoArgsConstructor の付与順序によってはコンパイルエラーが発生するという事象が確認されている。具体的には、@Data により生成されたデフォルトコンストラクタを@NoArgsConstructor が再度生成することによりコンパイルエラーになる事象で、Lombok 1.18.0 で修正されている。

この事象を回避するには、@NoArgsConstructor を@Data より上に付与すれば良い。また、Lombok のバージョンを 1.18.0 以上に上げることでも回避可能である。詳しくは、Lombok-1.16.22-constructor already defined in class を参照されたい。

ちなみに: 上記例で扱っている User クラスを、JavaBean ではなく、Immutable なクラスにしたい場合は、@lombok.Value アノテーションを使用するとよい。@Value アノテーションについては、Lombok のリ

ファレンス を参照されたい。

ロガーインスタンスの作成

デバッグログやアプリケーションログを出力するために、ロガーインスタンスを生成する必要がある場合は、ロガーインスタンスを生成するためのアノテーションを使用するとよい。

Lombok のアノテーションを使用しないでロガーインスタンスを作成する場合は、以下のようなコードになる。

```
package com.example.domain.service;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;

@Service
public class AuthenticationService {

    private static final Logger logger = LoggerFactory.
↪getLogger(AuthenticationService.class);

    public void login(String userId, String password) {
        log.info("{} had tried login.", userId);
        // ...
    }

}
```

以下に、Lombok のアノテーションを使用してロガーインスタンスを作成する方法を示す。

```
package com.example.domain.service;

import org.springframework.stereotype.Service;

import lombok.extern.slf4j.Slf4j;

@Slf4j // (1)
@Service
public class AuthenticationService {

    public void login(String userId, String password) {
```

```
        log.info("{} had tried login.", userId); // (2)
        // ...
    }

}
```

項番	説明
(1)	クラスレベルに`@Slf4j`アノテーションを指定し、SLF4J のロガーインスタンスを生成する。 本ガイドラインでは、SLF4J の `org.slf4j.Logger` を使用してログを出力する前提である。 デフォルトでは、アノテーションを付与したクラスの FQCN(上記例だと `com.example.domain.service.LoginService`) がロガー名として使用され、ロガー名に対応するロガーインスタンスが `log` という名前のフィールドに設定される。
(2)	Lombok によって生成された SLF4J のロガーインスタンスのメソッドを呼び出し、ログを出力する。 上記例では、 • `11:29:45.838 [main] INFO c.e.d.service.AuthenticationService - U00001 had tried login.` というログが出力される。

ちなみに: デフォルトで使用されるロガー名を変更したい場合は、`@Slf4j` アノテーションの `topic` 属性に、任意のロガー名を指定すればよい。

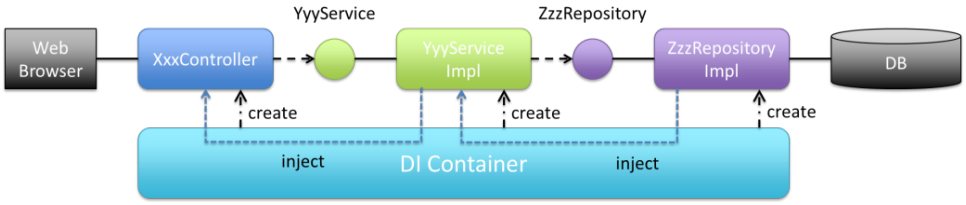
12.3 参考書籍

本ガイドラインを執筆する上で参考にした書籍を列挙する。必要に応じて参照されたい。

書籍名	出版社	備考
Spring 徹底入門	翔泳社	日本語
Pro Spring 4th Edition	APress	
Pro Spring 3	APress	
Pro Spring MVC: With Web Flow	APress	
Spring Persistence with Hibernate	APress	
Spring in Practice	Manning	
Spring in Action, Third Edition	Manning	
Spring Data Modern Data Access for Enterprise Java	O'Reilly Media	
Spring Security 3.1	Packt Publishing	
Spring3 入門—Java フレームワーク・より良い設計とアーキテクチャ	技術評論社	日本語
Beginning Java EE 6 GlassFish 3 で始めるエンタープライズ Java	翔泳社	日本語
Seasar2 と Hibernate で学ぶデータベースアクセス JPA 入門	毎日コミュニケーションズ	日本語

12.4 Spring Framework 理解度チェックテスト

1. Bean の依存関係が以下の図のようになるように (1)～(4) を埋めてください。import 文は省略してください。




```
@Controller
public class XxxController {
  (1)
  protected (2) yyyService;

  // omitted
}
```

```
@Service
@Transactional
public class YyyServiceImpl implements YyyService {
  (1)
  protected (4) zzzRepository;

  // omitted
}
```

注釈: @Service,@Controller は org.springframework.stereotype パッケージのアノテーション、@Transactional は org.springframework.transaction.annotation のアノテーションである。

---

2. @Controller と@Service と@Repository はそれぞれどういう場合に使用するか説明してください。

注釈: それぞれ org.springframework.stereotype パッケージのアノテーションです。

---

3. @Resource と@Inject の違いを説明してください

注釈: @Resource は javax.annotation パッケージ、@Inject は javax.inject パッケージのアノテーションです。

---

4. Scope が singleton の場合と prototype の場合の違いを説明してください。

5. Scope に関する次の説明で (1)〜(3) を埋めてください。ただし (1)、(2) には」singleton」または」prototype」のどちらが入り、同じ値は入りません。また import 文は省略してください。

```
@Component
(3)
public class XxxComponent {
  // omitted
}
```

注釈: `@Component` は `org.springframework.stereotype.Component`

`@Component` をつけた Bean の scope はデフォルトで (1) である。scope を (2) にする場合、(3) をつければよい (上記ソース参照)。

6. 次の Bean 定義を行った場合、どのような Bean が DI コンテナに登録されますか。

```
<bean id="foo" class="xxx.yyy.zzz.Foo" factory-method="create">
    <constructor-arg index="0" value="aaa" />
    <constructor-arg index="1" value="bbb" />
</bean>
```

7. `com.example.domain` パッケージ以下が component scan の対象となるように以下の Bean 定義の (1)〜(3) を埋めてください。

```
<context:(1)  (2)="(3)" />
```

注釈: Bean 定義ファイルには

xmlns:context=」http://www.springframework.org/schema/context」

の定義があるものとする。

---

8. プロパティファイルに関する次の説明で (1)〜(2) を埋めてください。import 文は省略してください。

   設定値をプロパティファイルに外出しし、Bean 定義ファイル内から`${key}`形式で参照したい場合に`<context:property-placeholder>`要素の `locations` 属性にプロパティファイルのパスを設定すれば読み込むことができる。クラスパス直下の META-INF/spring ディレクトリ以下の任意のプロパティファイルを読み込む場合は (1) のように指定する。また読み込んだプロパティ値は Bean にもインジェクション可能であり下記コードのように@(2) アノテーションをつければよい。

```
<context:property-placeholder locations="(1)" />
```

```
emails.min.count=1
emails.max.count=4
```

```
@Service
@Transactional
public class XxxServiceImpl implements XxxService {
  @xxx("${emails.min.count}") // (2)xxx 部分
  protected int emailsMinCount;
  @xxx("${emails.max.count}") // (2)xxx 部分
  protected int emailsMaxCount;
  // omitted
}
```

注釈: Bean 定義ファイルには

xmlns:context=」http://www.springframework.org/schema/context」

の定義があるものとする。

---

9. Spring が提供する AOP の Advice についての次の説明で (1)〜(5) を埋めてください。尚、(1)〜(5) には全て別の内容が入ります。

注釈: 特定のメソッド呼び出しの前に処理を割り込ませたい場合の Advice は (1) で、メソッド呼び出し後に割り込ませたい場合の Advice は (2) である。前後両方に割り込ませたい場合は (3) Advice を使用すればよい。メソッドが正常終了したときにのみ実行される Advice は (4) であり、例外発生時に実行される Advice は (5) である。

---

10. `@Transactional` アノテーションによるトランザクション管理を行うために以下の Bean 定義の (*) を埋めてください。

```
<tx:(*) />
```

注釈: Bean 定義ファイルには

xmlns:tx=」http://www.springframework.org/schema/tx」

の定義があるものとする。
