TERASOLUNA Global Framework Development Guideline Documentation

リリース *1.0.4.RELEASE*

NTT DATA

2016 年 02 月 23 日

目次

第 1 章　はじめに 3
1.1 利用規約 . . . 3
1.2 このドキュメントが示すこと . . . 5
1.3 このドキュメントの対象読者 . . . 5
1.4 このドキュメントの構成 . . . 5
1.5 このドキュメントの読み方 . . . 6
1.6 ガイドラインの観点別マッピング . . . 7
1.7 更新履歴 . . . 9

第 2 章　TERASOLUNA Global Framework のアーキテクチャ概要 17
2.1 TERASOLUNA Global Framework のスタック . . . 17
2.2 Spring MVC アーキテクチャ概要 . . . 26
2.3 はじめての Spring MVC アプリケーション . . . 32
2.4 アプリケーションのレイヤ化 . . . 51

第 3 章　チュートリアル (Todo アプリケーション) 69
3.1 はじめに . . . 69
3.2 作成するアプリケーションの説明 . . . 70
3.3 環境構築 . . . 72
3.4 Todo アプリケーションの作成 . . . 108
3.5 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 . . . 148
3.6 おわりに . . . 158

第 4 章　TERASOLUNA Global Framework によるアプリケーション開発 161
4.1 ドメイン層の実装 . . . 161
4.2 インフラストラクチャ層の実装 . . . 221
4.3 アプリケーション層の実装 . . . 229

第 5 章　TERASOLUNA Global Framework の機能詳細 315
5.1 データベースアクセス（共通編） . . . 315
5.2 データベースアクセス（JPA 編） . . . 347
5.3 データベースアクセス（Mybatis2 編） . . . 453
5.4 排他制御 . . . 534
5.5 入力チェック . . . 576

5.6 ロギング . . . 645
5.7 例外ハンドリング . . . 667
5.8 セッション管理 . . . 724
5.9 メッセージ管理 . . . 787
5.10 プロパティ管理 . . . 819
5.11 ページネーション . . . 832
5.12 二重送信防止 . . . 883
5.13 国際化 . . . 934
5.14 コードリスト . . . 945
5.15 Ajax . . . 983
5.16 RESTful Web Service . . . 1013
5.17 ファイルアップロード . . . 1164
5.18 ファイルダウンロード . . . 1206
5.19 Tiles による画面レイアウト . . . 1217
5.20 システム時刻 . . . 1237
5.21 ユーティリティ . . . 1255

第 6 章 TERASOLUNA Global Framework によるセキュリティ対策 1305
6.1 Spring Security 概要 . . . 1305
6.2 Spring Security チュートリアル . . . 1310
6.3 認証 . . . 1333
6.4 パスワードハッシュ化 . . . 1381
6.5 認可 . . . 1391
6.6 XSS 対策 . . . 1409
6.7 CSRF 対策 . . . 1418

第 7 章 Appendix 1433
7.1 チュートリアル (Todo アプリケーション REST 編) . . . 1433
7.2 Blank プロジェクトから新規プロジェクト作成 . . . 1501
7.3 共通ライブラリが提供する JSP Tag Library と EL Functions . . . 1508
7.4 NEXUS による Maven リポジトリの管理 . . . 1520
7.5 環境依存性の排除 . . . 1528
7.6 Project Structure Standard . . . 1536
7.7 参考書籍 . . . 1539
7.8 Spring Framework 理解度チェックテスト . . . 1540

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 Global 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 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。
- 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 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。

1.3 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。

- 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 3 (Apress)

1.4 このドキュメントの構成

- TERASOLUNA Global Framework のアーキテクチャ概要 Spring MVC の概要や、TERASOLUNA Global Framework の基本的な考え方を説明する。
- チュートリアル (Todo アプリケーション) 簡単なアプリケーション開発を通して、TERASOLUNA Global Framework によるアプリケーション開発を体験する。
- TERASOLUNA Global Framework によるアプリケーション開発 TERASOLUNA Global Framework 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。
- TERASOLUNA Global Framework の機能詳細 一般的にアプリケーション開発で必要となる機能を TERASOLUNA Global Framework を利用してどう実装するか、何に気を付けるべきかを機能ごとに説明する。

- TERASOLUNA Global Framework によ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。
- Appendix TERASOLUNA Global Framework を利用する場合の付加情報を説明する。

1.5 このドキュメントの読み方

まずは”TERASOLUNA Global Framework の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 (Todo アプリケーション)“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 Global Framework 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、”TERASOLUNA Global Framework による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 (Todo アプリケーション)“を振り返るとより理解が深まる。

ここまでは TERASOLUNA Global Framework を使用するすべての開発者が読むことを強く推奨する。

“TERASOLUNA Global Framework の機能詳細 “、”TERASOLUNA Global Framework によるセキュリティ対策 “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. TERASOLUNA Global Framework による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。

1.6 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。

1.6.1 セキュリティ対策に関するマッピング

OWASP Top 10 for 2013 を軸として、セキュリティに関連する機能の説明へのリンクを記載する。

項番	項目名	関連するガイドライン
A1	Injection SQL Injection	• データベースアクセス（JPA 編） • データベースアクセス（Mybatis2 編） (クエリにパラメータを埋め込む場合は、バインド変数を使用する旨を記載)
A1	Injection XXE(XML External Entity) Injection	• Ajax
A2	Broken Authentication and Session Management	• 認証
A3	Cross-Site Scripting (XSS)	• XSS 対策
A4	Insecure Direct Object References	特に言及なし
A5	Security Misconfiguration	• ロギング (ログのメッセージ内容に言及) • **システム例外の例外コードを、画面表示する方法** (システム例外時に出力するメッセージ内容に言及)
A6	Sensitive Data Exposure	• プロパティ管理 • パスワードハッシュ化 (パスワードハッシュにのみ言及)
A7	Missing Function Level Access Control	• 認可
A8	Cross-Site Request Forgery (CSRF)	• CSRF 対策
A9	Using Components with Known Vulnerabilities	特に言及なし
A10	Unvalidated Redirects and Forwards	• 認証 (オープンリダイレクタ脆弱性対策について言及)

1.7 更新履歴

更新日付	更新箇所	更新内容
2016-02-24	-	1.0.4 RELEASE 版公開 • 更新内容の詳細は、1.0.4 の Issue 一覧を参照されたい。
	TERASOLUNA Global Framework のスタック	利用する OSS のバージョンを更新 • GroupId「org.springframework」のバージョンを 3.2.14.RELEASE から 3.2.16.RELEASE に更新 • GroupId「commons-collections」のバージョンを 3.2.1 から 3.2.2 に更新
2015-07-31	-	1.0.3 RELEASE 版公開 更新内容の詳細は、1.0.3 の Issue 一覧を参照されたい。
	TERASOLUNA Global Framework のスタック	Spring Framework のバグ (セキュリティ脆弱性) 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 3.2.13.RELEASE から 3.2.14.RELEASE に更新
	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 (guideline#1211)
2015-02-23	-	1.0.2 RELEASE 版公開 更新内容の詳細は、1.0.2 の Issue 一覧を参照されたい。
	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 • 修正内容の詳細は、1.0.2 の Issue 一覧 (clerical error) を参照されたい。 記載内容の改善 • 改善内容の詳細は、1.0.2 の Issue 一覧 (improvement) を参照されたい。 新規追加 • 共通ライブラリが提供する JSP Tag Library と EL Functions
	TERASOLUNA Global Framework のスタック	Spring Framework のバグ (セキュリティ脆弱性) 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 3.2.10.RELEASE から 3.2.13.RELEASE に更新

次のページに続く

TABLE 1.1 – 前のページからの続き

更新日付	更新箇所	更新内容
	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。修正内容の詳細は、guideline#364 の Issue を参照されたい。
	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。修正内容の詳細は、guideline#712 の Issue を参照されたい。
	データベースアクセス（Mybatis2 編）	ガイドラインのバグ修正 • LOB 型を扱うための設定と説明を修正。修正内容の詳細は、guideline#402 の Issue を参照されたい。
	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。修正内容の詳細は、guideline#450 の Issue を参照されたい。
	入力チェック	ガイドラインのバグ修正 • `@GroupSequence` の説明を修正。修正内容の詳細は、guideline#296 の Issue を参照されたい。 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。修正内容の詳細は、guideline#766 の Issue を参照されたい。 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、guideline#320 の Issue を参照されたい。

次のページに続く

TABLE 1.1 – 前のページからの続き

更新日付	更新箇所	更新内容
	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をレスポンスすると 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、guideline#189 の Issue を参照されたい。
	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#256) に伴い、`@ExistInCodeList` のメッセージ定義に関する注意点を追加。修正内容の詳細は、guideline#766 の Issue を参照されたい。
	Ajax	記載内容の改善 • Spring Security 3.2(version 5.0.0 で利用) との互換性を持たせるため、CSRF 対策のサンプルコード (`<meta>`タグの `name` 属性に設定する値) を修正。改善内容の詳細は、guideline#680 の Issue を参照されたい。
	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、guideline#374 の Issue を参照されたい。
	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。修正内容の詳細は、guideline#846 の Issue を参照されたい。 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、guideline#778 の Issue を参照されたい。
	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、guideline#584 の Issue を参照されたい。

次のページに続く

TABLE 1.1 – 前のページからの続き

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。修正内容の詳細は、guideline#754 の Issue を参照されたい。 • 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)。修正内容の詳細は、guideline#765 の Issue を参照されたい。
	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。

次のページに続く

TABLE 1.1 – 前のページからの続き

更新日付	更新箇所	更新内容
	英語版	以下の英語版を追加 • 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（Mybatis2 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 • パスワードハッシュ化 • 認可 • Blank プロジェクトから新規プロジェクト作成 • NEXUS による Maven リポジトリの管理 • 環境依存性の排除 • Project Structure Standard • Spring Framework 理解度チェックテスト
2014-08-27	-	1.0.1 RELEASE 版公開 更新内容の詳細は、1.0.1 の Issue 一覧を参照されたい。
	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 更新内容の詳細は、1.0.1 の Issue 一覧を参照されたい。
	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)

次のページに続く

TABLE 1.1 – 前のページからの続き

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 • TERASOLUNA Global Framework の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) • TERASOLUNA Global Framework による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 Global Framework 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)

次のページに続く

TABLE 1.1 – 前のページからの続き

更新日付	更新箇所	更新内容
	日本語版 Ajax	XXE Injection 対策に関する記載を更新
	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。修正内容の詳細は、guideline#59 の Issue を参照されたい。
	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 • Blank プロジェクトから新規プロジェクト作成
2013-12-17	日本語版	1.0.0 Public Review 版公開

第 2 章

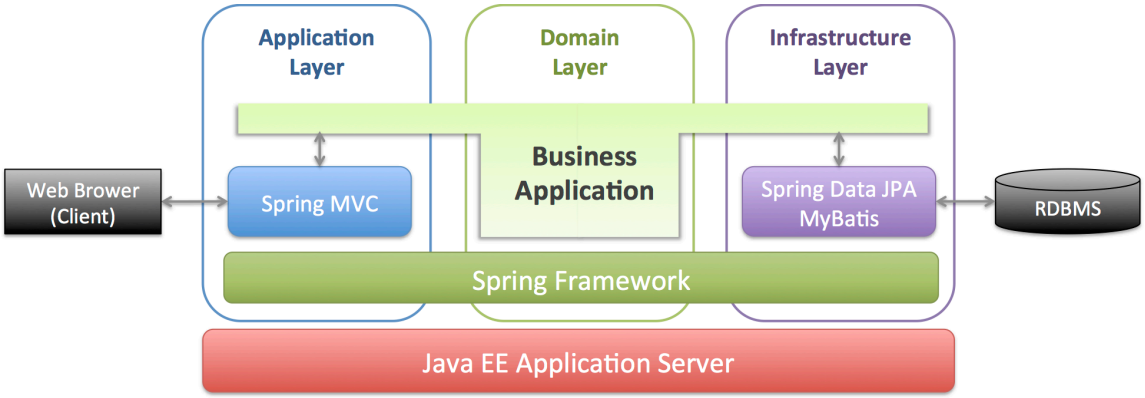
TERASOLUNA Global Framework のアーキテクチャ概要

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 Global Framework のスタック

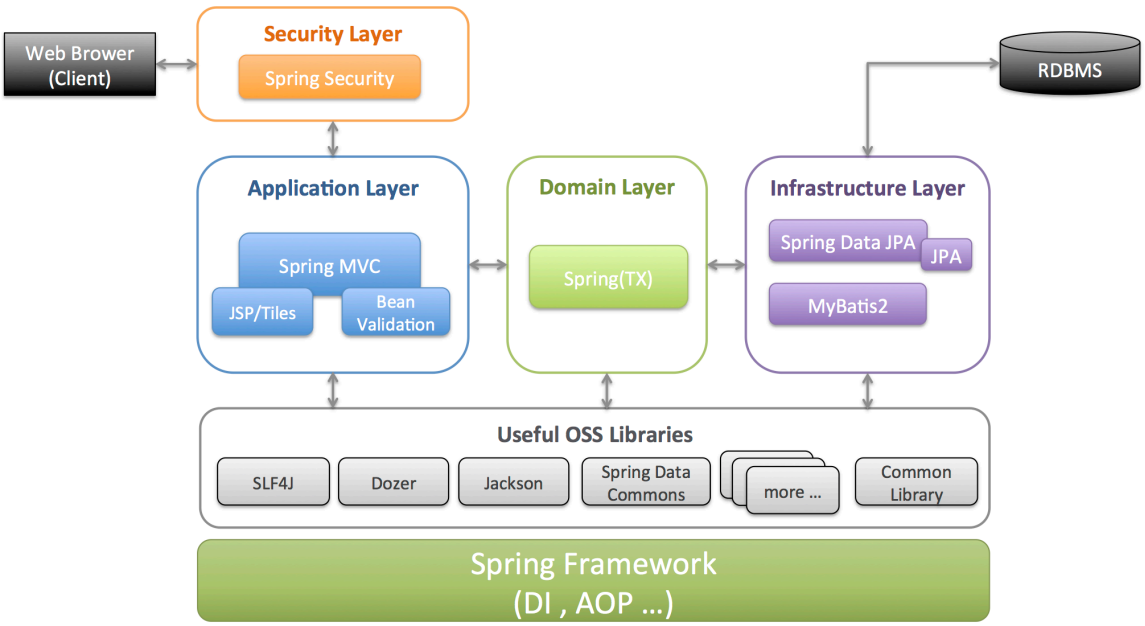
2.1.1 TERASOLUNA Global Framework の Software Framework 概要

TERASOLUNA Global Framework 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 Global Framework 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 3.2

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 3.2

O/R Mapper

本ガイドラインでは、以下の**どちらか**を想定している。

- JPA2.0
  - プロバイダは、Hibernate 4.2 を使用する。
- MyBatis 2.3.5
  - ラッパーとして、TERASOLUNA Framework の DAO(TERASOLUNA DAO) 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、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 2.2

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 3.1

課題

今後、Spring Security 3.2 に update する予定である。

バリデーション

- 単項目チェックには BeanValidation 1.0 を利用する。
  - 実装は、Hibernate Validator 4.3 を利用する。
- 相関チェックには BeanValidation、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 1.0.4.RELEASE で利用する OSS の一覧を以下に示す。

Type	GroupId	ArtifactId	Version	Remarks
Spring	org.springframework	spring-aop	3.2.16.RELEASE	
Spring	org.springframework	spring-aspects	3.2.16.RELEASE	
Spring	org.springframework	spring-beans	3.2.16.RELEASE	
Spring	org.springframework	spring-context	3.2.16.RELEASE	
Spring	org.springframework	spring-context-support	3.2.16.RELEASE	
Spring	org.springframework	spring-core	3.2.16.RELEASE	
Spring	org.springframework	spring-expression	3.2.16.RELEASE	
Spring	org.springframework	spring-jdbc	3.2.16.RELEASE	
Spring	org.springframework	spring-orm	3.2.16.RELEASE	
Spring	org.springframework	spring-tx	3.2.16.RELEASE	
Spring	org.springframework	spring-web	3.2.16.RELEASE	
Spring	org.springframework	spring-webmvc	3.2.16.RELEASE	
Spring	org.springframework.data	spring-data-commons	1.6.4.RELEASE	
Spring	org.springframework.security	spring-security-acl	3.1.4.RELEASE	
Spring	org.springframework.security	spring-security-config	3.1.4.RELEASE	
Spring	org.springframework.security	spring-security-core	3.1.4.RELEASE	
Spring	org.springframework.security	spring-security-taglibs	3.1.4.RELEASE	
Spring	org.springframework.security	spring-security-web	3.1.4.RELEASE	
JPA(Hibernate)	antlr	antlr	2.7.7	*1
JPA(Hibernate)	dom4j	dom4j	1.6.1	*1
JPA(Hibernate)	org.hibernate	hibernate-core	4.2.3.Final	*1
JPA(Hibernate)	org.hibernate	hibernate-entitymanager	4.2.3.Final	*1
JPA(Hibernate)	org.hibernate.common	hibernate-commons-annotations	4.0.2.Final	*1
JPA(Hibernate)	org.hibernate.javax.persistence	hibernate-jpa-2.0-api	1.0.1.Final	*1
JPA(Hibernate)	org.javassist	javassist	3.15.0-GA	*1
JPA(Hibernate)	org.jboss.spec.javax.transaction	jboss-transaction-api_1.1_spec	1.0.1.Final	*1
JPA(Hibernate)	org.springframework.data	spring-data-jpa	1.4.3.RELEASE	*1
MyBatis2	jp.terasoluna.fw	terasoluna-dao	2.0.5.0	*2
MyBatis2	jp.terasoluna.fw	terasoluna-ibatis	2.0.5.0	*2
MyBatis2	org.mybatis	mybatis	2.3.5	*2
DI	javax.inject	javax.inject	1	

TABLE 2.1 – 前のページからの続き

Type	GroupId	ArtifactId	Version	Remarks
AOP	aopalliance	aopalliance	1	
AOP	org.aspectj	aspectjrt	1.7.4	
AOP	org.aspectj	aspectjweaver	1.7.4	
ログ出力	ch.qos.logback	logback-classic	1.0.13	
ログ出力	ch.qos.logback	logback-core	1.0.13	
ログ出力	org.lazyluke	log4jdbc-remix	0.2.7	
ログ出力	org.slf4j	jcl-over-slf4j	1.7.5	
ログ出力	org.slf4j	slf4j-api	1.7.5	
JSON	org.codehaus.jackson	jackson-core-asl	1.9.7	
JSON	org.codehaus.jackson	jackson-mapper-asl	1.9.7	
入力チェック	javax.validation	validation-api	1.0.0.GA	
入力チェック	org.hibernate	hibernate-validator	4.3.1.Final	
入力チェック	org.jboss.logging	jboss-logging	3.1.0.GA	
Bean 変換	commons-beanutils	commons-beanutils	1.8.3	*3
Bean 変換	net.sf.dozer	dozer	5.4.0	*3
Bean 変換	org.apache.commons	commons-lang3	3.1	*3
日付操作	joda-time	joda-time	2.2	
日付操作	joda-time	joda-time-jsptags	1.1.1	*3
日付操作	org.jadira.usertype	usertype.core	3.0.0.GA	*1
日付操作	org.jadira.usertype	usertype.spi	3.0.0.GA	*1
コネクションプール	commons-dbcp	commons-dbcp	1.2.2.patch_DBCP264_DBCP372	*2
コネクションプール	commons-pool	commons-pool	1.6	*3
Tiles	commons-digester	commons-digester	2	*3
Tiles	org.apache.tiles	tiles-api	2.2.2	*3
Tiles	org.apache.tiles	tiles-core	2.2.2	*3
Tiles	org.apache.tiles	tiles-jsp	2.2.2	*3
Tiles	org.apache.tiles	tiles-servlet	2.2.2	*3
Tiles	org.apache.tiles	tiles-template	2.2.2	*3
ユーティリティ	com.google.guava	guava	13.0.1	
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	2.4	*3
サーブレット	javax.servlet	jstl	1.2	

1. 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
2. データアクセスに、MyBatis2 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 Global Framework 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

2.1.4 共通ライブラリの構成要素

共通ライブラリは、TERASOLUNA Global Framework が含む Spring Ecosystem や、その他依存ライブラリでは足りない +α 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 Global Framework 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。

項番	プロジェクト名	概要	Java ソースコード有無
(1)	terasoluna-gfw-common	Web に限らず、汎用的に使用できる機能	有
(2)	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群	有
(3)	terasoluna-gfw-jpa	JPA を使用する場合の、依存関係定義	無
(4)	terasoluna-gfw-mybatis2	MyBatis2 を使用する場合の、依存関係定義	無
(5)	terasoluna-gfw-security-core	Spring Security を使用する場合の、依存関係定義 (Web 以外)	無
(6)	terasoluna-gfw-security-web	Spring Security を使用する場合の依存関係定義 (Web 関連)、および Spring Security の拡張	有

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。

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	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-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。
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

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。
認証	リダイレクト先の指定が可能な認証成功ハンドラ	認証が成功した際に、Web アプリケーション内の任意のパスにリダイレクトするためのハンドラクラスを提供する。
CSRF 対策	CSRF(Cross site request forgeries) チェック	CSRF を使用した攻撃から 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。(Spring Security 3.2 を導入するまでの暫定措置)

2.2 Spring MVC アーキテクチャ概要

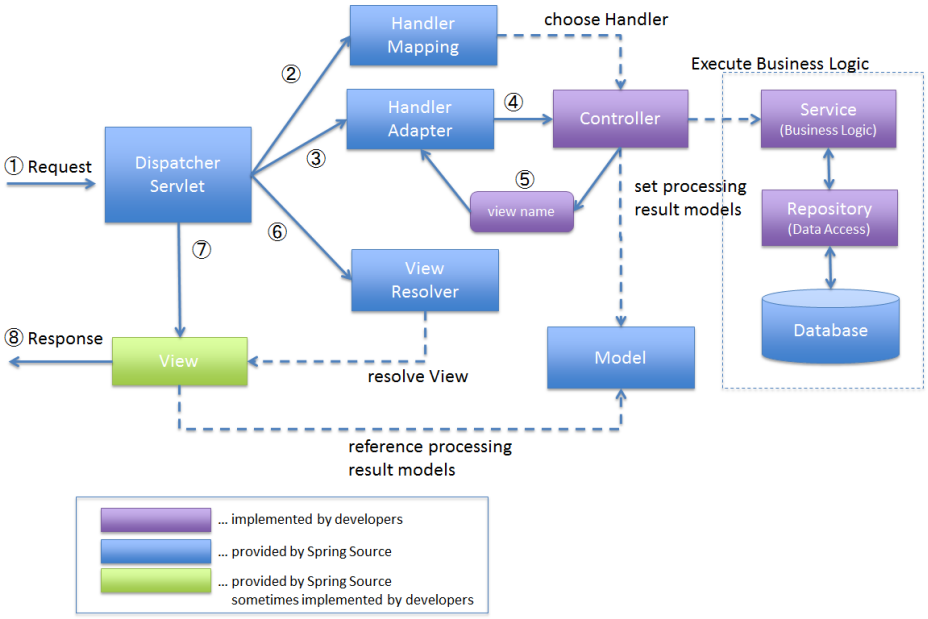
Spring MVC は、公式で以下のように説明されている。

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。

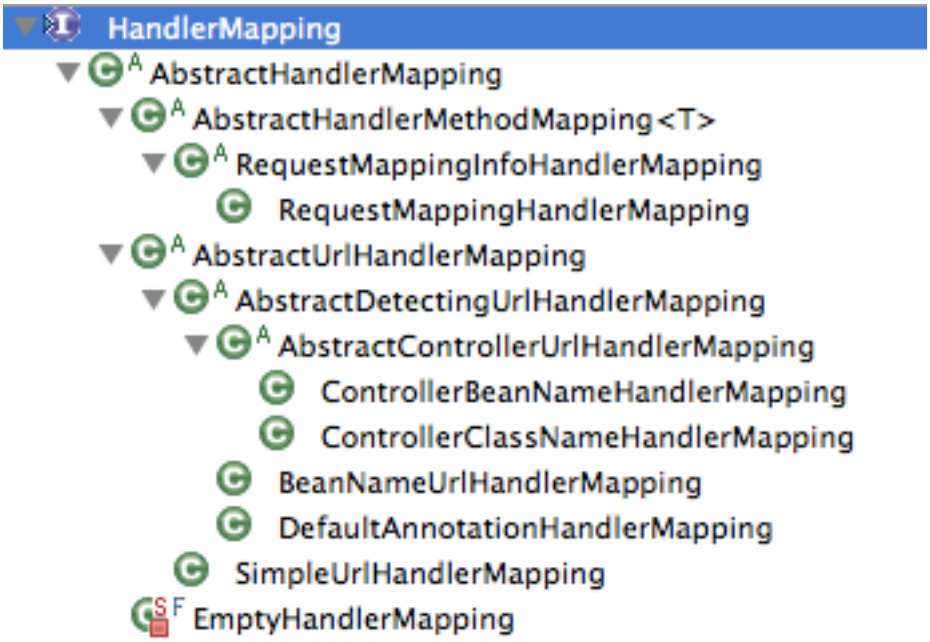
5. Controller は、ビジネスロジックを実行し、処理結果を Model に設定し、ビューの論理名を HandlerAdapter に返却する。
6. DispatcherServlet は、ビュー名に対応する View の解決を、ViewResolver に委譲する。ViewResolver は、ビュー名にマッピングされている View を返却する。
7. DispatcherServlet は、返却された View にレンダリング処理を委譲する。
8. View は、Model の持つ情報をレンダリングしてレスポンスを返却する。

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。

Implementaion 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。

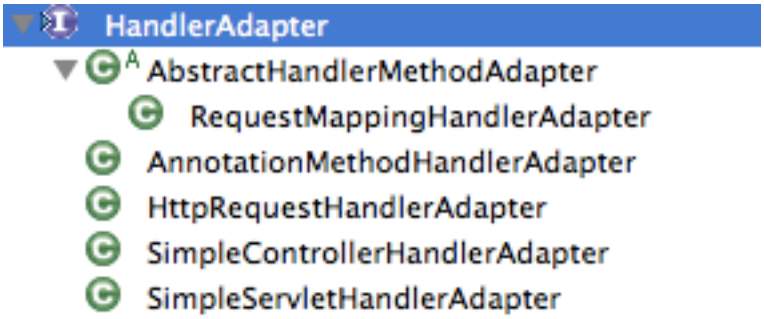
Spring3.1 からは、RequestMappingHandlerMapping は、DispatcherServlet が読み込む Bean 定義ファイルに、

<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。)

Implementaion 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 Spring3.1 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。

Implementaion 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.tiles2.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。

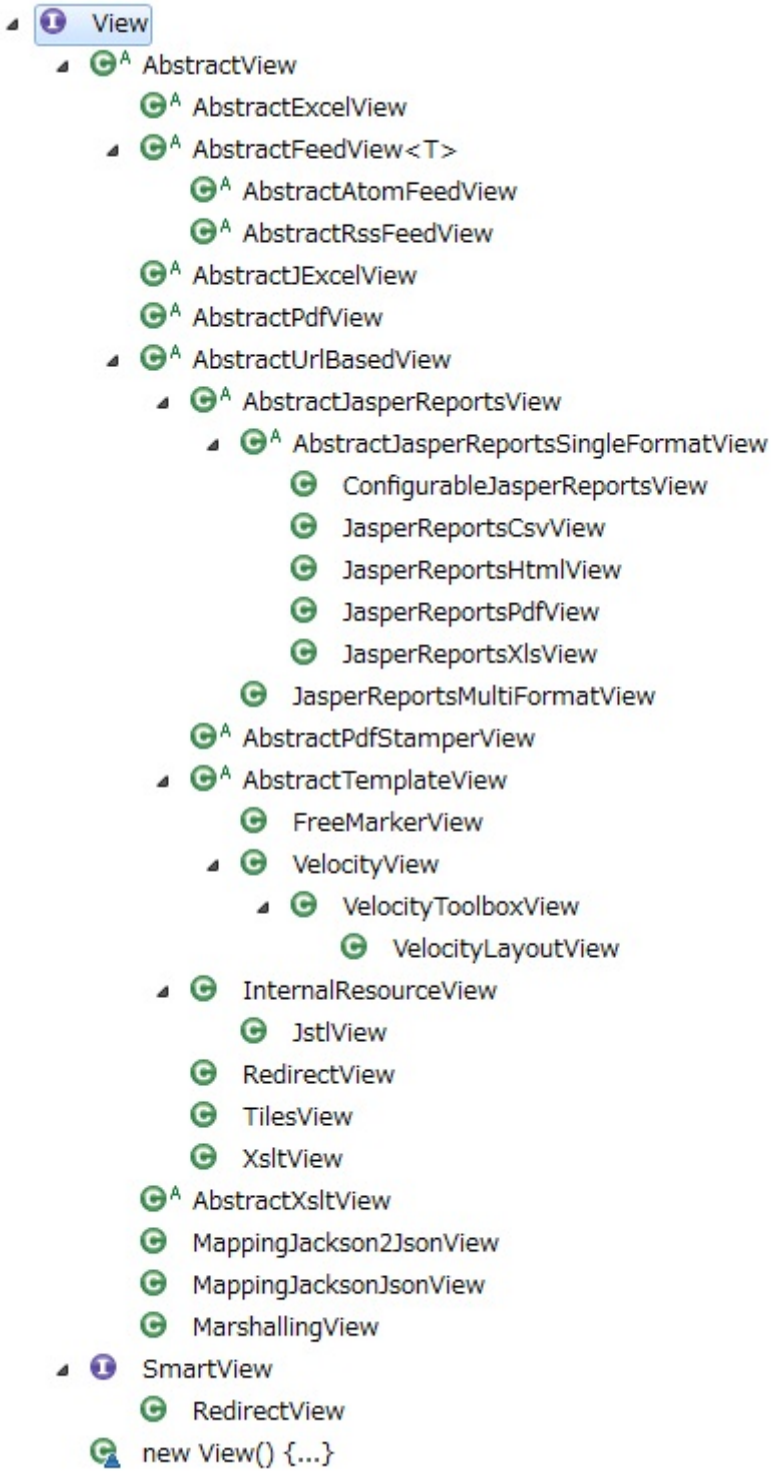
複数の種類の `View` を扱う場合、`ViewResolver` の定義が複数必要となるケースがある。
複数の `ViewResolver` を使う代表的な例として、ファイルのダウンロード処理が存在する画面アプリケーションが挙げられる。
画面 (JSP) は、`InternalResourceViewResolver` で `View` を解決し、
ファイルダウンロードは、`BeanNameViewResolver` などを使って `View` を解決する。
詳細はファイルダウンロードを参照されたい。

Implementaion 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


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

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。

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。)

Product	Version
JDK	1.7.0_55
Spring Tool Suite (STS)	3.5.0.RELEASE
VMware vFabric tc Server Developer Edition	2.9
Fire Fox	ESR 24

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。

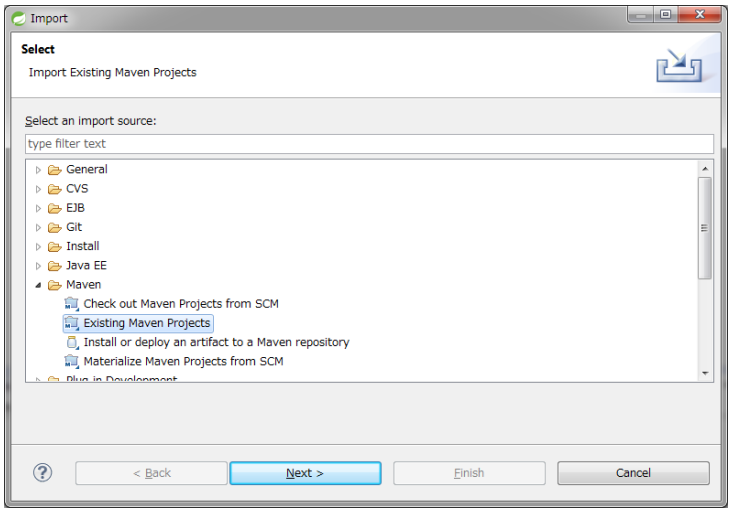
```
mvn archetype:generate -B^
 -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-releases
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype^
 -DarchetypeVersion=1.0.4.RELEASE^
 -DgroupId=com.example.helloworld^
 -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。

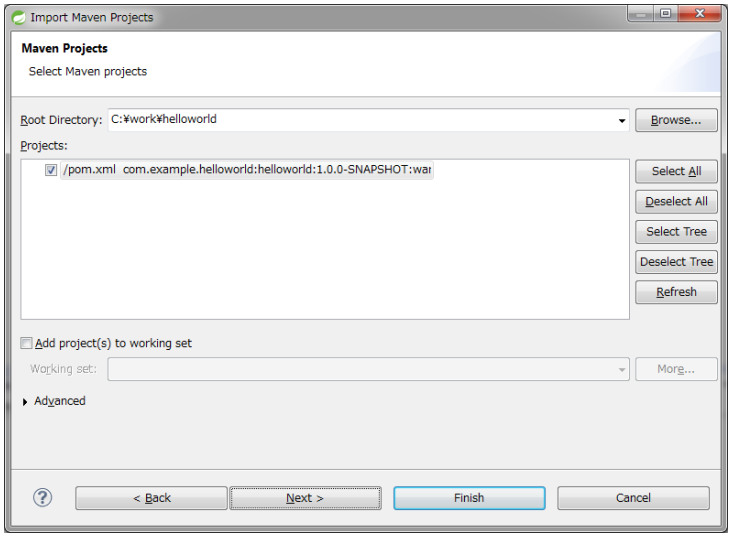
```
C:\work>mvn archetype:generate -B^
More?  -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-re
More?  -DarchetypeGroupId=org.terasoluna.gfw.blank^
More?  -DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype^
More?  -DarchetypeVersion=1.0.4.RELEASE^
More?  -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:2.2:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] Archetype repository missing. Using the one from [org.terasoluna.gfw.blank:terasoluna-gfw-w
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: terasoluna-gfw-web-blank-ar
[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: 6.047s
[INFO] Finished at: Fri Aug 22 14:51:57 JST 2014
[INFO] Final Memory: 14M/182M
[INFO] ------------------------------------------------------------------------
C:\work>
```

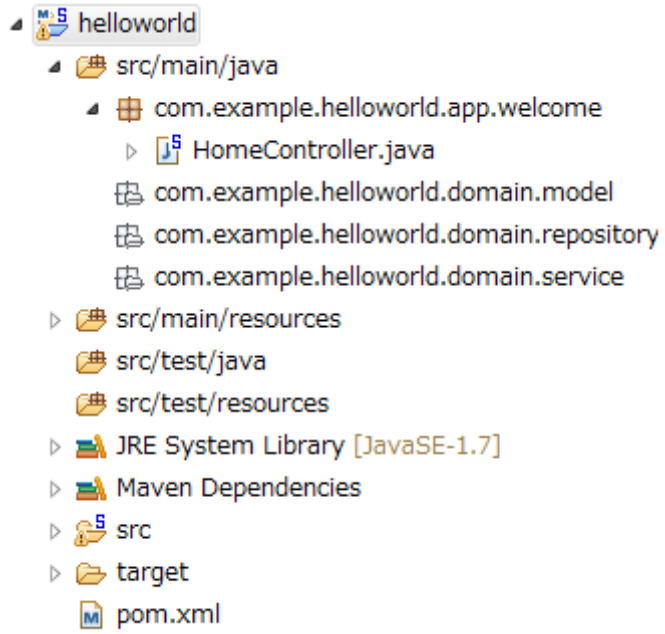
STS のメニューから、[File] -> [Import] -> [Maven] -> [Existing Maven Projects] -> [Next] を選択し、archetype で作成したプロジェクトを選択する。



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.springframewo
    xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:util="http://www.springframework.
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/
```

```
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/sp
        http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spri
        http://www.springframework.org/schema/context http://www.springframework.org/schema/contex
        http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/springc

    <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" />
        </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/**" />
            <mvc:exclude-mapping path="/**/*.html" />
            <bean
                class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor" />
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <mvc:exclude-mapping path="/**/*.html" />
            <bean
                class="org.terasoluna.gfw.web.token.transaction.TransactionTokenInterceptor" />
        </mvc:interceptor>
        <!--  REMOVE THIS LINE IF YOU USE JPA
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <mvc:exclude-mapping path="/**/*.html" />
            <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-
    <!-- Settings View Resolver. -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

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

    <!-- Setting Exception Handling. -->
    <!-- Exception Resolver. -->
    <bean 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/transactionToken
                <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="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.resolve
    </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)	View の Resolver を指定し、View 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.HomeController) を以下に示す。

```
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 HomeController {

    private static final Logger logger = LoggerFactory
            .getLogger(HomeController.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 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/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`) を以下に示す。

```
<!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” -> “VMware vFabric tc Server Developer Edition v2.9” -> “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。

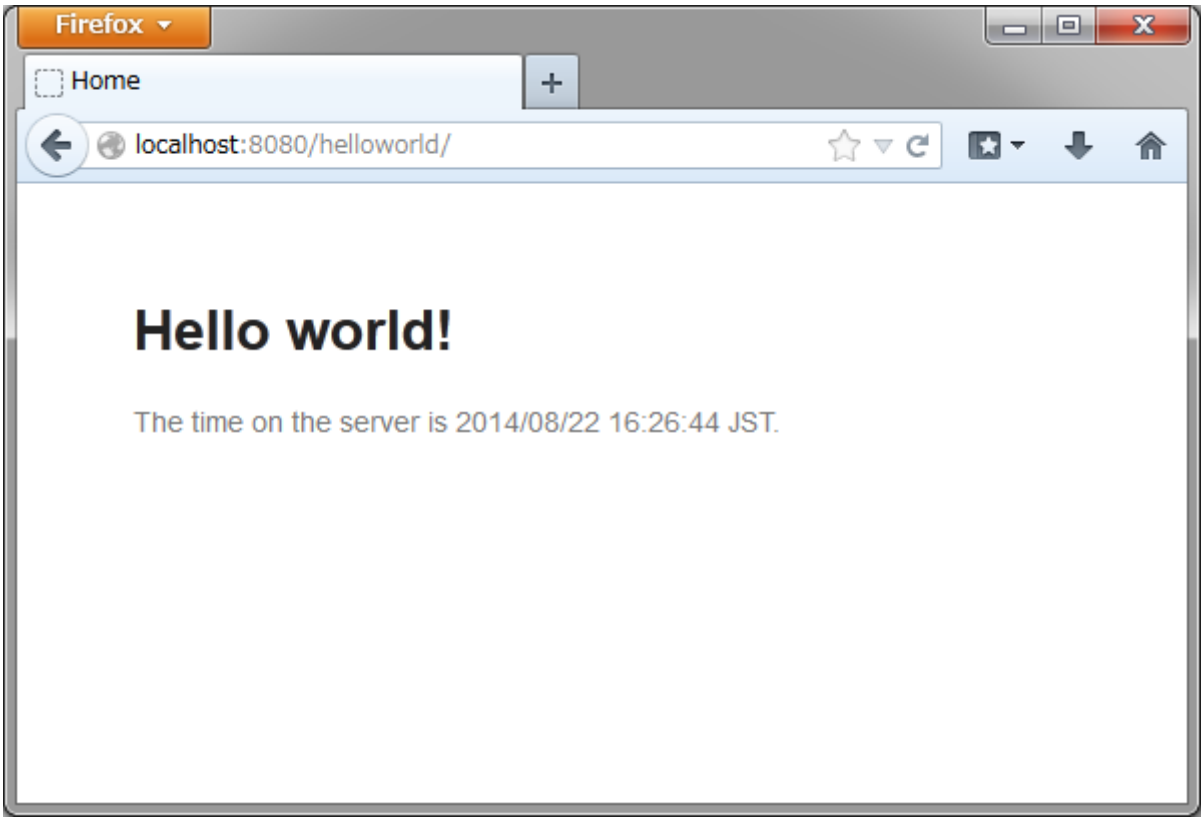
2.3.4 エコーアプリケーションの作成

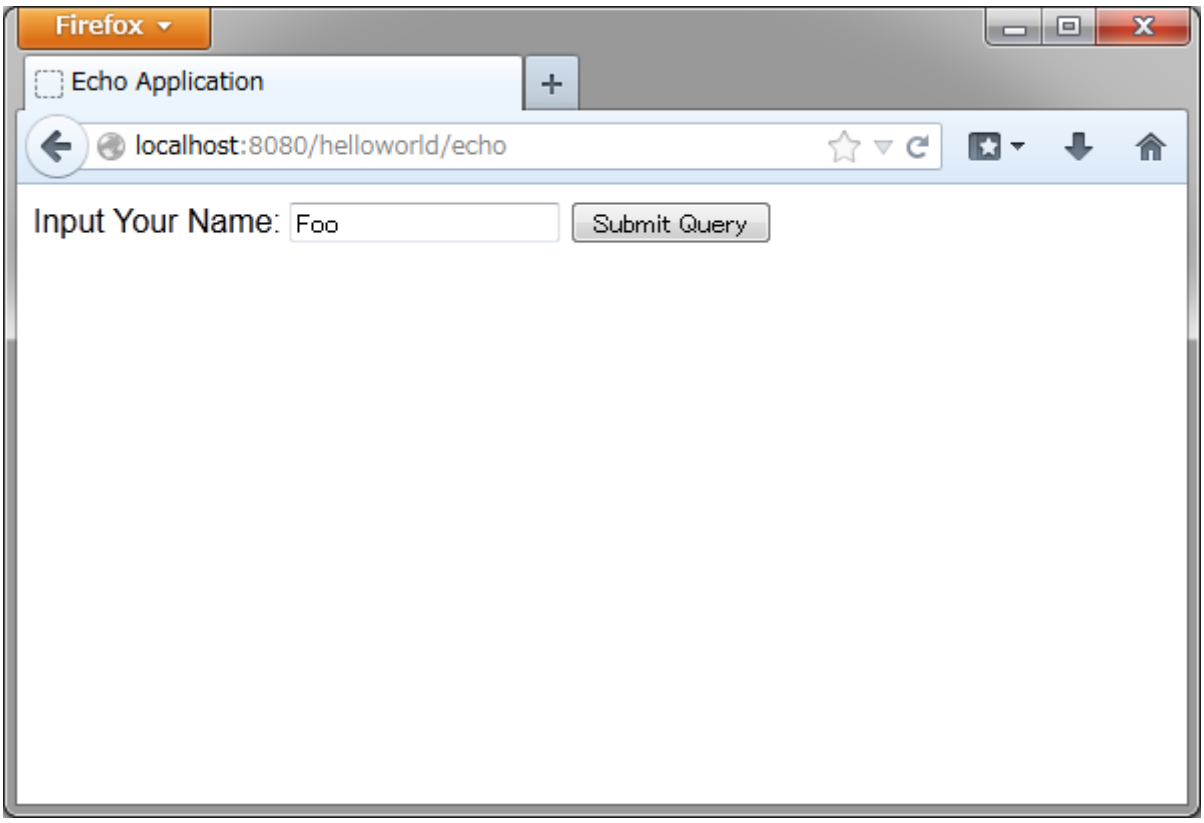
続いて、簡単なアプリケーションを作成する。作成するのは、次の図のようなテキストフィールドに、名前を入力するとメッセージを表示する、いわゆるエコーアプリケーションである。

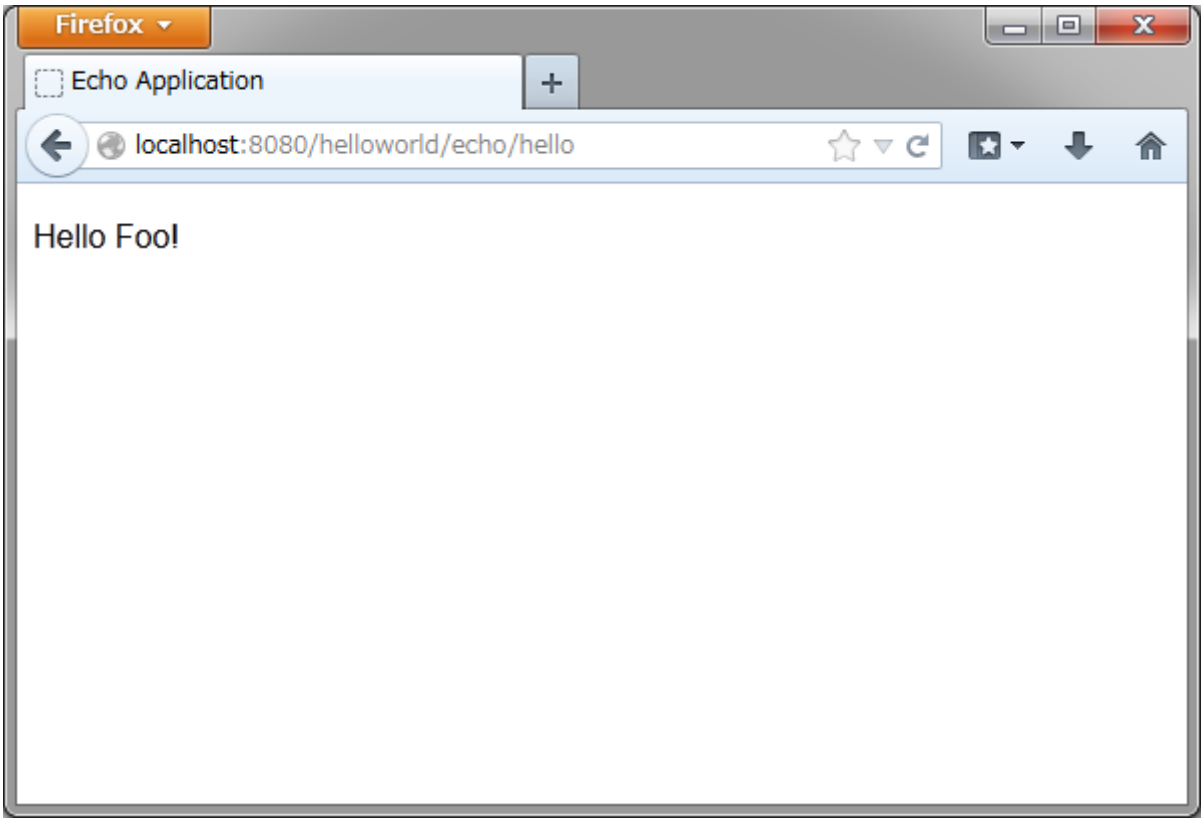
フォームオブジェクトの作成

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。


Firefox
Home
localhost:8080/helloworld/
Hello world!
The time on the server is 2014/08/22 16:26:44 JST.


Firefox
Echo Application
localhost:8080/helloworld/echo
Input Your Name:
Foo
Submit Query




```
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 フォールドに、入力チェックルールを指定するアノーテションを付与する。

```
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。

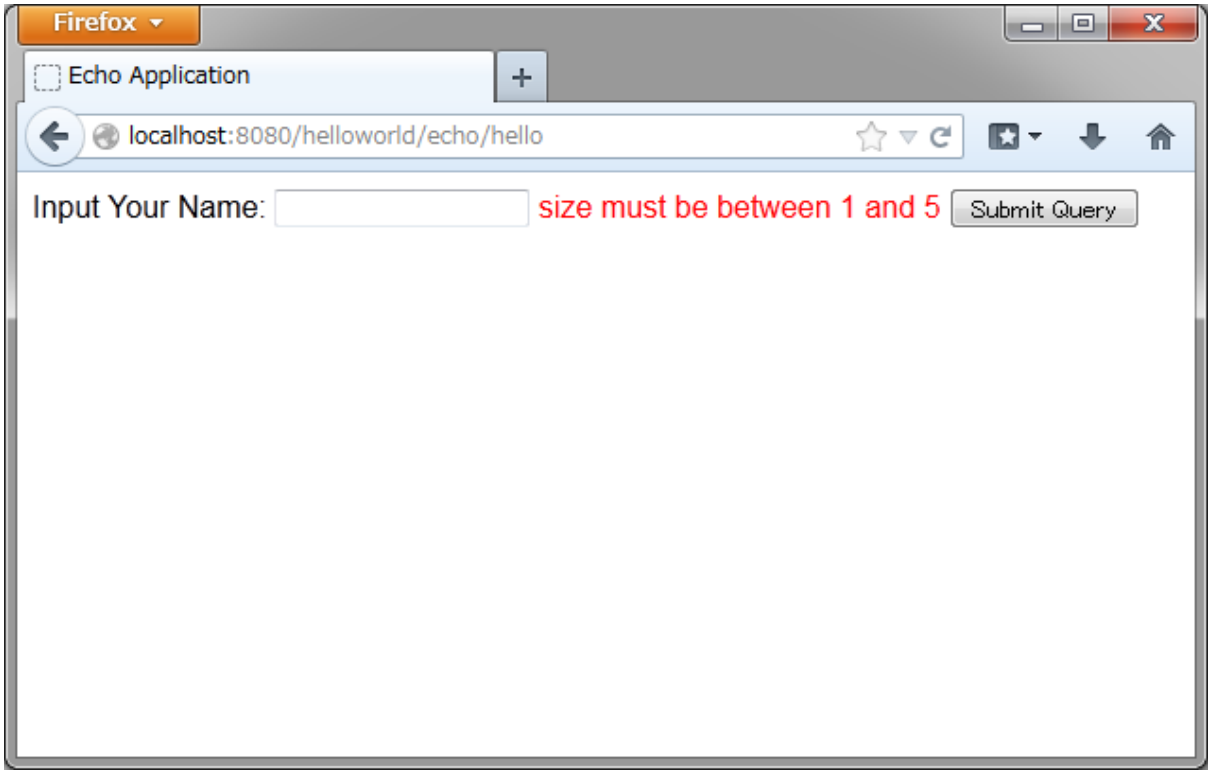
```
<!DOCTYPE html>
<html>
<head>
<title>Echo Application</title>
</head>
<body>
  <form:form modelAttribute="echoForm" action="${pageContext.request.contextPath}/echo/hello">
    <form:label path="name">Input Your Name:</form:label>
    <form:input path="name" />
    <form:errors path="name" cssStyle="color:red"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

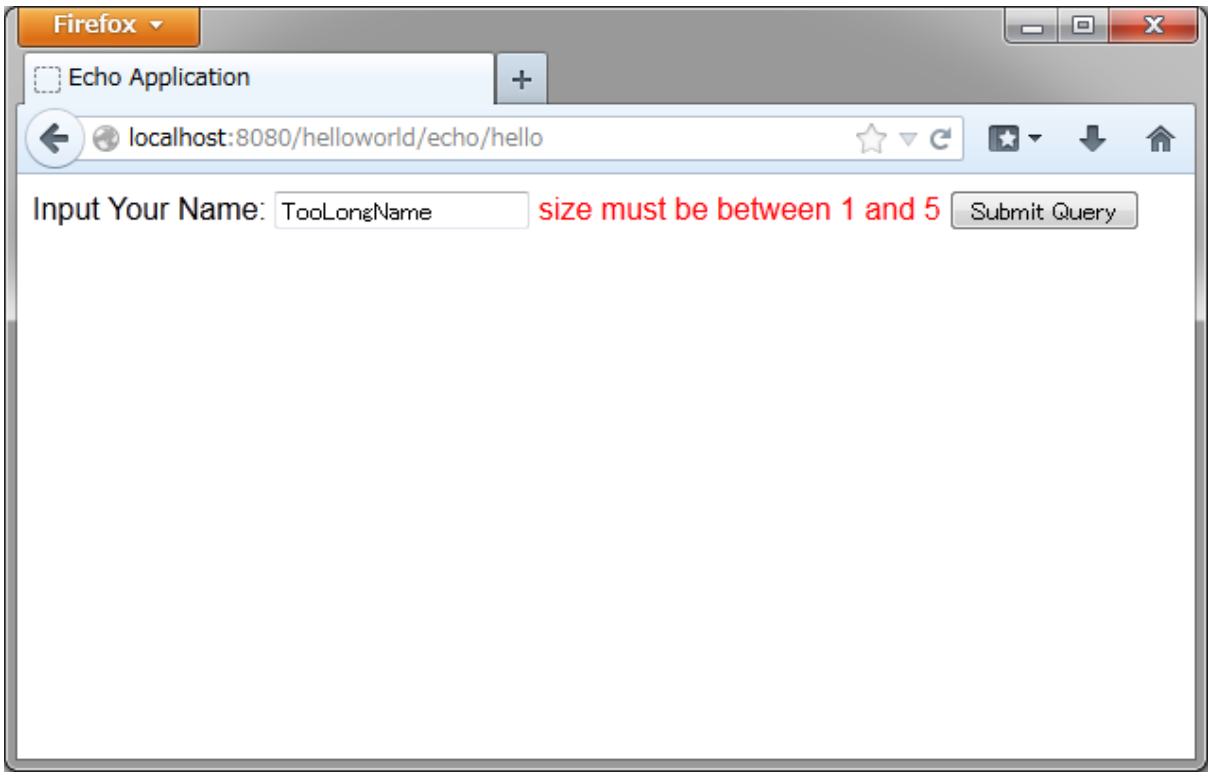
- 名前を空にして送信した場合
- 5 文字より大きいサイズで送信した場合



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. SpringMVC 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。

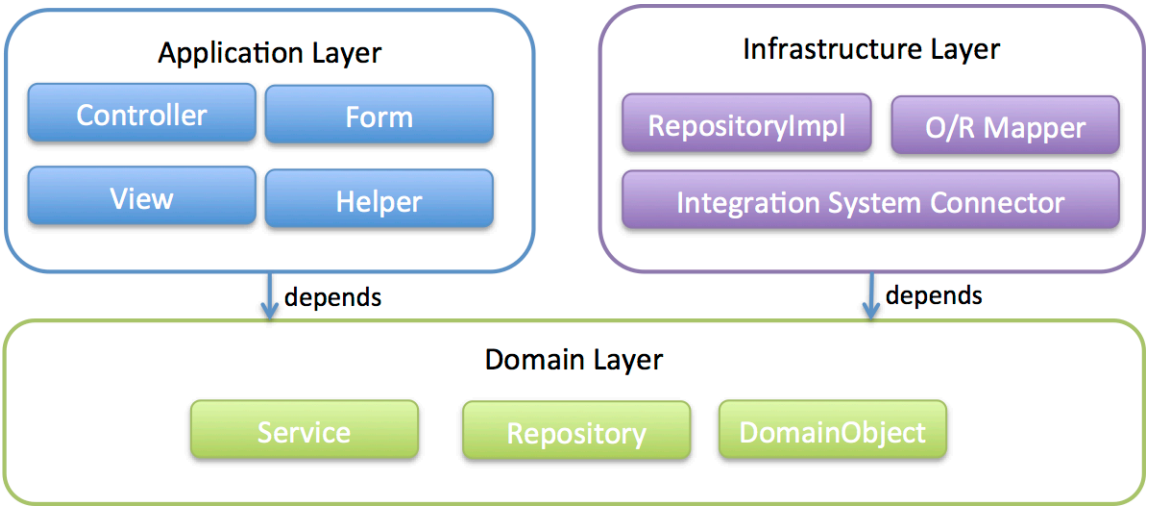
上記の内容が理解できていない場合は、もう一度、本節を読み、環境構築から始めて、進めていくことで理解が深まる。

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の実行する)

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: Spring Data JPA や MyBatis3 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。

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

具体的には、

- JPA を用いる場合は、`EntityManager`
- MyBatis3 を用いる場合は、Mapper インタフェースや `SqlSession`
- MyBatis2(TERASOLUNA DAO) を用いる場合は、`QueryDAO` や `UpdateDAO`
- 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

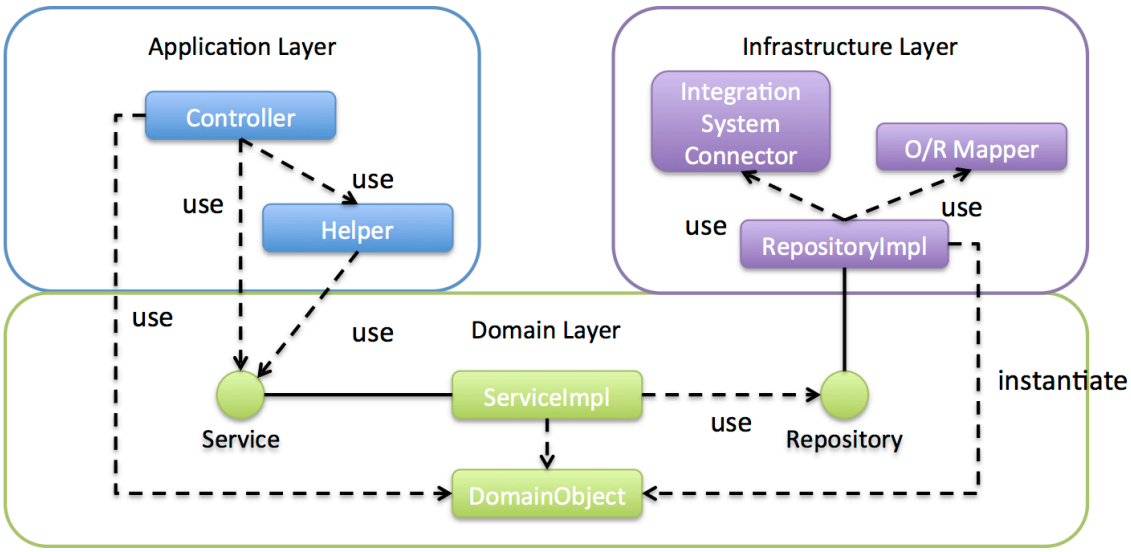
冒頭で説明したとおり、ドメイン層がコアとなり、アプリケーション層、インフラストラクチャ層がそれに依存する形となる。

本ガイドラインでは、実装技術として、

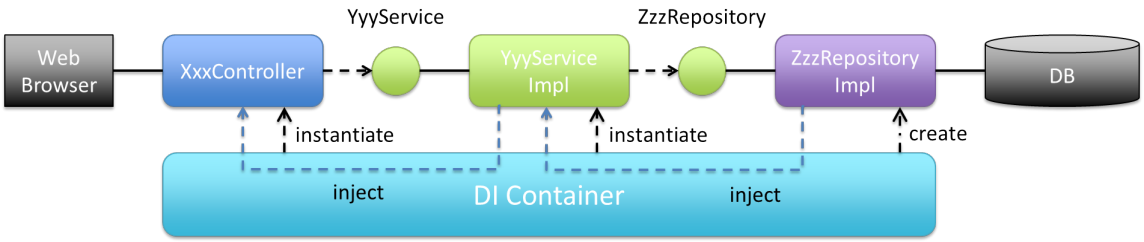
- アプリケーション層に Spring MVC
- インフラストラクチャ層に Spring Data JPA, MyBatis

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。



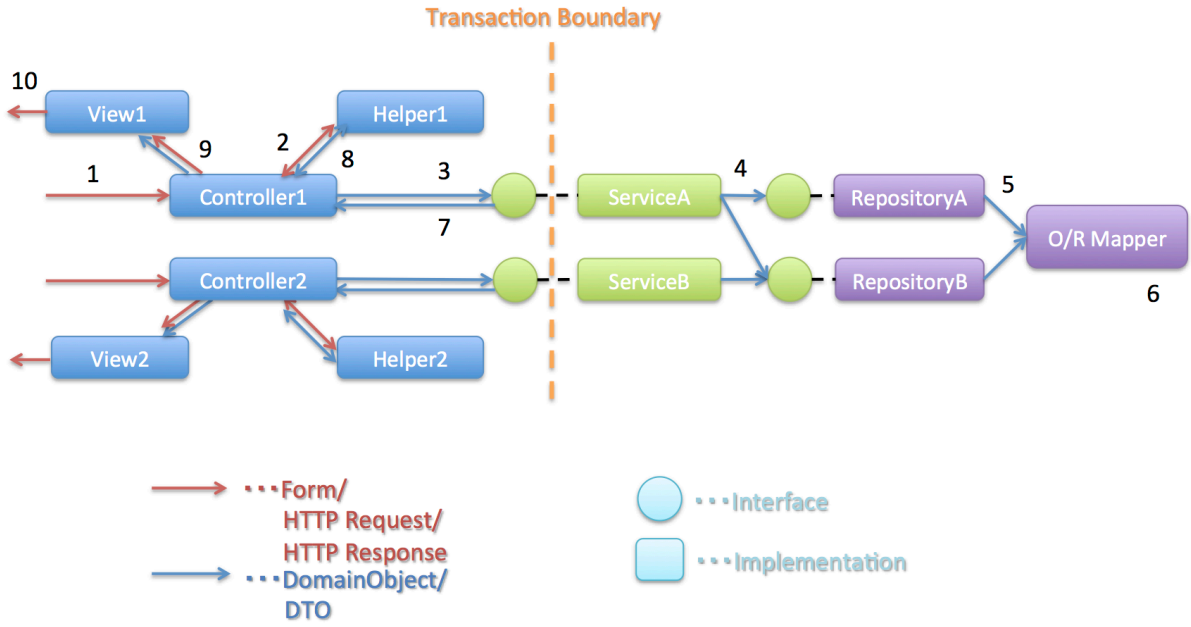
各レイヤのオブジェクトの依存関係は、DI コンテナによって解決される。



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。

TABLE 2.2　**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。

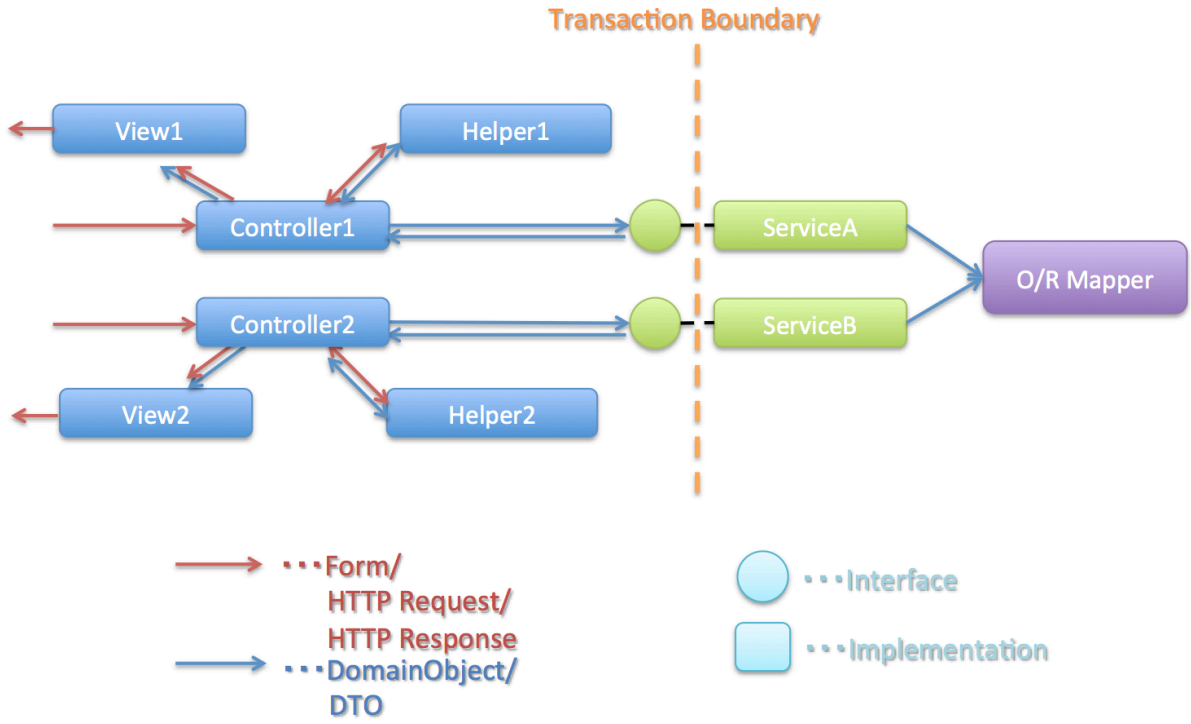


TABLE 2.3 コンポーネント間の呼び出し可否 (without Repository)

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)	SpringMVC 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。

環境依存に関するファイルを別プロジェクトにする意義については、環境依存性の排除を参照されたい。

第 3 章

チュートリアル (Todo アプリケーション)

3.1 はじめに

3.1.1 このチュートリアルで学ぶこと

- TERASOLUNA Global Framework による基本的なアプリケーションの開発方法および Eclipse プロジェクトの構築方法
- TERASOLUNA Global Framework の アプリケーションのレイヤ化 に従った開発方法

3.1.2 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

3.1.3 検証環境

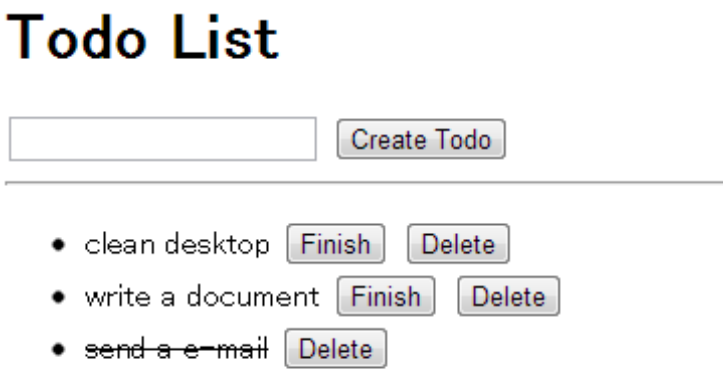
このチュートリアルは以下の環境で動作確認している。他の環境で実施する際は本書をベースに適宜読み替えて設定していくこと。

種別	名前
OS	Windows7 64bit
JVM	Java 1.7
IDE	Spring Tool Suite Version: 3.5.0.RELEASE Build Id: 201404011654 (以下 STS) Build Maven 3.0.4 (STS 付属)
Application Server	VMWare vFabric tc Server Developer Edition v2.9 (STS 付属)
Web Browser	Google Chrome 27.0.1453.94 m

3.2 作成するアプリケーションの説明

3.2.1 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。



3.2.2 アプリケーションの業務要件

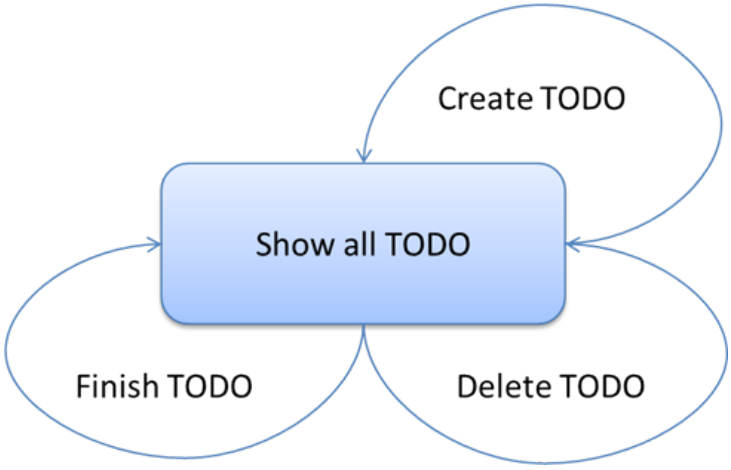
アプリケーションの業務要件は、以下の通りとする。

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。

3.2.3 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	説明
1	Show all TODO	GET	/todo/list	
2	Create TODO	POST	/todo/create	作成完了後 1 へリダイレクト
3	Finish TODO	POST	/todo/finish	作成完了後 1 へリダイレクト
4	Delete TODO	POST	/todo/delete	作成完了後 1 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

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

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
- **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

- 該当する TODO が存在しない場合はエラーコード E404 でビジネス例外をスローする

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 でビジネス例外をスローする

3.2.4 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

3.3 環境構築

3.3.1 プロジェクトの作成

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 Map を使ったインメモリ実装の RepositoryImpl
- Spring Data JPA の使用してデータベースにアクセスする RepositoryImpl
- TERASOLUNA DAO(MyBatis2) 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。

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 上にブランクプロジェクトを作成する手順となっている。

- データベースを使用せず 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、O/R Mapper に依存しないブランクプロジェクトを作成するために、コマンドプロンプトで以下のコマンドを実行する。

```
mvn archetype:generate -B^
 -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-rel
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype^
 -DarchetypeVersion=1.0.4.RELEASE^
 -DgroupId=todo^
 -DartifactId=todo^
 -Dversion=1.0.0-SNAPSHOT
```

- Spring Data JPA の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、JPA 用のブランクプロジェクトを作成するために、コマンドプロンプトで以下のコマンドを実行する。

```
mvn archetype:generate -B^
 -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-rel
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-jpa-archetype^
 -DarchetypeVersion=1.0.4.RELEASE^
 -DgroupId=todo^
 -DartifactId=todo^
 -Dversion=1.0.0-SNAPSHOT
```

- TERASOLUNA DAO(MyBatis2) 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、TERASOLUNA DAO(MyBatis2) 用のブランクプロジェクトを作成するために、コマンドプロンプトで以下のコマンドを実行する。

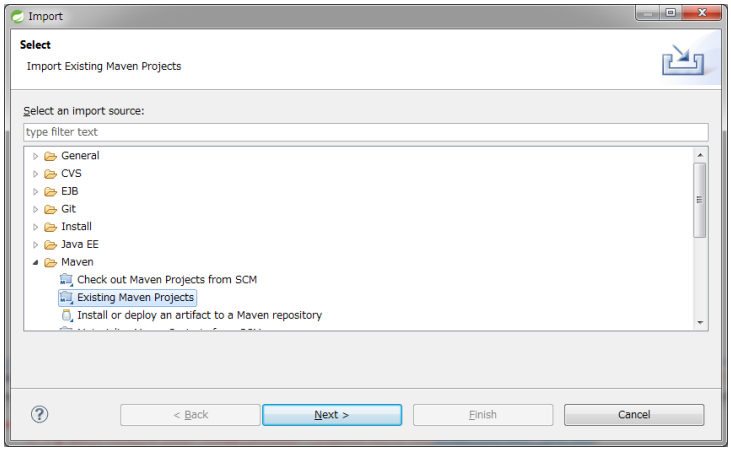
```
mvn archetype:generate -B^
 -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-rel
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-mybatis2-archetype^
 -DarchetypeVersion=1.0.4.RELEASE^
 -DgroupId=todo^
 -DartifactId=todo^
 -Dversion=1.0.0-SNAPSHOT
```

コンソール上に以下のようなログが表示されれば、ブランクプロジェクトの作成は成功となる。

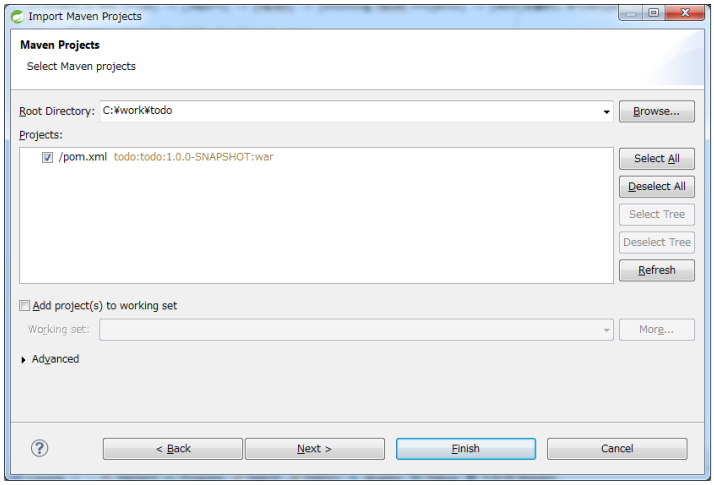
```
C:\work>mvn archetype:generate -B^
More?  -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-g
More?  -DarchetypeGroupId=org.terasoluna.gfw.blank^
More?  -DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype^
More?  -DarchetypeVersion=1.0.4.RELEASE^
More?  -DgroupId=todo^
More?  -DartifactId=todo^
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:2.2:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] Archetype repository missing. Using the one from [org.terasoluna.gfw.blank:terasoluna
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: terasoluna-gfw-web-bl
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: todo
[INFO] Parameter: artifactId, Value: todo
[INFO] Parameter: version, Value: 1.0.0-SNAPSHOT
[INFO] Parameter: package, Value: todo
[INFO] Parameter: packageInPathFormat, Value: todo
[INFO] Parameter: package, Value: todo
[INFO] Parameter: version, Value: 1.0.0-SNAPSHOT
[INFO] Parameter: groupId, Value: todo
[INFO] Parameter: artifactId, Value: todo
[INFO] project created from Archetype in dir: C:\work\todo
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12.084s
[INFO] Finished at: Mon Aug 25 18:59:57 JST 2014
[INFO] Final Memory: 9M/23M
[INFO] ------------------------------------------------------------------------
C:\work>
```

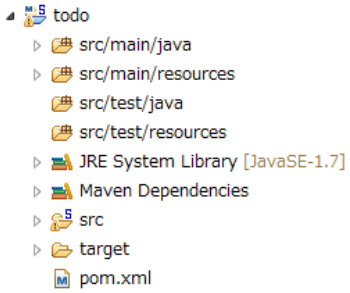
STS のメニューから、[File] -> [Import] -> [Maven] -> [Existing Maven Projects] -> [Next] を選択し、archetype で作成したプロジェクトを選択する。



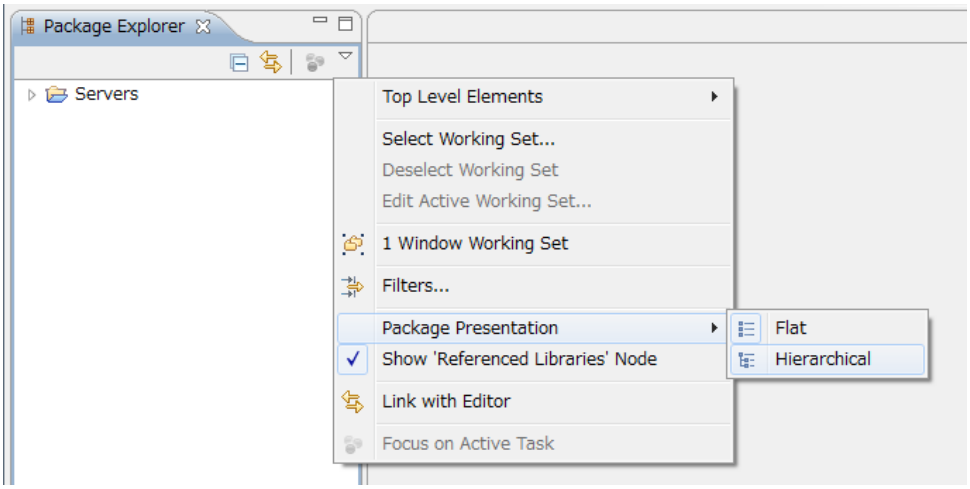
Root Directory に `C:\work\todo` を設定し、Projects に todo の pom.xml が選択された状態で、 [Finish] を押下する。



Package Explorer に、次のようなプロジェクトが生成される (**要インターネット接続**)。



注釈: パッケージ構成上、Package Presentaion を Hierarchical にしたほうが見通しがよい。



注釈: 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る必要がある。

```
mvn archetype:generate -B\
 -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-
 -DarchetypeGroupId=org.terasoluna.gfw.blank\
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype\
 -DarchetypeVersion=1.0.4.RELEASE\
 -DgroupId=todo\
 -DartifactId=todo\
 -Dversion=1.0.0-SNAPSHOT
```

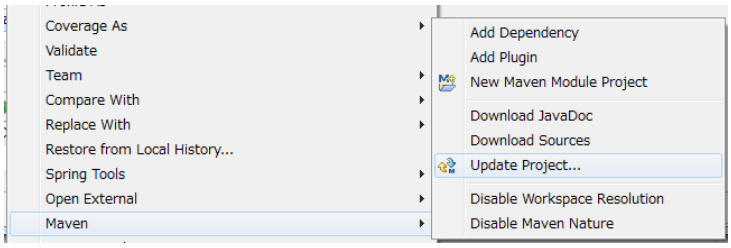
3.3.2 Maven の設定

チュートリアルで必要となるライブラリへの依存関係の設定などは、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。
そのため、設定の追加や変更は不要である。

注釈: Proxy サーバーを介してインターネットアクセスする必要がある場合は、<HOME>/.m2/settings.xml に以下のような設定を行う必要がある。(Windows7 の場合 C:\Users\<YourName>\.m2\settings.xml)

```
<settings>
  <proxies>
    <proxy>
      <active>true</active>
      <protocol>[Proxy Server Protocol (http)]</protocol>
      <port>[Proxy Server Port]</port>
      <host>[Proxy Server Host]</host>
      <username>[Username]</username>
      <password>[Password]</password>
    </proxy>
  </proxies>
</settings>
```

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。



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>
    <version>1.3.172</version>
    <scope>compile</scope>
</dependency>
```

3.3.3 プロジェクト構成

チュートリアルで作成していくプロジェクトの構成について、以下に示す。

```
src
  └main
      ├java
      |  └todo
      |     ├ app ... アプリケーション層を格納
      |     |   └todo ... todo 管理業務に関わるクラスを格納
      |     └domain ... ドメイン層を格納
      |         ├model ... Domain Object を格納
      |         ├repository ... Repository を格納
      |         |   └todo ... Todo 用 Repository
      |         └service ... Service を格納
      |             └todo ... TODO 業務 Service
      ├resources
      |  └META-INF
      |      └spring ... spring 関連の設定ファイルを格納
      └wepapp
          └WEB-INF
              └views ... jsp を格納
```

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。

3.3.4 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。

本節では、アプリケーションを動かすためにどのような設定ファイルが必要なのかを理解するために、 設定ファイルの確認をしていく。

注釈: まず、手を動かして Todo アプリを作成したい場合は、本節を読み飛ばしてもよいが、Todo アプリを作成した後に必ず一読して頂きたい。

web.xml の確認

`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/XMLSche
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
    version="3.0">
    <!-- (2) -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-clas
    </listener>
    <listener>
        <listener-class>org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener</list
    </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>

        <!-- (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>
        </session-config>

    </web-app>
```

項番	説明
(1)	Servlet3.0 を使用するための宣言。
(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 `ApplicatnionContext` の子として作成する。 (2) で作成した `ApplicatnionContext` 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、

インクルード JSP の確認

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/functions" prefix="f"%>
<%@ taglib uri="http://terasoluna.org/tags" prefix="t"%>
```

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 定義ファイルの確認

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

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 の確認

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。

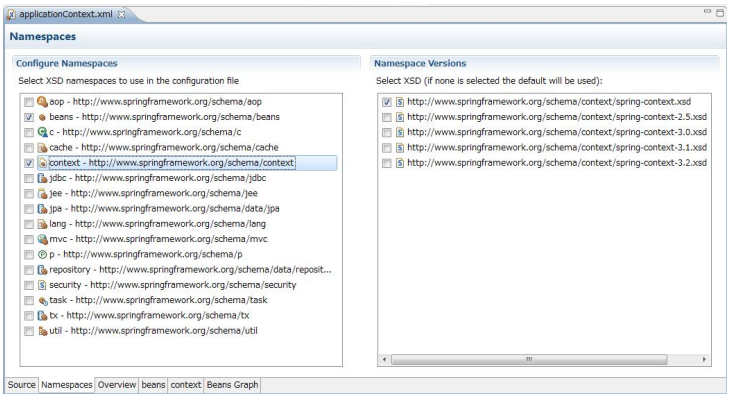
```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springfr
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframewo
            http://www.springframework.org/schema/context http://www.springframework.org/schema/
            http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/

        <!-- (1) -->
        <import resource="classpath:/META-INF/spring/todo-domain.xml" />

        <bean id="passwordEncoder" class="org.springframework.security.crypto.bcrypt.BCryptPassw

        <!-- (2) -->
        <context:property-placeholder
            location="classpath*:/META-INF/spring/*.properties" />

        <!-- (3) -->
        <bean 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>
```

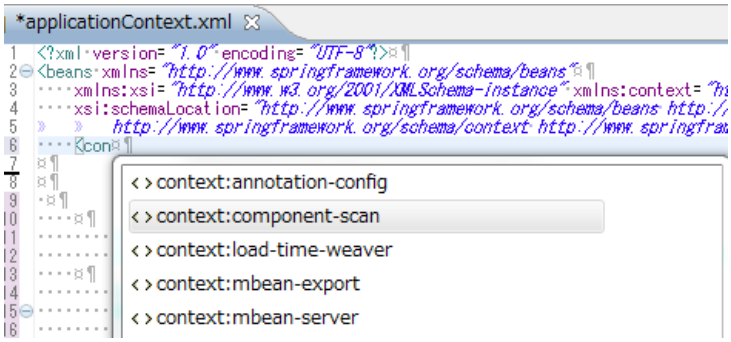
```
        <!-- Filter. -->
        <bean id="exceptionLoggingFilter"
            class="org.terasoluna.gfw.web.exception.ExceptionLoggingFilter" >
            <property name="exceptionLogger" ref="exceptionLogger" />
        </bean>

    </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 Dozer マニュアル を参照されたい。)

注釈: エディタの「Configure Namspecse」タブにて、以下のように「beans」と「context」にチェックを入れると、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 の確認

`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.springfr
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframewo
        http://www.springframework.org/schema/context http://www.springframework.org/schema/
        http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/

    <!-- (1) -->
    <import resource="classpath:META-INF/spring/todo-infra.xml" />

    <!-- (2) -->
    <context:component-scan base-package="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 todo.domain パッケージ配下を component-scan 対象とする。 これにより、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>`タグを設定されている。

```
        <tx:annotation-driven />
```

todo-infra.xml の確認

`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　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 http://www.springframewo
```

```
    </beans>
```

JPA 用のブランクプロジェクトを作成した場合　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.springframew
        xmlns:util="http://www.springframework.org/schema/util"
        xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframewo
            http://www.springframework.org/schema/util http://www.springframework.org/schema/uti
            http://www.springframework.org/schema/data/jpa http://www.springframework.org/schema

        <!-- (1) -->
        <import resource="classpath:/META-INF/spring/todo-env.xml" />

        <!-- (2) -->
        <jpa:repositories base-package="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
            class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
            id="entityManagerFactory">
            <!-- (5) -->
            <property name="packagesToScan" value="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="none" />
                    <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 属性で、対象の Repository を含む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。

TERASOLUNA DAO(MyBatis2) 用のブランクプロジェクトを作成した場合 TERASOLUNA DAO(MyBatis2) 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"
        xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframew

        <!-- (1) -->
        <import resource="classpath:/META-INF/spring/todo-env.xml" />

        <!-- (2) -->
        <bean id="sqlMapClient"
```

```
            class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
            <!-- (3) -->
            <property name="configLocations"
                value="classpath*:/META-INF/mybatis/config/*sqlMapConfig.xml" />
            <!-- (4) -->
            <property name="mappingLocations"
                value="classpath*:/META-INF/mybatis/sql/**/*-sqlmap.xml" />
            <property name="dataSource" ref="dataSource" />
        </bean>

        <!-- (5) -->
        <bean id="queryDAO" class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
            <property name="sqlMapClient" ref="sqlMapClient" />
        </bean>

        <bean id="updateDAO" class="jp.terasoluna.fw.dao.ibatis.UpdateDAOiBatisImpl">
            <property name="sqlMapClient" ref="sqlMapClient" />
        </bean>

        <bean id="spDAO"
            class="jp.terasoluna.fw.dao.ibatis.StoredProcedureDAOiBatisImpl">
            <property name="sqlMapClient" ref="sqlMapClient" />
        </bean>

        <bean id="queryRowHandleDAO"
            class="jp.terasoluna.fw.dao.ibatis.QueryRowHandleDAOiBatisImpl">
            <property name="sqlMapClient" ref="sqlMapClient" />
        </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)	SqlMapClient の定義を行う。
(3)	SqlMap 設定ファイルのパスを設定する。 ここでは、META-INF/mybatis/config 以下の、*sqlMapConfig.xml を読み込む。
(4)	SqlMap ファイルのパスを設定する。 ここでは、META-INF/mybatis/sql 以下の、任意のフォルダの*-sqlmap.xml を読み込む。
(5)	TERASOLUNA DAO の定義を行う。

注釈: `sqlMapConfig.xml` は、MyBatis2 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 sqlMapConfig
            PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
            "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">
<sqlMapConfig>
    <settings useStatementNamespaces="true" />
</sqlMapConfig>
```

`useStatementNamespaces` を `true` にすることで、SQLID にネームスペースを指定することが出来る。

todo-infra.properties の確認

todo-infra.properties には、Todo アプリのインフラストラクチャ層の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 の確認

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。

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。

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。

```
<?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 http://www.springframewo

    <bean id="dateFactory" class="org.terasoluna.gfw.common.date.DefaultDateFactory" />

    <!-- (1) -->
    <bean id="realDataSource" class="org.apache.commons.dbcp.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="maxActive" value="${cp.maxActive}" />
        <property name="maxIdle" value="${cp.maxIdle}" />
        <property name="minIdle" value="${cp.minIdle}" />
        <property name="maxWait" value="${cp.maxWait}" />
    </bean>

    <!-- (2) -->
    <bean id="dataSource" class="net.sf.log4jdbc.Log4jdbcProxyDataSource">
        <constructor-arg index="0" ref="realDataSource" />
    </bean>

    <!-- (3) -->
    <bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
    <!--  REMOVE THIS LINE IF YOU USE MyBatis2
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>
          REMOVE THIS LINE IF YOU USE MyBatis2  -->
</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)	トランザクションマネージャの設定。id は、transactionManager にすること。 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。

注釈: 使用するトランザクションマネージャは、使用する O/R Mapper によって切り替える必要がある。

作成したブランクプロジェクトが、

- JPA 用のブランクプロジェクトを作成した場合は、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`)
- TERASOLUNA DAO(MyBatis2) 用のブランクプロジェクトを作成した場合は、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: JavaEE コンテナ上にアプリケーションをデプロイする場合は、JTA の API を利用してトランザクションを制御するクラス (`org.springframework.transaction.jta.JtaTransactionManager`) を使用したほうがよい。`JtaTransactionManager` を使う場合は、`<tx:jta-transaction-manager />`を指定してトランザクションマネージャの定義を行う。

トランザクションマネージャの設定が、アプリケーションをデプロイする環境によって変わらないプロジェクト (例えば、Tomcat を使用する場合など) は、:file:‘todo-infra.xml‘に定義してもよい。

spring-mvc.xml の確認

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.springfr
    xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:util="http://www.springframe
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/be
        http://www.springframework.org/schema/util http://www.springframework.org/schema/uti
        http://www.springframework.org/schema/context http://www.springframework.org/schema/
        http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/

    <!-- (1) -->
    <context:property-placeholder
        location="classpath*:/META-INF/spring/*.properties" />

    <!-- (2) -->
    <mvc:annotation-driven>
        <mvc:argument-resolvers>
            <bean
                class="org.springframework.data.web.PageableHandlerMethodArgumentResolver" />
        </mvc:argument-resolvers>
    </mvc:annotation-driven>

    <mvc:default-servlet-handler />

    <!-- (3) -->
    <context:component-scan base-package="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/**" />
            <mvc:exclude-mapping path="/**/*.html" />
            <bean
                class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor" />
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <mvc:exclude-mapping path="/**/*.html" />
            <bean
                class="org.terasoluna.gfw.web.token.transaction.TransactionTokenInterceptor"
        </mvc:interceptor>
        <!--  REMOVE THIS LINE IF YOU USE JPA
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <mvc:exclude-mapping path="/**/*.html" />
            <bean
                class="org.springframework.orm.jpa.support.OpenEntityManagerInViewIntercepto
        </mvc:interceptor>
            REMOVE THIS LINE IF YOU USE JPA  -->
    </mvc:interceptors>

    <!-- (7) -->
    <!-- Settings View Resolver. -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

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

    <!-- Setting Exception Handling. -->
    <!-- Exception Resolver. -->
    <bean 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/resourceNotFoundE
                    <entry key="BusinessException" value="common/error/businessError" />
                    <entry key="InvalidTransactionTokenException" value="common/error/transactio
                    <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="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.r
        </aop:config>

    </beans>
```

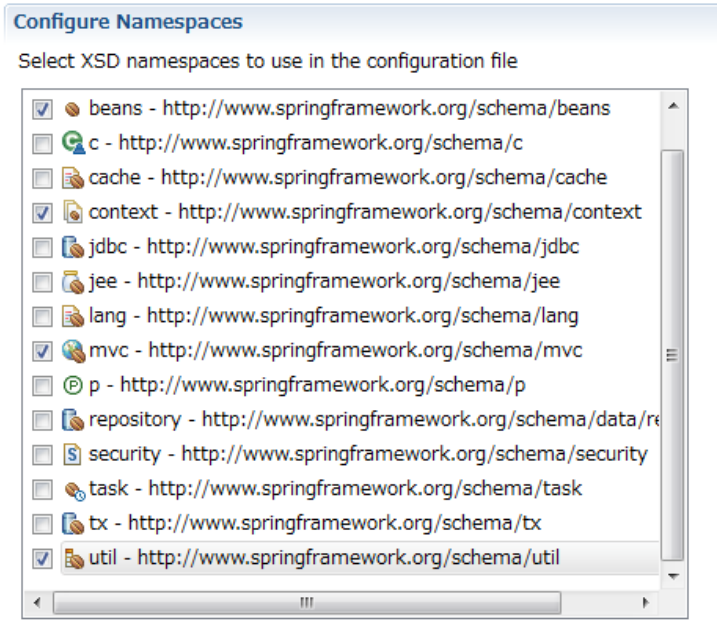
項番	説明
(1)	プロパティファイルの読み込み設定を行う。 src/main/resources/META-INF/spring 直下の任意のプロパティファイルを読み込む。 この設定により、プロパティファイルの値を Bean 定義ファイル内で ${propertyName}形式で埋め込んだり、Java クラスに @Value(“${propertyName}”) でインジェクションすることができる。
(2)	Spring MVC のアノテーションベースのデフォルト設定を行う。
(3)	アプリケーション層のクラスを管理する todo.app パッケージ配下を component-scan 対象とする。
(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>/rerources/css/styles.css に対してリクエストが来た場合、WEB-INF/resources/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 resources/css/style.css を探す。 WEB-INF/resources/css/styles.css が見つからなければ、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 以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 ” 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/**" />
            <mvc:exclude-mapping path="/**/*.html" />
            <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 (Contoller や、View クラス) での Lazy Load が、サポートされる。

注釈: エディタの「Configure Namspecse」タブにて、todo-domain.xml で説明した操作に加え、「mvc」と「util」にもチェックを入れると、「mvc」と「util」ネームスペースについても、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。



logback.xml の確認

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。

```
<!DOCTYPE logback>
<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-Tra
        </encoder>
    </appender>

    <appender name="APPLICATION_LOG_FILE" class="ch.qos.logback.core.rolling.RollingFileAppe
        <file>log/todo-application.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>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-Tra
        </encoder>
    </appender>

    <appender name="MONITORING_LOG_FILE" class="ch.qos.logback.core.rolling.RollingFileAppen
        <file>log/todo-monitoring.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>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:%-5le
        </encoder>
    </appender>

    <!-- Application Loggers -->
    <!-- (2) -->
    <logger name="todo">
        <level value="debug" />
    </logger>
```

```
        <!-- TERASOLUNA -->
        <logger name="org.terasoluna.gfw">
            <level value="debug" />
        </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
            <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 MyBatis2
        <logger name="org.springframework.jdbc.datasource.DataSourceTransactionManager">
            <level value="debug" />
        </logger>
              REMOVE THIS LINE IF YOU USE MyBatis2  -->

        <logger name="jdbc.sqltiming">
            <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)	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>
```

- TERASOLUNA DAO(MyBatis2) 用のブランクプロジェクト

```
<logger name="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <level value="debug" />
</logger>
```

3.3.5 ブランク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、ブランク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/todo/app/welcome/HomeController.java`)は、以下のような実装となっている。

```
package 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 HomeController {

    // (2)
    private static final Logger logger = LoggerFactory
            .getLogger(HomeController.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の生成している。 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。

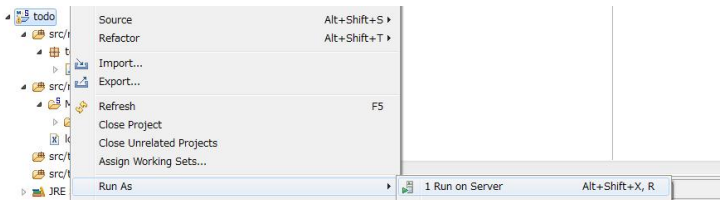
ブランクプロジェクトから提供されている JSP(`src/main/webapp/WEB-INF/views/welcome/home.jsp`) は、以下のような実装となっている。

```
<!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>
            <!-- (1) -->
            <p>The time on the server is ${serverTime}.</p>
        </div>
    </body>
    </html>
```

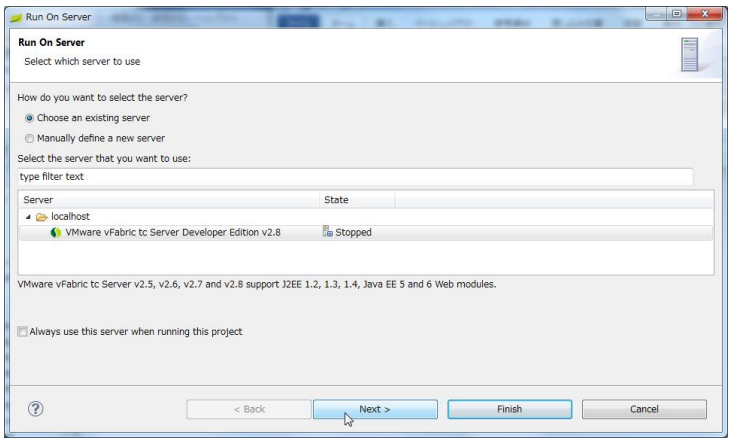
項番	説明
(1)	Controller で Model に設定した ” serverTime ” を表示する。 ここでは、XSS 対策を行っていないが、ユーザの入力値を表示する場合は、`f:h()` 関数を用いて、必ず XSS 対策を行うこと。

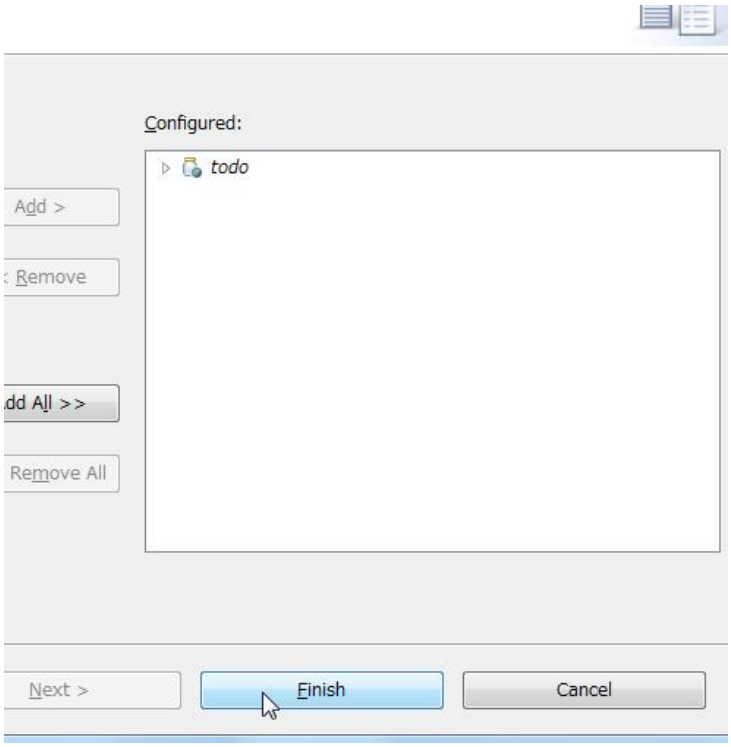
パッケージプロジェクト名 ” todo ” を右クリックして「Run As」->「Run on Server」



実行したい AP サーバー (ここでは VMWare vFabric tc Server Developer Edition v2.9) を選び「Next」をクリック

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。

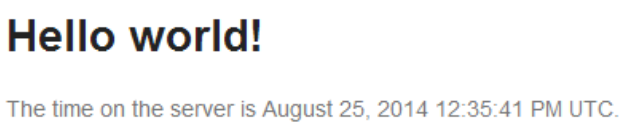




起動すると以下のようなログが出力される。"/"というパスに対してtodo.app.welcome.HomeControllerのhelloメソッドがマッピングされていることが分かる。

```
      date:2014-08-25 12:35:34    thread:localhost-startStop-1    X-Track:    level:INFO  logger:
      date:2014-08-25 12:35:35    thread:localhost-startStop-1    X-Track:    level:INFO  logger:
      date:2014-08-25 12:35:36    thread:localhost-startStop-1    X-Track:    level:INFO  logger:
      date:2014-08-25 12:35:36    thread:localhost-startStop-1    X-Track:    level:INFO  logger:
      date:2014-08-25 12:35:36    thread:localhost-startStop-1    X-Track:    level:INFO  logger:
```

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 `TraceLoggingInterceptor` による TRACE ログと Controller で実装した info ログが出力されていることがわかる。

```
    date:2014-08-25 12:35:41    thread:tomcat-http--3   X-Track:94bc5651406e4333a4b47b2276707b5c
    date:2014-08-25 12:35:41    thread:tomcat-http--3   X-Track:94bc5651406e4333a4b47b2276707b5c
    date:2014-08-25 12:35:41    thread:tomcat-http--3   X-Track:94bc5651406e4333a4b47b2276707b5c
    date:2014-08-25 12:35:41    thread:tomcat-http--3   X-Track:94bc5651406e4333a4b47b2276707b5c
```

> **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。

3.4 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。

本節では、データベースを使用せず 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っているので、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。

3.4.1 ドメイン層の作成

Domain Object の作成

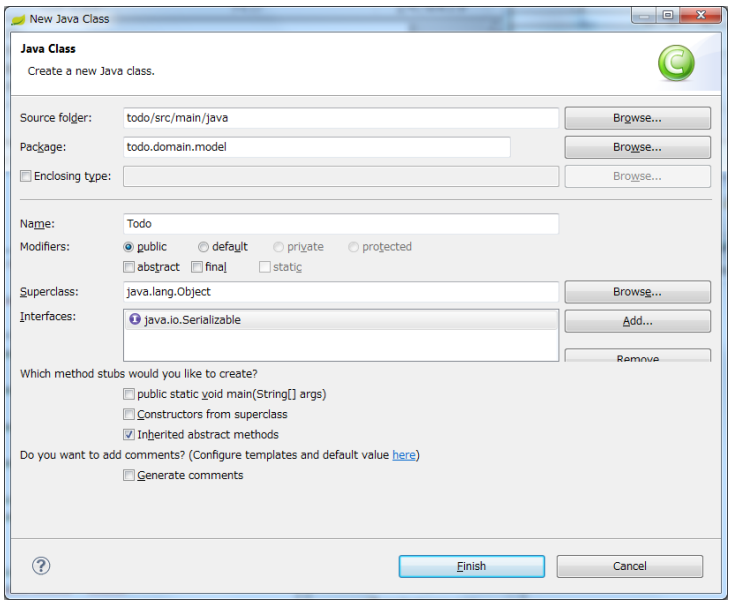
ドメインオブジェクトに必要なプロパティは、

1. ID
2. タイトル
3. 完了フラグ
4. 作成日

である。

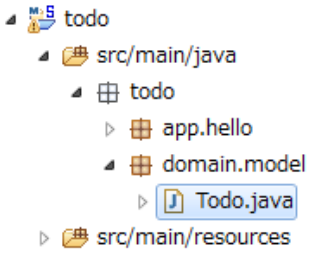
Domain オブジェクトを作成する。
FQCN は、`todo.domain.model.Todo` とする。

```
package 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;
```

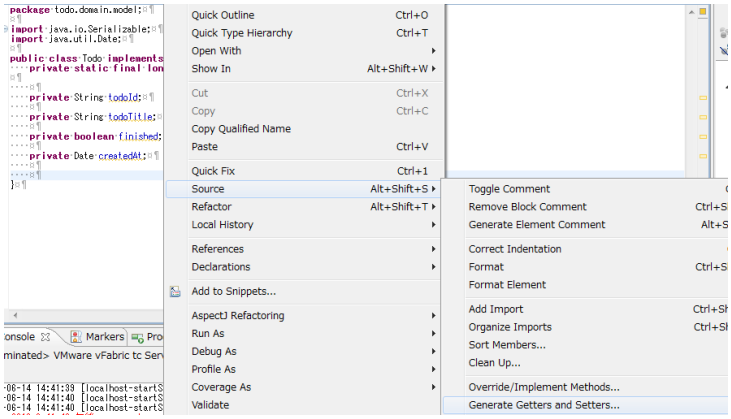



```
    private String todoTitle;

    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;
    }
}
```



注釈: Getter/Setter は自動生成できる。フィールドを定義した後、右クリックで「Source」->「Generate Getter and Setters…」

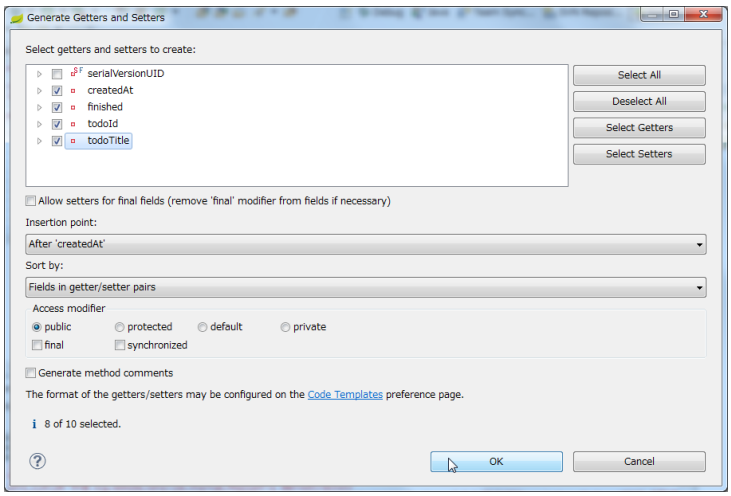


serialVersionUID 以外を選択して「OK」

Repository の作成

今回のアプリケーションで、必要な TODO オブジェクトに対する CRUD 系操作は、

- TODO の 1 件取得



- TODO の全件取得
- TODO の 1 件削除
- TODO の 1 件更新
- 完了済み TODO 件数の取得

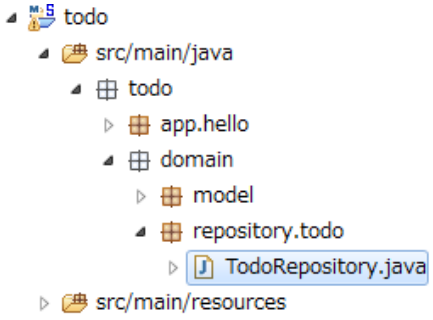
である。

これらの操作を定義するインタフェース TodoRepository を作成する。

FQCN は、`todo.domain.repository.todo.TodoRepository` とする。

```
package todo.domain.repository.todo;

import java.util.Collection;

import todo.domain.model.Todo;

public interface TodoRepository {
    Todo findOne(String todoId);

    Collection<Todo> findAll();

    Todo save(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、「完了済み件数の取得する」メソッド



(`long countFinished()`) ではなく、「完了状態が x 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)

本節では、説明を単純化するため、Map を使ったインメモリ実装の RepositotyImpl を作成する。

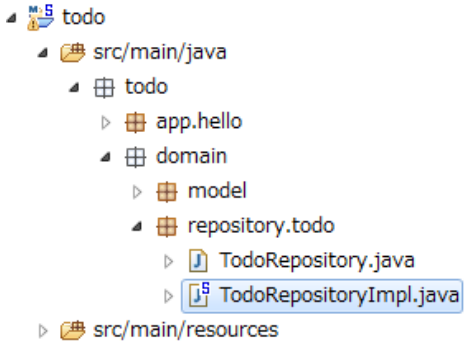
データベースを使用する場合は、「**データベースアクセスを伴うインフラストラクチャ層の作成**」に記載されている内容で読み替えて、RepositoryImpl を作成する。

FQCN は、`todo.domain.repository.todo.TodoRepositoryImpl` とする。

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 todo.domain.repository.todo;

import java.util.Collection;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import org.springframework.stereotype.Repository;

import 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 Todo save(Todo todo) {
        return TODO_MAP.put(todo.getTodoId(), todo);
    }

    @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 (`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、`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 Repositoty 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

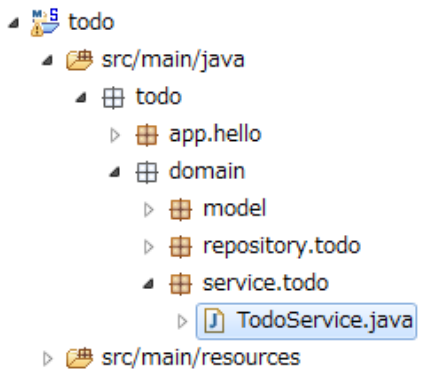
業務処理を実装する。必要な処理は、

- Todo の全件取得
- Todo の新規作成
- Todo の完了
- Todo の削除

である。

まずは、TodoService インタフェースを作成して、これらの処理を行うメソッドを定義する。
FQCN は、`todo.domain.serivce.todo.TodoService` とする。

```
package todo.domain.service.todo;

import java.util.Collection;

import todo.domain.model.Todo;

public interface TodoService {
    Collection<Todo> findAll();

    Todo create(Todo todo);

    Todo finish(String todoId);

    void delete(String todoId);
}
```



必要な処理と、実装するメソッドの対応は、以下の通りである。

- Todo の全件取得 →findAll メソッド
- Todo の新規作成 →create メソッド
- Todo の完了 →finish メソッド
- Todo の削除 →delete メソッド

実装クラスの FQCN は、`todo.domain.service.todo.TodoServiceImpl` とする。

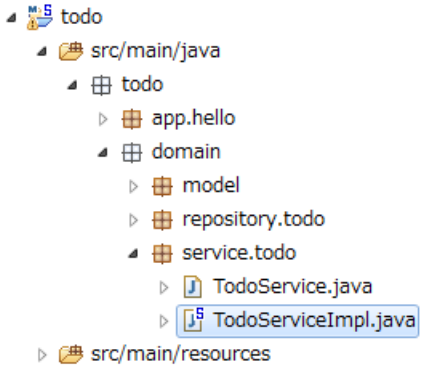
```
package todo.domain.service.todo;

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 todo.domain.model.Todo;
import 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)
```

```
    public 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.save(todo);

        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.save(todo);
            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。
(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を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、トランザクションを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、Error メッセージをスローするために、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

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。



Service の JUnit 作成

課題

TBD

Service の Unit テストの方法については、次版以降で記載する予定である。

3.4.2 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

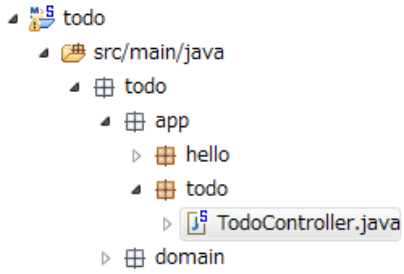
まずは、todo 管理業務にかかわる画面遷移を、制御する Controller を作成する。

FQCN は、`todo.app.todo.TodoController` とする。

上位パッケージがドメイン層とは異なるので注意すること。

```
package 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

この画面では、

- 新規作成フォームの表示
- TODO の全件表示

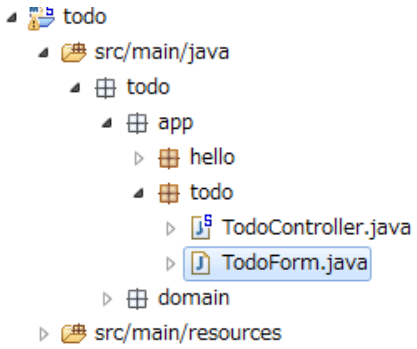
を行う。

Form の作成

Form には、タイトル情報があればよいので、次のような JavaBean になる。

FQCN は、todo.app.todo.TodoForm とする。

```
package 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

TodoController に、setUpForm メソッドと、list メソッドを実装する。

```
package 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.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;
```

```
@Controller
@RequestMapping("todo")
public class TodoController {
    @Inject // (3)
    TodoService todoService;

    @ModelAttribute // (4)
    public TodoForm setUpForm() {
        TodoForm form = new TodoForm();
        return form;
    }

    @RequestMapping(value = "list") // (5)
    public String list(Model model) {
        Collection<Todo> todos = todoService.findAll();
        model.addAttribute("todos", todos); // (6)
        return "todo/list"; // (7)
    }
}
```

項番	説明
(3)	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 TodoSerivce 型インスタンスがインジェクションされるため、結果として、TodoServiceImpl インスタンスがインジェクションされる。
(4)	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。 TodoController の各処理で、model.addAttribute(“ todoForm ” , form) が実行されるのと同義。
(5)	list メソッドを ” <contextPath>/todo/list ” にマッピングされるための設定。クラスレベルで@RequestMapping(“ todo ”) が設定されているため、ここでは @RequestMapping(value = “ list ”) だけで良い。
(6)	Model に Todo のリストを追加して、View に渡す。
(7)	View 名として ” todo/list ” を返すと、spring-mvc.xml に定義した InternalResourceViewResolver 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。

JSP の作成

`src/main/webapp/WEB-INF/views/todo/list.jsp` を作成し、Controller から渡された Model を表示する。

まずは、” Finish ” , ” Delete ” ボタン以外を作成する。

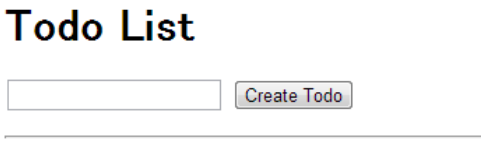
```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Todo List</title>
<style type="text/css">
```

```
    .strike {
        text-decoration: line-through;
    }
    </style>
    </head>
    <body>
        <h1>Todo List</h1>
        <div id="todoForm">
            <!-- (1) -->
            <form:form
               action="${pageContext.request.contextPath}/todo/create"
                method="post" modelAttribute="todoForm">
                <!-- (2) -->
                <form:input path="todoTitle" />
                <input type="submit" value="Create Todo" />
            </form:form>
        </div>
        <hr />
        <div id="todoList">
            <ul>
                <!-- (3) -->
                <c:forEach items="${todos}" var="todo">
                    <li><c:choose>
                            <c:when test="${todo.finished}"><!-- (4) -->
                                <span class="strike">
                                <!-- (5) -->
                                ${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)	<form:form>タグでフォームを表示する。modelAttribute 属性に、Controller で Model に追加した form の名前を指定する。 action 属性に指定する contextPath は、${pageContext.request.contextPath}で取得できる。
(2)	<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。
(3)	<c:forEach>タグを用いて、Todo のリストを全て表示する。
(4)	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。
(5)	**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。



Create TODO

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

`TodoController` に、create メソッドを追加する。

```
package 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.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
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 todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Controller
@RequestMapping("todo")
public class TodoController {
    @Inject
    TodoService todoService;

    // (8)
    @Inject
    Mapper beanMapper;

    @ModelAttribute
    public TodoForm setUpForm() {
        TodoForm form = new TodoForm();
        return form;
    }

    @RequestMapping(value = "list")
    public String list(Model model) {
        Collection<Todo> todos = todoService.findAll();
        model.addAttribute("todos", todos);
        return "todo/list";
```

```
    }

    @RequestMapping(value = "create", method = RequestMethod.POST) // (9)
    public String create(@Valid TodoForm todoForm, BindingResult bindingResult, // (10)
            Model model, RedirectAttributes attributes) { // (11)

        // (12)
        if (bindingResult.hasErrors()) {
            return list(model);
        }

        // (13)
        Todo todo = beanMapper.map(todoForm, Todo.class);

        try {
            todoService.create(todo);
        } catch (BusinessException e) {
            // (14)
            model.addAttribute(e.getResultMessages());
            return list(model);
        }

        // (15)
        attributes.addFlashAttribute(ResultMessages.success().add(
                ResultMessage.fromText("Created successfully!")));
        return "redirect:/todo/list";
    }

}
```

項番	説明
(8)	Form オブジェクトを、DomainObject に変換する際に、有用な Mapper をインジェクションする。
(9)	パスが/todo/create で、HTTP メソッドが POST に対応するように、`@RequestMapping` アノテーションを設定する。
(10)	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。
(11)	正常に作成が完了した後、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。
(12)	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。
(13)	Mapper を用いて、TodoForm 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、Mapper 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。
(14)	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。
(15)	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。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 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)	必須項目であるので、@NotNull アノテーションを付ける。
(2)	1 文字以上 30 文字以下であるので、@Size アノテーションで、範囲を指定する。

JSP の修正

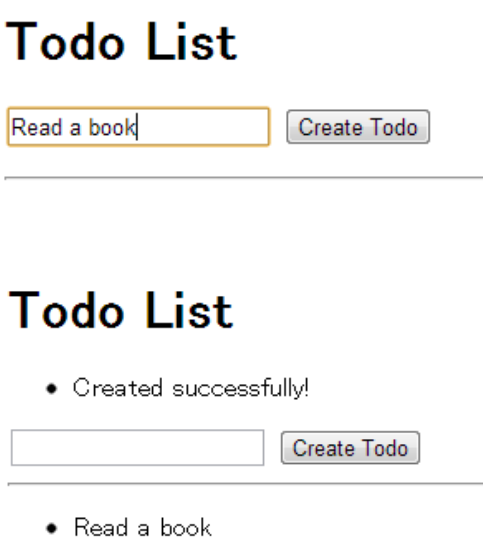
結果メッセージ表示用のタグを追加する。

```
<!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">
            <!-- (6) -->
            <t:messagesPanel />

            <form:form
               action="${pageContext.request.contextPath}/todo/create"
                method="post" modelAttribute="todoForm">
                <form:input path="todoTitle" />
                <form:errors path="todoTitle" /><!-- (7) -->
                <input type="submit" value="Create Todo" />
            </form:form>
        </div>
        <hr />
        <div id="todoList">
            <ul>
                <c:forEach items="${todos}" var="todo">
                    <li><c:choose>
                            <c:when test="${todo.finished}">
                                <span style="text-decoration: line-through;">
                                ${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
(6)	<t:messagesPanel>タグで、結果メッセージを表示する。
(7)	<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。

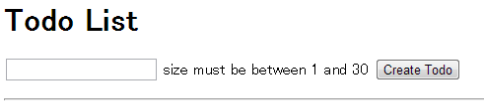
フォームに適切な値を入力して submit すると、以下のように、成功メッセージが表示される。



6 件以上登録した場合は、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

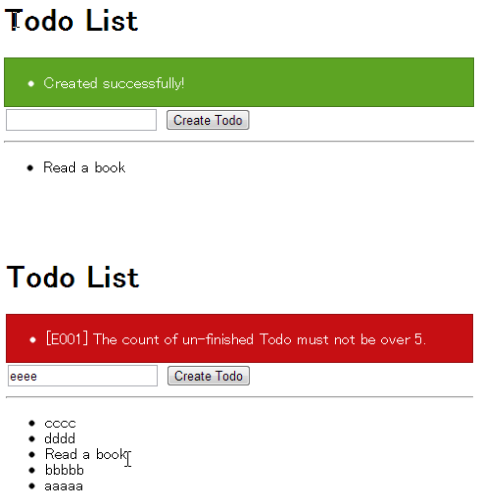
<t:messagesPanel>の結果はデフォルトで、

```
    <div class="alert alert-success"><ul><li>Created successfully!</li></ul></div>
```

と出力される。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;
}
```

入力エラーは、以下のように装飾される。



Finish TODO

一覧表示画面に ”Finish ”ボタンを追加して、ボタンを submit すると、hidden で対象の todoId が送られ、Todo を完了するように実装する。

JSP の修正

完了用の 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;
}

.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="${pageContext.request.contextPath}/todo/create"
                method="post" modelAttribute="todoForm">
                <form:input path="todoTitle" />
                <form:errors path="todoTitle" cssClass="text-error" />
                <input type="submit" value="Create Todo" />
            </form:form>
        </div>
        <hr />
        <div id="todoList">
            <ul>
                <c:forEach items="${todos}" var="todo">
                    <li><c:choose>
                            <c:when test="${todo.finished}">
                                <span class="strike">${f:h(todo.todoTitle)}</span>
                            </c:when>
                            <c:otherwise>
                                ${f:h(todo.todoTitle)}
                                <!-- (8) -->
                                <form:form
                                    action="${pageContext.request.contextPath}/todo/finish"
                                    method="post"
                                    modelAttribute="todoForm"
                                    cssStyle="display: inline-block;">
                                    <!-- (9) -->
                                    <form:hidden path="todoId"
                                        value="${f:h(todo.todoId)}" />
                                    <input type="submit" name="finish"
                                        value="Finish" />
                                </form:form>
                            </c:otherwise>
                        </c:choose></li>
                </c:forEach>
            </ul>
        </div>
    </body>
    </html>
```

項番	説明
(8)	未完了の場合に、完了用の form を表示する。<contextPath>/todo/finish に対して、POST で todoId を送信する。
(9)	<form:hidden>タグで todoId を渡す。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。**

Form の修正

完了用のフォームも、TodoForm を用いる。TodoForm に、todoId プロパティを追加する必要があるが、そのままだと、新規作成用の入力チェックルールが適用されてしまう。一つの Form に、新規作成用と完了用で、別々のルールを指定するために、group 属性を設定する。

```
package todo.app.todo;

import java.io.Serializable;

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

public class TodoForm implements Serializable {
    // (3)
    public static interface TodoCreate {
    };

    public static interface TodoFinish {
    };

    private static final long serialVersionUID = 1L;

    // (4)
    @NotNull(groups = { TodoFinish.class })
    private String todoId;

    // (5)
    @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
(3)	グループ化したバリデーションを行うためのグループ名となるクラスを作成する。 クラスは空でよいため、ここでは、インタフェースを定義する。 グループ化バリデーションについては、入力チェックを参照されたい。
(4)	todoId は、完了処理には必須であるため、`@NotNull` アノテーションをつける。 完了時にのみ必要なルールであるので、group 属性に TodoFinish.class を設定する。
(5)	新規作成用のルールは、完了処理には不要であるので、`@NotNull` アノテーション、`@Size` アノテーション、それぞれの group 属性に TodoCreate.class を設定する。

Controller の修正

完了処理を TodoController に追加する。グループ化したバリデーションを実行するために、**@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 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;
```

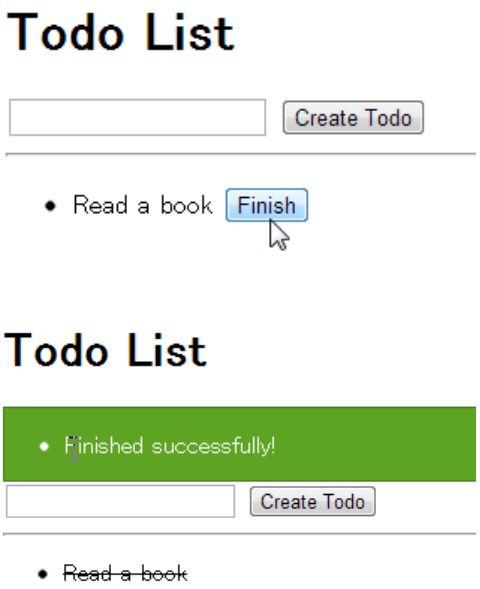
```
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.message.ResultMessage;
import org.terasoluna.gfw.common.message.ResultMessages;

import todo.app.todo.TodoForm.TodoCreate;
import todo.app.todo.TodoForm.TodoFinish;
import todo.domain.model.Todo;
import 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;
    }

    @RequestMapping(value = "list")
    public String list(Model model) {
        Collection<Todo> todos = todoService.findAll();
        model.addAttribute("todos", todos);
        return "todo/list";
    }

    @RequestMapping(value = "create", method = RequestMethod.POST)
    public String create(
            @Validated({ Default.class, TodoCreate.class }) TodoForm todoForm, // (16)
            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";
        }

        @RequestMapping(value = "finish", method = RequestMethod.POST) // (17)
        public String finish(
                @Validated({ Default.class, TodoFinish.class }) TodoForm form, // (18)
                BindingResult bindingResult, Model model,
                RedirectAttributes attributes) {
            // (19)
            if (bindingResult.hasErrors()) {
                return list(model);
            }

            try {
                todoService.finish(form.getTodoId());
            } catch (BusinessException e) {
                // (20)
                model.addAttribute(e.getResultMessages());
                return list(model);
            }

            // (21)
            attributes.addFlashAttribute(ResultMessages.success().add(
                    ResultMessage.fromText("Finished successfully!")));
            return "redirect:/todo/list";
        }
    }
```

項番	説明
(16)	グループ化したバリデーションを実施するために、`@Valid` アノテーションから `@Validated` アノテーションに変更する。 value には、対象のグループクラスを複数指定できる。Default.class はバリデーションルールに group が指定されていない場合のグループである。 `@Validated` アノテーションを使用する際は、Default.class も指定しておくのがよい。
(17)	パスが、/todo/finish で、HTTP メソッドが POST に対応するように、`@RequestMapping` アノテーションを設定する。
(18)	Finish 用のグループとして、TodoFinish.class を指定する。
(19)	入力エラーがあった場合、一覧画面に戻る。
(20)	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。
(21)	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。

注釈: Create 用、Finish 用に、別々の Form を作成しても良い。その場合は、必要なパラメータだけが、Form のプロパティになる。ただし、クラス数が増え、プロパティも重複することが多いので、仕様変更が発生した場合に、修正コストが高くなる。また、同一の Controller 内で、複数の Form オブジェクトを、`@ModelAttribute` メソッドによって初期化すると、毎回すべての Form が初期化されてしまうので、不要なインスタンスが生成されてしまう。そのため、基本的に、一つの Controller で利用する Form は、できるだけ集約し、グループ化したバリデーションの設定を行うことを推奨する。

Todo を新規作成した後に、Finish ボタンを Submit 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

一覧表示画面に”Delete” ボタンを追加して、ボタンを submit すると、hidden で対象の todoId が送られ、Todo を完了するように実装する。

JSP の修正

削除用の 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;
    }

    .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="${pageContext.request.contextPath}/todo/create"
                method="post" modelAttribute="todoForm">
                <form:input path="todoTitle" />
                <form:errors path="todoTitle" cssClass="text-error" />
                <input type="submit" value="Create Todo" />
            </form:form>
        </div>
        <hr />
        <div id="todoList">
            <ul>
                <c:forEach items="${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"
                                    cssStyle="display: inline-block;">
                                    <form:hidden path="todoId"
                                        value="${f:h(todo.todoId)}" />
                                    <input type="submit" name="finish"
                                        value="Finish" />
```

```
                            </form:form>
                        </c:otherwise>
                    </c:choose>
                    <!-- (10) -->
                    <form:form
                        action="${pageContext.request.contextPath}/todo/delete"
                        method="post" modelAttribute="todoForm"
                        cssStyle="display: inline-block;">
                        <!-- (11) -->
                        <form:hidden path="todoId"
                            value="${f:h(todo.todoId)}" />
                        <input type="submit" value="Delete" />
                    </form:form>
                </li>
            </c:forEach>
        </ul>
    </div>
</body>
</html>
```

項番	説明
(10)	削除用の form を表示する。<contextPath>/todo/delete に対して、POST で todoId を送信する。
(11)	<form:hidden>タグで、todoId を渡す。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。**

Form の修正

Delete 用のグループを、TodoForm に追加する。ルールは、Finish 用と同じである。

```
package 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 {
    };
```

```
        // (6)
        public static interface TodoDelete {
        }

        private static final long serialVersionUID = 1L;

        // (7)
        @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
(6)	Delete 用のグループ TodoDelete を定義する。
(7)	todoId プロパティに対して、TodoDelete グループのバリデーションを行うように設定する。

Controller の修正

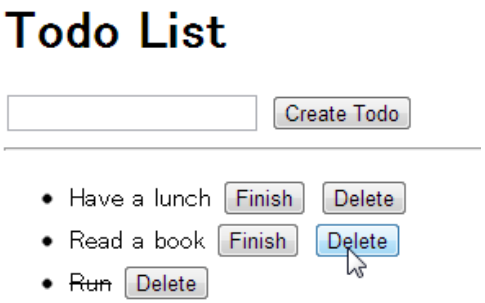
削除処理を、TodoController に追加する。完了処理とほぼ同じである。

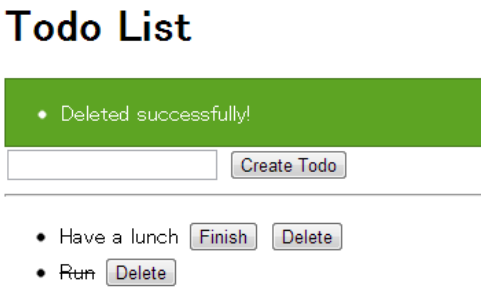
```
package todo.app.todo;

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.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
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 todo.app.todo.TodoForm.TodoDelete;
import todo.app.todo.TodoForm.TodoCreate;
import todo.app.todo.TodoForm.TodoFinish;
import todo.domain.model.Todo;
import 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;
    }

    @RequestMapping(value = "list")
    public String list(Model model) {
        Collection<Todo> todos = todoService.findAll();
        model.addAttribute("todos", todos);
        return "todo/list";
    }

    @RequestMapping(value = "create", method = RequestMethod.POST)
    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";
    }

    @RequestMapping(value = "finish", method = RequestMethod.POST)
    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";
    }

    @RequestMapping(value = "delete", method = RequestMethod.POST) // (22)
    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
(22)	Todo に対して、” Delete ” ボタンを submit すると、以下のように、対象の TODO が削除される。





3.5 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

本節では、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作成方法について、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作成について、説明する。

- Spring Data JPA
- TERASOLUNA DAO(MyBatis2)

3.5.1 共通設定

まずは、Spring Data JPA 版、TERASOLUNA Dao 版の両方に共通して適用する設定を行う。
今回は、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
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 文は以下の通り。

```
    create table if not exists todo(
        todo_id varchar(36) primary key,
        todo_title varchar(30),
        finished boolean,
```

```
            created_at timestamp
        )
```

3.5.2 Spring Data JPA を使用したインフラストラクチャ層の作成

本節では、インフラストラクチャ層において、 Spring Data JPA を使用する方法について、説明する。TERASOLUNA DAO を使用する場合は、本節を読み飛ばして、*TERASOLUNA DAO(MyBatis2)* **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。

Entity の設定

Todo クラスを、データベースと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 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
    // (3)
    @Column(name = "todo_id")
    private String todoId;

    @Column(name = "todo_title")
    private String todoTitle;

    @Column(name = "finished")
    private boolean finished;
```

```
        @Column(name = "created_at")
        // (4)
        @Temporal(TemporalType.TIMESTAMP)
        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)	`@Column` アノテーションで、対応するカラム名を設定する。
(4)	Date 型は、java.sql.Date, java.sql.Time, java.sql.Timestamp のどれに対応するか、明示的に指定する必要がある。ここでは、Timestamp を指定する。

TodoRepository の修正

Spring Data JPA の Repository 機能を使用するための修正を行う。

```
package 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 todo.domain.model.Todo;

// (1)
public interface TodoRepository extends JpaRepository<Todo, String> {

    @Query(value = "SELECT COUNT(x) FROM Todo x WHERE x.finished = :finished") // (2)
    long countByFinished(@Param("finished") boolean finished); // (3)

}
```

項番	説明
(1)	JpaRepository を拡張したインタフェースにする。Generics のパラメータには、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、上位のインタフェースに定義済みであるため、TodoRepository では、countByFinished のみ定義すればよい。
(2)	countByFinished を呼び出した際に、実行される JPQL を、`@Query` アノテーションで指定する。
(3)	(2) で指定した JPQL のバインド変数を`@Param` アノテーションで設定する。 ここでは、JPQL 中の”`:finished`”を埋めるためのメソッド引数に、 @Param(“ finished ”) を付けている。

TodoRepositoryImpl の作成

Spring Data JPA を使用した場合、RepositoryImpl は、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。

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:2014-08-26 12:57:48    thread:tomcat-http--3   X-Track:2705c06e0eae416591bc1f27efc84cd
      date:2014-08-26 12:57:48    thread:tomcat-http--3   X-Track:2705c06e0eae416591bc1f27efc84cd
      date:2014-08-26 12:57:48    thread:tomcat-http--3   X-Track:2705c06e0eae416591bc1f27efc84cd
      date:2014-08-26 12:57:48    thread:tomcat-http--3   X-Track:2705c06e0eae416591bc1f27efc84cd
      2. /* select generatedAlias0 from Todo as generatedAlias0 */ select todo0_.todo_id as todo1_
      date:2014-08-26 12:57:48    thread:tomcat-http--3   X-Track:2705c06e0eae416591bc1f27efc84cd
      date:2014-08-26 12:57:48    thread:tomcat-http--3   X-Track:2705c06e0eae416591bc1f27efc84cd
      date:2014-08-26 12:57:48    thread:tomcat-http--3   X-Track:2705c06e0eae416591bc1f27efc84cd
      date:2014-08-26 12:57:48    thread:tomcat-http--3   X-Track:2705c06e0eae416591bc1f27efc84cd
```

3.5.3 TERASOLUNA DAO(MyBatis2) を使用したインフラストラクチャ層の作成

本節では、インフラストラクチャ層において、TERASOLUNA DAO(MyBatis2) を使用する場合の設定方法について説明する。

注釈: TERASOLUNA DAO とは、MyBatis2.3.5 と Spring の連携クラスである org.springframework.orm.ibatis.support.SqlMapClientDaoSupport を、用途別に拡張した簡易 SQL マッパーを提供するライブラリである。以下 4 つのインタフェースをもつ DAO が、提供されている。

1. jp.terasoluna.fw.dao.QueryDAO
2. jp.terasoluna.fw.dao.UpdateDAO
3. jp.terasoluna.fw.dao.StoredProcedureDAO
4. jp.terasoluna.fw.dao.QueryRowHandleDAO

それぞれのインタフェースに対して、`jp.terasoluna.fw.dao.ibatis.XxxDAOiBatisImpl` という実装を持つ。

RepositoryImpl の作成

TERASOLUNA DAO(MyBatis2) を使用した場合、RepositoryImpl は、TERASOLUNA DAO から提供されているクラスを呼び出すことで、データベースにアクセスを行うように実装する。

```
package todo.domain.repository.todo;

import java.util.Collection;

import javax.inject.Inject;

import jp.terasoluna.fw.dao.QueryDAO;
import jp.terasoluna.fw.dao.UpdateDAO;

import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import todo.domain.model.Todo;

@Repository // (1)
@Transactional // (2)
public class TodoRepositoryImpl implements TodoRepository {
```

```
        // (3)
        @Inject
        QueryDAO queryDAO;

        @Inject
        UpdateDAO updateDAO;

        // (4)
        @Override
        @Transactional(readOnly = true)
        public Todo findOne(String todoId) {
            return queryDAO.executeForObject("todo.findOne", todoId, Todo.class);
        }

        @Override
        @Transactional(readOnly = true)
        public Collection<Todo> findAll() {
            return queryDAO.executeForObjectList("todo.findAll", null);
        }

        @Override
        public Todo save(Todo todo) {
            // (5)
            if (exists(todo.getTodoId())) {
                updateDAO.execute("todo.update", todo);
            } else {
                updateDAO.execute("todo.create", todo);
            }
            return todo;
        }

        @Transactional(readOnly = true)
        public boolean exists(String todoId) {
            long count = queryDAO.executeForObject("todo.exists", todoId,
                    Long.class);
            return count > 0;
        }

        @Override
        public void delete(Todo todo) {
            updateDAO.execute("todo.delete", todo);
        }

        @Override
        @Transactional(readOnly = true)
        public long countByFinished(boolean finished) {
            return queryDAO.executeForObject("todo.countByFinished", finished,
                    Long.class);
        }
    }
```

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。
(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。 Repository を呼び出す Service 側でも設定しているため、`@Transactional` をつけなくともトランザクション管理になるが、propagation 属性は、デフォルトの REQUIRED であるため、トランザクションがネストした場合、内側 (Repository 側) のトランザクションは、外側 (Service 側) のトランザクションに参加する。
(3)	`@Inject` アノテーションで、QueryDAO, UpdateDAO をインジェクションする。
(4)	Repository のメソッド実装は、基本的には、TERASOLUNA DAO に SQLID と、パラメータを渡すことになる。 参照系の場合は QueryDAO、更新系の場合は UpdateDAO を使用する。SQLID に対応する SQL の設定は、次に行う。
(5)	save メソッドで新規作成と、更新の両方を実装している。 どちらの処理を行うか判断するために、exists メソッドを作成する。 このメソッドでは、対象の todoId の件数を取得し、件数が 0 より大きいかどうかで存在を確認する。

注釈: save メソッドは、新規作成でも更新でも利用できるメリットがある。しかしながら、2 回 SQL が実行されるという性能面でのデメリットもある。性能を重視する場合は、新規作成用に create メソッドを、更新用に update メソッドを作成すること。

SQLMap ファイルの作成

`src/main/resources/META-INF/mybatis/sql/todo-sqlmap.xml` を作成し、`TodoRepositoryImpl` で使用した SQLID に対応する sql を、以下のように記述する。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
            PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
            "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="todo">
    <resultMap id="todo" class="todo.domain.model.Todo">
        <result property="todoId" column="todo_id" />
        <result property="todoTitle" column="todo_title" />
        <result property="finished" column="finished" />
        <result property="createdAt" column="created_at" />
    </resultMap>

    <select id="findOne" parameterClass="java.lang.String"
        resultMap="todo">
    <![CDATA[
        SELECT
            todo_id,
            todo_title,
            finished,
            created_at
        FROM
            todo
        WHERE
            todo_id = #value#
    ]]>
    </select>

    <select id="findAll" resultMap="todo">
    <![CDATA[
        SELECT
            todo_id,
            todo_title,
            finished,
            created_at
        FROM
            todo
    ]]>
    </select>

    <insert id="create" parameterClass="todo.domain.model.Todo">
    <![CDATA[
        INSERT INTO todo
        (
            todo_id,
            todo_title,
            finished,
            created_at
        )
        VALUES
        (
            #todoId#,
```

```
            #todoTitle#,
            #finished#,
            #createdAt#
        )
    ]]>
    </insert>

    <update id="update" parameterClass="todo.domain.model.Todo">
    <![CDATA[
        UPDATE todo
        SET
            todo_title = #todoTitle#,
            finished = #finished#,
            created_at = #createdAt#
        WHERE
            todo_id = #todoId#
    ]]>
    </update>

    <delete id="delete" parameterClass="todo.domain.model.Todo">
    <![CDATA[
        DELETE FROM
            todo
        WHERE
            todo_id = #todoId#
    ]]>
    </delete>

    <select id="countByFinished" parameterClass="java.lang.Boolean"
        resultClass="java.lang.Long">
    <![CDATA[
        SELECT
            COUNT(*)
        FROM
            todo
        WHERE
            finished = #value#
    ]]>
    </select>

    <select id="exists" parameterClass="java.lang.String"
        resultClass="java.lang.Long">
    <![CDATA[
        SELECT
            COUNT(*)
        FROM
            todo
        WHERE
            todo_id = #value#
    ]]>
    </select>
```

```
    </sqlMap>
```

以上で、TERASOLUNA DAO 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:2014-08-26 13:39:08    thread:tomcat-http--9   X-Track:49fbb5426f574375b8169d39c330ec7
    date:2014-08-26 13:39:08    thread:tomcat-http--9   X-Track:49fbb5426f574375b8169d39c330ec7
    date:2014-08-26 13:39:08    thread:tomcat-http--9   X-Track:49fbb5426f574375b8169d39c330ec7
    date:2014-08-26 13:39:08    thread:tomcat-http--9   X-Track:49fbb5426f574375b8169d39c330ec7
    4. SELECT             todo_id,             todo_title,             finished,             cr
    date:2014-08-26 13:39:08    thread:tomcat-http--9   X-Track:49fbb5426f574375b8169d39c330ec7
    date:2014-08-26 13:39:08    thread:tomcat-http--9   X-Track:49fbb5426f574375b8169d39c330ec7
    date:2014-08-26 13:39:08    thread:tomcat-http--9   X-Track:49fbb5426f574375b8169d39c330ec7
    date:2014-08-26 13:39:08    thread:tomcat-http--9   X-Track:49fbb5426f574375b8169d39c330ec7
    date:2014-08-26 13:39:08    thread:tomcat-http--9   X-Track:49fbb5426f574375b8169d39c330ec7
```

3.6 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 Global Framework による基本的なアプリケーションの開発方法、および Eclipse プロジェクトの構築方法
- STS の使用方法
- Maven で TERASOLUNA Global Framework を使用する方法
- TERASOLUNA Global Framework のアプリケーションのレイヤ化に従った開発方法
- POJO(+ Spring) によるドメイン層の実装
- 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
- Spring Data JPA によるインフラストラクチャ層の実装
- MyBatis2 によるインフラストラクチャ層の実装

ここで作成した 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
- ページング処理を追加する → 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
- CSRF 対策を追加する → CSRF 対策

第 4 章

TERASOLUNA Global Framework によるアプリケーション開発

TERASOLUNA Global Framework 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。

4.1 ドメイン層の実装

4.1.1 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。**

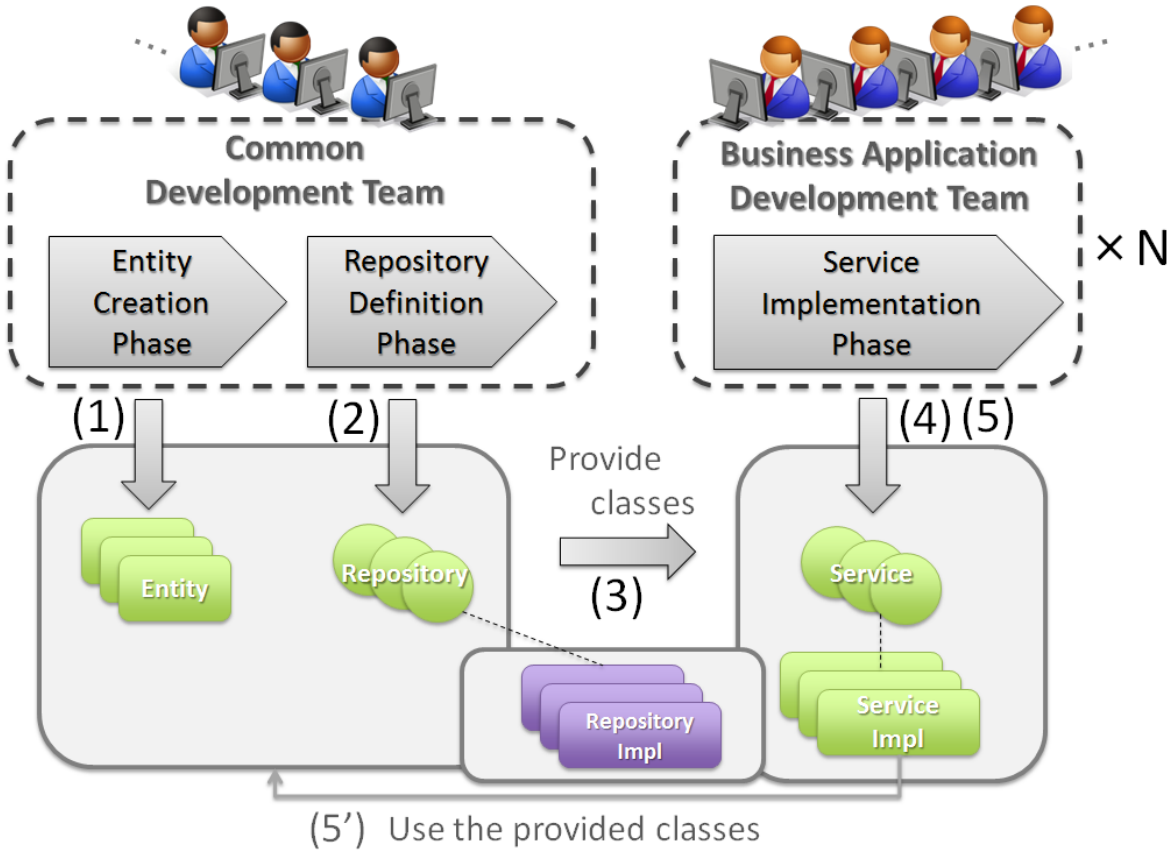
注釈: 本ガイドラインでは、Entity クラスおよび Repository を作成する構成を推奨しているが、この構成で開発することを強制するものではない。

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。

4.1.2 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

> **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 の作成せずに、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。

4.1.3 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。

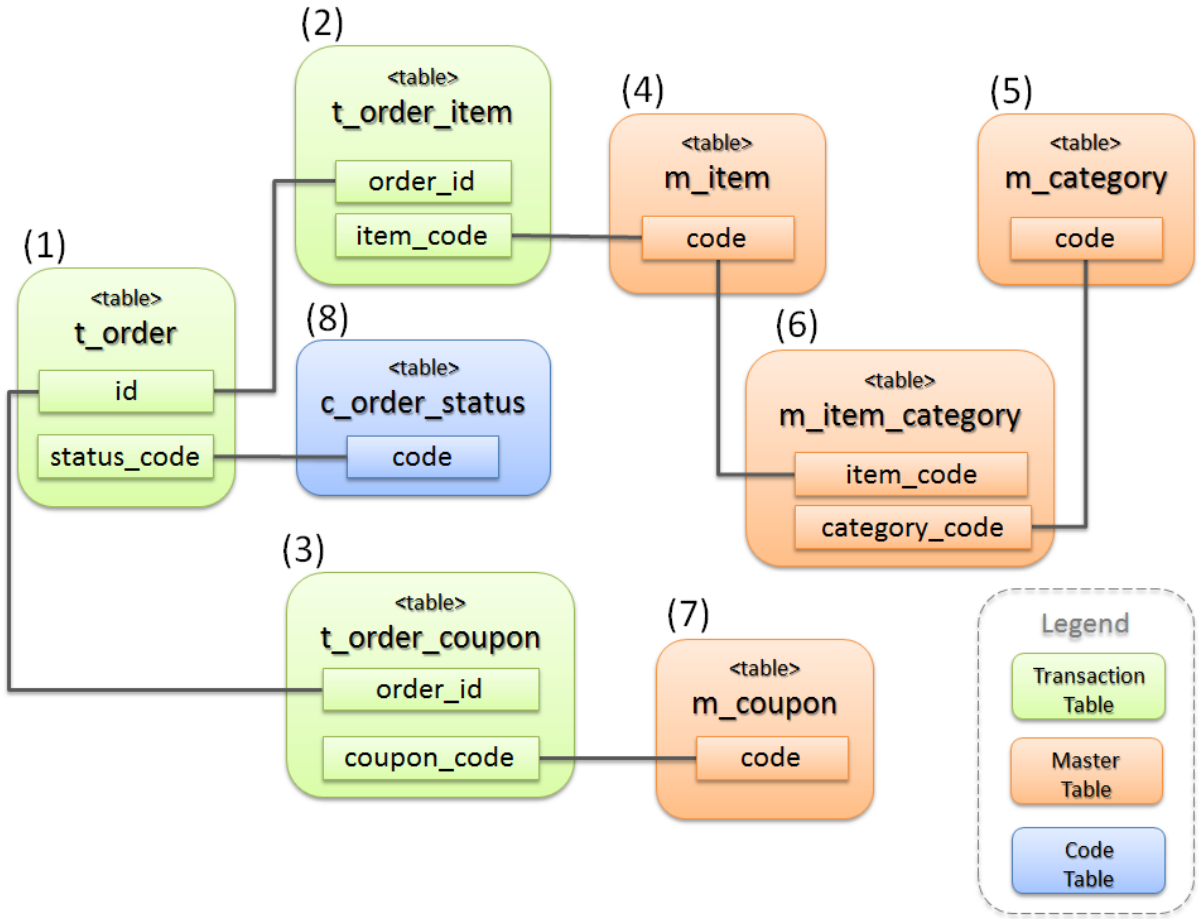
Mybatis は、データベースで管理されているレコードとオブジェクトをマッピングするという考え方ではなく、SQL とオブジェクトをマッピングという考え方で開発された O/R Mapper であるため、SQL の実装次第で、テーブル構成に依存しないオブジェクトへのマッピングができる。

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

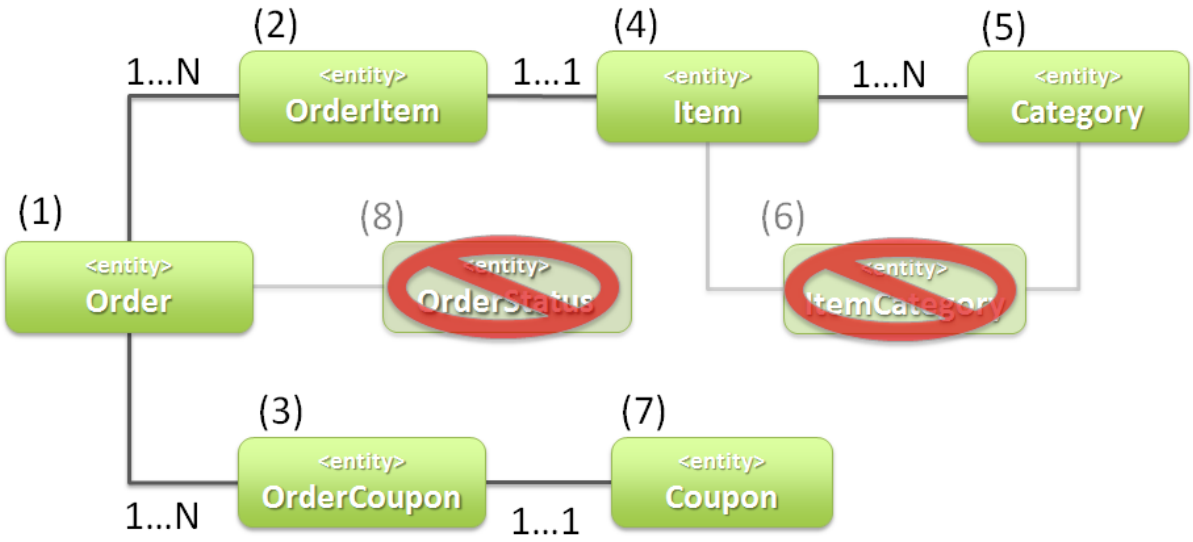
商品を購入する際に必要となる業務データを保持するテーブルは、以下の構成となっている。



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。

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。

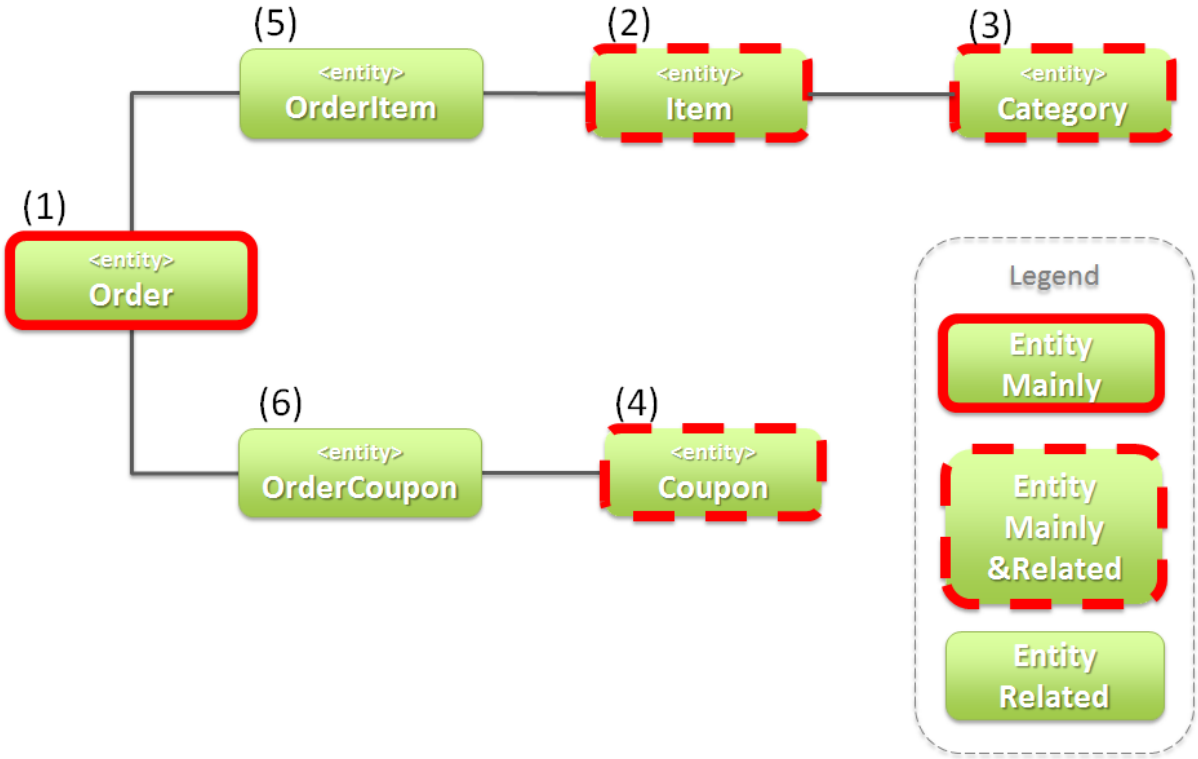


項番	クラス名	説明
(1)	Order	t_order テーブルの 1 レコードを表現する Entity クラス。 関連 Entity として、`OrderItem` および `OrderCoupon` を複数保持する。
(2)	OrderItem	t_order_item テーブルの 1 レコードを表現する Entity クラス。 関連 Entity として、 `Item` を保持する。
(3)	OrderCoupon	t_order_coupon テーブルの 1 コードを表現する Entity クラス。 関連 Entity として、`Coupon` を保持する。
(4)	Item	m_item 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。
(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。

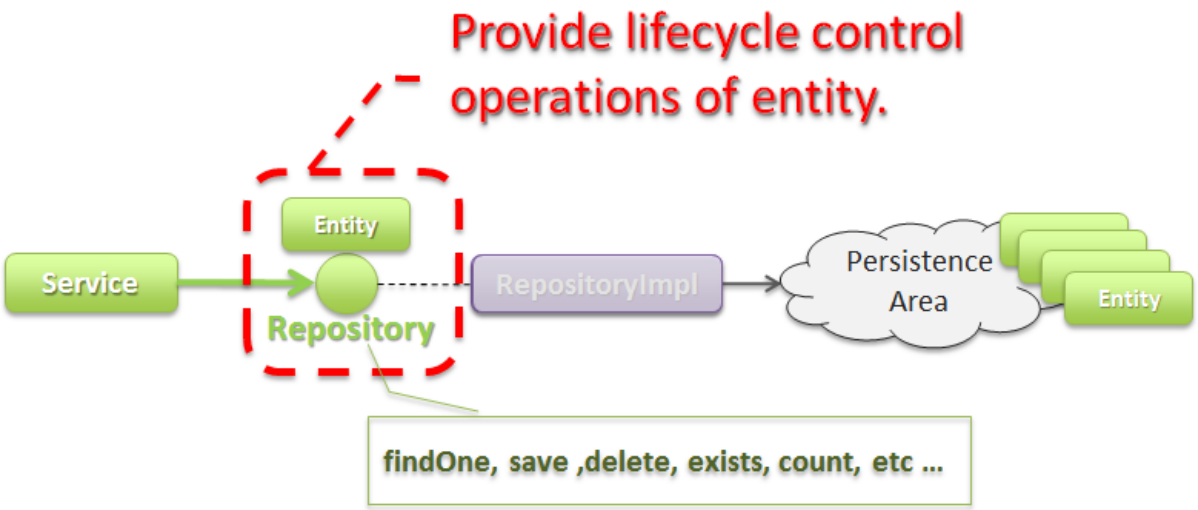
4.1.4 Repository の実装

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。



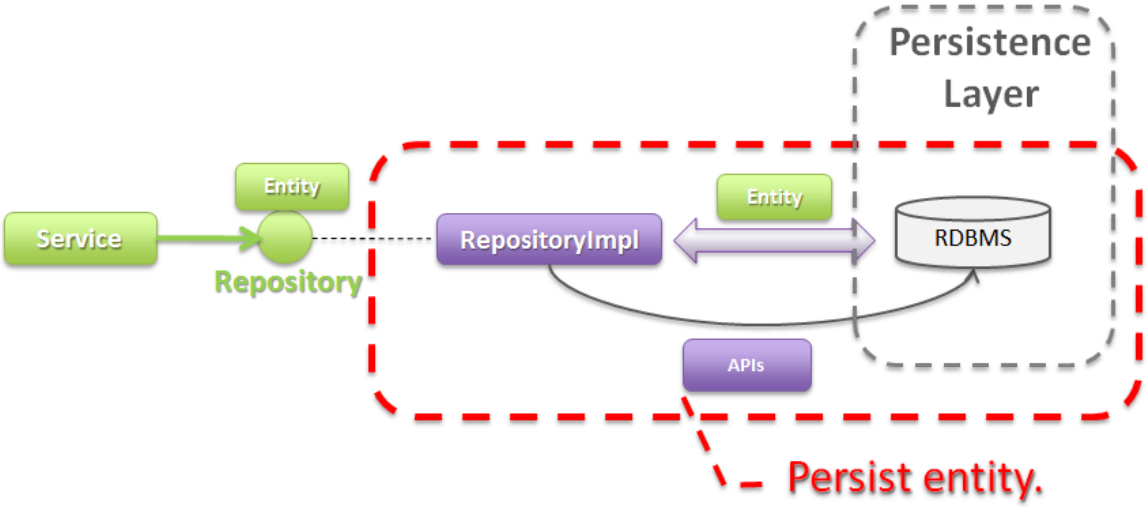
2. **Entity を永続化する処理 (Repository インタフェースの実装クラス) を提供する。**

   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。

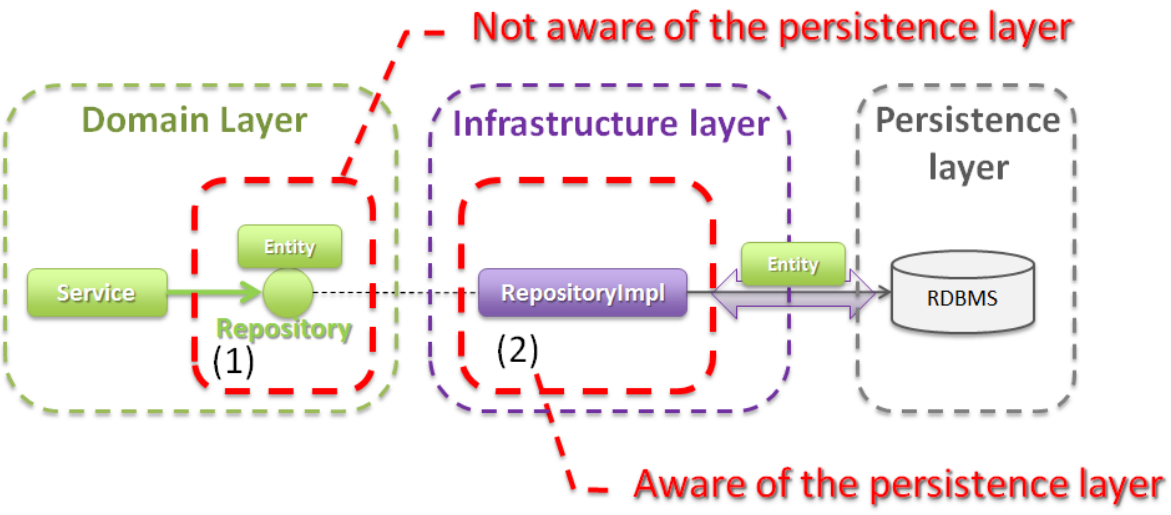
   実際の永続化処理は、O/R Mapper などから提供されている API を使って行う。

   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。

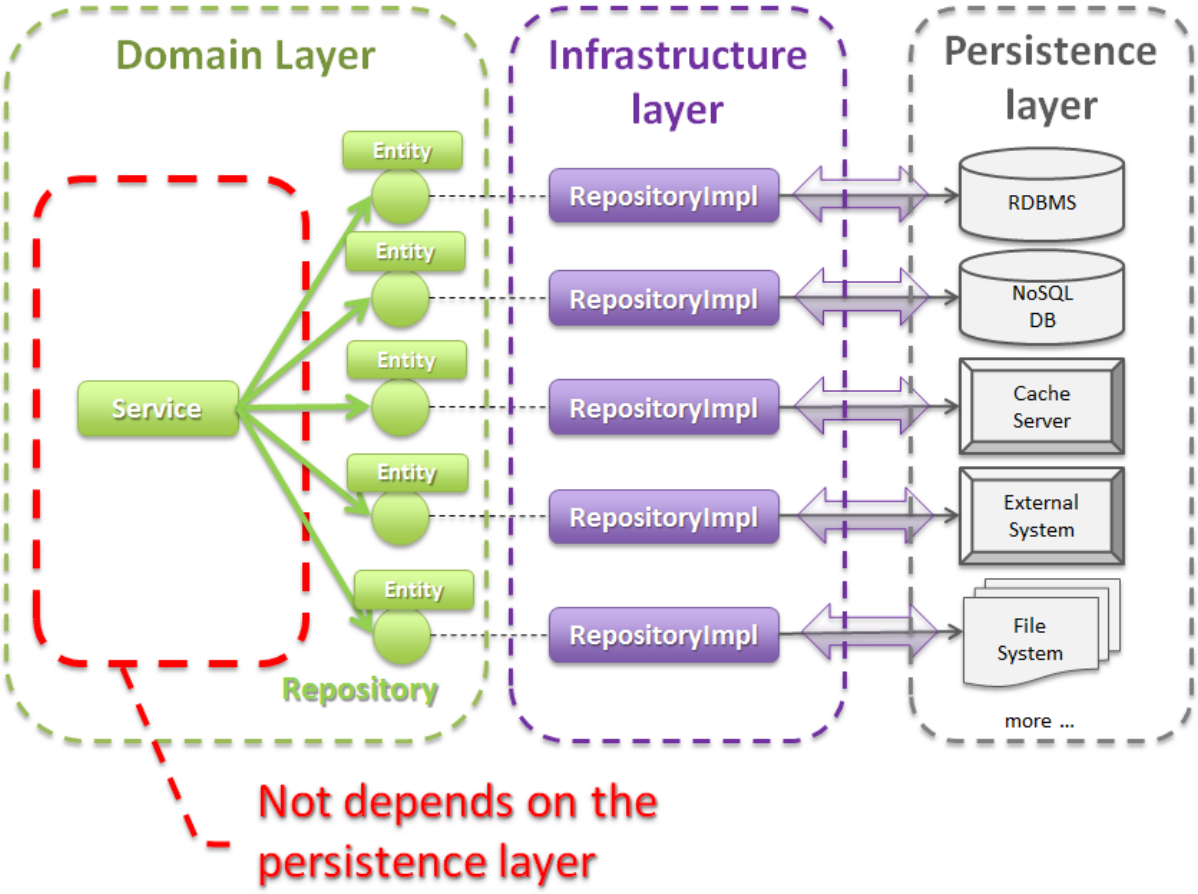
永続先が複数になる場合、以下のような構成となる。

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。

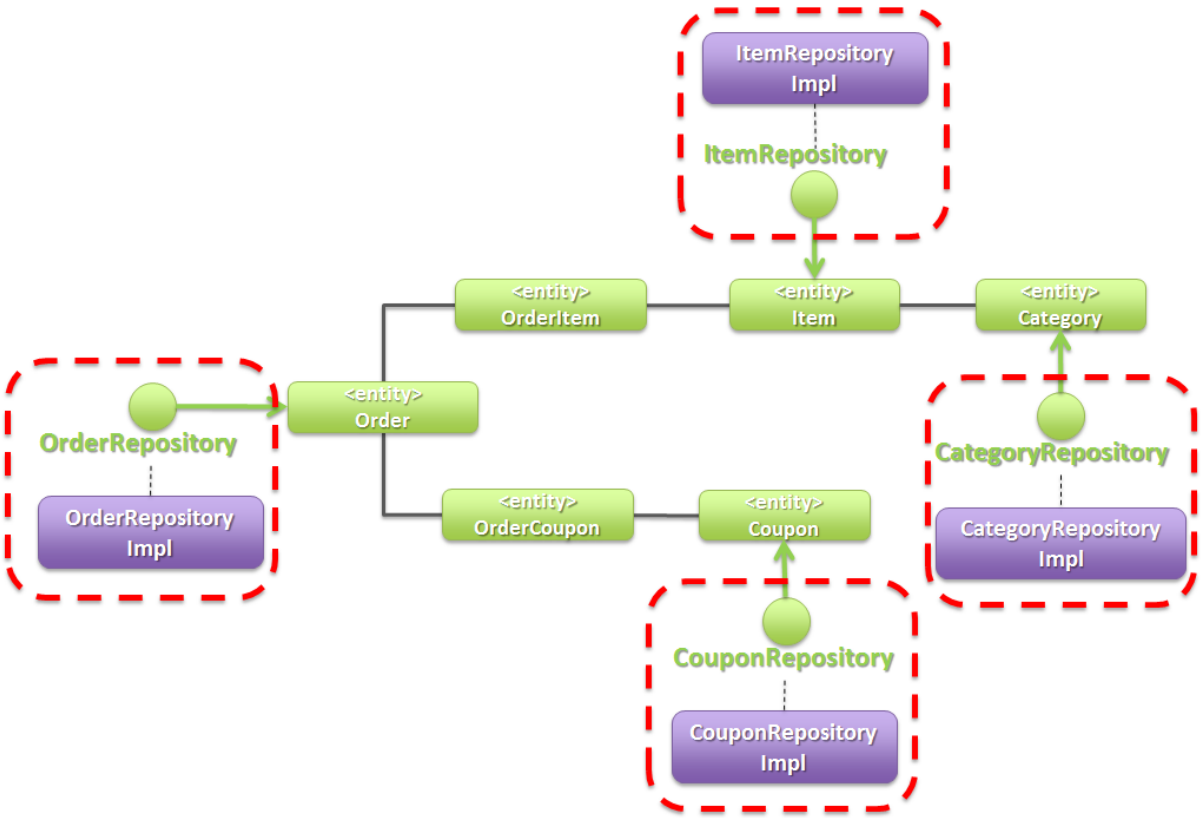
以下は、*Entity* **クラスの作成例**の説明で使用した、Entity クラスの Repository を作成する例となっている。

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。
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。
5.	存在判定系のメソッ	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。

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。

4.1.5 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。**

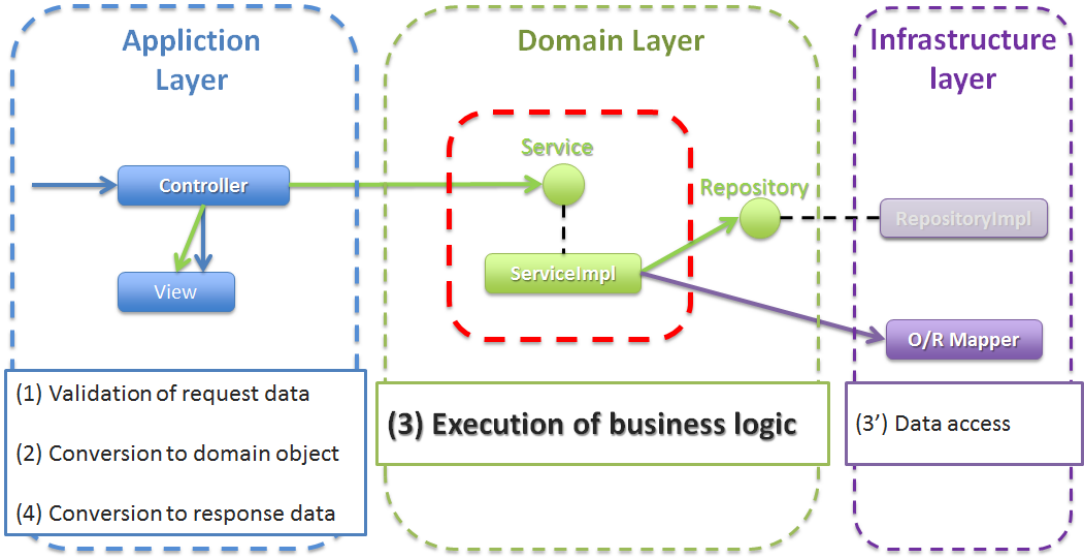
   業務ロジックは、アプリケーションで使用する業務データの参照、更新、整合性チェックおよびビジネスルールに関わる各種処理で構成される。

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。

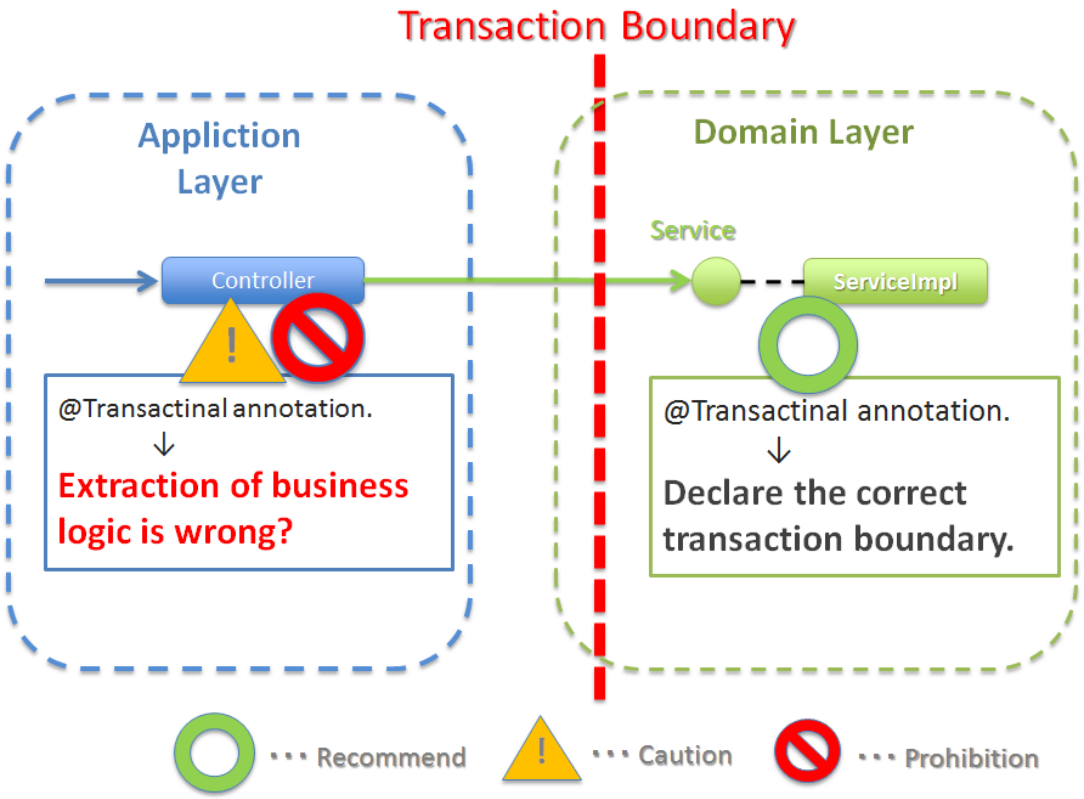


2. **トランザクション境界を宣言する。**

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

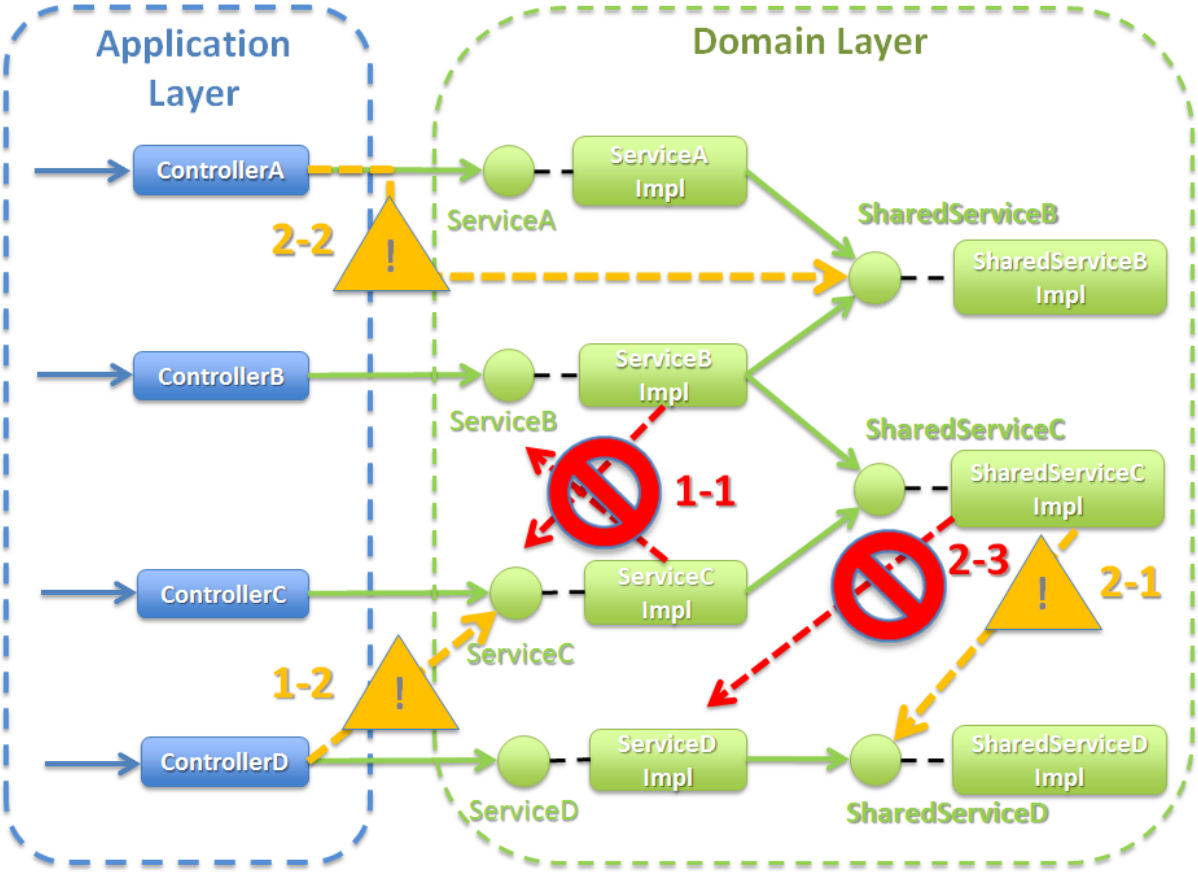
Service は、Service クラスと SharedService クラスで構成され、それぞれ以下の役割を担う。

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、開発者のひとりひとり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。

Appendix に、シグネチャを限定するようなインタフェースと規定クラスを作成するの、サンプルを示す。
詳細は、**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**を参照されたい。

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。
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

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。

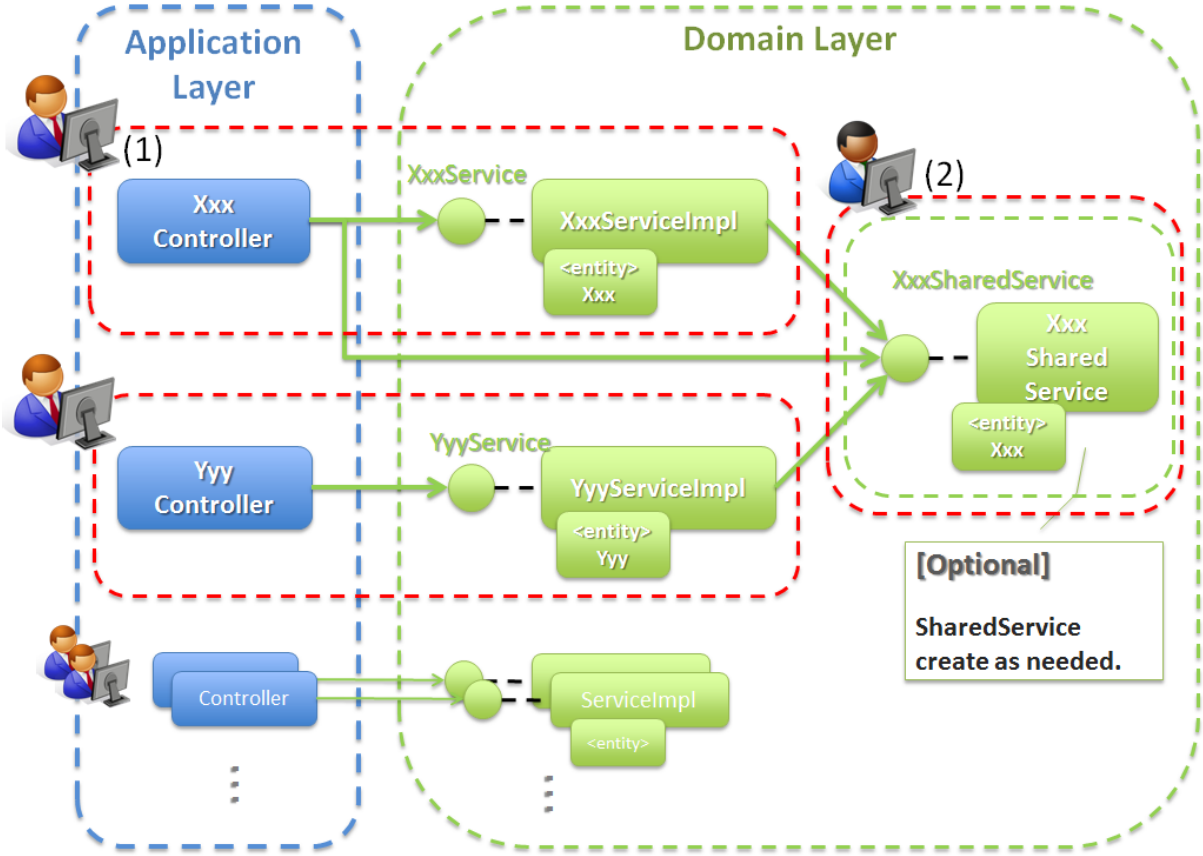
ちなみに: 「TERASOLUNA ViSC」を使用する場合は、BLogic は設計書から出力される。

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。



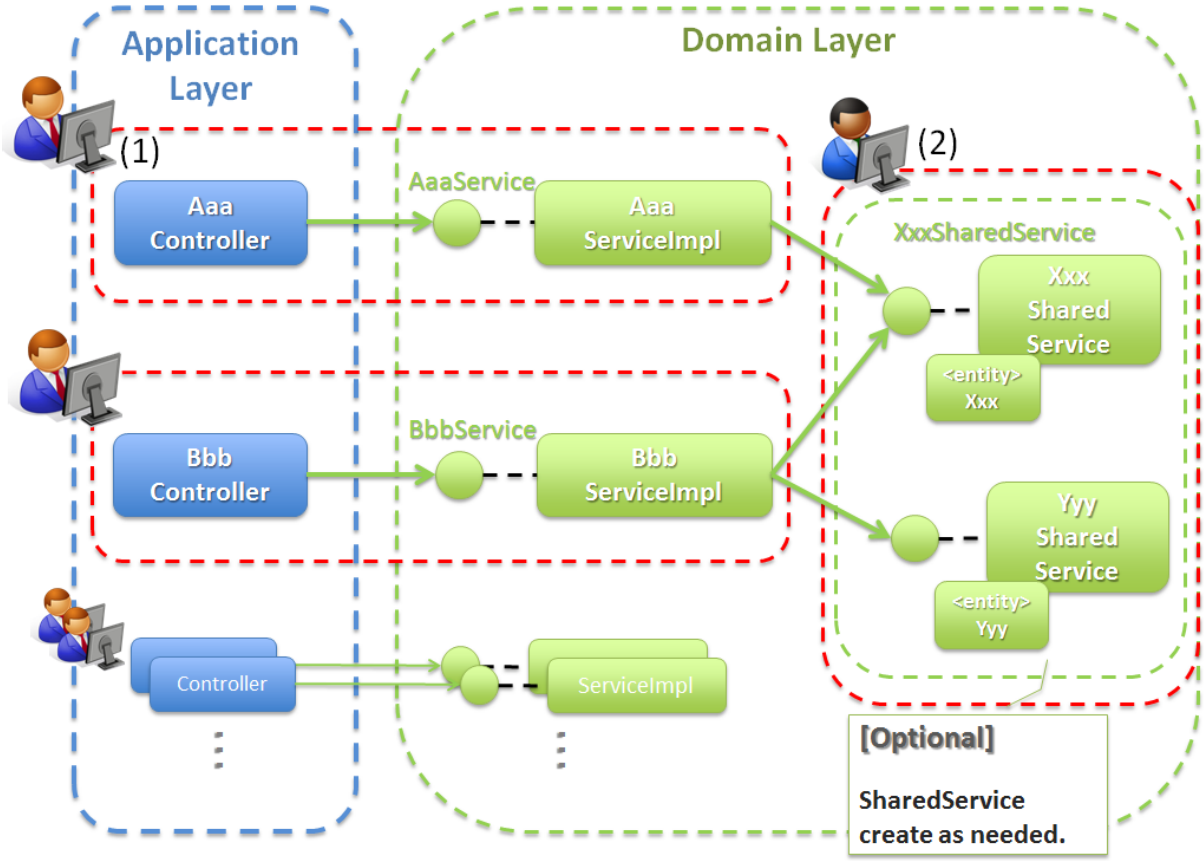
項番	説明
(1)	Entity 毎に開発者を割り当てて、Service を実装する。 特に理由がない場合は、Controller も Entity 毎に作成し、Service と同じ開発者を担当者にすることが望ましい。
(2)	複数の業務ロジックで共有したいロジックがある場合は、SharedService に実装する。 上の図では、別の開発者 (共通チームの担当者) を割り当てているが、プロジェクトの体制によっては (1) と同じ開発者でもよい。

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。

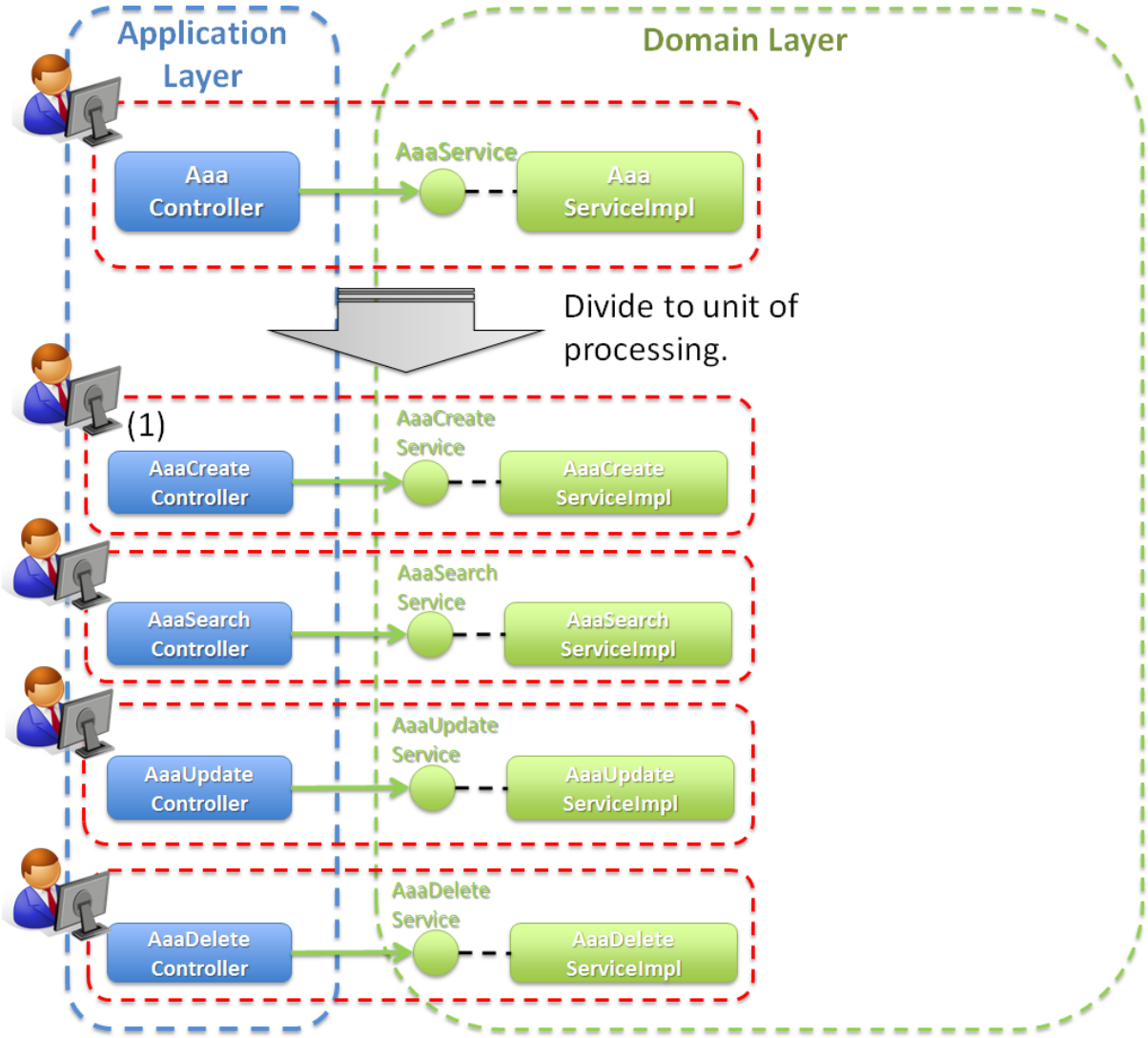


項番	説明
(1)	ユースケース毎に開発者を割り当てて、Service を実装する。 特に理由がない場合は、Controller もユースケース毎に作成し、Service と同じ開発者を担当者にすることが望ましい。
(2)	複数の業務ロジックで共有したいロジックがある場合は、SharedService に実装する。 上の図では、別の開発者 (共通チームの担当者) を割り当てているが、プロジェクトの体制によっては (1) と同じ開発者でもよい。

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。

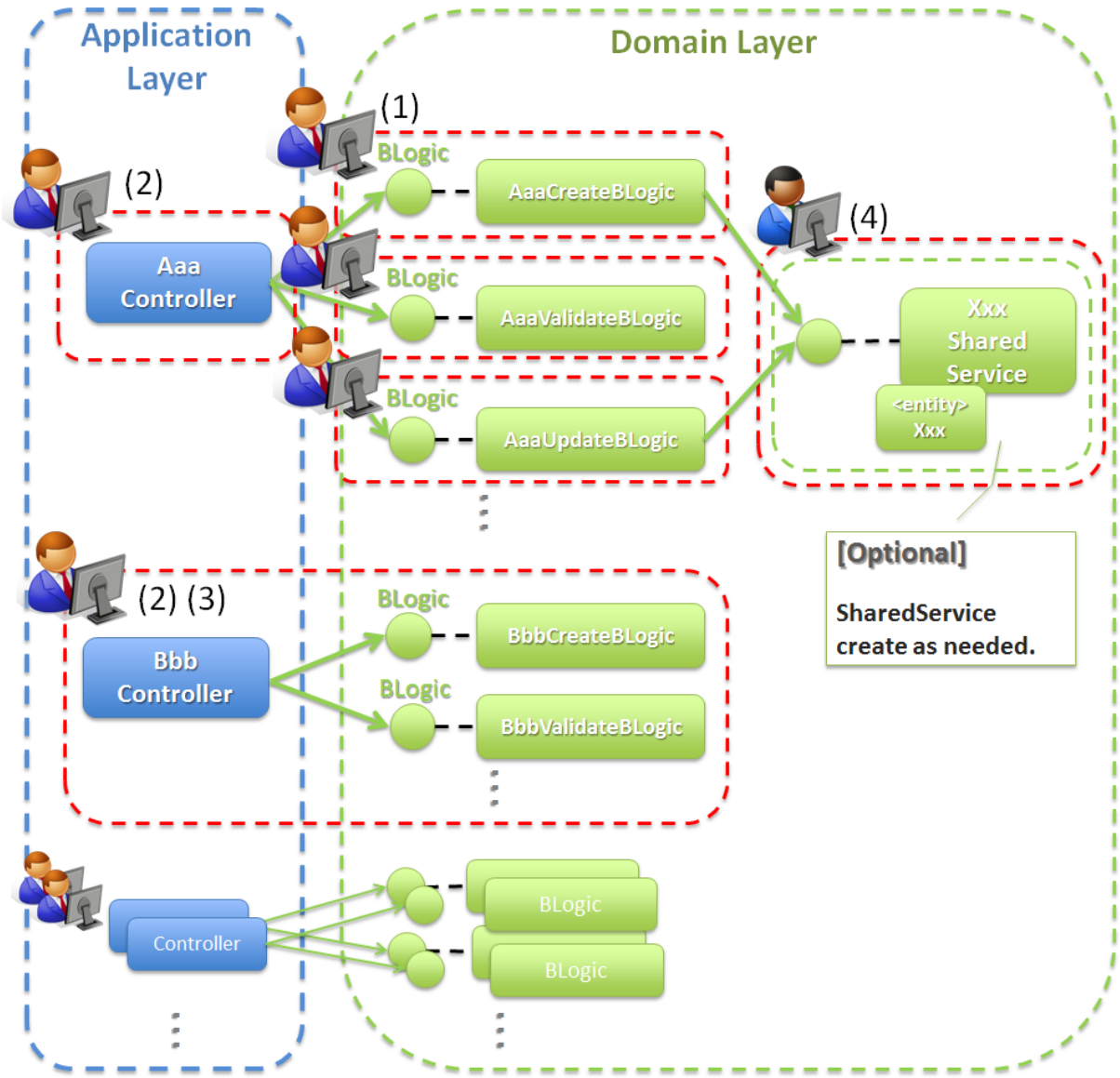


項番	説明
(1)	ユースケースを構成する処理単位に分割し、処理毎に開発者を割り当てて、Service を実装する。 ここで言う処理とは、検索処理、登録処理、更新処理、削除処理といった単位であり、画面から発生するイベント毎の処理ではない点に注意すること。 例えば「更新処理」であれば、「更新対象データの取得」や「更新内容の妥当性チェック」といった単位の処理が複数含まれる。 特に理由がない場合は、Controller も処理毎に作成し、Service と同じ開発者を担当者にすることが望ましい。

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。

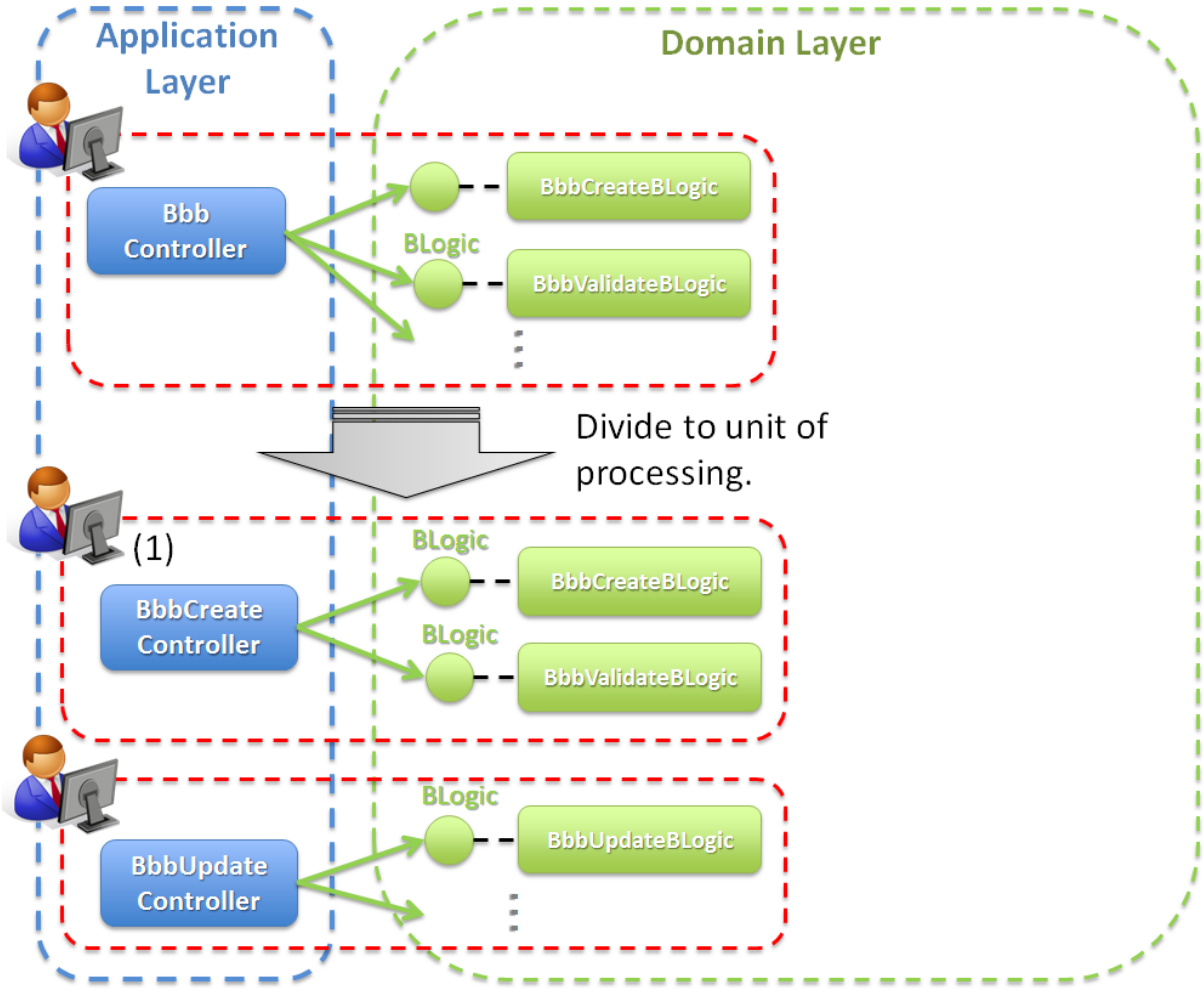


項番	説明
(1)	イベント毎に開発者を割り当てて、Service(BLogic) を実装する。 上記例ではそれぞれ別の担当者を割り当てる図になっているが、これは極端な例である。 実際は、ユースケース毎に担当者を割り当てる事になる。
(2)	特に理由がない場合は、Controller はユースケース毎に作成することが望ましい。
(3)	イベント毎に Service(BLogic) を実装する場合でも、担当者はユースケース毎に割り当てることを推奨する。
(4)	複数の業務ロジックで共有したいロジックがある場合は、SharedService に実装する。 上の図では、別の開発者 (共通チームの担当者) 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 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。
(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。

警告: 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の場合、`@Transactional(readOnly = true)` を設定を行うことで、参照系の処理として必要なトランザクション管理が行われるケースがあるが、JPA を使う場合は「readOnly = true」は意味がないので指定しなくてもよい。詳細は、IBM DeveloperWorks の記事の「Listing 7. Using read-only with REQUIRED propagation mode ― JPA」辺りを参照されたい。

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

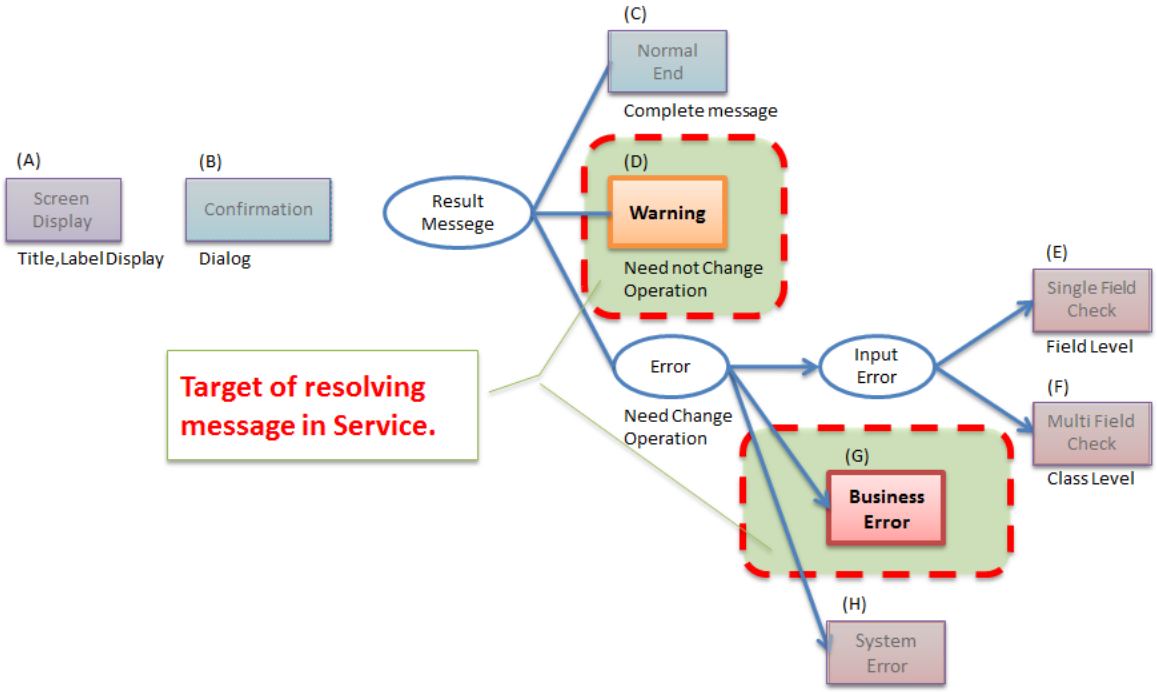
業務データ (Entity) の取得、更新の実装例については、

- JPA を使う場合は、データベースアクセス（JPA 編）
- Mybatis2 を使う場合は、データベースアクセス（Mybatis2 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.fw.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.fw.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.fw.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。

4.1.6 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 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。

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。 **[REQUIRE]** 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。
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。
3	timeout	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。

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 の 2 個めの 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 が一つの場合は指定は不要である。複数の Transaction Manager を使う必要がある場合は、Spring Reference Document 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。

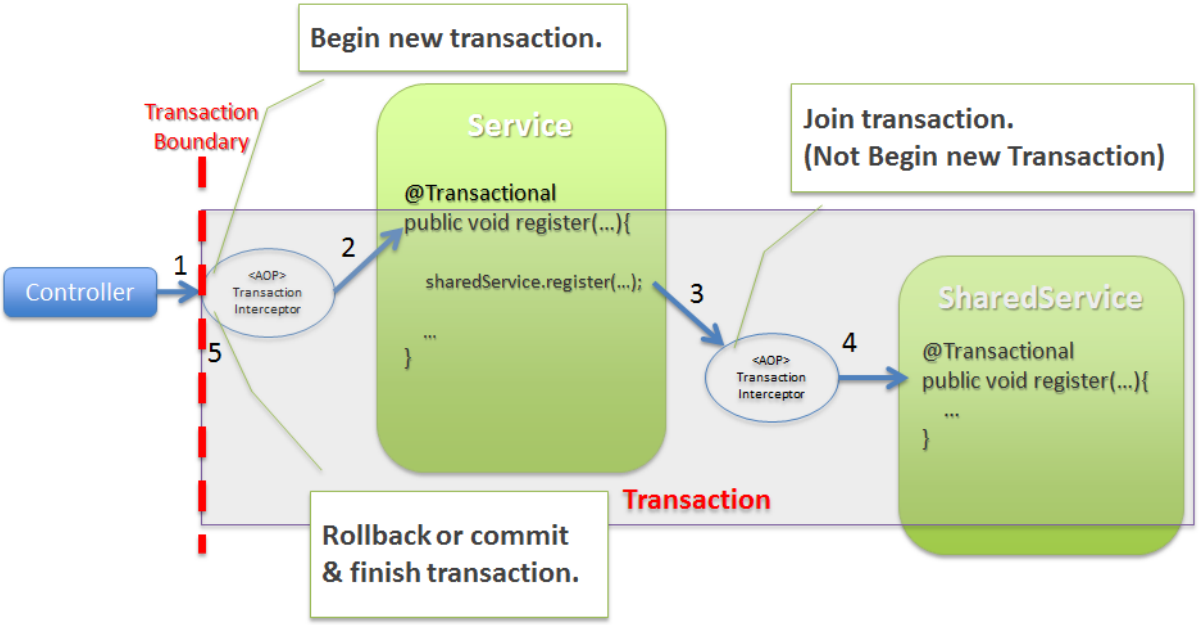
- Oracle : READ_COMMITTED
- DB2 : READ_COMMITTED
- PostgreSQL : READ_COMMITTED
- SQL Server : READ_COMMITTED
- MySQL : REPEATABLE_READ

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。

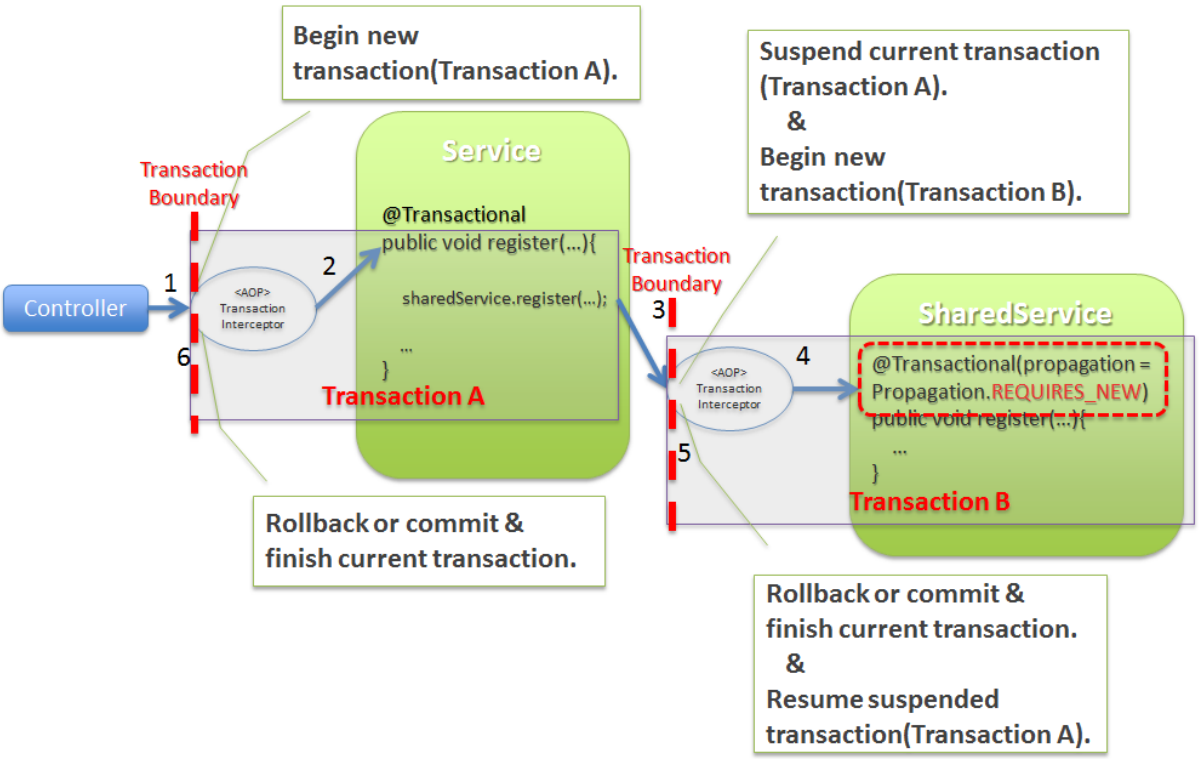
注釈: org.springframework.transaction.UnexpectedRollbackException が発生する理由

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。

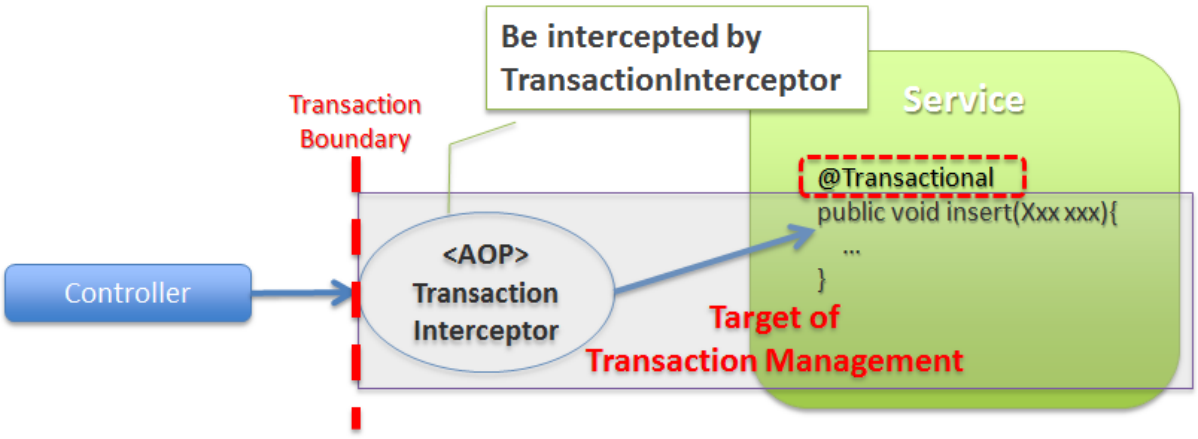
トランザクション管理対象となるメソッドの呼び出し方

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。

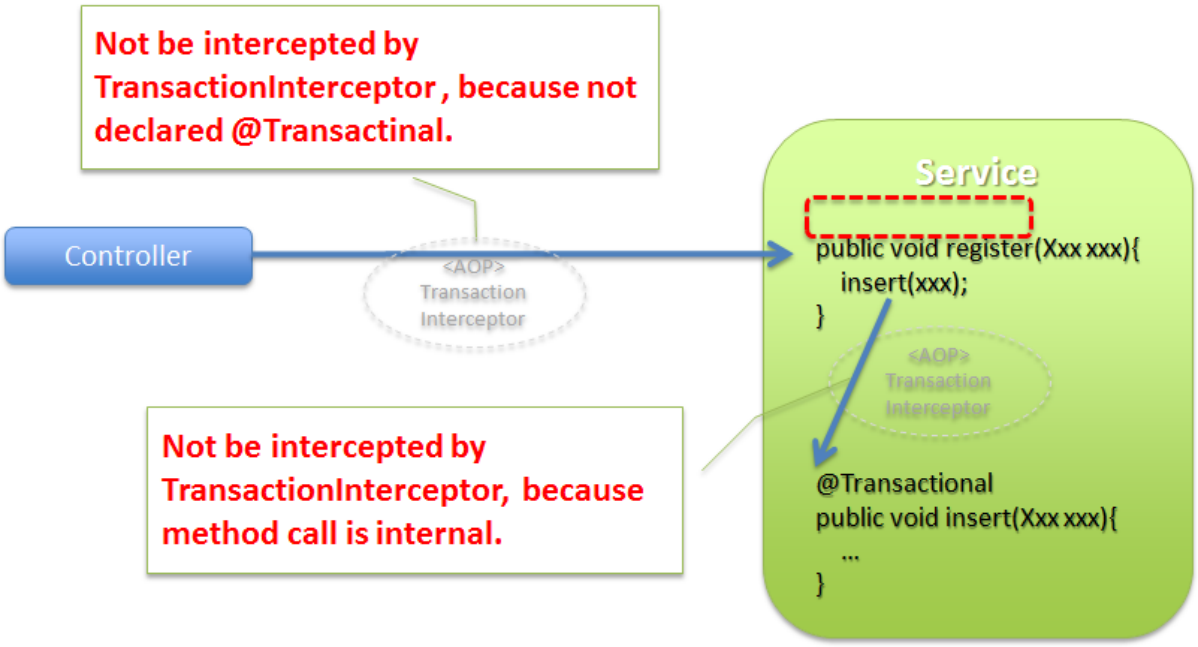
- **トランザクション管理対象となるメソッドの呼び出し方**



- **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" />
</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、Oracle OC4J 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。

TABLE 4.1 **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。

<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。

4.1.7 Appendix

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。

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 を参照されたい。

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。

- Controller

```
// (2)
@Inject
XxxBLogic<XxxInput, XxxOutput> xxxBLogic;

public String reserve(XxxForm form, RedirectAttributes redirectAttributes) {

    XxxInput input = new XxxInput();
    // omitted

    // (3)
    XxxOutput output = xxxBlogic.execute(input);

    // omitted

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

項番	説明
(2)	Controller は、呼び出す BLogic インタフェースを Inject する。
(3)	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。

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

```
public abstract class AbstractBLogic<I, O> implements BLogic<I, O> {

    public O execute(I input){
      try{

          // omitted

          // (4)
          preExecute(input);

          // (5)
          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
(4)	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。
(5)	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)

```
public class XxxBLogic extends AbstractBLogic<XxxInput, XxxOutput> {

    // (6)
    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)
    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。

4.1.8 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。

4.2 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。

4.2.1 RepositoryImpl の実装

以下に、JPA と MyBatis2 系を使って、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。

- *JPA* **を使って** *Repository* **を実装**
- *MyBatis2* **系を使って** *Repository* **を実装**

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。

MyBatis2 系を使って 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 MyBatis2 系を使う場合、RepositoryImpl は、以下のようになる。

MyBatis2 の使用方法の詳細は、データベースアクセス（Mybatis2 編）を参照されたい。

なお、本ガイドラインでは MyBatis を直接使うのではなく、MyBatis の API をラップしている TERASOLUNA DAO を使うことを前提としている。

MyBatis を使う場合、Repository インタフェースは、**必要なメソッドの定義のみ行えばよい。**

もちろん、Spring Data から提供されている CrudRepository や、PagingAndSortingRepository を使ってもよいが、すべてのメソッドを使うケースは稀なので、余計な実装が必要になってしまう。

MyBatis を使う場合、Repository インタフェースの定義に加え、RepositoryImpl の実装と、SQL 定義ファイルの実装が必要となる。

下記に、以下 2 点を目的とした、JpaRepository の親インタフェースである PagingAndSortingRepository を実装例を示す。

1. 汎用的な CRUD 操作を MyBatis で実装する際のサンプルの提示
2. Spring Data JPA の仕組みを使って Repository を実装した時との実装比較

- TodoRepository.java

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

項番	説明
(1)	Spring Data より提供されている org.springframework.data.repository.PagingAndSortin の子インタフェース) を継承することで、 Repository インタフェースとして必要な、基本的なメソッドの定義が行われる。MyBatis の場合、インタフェースの定義に加えて、RepositoryImpl の実装も必要である。

- TodoRepositoryImpl.java

```
@Repository // (1)
@Transactional // (2)
public class TodoRepositoryImpl implements TodoRepository {
    @Inject
    QueryDAO queryDAO; // (3)

    @Inject
    UpdateDAO updateDAO; // (4)

    @Override
    @Transactional(readOnly = true) // (5)
    public Todo findOne(String id) { // (6)
        return queryDAO.executeForObject("todo.findOne", todoId, Todo.class);
    }

    @Override
    @Transactional(readOnly = true) // (5)
    public boolean exists(String id) { // (6)
        Long count = queryDAO.executeForObject("todo.exists", todoId,
            Long.class);
        return 0 < count.longValue();
    }

    @Override
    @Transactional(readOnly = true) // (5)
    public Iterable<Todo> findAll() { // (6)
        return findAll((Sort) null);
    }

    @Override
    @Transactional(readOnly = true) // (5)
    public Iterable<Todo> findAll(Iterable<String> ids) { // (6)
        return queryDAO.executeForObjectList("todo.findAll", ids);
    }

    @Override
    @Transactional(readOnly = true) // (5)
```

```
    public Iterable<Todo> findAll(Sort sort) { // (7)
        return queryDAO.executeForObjectList("todo.findAllSort", sort);
    }

    @Override
    @Transactional(readOnly = true) // (5)
    Page<Todo> findAll(Pageable pageable) { // (7)
        long count = count();
        List<Todo> todos = null;
        if(0 < count){
            todos = queryDAO.executeForObjectList("todo.findAllSort",
                pageable.getSort(),pageable.getOffset(),pageable.getPageSize());
        } else {
            todos = Collections.emptyList();
        }
        Page page = new PageImpl(todos,pageable,count);
        return page;
    }

    @Override
    @Transactional(readOnly = true) // (5)
    public long count() { // (6)
        Long count = queryDAO.executeForObject("todo.count", null, Long.class);
        return count.longValue();
    }

    @Override
    public <S extends Todo> S save(S todo) { // (6)
        if(exists(todo.getTodoId())){
            updateDAO.execute("todo.update", todo);
        } else {
            updateDAO.execute("todo.insert", todo);
        }
        return todo;
    }

    @Override
    public <S extends Todo> Iterable<S> save(Iterable<S> todos) { // (6)
        for(Todo todo : todos){
            save(todo);
        }
        return todos;
    }

    @Override
    public void delete(String id) { // (6)
        updateDAO.execute("todo.delete", id);
    }

    @Override
    public void delete(Todo todo) { // (6)
```

```
            delete(todo.getTodoId());
        }

        @Override
        public void delete(Iterable<? extends Todo> todos) { // (6)
            for(Todo todo : todos){
                delete(todo);
            }
        }

        public long countByFinished(boolean finished) { // (8)
            Long count = queryDAO.executeForObject("todo.countByFinished", finished, Long.cla
            return count.longValue();
        }

    }
```

項番	説明
(1)	クラスアノテーションとして、`@Repository` アノテーションを付与する。アノテーションを付与することで、component-scan 対象となり、設定ファイルへの bean 定義が不要となる。
(2)	クラスアノテーションとして、`@Transactional` アノテーションを付与する。トランザクション境界は、Service で制御するが、Repository にも付与しておくこと。
(3)	問い合わせ処理を行うための `jp.terasoluna.fw.dao.QueryDAO` をインジェクションする。
(4)	更新処理を行うための `jp.terasoluna.fw.dao.UpdateDAO` をインジェクションする。
(5)	問い合わせ系のメソッドには、`@Transactional(readOnly = true)` を付与する。
(6)	`CrudRepository` で定義されているメソッドを実装している。
(7)	`PagingAndSortingRepository` で定義されているメソッドを実装している。
(8)	`TodoRepository` で追加したメソッドを実装している。

- sqlMap.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
            PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
            "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="todo"> <!-- (1) -->

    <resultMap id="todo" class="todo.domain.model.Todo"> <!-- (2) -->
        <result 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" parameterClass="java.lang.String" resultMap="todo">
            <!-- ... -->
        </select>

        <select id="exists" parameterClass="java.lang.String" resultClass="java.lang.Long">
            <!-- ... -->
        </select>

        <select id="findAll" resultMap="todo">
            <!-- ... -->
        </select>

        <select id="findAllSort" parameterClass="org.springframework.data.domain.Sort"
                resultMap="todo">
            <!-- ... -->
        </select>

        <select id="count" resultClass="java.lang.Long">
            <!-- ... -->
        </select>

        <insert id="insert" parameterClass="todo.domain.model.Todo">
            <!-- ... -->
        </insert>

        <update id="update" parameterClass="todo.domain.model.Todo">
            <!-- ... -->
        </update>

        <delete id="delete" parameterClass="todo.domain.model.Todo">
            <!-- ... -->
        </delete>

        <select id="countByFinished" parameterClass="java.lang.Boolean" resultClass="java.lang.
            <!-- ... -->
        </select>

    </sqlMap>
```

項番	説明
(1)	namespace を指定する。Entity を一意に特定できる名前を付与する。
(2)	Entity の型の指定とフィールドとカラムのマッピングを行う。
(3)	SQLID 毎に SQL を実装する。

RestTemplate を使って外部システムと連携する Repository を実装

課題

TBD

次版以降で詳細化する予定である。

4.3 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。

4.3.1 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。

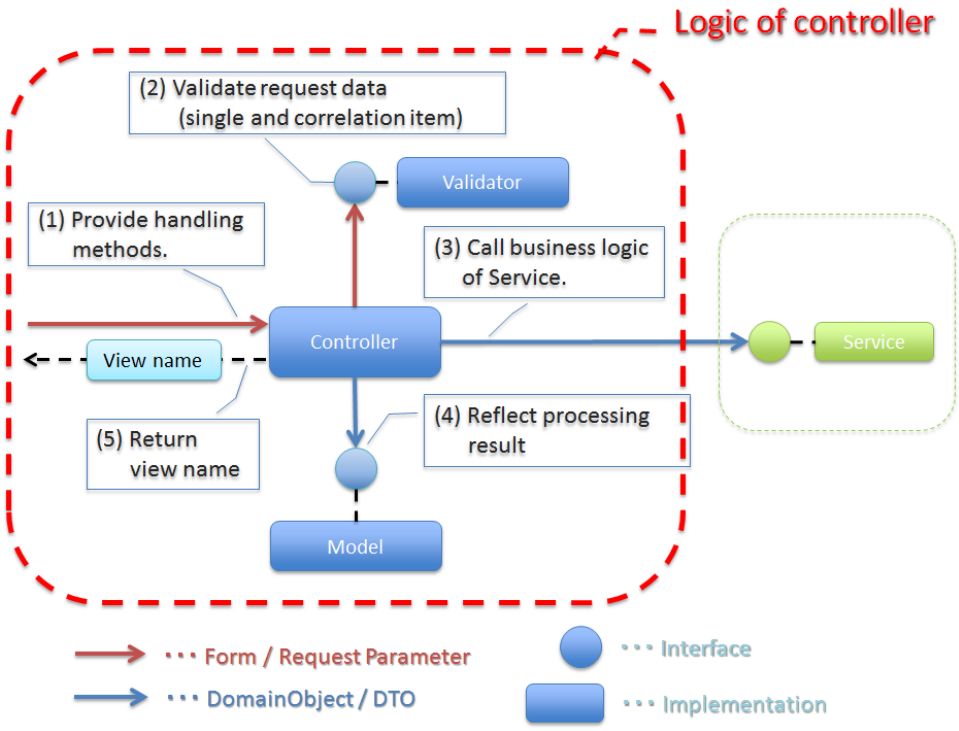


図 4.1　**Picture - Logic of controller**

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
- リクエストと処理メソッドのマッピング方法
- 処理メソッドの引数について
- 処理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 (Interfece-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 {
    // ...
}
```

リクエストと処理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「処理メソッド」と呼ぶ。

```
@RequestMapping(value = "hello")
public String hello() {
    // ...
}
```

リクエストと処理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。

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 クラスに処理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 Annotaion-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処理メソッドが 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 Spring Framework 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

リクエストと処理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処理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、リクエストと処理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*

- リクエストと処理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。

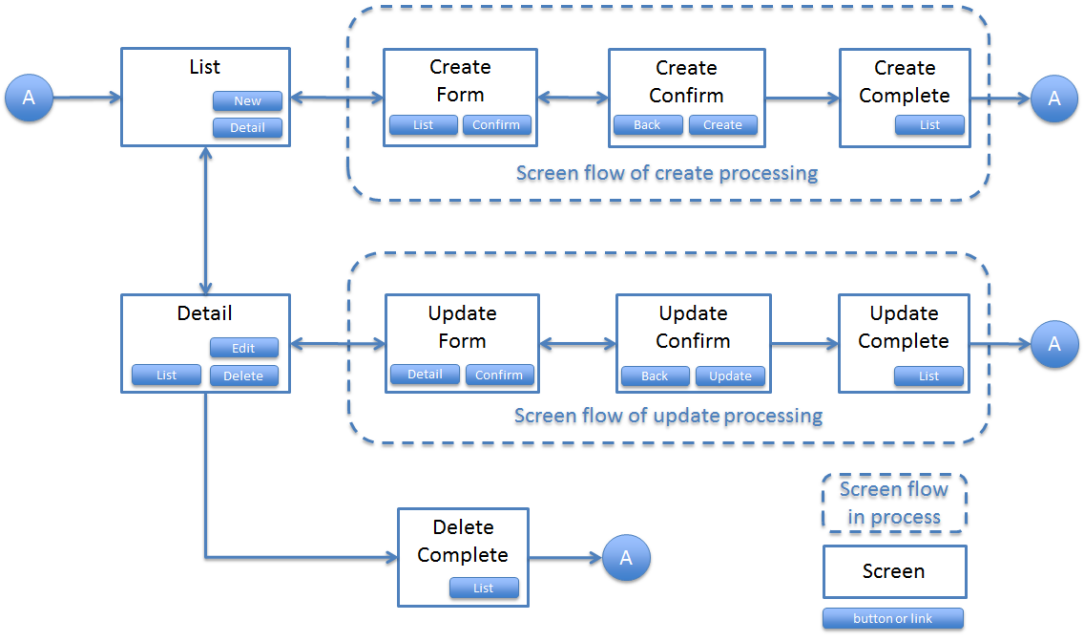


図 4.2　**Picture - Screen flow of entity management function**

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。

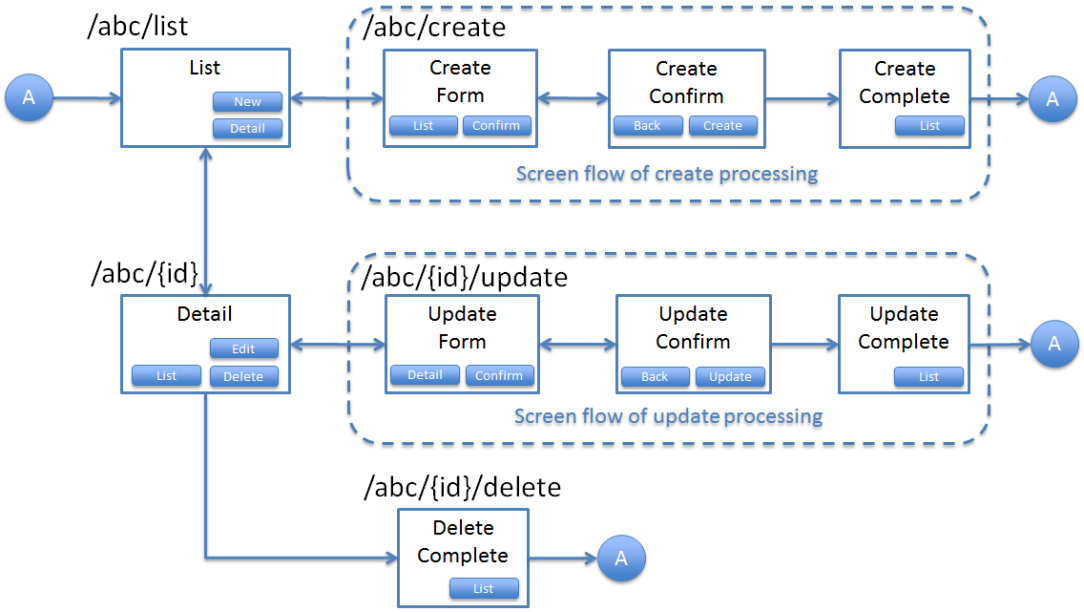


図 4.3　**Picture - Screen flow of entity management function and assigned URL**

リクエストと処理メソッドのマッピング

リクエストと処理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	処理メソッド
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	createComple
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
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	updateComple
13.	Entity 削除	/abc/{id}/delete	削除	POST	-	delete
14.			削除完了表示	GET	complete	deleteComple

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処理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処理メソッドを切り替えている点に注目すること。

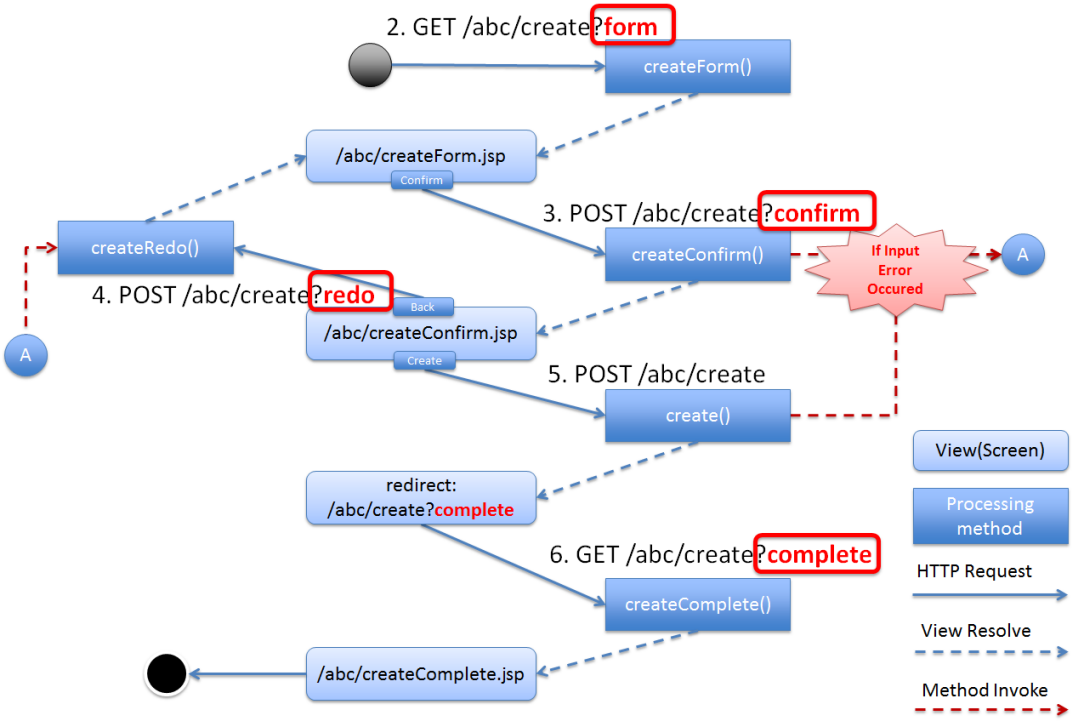


図 4.4 **Picture - Request flow of entity create processing**

以下に、Entity 新規作成処理の処理メソッドの実装コードを示す。

ここではリクエストと処理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。

処理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) {
    // ommited
    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、処理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、処理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;

    // ommited 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>
```

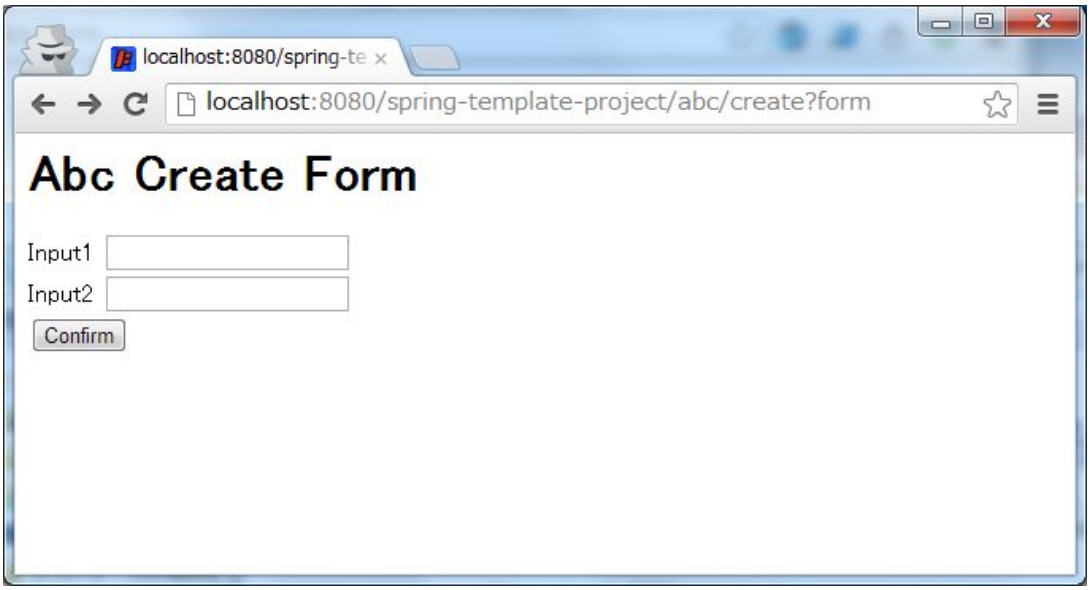
項番	説明
(1)	確認画面へ遷移するためにの submit ボタンには `name="confirm"`というパラメータを指定しておく。

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)
        }
        // ommited
        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処理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、処理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、処理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。

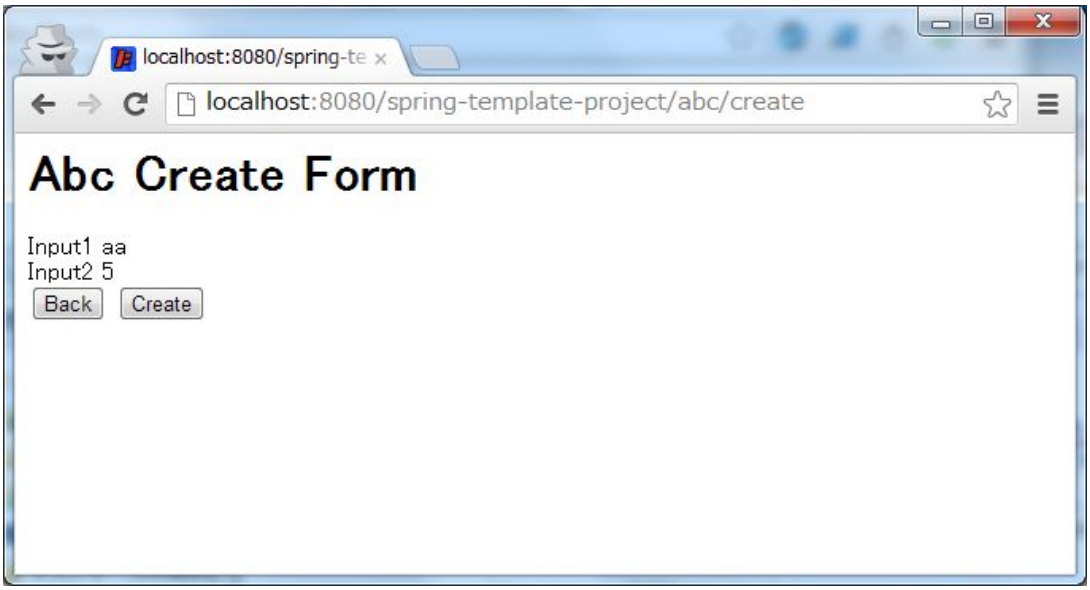
入力内容確認表示表示処理を呼び出す。

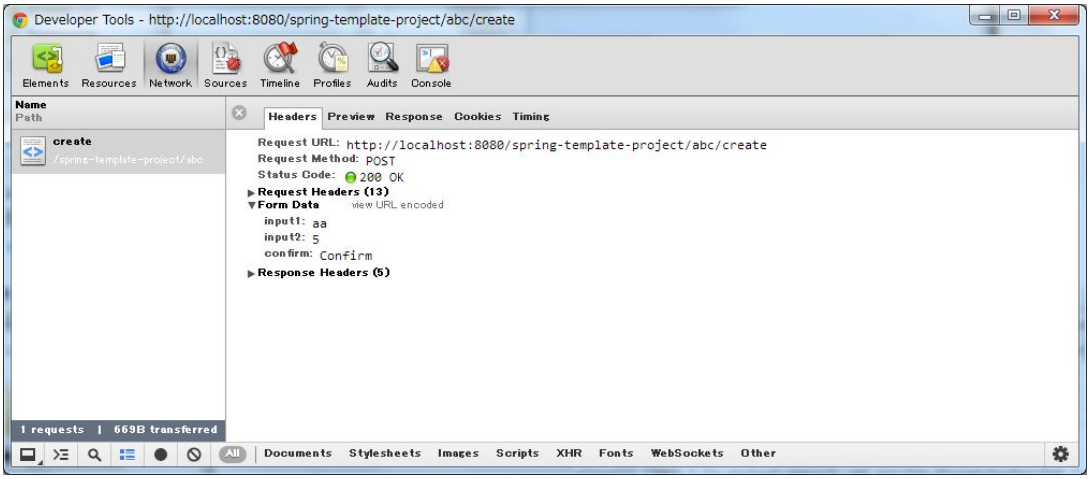
フォーム画面で Input1 に `"aa"` を、Input2 に `"5"` を入力し、Confirm ボタンを押下する。

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) {
        // ommited
        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。

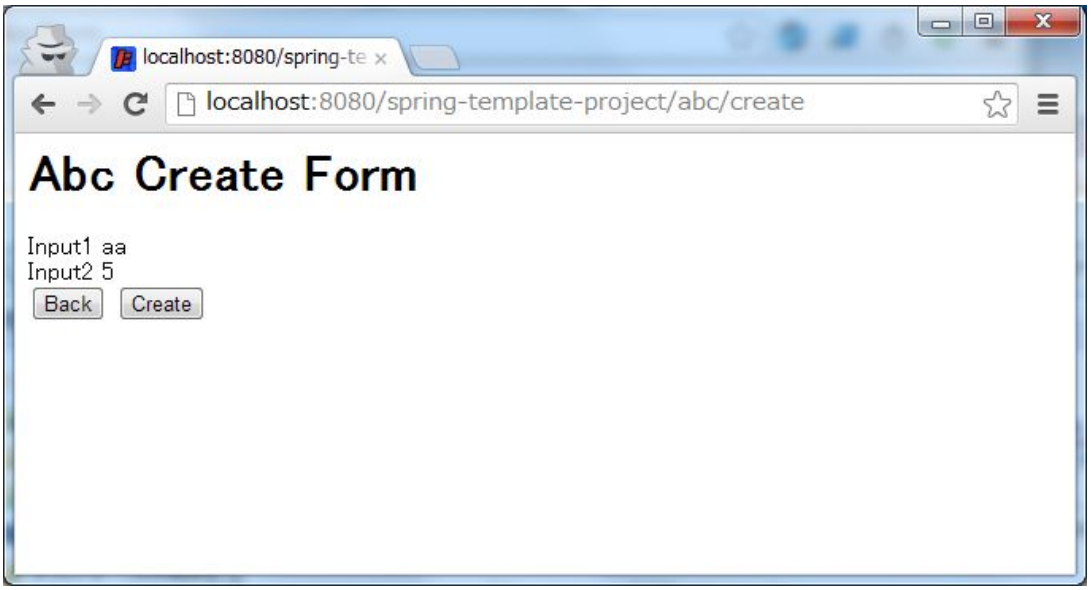
以下に、フォーム再表示の動作について説明する。

フォーム再表示リクエストを呼び出す。

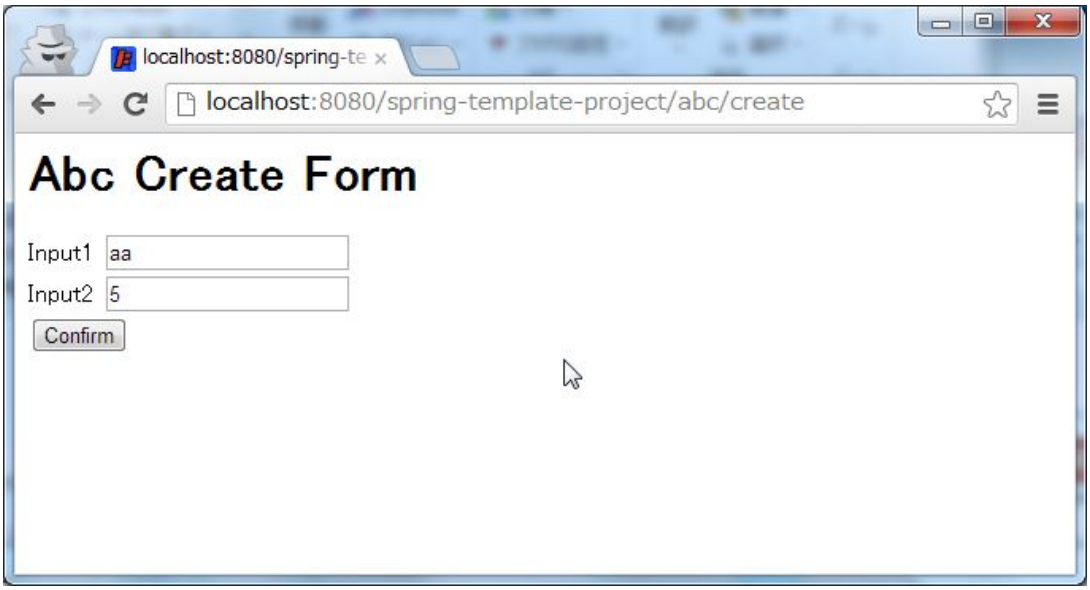
入力内容確認画面で、Back ボタンを押下する。

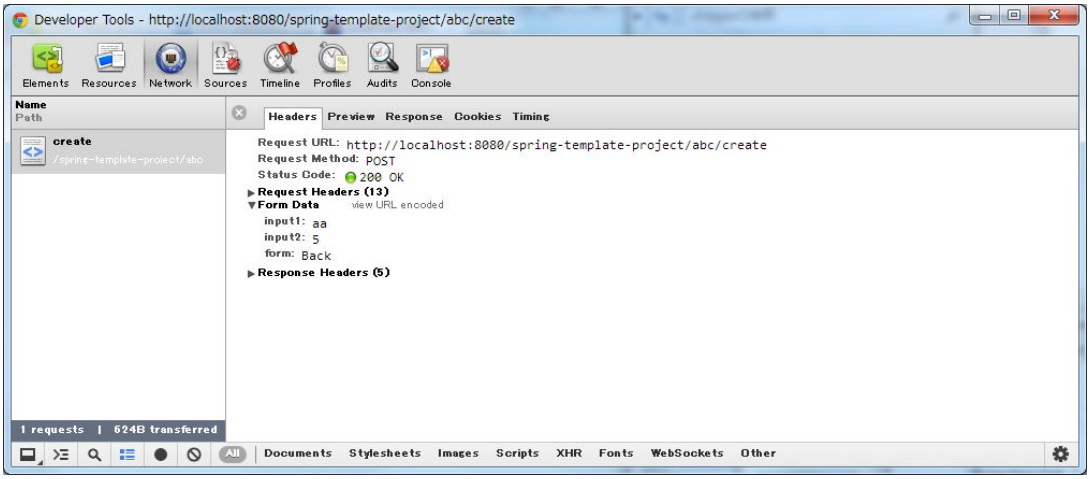
Back ボタンを押下すると、 `abc/create?redo` という URI に POST メソッドでアクセスする。

`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";
        }
        // ommited
        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。

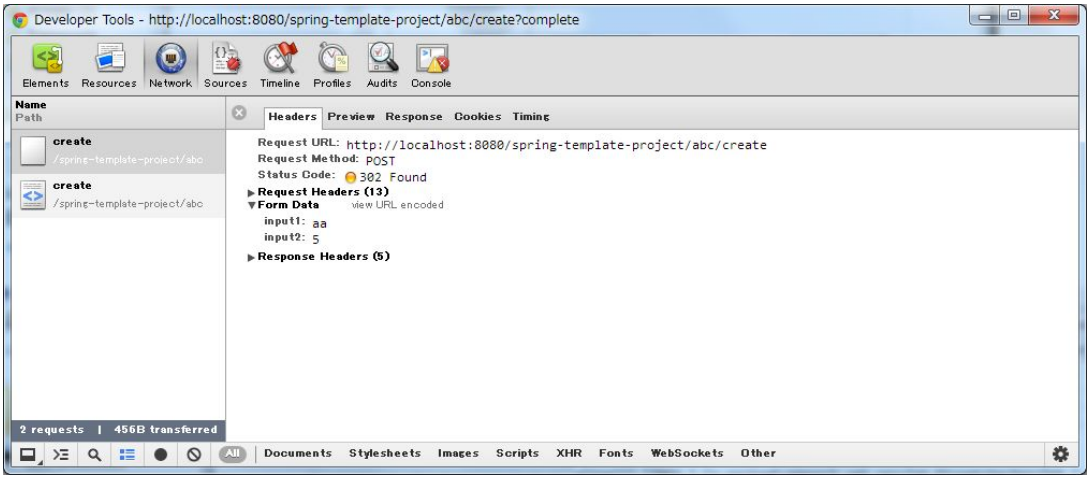
新規作成処理を呼び出す。

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。

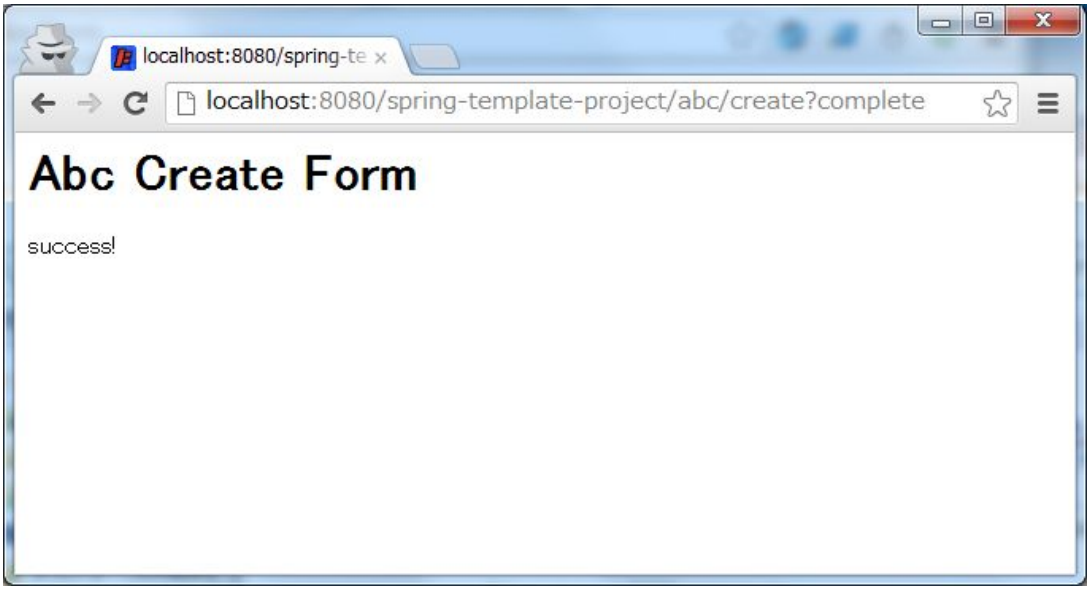
```
@RequestMapping(value = "create", params = "complete") // (1)
public String createComplete() {
    // ommited
    return "abc/createComplete"; // (2)
}
```

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。

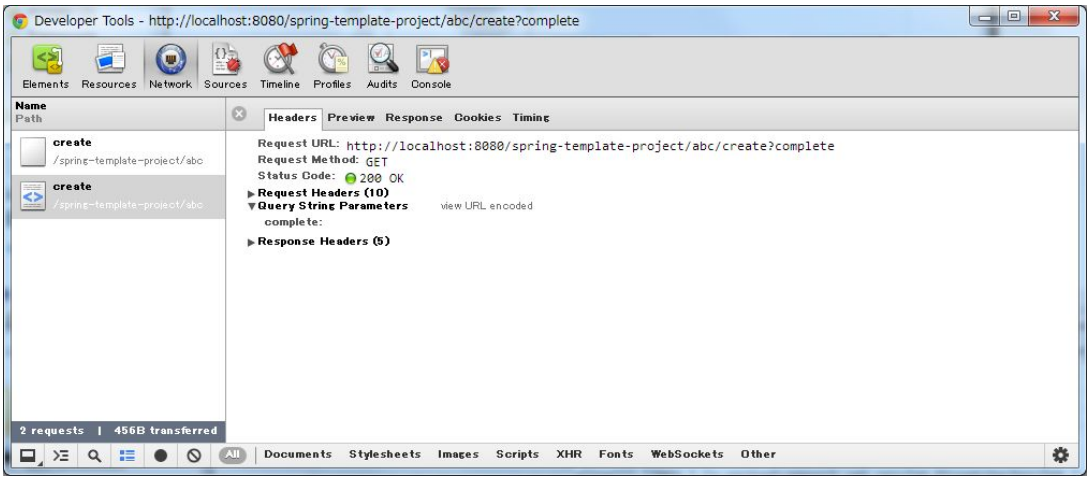
注釈: この処理も HTTP メソッドを GET に限る必要がないので method 属性を指定しなくても良い。

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、実行する処理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。

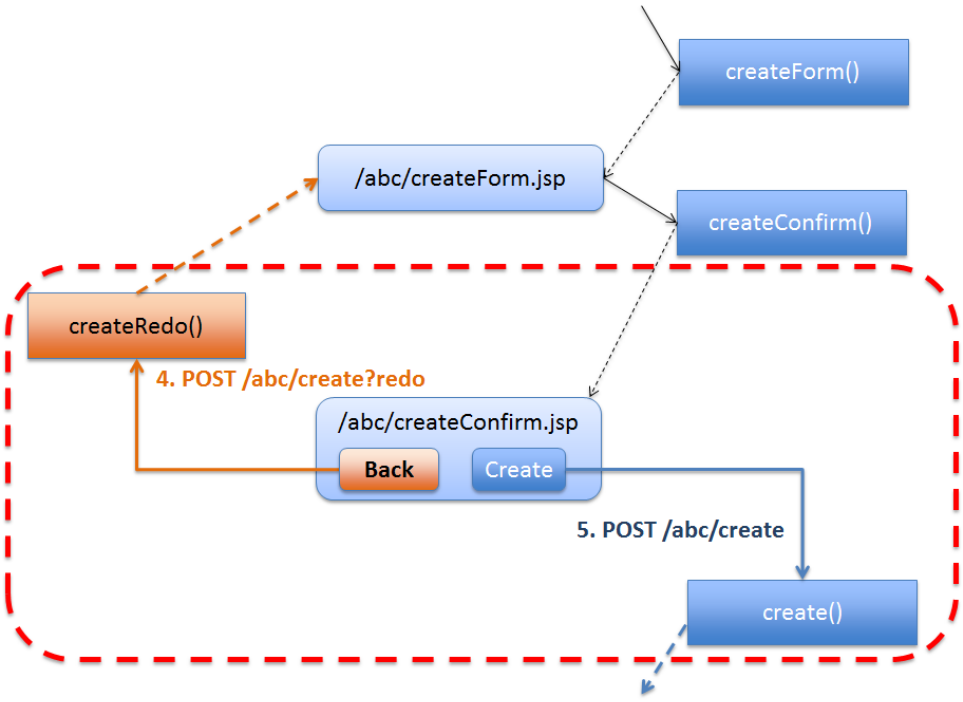


図 4.5　**Picture - Multiple button in the HTML form**

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) {
        // ommited
        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);
        }
        // ommited
        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) {
            // ommited
            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);
            }
            // ommited
            return "redirect:/abc/create?complete";
        }

        // Handling request of "/abc/create?complete"
        @RequestMapping(value = "create", params = "complete")
        public String createComplete() {
            // ommited
            return "abc/createComplete";
        }

    }
```

処理メソッドの引数について

処理メソッドの引数は 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` や `HttpSession` 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。

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) を処理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、この方法はコンパイルを debug モードで行う必要がある。

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 debug モードでコンパイルする必要がある。debug モードでコンパイルした場合、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") Stri
            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、この方法はコンパイルを debug モードで行う必要がある。
(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は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処理メソッドを、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処理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") Stri
            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、処理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

処理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) を処理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)	リダイレクト後の処理メソッドでは、(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。
(6)	View(JSP) 側では、「${属性名}」と記述することで `RedirectAttributes` オブジェクトに追加したデータを取得することができる。 例では HTML エスケープを行う EL 式の関数を呼び出しているため、「${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。
(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。

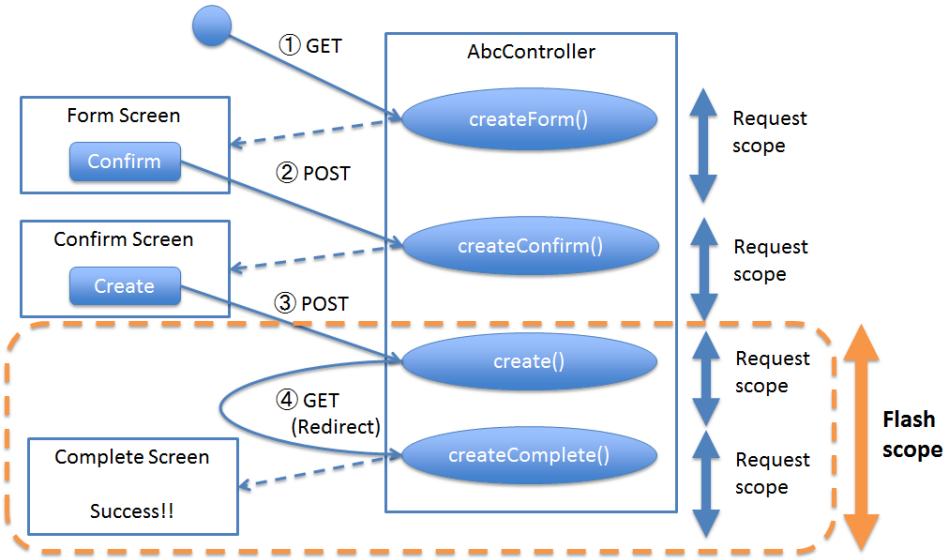


図 4.6 **Picture - Survival time of flush scope**

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", "hello world!");
        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 "hello world!" 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。

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) オブジェクトを処理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` を処理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` オブジェクトを処理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。

処理メソッドの返り値について

処理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 論理名)

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

処理メソッドの実行結果を HTML として応答する場合、処理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 (`InternalViewResolver` や `TilesViewResolver` 等) となる。

以下では、InternalViewResolver 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

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" /> <!-- (1) -->
    <property name="suffix" value=".jsp" /> <!-- (2) -->
    <property name="order" value="1" /> <!-- (3) -->
</bean>
```

- SampleController.java

```
@RequestMapping("hello")
public String hello() {
    // ommited
    return "sample/hello"; // (4)
}
```

項番	説明
(1)	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。 プレフィックスを指定しておくことで、Controller で View 名を返却する際に、JSP の物理的な格納場所を意識する必要がなくなる。
(2)	JSP ファイルの拡張子 (ファイルパスのサフィックス) を指定する。 サフィックスを指定しておくことで、Controller で View 名を返却する際に、JSP の拡張子を意識する必要がなくなる。
(3)	複数の ViewResolver を指定した場合の実行順番を指定する。Integer の範囲で指定することが可能で、値が小さいものから順に実行される。
(4)	処理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、処理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。

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。

処理メソッドでは、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

```
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver"> <!-- (1) -->
     <property name="order" value="0" /> <!-- (2) -->
</bean>
```

- SampleController.java

```
@RequestMapping("report")
public String report() {
    // ommited
    return "sample/report"; // (3)
}
```

- XxxExcelView.java

```
@Component("sample/report") // (4)
public class XxxExcelView extends AbstractExcelView { // (5)
    @Override
    protected void buildExcelDocument(Map<String, Object> model,
            HSSFWorkbook workbook, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
```

```
            HSSFSheet sheet;
            HSSFCell 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)	BeanNameViewResolver は返却された View 名に一致する Bean をアプリケーションコンテキストから探して View を解決するクラスとなっている。
(2)	InternalViewResolver や TilesViewResolver と併用する場合は、これらの ViewResolver より、高い優先度を指定する事を推奨する。 上記例では、 "0" を指定することで、InternalViewResolver より先に BeanNameViewResolver による View 解決が行われる。
(3)	処理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、 (4) 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。
(4)	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。
(5)	View の実装例。上記例では、 org.springframework.web.servlet.view.document.AbstractExcelView 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 の処理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、処理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 void 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 void hello(@Validated SampleForm form, BindingResult result, Model model){
        // ommited
        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 の処理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処理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 void hello(@Validated SampleForm form, BindingResult result){
    // ommited
    String message = sampleHelper.hello(form); // (2)
    model.addAttribute("message", message);
    return "sample/hello";
}
```

- SampleHelper.java

```
public class SampleHelper {

    @Inject
    SampleService sampleService;

    public void 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 void hello(SampleForm form, BindingResult result, Model model){
        // ommited
        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 の処理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処理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 void hello(@Validated SampleForm form, BindingResult result){
    // ommited
    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 ...
    // ...
}
```

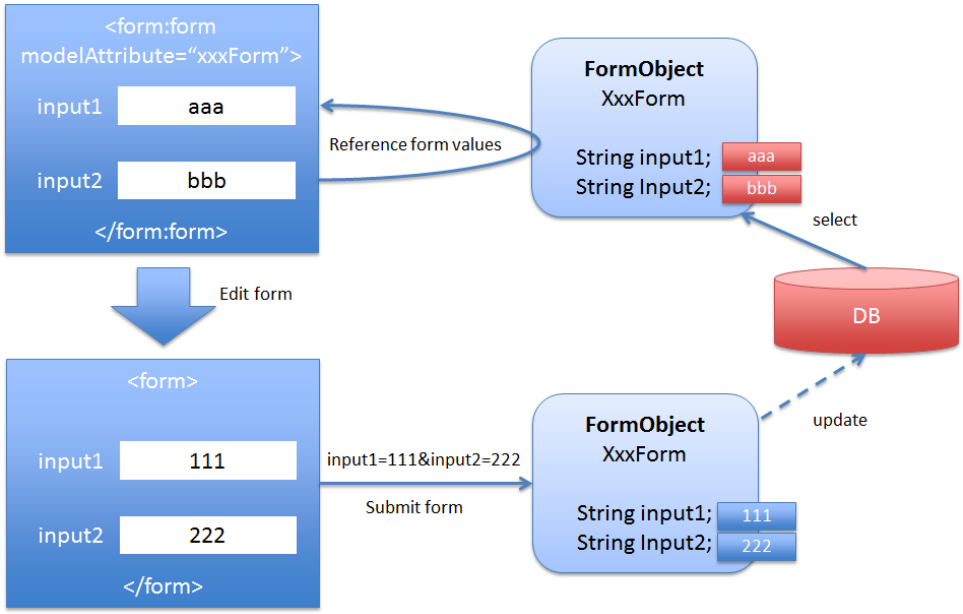
項番	説明
(1)	ドメインオブジェクトの値をフォームオブジェクトに反映するためのメソッドを呼び出す。
(2)	ドメインオブジェクトの値をフォームオブジェクトに反映するためのメソッドにて、ドメインオブジェクトの値をフォームオブジェクトに反映する。

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。

4.3.2 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、処理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 3 Type Conversion
- Spring 3 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（NUMBER,CURRENCY,PERCENT）を指定する。詳細は、Spring Framework 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、JAVASE 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、Spring Framework の Javadoc を参照。
2.	pattern	Java の日時形式を指定する。詳細は、JAVASE 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、処理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、処理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 の処理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毎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を 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、処理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 の処理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 の処理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。処理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

4.3.3 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**
- *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	• http://static.springsource.org/spring/docs/3.2.16.RELEASE/spring-framework-reference/html/view.html#view-jsp-formtaglib • http://static.springsource.org/spring/docs/3.2.16.RELEASE/spring-framework-reference/html/spring-form.tld.html
2.	Spring's tag library	• http://static.springsource.org/spring/docs/3.2.16.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。

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) を行う必要ががある場合は、JavaScript を使用して、項目の制御を行う。

課題

TBD

次版以降で詳細を記載する予定である。

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。

4.3.4 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 の処理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,
```

```
            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> <!-- (
    </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処理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 の処理メソッドを実行する前

  `HandlerInterceptor#preHandle` メソッドとして実装する。

- Controller の処理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 の処理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) -->
            <mvc:exclude-mapping path="/**/*.html" />
            <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 の処理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 の処理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 では、全ての Controller で実行したい処理を実装する。

@ControllerAdvice では、以下の処理を共通化することができる。

- @InitBinder メソッドの共通化
- @ExceptionHandler メソッドの共通化
- @ModelAttribute メソッドの共通化

以下に、@InitBinder メソッドのサンプルを示す。
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)
@Component // (2)
@Order(0) // (3)
public class SampleControllerAdvice {

    // (4)
    @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)	`@Component` アノテーションを付与することで、component-scan 対象にしている。
(3)	`@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。ControllerAdvice を複数作成する場合は、必ず指定すること。
(4)	`@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。

4.3.5 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。

4.3.6 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。

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 5 章

TERASOLUNA Global Framework の機能詳細

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。

5.1 データベースアクセス（共通編）

課題

TBD

本章では以下の内容について、現在精査中である。

- 複数データソースについて
  具体的な内容は、*Overview* **の複数データソースについて**および *How to extends* **の複数データソースについて**を参照されたい。
- JPA 利用時の一意制約エラー及び悲観排他エラーのハンドリング方法について
  具体的な内容は、*Overview* **の例外ハンドリングについて**を参照されたい。

5.1.1 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（JPA 編）、データベースアクセス（Mybatis2 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。

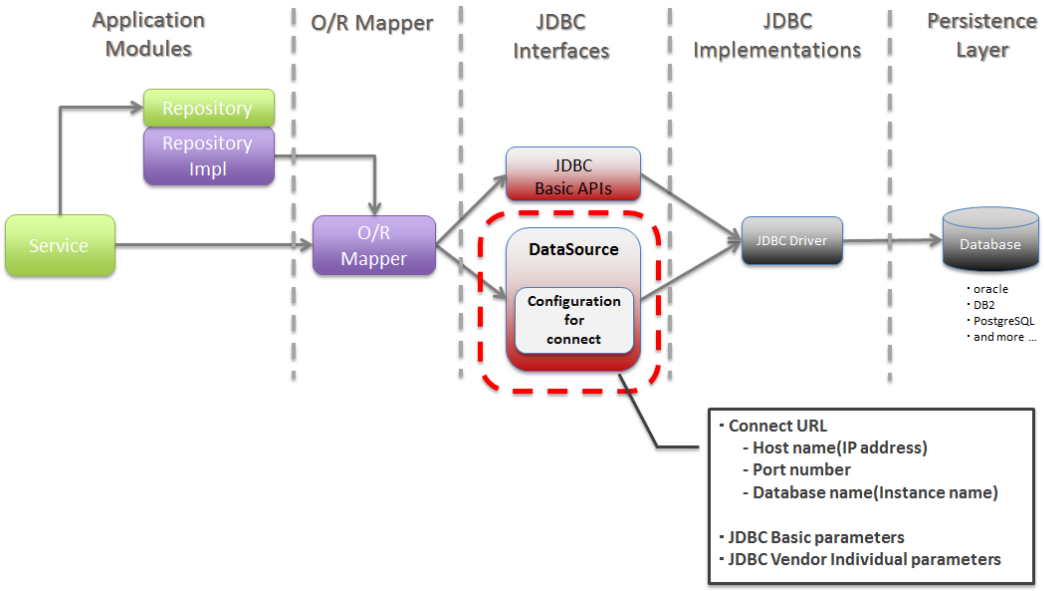


図 5.1 **Picture - About JDBC DataSource**

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**
- ***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。

TABLE 5.1 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 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。
2.	Oracle WebLogic Server 12c	Oracle WebLogic Server Product Documentation を参照されたい。
3.	IBM WebSphere Application Server Version 8.5	WebSphere Application Server Online information center を参照されたい。
4.	Resin 4.0	Resin 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。

TABLE 5.2 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。

TABLE 5.3 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。
- 悲観ロックに失敗した場合、`PessimisticLockingFailureException` ではなく、

org.springframework.dao.UncategorizedDataAccessException の子クラスが発生する。

悲観エラー時に発生する UncategorizedDataAccessException は、システムエラーに分類される例外なので、アプリケーションでハンドリングすることは推奨されないが、最悪ハンドリングを行う必要があるかもしれない。原因例外には、悲観ロックエラーが発生したことを通知する例外が格納されているので、ハンドリングできる。

⇒継続調査。

現状以下の動作となる。

- PostgreSQL + for update nowait
  - org.springframework.orm.hibernate3.HibernateJdbcException
  - Caused by: org.hibernate.PessimisticLockException
- Oracle + for update
  - org.springframework.orm.hibernate3.HibernateSystemException
  - Caused by: Caused by: org.hibernate.dialect.lock.PessimisticEntityLockException
  - Caused by: org.hibernate.exception.LockTimeoutException
- Oracle / PostgreSQL + 一意制約
  - org.springframework.dao.DataIntegrityViolationException
  - Caused by: org.hibernate.exception.ConstraintViolationException

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。

TABLE 5.4 **複数のデータソースが必要になるう代表的なケース**

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。

課題

TBD

今後、以下の内容を追加する予定である。

- 概念レベルのイメージ図
- 上記２ケースについての詳細。特に、(1) のケースは、処理パターン（複数のデータソースに対して更新あり、更新は１つのデータソース、参照のみ、同時アクセスはなしなど）によってトランザクション管理の方法がかわると思うので、その辺りを中心にブレークダウンする予定である。

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はについては、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**

5.1.2 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 Tomcat7 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)	-	Tomcat7 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 DataSouce 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.dbcp.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.dbcp.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（JPA 編）、データベースアクセス（Mybatis2 編）を参照されたい。

JDBC の Debug 用ログの設定

O/R Mapper(Hibernate, MyBatis) 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: 本設定は Debug 用の設定なので、性能試験および商用環境にリリースする資材には、設定しないようにすること。**

log4jdbc 提供のデータソースの設定

- xxx-env.xml

```
<jee:jndi-lookup id="dataSourceSpied" jndi-name="jdbc/SampleDataSource" /> <!-- (1) -->

<bean id="dataSource" class="net.sf.log4jdbc.Log4jdbcProxyDataSource"> <!-- (2) -->
    <constructor-arg ref="dataSourceSpied" /> <!-- (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、"dataSourceSpied"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 を参照されたい。

5.1.3 How to extend

複数データソースを使用するための設定

課題

TBD

今後、以下の内容を追加する予定である。

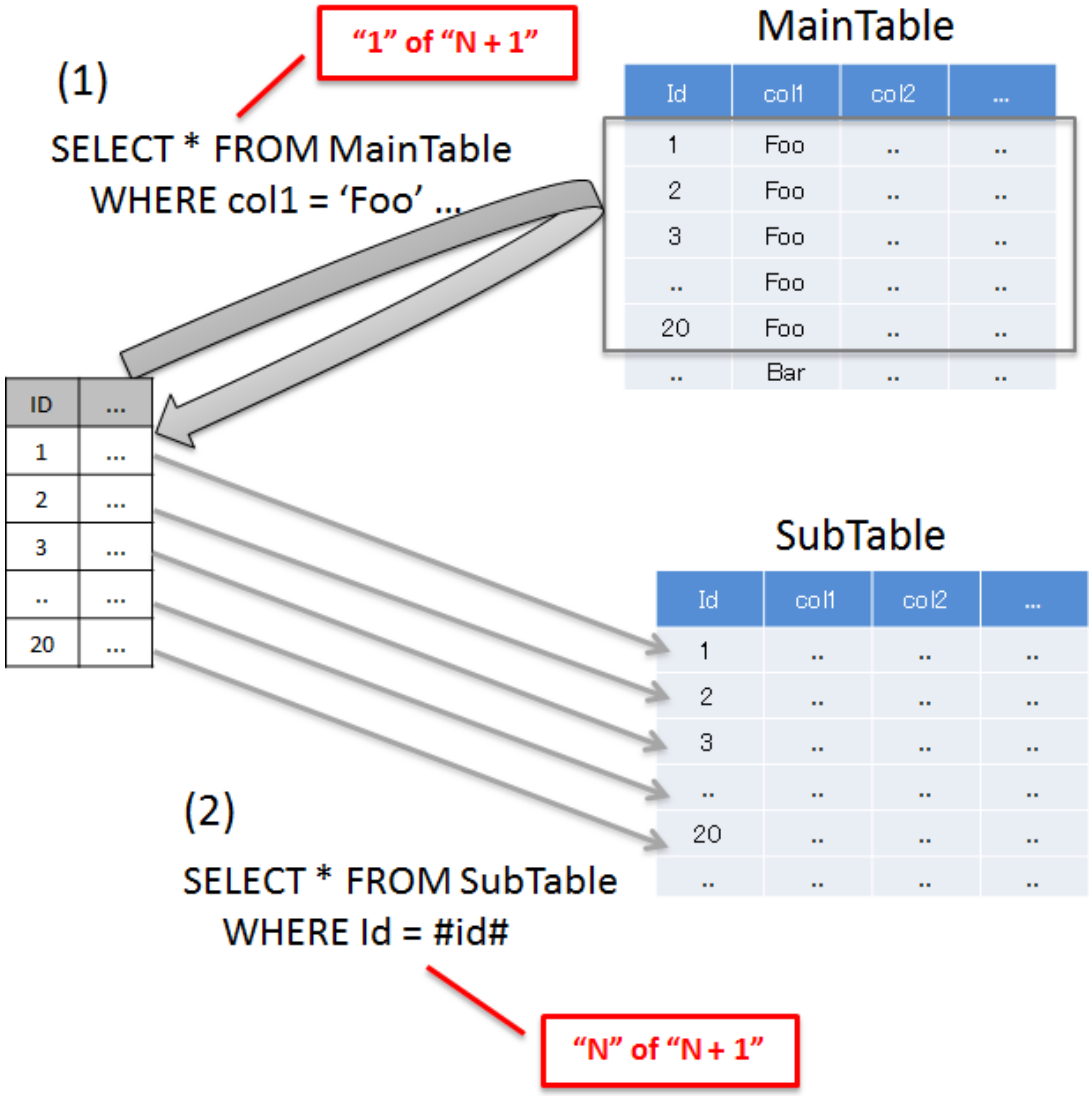
- 複数データソースを使う際の注意点などを踏まえた設定例。
- 実装にも影響がある場合は、実装例。

5.1.4 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。**

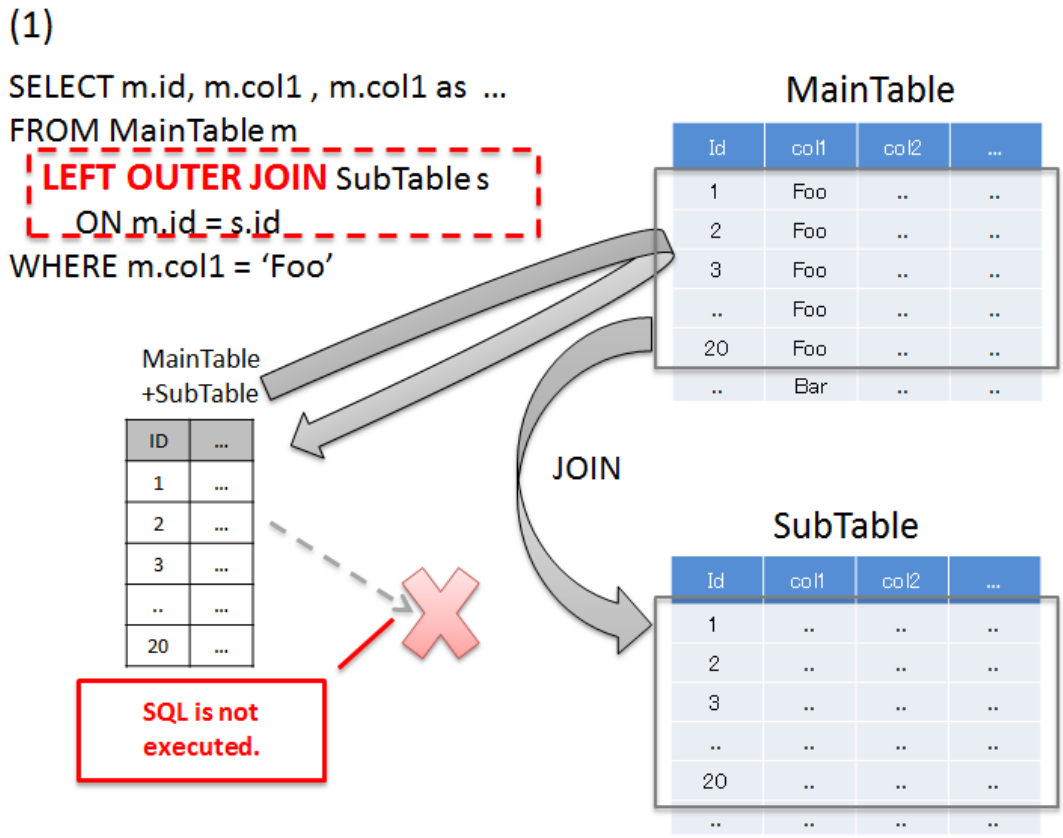
N+1 問題の解決方法の代表例を、以下に示す。

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。

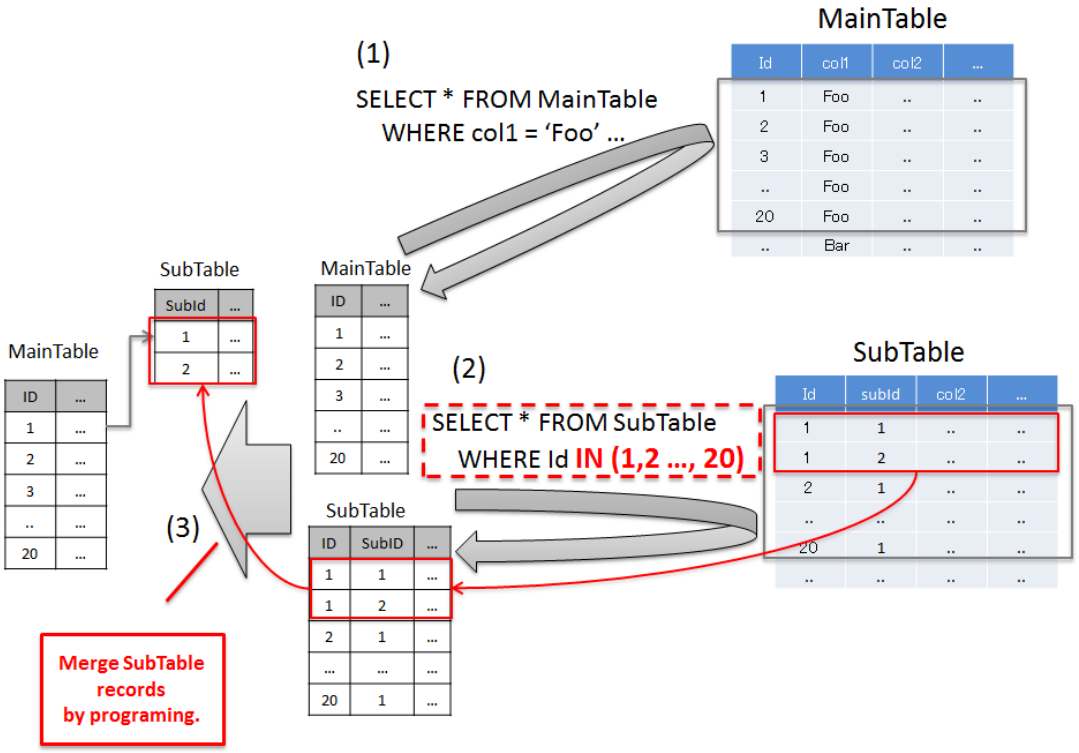
警告: 関連テーブルとの関連が、1:N の場合は、JOIN(Join Fetch) による解決も可能だが、以下の点に注意すること。

- 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。

5.1.5 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。

- 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。
- Mybatis2(TERASOLUNA DAO) を使用する場合は、データベースアクセス（Mybatis2 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、データベースにストアード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` のジェネリックス型には、`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。

TABLE 5.5 **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.orm.hibernate3.SessionFactoryUtils	JPA(Hibernate の実装) を使った場合、本クラスによって、O/R Mapper 例外が Spring Framework のデータアクセス例外に変換される。
2.	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。
3.	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。

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。

TABLE 5.6 **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。

TABLE 5.7 **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。

5.2 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。

5.2.1 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。

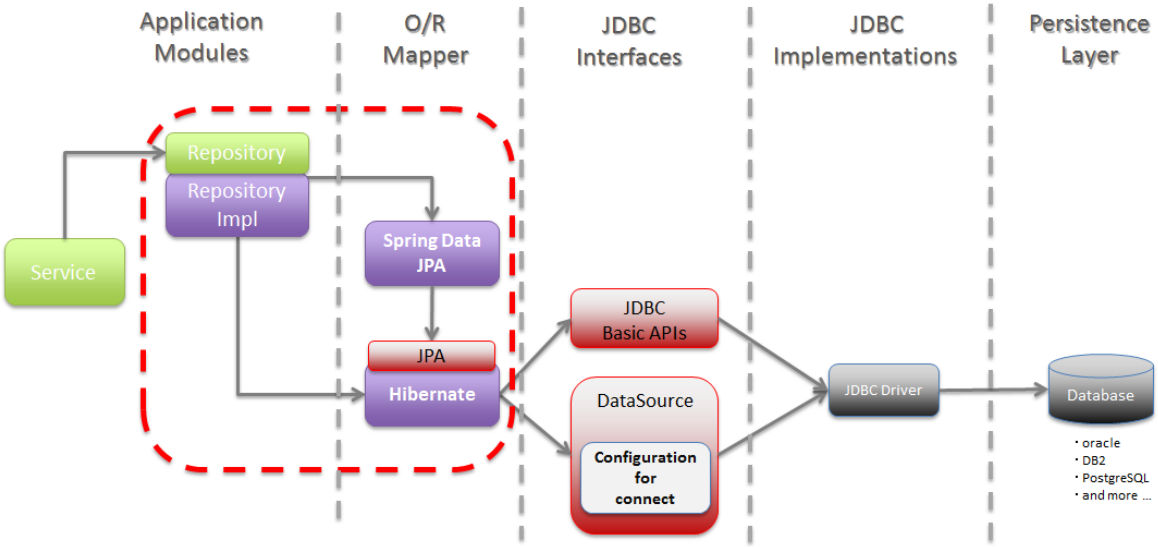


図 5.2 **Picture - Target of description**

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、仕様を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。

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。

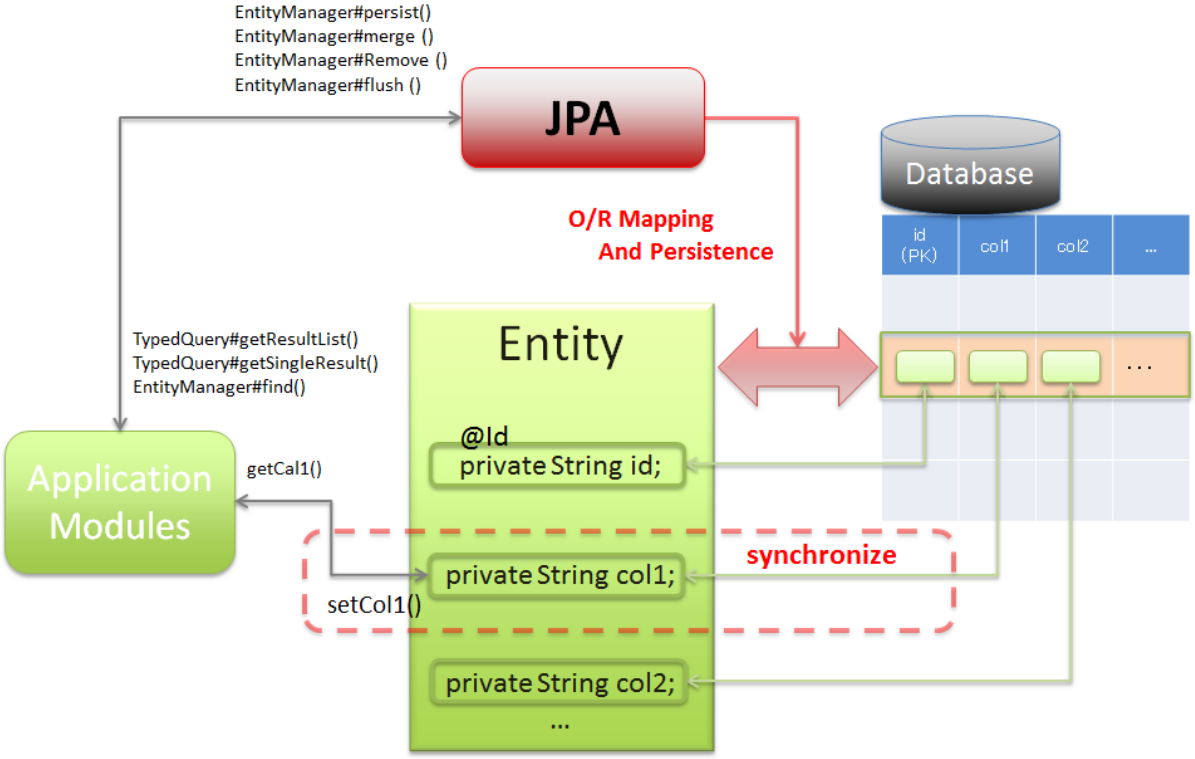


図 5.3 **Picture - Image of O/R Mapping**

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。

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。

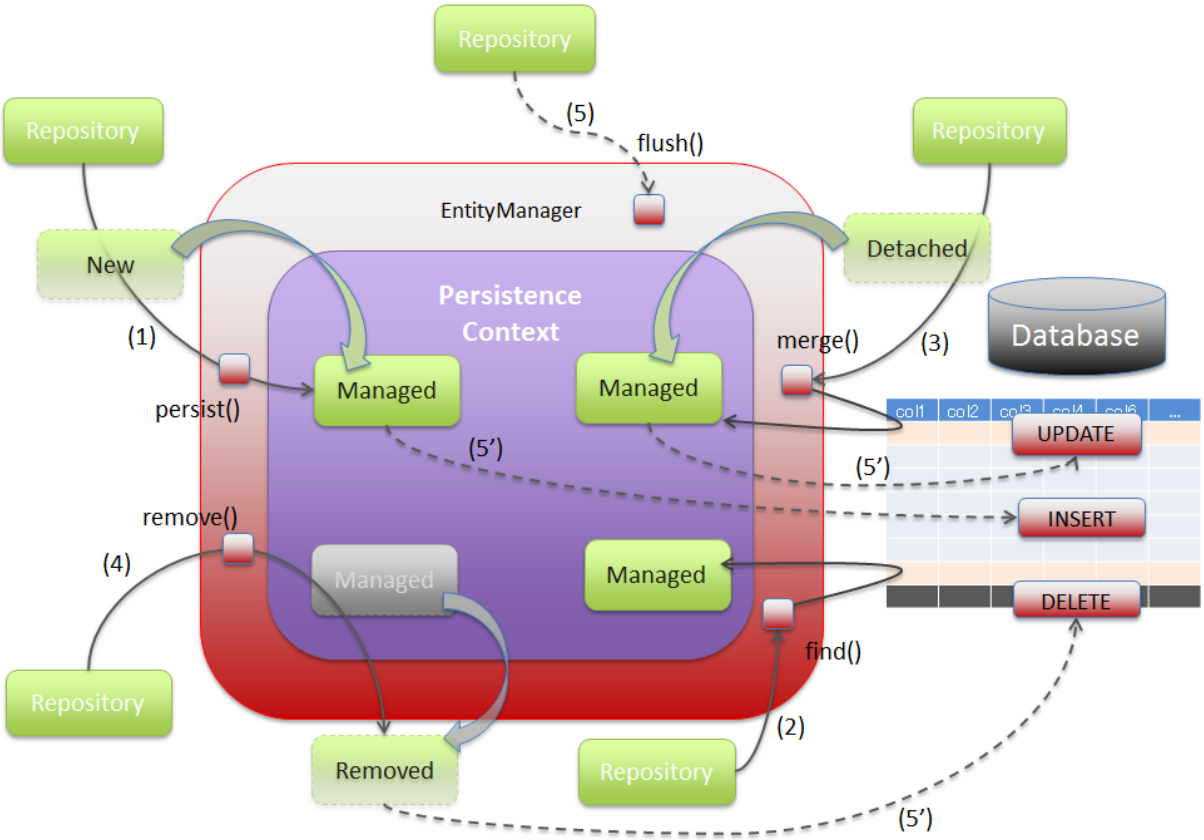


図 5.4 **Picture - Life cycle of entity**

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。
(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。

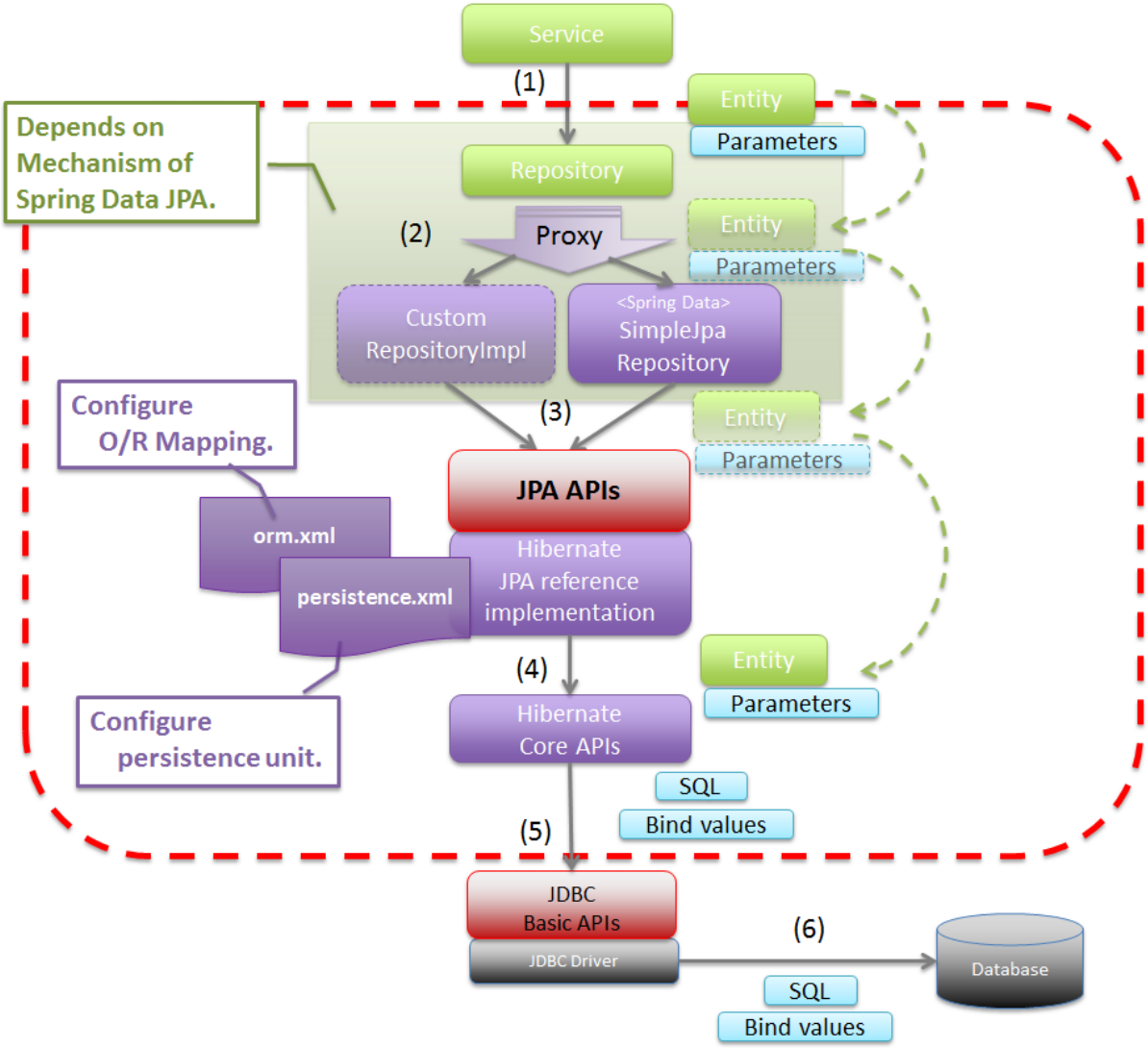


図 5.5　**Picture - Basic flow of Spring Data JPA**

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。
(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。

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。

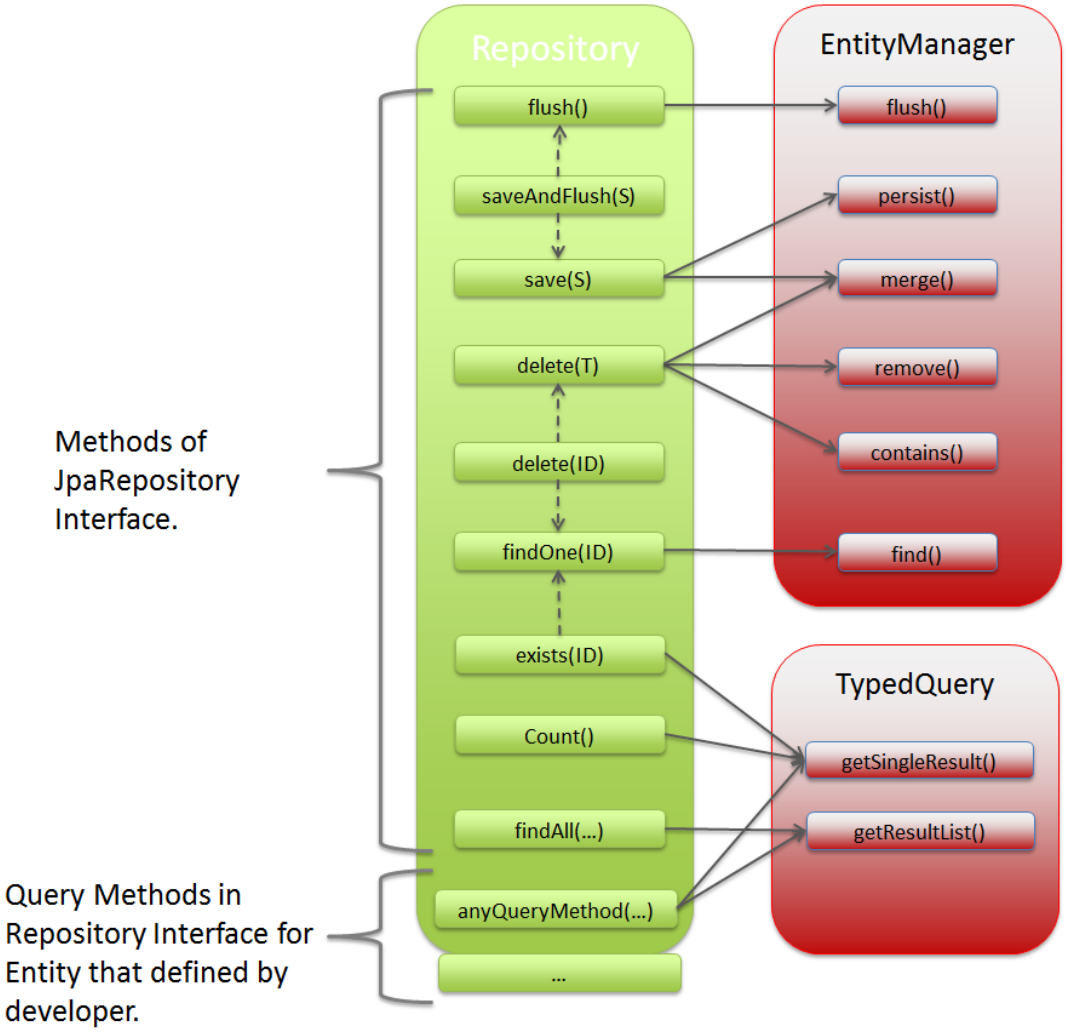


図 5.6　**Picture - API Mapping of Spring Data JPA and JPA**

5.2.2 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</artifactId>
</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` を、dependency に追加する。

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="none" />
                <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。
(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。

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.Oracle10gDialect" />  <!-- (9) -->
                </util:map>
            </property>
        </bean>
```

項番	説明
(9)	`"hibernate.dialect"`に `org.hibernate.dialect.Oracle10gDialect` を指定する。 `Oracle10gDialect` 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
    http://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 の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 JPA - 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.JpaRepositor` 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。
6.	repository-impl-postfix	カスタム Repository の実装クラスを示す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。

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。

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。

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。

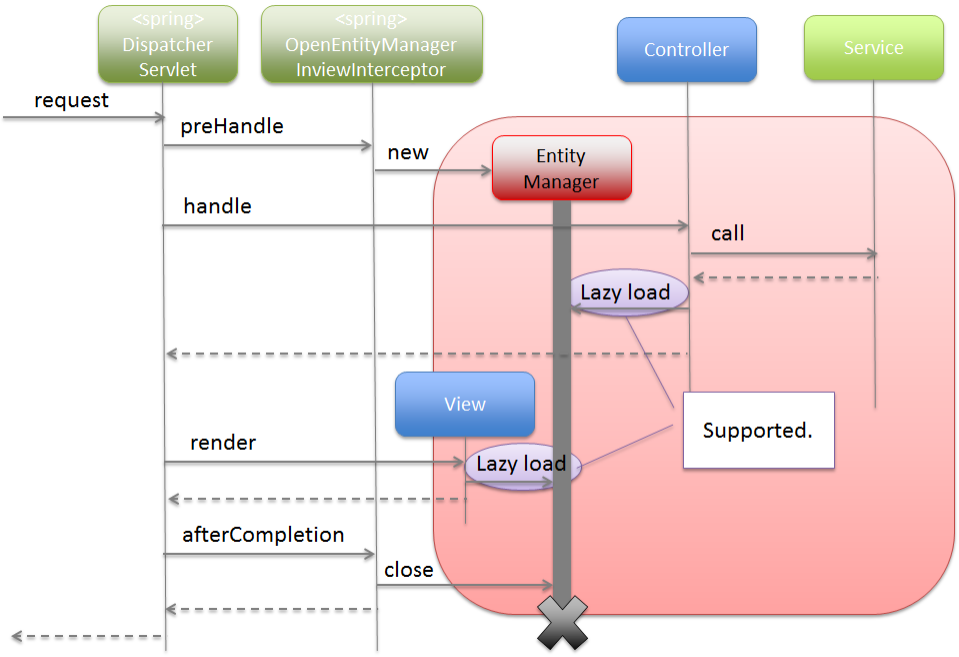


図 5.7 **Picture - Lifetime of EntityManager on OpenEntityManagerInViewInterceptor**

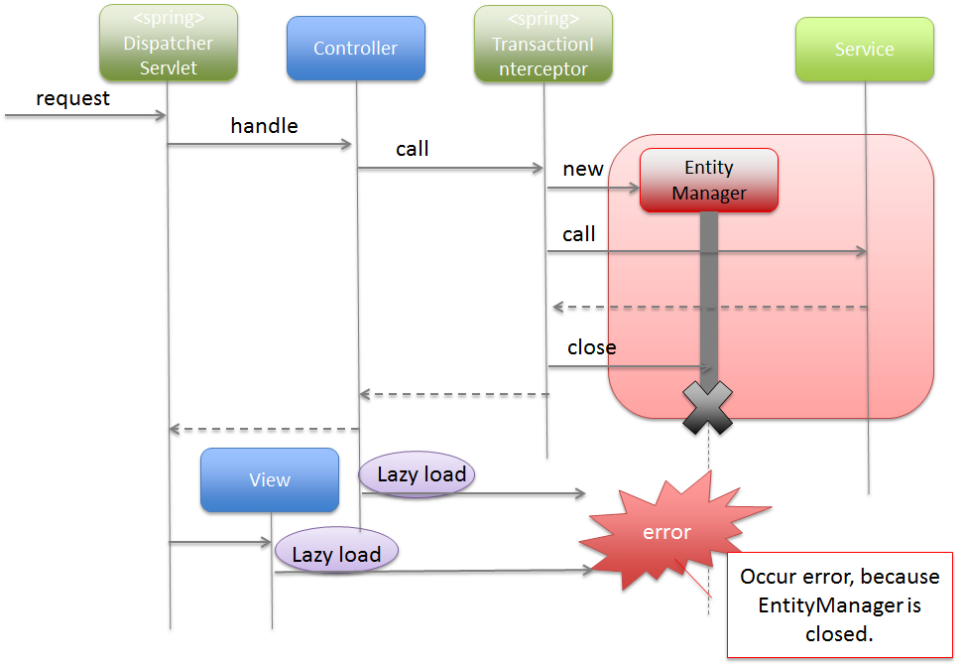


図 5.8 **Picture - Default Life time of EntityManager**

- spring-vmc.xml

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**" /> <!-- (1) -->
        <mvc:exclude-mapping path="/resources/**" /> <!-- (1) -->
        <mvc:exclude-mapping path="/**/*.html" /> <!-- (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) のパスと、静的 Web ページ (HTML) 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 層までの伸ばす必要がある。

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。

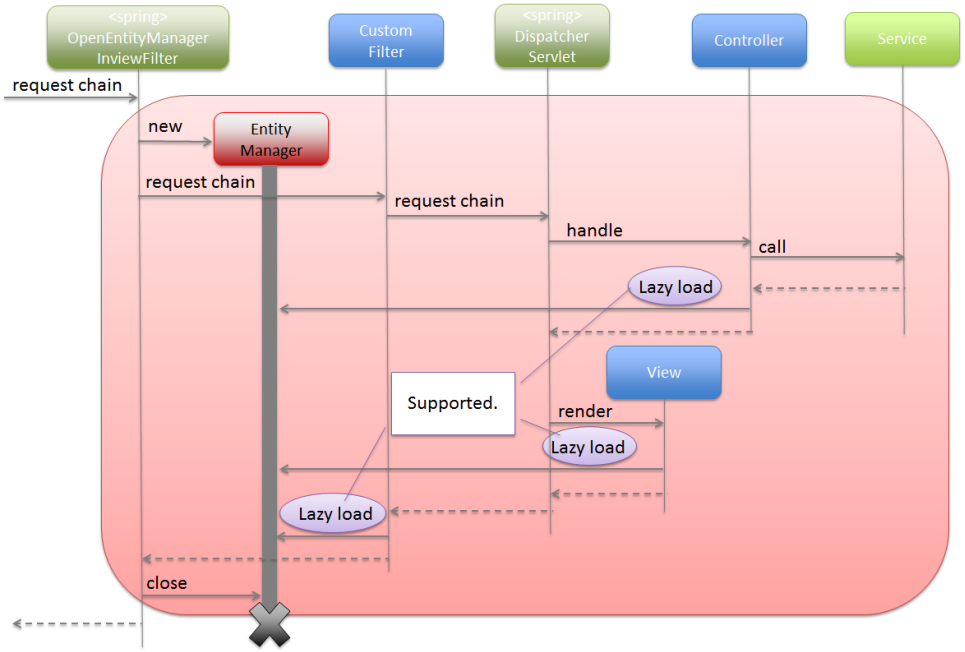


図 5.9 **Picture - Lifetime of EntityManager on OpenEntityManagerInViewFilter**

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-
</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。

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。
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.EmptyResultDataAccessE 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。

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)`

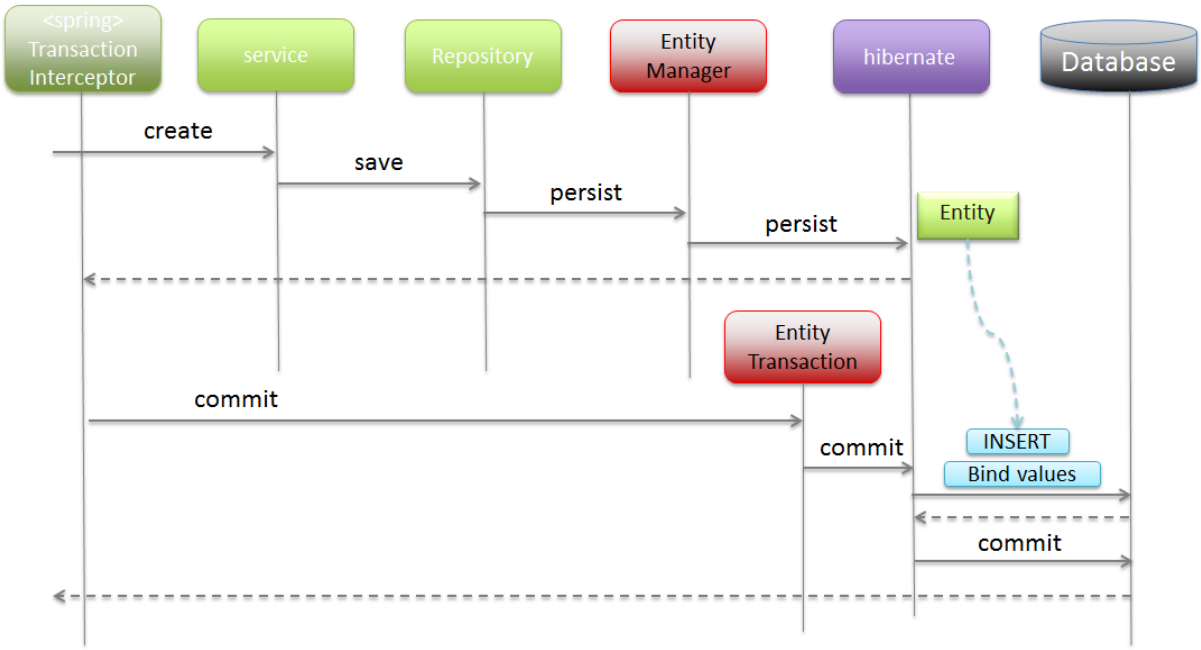


図 5.10 **Picture - Normal sequence of persistence processing**

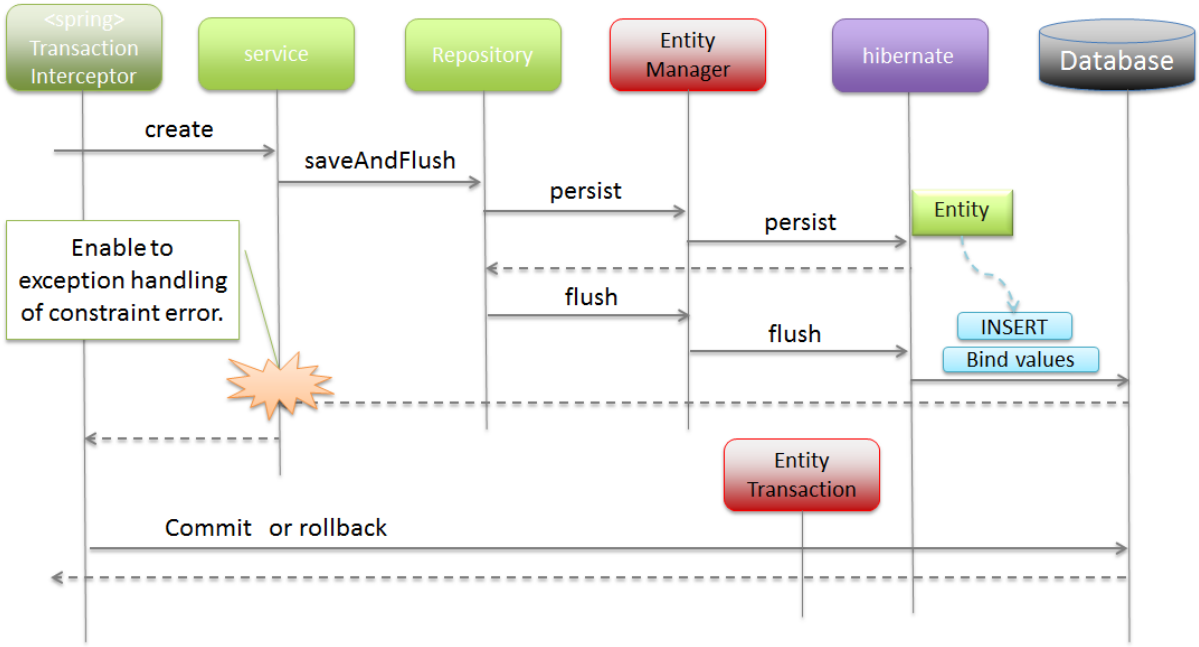


図 5.11 **Picture - Sequence of persistence processing when using flush method**

- `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

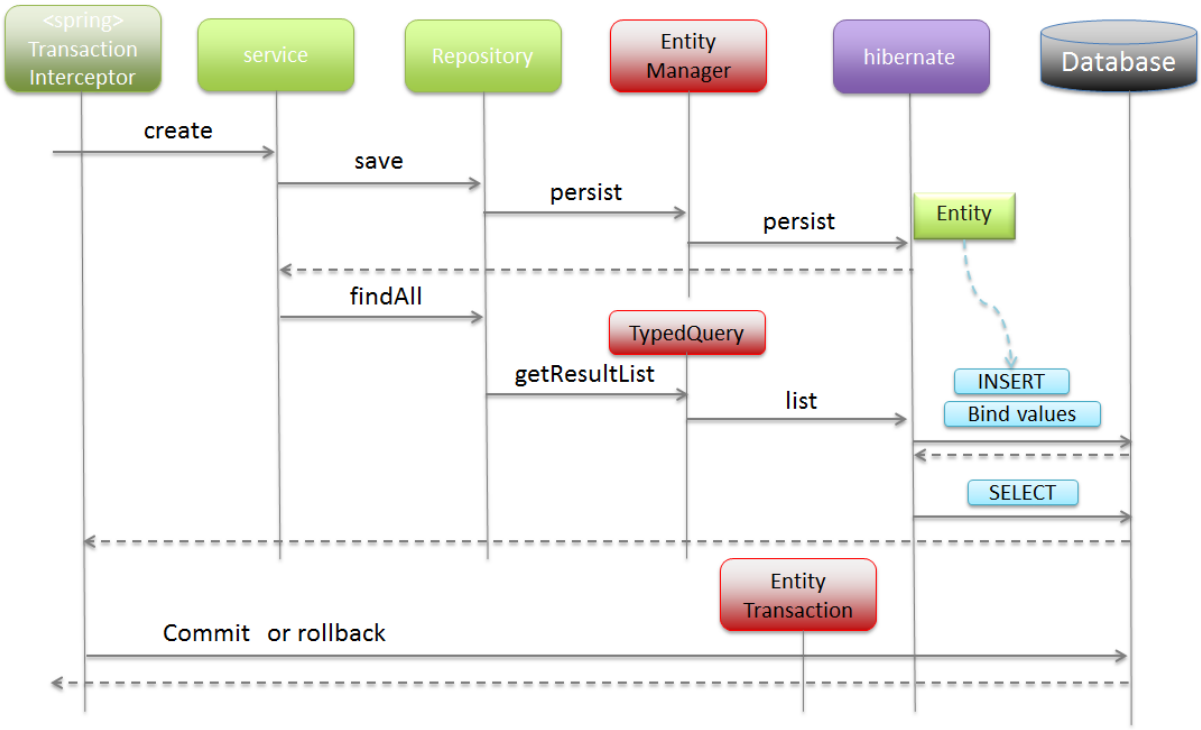


図 5.12 **Picture - Sequence of persistence processing when using query method**

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。

```
public interface MyProjectRepository<T, ID extends Serializable> extends
        Repository<T, ID> { // (1)

    T findOne(ID id); // (2)

    T save(T entity); // (2)

    // ...

}
```

```
public interface OrderRepository extends MyProjectRepository<Order, Integer> { // (3)

}
```

項番	説明
(1)	`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。
(2)	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。
(3)	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 317: Java Persistence API, Version 2.0 の Specification(PDF)「Chapter 4 Query Language」(P.131-178) 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 JPA - Reference Documentation「1.2.2 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 ") //
```

```
    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") }) //
    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 PostgreSQL)。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 317: Java Persistence API, Version 2.0 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 User guide の「3.4.1.8. 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")
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、 Qyery 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。

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 JPA - Reference Documentation「1.2.2 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 JPA - Reference Documentation「1.2.2 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 JPA - Reference Documentation「1.2.2 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「2.2.2. 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 JPA - Reference Documentation「Appendix B. 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.createdDat
    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) {
    DateMidnight targetDateMidnight = new DateMidnight(targetDate);
    Date fromDate = targetDateMidnight.toDate();
    Date toDate = targetDateMidnight.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.createdDat
    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,
                   @PageableDefaults(
                       pageNumber = 0,
                       value = 5,
                       sort = { "createdDate" },
                       sortDir = 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) {

    DateMidnight targetDateMidnight = new DateMidnight(targetDate);
    Date fromDate = targetDateMidnight.toDate();
    Date toDate = targetDateMidnight.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", SqlUtils
                    .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。
(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。

- 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", SqlUtils.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。
(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。

- 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 317: Java Persistence API, Version 2.0 の Specification(PDF)「OrderBy Annotation」(P.404-406)

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 317: Java Persistence API, Version 2.0 の Specification(PDF) を参照されたい。

- `@GeneratedValue` : P.375-376
- `@SequenceGenerator` : P.413-414
- `@TableGenerator` : P.415-417

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との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 =
    @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。

- 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。

5.2.3 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;

        // (7)
        public MyProjectRepositoryImpl(
                JpaEntityInformation<T, ID> entityInformation,
                EntityManager entityManager) {
            super(entityInformation, entityManager);

            this.entityInformation = entityInformation; // (8)
            this.entityManager = entityManager; // (8)

            try {
                return domainClass.getMethod("getVersion");
            } catch (NoSuchMethodException | SecurityException e) {
                return null;
            }
        }
```

```
        // (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。

- 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

```
    // (10)
    private static class MyProjectRepositoryFactory<T, ID extends Serializable>
            extends JpaRepositoryFactory {

        // (11)
        public MyProjectRepositoryFactory(EntityManager entityManager) {
            super(entityManager);
        }

        // (12)
        protected JpaRepository<T, ID> getTargetRepository(
                RepositoryMetadata metadata, EntityManager entityManager) {

            @SuppressWarnings("unchecked")
            JpaEntityInformation<T, ID> entityInformation = getEntityInformation((Class<T>) meta
                    .getDomainType());
```

```
            MyProjectRepositoryImpl<T, ID> repositoryImpl = new MyProjectRepositoryImpl<T, ID>(
                    entityInformation, entityManager);
            repositoryImpl
                    .setLockMetadataProvider(LockModeRepositoryPostProcessor.INSTANCE
                            .getLockMetadataProvider()); // (13)

            return repositoryImpl;
        }

        // (14)
        protected Class<?> getRepositoryBaseClass(RepositoryMetadata metadata) {
            return MyProjectRepository.class;
        }
    }
```

項番	説明
(10)	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 クラスを作成する。 例では、Factory クラスの実装を最小限にするために、`org.springframework.data.jpa.repository.support.JpaRepositoryFactory` を継承している。
(11)	`EntityManager` を引数とするコンストラクタを定義し、親クラスのコンストラクタを呼び出す。
(12)	実装クラスのインスタンスを生成するメソッドをオーバライドし、作成した実装クラス (例だと `MyProjectRepositoryImpl`) のインスタンスを生成し返却する。
(13)	Spring Data JPA より提供されている `SimpleJpaRepository` クラスの `findOne` メソッドおよび `findAll` 系メソッドに対して、 `@Lock` アノテーションを使ってロックモードを指定できるようにするために必要なコードとなる。 オーバーライド元の `JpaRepositoryFactory` のメソッドで実装されている処理なので、必ず実装すること。 拡張するメソッドの中で `findOne` メソッドや `findAll` メソッドを使う際に、ロックモードを指定したい場合に必要になる。
(14)	Entity 毎の Repository インタフェースの基底インタフェースを返却するメソッドをオーバライドし、作成したインタフェース (例だと `MyProjectRepository`) の型を返却する。

- Factory クラスのインスタンスを生成するための FactoryBean

```
    // (15)
    public class MyProjectRepositoryFactoryBean<R extends JpaRepository<T, ID>, T, ID extends Se
            extends JpaRepositoryFactoryBean<R, T, ID> {

        // (16)
        protected RepositoryFactorySupport createRepositoryFactory(
                EntityManager entityManager) {
            return new MyProjectRepositoryFactory<T, ID>(entityManager);
        }
```

```
    }
```

項番	説明
(15)	Factory クラスのインスタンスを作成するための FactoryBean クラスを作成する。 例では、FactoryBean クラスの実装を最小限にするために、`org.springframework.data.jpa.repository.support.JpaRepositoryFactory` を継承している。
(16)	Factory クラスのインスタンスを作成するためのメソッドをオーバーライドし、作成した Factory クラスのインスタンスを生成し返却する。

- Entity 毎の Repository インタフェース

```
    public interface OrderRepository extends MyProjectRepository<Order, Integer> { // (17)
        // ...
    }
```

項番	説明
(17)	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"
        factory-class="x.y.z.domain.repository.MyProjectRepositoryFactoryBean" /> <!-- (18) -->
```

項番	説明
(18)	<jpa:repositories>要素の factory-class 属性に、作成した FactoryBean クラス (例だと `MyProjectRepositoryFactoryBean`) 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); //

        // ....

        return order;
```

```
    }
```

項番	説明
(19)	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、を 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.lang.Long` 、 `long` 型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.AuditingEntityListene
                </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 `getDateTime()` メソッドから返却される `org.joda.time.DateTime` 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
        DateFactory dateFactory;

        // (3)
        public DateTime getDateTime() {
            return dateFactory.newDateTime();
        }

    }
```

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.DateFactory` 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。 `DateFactory` 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で有効である。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: 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。

5.3 データベースアクセス（Mybatis2 編）

5.3.1 Overview

本節では、MyBatis2.x 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、TERASOLUNA DAO(Mybatis のラッパー DAO) を使用することを前提としている。

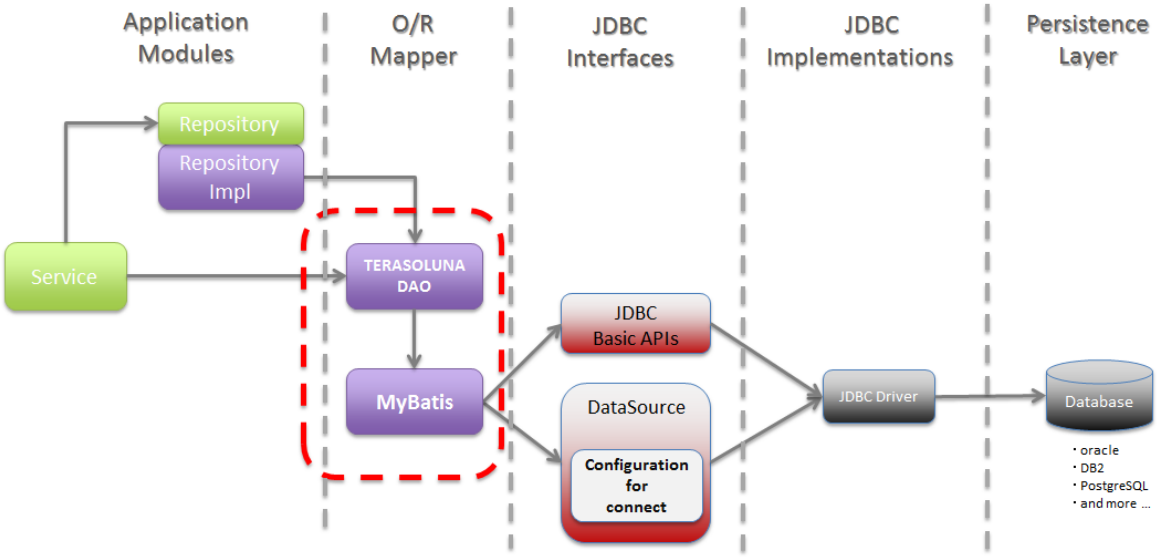


図 5.13　**Picture - Target of description**

Mybatis について

Mybatis 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。

Mybatis2.x の詳細については、Mybatis Developer Guide(PDF) を参照されたい。

TERASOLUNA DAO について

TERASOLUNA DAO は、O/R Mapper に依存する処理を隠蔽するための DAO インタフェースと、Mybatis2.x を使用した DAO 実装クラスを提供している。

TERASOLUNA DAO から提供されている DAO インタフェースは、以下の通りである。

TABLE 5.8 **TERASOLUNA DAO から提供されている DAO インタフェース**

項番	クラス名	説明
1.	jp.terasoluna.fw.dao.QueryDAO	参照系 SQL を実行するための DAO インタフェース
2.	jp.terasoluna.fw.dao.UpdateDAO	更新系 SQL を実行するための DAO インタフェース
3.	jp.terasoluna.fw.dao.StoredProcedureDAO	StoredProcedure を実行するための DAO インタフェース
4.	jp.terasoluna.fw.dao.QueryRowHandleDAO	参照系 SQL を実行して取得されるレコードに対して、一レコードずつ処理を行うための DAO インタフェース。

TERASOLUNA DAO(Mybatis 実装) 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。

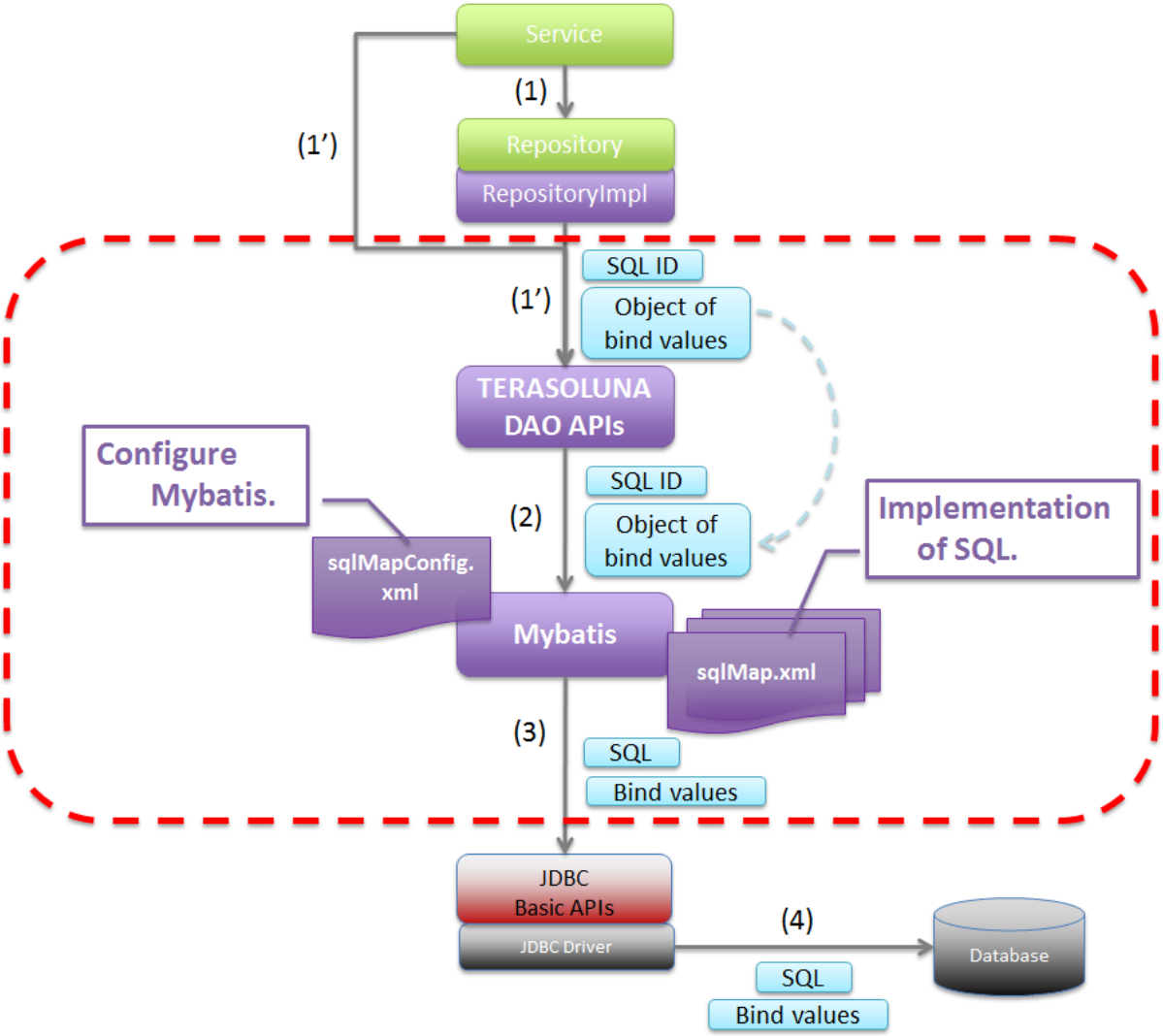


図 5.14　**Picture - Basic flow of TERASOLUNA DAO**

項番	説明
(1)	Service または Repository から、TERASOLUNA DAO から提供されている DAO 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、SQLID と SQL に埋め込む値を保持しているオブジェクトを渡す。
(2)	TERASOLUNA DAO は、Mybatis の API に、処理を委譲する。 Service または Repository から指定された SQLID と、SQL に埋め込む値を保持しているオブジェクトも Mybatis に渡される。
(3)	Mybatis は、指定された SQLID に対応する SQL を、設定ファイル (`sqlMap.xml`) から取得し、SQL とバインド値を、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)
	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

5.3.2 How to use

pom.xml の設定

インフラストラクチャ層に MyBatis2(TERASOLUNA DAO) 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-mybatis2</artifactId>
    </dependency>
```

項番	説明
(1)	MyBatis2 に関連するライブラリ群が定義してある `terasoluna-gfw-mybatis2` を dependency に追加する。

アプリケーションの設定

データソースの設定

データソースに設定は、共通編の**データソースの設定**を参照されたい。

PlatformTransactionManager の設定

ローカルトランザクションを使用する場合は、以下の通りに設定する。

> ローカルトランザクションを使用する場合、JDBC の API を呼び出してトランザクション制御を行う `org.springframework.jdbc.datasource.DataSourceTransactionManager` を使用する。

- xxx-env.xml

```
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"> <!-- (1) -->
        <property name="dataSource" ref="dataSource" /> <!-- (2) -->
    </bean>
```

項番	説明
(1)	`org.springframework.jdbc.datasource.DataSourceTransactionManager` を指定する。
(2)	設定済みのデータソースの 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を使用する場合は、以下の通りに設定する。

> 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を使用する場合、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。

- 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 `JtaTransactionManager` が、`"transactionManager"`という id で、bean 定義される。

TERASOLUNA DAO の設定

Spring Framework から提供されている `SqlMapClient` のファクトリクラスと、TERASOLUNA DAO の bean 定義を行う。

- xxx-infra.xml

```
<bean id="sqlMapClient"
    class="org.springframework.orm.ibatis.SqlMapClientFactoryBean"> <!-- (1) -->
    <property name="configLocations"
        value="classpath*:/META-INF/mybatis/config/*sqlMapConfig.xml" /> <!-- (2) -->
    <property name="mappingLocations"
        value="classpath*:/META-INF/mybatis/sql/**/*-sqlmap.xml" /> <!-- (3) -->
    <property name="dataSource" ref="dataSource" /> <!-- (4) -->
</bean>

<bean id="queryDAO"
    class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl"> <!-- (5) -->
    <property name="sqlMapClient" ref="sqlMapClient" /> <!-- (6) -->
</bean>
```

```
    <!-- (5) (6) -->
    <bean id="updateDAO"
        class="jp.terasoluna.fw.dao.ibatis.UpdateDAOiBatisImpl">
        <property name="sqlMapClient" ref="sqlMapClient" />
    </bean>

    <!-- (5) (6) -->
    <bean id="spDAO"
        class="jp.terasoluna.fw.dao.ibatis.StoredProcedureDAOiBatisImpl">
        <property name="sqlMapClient" ref="sqlMapClient" />
    </bean>

    <!-- (5) (6) -->
    <bean id="queryRowHandleDAO"
        class="jp.terasoluna.fw.dao.ibatis.QueryRowHandleDAOiBatisImpl">
        <property name="sqlMapClient" ref="sqlMapClient" />
    </bean>
```

項番	説明
(1)	SqlMapClient クラスのファクトリクラスとして、org.springframework.orm.ibatis.SqlMapClientFactoryBean を指定する。
(2)	Mybatis の設定ファイルのロケーションを指定する。 例では、クラスパス上の「/META-INF/mybatis/config/」ディレクトリに格納されている「sqlMapConfig.xml」で終わるファイルが、対象ファイルとなる。 設定ファイルについては、*Mybatis* **の設定**を参照されたい。
(3)	Mybatis の SQL マッピングファイルのロケーションを指定する。 例では、クラスパス上の「/META-INF/mybatis/sql/」ディレクトリ配下（サブディレクトリも含む）に格納されている「-sqlmap.xml」で終わるファイルが、対象ファイルとなる。 SQL マッピングファイルについては、*SQL* **マッピングの実装** *(基本編)* を参照されたい。
(4)	設定済みのデータソースの bean を指定する。
(5)	TERASOLUNA DAO の Mybatis 実装クラスを指定して、bean 定義する。
(6)	(1) で定義した SqlMapClient クラスのファクトリクラスの bean を指定する。

LOB 型を扱う場合の設定

BLOB や CLOB などの Large Object を扱う場合は、SqlMapClient クラスのファクトリクラスに、LobHandler を指定する。

- xxx-infra.xml

```
    <!-- (1) -->
    <bean id="nativeJdbcExtractor"
        class="org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor" />

    <!-- (2) -->
    <bean id="lobHandler" class="org.springframework.jdbc.support.lob.OracleLobHandler">
```

```
        <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor" /> <!-- (3) -->
    </bean>

    <bean id="sqlMapClient"
        class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
        <property name="configLocations"
            value="classpath*:/META-INF/mybatis/config/*sqlMapConfig.xml" />
        <property name="mappingLocations"
            value="classpath*:/META-INF/mybatis/sql/**/*-sqlmap.xml" />
        <property name="dataSource" ref="dataSource" />
       <property name="lobHandler" ref="lobHandler" /> <!-- (4) -->
    </bean>
```

項番	説明

(1)	`org.springframework.jdbc.support.nativejdbc.NativeJdbcExtractor` インタフェースの実装クラスを、bean 定義する。
	例では、
	`org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtracto`
	を指定しているが、
	Tomcat 以外の AP サーバでは、コネクションプールの実装によって、JDBC コネクションを取得できない場合がある。
	その場合、Spring が提供している他の
	`NativeJdbcExtractor(org.springframework.jdbc.support.nativejdbc.WebLogicN`
	など) を指定するか、各 AP サーバ用に、新たに `NativeJdbcExtractor` を作成する必要がある。

(2)	`org.springframework.jdbc.support.lob.LobHandler` インタフェースの実装クラスを bean 定義する。 例では、Oracle 使用時に指定する `org.springframework.jdbc.support.lob.OracleLobHandler` を指定しているが、 Oracle 以外の場合は、 `org.springframework.jdbc.support.lob.DefaultLobHandler` を指定する。
(3)	(1) で定義した `NativeJdbcExtractor` の bean を指定する。
(4)	(3) で定義した `LobHandler` の bean を指定する。

Mybatis の設定

`SqlMapClient` のデフォルトの動作をカスタマイズする。必要に応じてカスタマイズすること。

- sqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
            PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
            "http://ibatis.apache.org/dtd/sql-map-config-2.dtd"> <!-- (1) -->
```

```
    <sqlMapConfig>
        <settings useStatementNamespaces="true" /> <!-- (2) -->
    </sqlMapConfig>
```

項番	説明
(1)	DTD ファイルを指定する。指定することで、スキーマのチェックと、IDE 上でのコード補完が有効となる。
(2)	useStatementNamespaces="true"を設定することで、SQL マッピングファイルで指定するネームスペースを、SQLID として使用するように設定している。

- sqlMapConfig の子要素について

  子要素として、properties, settings,resultObjectFactory, typeAlias, transactionManager, sqlMap が存在する。

  必要に応じて、設定を行うこと。

  詳細は、Mybatis Developer Guide(PDF) の「The SQL Map XML Configuration File」(P.8-16) を参照されたい。

TABLE 5.9 **sqlMapConfig の子要素**

項番	要素	説明
1.	properties	プロパティファイルを読み込むための要素。読み込んだプロパティファイルに定義されているプロパティは、 Mybatis の設定ファイルおよび SQL マッピングファイル内から、"${**プロパティ名**}"の形式で参照することができる。 環境に依存する値や、共通的な設定値を定義する際に使用する。 詳細は、Mybatis Developer Guide(PDF) の「The SQL Map XML Configuration File」(P.9) を参照されたい。
2.	settings	`SqlMapClient` のデフォルト動作のカスタマイズを行うための要素。 設定項目の詳細については、Mybatis Developer Guide(PDF) の「The SQL Map XML Configuration File」(P.9-11) を参照されたい。
3.	resultObjectFactory	SQL マッピングファイルの select 要素、statement 要素、procedure 要素の resultClass 属性、または、resultMap 要素の class 属性に指定されたクラスのインスタンスを生成するファクトリクラスを指定する要素。 指定しない場合は、デフォルト実装である `java.lang.Class#newInstance()` メソッドで生成されたインスタンスが使用される。 詳細については、Mybatis Developer Guide(PDF) の「The SQL Map XML Configuration File」(P.11-12) を参照されたい。
4.	typeAlias	クラス名 (FQCN) に別名（短縮名）を付けるための要素。 ここで定義した別名は、Mybatis の設定ファイルおよび SQL マッピングファイルのクラスを指定する箇所で使うことができる。通常、パッケージを除いたシンプルなクラス名を指定することが多い。 詳細については、Mybatis Developer Guide(PDF) の「The SQL Map XML Configuration File」(P.12) を参照されたい。
5.	transactionManager	トランザクション管理は、Spring Framework の機能を使うため、定義は不要である。
6.	sqlMap	TERASOLUNA DAO の設定で設定済みのため、定義は不要である。

SQL マッピングの実装 (基本編)

以下に、基本的な SQL マッピングの実装例を示す。

アプリケーション内で使用する SQL を実装する。

- xxx-sqlmap.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
            PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
            "http://ibatis.apache.org/dtd/sql-map-2.dtd"> <!-- (1) -->

<sqlMap namespace="xxx"> <!-- (2) -->

    <!-- (3) -->
    <select id="findOne">
        <!-- ... -->
    </select>

    <!-- ... -->

</sqlMap>
```

項番	説明
(1)	DTD ファイルを指定する。指定することで、スキーマのチェックと、STS 上でのコード補完が有効となる。
(2)	ネームスペースを指定する。
(3)	`sqlMapConfig.xml` にて、ネームスペースを SQLID として使用するように設定しているので、この SQL を実行するために指定する SQLID は「`xxx.findOne`」となる。

- sqlMap の子要素について

  子要素として、`cacheModel`, `typeAlias`, `parameterMap`, `resultMap`, `select`, `insert`, `update`, `delete`, `statement`, `sql`, `procedure` が存在する。

TABLE 5.10 **sqlMap の子要素**

項番	要素	説明
1.	typeAlias	`sqlMapConfig.xml` の `typeAlias` と同じ。
2.	cacheModel	オブジェクトのキャッシュの定義を行う要素
3.	parameterMap	SQL にバインドするパラメータ（オブジェクト）のマッピングに関する定義を行う要素
4.	resultMap	SQL の実行結果として返却されるレコードとオブジェクトのマッピングに関する定義を行う要素
5.	select	SELECT 文を記載する要素
6.	insert	INSERT 文を記載する要素
7.	update	UPDATE 文を記載する要素
8.	delete	DELETE 文を記載する要素
9.	statement	`select`, `insert`, `update`, `delete`, `procedure` 要素を包含している汎用要素。個別の要素 (`select`, `insert`, `update`, `delete`, `procedure`) を使用することを推奨する。
10.	sql	`select`, `insert`, `update`, `delete`, `statement` からインクルードするための SQL 文（SQL 文の一部）を記載する要素。この要素をうまく使うことで、複数の SQL で重複している部分を、共通化することができる。
11.	procedure	PROCEDURE 呼び出しを記載する要素

注釈: 詳細は、Mybatis Developer Guide(PDF) の、以下の章を参照されたい。

- The SQL Map XML File(P.17-18)

  SQL マッピングファイルの簡単な定義例が記載されている。

- Mapped Statements(P.18-26)

  SQL を組み立てるための要素の、基本的な使い方が記載されている。

- Parameter Maps and Inline Parameters(P.27-31)

  SQL にバインドするパラメータ（オブジェクト）のマッピングに関する、詳細な説明が記載されて

いる。

- Substitution Strings(P.32)

  SQL のバインド変数について、記載されている。

- Result Maps(P.32-41)

  SQL の実行結果として返却されるレコードと、オブジェクトのマッピングに関する、詳細な説明が記載されている。

- Supported Types for Parameter Maps and Result Maps(P.42-43)

  ParameterMap と、ResultMap でサポートされている型と、拡張方法が記載されている。

- Caching Mapped Statement Results(P.44-47)

  キャッシュに関する詳細な説明が、記載されている。

- Dynamic Mapped Statements(P.48-53)

  動的 SQL に関する詳細な説明が、記載されている。

- Simple Dynamic SQL Elements(P.53)

  動的 SQL の簡易的な実装方法の説明が、記載されている。

> **警告:** `statement`, `select`, `procedure` 要素を使用して、大量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しておくこと。fetchSize 属性は、JDBC ドライバとデータベース間の通信において、一度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 属性を省略した場合は、各 JDBC ドライバのデフォルト値が利用されるため、デフォルト値が全件取得する JDBC ドライバの場合、メモリの枯渇の原因になる可能性があるので、注意が必要となる。

select 要素の実装例

select 要素を実装する前に、検索したレコードのカラムと、JavaBean のプロパティのマッピング定義を行う。

- xxx-sqlmap.xml

```
<resultMap id="resultMap_Todo"
           class="xxxxxx.yyyyyy.zzzzzz.domain.model.Todo"> <!-- (1) -->
    <result property="todoId" column="todo_id" /> <!-- (2) -->
    <result property="todoTitle" column="todo_title" />
    <result property="finished" column="finished" />
    <result property="createdAt" column="created_at" />
    <result property="version" column="version" />
</resultMap>
```

項番	属性	説明
(1)	-	検索したレコードと JavaBean のマッピングを行う。詳細は、Developer Guide を参照されたい。
	id	マッピングを識別するための ID を指定する。select 属性から参照される。
	class	マッピングする JavaBean の FQCN を指定する。
(2)	-	JavaBean のプロパティと、検索したレコードのカラムのマッピングを行う。
	property	JavaBean のプロパティ名を指定する。
	column	property 属性で指定したプロパティに、マッピングするレコードのカラム名を指定する。

select 要素を実装する。

- xxx-sqlmap.xml

```
    <select id="findOne"
            parameterClass="java.lang.String"
            resultMap="resultMap_Todo"> <!-- (3) -->
        SELECT
            *
        FROM
            todo
        WHERE
            todo_id = #todoId#    /* (4) */
    </select>
```

項番	属性	説明
(3)	-	検索用 SQL を実装する。
	id	検索 SQL を識別するための ID を指定する。
	parameterClass	バインド用オブジェクトの型を指定する。 例では、`java.lang.String` を指定しているが、複数のパラメータ (検索条件) を渡したい場合は、JavaBean を指定することもできる。
	resultMap	(1) で定義した resultMap を指定する。 resultMap を使わずに、自動的に class 属性で指定した JavaBean のプロパティにマッピングすることもできるが、取得レコードのカラム名と、JavaBean のプロパティ名が一致している必要がある。 取得レコードのカラム名と JavaBean のプロパティ名を一致させる方法として、AS 句を使って、カラム名に別名を付与する方法がある。 例えば、SQL を"`SELECT todo_title AS todoTitle, ...`"とすると、JavaBean の todoTitle プロパティに、値が設定される。
(4)	-	SQL にバインド値を指定する。 例では、JavaBean ではなく単一オブジェクト（`java.lang.String`）を使用しているので、バインド変数名は任意の名前を指定することができる。 バインド用オブジェクトに JavaBean を使用する場合は、バインド用の変数名は、JavaBean のプロパティ名と一致させる必要がある。

注釈: 自動マッピングについて

resultMap 属性を使わずに、resultClass 属性で指定した JavaBean のプロパティに、自動的にマッピングすることもできるが、取得レコードのカラム名と、JavaBean のプロパティ名が一致している必要がある。取得レコードのカラム名と、JavaBean のプロパティ名を一致させる方法として、AS 句を使って、カラム名に別名を付与する方法がある。下記に、自動マッピングを使用した場合の実装例を示す。

```
        <select id="findOne"
                parameterClass="java.lang.String"
                resultClass="xxxxxx.yyyyyy.zzzzzz.domain.model.Todo">
```

```
            SELECT
                todo_id AS todoId,
                todo_title AS todoTitle,
                finished,
                created_at AS createdAt,
                version
            FROM
                todo
            WHERE
                todo_id = #todoId#
        </select>
```

自動マッピングは、取得したレコードと JavaBean をマッピングする手段としては、もっとも簡単な方法である。ただし、自動マッピングを使用した場合は、以下の制約や注意点があることを考慮し、使用すること。

- SQL で取得した値の型宣言や、変換定義などが行えない。
- 複雑なマッピング（例えば、ネストされている JavaBean へのマッピング）が行えない。
- マッピングする際に、`java.sql.ResultSetMetaData` にアクセスするため、若干のパフォーマンス劣化が発生する。

insert 要素の実装例

insert 要素を実装する。

- xxx-sqlmap.xml

```
    <insert id="insert"
            parameterClass="xxxxxx.yyyyyy.zzzzzz.domain.model.Todo"> <!-- (1) -->
        INSERT INTO todo
            (
                todo_id
                ,todo_title
                ,finished
                ,created_at
                ,version
            )
            values(
                #todoId#       /* (2) */
                ,#todoTitle#
                ,#finished#
                ,#createdAt#
                ,1
            )
    </insert>
```

項番	属性	説明
(1)	-	挿入用 SQL を実装する。
	id	挿入用 SQL を識別するための ID を指定する。
	parameterClass	バインド用オブジェクトの型を指定する。JavaBean を指定することもできる。
(2)	-	SQL にバインド値を指定する。バインド用オブジェクトに JavaBean を使用する場合は、バインド用の変数名は、JavaBean のプロパティ名と一致させる必要がある。

注釈: parameterMap 属性や、”Inline Parameter Maps” の仕組みを使用することで、SQL にバインドする値の型の宣言や、変換定義を行うことができる。例えば、バインド値が `null` の場合に、デフォルト値を設定することができる。詳細は、Mybatis Developer Guide(PDF) の「Parameter Maps and Inline Parameters」(P.27-31) を参照されたい。

update 要素の実装例

update 要素を実装する。

- xxx-sqlmap.xml

```
<update id="update"
        parameterClass="xxxxxx.yyyyyy.zzzzzz.domain.model.Todo"> <!-- (1) -->
    UPDATE todo SET
        todo_id = #todoId#
        ,todo_title = #todoTitle#
        ,finished = #finished#
        ,version = (#version# + 1)
    WHERE
        todo_id = #todoId#
    AND version = #version#
</update>
```

項番	説明
(1)	更新用 SQL を実装する。

delete 要素の実装例

delete 要素を実装する。

- xxx-sqlmap.xml

```
<delete id="delete" parameterClass="java.lang.String">  <!-- (1) -->
    DELETE FROM
        todo
    WHERE
        todo_id = #todoId#
</delete>
```

項番	説明
(1)	削除用 SQL を実装する。

procedure 要素の実装例

以下は、PostgreSQL に作成したファンクションを、procedure 要素を使って呼び出す例となっている。

テーブルと、ファンクション (PL/pgSQL 実装) を作成する SQL は、以下の通りである。

```
CREATE TABLE sales (
    itemno INT4 PRIMARY KEY,
    quantity INT4 NOT NULL,
    price INT4 NOT NULL
);
```

```
CREATE
    FUNCTION sales_item(p_itemno INT4) RETURNS TABLE (
        quantity INT4
        ,total INT4
    ) AS $$ BEGIN RETURN QUERY
        SELECT
                s.quantity
                ,s.quantity * s.price
            FROM
                sales s
            WHERE
                itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

parameterMap 要素を実装する。

```
<!-- (1) -->
<parameterMap id="salesItemMap" class="xxxxxx.yyyyyy.zzzzzz.domain.model.SalesItem">
```

```
        <!-- (2) -->
        <parameter property="id" jdbcType="INTEGER" mode="IN" />
        <!-- (3) -->
        <parameter property="quantity" jdbcType="INTEGER" mode="OUT" />
        <parameter property="total" jdbcType="INTEGER" mode="OUT" />
    </parameterMap>
```

```
    // (4)
    public class SalesItem implements Serializable {
        private Integer id;
        private Integer quantity;
        private Integer total;
        // ...
    }
```

項番	説明
(1)	ファンクションに渡す IN パラメータと、OUT パラメータのマッピングを定義する。
(2)	IN パラメータのマッピングを定義している。IN パラメータに、`SalesItem#id` をマッピングしている。
(3)	OUT パラメータのマッピングを定義している。OUT パラメータの 1 番目を `SalesItem#quantity` に、2 番目を `SalesItem#total` にマッピングしている。
(4)	マッピング対象となる JavaBean。

注釈: parameterMap 属性を使わずに、”Inline Parameter Maps” の仕組みでマッピングする事もできる。具体例は、Mybatis Developer Guide(PDF) の「Parameter Maps and Inline Parameters」(P.31) を参照されたい。

procedure 要素を実装する。

```
    <procedure id="findSalesItem" parameterMap="salesItemMap"> <!-- (1) -->
        {call sales_item(?,?,?)}
    </procedure>
```

項番	説明
(1)	呼び出す Procedure や Function を、”{call Procedure/Function 名 (IN パラメータ ...,OUT パラメータ...)}” の形式で指定する。 例では、sales_item という Function に対して、IN パラメータ 1 つと、OUT パラメータ 2 つを指定している。 バインドされる値は、parameterMap 要素で指定したマッピング定義の定義順となる。

sql 要素の実装例

sql 要素の実装する。

- xxx-sqlmap.xml

```
    <sql id="fragment_where_byFinished"> <!-- (1) -->
        WHERE
            finished = #finished#
    </sql>

    <select id="findByFinished"
            parameterClass="boolean"
            resultMap="resultMap_Todo">  <!-- (2) -->
        SELECT
            *
        FROM
            todo
        <include refid="fragment_where_byFinished" /> <!-- (3) -->
        ORDER BY
            created_at DESC
    </select>

    <select id="countByFinished"
            parameterClass="boolean"
            resultClass="long"> <!-- (4) -->
        SELECT
            count(*)
        FROM
            todo
        <include refid="fragment_where_byFinished" /> <!-- (5) -->
    </select>
```

項番	説明
(1)	(2) と (4) の SQL で共有する WHERE 句を定義している。include される SQL の定義は、include する側の SQL より先に、定義する必要がある。
(2)	条件に一致するデータを取得するための SQL
(3)	(1) で定義した WHERE 句が実装されている SQL を、include する。
(4)	条件に一致するデータ件数を取得するための SQL
(5)	(1) で定義した WHERE 句が実装されている SQL を、include する。

LOB 型更新の実装例

BLOB および CLOB などの Large Object を、データベースに更新する場合の実装例を、以下に示す。
下記は、BLOB を扱うテーブルへレコードを挿入する例となっている。

- DDL

```
CREATE TABLE upload_binary (
    file_id CHAR(36) NOT NULL,
    file_name VARCHAR(256) NOT NULL,
    content BLOB NOT NULL, -- (1)
    CONSTRAINT pk_upload_binary PRIMARY KEY (file_id)
);
```

項番	説明
(1)	BLOB 型のカラムを定義する。 上記例では、データベースとして Oracle を使用する前提の DDL となっている。

- DTO(JavaBean)

```
public class BinaryFile implements Serializable {
    // omitted
```

```
    private String fileId;
    private String fileName;
    private InputStream content; // (2)

    // omitted setter/getter

}
```

項番	説明
(2)	BLOB 型の値を保持するプロパティを java.io.InputStream 型で定義する。 上記例では、InputStream に、アップロードされたファイルの入力ストリームが設定される。

警告: BLOB を扱う プロパティの型は、原則 InputStream 型で定義することを推奨する。BLOB はバイト配列として扱うこともできるが、データの容量が大きくなると、メモリ枯渇の原因となる可能性がある。
CLOB を扱うプロパティの型は、 原則 java.io.Reader 型で定義することを推奨する。CLOB は文字列として扱うこともできるが、データの容量が大きくなると、メモリ枯渇の原因となる可能性がある。

- xxx-sqlmap.xml

```
<parameterMap id="uploadBinaryParameterMap"
              class="xxxxxx.yyyyyy.zzzzzz.domain.service.BinaryFile">
    <parameter property="fileId" />
    <parameter property="fileName" />
    <!-- (3) -->
    <parameter property="content"
               jdbcType="BLOB"
               typeHandler="jp.terasoluna.fw.orm.ibatis.support.BlobInputStreamTypeHandler" />
</parameterMap>

<!-- (4) -->
<insert id="uploadBinary" parameterMap="uploadBinaryParameterMap">
    INSERT INTO upload_binary
    (
        file_id
        ,file_name
        ,content
    )
    VALUES
    (
        ?
        ,?
        ,?
    )
```

```
    </insert>
```

項番	説明
(3)	BLOB 型のカラムの登録値を保持するパラメータに対して、登録するために必要な定義を指定する。 jdbcType 属性には"BLOB"を、typeHandler 属性には"jp.terasoluna.fw.orm.ibatis.support.BlobInputStreamTypeHandler"を指定する。
(4)	BLOB 型のカラムをもつテーブルに、レコードを登録するための SQL。

注釈: CLOB を扱う場合は、jdbcType 属性には"CLOB"を、typeHandler 属性には"jp.terasoluna.fw.orm.ibatis.support.ClobReaderTypeHandler"を指定する。

ちなみに: FQCN で指定しているクラス名は、 typeAlias 要素を使って別名を付与することで、シンプルに記載することができる。

```
    <!-- (5) -->
    <typeAlias alias="BinaryFile"
               type="xxxxxx.yyyyyy.zzzzzz.domain.service.BinaryFile"/>
    <typeAlias alias="BlobInputStreamTypeHandler"
               type="jp.terasoluna.fw.orm.ibatis.support.BlobInputStreamTypeHandler"/>

    <parameterMap id="uploadBinaryParameterMap"
                  class="BinaryFile"> <!-- (6) -->

        <!-- omitted -->

        <parameter property="content" jdbcType="BLOB"
                   typeHandler="BlobInputStreamTypeHandler" /> <!-- (6) -->

    </parameterMap>
```

項番	説明
(5)	typeAlias 要素を使って、クラス名 (FQCN) に別名を付与する。 上記例では、`BinaryFile` と `BlobInputStreamTypeHandler` クラスに対して、別名を付与している。 typeAlias 要素は、`sqlMapConfig.xml` と `xxx-sqlmap.xml` の両方で、定義することができる。
(6)	(5) で付与したクラス名 (FQCN) の別名を指定する。

- Service

```
// omitted

@Inject
UpdateDAO updateDAO;

// omitted

public BinaryFile uploadBinaryFile(String fileName,
        InputStream contentInputStream) {

    // (7)
    BinaryFile inputFile = new BinaryFile();
    inputFile.setFileId(UUID.randomUUID().toString());
    inputFile.setFileName(fileName);
    inputFile.setContent(contentInputStream);

    // (8)
    updateDAO.execute("example.uploadBinary", inputFile);

    return inputFile;
}

// omitted
```

項番	説明
(7)	レコードを登録するために必要な情報を、DTO に設定する。 上記例では、ファイル ID を UUID として採番し、引数で受け取ったファイル名と、ファイルの中身が格納されている `InputStream` オブジェクトを、DTO に設定している。
(8)	登録するために必要な情報を、保持する DTO を引数に、`UpdateDAO` を呼び出す。 DAO の呼び出し方法は、BLOB を扱わない場合と同じである。

- Controller

```
    @RequestMapping("uploadBinary")
    public String uploadBinaryFile(
            @RequestPart("file") MultipartFile multipartFile, Model model) throws IOException {
        // (9)
        BinaryFile uploadedFile = uploadService.uploadBinaryFile(multipartFile
                .getOriginalFilename(), multipartFile.getInputStream());
        model.addAttribute(uploadedFile);
        return "upload/form";
    }
```

項番	説明
(9)	アップロードされたファイルのファイル名と、ファイルの中身が格納されている `InputStream` を引数に、Service のメソッドを呼び出す。

LOB 型取得の実装例

BLOB および CLOB などの Large Object を、データベースから取得する場合の実装例を、以下に示す。
下記は、BLOB を扱うテーブルからレコードを取得する例となっている。
必要なテーブルを作成する DDL や DTO(JavaBean) は、*LOB* **型更新の実装例**を参照されたい。

- xxx-sqlmap.xml

```
    <resultMap id="selectBinaryResultMap" class="BinaryFile">
        <result property="fileId" column="file_id" />
        <result property="fileName" column="file_name" />
        <!-- (1) -->
        <result property="content" column="content" jdbcType="BLOB"
                typeHandler="BlobInputStreamTypeHandler" />
    </resultMap>
```

```
<!-- (2) -->
<select id="selectBinary" parameterClass="java.lang.String"
        resultMap="selectBinaryResultMap">
    SELECT
        *
    FROM
        upload_binary
    WHERE
        file_id = #fileId#
</select>
```

項番	説明
(1)	BLOB 型のカラムから取得した値を保持するプロパティに対して、値を取得するために必要な定義を指定する。 jdbcType 属性には"BLOB"を、typeHandler 属性には"jp.terasoluna.fw.orm.ibatis.support.BlobInputStreamTypeHandler"を指定する。
(2)	BLOB 型のカラムをもつテーブルから、レコードを取得するための SQL。

注釈: CLOB を扱う場合は、jdbcType 属性には"CLOB"を、typeHandler 属性には"jp.terasoluna.fw.orm.ibatis.support.ClobReaderTypeHandler"を指定する。

- Service / Repository

```
// omitted

@Inject
QueryDAO queryDAO;

// omitted

public BinaryFile getBinaryFile(String fileId) {
    // (3)
    BinaryFile loadedFile = queryDAO.executeForObject(
            "article.selectBinary", fileId, BinaryFile.class);
    return loadedFile;
}

// omitted
```

項番	説明
(3)	Controller から指定された取得条件を引数に、QueryDAO を呼び出す。 上記例では、ファイル ID に一致するアップロードファイルの情報を取得している。 DAO の呼び出し方法は、BLOB を扱わない場合と同じである。

SQL マッピングの実装例 (動的 SQL 編)

Mybatis では、動的に SQL を組み立てる仕組みが、デフォルトで用意されている。

以下に、動的に SQL を組み立てる方法について説明する。

詳細については、Developer Guide(PDF) の「Dynamic Mapped Statements」(P.48-53) を参照されたい。

パラメータオブジェクトの指定有無を判定

SQL に渡されたパラメータオブジェクトが指定されているかを判定し、SQL を組み立てることができる。

判定用の要素は、以下の通りである。

項番	要素	説明
1.	isParameterPresent	パラメータオブジェクトが指定されている (NULL でない) 時の SQL を組み立てるための要素。
2.	isNotParameterPresent	パラメータオブジェクトが指定されていない (NULL である) 時の SQL を組み立てるための要素。

実装例は、以下の通りである。

```
<select id="findOne" parameterClass="java.lang.Integer" resultMap="...">
    SELECT
        *
    FROM
        t_order
    WHERE

    <isParameterPresent> <!-- (1) -->
        id = #id#
    </isParameterPresent>

    <isNotParameterPresent> <!-- (2) -->
        1 = 2
    </isNotParameterPresent>

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

```
    </select>
```

項番	説明
(1)	例では、パラメータオブジェクトが指定されている時に、id カラムを WHERE 句に設定している。
(2)	例では、パラメータオブジェクトが指定されていない時に、一致するレコードが 0 件になるように、条件「1=2」を設定している。

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) parameterObject(id)=1
    SELECT * FROM t_order WHERE id = 1

    -- (2)
    SELECT * FROM t_order WHERE 1 = 2
```

パラメータオブジェクト (JavaBean) のプロパティの存在有無を判定

SQL に渡されたパラメータオブジェクト (JavaBean) に指定したプロパティが存在するか判定し、SQL を組み立てることができる。

判定用の要素は、以下の通りである。

項番	要素	説明
1.	isPropertyAvailable	指定したプロパティが、存在する時の SQL を組み立てるための要素。
2.	isNotPropertyAvailable	指定したプロパティが、存在しない時の SQL を組み立てるための要素。

実装例は、以下の通りである。

```
    <select id="findOne" parameterClass="OrderCriteria" resultMap="...">
        SELECT
            *
        FROM
            t_order
        WHERE

        <isPropertyAvailable property="statusCode"> <!-- (1) -->
            status_code = #statusCode#
        </isPropertyAvailable>
```

```
        <isNotPropertyAvailable property="statusCode"> <!-- (2) -->
            <![CDATA[
            status_code <> 'completed'
            ]]>
        </isNotPropertyAvailable>

        <!-- ... -->

    </select>
```

項番	説明
(1)	例では、statusCode プロパティが存在する場合に、status_code カラムが、statusCode と一致するレコードが取得されるように、WHERE 句を設定している。
(2)	例では、statusCode プロパティが存在しない場合に、status_code カラムが、'completed' 以外のレコードが取得されるように、WHERE 句を設定している。

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) statusCode='checking'
    SELECT * FROM t_order WHERE status_code = 'checking'

    -- (2)
    SELECT * FROM t_order WHERE status_code <> 'completed'
```

パラメータオブジェクト (JavaBean) のプロパティ値の設定有無を判定

SQL に渡されたパラメータオブジェクト (JavaBean) のプロパティに値が指定されているか判定し、SQL を組み立てることができる。

判定用の要素は、以下の通りである。

項番	要素	説明
1.	isNull	プロパティの値が、null 時の SQL を組み立てるための要素。
2.	isNotNull	プロパティの値が、null でない時の SQL を組み立てるための要素。
3.	isEmpty	プロパティの値が、null または、空の時の SQL を組み立てるための要素。Collection および、String に対して、指定することができる。
4.	isNotEmpty	プロパティの値が、null および、空でない時の SQL を組み立てるための要素。Collection および、String に対して、指定することができる。

実装例は、以下の通りである。

```
    <select id="findOne" parameterClass="OrderCriteria" resultMap="">
        SELECT
            *
        FROM
            t_order
        WHERE

        <isNull property="orderedDate"> <!-- (1) -->
            <![CDATA[
            CURRENT_DATE - '1 months'::interval <= ordered_date
            ]]>
        </isNull>

        <isNotNull property="orderedDate"> <!-- (2) -->
            ordered_date = #orderedDate#
        </isNotNull>

        <isEmpty property="statusCodes" prepend="AND"> <!-- (3) -->
            <![CDATA[
            status_code <> 'completed'
            ]]>
        </isEmpty>

        <isNotEmpty property="statusCodes" prepend="AND"> <!-- (4) -->
            status_code IN
            <iterate property="statusCodes" open="(" close=")" conjunction=",">
                #statusCodes[]#
            </iterate>
        </isNotEmpty>

    </select>
```

項番	説明
(1)	例では、orderedDate プロパティ (Date 型) の値が null の場合に、ordered_date カラムが、1 ヶ月前以降のレコードが取得されるように、WHERE 句を設定している。
(2)	例では、orderedDate プロパティ (Date 型) の値が null でない場合に、ordered_date カラムが orderedDate と一致するレコードが取得されるように、WHERE 句を設定している。
(3)	例では、statusCodes プロパティ (List<String>型) の値が、空の場合に、status_code カラムが、'completed' 以外のレコードが取得されるように、WHERE 句を設定している。
(4)	例では、statusCodes プロパティ (List<String>型) の値が、空でない場合に、status_code カラムが、statusCodes に格納されているいずれかの値と一致するレコードが取得されるように、WHERE 句を設定している。 iterate 要素の説明は、後述する。

上記の動的 SQL で生成される SQL は、以下 4 パターンとなる。

```
-- (1) orderedDate=null, statusCodes=[]
SELECT * FROM t_order WHERE CURRENT_DATE - '1 months'::interval <= ordered_date
    AND status_code <> 'completed'

-- (2) orderedDate=null, statusCodes=['accepted','checking']
SELECT * FROM t_order WHERE CURRENT_DATE - '1 months'::interval <= ordered_date
    AND status_code IN ('accepted','checking')

-- (3) orderedDate=2013/12/31, statusCodes=null
SELECT * FROM t_order WHERE ordered_date = '2013/12/31'
    AND status_code <> 'completed'

-- (4) orderedDate=2013/12/31, statusCodes=['accepted']
SELECT * FROM t_order WHERE ordered_date = '2013/12/31'
    AND status_code IN ('accepted')
```

パラメータオブジェクト (JavaBean) のプロパティ値を判定

SQL に渡されたパラメータオブジェクト (JavaBean) のプロパティに指定されている値を判定し、SQL を組み立てることができる。

判定用の要素は、以下の通りである。

項番	要素	説明
1.	isEqual	プロパティの値が、指定した値と一致する時の SQL を組み立てるための要素。
2.	isNotEqual	プロパティの値が、指定した値と一致しない時の SQL を組み立てるための要素。
3.	isGreaterThan	プロパティの値が、指定した値より大きい時の SQL を組み立てるための要素。
4.	isGreaterEqual	プロパティの値が、指定した値以上の時の SQL を組み立てるための要素。
5.	isLessThan	プロパティの値が、指定した値より小さい時の SQL を組み立てるための要素。
6.	isLessEqual	プロパティの値が、指定した値以下の時の SQL を組み立てるための要素。

実装例は、以下の通りである。

```
    <select id="findOne" parameterClass="OrderCriteria" resultMap="...">
        SELECT
            *
        FROM
            t_order
        WHERE
            (
            <![CDATA[
            status_code <> 'completed'
            ]]>
            <isEqual property="containCompletedOrder"
                     compareValue="true"
                     prepend="OR"> <!-- (1) -->
                status_code = 'completed'
            </isNull>
            )

    </select>
```

項番	説明
(1)	例では、containCompletedOrder プロパティ (Boolean 型) の値が、true の場合に、status_code カラムが'completed' のレコードも取得されるように、WHERE 句を設定している。

注釈: compareProperty 属性を使用することで、JavaBean 内の別のプロパティの値と、比較することもできる。

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) containCompletedOrder=false
SELECT * FROM t_order WHERE (status_code <>  'completed')

-- (2) containCompletedOrder=true
SELECT * FROM t_order WHERE (status_code <>  'completed' OR status_code = 'completed')
```

判定要素の共通属性

動的 SQL を組み立てるための要素には、以下の共通的な属性が存在する。

項番	属性	説明
1.	prepend	動的 SQL を組み立てるための判定要素で、true と判断され、SQL が組み立てられた際に、SQL の先頭に設定する文字列を指定する。
2.	open	動的 SQL を組み立てるための判定要素の中で、組み立てた SQL の前に追加する文字列を指定する。
3.	close	動的 SQL を組み立てるための判定要素の中で、組み立てた SQL の後に付与する文字列を指定する。

実装例は、以下の通りである。

```
<select id="findOne" parameterClass="OrderCriteria" resultMap="...">
    SELECT
        *
    FROM
        t_order

    <isNotEmpty property="statusCode"
                prepend="WHERE"
                open="("
                close=")"> <!-- (1) -->
        status_code = #statusCode#
        <isEqual property="containCompletedOrder" compareValue="true" prepend="OR">
```

```
                status_code = 'completed'
            </isEqual>
        </isNotEmpty>

    </select>
```

項番	属性	説明
(1)	-	例では、statusCode プロパティに値が指定されている場合に、status_code カラムを WHERE 句にし、containCompletedOrder プロパティ (Boolean 型) の値が true の場合は、status_code カラムが'completed' のレコードも取得されるように、WHERE 句を設定している。
-	prepend	statusCode プロパティに値が指定されている場合に、SQL に"WHERE"を設定している。
-	open	containCompletedOrder プロパティ (Boolean 型) の値が、true の場合は、OR 条件を加えるため、status_code カラムに対する条件をグループ化するための開始文^"("を指定している。
-	close	status_code カラムに対する条件をグループ化するための終了文字")"を指定している。

上記の動的 SQL で生成される SQL は、以下 3 パターンとなる。

```
    -- (1) statusCode=null, containCompletedOrder=false
    SELECT * FROM t_order

    -- (2) statusCode='accepted', containCompletedOrder=false
    SELECT * FROM t_order WHERE (status_code = 'accepted')

    -- (3) statusCode='checking', containCompletedOrder=true
    SELECT * FROM t_order WHERE (status_code = 'checking' OR status_code = 'completed')
```

コレクションの繰り返し

SQL に渡されたバインド値が、コレクションや配列の場合、コレクションおよび配列の要素分処理を繰り返して、SQL を組み立てることができる。

要素は、以下の通りである。

項番	要素	説明
1.	iterate	コレクションおよび配列に対して、繰り返し処理を行い、SQL を組み立てるための要素。

実装例は、以下の通りである。

```
    <select id="findOne" parameterClass="OrderCriteria" resultMap="...">
        SELECT
            *
        FROM
            t_order

        <isNotNull property="statusCodes" prepend="WHERE">
            <iterate property="statusCodes"
                     prepend="status_code IN"
                     open="("
                     conjunction=","
                     close=")" > <!-- (1) -->
                #statusCodes[]#
            </iterate>
        </isNotNull>

    </select>
```

項番	属性	説明
(1)	-	例では、statusCodes プロパティ (List<String>) に格納されている値を、IN 句の値として設定している。
-	prepend	コレクションまたは配列の要素が存在する場合に、最初に設定する文字列を指定する。例では、条件に加えるカラム名と、IN 句を指定している。
-	open	コレクションまたは配列の最初の要素を処理する前に設定する文字列を指定する。例では、IN 句に指定する値の開始囲い文字"("を指定している。
-	conjunction	次の要素がある場合、次の要素の処理を行う前に設定する文字列を指定する。例では、IN 句に指定する値の区切り文字","を指定している。
-	close	コレクションまたは配列の最後の要素の処理を行った後に、設定する文字列を指定する。例では、IN 句に指定する値の終了囲い文字")"を指定している。

注釈: 上記例は、JavaBean の中のプロパティがコレクションの場合の実装例であるが、パラメータオブジェクト自体をコレクションにすることもできる。その場合は、property 属性は指定せず、#[]#という形式でアクセスすることができる。

コレクションには、JavaBean を格納することもでき、JavaBean にネストされているコレクションにも、アクセスすることができる。詳細は、Developer Guide(PDF) の「Dynamic Mapped Statements」(P.52) を参照されたい。

上記の動的 SQL で生成される SQL は、以下 3 パターンとなる。

```
-- (1) statusCodes=null
SELECT * FROM t_order

-- (2) statusCodes=[]
SELECT * FROM t_order

-- (3) statusCodes=['accepted','checking']
SELECT * FROM t_order WHERE status_code IN ('accepted' , 'checking')
```

動的 SQL のブロック化

個々の動的 SQL をブロック化することで、ブロック全体として、prepend, open, close 属性を制御することができる。

要素は、以下の通りである。

項番	要素	説明
1.	dynamic	動的 SQL を組み立てる要素をブロック化するための要素。

実装例は、以下の通りである。

```
<select id="findOne" parameterClass="OrderCriteria" resultMap="...">
    SELECT
        *
    FROM
        t_order
    WHERE

    <dynamic prepend="WHERE"
             open="("
             close=")"> <!-- (1) -->

        <isNotEmpty property="id" prepend="AND"> <!-- (2) -->
            id = #id#
        </isNotEmpty>

        <isNotEmpty property="statusCode" prepend="AND"> <!-- (3) -->
            status_code = #statusCode#
        </isNotEmpty>

    </dynamic>

</select>
```

項番	属性	説明
(1)	-	(2) と (3) の動的 SQL を、ブロック化している。
-	prepend	ブロック内で組み立てた SQL の先頭に設定する文字列を指定する。ここで指定した値は、ブロック内で最初に一致した動的 SQL の prepend 属性の値として使用される。 上記例だと、id プロパティに値を指定した場合、(2) の prepend 属性の値は、"AND"ではなく、"WHERE"となる。
-	open	ブロック中で組み立てた SQL の前に、追加する文字列を指定する。
-	close	ブロック中で組み立てた SQL の後に、追加する文字列を指定する。

上記の動的 SQL で生成される SQL は、以下 4 パターンとなる。

```
-- (1) id=null, statusCode=null
SELECT * FROM t_order

-- (2) id=1, statusCode=null
SELECT * FROM t_order WHERE (id = 1)

-- (3) id=null, statusCode='accepted'
SELECT * FROM t_order WHERE (status_code = 'accepted')

-- (4) id=1, statusCode='accepted'
SELECT * FROM t_order WHERE (id = 1 AND status_code = 'accepted')
```

QueryDAO の使用例

1 件検索

検索結果が、0〜1 件となるクエリを発行したい場合、以下のような実装となる。

- Xxx.java

```
String todoId = "xxxxx....";
Todo loadedTodo = queryDAO.executeForObject( // (1)
        "todo.findOne",     // (2)
        todoId,             // (3)
```

```
            Todo.class);        // (4)
    if (loadedTodo == null) {   // (5)
        // ...                  // (6)
    }
```

項番	説明
(1)	検索結果を (4) で指定した型のオブジェクトとして取得するためのメソッド (QueryDAO#executeForObject) を呼び出す。
(2)	検索結果が 0〜1 件となる SQL の SQLID を指定する。 検索結果が複数件になる場合は、Mybatis が java.sql.SQLException を発生させる。
(3)	SQL のバインドパラメータを指定する。 例では、java.lang.String にしているが、複数のパラメータ (検索条件) を渡したい場合は、JavaBean を指定することもできる。
(4)	SQL の取得結果をマッピングするオブジェクトの型を指定する。
(5)	検索結果が 0 件の場合は、null になるので、null 判定が必要である。
(6)	検索結果が、0 件の場合の処理を実装する。

複数件検索

検索結果が、0〜N 件となるクエリを発行し、条件に一致するデータをすべて取得する場合は、以下のような実装となる。

- Xxx.java

```
    boolean finished = false;
    List<Todo> unfinishedTodoList = queryDAO.executeForObjectList( // (1)
            "todo.findByFinished",      // (2)
            finished);                  // (3)
    if(unfinishedTodoList.isEmpty()){   // (4)
        // ...                          // (5)
```

```
    }
```

項番	説明
(1)	オブジェクトのリストを取得するための、メソッドを呼び出す。
(2)	検索結果が、0～N 件となる SQL の SQLID を指定する。
(3)	SQL のバインドパラメータを指定する。 例では、boolean にしているが、複数のパラメータ (検索条件) を渡したい場合は、JavaBean を指定することもできる。
(4)	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 チェックは不要である。
(5)	検索結果が、0 件の場合の処理を実装する。

ページネーション検索（TERASOLUNA DAO 標準機能方式）

検索結果が、0～N 件となるクエリを発行し、条件に一致するデータの一部 (指定ページ部分) を取得する場合は、以下のような実装となる。

以下の例では、TERASOLUNA DAO から提供されている API を使って、実現する実装例となっている。

> **警告: 検索条件に一致するデータ件数が非常に多くなる場合の注意点**
> TERASOLUNA DAO 標準機能のページネーション検索は、`java.sql.ResultSet#next` を使って取得するレコードの開始位置までスキップする実装となっているため、検索条件に一致するデータ件数が、非常に多い場合、処理性能に影響を与える可能性がある。検索条件に一致するデータ件数が、非常に多くなる可能性がある場合は、TERASOLUNA DAO 標準機能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すること。

- Xxx.java

```
    Pageable pageable = new PageRequest(0, 10); // (1)
    boolean finished = false;
    long totalCount = queryDAO.executeForObject(
            "todo.countByFinished", // (2)
```

```
            finished,
            Long.class);            // (3)

    List<Todo> unfinishedTodoList = null;
    if(0 < totalCount) {
        unfinishedTodoList = queryDAO.executeForObjectList(
            "todo.findByFinished",   // (4)
            finished,
            pageable.getOffset(),    // (5)
            pageable.getPageSize()); // (6)
    } else {
        unfinishedTodoList = new ArrayList<Todo>();
    }

    Page<Todo> page = new PageImpl<Todo>( // (7)
            unfinishedTodoList, // (8)
            pageable,           // (9)
            totalCount);        // (10)
```

- xxx-sqlmap.xml

```
    <select id="findByFinished"
            parameterClass="boolean"
            resultMap="resultMap_Todo"> <!-- (11) -->
        SELECT
            *
        FROM
            todo
        WHERE
            finished = #finished#
        ORDER BY
            created_at DESC
    </select>
```

項番	説明
(1)	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.PageRequest`）を生成する。Pageable オブジェクトは、リクエストパラメータに指定して、Controller の引数として受け事もできる。詳細は、ページネーションを参照されたい。
(2)	条件に一致するデータの合計件数を、取得するための SQL の、SQLID を指定して実行する。
(3)	件数の取得なので、Long.class を指定する。
(4)	検索結果が、0〜N 件となる SQL の、SQLID を指定して実行する。
(5)	取得開始位置を指定する。 0 開始。取得件数が 10 件のときに、10 を指定すると、11〜20 件目が取得される。
(6)	取得件数を指定する。 取得開始位置が 0 のときに、10 を指定すると、1〜10 件目が取得される。
(7)	Spring Data より提供されているページ用のオブジェクト（`org.springframework.data.domain.PageImpl`）を生成する。
(8)	ページネーション検索して、取得したリストを指定する。
(9)	ページネーション検索で使用したページング検索用のオブジェクト (Pageable) を指定する。
(10)	条件に一致するデータの、合計件数を指定する。
(11)	SQL の実装例。SQL としては、取得位置を意識する必要はない。

ページネーション検索（SQL 絞り込み方式）

検索結果が 0〜N 件となるクエリを発行し、条件に一致するデータの一部 (指定ページ部分) を取得する場合は、以下のような実装となる。
以下の例では、TERASOLUNA DAO から提供されている API を使わずに、SQL を使って実現する実装例となっている。

- PageableBindParams.java (サンプルクラス)

```
public class PageableBindParams<P> implements Serializable { // (1)
    private static final long serialVersionUID = 1L;
    private final P bindParams;
    private final Pageable pageable;
    public PageableBindParams(P bindParams, Pageable pageable) {
        this.bindParams = bindParams;
        this.pageable = pageable;
    }
    public P getBindParams() {
        return bindParams;
    }
    public Pageable getPageable() {
        return pageable;
    }
}
```

- Xxx.java

```
Pageable pageable = new PageRequest(0, 10);
boolean finished = false;
long totalCount = queryDAO.executeForObject(
        "todo.countByFinished",
        finished,
        Long.class); // (2)

List<Todo> unfinishedTodoList = null;
if(0 < totalCount) {
    PageableBindParams<Boolean> pageableBindParams =
            new PageableBindParams<Boolean>( // (3)
                    finished,  // (4)
                    pageable); // (5)
    unfinishedTodoList = queryDAO.executeForObjectList(
            "todo.findPageByFinished", // (6)
            pageableBindParams);       // (7)
} else {
    unfinishedTodoList = new ArrayList<Todo>();
}

Page<Todo> page = new PageImpl<Todo>(
        unfinishedTodoList,
```

```
            pageable,
            totalCount); // (8)
```

- xxx-sqlmap.xml

```
    <select id="findPageByFinished"
            parameterClass="xxxxxx.yyyyyy.zzzzzz.domain.dto.PageableBindParams"
            resultMap="resultMap_Todo"> <!-- (9) -->
        SELECT
            *
        FROM
            todo
        WHERE
            finished = #bindParams#
        ORDER BY
            created_at DESC
        OFFSET
            #pageable.offset#    /* (10) */
        LIMIT
            #pageable.pageSize#  /* (11) */
    </select>
```

項番	説明
(1)	検索条件となるパラメータ（バインドパラメータ）と、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`）を保持する JavaBean。DAO に渡せるバインドオブジェクトは一つのみなので、本クラスのような集約オブジェクトが、必要となる。本クラスは、サンプル実装なので、各プロジェクトで必要に応じて用意すること。
(2)	TERASOLUNA DAO 標準機能使用時と同様に、合計件数を取得する。
(3)	DAO に渡すバインド用オブジェクトを生成する。 例では、(1) で用意したクラスを使用する。
(4)	対象データを絞り込むための、検索条件を指定する。 例では、finished の値として、「false」を指定する。
(5)	該当ページのデータを絞り込むための、検索条件を指定する。 例では、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.PageRequest`）を指定している。 Pageable オブジェクトは、リクエストパラメータに指定して、Controller の引数として受けることもできる。詳細は、ページネーションを参照されたい。
(6)	該当ページのデータを抽出する SQL が実装されている SQL の SQLID を指定する。
(7)	(3) で生成したバインド用オブジェクトを指定する。
(8)	TERASOLUNA DAO 標準機能使用時と同様に、Spring Data より提供されているページ用のオブジェクト（`org.springframework.data.domain.PageImpl`）を生成する。
(9)	SQL の実装例。例では、PostgreSQL から提供されている機能 (OFFSET,LIMIT) を使用している。SQL として、取得位置を意識する。
(10)	取得開始位置を指定する。 0 開始。取得件数が 10 件のとき、10 件目から 20 件目が取得される。 (PostgreSQL の機能を使用する)
(11)	取得件数を指定する。

UpdateDAO の使用例

1 件挿入

1 件のデータの挿入する場合、以下のような実装となる。

- Xxx.java

```
// (1)
Todo todo = new Todo();
todo.setTodoId(todoId);
todo.setTodoTitle(todoTitle);
todo.setFinished(false);
todo.setCreatedAt(now);
int insertedCount = updateDAO.execute("todo.insert", todo); // (2)
if(insertedCount != 1){  // (3)
    // ...               // (4)
}
```

項番	説明
(1)	挿入対象のデータ (JavaBean) を生成する。
(2)	挿入用 SQL の SQLID と、挿入対象のデータ (JavaBean) を指定して、DAO を実行する。
(3)	必要に応じて、実際に挿入されたデータの件数を、チェックする。例では、挿入件数が 1 件であるかをチェックしている。
(4)	必要に応じて、実際に挿入された件数が、想定件数と異なる場合の処理を行う。

複数件挿入 (バッチ実行)

複数の SQL を、バッチ実行することで、複数件のデータを挿入する場合は、以下のような実装となる。
TERASOLUNA DAO から提供されている `jp.terasoluna.fw.dao.SqlHolder` を使用する。

- Xxx.java

```
// (1)
Todo todo = new Todo();
todo.setTodoId(todoId);
todo.setTodoTitle(todoTitle);
```

```
    todo.setFinished(false);
    todo.setCreatedAt(now);

    // (2)
    Todo todo2 = new Todo();
    todo2.setTodoId(todoId2);
    todo2.setTodoTitle(todoTitle2);
    todo2.setFinished(false);
    todo2.setCreatedAt(now);

    List<SqlHolder> sqlHolders = new ArrayList<SqlHolder>(); // (3)
    sqlHolders.add(new SqlHolder("todo.insert", todo));      // (4)
    sqlHolders.add(new SqlHolder("todo.insert", todo2));     // (4)
    int insertedCount = updateDAO.executeBatch(sqlHolders);  // (5)
    if(insertedCount != 2){  // (6)
        // ...               // (7)
    }
```

項番	説明
(1)	挿入対象のデータ (JavaBean) を生成する。1 件目のデータ。
(2)	挿入対象のデータ (JavaBean) を生成する。2 件目のデータ。
(3)	バッチ実行用に、TERASOLUNA DAO から提供されている jp.terasoluna.fw.dao.SqlHolder のリストを生成する。
(4)	(1), (2) で生成したデータを、バインド用オブジェクトとして、SqlHolder のリストに追加する。例では、2 件リストに追加している。
(5)	(1)〜(4) で生成した SqlHolder のリストを指定して、バッチを実行する。
(6)	必要に応じて、実際に挿入されたデータの件数をチェックする。 例では、挿入件数が 2 件であるかをチェックしている。
(7)	必要に応じて、実際に挿入された件数が、想定件数と異なる場合の処理を行う。

警告: バッチ実行における挿入件数について
バッチ実行した場合、JDBC ドライバによっては、正確な行数が取得できないケースがある。正確に取得できないドライバを使用する場合に、挿入件数をチェックする必要があるケースで、バッチ実行を使用しないこと。(更新時の更新件数、削除時の削除件数も同様である。)

1 件更新

1 件のデータの更新する場合、以下のような実装となる。

1 件挿入の場合と同じである。使用する SQL が、更新用の SQL になる。

- Xxx.java

```
Todo loadedTodo = queryDAO.executeForObject("todo.findOne",
        todoId,
        Todo.class);     // (1)
todo2.setFinished(true); // (2)
int updatedCount = updateDAO.execute("todo.update", todo); // (3)
if(updatedCount != 1){   // (4)
    // ...               // (5)
}
```

項番	説明
(1)	更新対象のデータ (JavaBean) を、検索する。
(2)	データを更新する。例では、finished を、false から true に更新する。
(3)	更新用 SQL の SQLID と、更新対象のデータ (JavaBean) を指定して、DAO を実行する。
(4)	必要に応じて、実際の更新されたデータの件数をチェックする。 例では、更新件数が 1 件であるかをチェックしている。
(5)	必要に応じて、実際に更新された件数が、想定件数と異なる場合の処理を行う。

複数件更新 (バッチ実行)

複数の SQL をバッチ実行することで、複数件のデータを更新する場合の実装例は、複数件挿入 (バッチ実行) と同じである。
更新値が、レコード毎に異なる場合、バッチ実行による複数件更新が有効である。

複数件更新 (WHERE 句指定)

SQL で指定した条件に一致するデータを一括で更新する場合、以下のような実装となる。
全レコードを同じ値に一括更新する場合は、WHERE 句指定による複数件更新が有効的である。

- Xxx.java

```
int deadlineDays = 7;
int updatedCount = updateDAO.execute("todo.update", deadlineDays); // (1)
```

- xxx-sqlmap.xml

```
<update id="updateFinishedDeadlineByUnfinished" parameterClass="int"> <!-- (2) -->
    <![CDATA[
    UPDATE
        todo
    SET
        todo_title = '[Finished Deadline] ' || todo_title
        ,version = (version + 1)
    WHERE
        finished = false
    AND
        created_at < current_date - #deadlineDays#
    ]]>
</update>
```

項番	説明
(1)	一括更新用 SQL の SQLID と、更新対象のデータを抽出するための条件を指定して、DAO を実行する。
(2)	一括更新する SQL の実装例。例では、作成してから 7 日経過して、完了していない TODO のタイトルに”[Finished Deadline] “という文字列を先頭に付与している。

1 件削除

1 件のデータの削除する場合、以下のような実装となる。

- Xxx.java

```
String todoId = "xxxxx....";
int deletedCount = updateDAO.execute("todo.delete", todoId); // (1)
if(deletedCount != 1){
    // ...               // (2)
}
```

項番	説明
(1)	削除用 SQL の SQLID と PK を指定して、DAO を実行する。 例では、`java.lang.String` にしているが、複合キーの場合は、JavaBean を指定することもできる。
(2)	必要に応じて、実際に削除された件数が、想定件数と異なる場合の処理を行う。

複数件削除 (バッチ実行)

複数の SQL をバッチ実行することで、複数件のデータを削除する場合の実装例は、複数件更新 (バッチ実行) と同じである。

1 件削除時の処理を共有する必要がある場合は、バッチ実行による複数件削除を使用する。ただし、削除する対象データが大量になる場合は、WHERE 句指定による一括削除の方式を検討した方がよい。

複数件削除 (WHERE 句指定)

SQL で指定した条件に一致するデータを一括で削除する場合の実装例は、複数件更新 (WHERE 句指定) と同じである。

削除対象のレコードが大量になる場合は、WHERE 句指定による複数件削除が有効的である。

StoredProcedureDAO の使用例

プロシージャや、ファンクションを呼び出す場合、以下のような実装となる。

- Xxx.java

```
SalesItem item = new SalesItem(); // (1)
item.setId(Integer.valueOf(1));  // (2)
storedProcedureDAO.executeForObject("todo.findSalesItem", item);  // (3)
// (4)
logger.debug("Quantity is {}.", item.getQuantity());
logger.debug("Total is {}.", item.getTotal());
```

項番	説明
(1)	プロシージャや、ファンクションの IN パラメータを、OUT パラメータを保持するバインド用オブジェクトを生成する。
(2)	IN パラメータとして、ID ををを設定する。例では、ID として、`1` を設定している。
(3)	ストアードプロシージャ呼び出し用 SQL の、SQLID とバインド用オブジェクトを引数に、`StoredProcedureDAO` のメソッドを呼び出す。
(4)	`StoredProcedureDAO` のメソッドの呼び出しが、正常に終了した場合、 プロシージャや、ファンクションの OUT パラメータが、バインド用オブジェクトに設定される。 例では、バインド用オブジェクトに設定された OUT パラメータの値を、ログに出力している。

QueryRowHandleDAO の使用例

- Xxx.java

```
    boolean finished = false;
    queryRowHandleDAO.executeWithRowHandler(
            "todo.findByFinished",              // (1)
            finished,                           // (2)
            new DataRowHandler() {              // (3)
                public void handleRow(Object valueObject) { // (4)
                    Todo todo = (Todo) valueObject;
                    logger.info(todo.toString());  // (5)
                }
            });
```

項番	説明
(1)	検索結果が、0〜N 件となる SQL の、SQLID を指定する。
(2)	SQL のバインドパラメータを指定する。 例では、boolean にしているが、複数のパラメータ (検索条件) を渡したい場合は、JavaBean を指定することもできる。
(3)	`jp.terasoluna.fw.dao.event.DataRowHandler` の実装オブジェクトを指定する。 例では、無名クラスを使用しているが、実際のプロジェクトでは、実装クラスを作成することを検討すること。
(4)	検索結果の 1 レコード毎に、handleRow メソッドが呼び出される。 引数にわたってくるオブジェクトは、select 要素に resultClass 属性、または、resultMap 要素の class 属性に指定したクラスのオブジェクトとなる。
(5)	例では、ログ出力しているだけだが、実際のプロジェクトで使う場合は、値の加工、各レコード値の集計、ファイル出力などの処理を行うことになる。

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ことで、実現できる。

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。

- xxx-sqlmap.xml

```
// (1) (2)
<select id="findAllByWord" parameterClass="String" resultMap="resultMap_Article">
  SELECT
      *
  FROM
      article
  WHERE
      title LIKE '%' || #word# || '%' ESCAPE '~'
  OR
      overview LIKE '%' || #word# || '%' ESCAPE '~'
</select>
```

項番	説明
(1)	SQL 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。

- Service or Repository

```
@Inject
QueryDAO queryDAO;

@Transactional(readOnly = true)
public Page<Article> searchArticle(ArticleSearchCriteria criteria,
        Pageable pageable) {

    String escapedWord = QueryEscapeUtils.toLikeCondition(criteria.getWord()); // (3)

    long total = queryDAO.executeForObject("article.countByWord",
            escapedWord, Long.class);
    List<Article> contents = null;
    if (0 < total) {
        contents = queryDAO.executeForObjectList("article.findAllByWord",
                escapedWord, pageable.getOffset(), pageable.getPageSize()); // (4)
    } else {
```

```
            contents = Collections.emptyList();
        }
        return new PageImpl<Article>(contents, pageable, total);
    }
```

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、`QueryDAO` のバインドパラメータに渡す。

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。

- `xxx-sqlmap.xml`

```
    // (1)
    <select id="findAllByWord" parameterClass="String" resultMap="resultMap_Article">
      SELECT
          *
      FROM
          article
      WHERE
          title LIKE #word# ESCAPE '~'
      OR
          overview LIKE #word# ESCAPE '~'
    </select>
```

項番	説明
(1)	SQL 内に、LIKE 検索用のワイルドカードは、指定しない。

- Service or Repository

```
    @Inject
    QueryDAO queryDAO;

    @Transactional(readOnly = true)
    public Page<Article> searchArticle(ArticleSearchCriteria criteria,
            Pageable pageable) {
```

```
        String escapedWord  = QueryEscapeUtils
                .toContainingCondition(criteria.getWord()); // (2)

        long total = queryDAO.executeForObject("article.countByWord",
                escapedWord, Long.class);
        List<Article> contents = null;
        if (0 < total) {
            contents = queryDAO.executeForObjectList("article.findAllByWord",
                    escapedWord, pageable.getOffset(), pageable.getPageSize()); // (3)
        } else {
            contents = Collections.emptyList();
        }
        return new PageImpl<Article>(contents, pageable, total);
    }
```

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、`QueryDAO` のバインドパラメータに渡す。

SQL Injection 対策について

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。

Mybatis2 では、SQL に値を埋め込む仕組みを、2 つ提供している。

- バインド変数を使って埋め込む方法。

  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。

- 置換変数を使って埋め込む方法。

  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、かならず SQL Injection が発生しないことを保障するための、入力チェックを実施すること。
> 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

バインド変数を使用する場合は、 ParameterMap または Inline Parameters を使用する。
以下に、使用例を示す。

ParameterMap の使用例を、以下に示す。

```
<!-- (1) -->
<parameterMap id="uploadBinaryParameterMap" class="BinaryFile">
    <parameter property="fileId" />
    <parameter property="fileName" />
    <parameter property="content" jdbcType="BLOB" typeHandler="BlobInputStreamTypeHandler" />
</parameterMap>

<insert id="uploadBinary" parameterMap="uploadBinaryParameterMap">
    INSERT INTO upload_binary
    (
        file_id
        ,file_name
        ,content
    )
    VALUES
    (
        ?   /* (2) */
        ,?
        ,?
    )
</insert>
```

項番	説明
(1)	バインド変数として値を埋め込むプロパティを定義する。定義した順番が、(2) で指定している?の位置に対応する。
(2)	SQL にバインド変数を指定する。(1) で定義した順番で、?の部分に、値がバインドされる。

Inline Parameters の使用例を、以下に示す。

```
    <insert id="insert"
            parameterClass="xxxxxx.yyyyyy.zzzzzz.domain.model.Todo"> <!-- (1) -->
        INSERT INTO todo
            (
                todo_id
                ,todo_title
                ,finished
                ,created_at
                ,version
            )
            values(
                #todoId#        /* (3) */
                ,#todoTitle#
                ,#finished#
                ,#createdAt#
                ,1
            )
    </insert>
```

項番	説明
(3)	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、#で囲み、バインド変数として指定する。

置換変数を使って埋め込む方法

バインド変数を使用する場合の使用例を、以下に示す。

```
    <select id="findByFinished"
            parameterClass="..."
            resultMap="resultMap_Todo">
        SELECT
            *
        FROM
            todo
        WHERE
            finished = #finished#
        ORDER BY
            created_at $direction$  /* (4) */
    </select>
```

項番	説明
(4)	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$で囲み、置換変数として指定する。 上記例では、$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実際に使用する安全な値をペアで Map に格納し、

```
Map<String, String> safeValueMap = new HashMap<String, String>();
safeValueMap.put("1", "ASC");
safeValueMap.put("2", "DESC");
```

実際の入力はコード値になるようにして、SQL を実行する処理中で変換することが望ましい。

```
String direction = safeValueMap.get(input.getDirection());
```

コードリストを使用しても良い。

5.3.3 Appendix

関連オブジェクトを１回の SQL でまとめて取得する実装例

テーブル毎に Entity のような JavaBean を用意して、データベースにアクセスする際に、関連オブジェクトを、1 回の SQL でまとめて取得する方法について説明する。
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。

警告: 以下の点に注意して、使用すること。

- 本例では、使い方を説明するために、すべての関連オブジェクトを、1 回の SQL でまとめて取得している。しかしながら、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オブジェクトのみ取得するようにすること。なぜなら、使用しない関連オブジェクトを、同時に取得してしまった場合、性能劣化の原因となるケースがあるからである。
- 使用頻度の低い、1:N の関係をもつ関連オブジェクトについては、まとめて取得しない。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。性能要件を満たせる場合は、まとめて取得してもよい。
- 1:N の関係となる関連オブジェクトが、多く含まれる場合、まとめて取得すると、マッピング処理に使用されない無駄なデータの取得が行われ、性能劣化の原因となるケースがある。性能要件を満たせる場合は、まとめて取得してもよいが、他の方法を検討した方がよい。

ちなみに: N+1 問題の回避手段については、Mybatis Developer Guide(PDF) の「Result Maps/Avoiding N+1 Selects (1:1)」(P.37-38) 及び「Result Maps/Avoiding N+1 Selects (1:M and M:N)」(P.39-40) を参照されたい。

以降では、注文テーブルを使って、具体的に実装例について説明する。
説明で使用するテーブルは、以下の通りである。

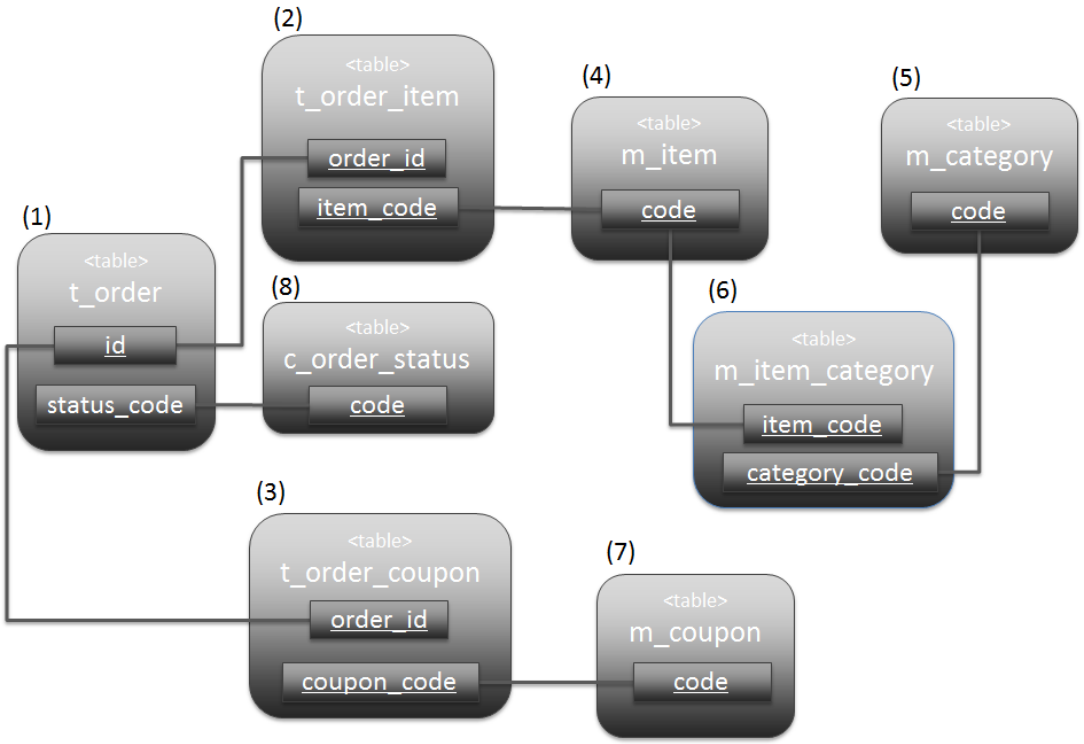


図 5.15 **Picture - ER diagram**

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	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。

トランザクション系テーブルのレイアウトと、格納されているレコードは、以下の通りである。

t_order

id(PK)	status_code
1	accepted
2	checking

t_order_item

order_id(PK)	item_code(PK)	quantity
1	ITM0000001	10
1	ITM0000002	20
2	ITM0000001	30
2	ITM0000002	40

t_order_coupon

order_id(PK)	coupon_code(PK)
1	CPN0000001
1	CPN0000002

マスタ系テーブルのレイアウトと、格納されているレコードは、以下の通りである。

m_item

code(PK)	name	price
ITM0000001	Orange juice	100
ITM0000002	NotePC	100000

m_category

code(PK)	name
CTG0000001	Drink
CTG0000002	PC
CTG0000003	Hot selling

m_item_category

item_code(PK)	category_code(PK)
ITM0000001	CTG0000001
ITM0000002	CTG0000002
ITM0000002	CTG0000003

m_coupon

code(PK)	name	price
CPN0000001	Join coupon	3000
CPN0000002	PC coupon	30000

コード系テーブルのレイアウトと、格納されているレコードは、以下の通りである。

c_order_status

code(PK)	name
accepted	Order accepted
checking	Stock checking
shipped	Item Shipped

以降で説明する実装例では、上記テーブルに格納されているデータを、以下の JavaBean にマッピングして、取得する。

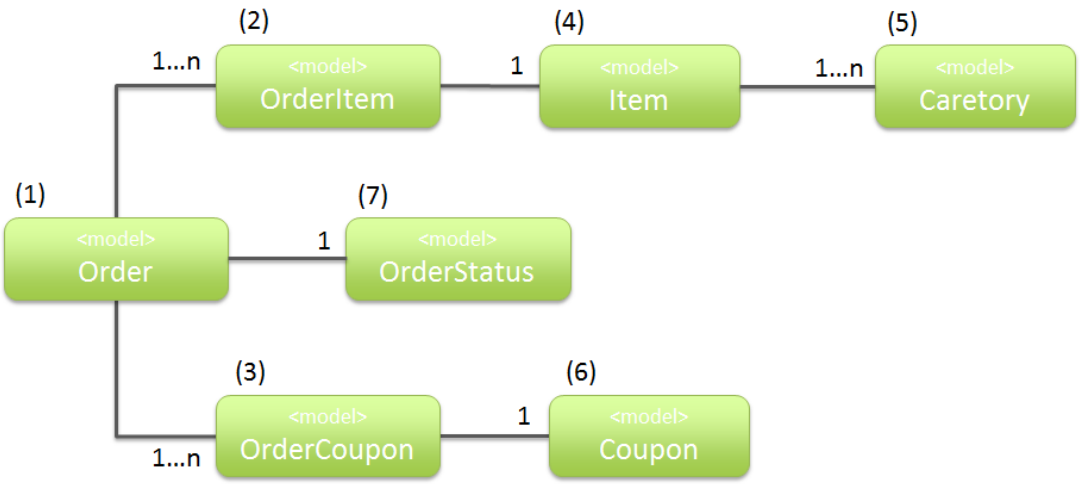


図 5.16　**Picture - Class(JavaBean) diagram**

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。 関連オブジェクトとして、`OrderStatus` と `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 JavaBean。 関連オブジェクトとして、`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 JavaBean。関連オブジェクトとして、`Coupon` を保持する。
(4)	Item	m_item テーブルの 1 コードを表現する 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。`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 JavaBean。
(6)	Coupon	m_coupon テーブルの 1 レコードを表現する JavaBean。
(7)	OrderStatus	c_order_status テーブルの 1 レコードを表現する JavaBean。

JavaBean のプロパティ定義は、以下の通りである。

- Order.java

```
public class Order implements Serializable {
    private int id;
    private List<OrderItem> orderItems;
    private List<OrderCoupon> orderCoupons;
    private OrderStatus status;
    // ...
}
```

- OrderItem.java

```
public class OrderItem implements Serializable {
    private int orderId;
    private String itemCode; // <!-- (1) -->
    private Item item;
    private int quantity;
```

```
        // ...
    }
```

項番	説明
(1)	保持する値が、直後の変数 item の code プロパティと重複する。これは、後述する resultMap 要素の、groupBy 属性によるレコードの、グルーピングを行う際に必要になるため、定義している。

- OrderCoupon.java

```
    public class OrderCoupon implements Serializable {
        private int orderId;
        private String couponCode; // (1)
        private Coupon coupon;
        // ...
    }
```

項番	説明
(1)	保持する値が、直後の変数 Coupon の code プロパティと重複する。これは、後述する resultMap 要素の、groupBy 属性によるレコードの、グルーピングを行う際に必要になるため、定義している。

- Item.java

```
    public class Item implements Serializable {
        private String code;
        private String name;
        private int price;
        private List<Category> categories;
        // ...
    }
```

- Category.java

```
    public class Category implements Serializable {
        private String code;
        private String name;
        // ...
    }
```

- Coupon.java

```
    public class Coupon implements Serializable {
        private String code;
        private String name;
        private int price;
        // ...
    }
```

- OrderStatus.java

```
public class OrderStatus implements Serializable {
    private String code;
    private String name;
    // ...
}
```

SQL マッピングを実装する。

関連するオブジェクトを、1 回の SQL でまとめて取得する場合、取得したいテーブルを JOIN して、マッピングに必要なすべてのレコードを取得する。

取得したレコードは、resultMap 要素にマッピング定義を行い、JavaBean にマッピングする。

以下では、1 件の Order を取得する SQL(findOne) と、すべての Order を取得する SQL(findAll) の実装例となっている。

なお、以下の実装例では、ネームスペースに"order"を指定し、 マッピングする JavaBean は、typeAlias を使って、パッケージ名を除いたクラス名が定義してある前提となっている。

- sqlMapConfig.xml

```
<typeAlias alias="Order" type="xxxxxx.yyyyyy.zzzzzz.domain.model.Order"/>
<typeAlias alias="OrderStatus" type="xxxxxx.yyyyyy.zzzzzz.domain.model.OrderStatus"/>
<typeAlias alias="OrderItem" type="xxxxxx.yyyyyy.zzzzzz.domain.model.OrderItem"/>
<typeAlias alias="OrderCoupon" type="xxxxxx.yyyyyy.zzzzzz.domain.model.OrderCoupon"/>
<typeAlias alias="Item" type="xxxxxx.yyyyyy.zzzzzz.domain.model.Item"/>
<typeAlias alias="Category" type="xxxxxx.yyyyyy.zzzzzz.domain.model.Category"/>
<typeAlias alias="Coupon" type="xxxxxx.yyyyyy.zzzzzz.domain.model.Coupon"/>
```

- order-sqlmap.xml

```
<sqlMap namespace="order">

    <!-- ... -->

</sqlMap>
```

まず、SQL について説明する。

実際の定義は、resultMap 要素の後に、定義すること。

SQL の実装

- findOne/findAll 共通部分の SQL 定義 (sql 要素)

```
    <sql id="fragment_selectFormJoin">          <!-- (1) -->
        SELECT                                    /* (2) */
            o.id
            ,os.code AS status_code
            ,os.name AS status_name
            ,ol.quantity
            ,i.code AS item_code
            ,i.name AS item_name
            ,i.price AS item_price
            ,ct.code AS category_code
            ,ct.name AS category_name
            ,cp.code AS coupon_code
            ,cp.name AS coupon_name
            ,cp.price AS coupon_price
        FROM
            t_order o
        INNER JOIN                                /* (3) */
            c_order_status os
                ON os.code = o.status_code
        INNER JOIN
            t_orderline ol
                ON ol.order_id = o.id
        INNER JOIN
            m_item i
                ON i.code = ol.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
        LEFT JOIN                                 /* (4) */
            t_order_coupon oc
                ON oc.order_id = o.id
        LEFT JOIN
            m_coupon cp
                ON cp.code = oc.coupon_code
    </sql>
```

項番	説明
(1)	findOne と、findAll で SELECT 句、FROM 句、JOIN 句を共有するための sql 要素。findOne と findAll で、多くの共通部分があったので共通化している。
(2)	関連オブジェクトを生成するために、必要なデータをすべて取得する。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 句で別名を指定している。
(3)	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。
(4)	データ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_group_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_group_coupon と結合する t_coupon も同様である。

- findOne の SQL 定義

```
    <select id="findOne" parameterClass="java.lang.Integer" resultMap="orderResultMap"> <!-- (1)
        <include refid="fragment_selectFormJoin"/> <!-- (2) -->
        WHERE
            o.id = #id#          /* (3) */
        ORDER BY                 /* (4) */
            item_code ASC        /* (5) */
            ,category_code ASC   /* (6) */
            ,coupon_code ASC     /* (7) */
    </select>
```

項番	説明
(1)	指定された注文 ID の、Order オブジェクトおよび関連オブジェクトを取得するための SQL。
(2)	findAll と共有する SELECT 句、FROM 句、JOIN 句が実装された SQL を、インクルードしている。
(3)	バインド値で渡された注文 ID を、WHERE 句に指定する。
(4)	1:N の関係の関連オブジェクトがある場合は、リスト内の並び順を制御するための、ORDER BY 句を指定する。並び順を意識する必要がない場合は、指定は不要である。
(5)	Order#orderItems のリストを、t_item テーブルの code カラムの昇順にするための指定。
(6)	Item#categories のリストを、t_category テーブルの code カラムの昇順にするための指定。
(7)	Order#orderCoupons のリストを、t_coupon の code の昇順にするための指定。

- findAll の SQL 定義

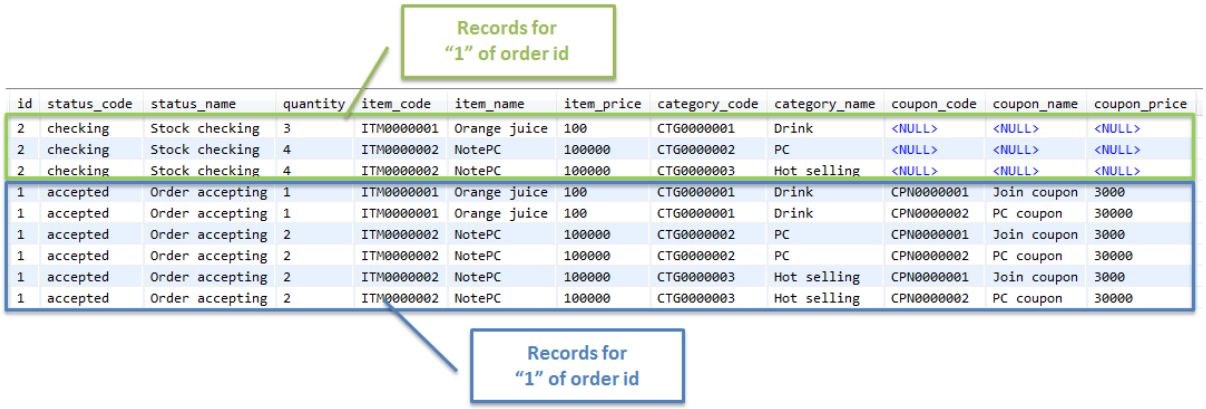
```
    <select id="findAll" resultMap="orderResultMap"> <!-- (1) -->
        <include refid="fragment_selectFormJoin"/> <!-- (2) -->
        ORDER BY
            o.id DESC      /* (3) */
            ,i.code ASC
            ,ct.code ASC
            ,cp.code ASC
    </select>
```

項番	説明
(1)	すべての Order、および、関連オブジェクトを取得するための SQL。
(2)	findOne と findAll で SELECT 句、FROM 句、JOIN 句を共有するための sql 要素。
(3)	取得されるリストの並び順を、t_order の id の降順にするための指定。

上記 SQL(findAll) を実行した結果、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。

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のレコードとなる。



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 5.17　**Picture - Result Set of findAll**

上記レコードを、`Order` オブジェクト、および、関連オブジェクトにマッピングする方法について説明する。

resultMap 要素の実装

それぞれの説明については、後述する。

```
    <resultMap id="orderResultMap" class="Order" groupBy="id">
        <result property="id" column="id" />
        <result property="status" resultMap="order.orderStatusResultMap" />
        <result property="orderItems" resultMap="order.orderItemResultMap" />
        <result property="orderCoupons" resultMap="order.orderCouponResultMap" />
    </resultMap>

    <resultMap id="orderStatusResultMap" class="OrderStatus" groupBy="code">
        <result property="code" column="status_code" />
        <result property="name" column="status_name" />
    </resultMap>

    <resultMap id="orderItemResultMap" class="OrderItem" groupBy="itemCode">
        <result property="itemCode" column="item_code" />
        <result property="item" resultMap="order.itemResultMap" />
        <result property="quantity" column="quantity" />
    </resultMap>

    <resultMap id="itemResultMap" class="Item" groupBy="code">
        <result property="code" column="item_code" />
        <result property="name" column="item_name" />
        <result property="price" column="item_price" />
        <result property="categories" resultMap="order.categoryResultMap" />
    </resultMap>

    <resultMap id="categoryResultMap" class="Category" groupBy="code">
        <result property="code" column="category_code" />
        <result property="name" column="category_name" />
    </resultMap>

    <resultMap id="orderCouponResultMap" class="OrderCoupon" groupBy="couponCode">
        <result property="couponCode" column="coupon_code" />
        <result property="coupon" resultMap="order.couponResultMap" />
    </resultMap>

    <resultMap id="couponResultMap" class="Coupon" groupBy="code">
        <result property="code" column="coupon_code" />
        <result property="name" column="coupon_name" />
        <result property="price" column="coupon_price" />
    </resultMap>
```

各 resultMap 要素の役割と依存関係を、以下に示す。

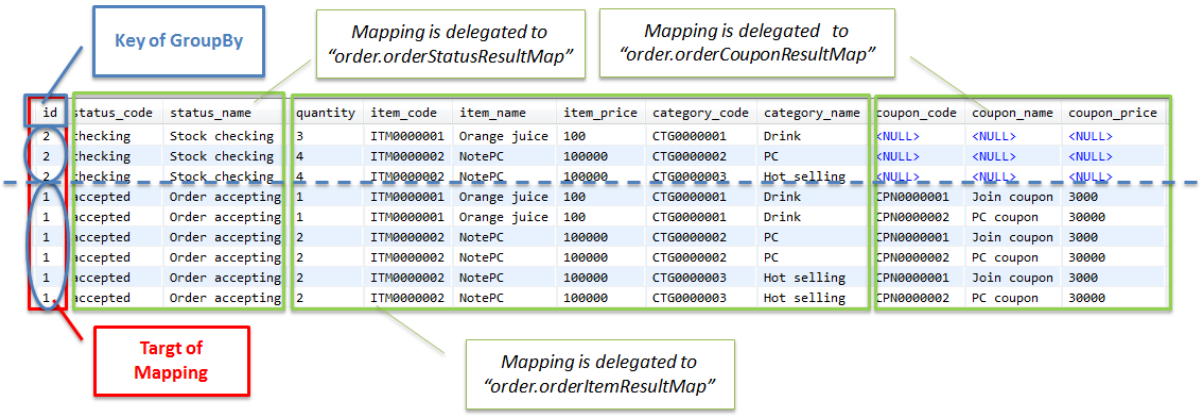


図 5.18　**Picture - Implementation of ResultMap**

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。 関連オブジェクト (OrderStatus, OrderItem, OrderCoupon) のマッピングは、別の resultMap に委譲している。
(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。 関連オブジェクト (Item) のマッピングは別の resultMap に委譲している。
(4)	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。 関連オブジェクト (Category) のマッピングは、別の resultMap に委譲している。
(5)	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。
(6)	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。 関連オブジェクト (Coupon) のマッピングは、別の resultMap に委譲している。
(7)	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。

Order オブジェクトへのマッピングを行う。

```
    <resultMap id="orderResultMap" class="Order" groupBy="id"> <!-- (1) -->
        <result property="id" column="id" /> <!-- (2) -->
        <result property="status" resultMap="order.orderStatusResultMap" /> <!-- (3) -->
        <result property="orderItems" resultMap="order.orderItemResultMap" /> <!-- (4) -->
        <result property="orderCoupons" resultMap="order.orderCouponResultMap" /> <!-- (5) -->
    </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 5.19 **Picture - ResultMap for Order**

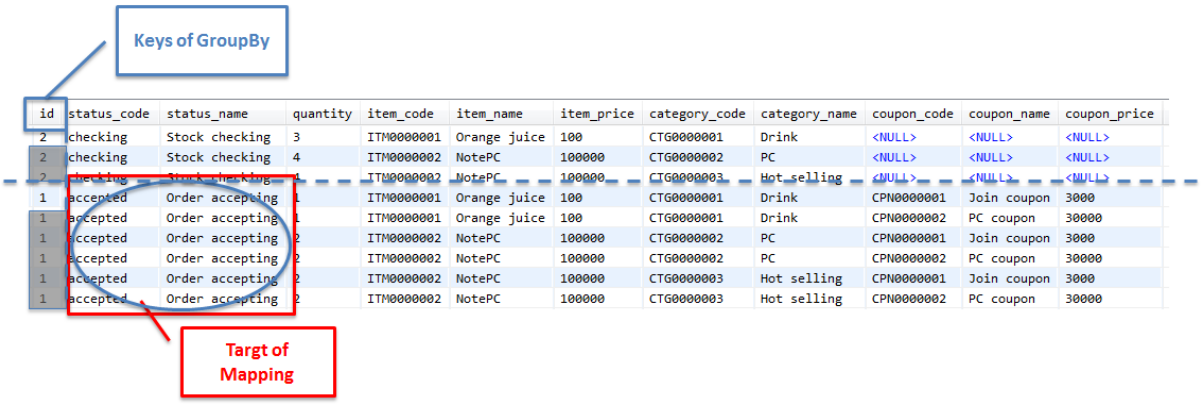
項番	説明
(1)	取得したレコードは、注文毎にグループ化する必要があるため、注文を一意に識別するための値が格納されている id プロパティを、groupBy 属性に指定する。 本例では、id プロパティでグループ化されるため、id=1 と id=2 の 2 つの Order オブジェクトが、生成される。
(2)	取得したレコードの id カラムの値を、Order#id に設定する。
(3)	OrderStatus オブジェクトの生成を、id="order.orderStatusResultMap"の resultMap に委譲し、生成されたオブジェクトを、Order#status に設定する。
(4)	OrderItem オブジェクトの生成を、id="order.orderItemResultMap"の resultMap に委譲し、生成されたオブジェクトを、Order#orderItems のリストに追加する。
(5)	OrderCoupon オブジェクトの生成を、id="order.orderCouponResultMap"の resultMap に委譲し、生成されたオブジェクトを、Order#orderCoupons のリストに追加する。

以降では、id=1 の Order オブジェクトへのマッピングに、焦点を当てて説明する。

OrderStatus オブジェクトへのマッピングを行う。

```
    <resultMap id="orderStatusResultMap" class="OrderStatus"> <!-- (1) -->
        <result property="code" column="status_code" /> <!-- (2) -->
```

```
        <result property="name" column="status_name" /> <!-- (3) -->
    </resultMap>
```



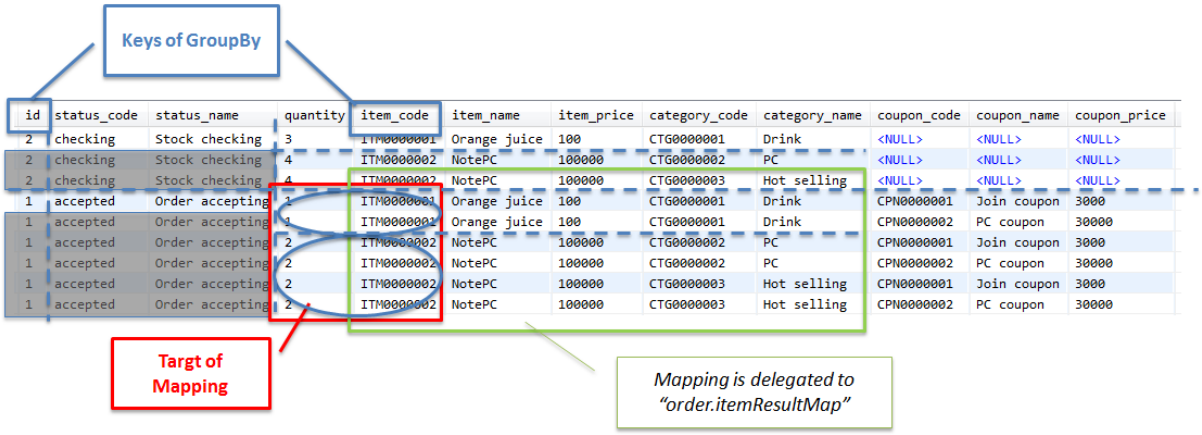
id	status_code	status_name	quantity	item_code	item_name	item_price	category_code	category_name	coupon_code	coupon_name	coupon_price
2	checking	Stock checking	3	ITM0000001	Orange juice	100	CTG0000001	Drink	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000002	PC	<NULL>	<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 5.20 **Picture - ResultMap for OrderStatus**

項番	説明
(1)	Order と、OrderStatus オブジェクトは、1:1 の関係なので、groupBy 属性の指定は不要である。 本例では、code=accepted の OrderStatus オブジェクトが生成される。
(2)	取得したレコードの、status_code カラムの値を、OrderStatus#code に設定する。
(3)	取得したレコードの、status_name カラムの値を、OrderStatus#name に設定する。

OrderItem オブジェクトへのマッピングを行う。

```
    <resultMap id="orderItemResultMap" class="OrderItem" groupBy="itemCode"> <!-- (1) -->
        <result property="itemCode" column="item_code" /> <!-- (2) -->
        <result property="item" resultMap="order.itemResultMap" /> <!-- (3) -->
        <result property="quantity" column="quantity" /> <!-- (4) -->
    </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

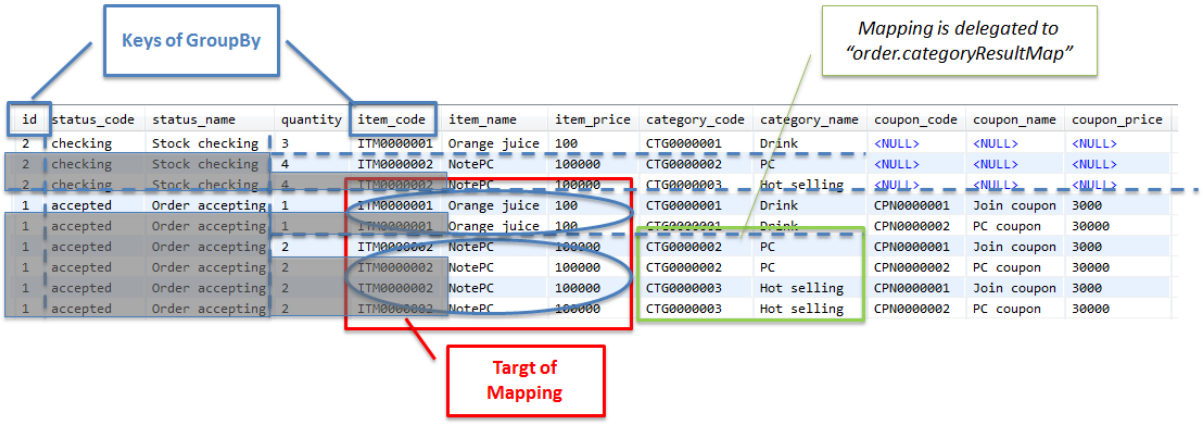
図 5.21 **Picture - ResultMap for OrderItem**

項番	説明
(1)	Order と OrderItem は、1:N の関係なので、groupBy 属性の指定が必要である。 注文商品は、t_order_item のプライマリキー (order_id,item_code) でグループ化する必要があるが、order_id カラムについては、親の resultMap で指定されているため、ここでは、item_code カラムの値を保持する itemCode プロパティのみ指定する。 本例では、itemCode プロパティでグループ化されるため、itemCode=ITM0000001 と itemCode=ITM0000002 の、２つの OrderItem オブジェクトが生成される。
(2)	取得したレコードの item_code カラムの値を、OrderItem#itemCode に設定する。 (3) で生成される Item#code と重複するが、itemCode プロパティは、OrderItem をグループ化するために必要なプロパティとなる。
(3)	Item オブジェクトの生成を、id="order.itemResultMap"の resultMap に委譲し、生成されたオブジェクトを OrderItem#item に設定する。
(4)	取得したレコードの quantity カラムの値を、OrderItem#quantity に設定する。

Item オブジェクトへのマッピングを行う。

```
    <resultMap id="itemResultMap" class="Item" groupBy="code"> <!-- (1) -->
        <result property="code" column="item_code" /> <!-- (2) -->
        <result property="name" column="item_name" /> <!-- (3) -->
        <result property="price" column="item_price" /> <!-- (4) -->
```

```
        <result property="categories" resultMap="order.categoryResultMap" /> <!-- (5) -->
    </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 5.22　**Picture - ResultMap for Item**

項番	説明
(1)	OrderItem と Item オブジェクトは、1:1 の関係だが、Item と Category は、1:N の関係なので、groupBy 属性の指定が必要である。 カテゴリは商品毎にグループ化する必要があるため、商品を一意に識別するための値が格納されている code プロパティを、groupBy 属性に指定する。 本例では、OrderItem#itemCode=ITM0000001 用に、 code=ITM0000001 の Item オブジェクトが、OrderItem#itemCode=ITM0000002 用に、code=ITM0000002 の Item オブジェクトが生成される。(計 2 つのオブジェクトが生成される。)
(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。
(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。
(5)	Category オブジェクトの生成を、id="order.categoryResultMap"の resultMap に委譲し、生成されたオブジェクトを Item#categories のリストに追加する。

Category オブジェクトへのマッピングを行う。

```
    <resultMap id="categoryResultMap" class="Category" groupBy="code"> <!-- (1) -->
        <result property="code" column="category_code" /> <!-- (1) -->
        <result property="name" column="category_name" /> <!-- (1) -->
    </resultMap>
```

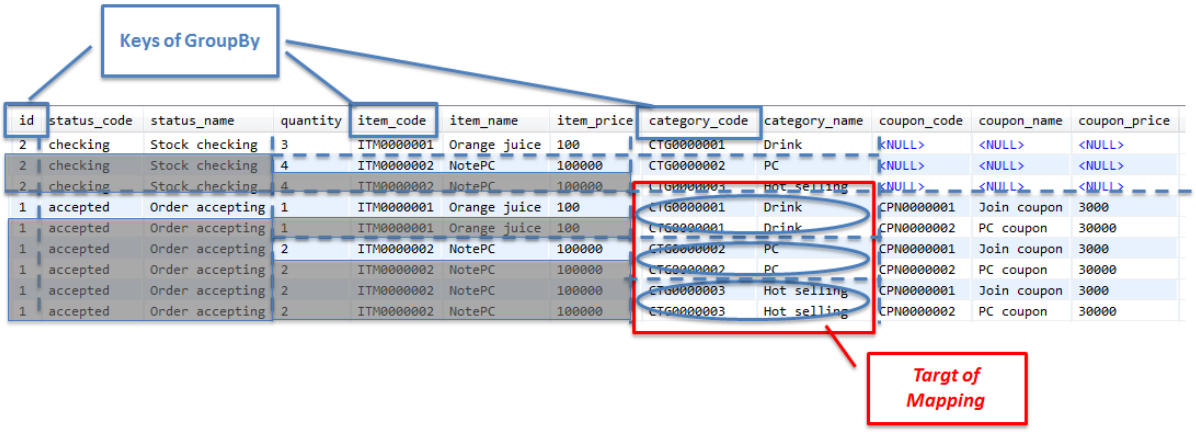


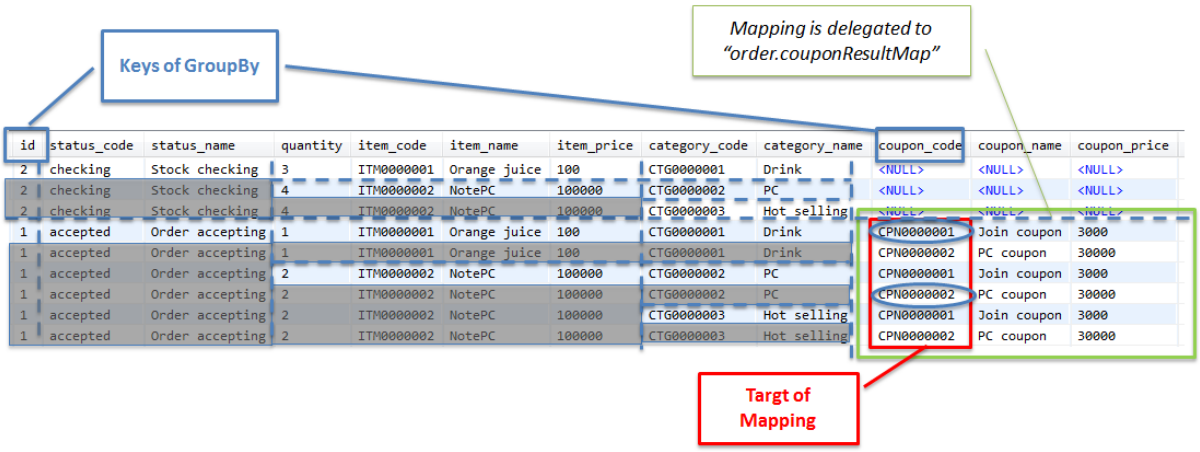
図 5.23 **Picture - ResultMap for Item**

項番	説明
(1)	本例では、1:N の関係のテーブル (t_order と t_order_line、t_order と t_order_coupon) を複数結合しているため、 t_order_coupon に複数レコードが格納されていると `Item` オブジェクト内に保持する `Category` オブジェクトのリストが、重複してしまう。 重複をなくすために、カテゴリを一意に識別するための値が格納されている `code` プロパティを、groupBy 属性に指定する。`code` プロパティの値が、同じ `Category` オブジェクトが一つにマージされ、重複をなくすことができる。 本例では、`Item#code=ITM0000001` 用に、`code=CTG0000001` の `Category` オブジェクトが、`Item#code=ITM0000002` 用に、`code=CTG0000002` と、`code=CTG0000003` の 2 つの `Category` オブジェクトが生成される。(計 3 つのオブジェクトが生成される。)
(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。
(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。

`OrderCoupon` オブジェクトへのマッピングを行う。

```
    <resultMap id="orderCouponResultMap" class="OrderCoupon" groupBy="couponCode"> <!-- (1) -->
        <result property="couponCode" column="coupon_code" /> <!-- (2) -->
        <result property="coupon" resultMap="order.couponResultMap" /> <!-- (3) -->
```

```
    </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 5.24　**Picture - ResultMap for OrderCoupon**

項番	説明
(1)	Order と OrderCoupon は、1:N の関係なので、groupBy 属性の指定が必要である。 注文クーポンは、t_order_coupon のプライマリキー (order_id,coupon_code) でグループ化する必要があるが、order_id カラムについては親の resultMap で指定されているため、ここでは、coupon_code カラムの値を保持する couponCode プロパティのみ指定する。 本例では、couponCode プロパティでグループ化されるため、couponCode=CPN0000001 と couponCode=CPN0000002 、の 2 つの OrderCoupon オブジェクトが生成される。
(2)	取得したレコードの coupon_code カラムの値を、OrderCoupon#couponCode に設定する。 (3) で生成される Coupon#code と重複するが、couponCode プロパティは、OrderCoupon をグループ化するために必要なプロパティとなる。
(3)	Coupon オブジェクトの生成を id="order.couponResultMap"の resultMap に委譲し、生成されたオブジェクトを OrderCoupon#coupon に設定する。

Coupon オブジェクトへのマッピングを行う。

```
    <resultMap id="couponResultMap" class="Coupon"> <!-- (1) -->
        <result property="code" column="coupon_code" /> <!-- (2) -->
        <result property="name" column="coupon_name" /> <!-- (3) -->
        <result property="price" column="coupon_price" /> <!-- (4) -->
```

```
    </resultMap>
```

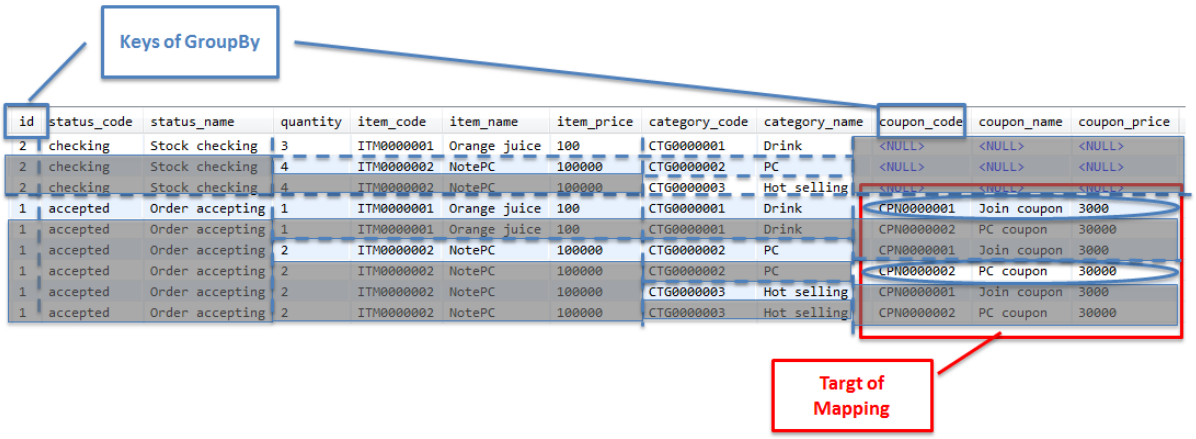
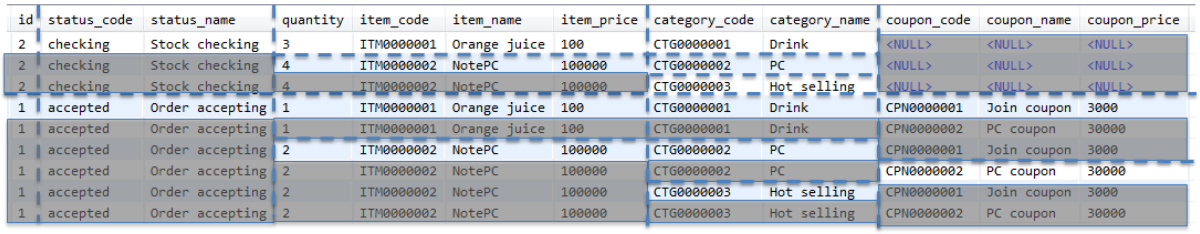


図 5.25　**Picture - ResultMap for Coupon**

項番	説明
(1)	`OrderCoupon` と `Coupon` オブジェクトは、1:1 の関係なので、groupBy 属性の指定が不要である。 本例では、`OrderCoupon#couponCode=CPN0000001` 用に、`code=CPN0000001` の `Coupon` オブジェクトが、`OrderCoupon#couponCode=CPN0000001` 用に、`code=CPN0000001` の `Coupon` オブジェクトが生成される。(計 2 つのオブジェクトが生成される。)
(2)	取得したレコードの `coupon_code` カラムの値を、`Coupon#code` に設定する。
(3)	取得したレコードの `coupon_name` カラムの値を、`Coupon#name` に設定する。
(4)	取得したレコードの `coupon_price` カラムの値を、`Coupon#price` に設定する。

JavaBean 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、groupBy 属性に指定によって、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 5.26　**Picture - Valid Result Set for result mapping**

> **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。

実際にマッピングされた `Order` オブジェクトおよび関連オブジェクトの状態は、以下の通りである。

ちなみに: 関連オブジェクトを取得する別の方法として、取得したレコードの値を使って、内部で別の SQL を実行して、取得する方法がある。内部で別の SQL を実行する方法は、個々の SQL や、resultMap 要素の定義が、非常にシンプルとなる。ただし、この方法で取得する場合は、N+1 問題を引き起こす要因となることを、意識しておく必要がある。

内部で別の SQL を実行する方法については、Mybatis Developer Guide(PDF) の「Result Maps/Complex Properties」(P.36-37) および「Result Maps/Composite Keys or Multiple Complex Parameters Properties」(P.40-41) を参照されたい。

ちなみに: 内部で別の SQL を実行する方法を使う場合、関連オブジェクトは “Eager Load” されるため、関連オブジェクトを使用しない場合も、SQL が実行されてしまう。この動作回避する方法として、Mybatis では、関連オブジェクトを “Lazy Load” する方法を、オプションとして提供している。

“Lazy Load” を有効にするための設定は、以下の通りである。

- Mybatis 設定ファイルの setting 要素の enhancementEnabled 属性を、`true` に設定する。
- CGLIB 2.x を、クラスパスに追加する。

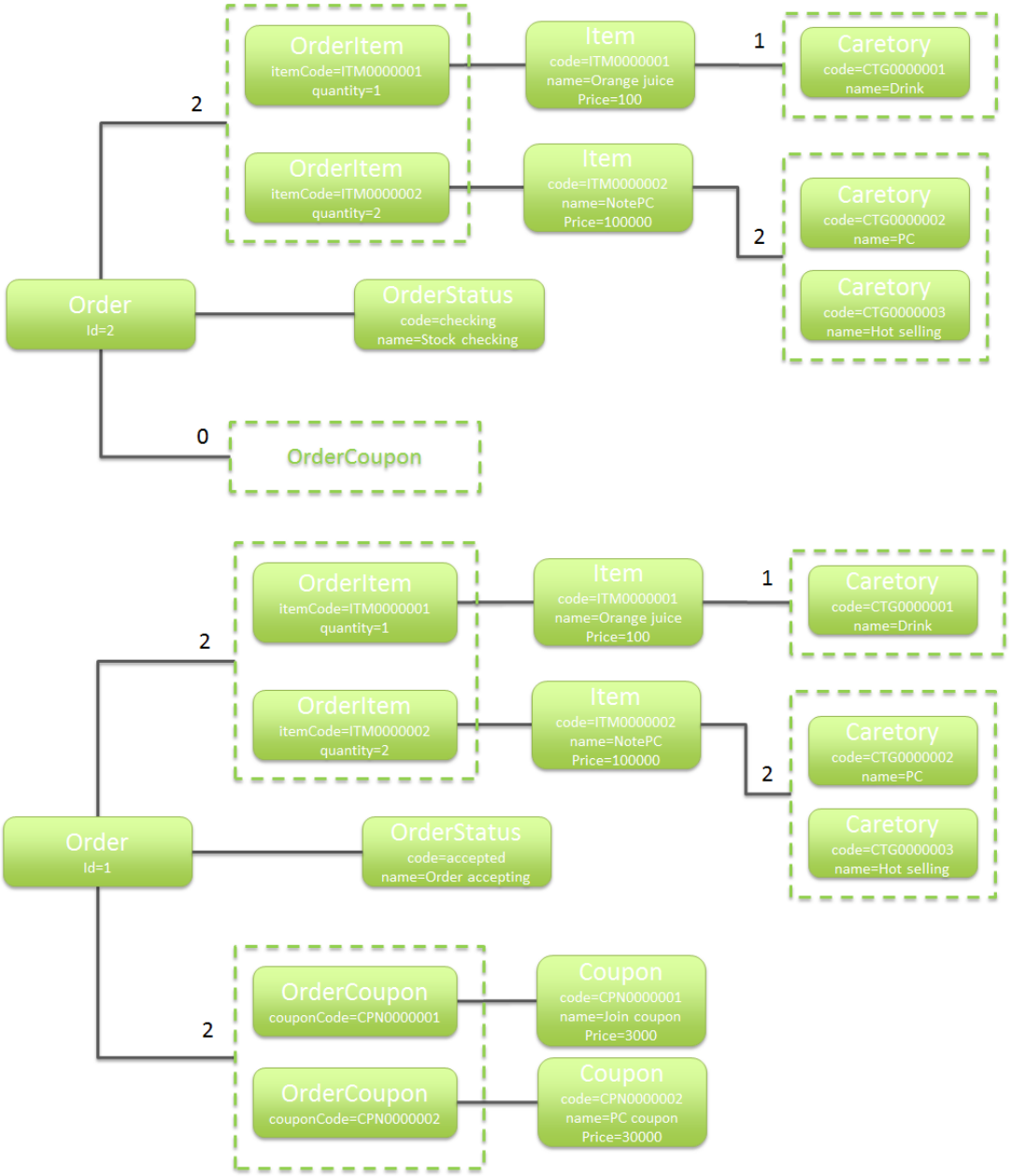


図 5.27　**Picture - Mapped object diagram**

5.4 排他制御

5.4.1 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

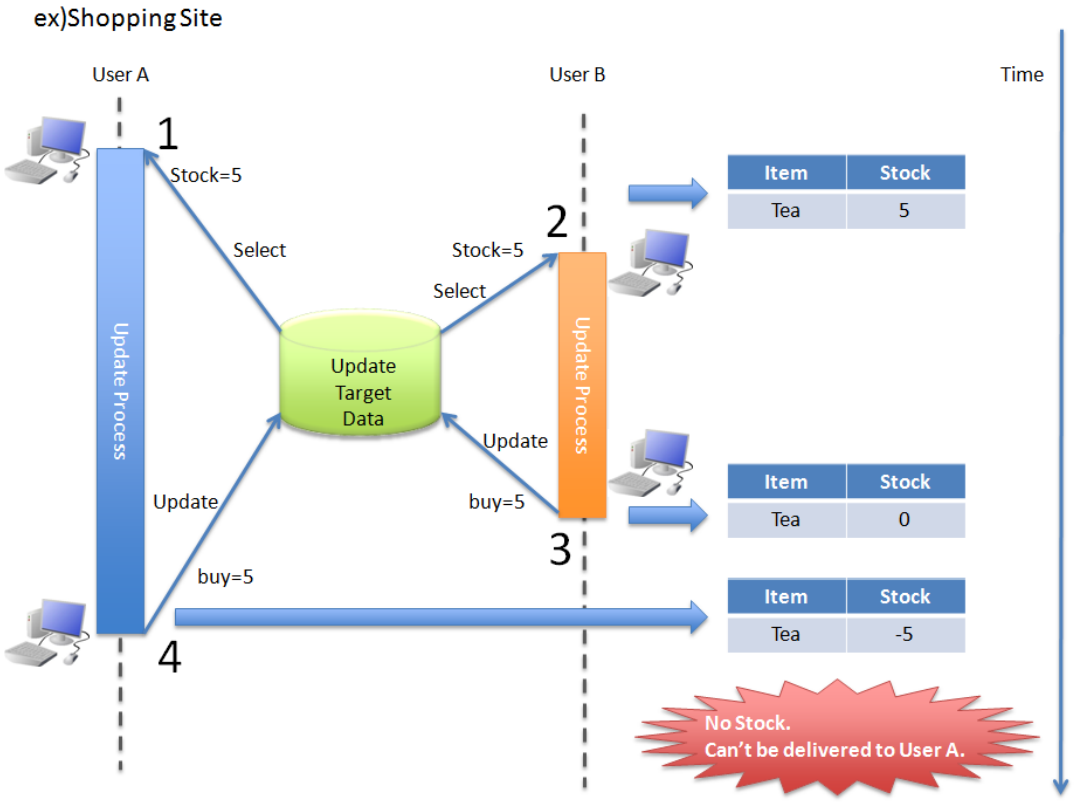
まず、排他制御の必要性を理解してもらうために、排他制御を行わなかった際に発生する問題について、具体例を 3 つ挙げて説明する。

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。



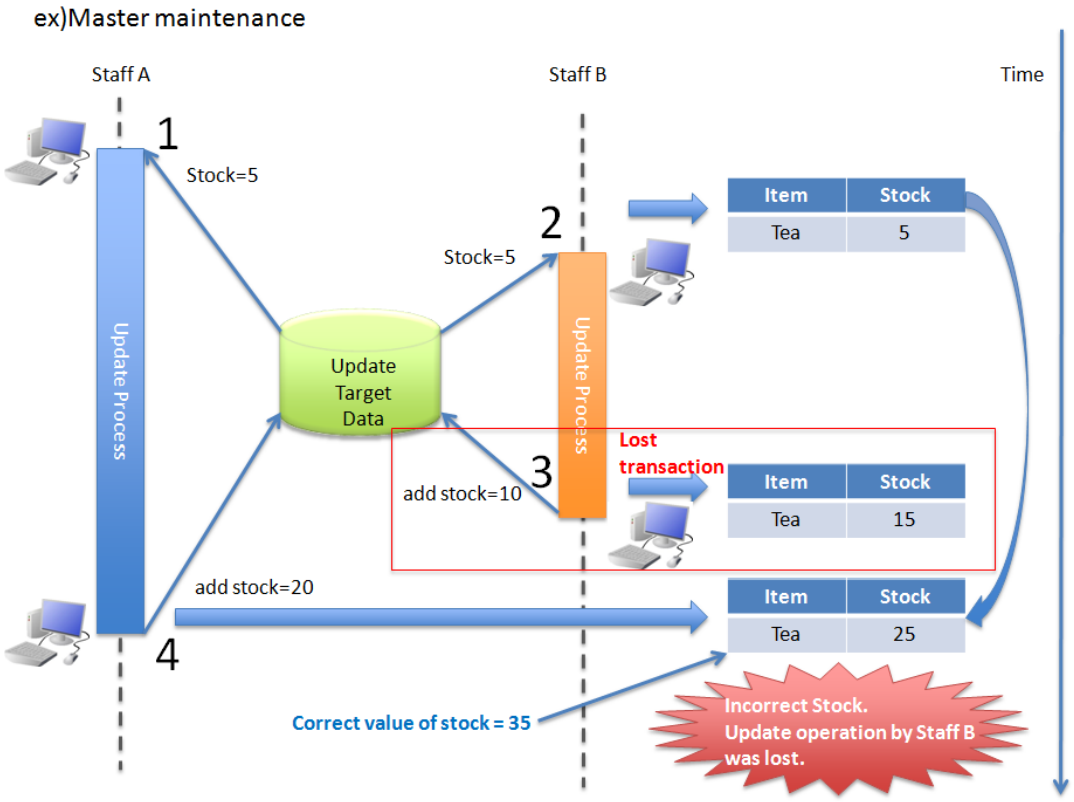
テーブルで管理している Tea の在庫数についても、実際の Tea の在庫数と異なる値 (マイナス値) になってしまう。

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。

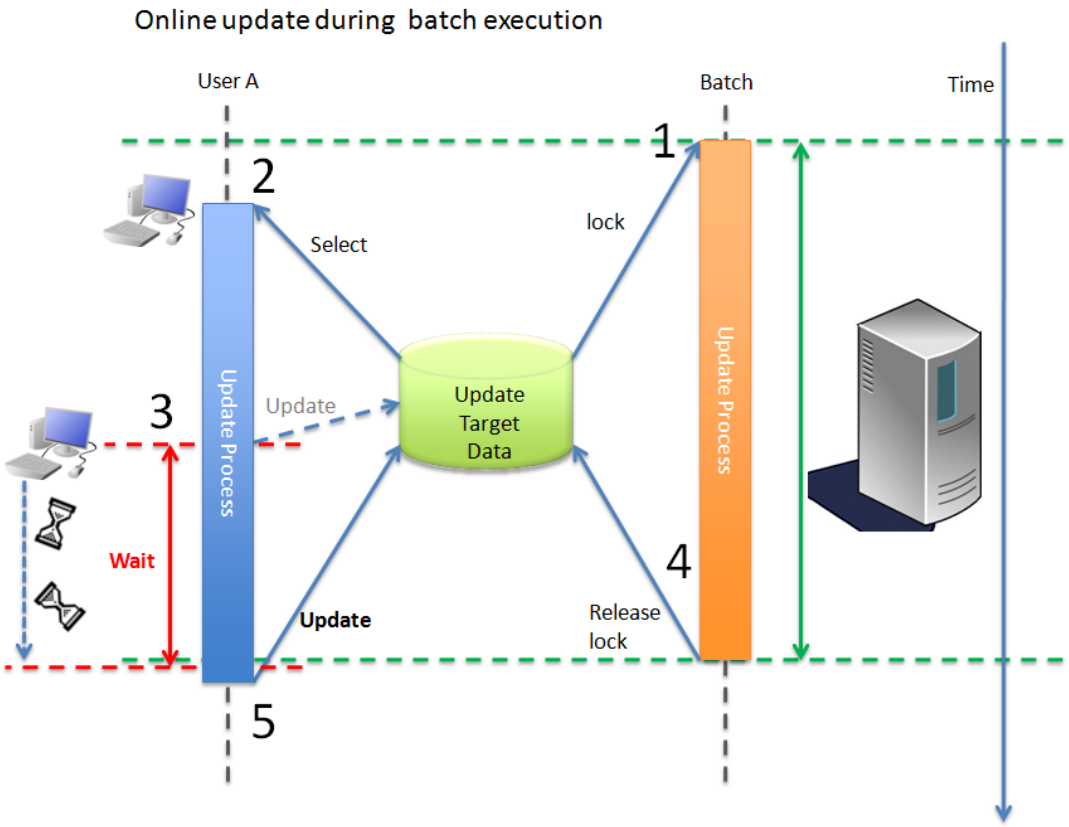


Problem3

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。

TABLE 5.11　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。

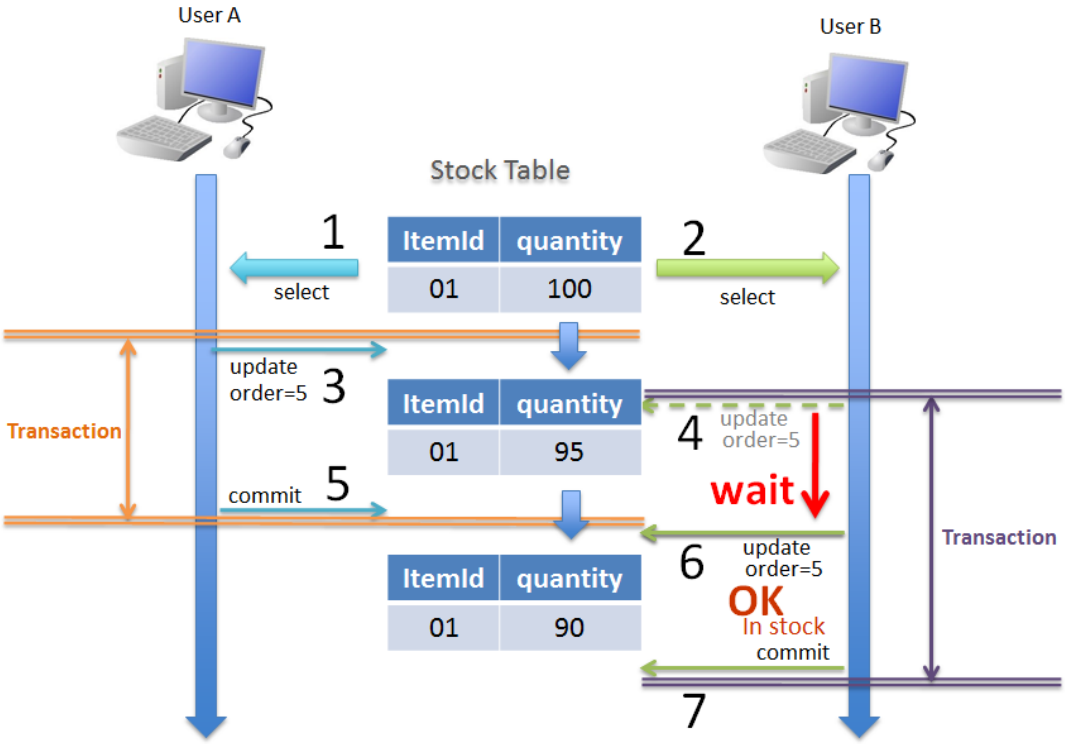
データベースの行ロック機能による排他制御は、他のトランザクションによって更新した内容を確認する必要がない場合に使用することができる。

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。



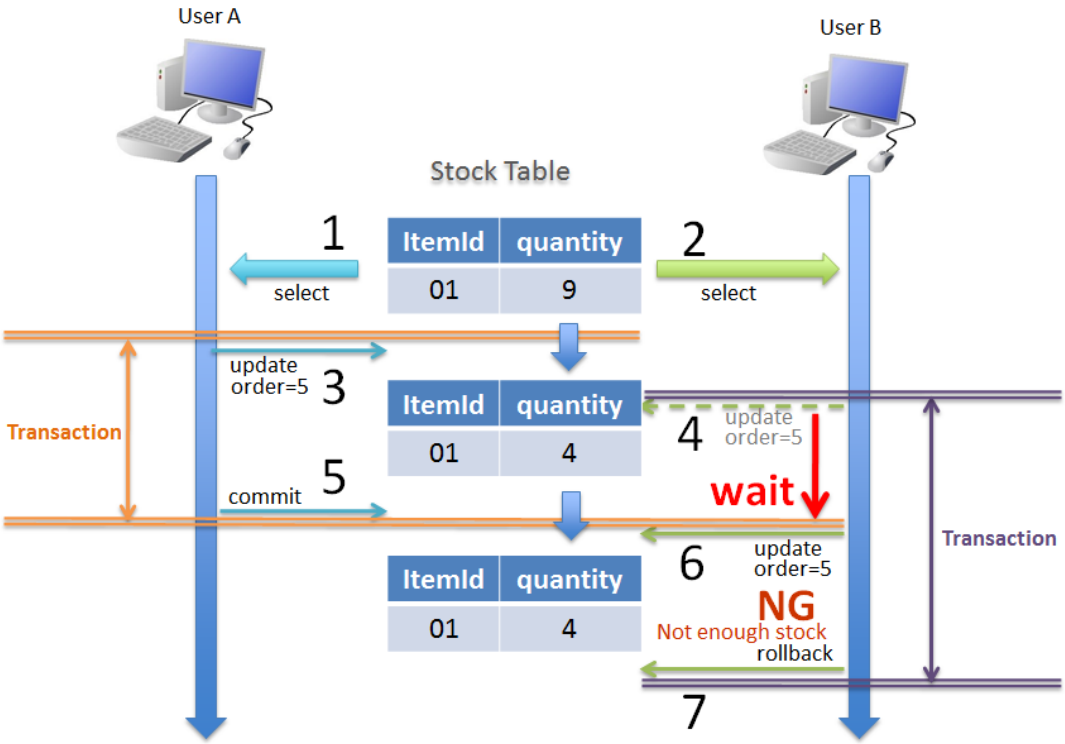
項番	UserA	UserB	説明
1.	○	-	User A が、商品の購入画面を表示する。在庫数が 100 個で画面に表示されている。 `select quantity from Stock where ItemId = '01'`
2.	-	○	User B が、商品の購入画面を表示する。在庫数が 100 個で画面に表示されている。 `select quantity from Stock where ItemId = '01'`
3.	○	-	User A が、ItemId=01 の商品を 5 個購入する。Stock Table から個数を-5 する。 `Update from Stock 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。

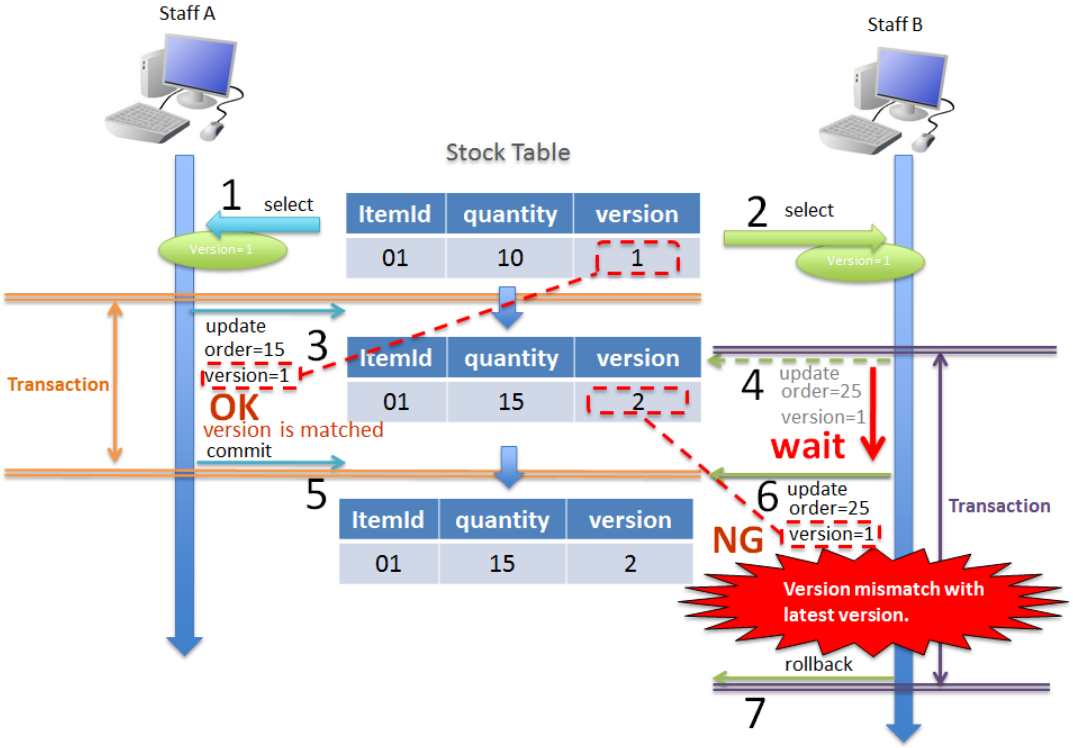
例えば、ワークフローアプリケーションにおいて、申請者と承認者が同時に操作（引き戻しと承認）を行った場合を想像してほしい。

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。

TABLE 5.12　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。

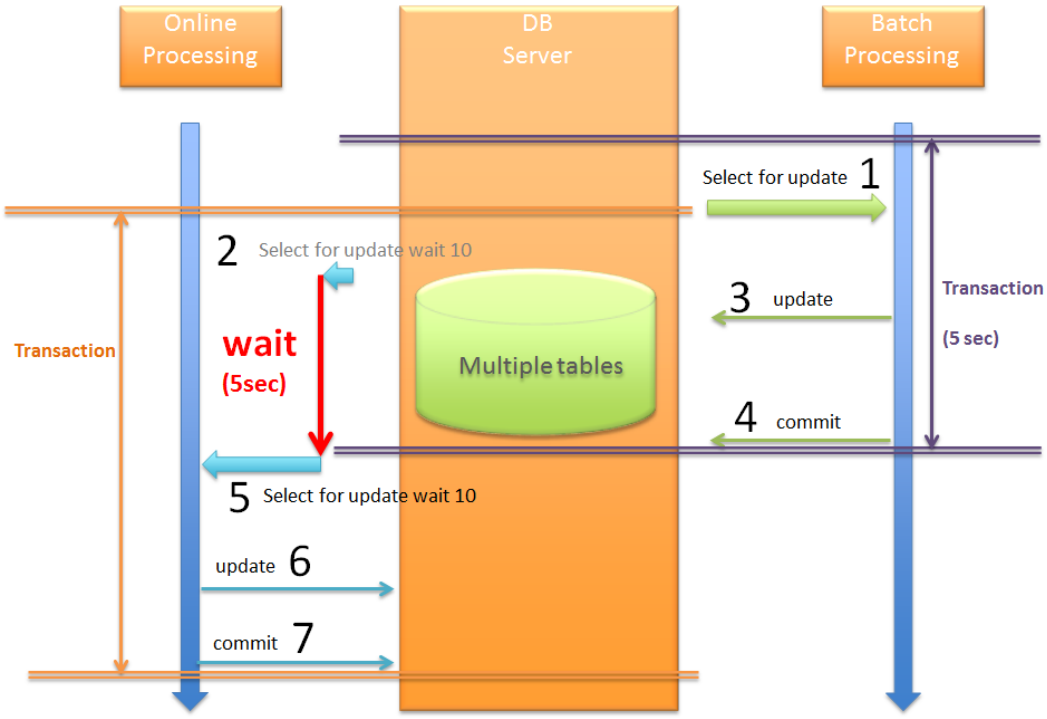
3. バッチ実行中にオンラインの処理が実行されることがある。

   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。

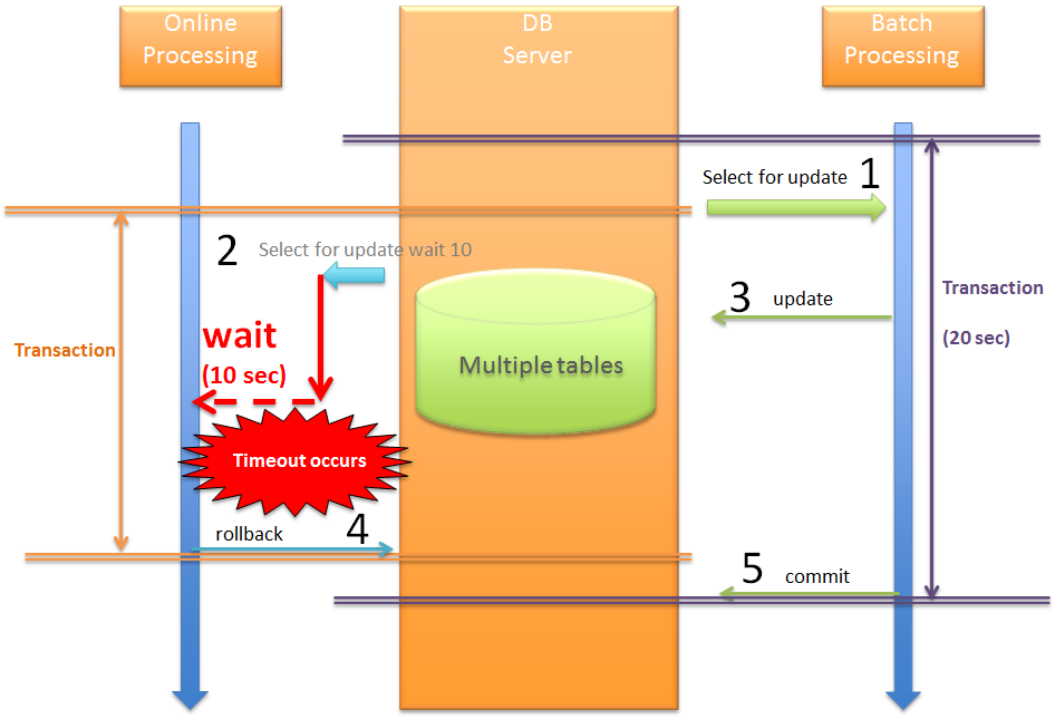
- バッチ処理が既に実行済みで、オンラインで更新するデータを悲観ロックしている。
- オンライン処理は 10 秒のタイムアウト時間を指定して、更新対象のデータのロックを取得する。
- バッチ処理は 5 秒後 (タイムアウト前) に終了する。



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。

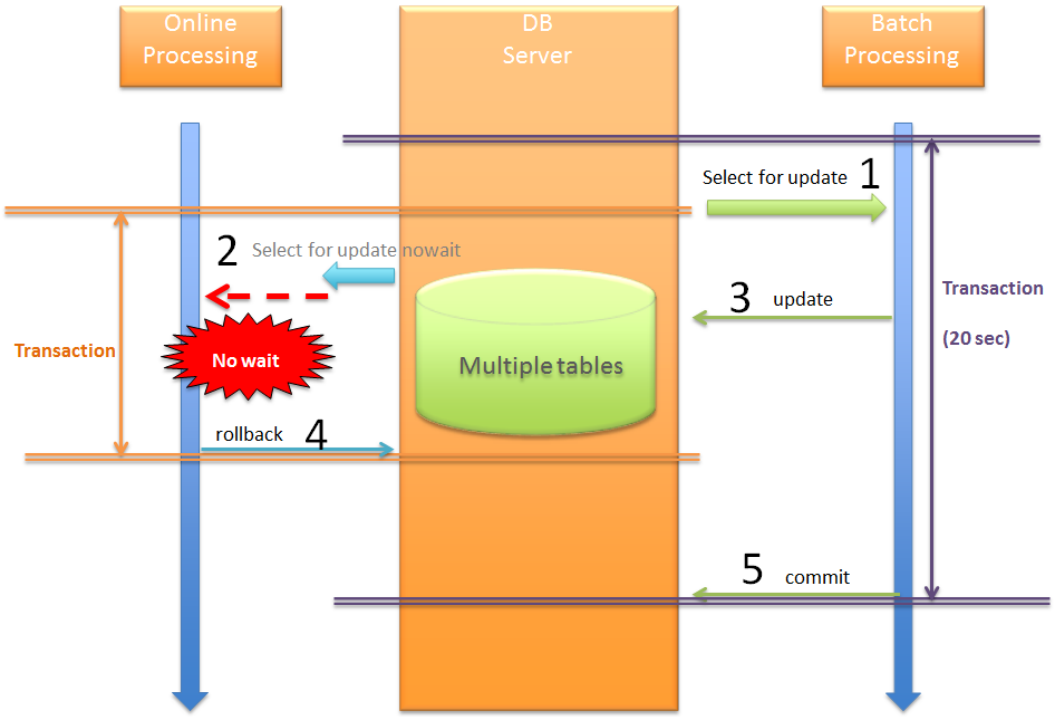
悲観ロックの解放を待つことなく、すぐに排他エラーとなる。

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

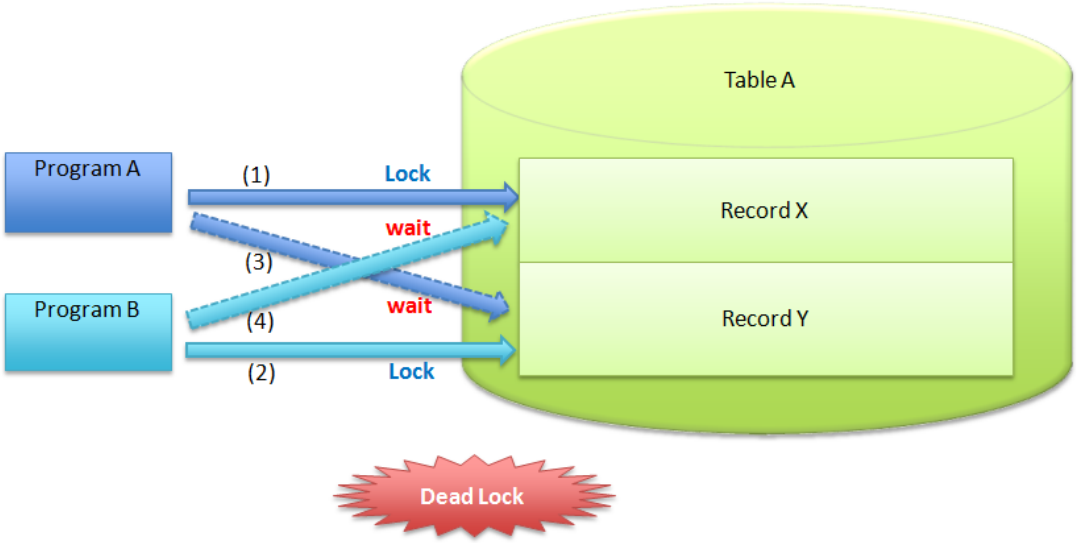
データベースのロック機能を使用する場合、同一トランザクション内で複数のレコードを更新すると、以下 2 通りの、デッドロックが発生する可能性があるため、注意する必要がある。

- 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、お互いが保持しているロックの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。

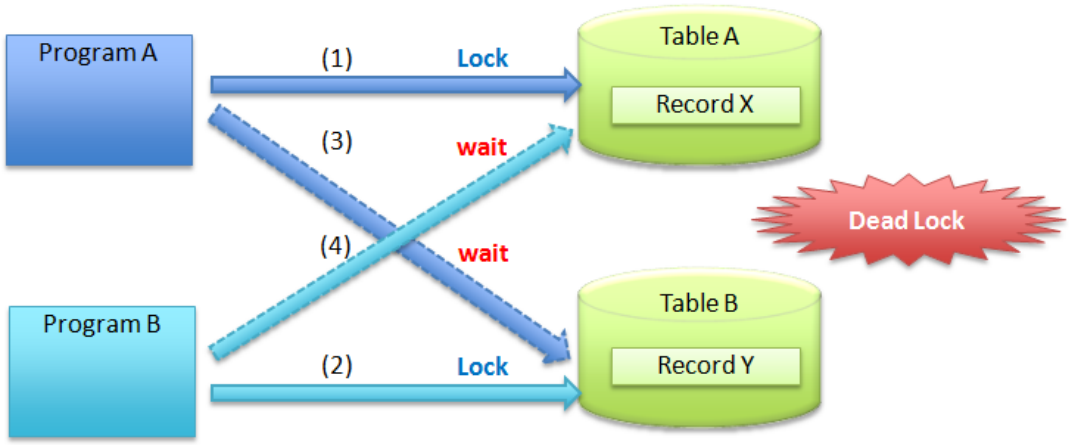
注釈: デッドロックの解決方法について

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、お互いが保持しているロックの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。

5.4.2 How to use

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への値の設定について

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 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。

Mybatis 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、sqlmap ファイルに、SQL 定義を追加して実現する。

- xxx-sqlmap.xml

```
<update id="decrementQuantity" parameterClass="OrderItem">

    UPDATE m_stock SET
        quantity = quantity - #quantity#
    WHERE item_code = #itemCode#
    AND #quantity# <![CDATA[ <= ]]> quantity <!-- (1) -->

</update>
```

項番	説明
(1)	sqlmap ファイルに、在庫数が注文数以上ある場合に、在庫数を減らす SQL を指定する。

- Repository(RepositoryImpl)

```
public interface StockRepository {
    int decrementQuantity(String itemCode, int quantity);
}
```

```
public class StockRepositoryImpl implements StockRepository {

    public int decrementQuantity(String itemCode, int quantity) { // (2)
        OrderItem orderItem = new OrderItem();
        orderItem.setItemCode(itemCode);
        orderItem.setQuantity(quantity);
        return updateDAO.execute("stock.decrementQuantity", orderItem); // (3)
    }

}
```

項番	説明
(2)	Repository にメソッドを追加する。
(3)	SQL 実行に必要なパラメータを生成し、SQL を呼び出す。

注釈: Repository を作成しない場合

Repository を作成しない場合は、上記処理は Service で実装することになる。

- Service

```
    String itemCodeOfOrder = "ITM0000001";
    int quantityOfOrder = 31;

    int updateCount = stockRepository.decrementQuantity(itemCodeOfOrder, quantityOfOrder); // (4
    if (updateCount == 0) { // (5)
        ResultMessages message = ResultMessages.error();
        message.add(ResultMessage
                .fromText("Not enough stock. Please, change quantity."));
        throw new BusinessException(message); // (6)
    }
```

項番	説明
(4)	Query メソッドを呼び出す。
(5)	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。
(6)	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。

楽観ロック

Mybatis 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

```
    public class Stock implements Serializable {

        private String itemCode;
        private int quantity;
        private long version; // (1)

        // ...

    }
```

項番	説明
(1)	バージョン管理用のプロパティを用意する。

- xxx-sqlmap.xml

```
<resultMap id="stockResultMap" class="Stock">
    <result property="itemCode" column="item_code" />
    <result property="quantity" column="quantity" />
    <result property="version" column="version" /> <!-- (2) -->
</resultMap>

<select id="findOne" parameterClass="java.lang.String" resultMap="stockResultMap">
    SELECT * FROM m_stock WHERE item_code = #itemCode#
</select>

<update id="update" parameterClass="Stock">
    UPDATE m_stock SET
        quantity = quantity
        ,version = version + 1    <!-- (3) -->
    WHERE item_code = #itemCode#
    AND version = #version#       <!-- (4) -->
</update>
```

項番	説明
(2)	更新対象のデータを取得する SQL にて、バージョン管理用のカラムに設定されている値を取得する。
(3)	更新する際は、バージョンをインクリメントする。
(4)	更新条件として、バージョンが一致することを加える。

- Repository(RepositoryImpl)

```
public interface StockRepository {
    Stock findOne(String itemCode);
    Stock save(Stock stock);
}
```

```
public class StockRepositoryImpl implements StockRepository {

    public Stock findOne(String itemCode) {
```

```
            return queryDAO.executeForObject("stock.findOne", itemCode, Stock.class);
        }

        public Stock save(Stock stock) {
            if(exists(stock.getItemCode())){
                int updateCount = updateDAO.execute("stock.update", stock);
                if(updateCount == 0){
                    throw new ObjectOptimisticLockingFailureException(Stock.class, itemCode); //
                }
            } else {
                updateDAO.execute("stock.insert", stock);
            }
            return stock;
        }

    }
```

項番	説明
(5)	更新結果が 0 件の場合、他のトランザクションによって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。

注釈: Repository を作成しない場合

Repository を作成しない場合は、上記処理は Service で実装することになる。

- 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)

    stock = stockRepository.save(stock); // (4)
```

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。

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して楽観ロックを行う場合は、更新時のチェックに加えて、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。

- 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);

stock = stockRepository.save(stock);
```

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。
(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。

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 SQL 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 SQL 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。

- `xxx-sqlmap.xml`

```
<update id="decrementQuantity" parameterClass="OrderItem">

    UPDATE m_stock SET
        quantity = quantity - #quantity#
        ,version = version + 1 <!-- (1) -->
    WHERE item_code = #itemCode#
    AND #quantity# <![CDATA[ <= ]]> quantity

</update>
```

項番	説明
(1)	Version の更新 (`version = version + 1`) を行う必要がある。

悲観ロック

Mybatis 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。

- xxx-sqlmap.xml

```
<select id="findOneForUpdate" parameterClass="java.lang.String" resultMap="stockResultMap">
    SELECT * FROM m_stock
    WHERE item_code = #itemCode#
    FOR UPDATE <!-- (1) -->
</select>
```

項番	説明
(1)	悲観ロックが必要な 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。 キーワードは、データベースによって異なる。

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 String handleOptimisticLockingFailureException(
        OptimisticLockingFailureException e) {
    // (2)
    ExtendedModelMap modelMap = new ExtendedModelMap();
    ResultMessages resultMessages = ResultMessages.warn();
    resultMessages.add(ResultMessage.fromText("Other user updated!!"));
    modelMap.addAttribute(setUpForm());
    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 の処理メソッドの中で、`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 String handlePessimisticLockingFailureException(
        PessimisticLockingFailureException e) {
    // (2)
    ExtendedModelMap modelMap = new ExtendedModelMap();
    ResultMessages resultMessages = ResultMessages.warn();
    resultMessages.add(ResultMessage.fromText("Other user updated!!"));
    modelMap.addAttribute(setUpForm());
    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 の処理メソッドの中で、`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。

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。

- 悲観ロックに失敗した場合、PessimisticLockingFailureException ではなく、org.springframework.dao.UncategorizedDataAccessException の子クラスが発生する。

悲観エラー時に発生する UncategorizedDataAccessException は、システムエラーに分類される例外なので、アプリケーションでハンドリングすることは推奨されないが、最悪ハンドリングを行う必要があるかもしれない。原因例外には、悲観ロックエラーが発生したことを通知する例外が格納されているので、ハンドリングができる。

⇒継続調査。

現状以下の動作となる。

- PostgreSQL + for update nowait
  - org.springframework.orm.hibernate3.HibernateJdbcException
  - Caused by: org.hibernate.PessimisticLockException
- Oracle + for update
  - org.springframework.orm.hibernate3.HibernateSystemException
  - Caused by: Caused by: org.hibernate.dialect.lock.PessimisticEntityLockException

- Caused by: org.hibernate.exception.LockTimeoutException

5.5 入力チェック

5.5.1 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 EMail 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。

課題

クライアントサイドの入力チェックについては今後追記する。初版では、サーバーサイドの入力チェックのみ言及する。

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。

5.5.2 How to use

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 Email 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 RFC2822 準拠の 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 RFC2822 準拠の制限よりも緩い場合は、@Email を使用せず、javax.validation.constraints.Pattern を用いて、正規表現を指定する必要がある。
(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) */ re
    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。

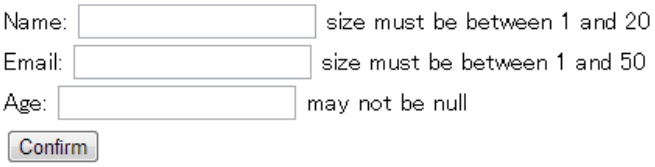
```
<!DOCTYPE html>
<html>
<%-- WEB-INF/views/user/createForm.jsp --%>
<body>
    <form:form modelAttribute="userForm" method="post"
        action="${pageContext.request.contextPath}/user/create">
        <form:label path="name">Name:</form:label>
        <form:input path="name" />
        <form:errors 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。



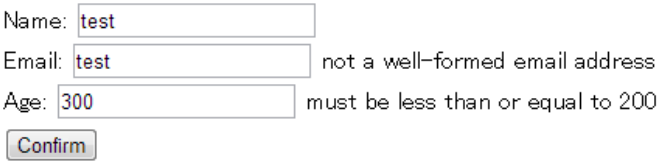
Name と Email が空文字であることに対するエラーメッセージと、Ag が `null` であることに対するエラーメッセージが表示されている。

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。

Email の入力値は文字列長に関する条件は満たすが、Email 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>
```

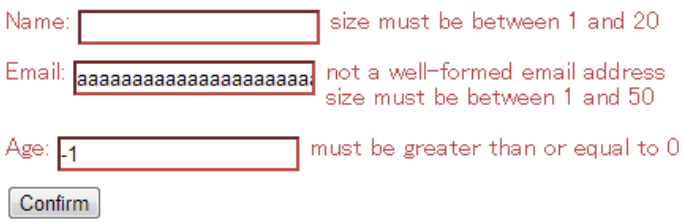
項番	説明
(1)	エラー時に<label>タグへ加えるクラス名を、cssErrorClass 属性で指定する。
(2)	エラー時に<input>タグへ加えるクラス名を、cssErrorClass 属性で指定する。
(3)	エラーメッセージに加えるクラス名を、cssClass 属性で指定する。

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。

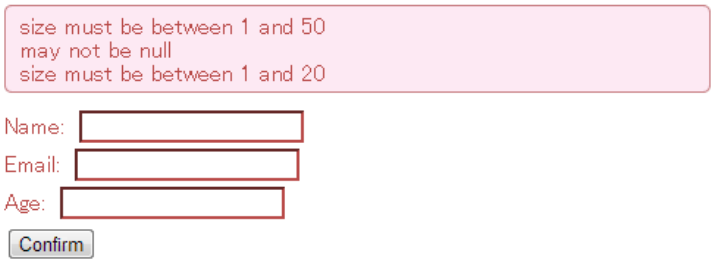
エラーメッセージを、入力フィールドの横に一件一件出力する代わりに、まとめて出力することもできる。

```
<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>
```

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。

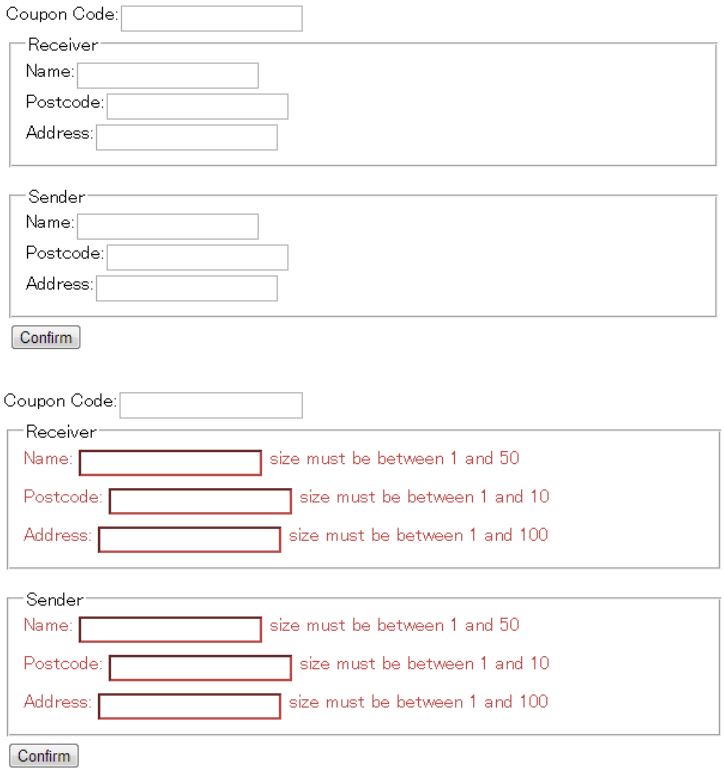
フォームは、以下のように表示される。

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 type="text/javascript"
            src="${pageContext.request.contextPath}/resources/app/js/AddressesView.js"></script>
    </body>
    </html>
```

項番	説明
(1)	address 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

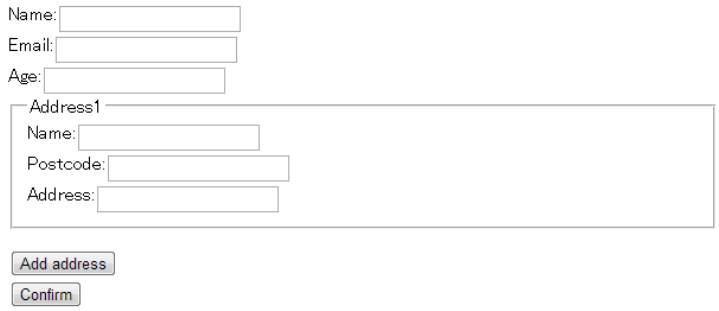
動的にアドレス入力フィールドを追加するための JavaScript も記載するが、このコードの説明は、本質

的ではないため割愛する。

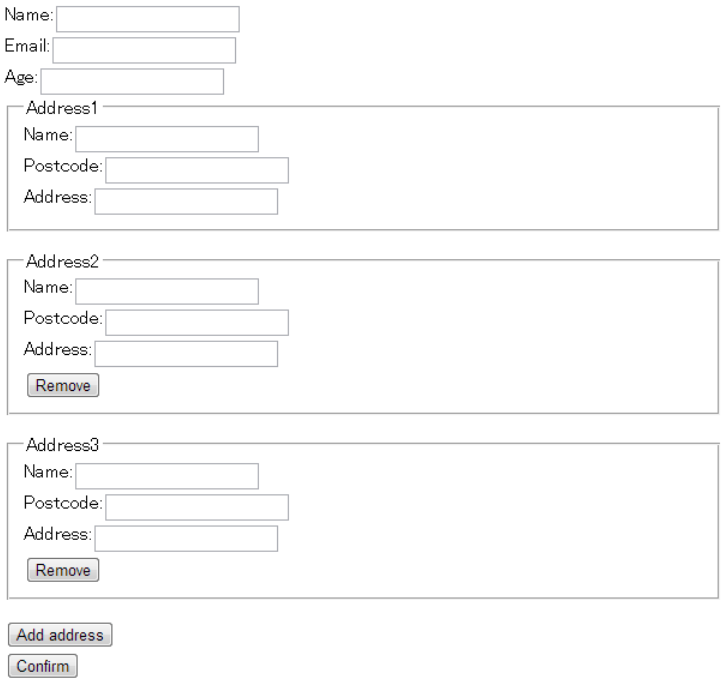
```
// webapp/resources/app/js/AddressesView.js

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="tex
    <label for="addresses' + number + '.postcode">Postcode:</label>\
    <input id="addresses' + number + '.postcode" name="addresses[' + number + '].postcode" t
    <label for="addresses' + number + '.address">Address:</label>\
    <input id="addresses' + number + '.address" name="addresses[' + number + '].address" typ
    <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。

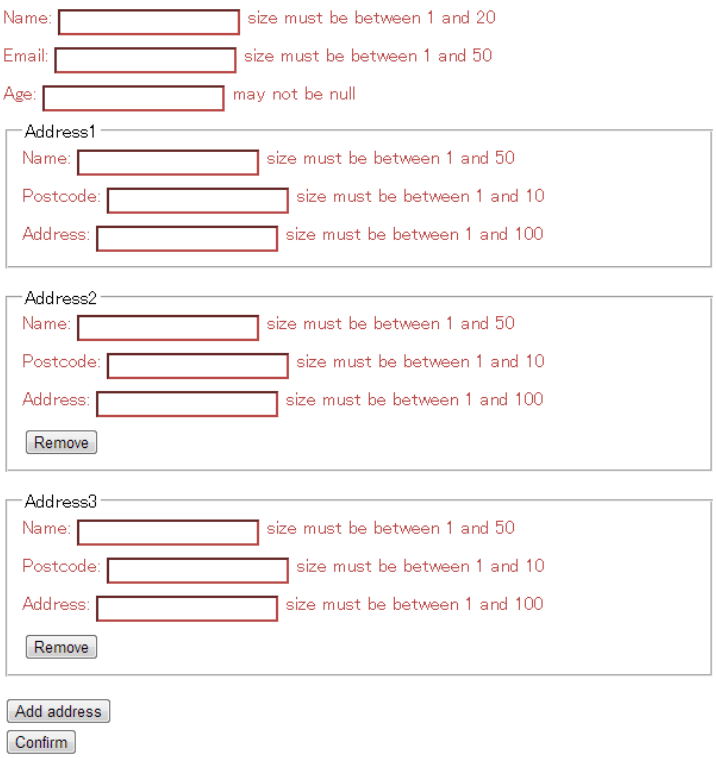


このフォームに対して、すべての入力フィールドを未入力のまま送信すると、以下のようにエラーメッセージが表示される。

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)	各グループごとにルールを定義し、グループを指定するために、`group` 属性に対象のグループクラスを指定する。 `group` 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)	age フィールドの@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” のいづ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(200)
        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)	age フィールドの@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 の処理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 `passwordConfirm` 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";
    }
}
```

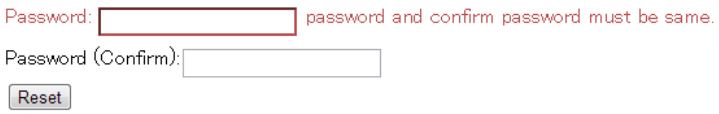
項番	説明
(1)	使用する Spring Validator を、インジェクションする。
(2)	`@InitBinder` アノテーションがついたメソッド内で、`WebDataBinder.addValidators` メソッドにより、Validator を追加する。 これにより、`@Validated` アノテーションでバリデーションをする際に、追加した Validator も呼び出される。
(3)	入力チェックの実装は、これまで通りである。

- 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: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: 一つの Controller で複数のフォームを扱う場合は、Validator の対象を限定するために、@InitBinder("xxx") でモデル名を指定する必要がある。

```
@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 SmartValidato

    @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'
        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

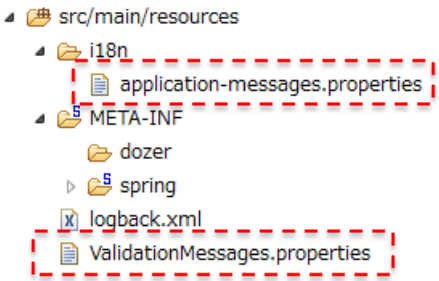
基本的にエラーメッセージは、properties ファイルに定義することを推奨する。

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。

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
}
```

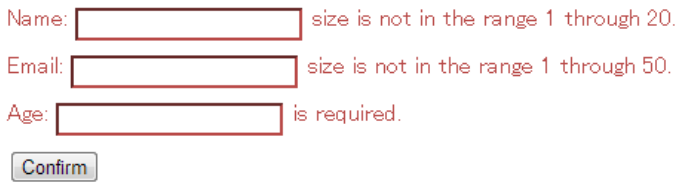
- ValidationMessages.properties

  `@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}.
# (2)
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。
(2)	不正となった入力値は、{value}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。

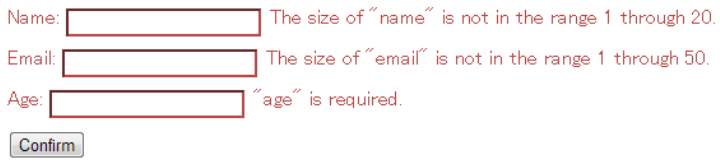
エラーメッセージに、フィールド名を含める場合は、以下のように、メッセージに`{0}`を加える。

- 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} throug
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
```

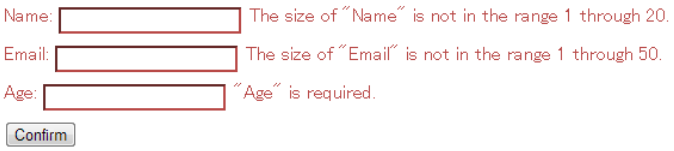
形式でフィールド名を定義すればよい。

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むのは、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。

application-messages.properties に定義するメッセージ

ValidationMessages.properties でシステムで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

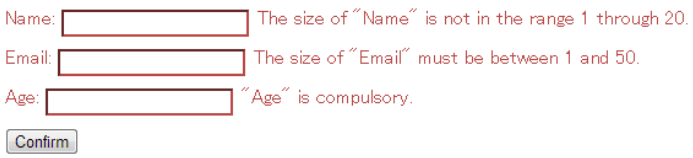
ValidationMessages.properties **に定義するメッセージ**の設定がある前提で、以下の設定で age フィールドのメッセージを上書きする。

- application-messages.properties

```
# override messages
NotNull.userForm.age="{0}" is compulsory.
Max.userForm.age="{0}" must be less than or equal to {1}.
Max.userForm.age="{0}" must be less than or equal to {1}.
NotNull.userForm.email="{0}" is compulsory.
Size.userForm.age=The size of "{0}" must be between {2} and {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。

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。

5.5.3 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
    public @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 = {})
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @ReportAsSingleViolation // (1)
    @Pattern(regexp = "[a-zA-Z0-9]*") // (2)
    public @interface AlphaNumeric {
        String message() default "{com.example.common.validation.AlphaNumeric.message}"; // (3)

        Class<?>[] groups() default {};

        Class<? extends Payload>[] payload() default {};

        @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
        @Retention(RUNTIME)
        @Documented
        public @interface List {
            AlphaNumeric[] value();
        }
    }
```

項番	説明
(1)	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。
(2)	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。
(3)	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@NotNegaitive 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 NotNegaitive {
    String message() default "{com.example.common.validation.NotNegaitive.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    public @interface List {
        NotNegaitive[] 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
    public @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
        public @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。

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
    public @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_Num
        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。

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
```

```
        public @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)
```

```
                        .addNode(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.addNode` 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、以下の JavaDoc を参照されたい。

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
    public @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。

5.5.4 Appendix

Hibernate Validator が用意する入力チェックルール

Hibernate Validator はで定義されたアノテーションに加え、検証できるアノテーションを追加している。
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を、以下に示す。

詳細は、Bean Validation specification の 6 章を参照されたい。

アノテーション (javax.validation.*)	対象の型	用途	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)	@Nulll(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 継承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,CharSequence 継承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,CharSequence 継承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;

@DecimalMin	BigDecimal, BigInteger, String, byte, short, int, long およびラッパー (Hibernate Validator 実装では任意の	Decimal 型の値が最小値以上であるかどうかを検証する。	@DecimalMax 参照

Hibernate Validator のチェックルール

Hibernate Validator の代表的なアノテーションを、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 (org.hibernate.validator.constraints.*)	対象の型	用途	使用例
`@CreditCardNumber`	任意の CharSequence 継承クラスに適用可能	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。	@CreditCardNumber private String cardNumber;
`@Email`	任意の CharSequence 継承クラスに適用可能	RFC2822 に準拠した Email アドレスかどうか検証する。	@Email private String email;
`@URL`	任意の CharSequence 継承クラスに適用可能	RFC2396 に準拠しているかどうか検証する。	@URL private String url;
`@NotBlank`	任意の CharSequence 継承クラスに適用可能	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、arrays、任意の CharSequence 継承クラスに適用可能	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;

Hibernate Validator が用意するデフォルトメッセージ

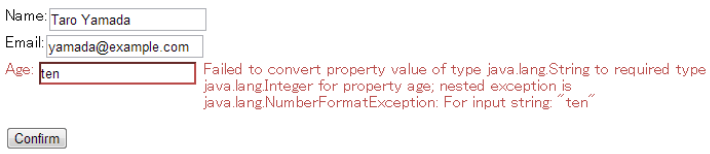
hibernate-validator-<version>.jar 内の org/hibernate/validator に、ValidationMessages.properties のデフォルト値が定義されている。

```
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 or equal to {value}
javax.validation.constraints.DecimalMin.message  = must be greater than or equal to {value}
javax.validation.constraints.Digits.message      = numeric value out of bounds (<{integer} digits>
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.Email.message            = not a well-formed email address
org.hibernate.validator.constraints.Length.message           = length must be between {min} and {m
org.hibernate.validator.constraints.NotBlank.message         = may not be empty
org.hibernate.validator.constraints.NotEmpty.message         = may not be empty
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
org.hibernate.validator.constraints.URL.message              = must be a valid URL
org.hibernate.validator.constraints.br.CNPJ.message          = invalid Brazilian corporate taxpaye
org.hibernate.validator.constraints.br.CPF.message           = invalid Brazilian individual taxpay
org.hibernate.validator.constraints.br.TituloEleitor.message = invalid Brazilian Voter ID card nu
```

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、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。

```
@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。

5.6 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。

5.6.1 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	開発時のデバック (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。
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

デバックログ、アクセスログ、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 C
date:2013-09-06 19:36:31    X-Track:85a437108e9f4a959fd227f07f72ca20        message:[END CON
date:2013-09-06 19:36:31    X-Track:85a437108e9f4a959fd227f07f72ca20        message:[HANDLIN
```

```
    date:2013-09-06 19:36:33    X-Track:948c8b9fd04944b78ad8aa9e24d9f263        message:[START C
    date:2013-09-06 19:36:33    X-Track:142ff9674efd486cbd1e293e5aa53a78        message:[START C
    date:2013-09-06 19:36:33    X-Track:142ff9674efd486cbd1e293e5aa53a78        message:[END CON
    date:2013-09-06 19:36:33    X-Track:142ff9674efd486cbd1e293e5aa53a78        message:[HANDLIN
    date:2013-09-06 19:36:33    X-Track:948c8b9fd04944b78ad8aa9e24d9f263        message:[END CON
    date:2013-09-06 19:36:33    X-Track:948c8b9fd04944b78ad8aa9e24d9f263        message:[HANDLIN
```

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、SpringMVC の Controller の処理メソッドの処理時間を、Controller の処理メソッド実行後に、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	開発時にデバック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。
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。

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。

5.6.2 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、公式マニュアルを参照されたい。

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.grovy
2.「1」のファイルが見つからない場合、クラスパス上の logback-text.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、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

```
<!DOCTYPE logback>
<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}\t
        </encoder>
    </appender>

    <appender name="APPLICATION_LOG_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>log/projectName-application.log</file> <!-- (4) -->
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>log/projectName-application-%d{yyyyMMddHH}.log</fileNamePattern> <!-
            <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}\t
        </encoder>
    </appender>

    <appender name="MONITORING_LOG_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>log/projectName-monitoring.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>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\t
        </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="fals
        <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>

    <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。
(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。

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ォ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、公式マニュアルを参照されたい。

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);

    @RequestMapping(value = "/", method = { RequestMethod.GET,
            RequestMethod.POST })
    public String home(Model model) {
        logger.trace("This log is trace log.");
        logger.debug("This log is debug log.");
        logger.info("This log is info log.");
        logger.warn("This log is warn log.");
        logger.error("This log is error log.");
        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    leve
date:2013-11-06 20:13:05    thread:tomcat-http--3 X-Track:5844f073b7434b67a875cb85b131e686    leve
date:2013-11-06 20:13:05    thread:tomcat-http--3 X-Track:5844f073b7434b67a875cb85b131e686    leve
date:2013-11-06 20:13:05    thread:tomcat-http--3 X-Track:5844f073b7434b67a875cb85b131e686    leve
```

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    le
date:2013-11-06 20:32:45    thread:tomcat-http--3   X-Track:853aa701a401404a87342a574c69efbc    le
```

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) {
    // ommited

    try {
        throwException();
    } catch (Exception e) {
        logger.error("Exception happend!", e);
        // ommited
    }
    // ommited
}

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    le
java.lang.Exception: Test Exception!
    at com.example.sample.app.welcome.HomeController.throwException(HomeController.java:40) ~[Home
    at com.example.sample.app.welcome.HomeController.home(HomeController.java:31) ~[HomeControlle
    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);
            // ommited
        }
```

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());
        }
```

5.6.3 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}\
    </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.UserIdMD`

を提供している。

Filter で独自の値を MDC を追加したい場合は
`org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter` の実装を参考に
`AbstractMDCPutFilter` を実装すればよい。

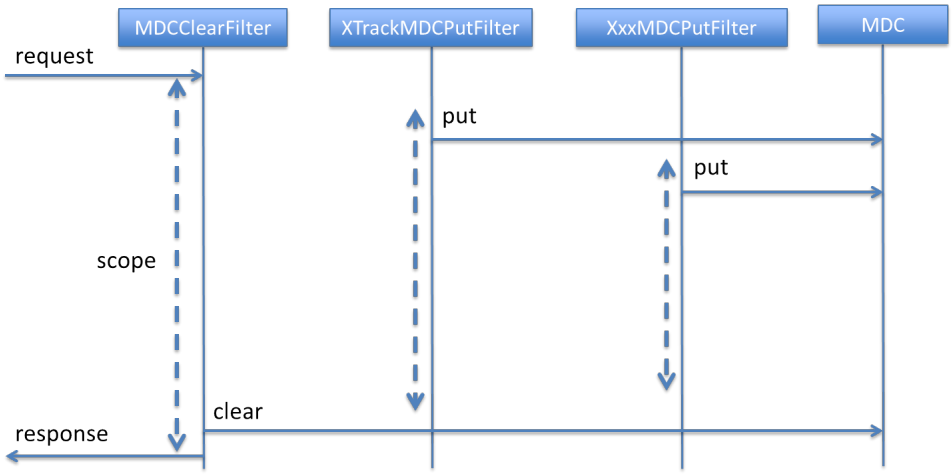
MDCFilter の使用方法

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>log/projectName-application.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>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
```

```
    </encoder>
</appender>
<!-- omitted -->
```

ログの出力例

```
date:2013-09-06 23:05:22  thread:tomcat-http--3   USER:   X-Track:97988cc077f94f9d9d435f6f7602742
date:2013-09-06 23:05:22  thread:tomcat-http--3   USER:anonymousUser  X-Track:97988cc077f94f9d9d4
date:2013-09-06 23:05:22  thread:tomcat-http--3   USER:anonymousUser  X-Track:97988cc077f94f9d9d4
date:2013-09-06 23:05:22  thread:tomcat-http--3   USER:anonymousUser  X-Track:97988cc077f94f9d9d4
date:2013-09-06 23:05:22  thread:tomcat-http--3   USER:anonymousUser  X-Track:97988cc077f94f9d9d4
```

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 auto-config="true" use-expressions="true">
        <!-- omitted -->
        <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/> <!-- (1) -->
        <!-- omitted -->
    </sec:http>

    <!-- (2) -->
    <bean id="userIdMDCPutFilter" class="org.terasoluna.gfw.security.web.logging.UserIdMDCPut
    </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.fw.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-ins
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-ap
    version="3.0">
    <listener>
        <listener-class>org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener</listener-c
    </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:c004ddb56a3642d5bc5f6b5d884e5c
```

@SessionAttribute 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。

```
<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="#{10 * 1000 * 1000 * 1000}" />
        </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。

5.7 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。

5.7.1 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。

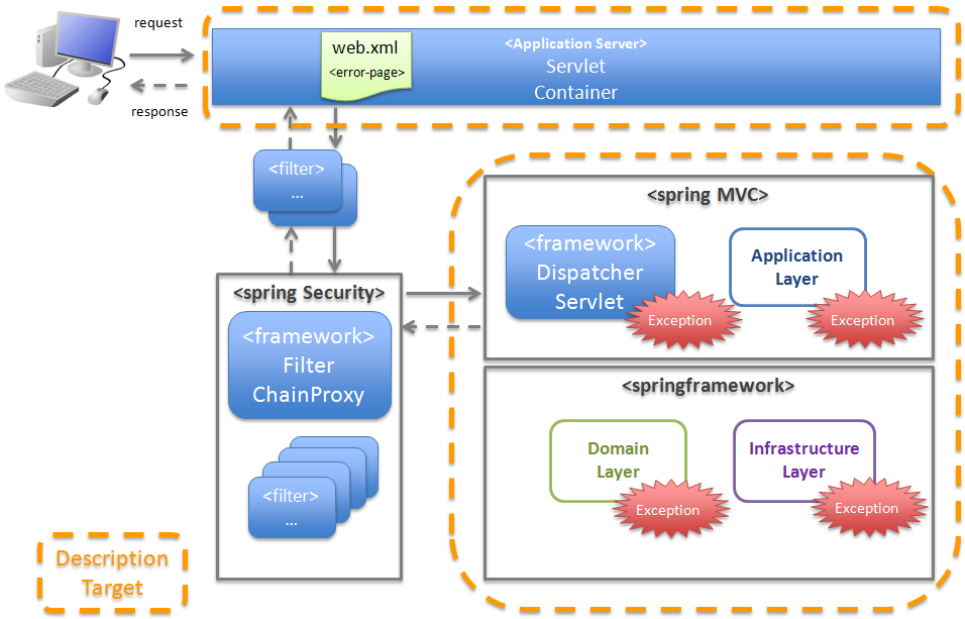


図 5.28　**図-説明対象**

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。

TABLE 5.13 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。

TABLE 5.14 **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 HanlderExceptionResolver 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。 HanlderExceptionResolver 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**
(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**

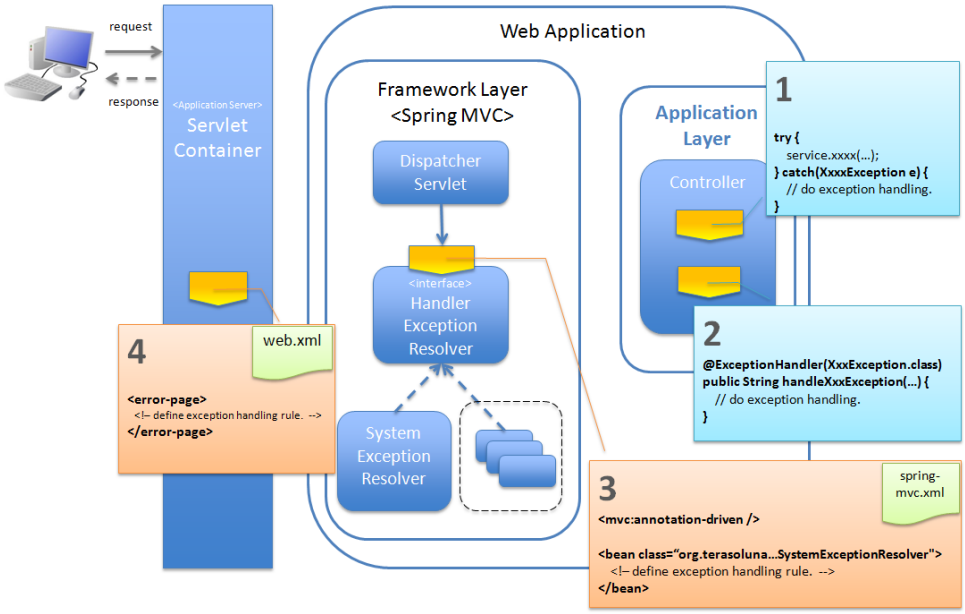


図 5.29　**図-例外のハンドリング方法**

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 HanlderExceptionResolver について

<mvc:annotation-driven> を指定した際に、自動的に登録される HanlderExceptionResolver 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 HanlderExceptionResolver 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。

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)。

5.7.2 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。

TABLE 5.15 **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。
- 悲観ロックに失敗した場合、PessimisticLockingFailureException ではなく、org.springframework.dao.UncategorizedDataAccessException の子クラスが発生する。

悲観エラー時に発生する UncategorizedDataAccessException は、システムエラーに分類される例外なので、アプリケーションでハンドリングすることは推奨しない。しかしながら、最悪ハンドリングを行う必要があるかもしれない。原因例外には、悲観ロックエラーが発生したことを通知する例外が格納されているので、ハンドリングできる。

⇒継続調査。

現状以下の動作となる。

- PostgreSQL + for update nowait
  - org.springframework.orm.hibernate3.HibernateJdbcException
  - Caused by: org.hibernate.PessimisticLockException
- Oracle + for update
  - org.springframework.orm.hibernate3.HibernateSystemException
  - Caused by: Caused by: org.hibernate.dialect.lock.PessimisticEntityLockException
  - Caused by: org.hibernate.exception.LockTimeoutException
- Oracle / PostgreSQL + 一意制約
  - org.springframework.dao.DataIntegrityViolationException
  - Caused by: org.hibernate.exception.ConstraintViolationException

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。

TABLE 5.16 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
(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**	1. プレゼンテーション層で	サーブ	Web アプリケー

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 ...

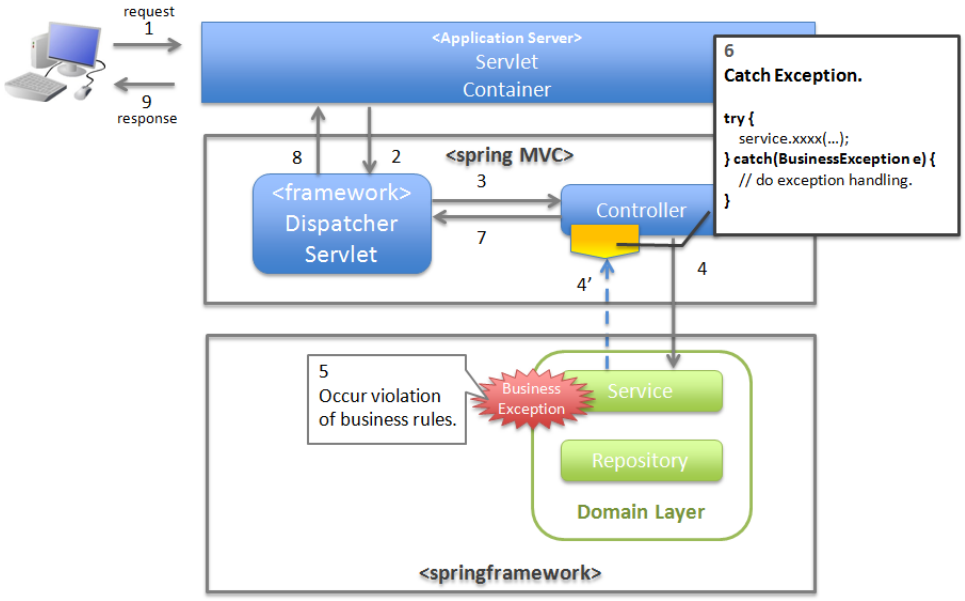


図 5.30 **図-ユースケースの一部やり直し (途中からのやり直し) を促す場合のハンドリング方法**

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 ...

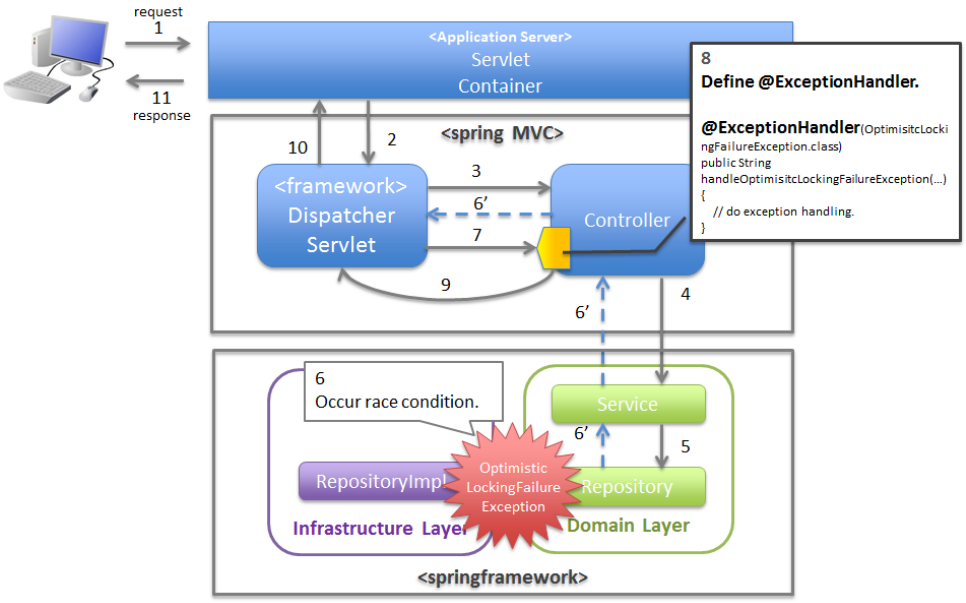


図 5.31　**図-ユースケースのやり直し (先頭からのやり直し) を促す場合のハンドリング方法**

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 ...

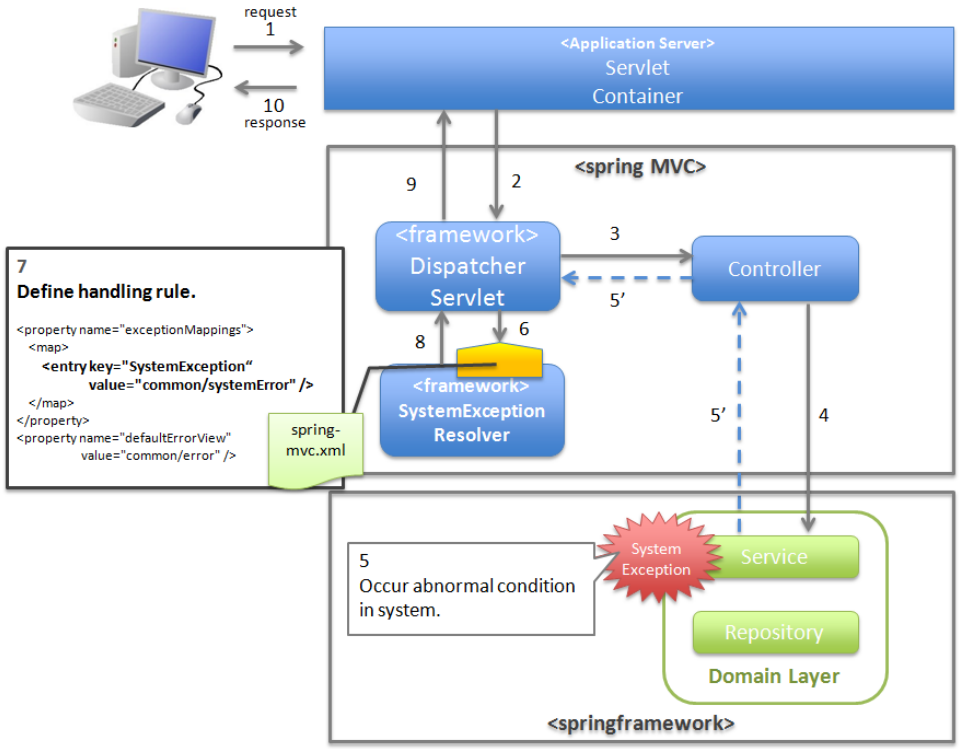


図 5.32　**図-システム、またはアプリケーションが、正常な状態でないことを通知する例外を検知する場合のハンドリング方法**

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。

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 ...

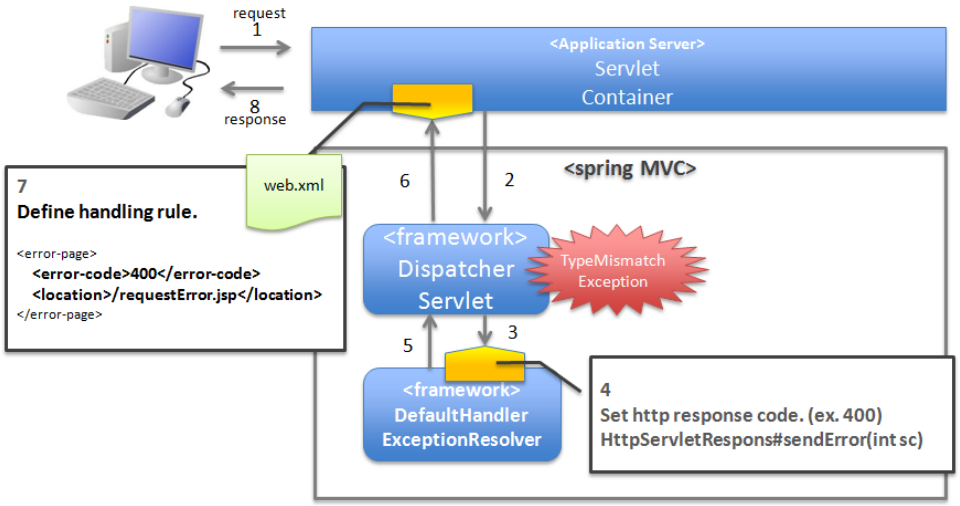


図 5.33　**図-リクエスト内容が不正であることを通知する場合のハンドリング方法**

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。

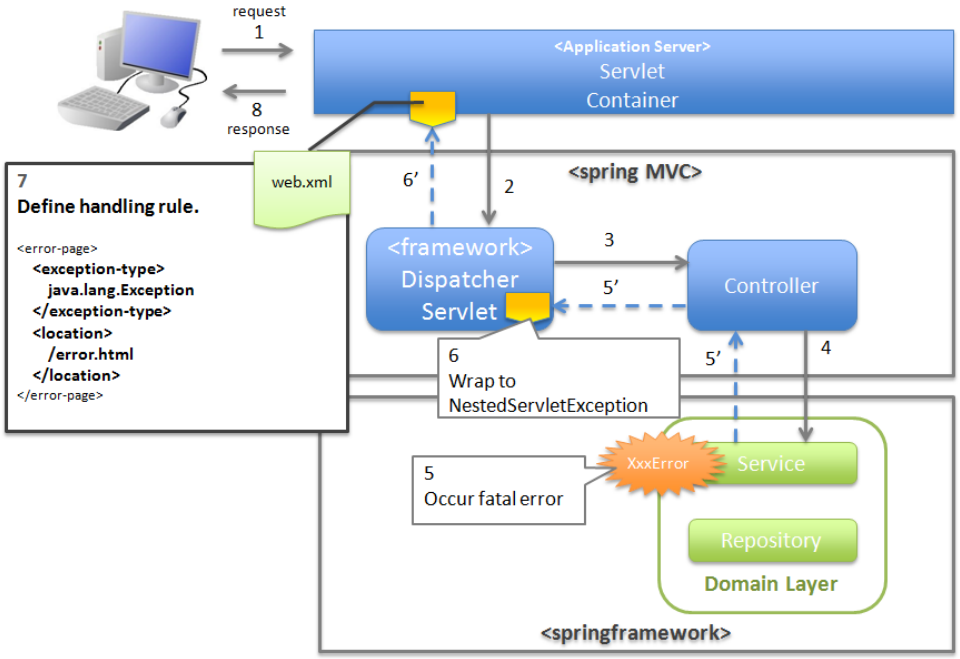


図 5.34　**図-致命的なエラーが発生したことを検知する場合のハンドリング方法**

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。

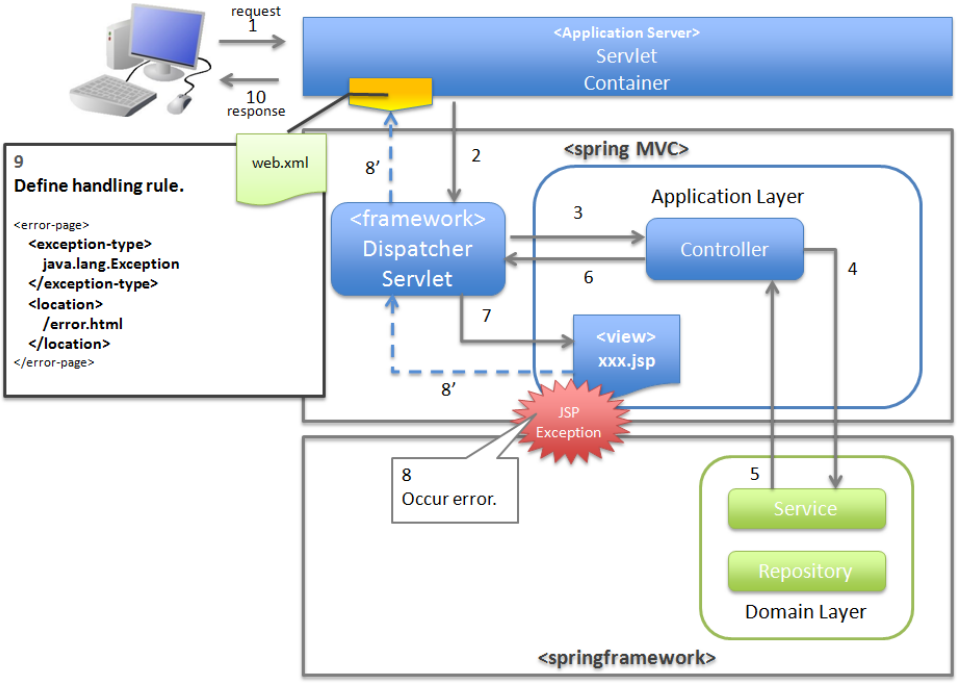


図 5.35　**図-プレゼンテーション層 (JSP など) で例外が発生した事を通知する場合のハンドリング方法**

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。

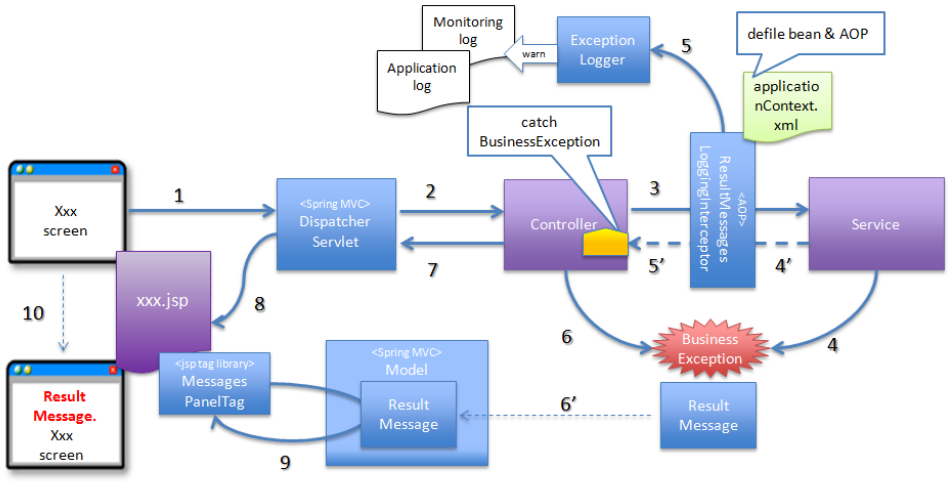


図 5.36　**図-リクエスト単位で Controller クラスがハンドリングする場合の基本フロー**

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。

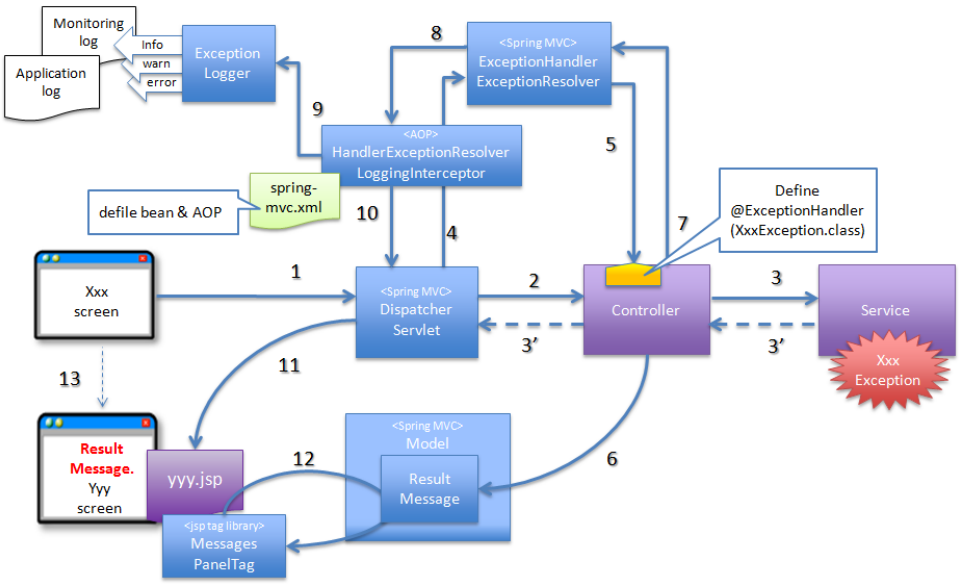


図 5.37　**図-ユースケース単位で、Controller クラスがハンドリングする場合の基本フロー**

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、例外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。

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、例外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。**

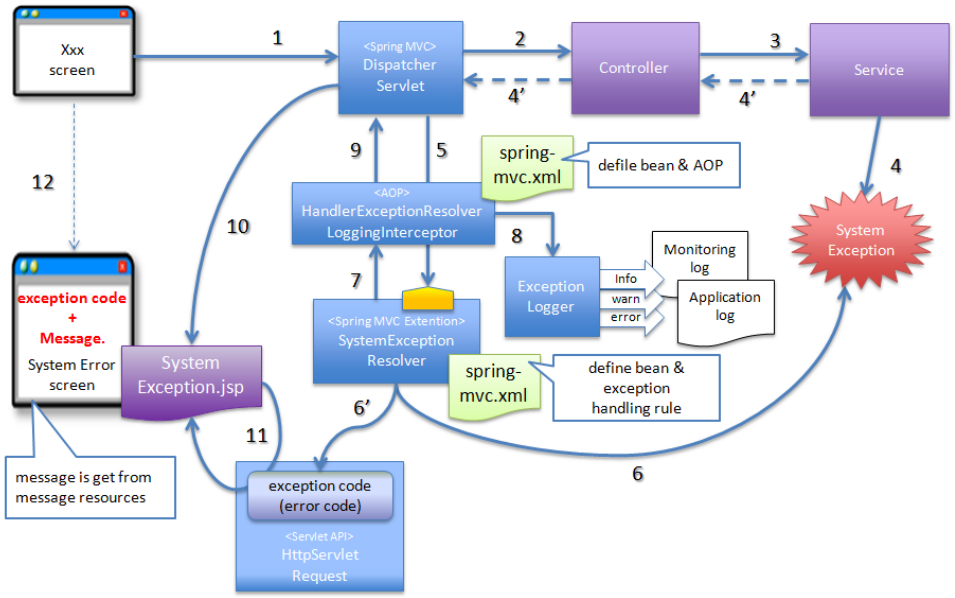


図 5.38　**図-サーブレット単位でフレームワークがハンドリングする場合の基本フロー**

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。

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、例外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。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。

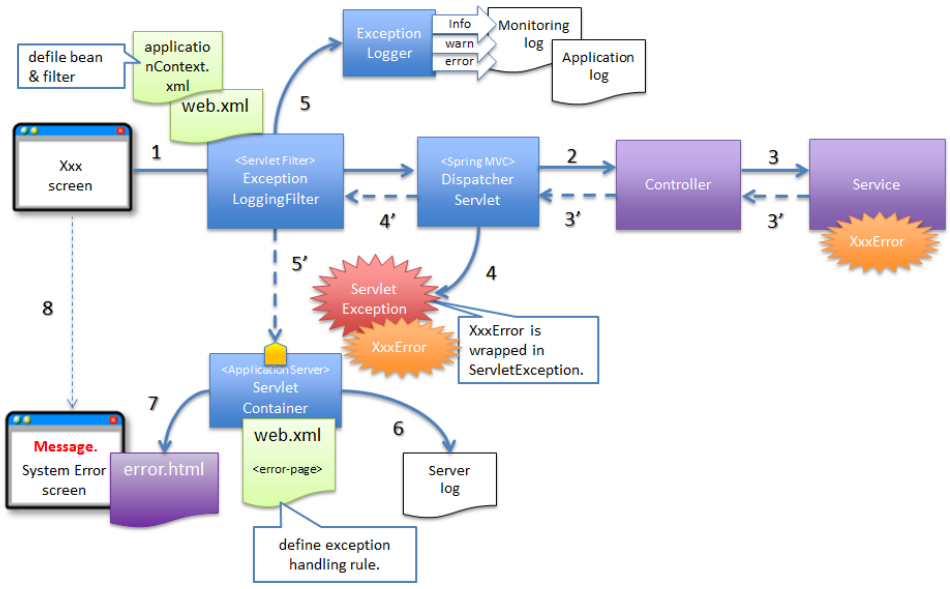


図 5.39　**図-Web アプリケーション単位でサーブレットコンテナがハンドリングする場合の基本フロー**

5.7.3 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、”w.xx.fw.8001”、 “ResourceNotFoundException” が含まれている場合は、”w.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、「例外コード (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】**
(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、ログに出力するのみ。（画面での取得もできる。JSP へのリンク）プロパティに定義している形式でなくとも、運用上でわかる ID にすることが可能である。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"
        <file>log/projectName-monitoring.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>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\
        </encoder>
    </appender>

    <logger name="org.terasoluna.gfw.common.exception.ExceptionLogger.Monitoring" additivity="fa
        <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
        <file>log/projectName-application.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>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}\
        </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、例外の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 HanlderExceptionResolver 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/transactionTok
                <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="defaultErrorView" value="common/error/systemError" /> <!-- (6) -->
        <property name="defaultStatusCode" value="500" /> <!-- (7) -->
    </bean>

    <!-- Settings View Resolver. -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver"> <!-- (8) -->
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
```

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。
(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、`HanlderExceptionResolver` 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)	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

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"> <!-
    <property name="exceptionLogger" ref="exceptionLogger" /> <!-- (2) -->
</bean>
<aop:config>
    <aop:advisor advice-ref="handlerExceptionResolverLoggingInterceptor"
        pointcut="execution(* org.springframework.web.servlet.HandlerExceptionResolver.resol
</aop:config>
```

項番	説明
(1)	ExceptionHandlerLoggingInterceptor 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> <!-- (
</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
```

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。

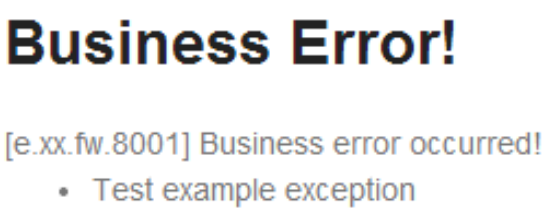
```
e.ad.od.5001 = Order number is higher than the stock quantity={0}. Change the order number.
```

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
org.terasoluna.gfw.common.exception.BusinessException: ResultMessages [type=error, list=[Res

// stackTarace ommited
...
```

```
date:2013-09-17 22:25:55    thread:tomcat-http--8    X-Track:6cfb0b378c124b918e40ac0c32a1fac7
date:2013-09-17 22:25:55    thread:tomcat-http--8    X-Track:6cfb0b378c124b918e40ac0c32a1fac7
date:2013-09-17 22:25:55    thread:tomcat-http--8    X-Track:6cfb0b378c124b918e40ac0c32a1fac7
date:2013-09-17 22:25:55    thread:tomcat-http--8    X-Track:6cfb0b378c124b918e40ac0c32a1fac7
```

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。

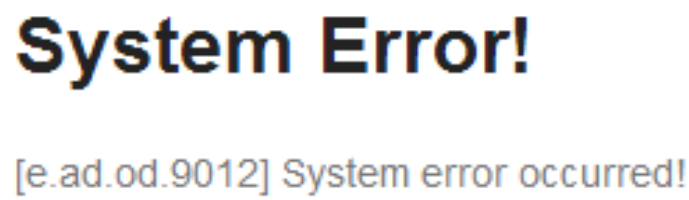
```
if (itemEntity == null) {                                    // (1)
    throw new SystemException("e.ad.od.9012",
        "not found item entity. item code [" + itemId + "]."); // (2)
}
```

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:c19eec546b054d54a13658f94292b2
date:2013-09-19 21:03:06      thread:tomcat-http--3   X-Track:c19eec546b054d54a13658f94292b2
date:2013-09-19 21:03:06      thread:tomcat-http--3   X-Track:c19eec546b054d54a13658f94292b2
date:2013-09-19 21:03:06      thread:tomcat-http--3   X-Track:c19eec546b054d54a13658f94292b2
date:2013-09-19 21:03:06      thread:tomcat-http--3   X-Track:c19eec546b054d54a13658f94292b2
org.terasoluna.gfw.common.exception.SystemException: not found item entity. item code [10-12
      at org.terasoluna.exception.domain.service.ExampleExceptionServiceImpl.throwSystemExce
...
// stackTarace ommited
```

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。

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)	共通ライブラリで提供している `org.terasoluna.gfw.common.exception.ExceptionLogger` をログ出力するため、オブジェクトを DI する。
(2)	ハンドリング対象の例外を、キャッチする。
(3)	ハンドリングした例外を、ログに出力する。例では、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。

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 21:31:47      thread:tomcat-http--3   X-Track:df5271ece2304b12a2c59ff4948063
org.springframework.web.client.RestClientException: Test example exception
...
// stackTarace ommited
```

> **警告:** `exceptionLogger` で、log() を使用した場合には、error レベルで出力されるため、デフォルトで監視ログにも出力される。

```
date:2013-09-19 21:31:47  X-Track:df5271ece2304b12a2c59ff494806397        level:ERROR     me
```

次の例のように、処理を継続させて問題ない場合に、運用監視で監視ログを監視している場合は、出力レベルで監視されないレベルにするか、メッセージから監視されないよう定義が必要である。

```
} catch (RestClientException e) {
    exceptionLogger.info(e);
}
```

デフォルトの設定では、error レベル以外の監視ログは出力されない。アプリケーションログには、以下のように出力される。

```
    date:2013-09-19 22:17:53    thread:tomcat-http--3   X-Track:999725b111b4445b8d10b4ea44639c61
    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 handdleBusinessException(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。

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: Internet Explorer がサポートブラウザとなっている場合は、エラー画面として応答する HTML のサイズが 513 バイト以上になるように実装する必要がある。

Internet Explorer では、

- 応答されたステータスコードがエラー系 (4xx と 5xx)
- 応答された HTML が 512 バイト以下

- ブラウザの設定が「HTTP 簡易メッセージを表示する」が有効な状態

という 3 つの条件を充たした際に、Internet Explorer が用意している簡易メッセージが表示される仕組みになっているためである。

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

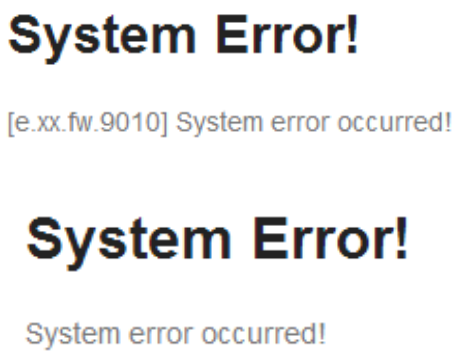
任意の場所に、例外コード (メッセージ ID) と、固定メッセージを表示する際の実装例を、以下に示す。

```
<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。

5.7.4 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。

5.7.5 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。

TABLE 5.17 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。
(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`

TABLE 5.18 **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 HanlderExceptionResolver 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、例外コードの 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。
(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。

TABLE 5.19 **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	exceptionCode Attribute	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) のヘッダー名	exceptionCode Header	例外コード (メッセージ ID) として、HttpServletResponse のレスポンスヘッダーに設定する際のヘッダー名 (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。**
(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。**

(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" /> <

    <!-- 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" /> <!-

    <!-- 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) のヘッダー名

デフォルトのヘッダー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" /> <!

    <!-- omitted -->
</bean>
```

項番	説明
(1)	例外コード (メッセージ ID) のヘッダー名 (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ー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 レスポンスヘッダー

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 レスポンスヘッダーが出力される。

Cache-Control:no-store

Cache-Control:no-cache

Expires:Thu, 01 Jan 1970 00:00:00 GMT

Pragma:no-cache

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。

TABLE 5.20 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) でログ出力さえっる例外のみ指定すること。	ResultMessagesNoti ResultMessagesNoti 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</
            <!-- (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.MissingServletRequestParameterException	400
(6)	org.springframework.web.bind.ServletRequestBindingException	400
(7)	org.springframework.beans.ConversionNotSupportedException	500
(8)	org.springframework.beans.TypeMismatchException	400
(9)	org.springframework.http.converter.HttpMessageNotReadableException	400
(10)	org.springframework.http.converter.HttpMessageNotWritableException	500
(11).	org.springframework.web.bind.MethodArgumentNotValidException	400
(12)	org.springframework.web.multipart.support.MissingServletRequestPartException	400

5.8 セッション管理

5.8.1 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。

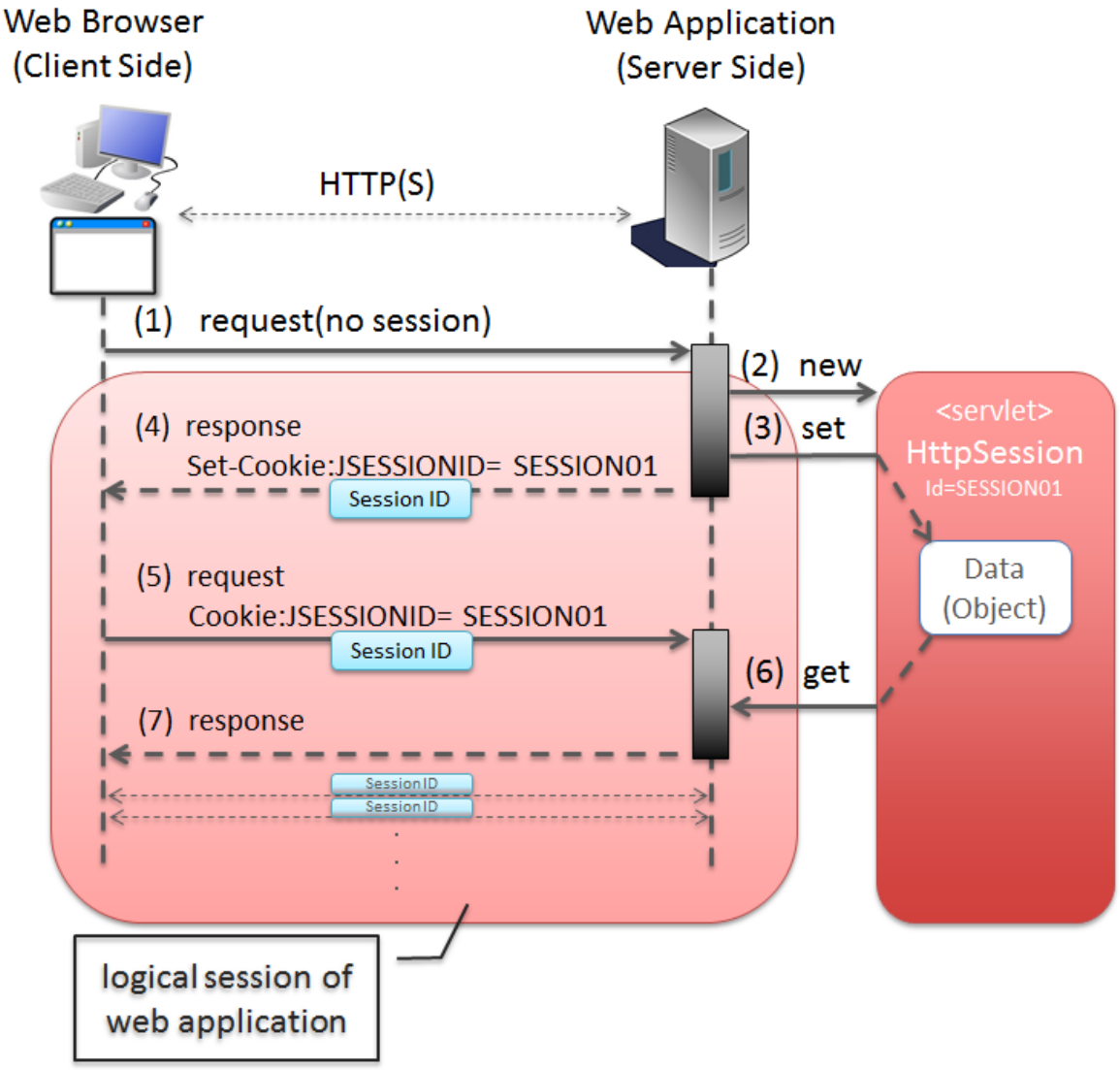


図 5.40 **Picture - Establishment of logical session**

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、

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

JavaEE 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、*Spring Security* **におけるセッション管理**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。
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。
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。

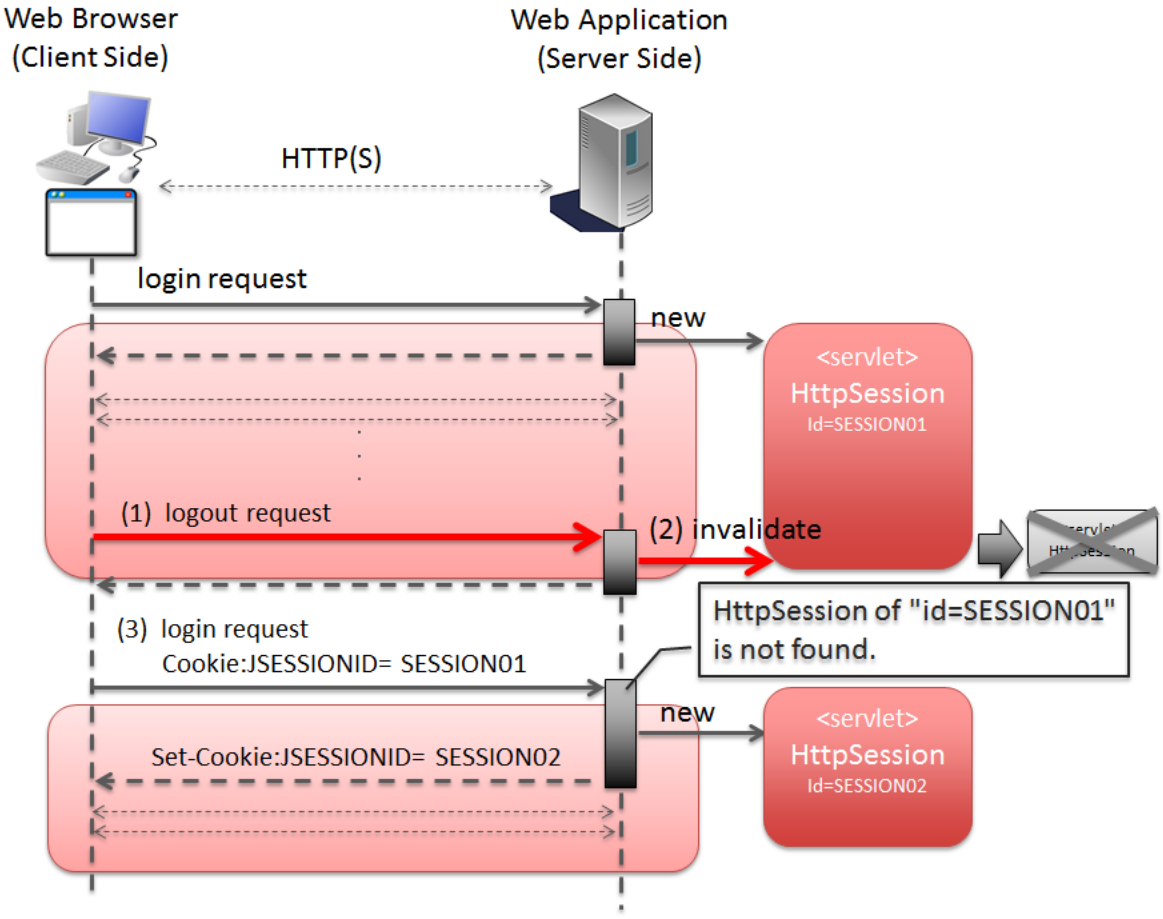


図 5.41 **Picture - Invalidate session by processing of Web Application**

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

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

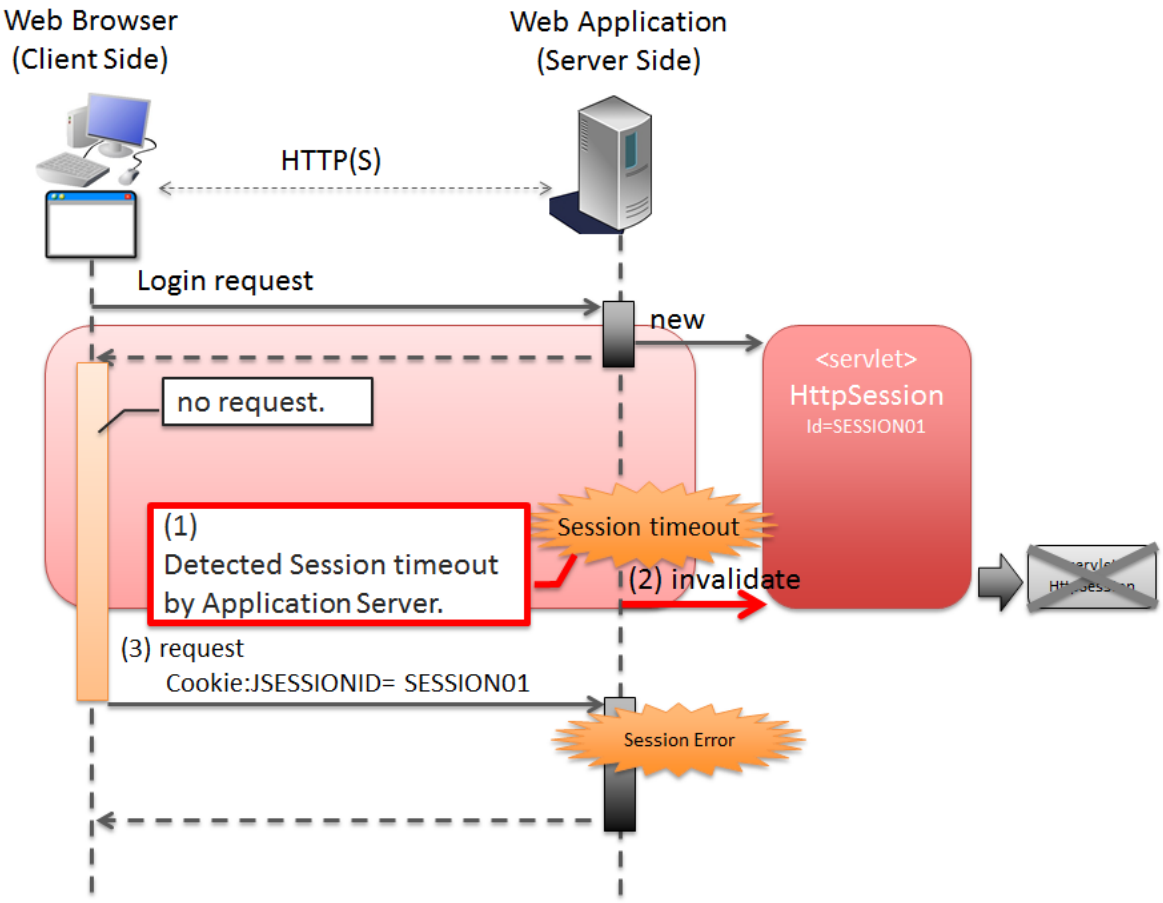


図 5.42　**Picture - Invalidate session by Application Server**

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)
- Resin : 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、*Spring Security* **におけるセッション管理**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。

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

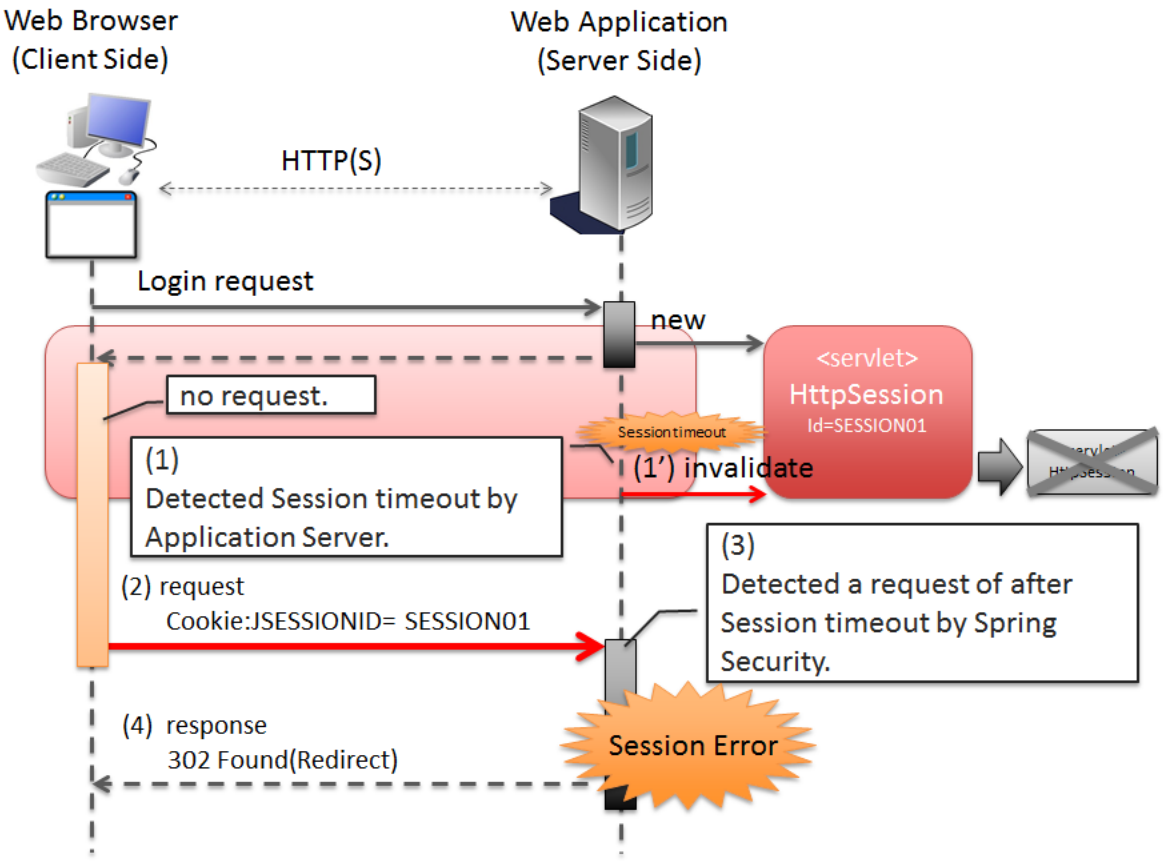


図 5.43　**Picture - Detected a request of after session timeout by Spring Security**

「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。

課題

TBD

Spring Security から提供されている CSRF 対策を有効にすると、POST メソッドの呼び出し時にセッションタイムアウトが検知される前に SCRF トークンエラーが発生してしまう。

現在、 Spring Security の JIRA(SEC-2422) にて、Spring Security の見解を問い合わせ中である。Spring Security の見解によっては、Servlet Filter を作成し、Spring Security の CSRF 対策用の Servlet Filter より前でチェックする必要がある。

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。

セッションの利用時のメリットでデメリット

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に格納することで View(JSP) で表示すれば、セッションに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。

5.8.2 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 の処理メソッドの引数に `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 の処理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

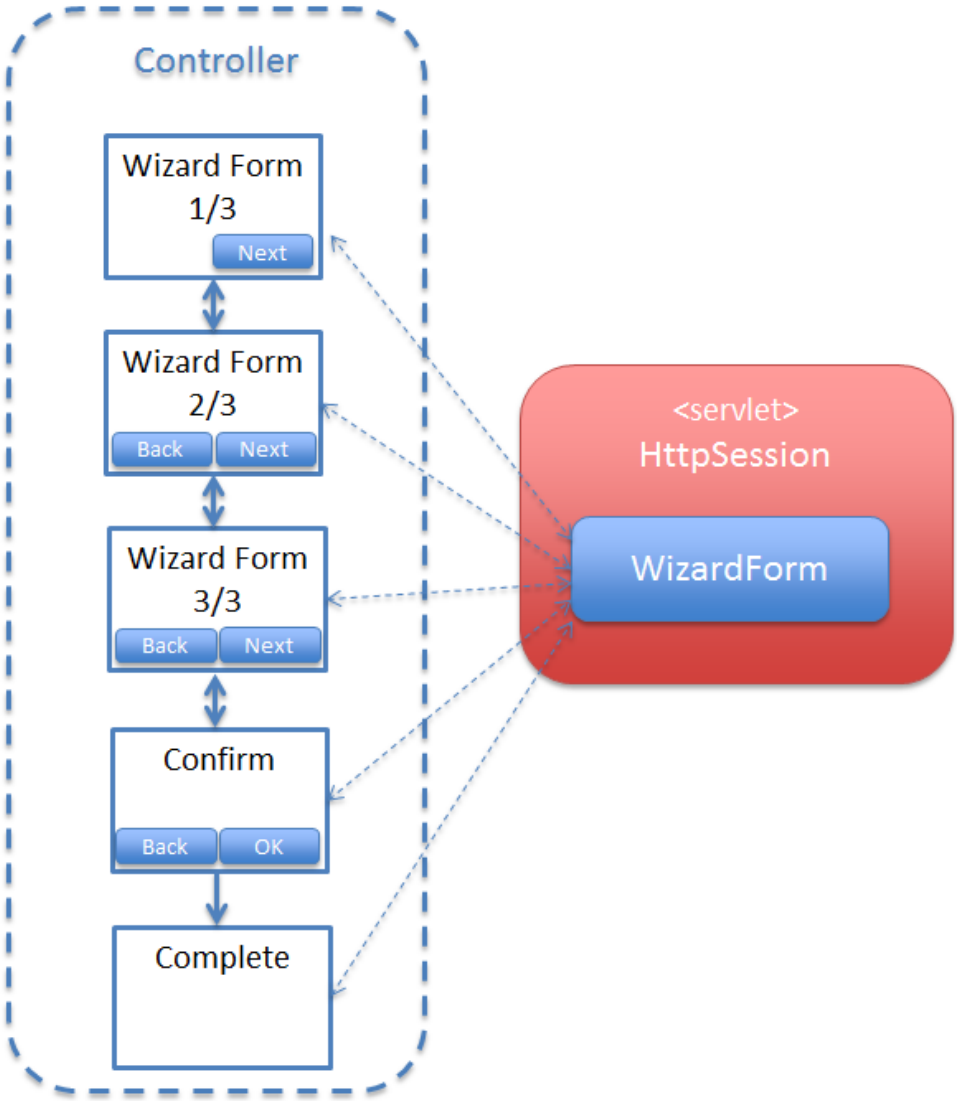
`@SessionAttributes` アノテーションは、Controller 内で行われる画面遷移において、データを持ち回る場合に使用する。

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の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、@SessionAttribute 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: @SessionAttribute 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、@SessionAttribute 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 の処理メソッドで、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"アノテーション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 の処理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 の処理メソッドでは、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。

Controller の処理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として、 /WEB-INF/views/common/error/operationError.jsp 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 の処理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 の処理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ト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処理メソッド。
(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処理メソッド。
(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処理メソッド。
(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処理メソッド。
(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処理メソッド。

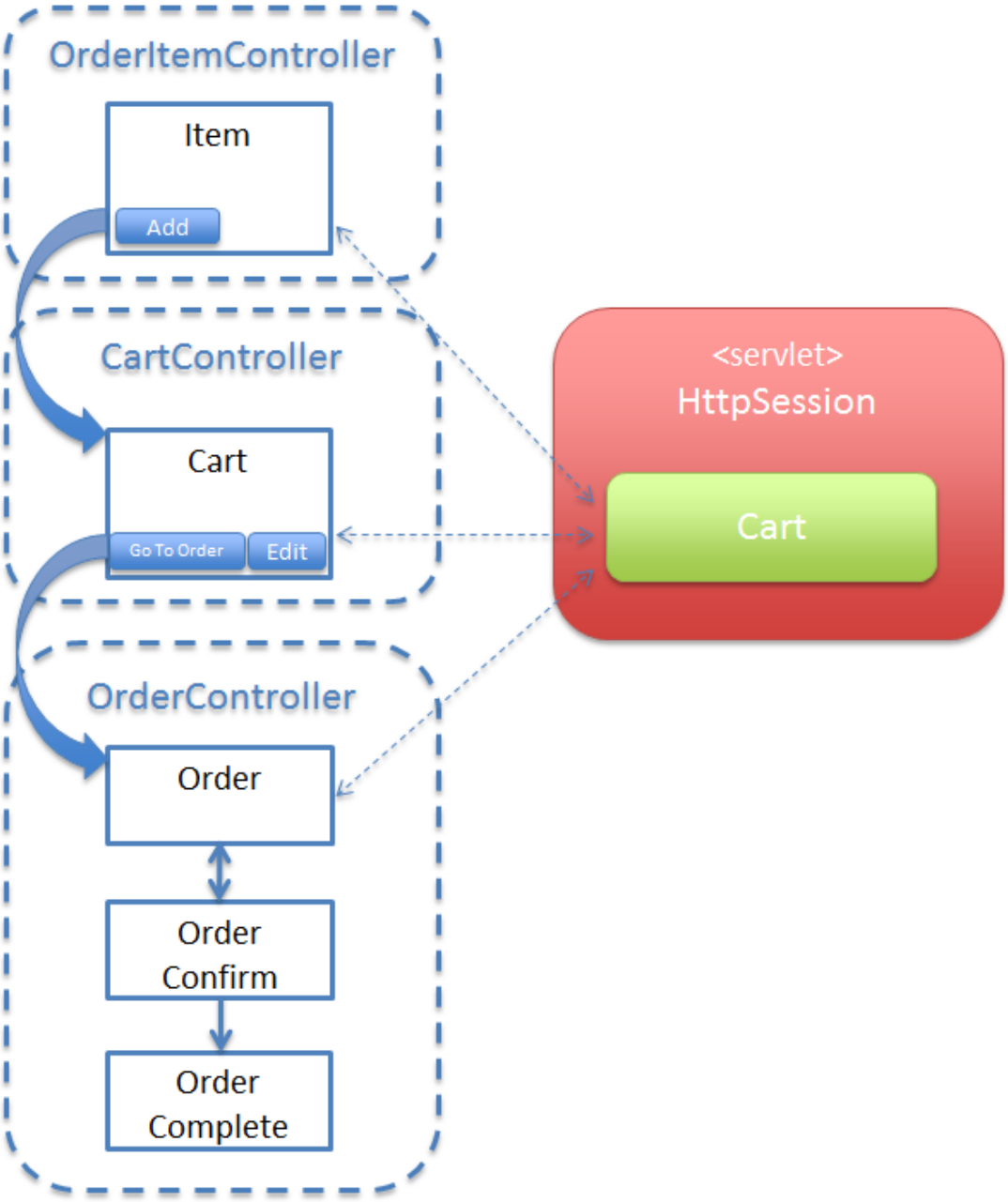
@SessionAttributes を使った処理の実装例

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("session") // (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;
    }

}
```

項番	説明
(1)	Bean のスコープを"session"にする。

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。

- spring-mvc.xml

```
<context:component-scan base-package="xxx.yyy.zzz.app"
    scoped-proxy="targetClass" /> // (2)
```

項番	説明
(2)	<context:component-scan> 要素の scoped-proxy 属性に、 "targetClass" を指定し、scoped-proxy を有効にする。

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。

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)

// (2)
@ModelAttribute
```

```
    public SessionCart setUpSessionCart() {
        return sessionCart;
    }

    @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(), // (3)
                cartItem);
        sessionCart.setCart(addedCart); // (4)
        return "redirect:/cart";
    }
```

項番	説明
(1)	session スコープの Bean を、Controller に Inject する。
(2)	View(JSP) から参照できるようにするために、`Model` オブジェクトに、session スコープの Bean を追加する。
(3)	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。
(4)	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。 (2) で説明した処理によって、session スコープの Bean は、`Model` オブジェクトに格納されているため、 View(JSP) からも、`CartService` の addCartItem メソッドから返却された `Cart` オブジェクトを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

session スコープの Bean を利用して、セッションに格納したオブジェクトを削除する場合、
`@SessionAttributes` アノテーションを使用したときと同様に、
`org.springframework.web.bind.support.SessionStatus` の setComplete メソッドを、Controller の処理メソッドから呼びだす。

`SessionStatus` オブジェクトの setComplete メソッドを呼び出して、セッションから削除するために、
`@SessionAttributes` アノテーションの value 属性に、session スコープの Bean の属性名を指定する必要がある。

```
@Controller
@RequestMapping("order")
@SessionAttributes("scopedTarget.sessionCart") // (1)
public class OrderController {

    @Inject
    SessionCart sessionCart;

    // ...

    @RequestMapping(method = RequestMethod.POST)
    public String order() {
        // ...
        return "redirect:/order?complete";
    }

    @RequestMapping(params = "complete", method = RequestMethod.GET)
    public String complete(Model model, SessionStatus sessionStatus) {
        sessionStatus.setComplete(); // (2)
        model.addAttribute(sessionCart.getCart()); // (3)
        return "order/complete";
    }

}
```

項番	説明
(1)	`@SessionAttributes` アノテーションの value 属性に、session スコープの Bean の属性名を指定する。 属性名は、`"scopedTarget."`+ Bean 名 となる。
(2)	`SessionStatus` オブジェクトの、setComplete メソッドを呼び出す。 上記例では、`"scopedTarget.sessionCart"`という属性名で格納されているオブジェクトが、セッションから削除される。
(3)	View(JSP) にて、session スコープの Bean で保持しているオブジェクトを参照する必要がある場合は、View(JSP) で参照するオブジェクトを、`Model` オブジェクトに格納する必要がある。

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。

5.8.3 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.RequestMappingHandl 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 を定義する。

5.8.4 Appendix

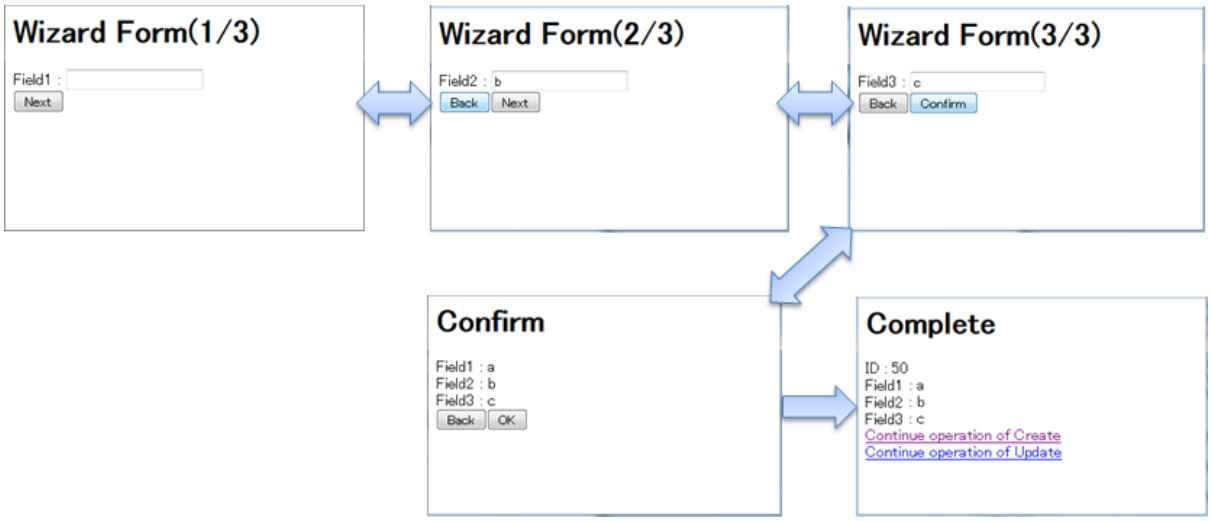
`@SessionAttributes` アノテーションを使ったウィザード形式の画面遷移の実装例

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処理メソッド。 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、この処理メソッドで、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、表示するための処理メソッド。

```
    // (11)
    @RequestMapping(value = "{id}/update", method = RequestMethod.GET)
    public String initializeUpdateWizardForm(@PathVariable("id") Integer id,
```

```
            RedirectAttributes redirectAttributes, SessionStatus sessionStatus) {
        sessionStatus.setComplete();
        redirectAttributes.addAttribute("id", id);
        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処理メソッド。
(12)	1 ページ目の更新用入力画面を、表示するための処理メソッド。
(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、表示するための処理メソッド。
(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、表示するための処理メソッド。
(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処理メソッド。
(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処理メソッド。
(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)	リダイレクト先の処理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処理メソッド。

```
        // (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、再表示するための処理メソッド。
(29)	2 ページ目の入力画面を、再表示するための処理メソッド。
(30)	3 ページ目の入力画面を、再表示するための処理メソッド。

- 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();
            redirectAttributes.addAttribute("id", id);
            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。

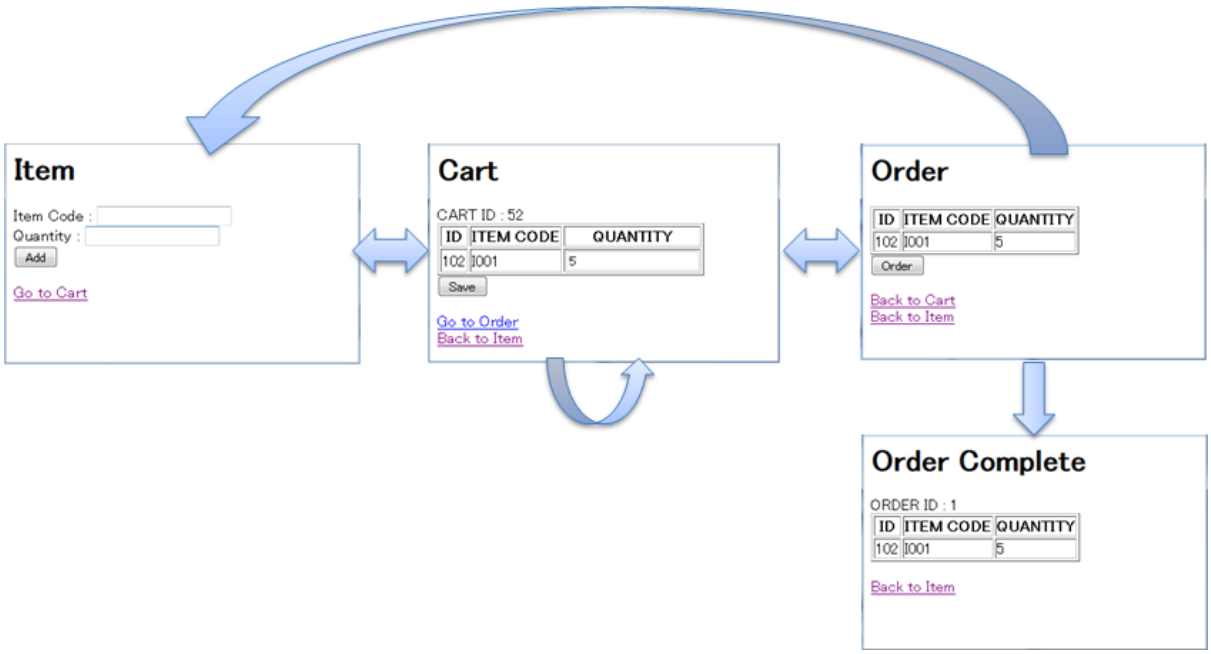
- 商品をカートに追加する処理を提供する。
- カートに追加されている商品の、数量変更を行う処理を提供する。
- カートに格納されている商品を、注文する処理を提供する。
- 上記 3 つの処理は、それぞれ独立した機能として提供するため、別 Controller(ItemController, CartController, OrderController) とする。
- カートは、上記 3 つの処理で共有するため、セッションに格納する。
- 商品をカートに追加した場合は、カート画面に遷移する。

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("session")
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;
        }

    }
```

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。

- ItemController

```
    @Controller
    @RequestMapping("item")
    public class ItemController {

        @Inject
        SessionCart sessionCart;

        @Inject
        CartService cartService;
```

```
        @Inject
        Mapper beanMapper;

        @ModelAttribute
        public ItemForm setUpItemForm() {
            return new ItemForm();
        }

        // (2)
        @RequestMapping
        public String view(Model model) {
            return "item/item";
        }

        // (3)
        @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(), // (4)
                    cartItem);
            sessionCart.setCart(cart); // (5)
            return "redirect:/cart"; // (6)
        }
    }
```

項番	説明
(2)	商品画面を、表示するための処理メソッド。
(3)	指定された商品を、カートに追加するための処理メソッド。
(4)	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。
(5)	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、セッションおよび `Model` オブジェクトに反映される。
(6)	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();
    }
```

```
        // (7)
        @ModelAttribute("sessionCart")
        public SessionCart setUpSessionCart() {
            return sessionCart;
        }

        // (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
(7)	View(JSP) で参照するために、`Model` オブジェクトに追加する。
(8)	カート画面 (数量変更画面) を表示するための処理メソッド。
(9)	数量変更を、行うための処理メソッド。
(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、セッションおよび `Model` オブジェクトに反映される。
(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")
@SessionAttributes("scopedTarget.sessionCart")
public class OrderController {

    @Inject
    SessionCart sessionCart;

    @ModelAttribute
    public OrderForm setUpOrderForm() {
        return new OrderForm();
    }

    // (12)
    @ModelAttribute("sessionCart")
    public SessionCart setUpSessionCart() {
        return sessionCart;
    }
```

```
        // (13)
        @RequestMapping
        public String view() {
            return "order/order";
        }

        // (14)
        @RequestMapping(method = RequestMethod.POST)
        public String order() {
            // ...
            return "redirect:/order?complete";
        }

        // (15)
        @RequestMapping(params = "complete", method = RequestMethod.GET)
        public String complete(Model model, SessionStatus sessionStatus) {
            sessionStatus.setComplete();
            return "order/complete";
        }

    }
```

項番	説明
(12)	View(JSP) で参照するために、Model オブジェクトに追加する。
(13)	注文画面を、表示するための処理メソッド。
(14)	注文処理を行うための処理メソッド。
(15)	注文完了画面を表示するための処理メソッド。

- 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>
        <h1>Cart</h1>
        <c:choose>
            <c:when test="${ empty sessionCart.cart.cartItems }">
                <div>Cart is empty.</div>
            </c:when>
            <c:otherwise>
                CART ID :
                ${f:h(sessionCart.cart.id)}
                <form:form modelAttribute="cartForm">
                    <table border="1">
                        <thead>
                            <tr>
                                <th>ID</th>
                                <th>ITEM CODE</th>
                                <th>QUANTITY</th>
                            </tr>
                        </thead>
                        <tbody>
                            <c:forEach var="item"
                                items="${sessionCart.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>
                    <%-- (16) --%>
                    <form:button name="edit">Save</form:button>
                </form:form>
            </c:otherwise>
        </c:choose>
        <c:if test="${ not empty sessionCart.cart.cartItems }">
            <div>
                <%-- (17) --%>
                <a href="${pageContext.request.contextPath}/order">Go to Order</a>
            </div>
        </c:if>
        <div>
            <a href="${pageContext.request.contextPath}/item">Back to Item</a>
        </div>
    </body>
    </html>
```

項番	説明
(16)	数量を更新するためのボタン。
(17)	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>
        <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="${sessionCart.cart.cartItems}"
                    varStatus="rowStatus">
                    <tr>
                        <td>${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">
            <%-- (18) --%>
            <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
(18)	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>
```

5.9 メッセージ管理

5.9.1 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

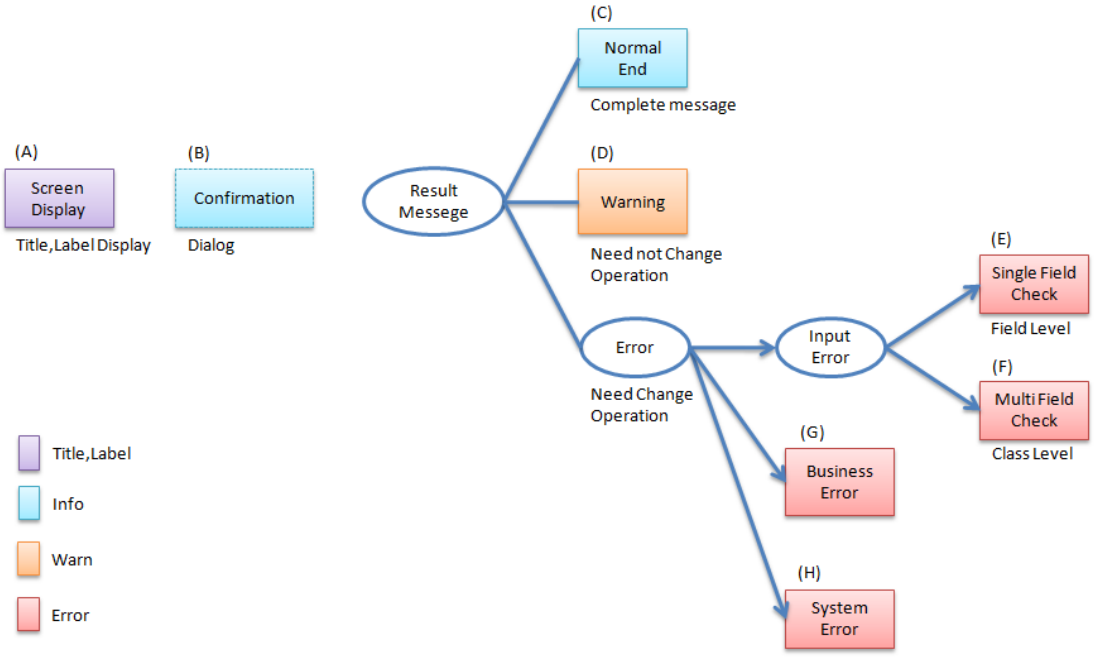
ユーザの画面操作の結果に応じて表示するメッセージは、内容に応じて、以下 3 種類のメッセージタイプに分けて管理する。

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。
(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 p
# 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。

5.9.2 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` が使用するため、`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。

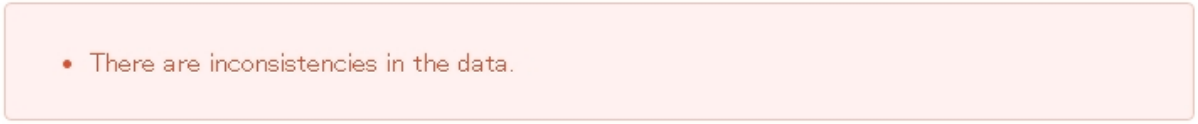
標準では、以下のメッセージタイプが用意されている。

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
success	`ResultMessages.success()`	alert alert-success
info	`ResultMessages.info()`	alert alert-info
warn	`ResultMessages.warn()`	alert alert-warn
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-warn {
  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.warn().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 3.0.0 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.warn().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”warn” 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。

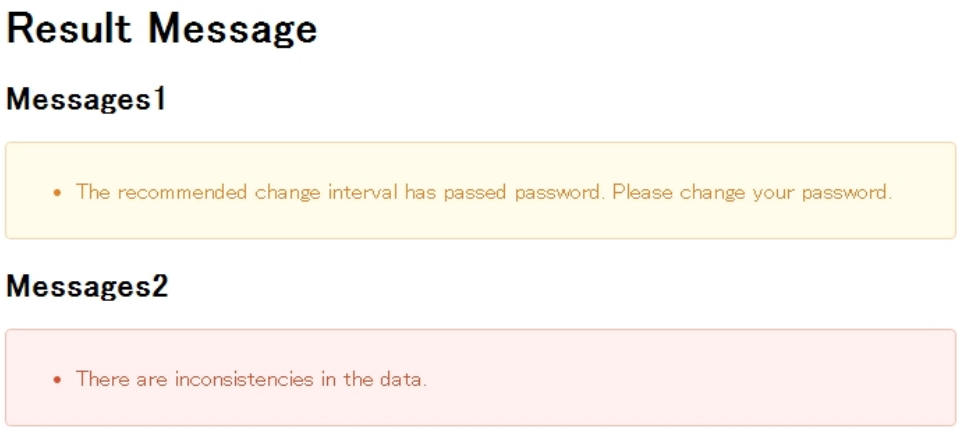
- JSP (WEB-INF/views/message/showMessages.jsp)

```
<!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-warn {
    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。

5.9.3 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。

5.9.4 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。

TABLE 5.21 <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。

- There are inconsistencies in the data.

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 t
```

- 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 は認証エラー時に、”SPRING_SECURITY_LAST_EXCEPTION” という属性名で発生した例外クラスを
リクエストスコープに設定する。

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);
}
```

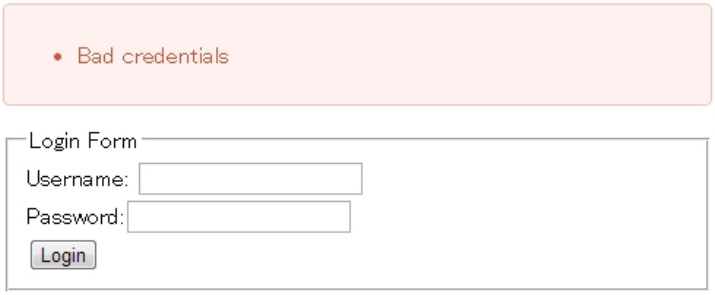
```
</style>
</head>
<body>
    <c:if test="${param.error}">
        <t:messagesPanel messagesType="error"
            messagesAttributeName="SPRING_SECURITY_LAST_EXCEPTION" /><!-- (2) -->
    </c:if>
    <form:form
        action="${pageContext.request.contextPath}/authentication"
        method="post">
        <fieldset>
            <legend>Login Form</legend>
            <div>
                <label for="username">Username: </label><input
                    type="text" id="username" name="j_username">
            </div>
            <div>
                <label for="username">Password:</label><input
                    type="password" id="password" name="j_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。
(1)	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;
```

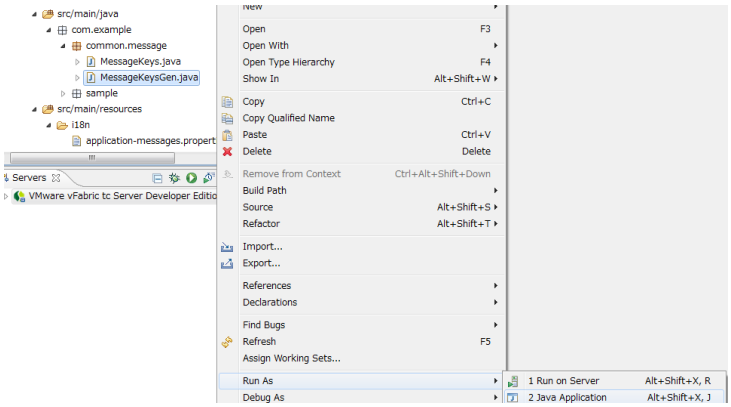
```
import java.io.PrintWriter;
import java.util.regex.Pattern;

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/applicatio
        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 pass
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 y
    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";
}
```

5.10 プロパティ管理

5.10.1 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。

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方法については、「環境依存性の排除」を参照されたい。

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

5.10.2 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.dbcp.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.dbcp.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。

5.10.3 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=
```

5.11 ページネーション

5.11.1 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。

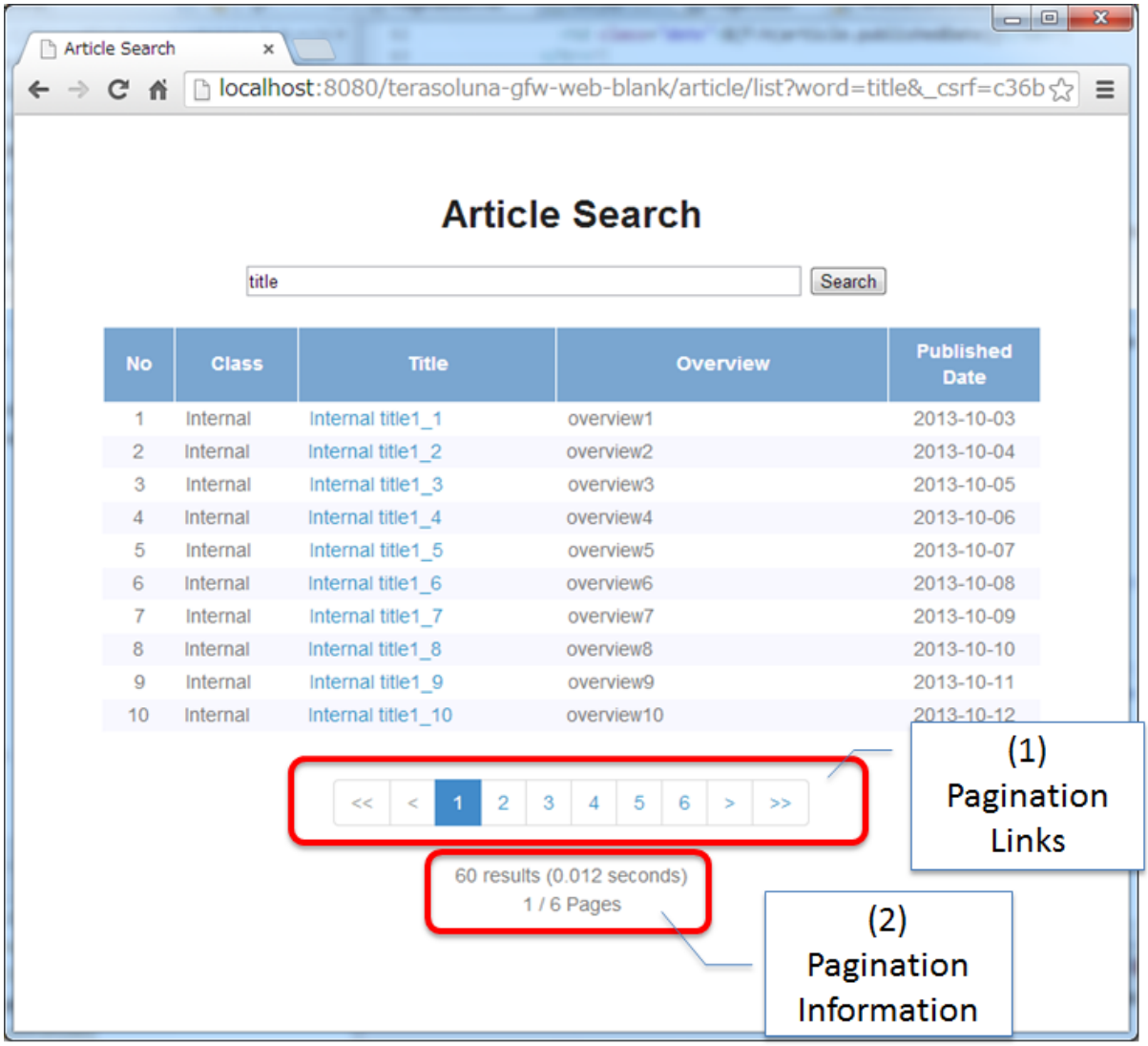
一度に大量のデータを取得し画面に表示すると、以下 3 点の問題が発生する可能性がある。

- 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。

項番	説明
(1)	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。この領域を表示するための JSP タグライブラリを共通ライブラリとして提供している。
(2)	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。 この領域を表示するためのタグライブラリは存在しないため、JSP の処理として個別に実装する必要がある。



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、Mybatis2 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 又は 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"` を指定した場合、 `"ORDER BY lastModifiedDate DESC, id DESC, subId ASC"` という Order By 句が Query に追加される。

警告: spring-data-commons 1.6.1.RELEASE における「size=0」指定時の動作について

terasoluna-gfw-common 1.0.0.RELEASE が依存する spring-data-commons 1.6.1.RELEASE では、`"size=0"` を指定すると条件に一致するレコードを全件取得するという不具合がある。そのため、大量のレコードが取得対象となる可能性がある場合は、`java.lang.OutOfMemoryError` が発生する可能性が高くなる。

この問題は Spring Data Commons の JIRA「DATACMNS-377」で対応され、spring-data-commons 1.6.3.RELEASE で解消されている。改修後の動作としては、`"size<=0"` を指定した場合は、 size パラメータ省略時のデフォルト値が適用される。

terasoluna-gfw-common 1.0.0.RELEASE を使用している場合は、terasoluna-gfw-common 1.0.1.RELEASE 以上へバージョンアップする必要がある。

警告: spring-data-commons 1.6.1.RELEASE におけるリクエストパラメータに不正な値を指定した際の動作について

terasoluna-gfw-common 1.0.0.RELEASE が依存する spring-data-commons 1.6.1.RELEASE では、ページ検索用のリクエストパラメータ (page, size, sort) に不正な値を指定した場合、`java.lang.IllegalArgumentException` 又は `java.lang.ArrayIndexOutOfBoundsException` が発生し、SpringMVC のデフォルトの設定だとシステムエラー (HTTP ステータスコード=500) となってしまうという不具合がある。
この問題は Spring Data Commons の JIRA「DATACMNS-379」と「DATACMNS-408」で対応され、spring-data-commons 1.6.3.RELEASE で解消されている。改修後の動作としては、不正な値を指定した場合は、 パラメータ省略時のデフォルト値が適用される。
terasoluna-gfw-common 1.0.0.RELEASE を使用している場合は、terasoluna-gfw-common 1.0.1.RELEASE 以上へバージョンアップする必要がある。

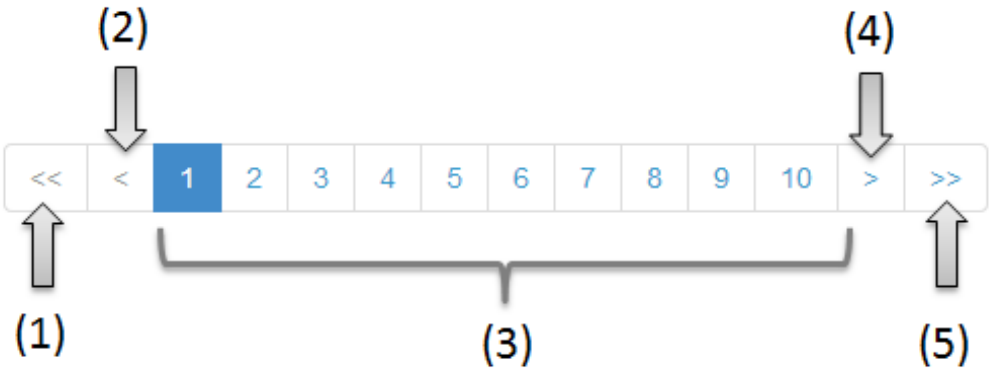
ページネーションリンクの表示について

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説明する。

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。

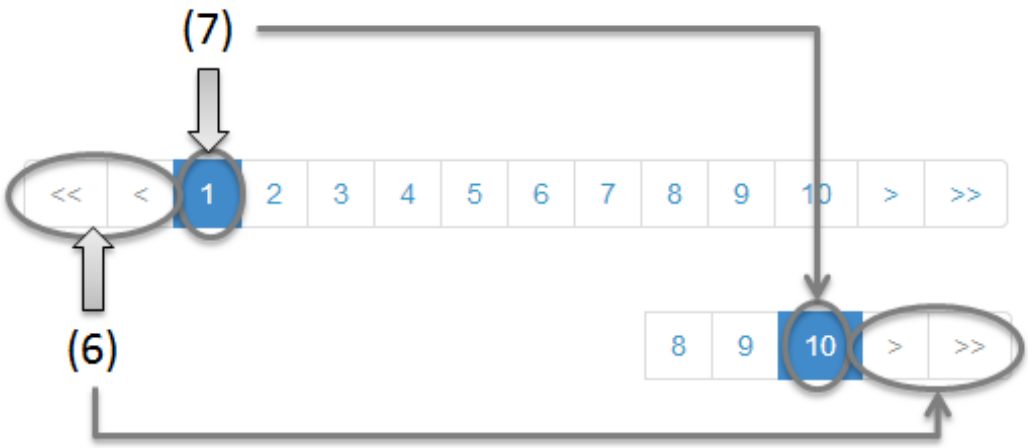
ページネーションリンクの構成

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。

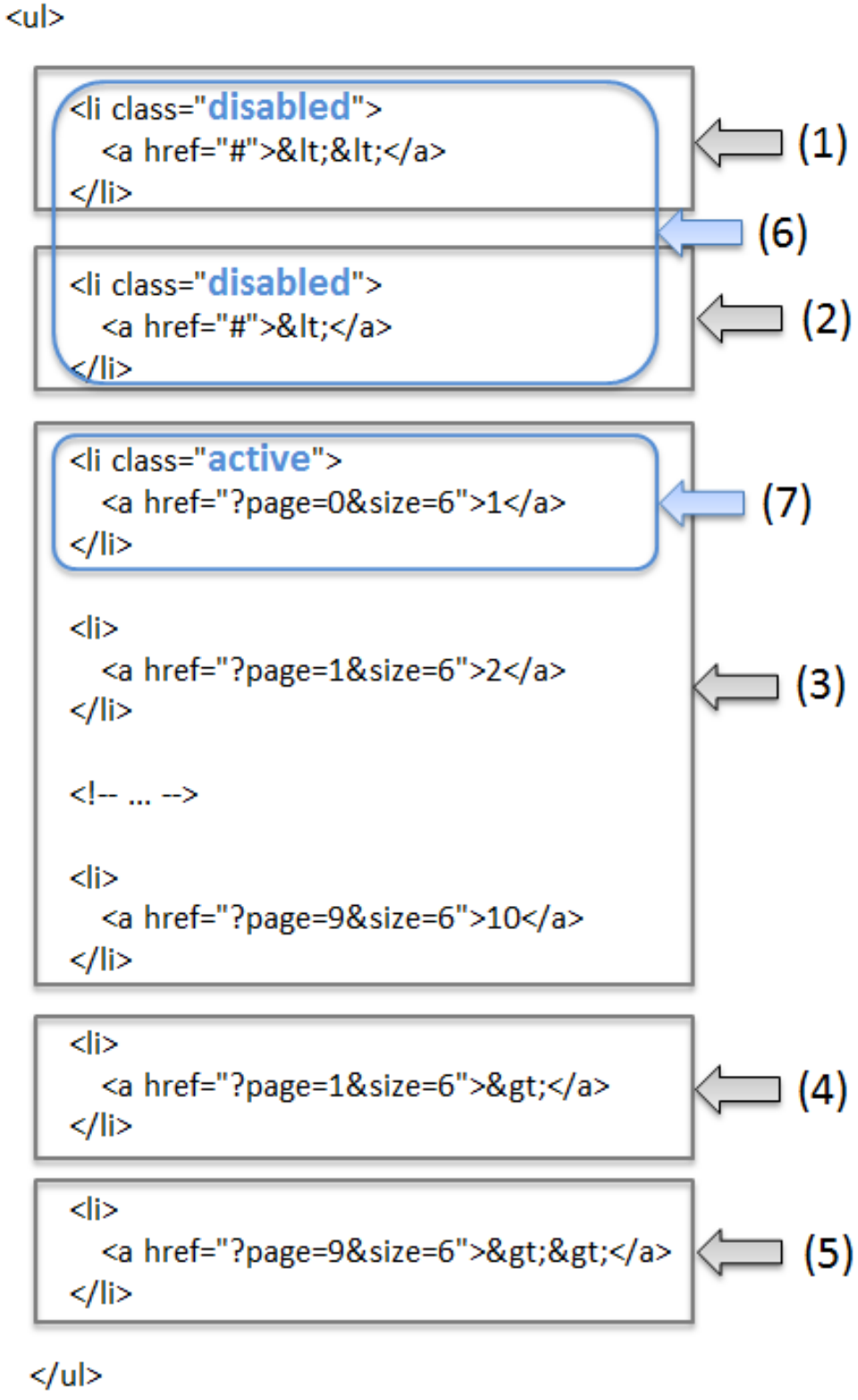
共通ライブラリを使って出力される HTML は、以下の構造となる。

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



<li class="disabled">
<<

(1)
(6)
<li class="disabled">
<

(2)
<li class="active">
1

(7)

2

(3)
<!-- ... -->

10

>

(4)

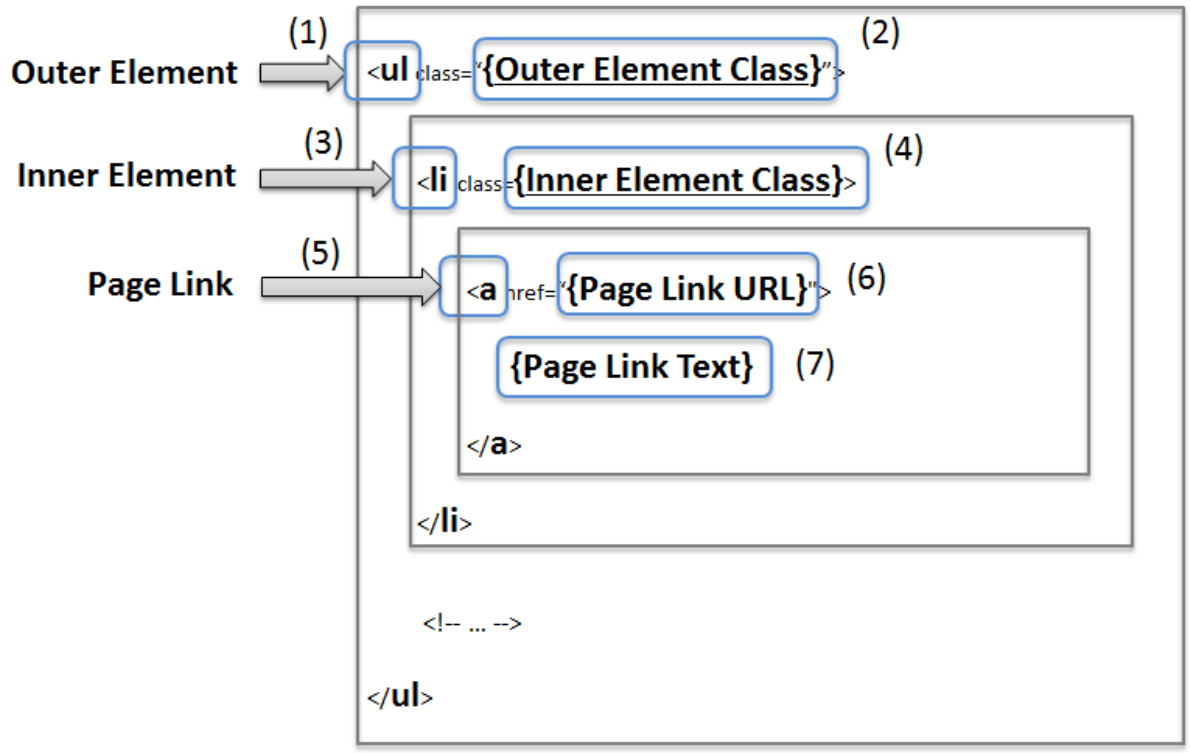
>>

(5)

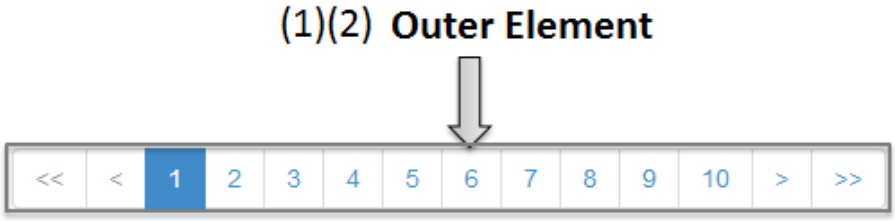
ページネーションリンクの HTML 構造

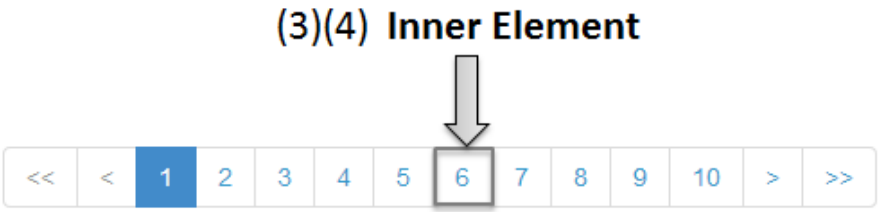
共通ライブラリを使って出力されるページネーションリンクの HTML は、以下の構造となる。

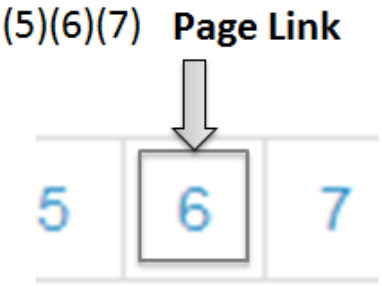
- HTML



- 画面イメージ







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	「*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"` の場合は `"#"`、`"disabled"` 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。

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
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

警告: active 状態のページリンクの制約事項

"active" 状態のページリンクを無効化する仕組みは提供していないため、ページリンクが押下された場合は、該当ページを取得するためのリクエストが送信される。リクエストが送信されないように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か、JavaScript によってリクエストが送信されないように制御する必要がある。

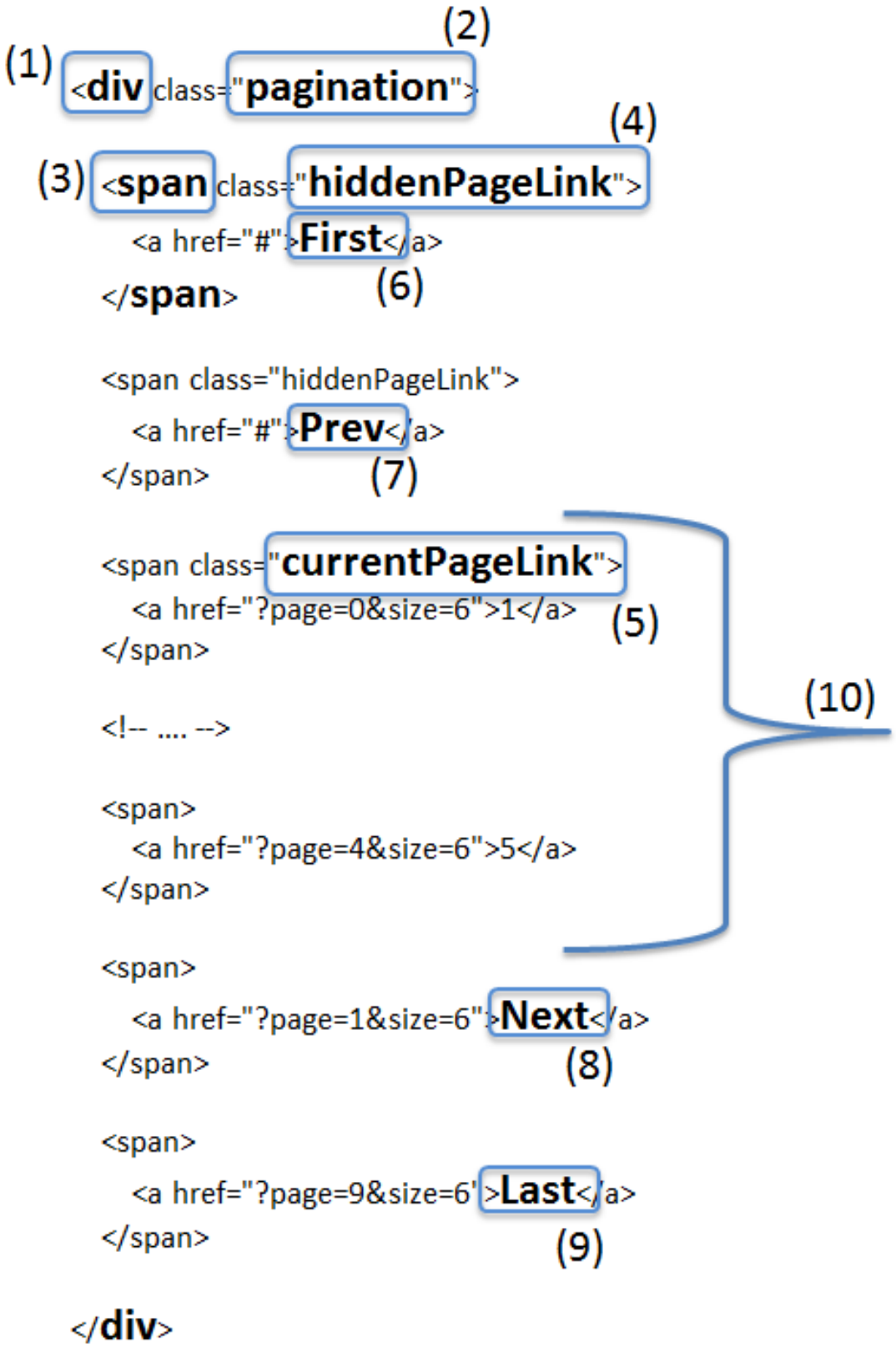
レイアウトを制御するためのパラメータを、全てデフォルトから変更した時に出力される HTML は以下の通り。図中の番号は、上記で説明したパラメータ一覧の項番に対応している。

- 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

動作を制御するためのパラメータ


(1)
(2)

(3)
(4)

First
(6)

Prev
(7)

1
(5)

<!-- .... -->

5

(10)

Next
(8)

Last
(9)

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。 例) javascript:void(0);
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。
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

注釈: disabledHref の設定値について

デフォルトでは、 `disabledHref` には `"#"` が設定されているため、ページリンクを押すとフォーカスがページのトップに移動する。ページリンク押下時の動作を完全に無効化したい場合は、`"javascript:void(0);"` を指定する必要がある。

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、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

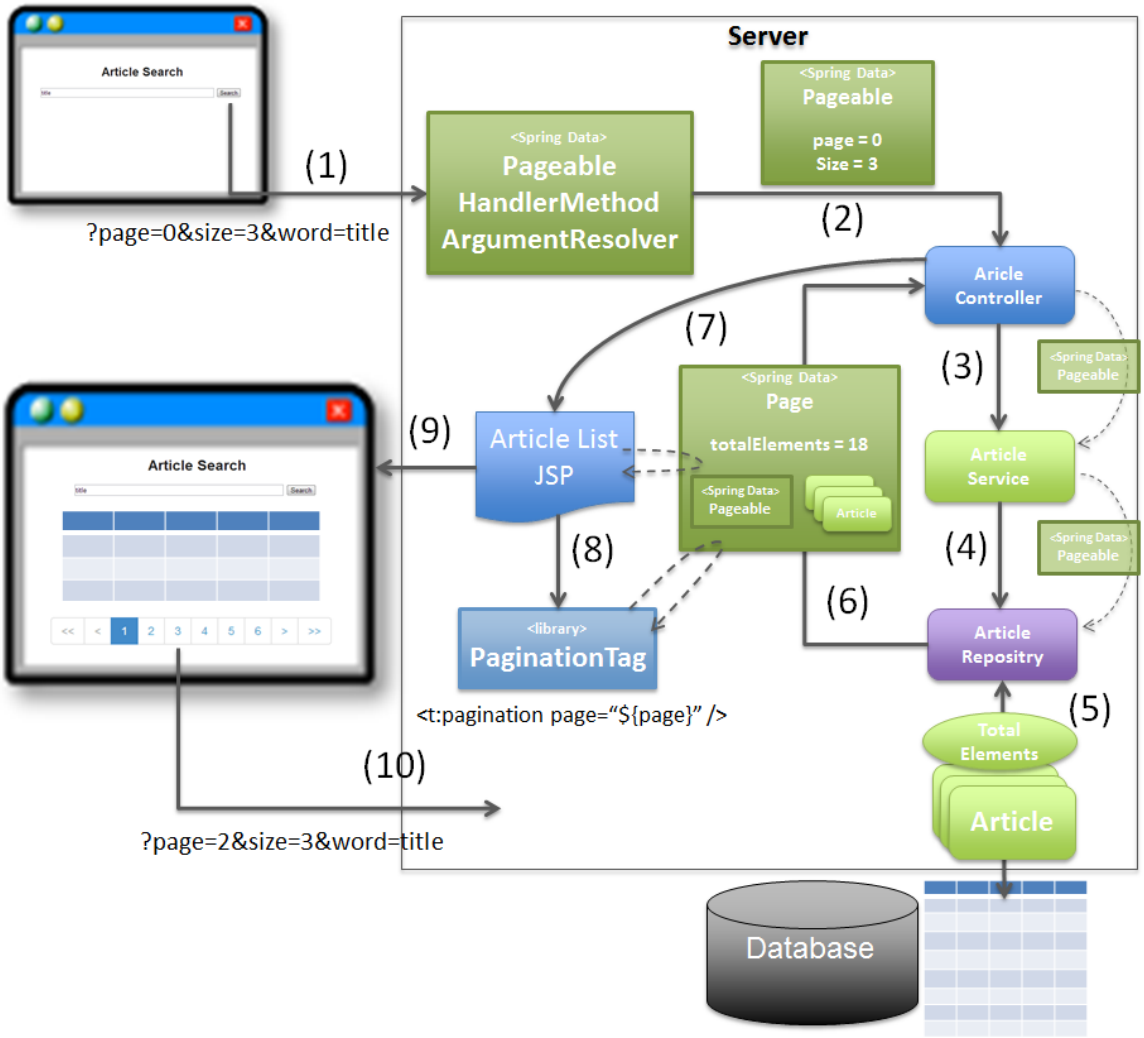
```
<t:pagination page="${page}"
    disabledHref="javascript:void(0);"
    pathTmpl="${pageContext.request.contextPath}/article/list/{page}/{size}"
    queryTmpl="sort={sortOrderProperty},{sortOrderDirection}"
    criteriaQuery="${f:query(articleSearchCriteriaForm)}" />
```

- 出力される HTML



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 の処理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。

注釈: Repository の実装について

Spring Data JPA の Repository インタフェースのメソッドを使用した場合は、(5) と (6) の処理は Spring Data JPA の機能で自動的に行われる。

Mybatis2 を使用する場合は、Java 又は SQL マッピングファイル内で実装する必要がある。具体的な実装例については、「データベースアクセス（Mybatis2 編）」を参照されたい。

5.11.2 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)	処理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。

- 処理メソッドの `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。 上記例では、`"ORDER BY publishedDate DESC, articleId DESC"` というソート条件が Query に追加される。	空の配列 (ソート項目なし)

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)

上記例では、 "ORDER BY publishedDate DESC, articleId ASC" というソート条件が Query に追加される。

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 (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。

以下にもっともシンプルな実装例を示す。

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、「データベースアクセス（JPA 編）」を参照されたい。

- Service

```
public Page<Article> searchArticle(ArticleSearchCriteria criteria,
        Pageable pageable) { // (1)

    String word = QueryEscapeUtils.toLikeCondition(criteria.getWord());

    Page<Article> page = articleRepository.findPageBy(word, pageable); // (2)

    return page; // (3)
}
```

項番	説明
(1)	ページ検索に必要な情報 (`Pageable`) を Service のメソッドの引数として受け取る。
(2)	Repository の Query 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)	Repository から返却された検索結果 (`Page` オブジェクト) を Controller に返却する。

- Repository

```
@Query("SELECT a FROM Article a WHERE a.title LIKE %:freeWord% ESCAPE '~' OR a.overview LIKE
Page<Article> findPageByFreeWord(@Param("freeWord") String word, Pageable pageable); // (4)
```

項番	説明
(4)	ページ検索に必要な情報 (`Pageable`) を Repository の Query メソッドの引数として受け取る。 返り値の型は、`Page<Entity>` とする。 上記のようなメソッドを用意すると、Spring Data JPA の機能が `Pageable` オブジェクトの状態に該当するデータを抽出して `Page` オブジェクトを返却してくれる。

Service の実装 (Mybatis2 編)

Mybatis2(TERASOLUNA DAO) 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 TERASOLUNA DAO のメソッドを呼び出す。
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、「データベースアクセス（Mybatis2 編）」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

```
<%-- ... --%>

<%-- (2) --%>
<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>

    <%-- (3) --%>
    <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
(2)	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。
(3)	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="#">&lt;&lt;</a></li>
   <li class="disabled"><a href="#">&lt;</a></li>
   <li class="active"><a href="?page=0&size=6">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で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

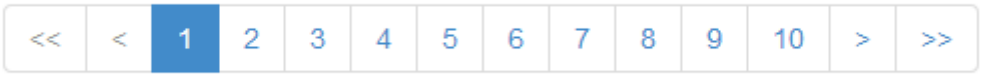
```
.pagination li {
    display: inline;
}

.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

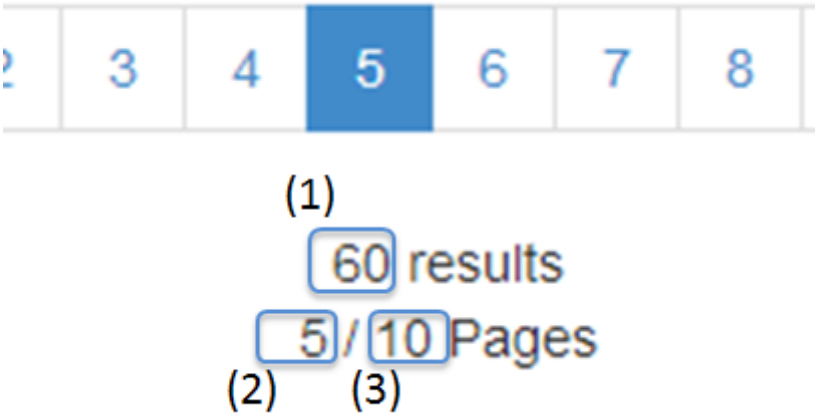
  JSP では配置した css ファイルを読み込む定義を追加する。

```
<link rel="stylesheet"
    href="${pageContext.request.contextPath}/resources/vendor/bootstrap-3.0.0/css/bootstrap.
    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



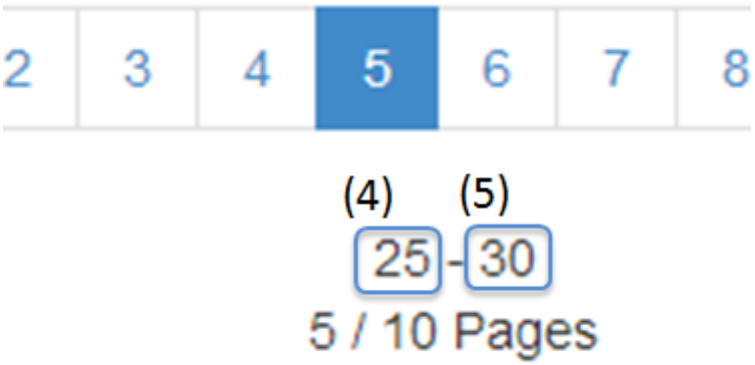
- JSP

```
<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

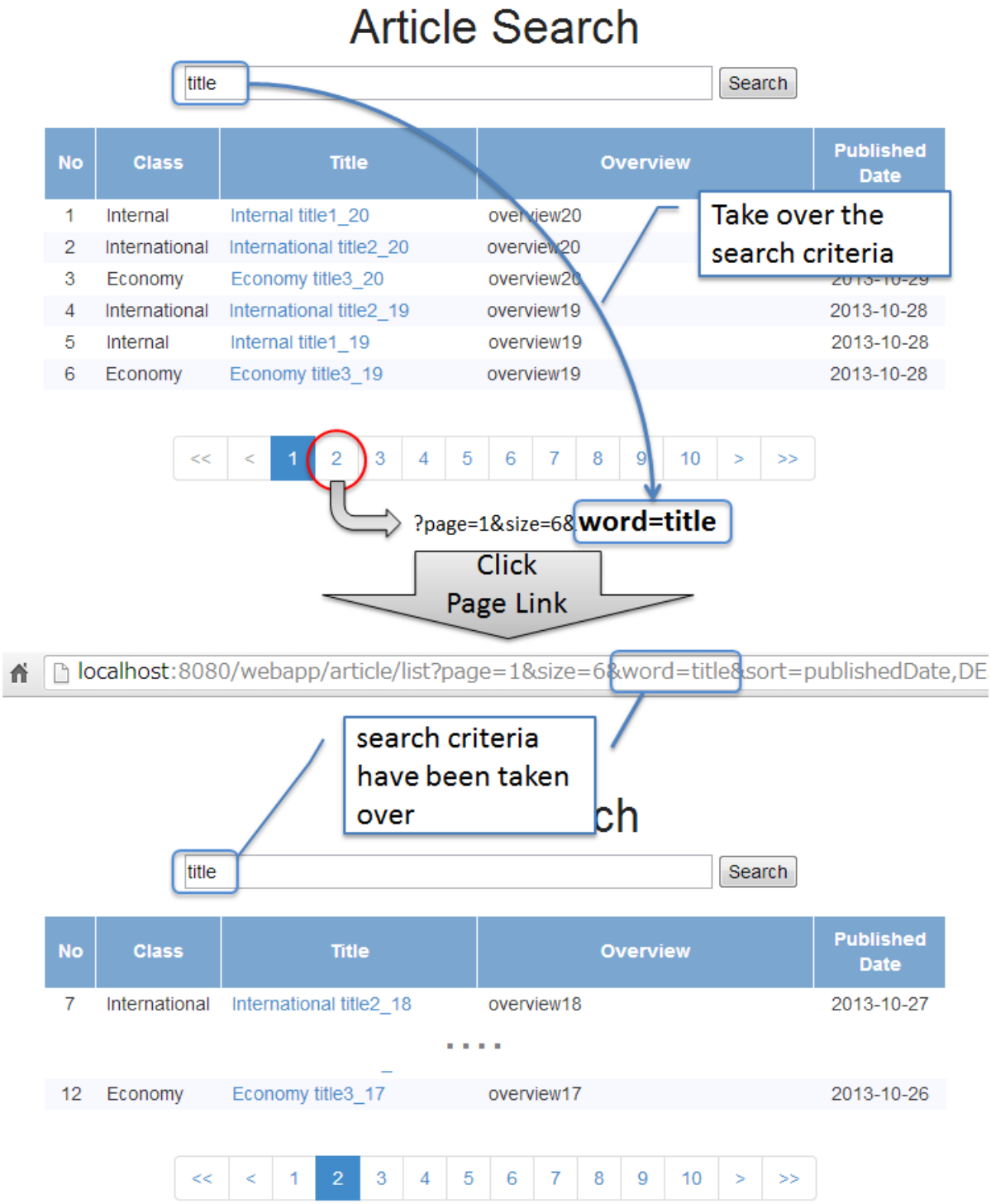
表示する数値をフォーマットする必要がある場合は、 JSTL から提供されているタグライブラリ (<fmt:formatNumber>) を使用する。

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>
```


Article Search
title
Search
No
Class
Title
Overview
Published Date
1 Internal Internal title1_20 overview20
2 International International title2_20 overview20
3 Economy Economy title3_20 overview20
4 International International title2_19 overview19 2013-10-28
5 Internal Internal title1_19 overview19 2013-10-28
6 Economy Economy title3_19 overview19 2013-10-28
Take over the search criteria
<< < 1 2 3 4 5 6 7 8 9 10 > >>
?page=1&size=6&word=title
Click Page Link
localhost:8080/webapp/article/list?page=1&size=6&word=title&sort=publishedDate,DE
search criteria have been taken over
title
Search
No
Class
Title
Overview
Published Date
7 International International title2_18 overview18 2013-10-27
12 Economy Economy title3_17 overview17 2013-10-26
<< < 1 2 3 4 5 6 7 8 9 10 > >>

```
    </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。

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}" />  <%
```

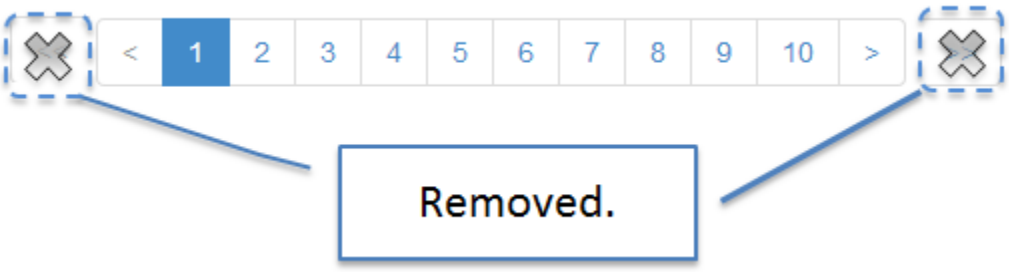
項番	説明
(1)	ページ移動時のリクエストにソート条件を引き継ぐ場合は、 `queryTmpl` を指定し、クエリ文字列にソート条件を追加する。 ソート条件を指定するためのパラメータの仕様については、「**ページ検索用のリクエストパラメータについて**」を参照されたい。 上記例の場合、 `"?page=0&size=20&sort=`**ソート項目**`,`**ソート順**`(ASC or DESC)"` がクエリ文字列となる。

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

「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"
        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

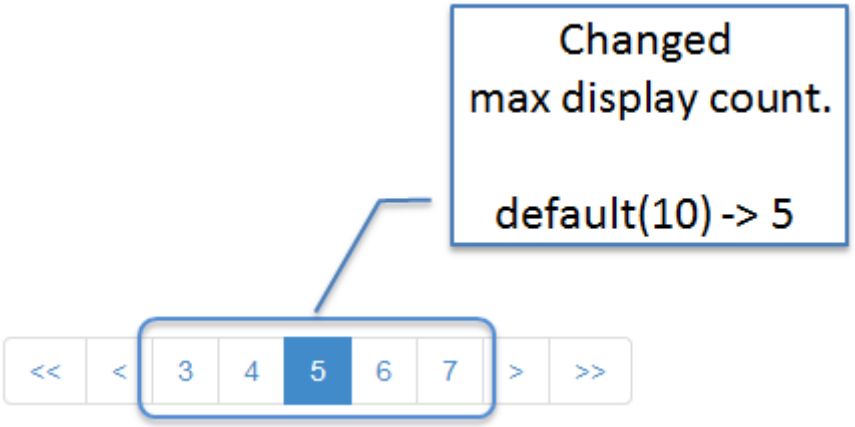
```
    .pagination .disabled {
        display: none;  /* (1) */
    }
```

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

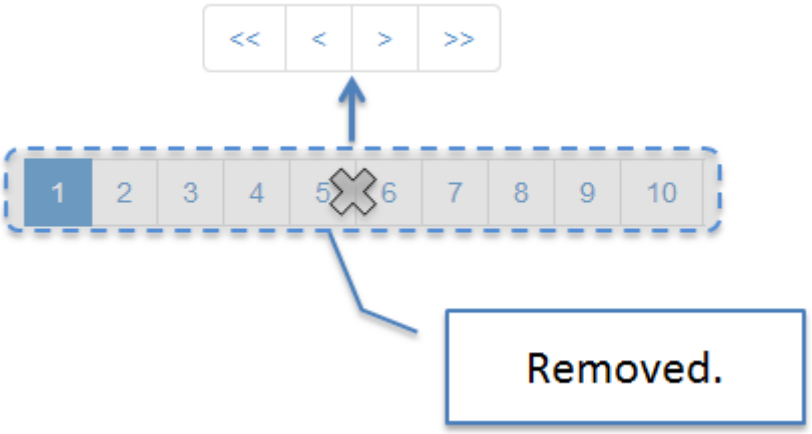
```
    <t:pagination page="${page}"
        outerElementClass="pagination"
        maxDisplayCount="5" /> <%-- (1) --%>
```

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。

5.11.3 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*
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` となる。	`"_"`

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。
(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。 上記例ではリクエストパラメータにソート条件の指定がない場合は `"ORDER BY x.lastModifiedDate DESC, x.id ASC"` というソート条件が Query に追加される。
(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。	"_"

5.12 二重送信防止

5.12.1 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。

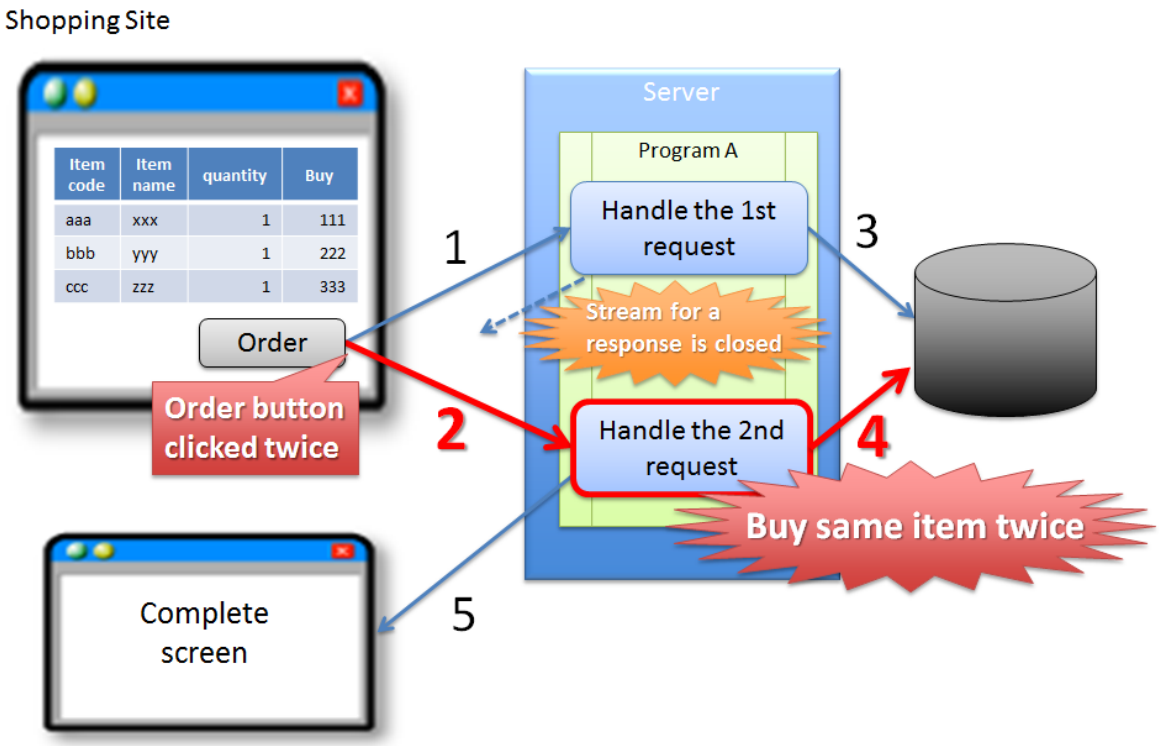
項番	操作	操作概要
(1)	更新系ボタンの二重クリック	更新処理を行うボタンを連続してクリックする。
(2)	更新処理完了後の画面の再読み込み	ブラウザの更新ボタンを使用することで、更新処理完了後の画面の再読み込みを行う。
(3)	ブラウザの戻るボタンを使用した不正な画面遷移	更新処理の完了画面からブラウザの戻るボタンを使用してページを戻し、更新処理を行うボタンを再度クリックする。

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。



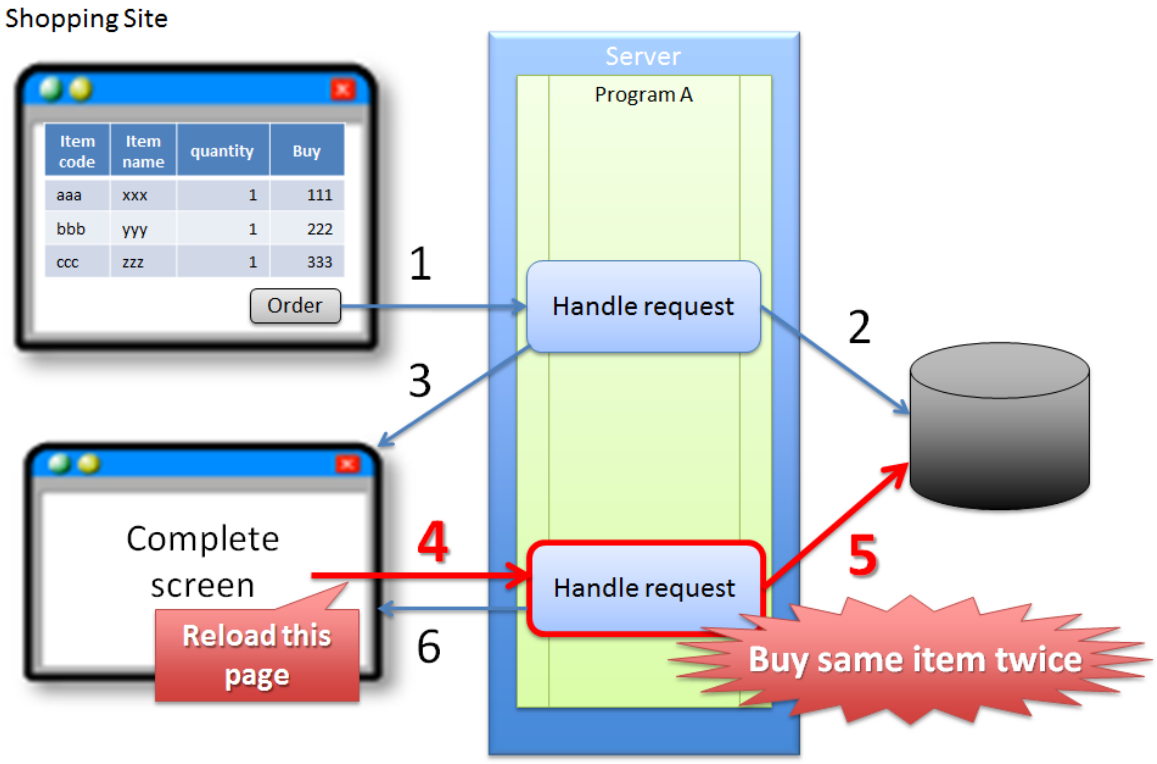
項番	説明
(1)	購買者が、商品購入画面で注文ボタンをクリックする。
(2)	(1) のレスポンスが返る前に、購買者が誤って注文ボタンをもう一度クリックする。
(3)	サーバは、(1) のリクエストで受けた商品の購入処理を DB に対して反映する。
(4)	サーバは、(2) のリクエストで受けた商品の購入処理を DB に対して反映する。
(5)	サーバは、(2) のリクエストで受けた商品の購入完了画面を応答する。

> **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。



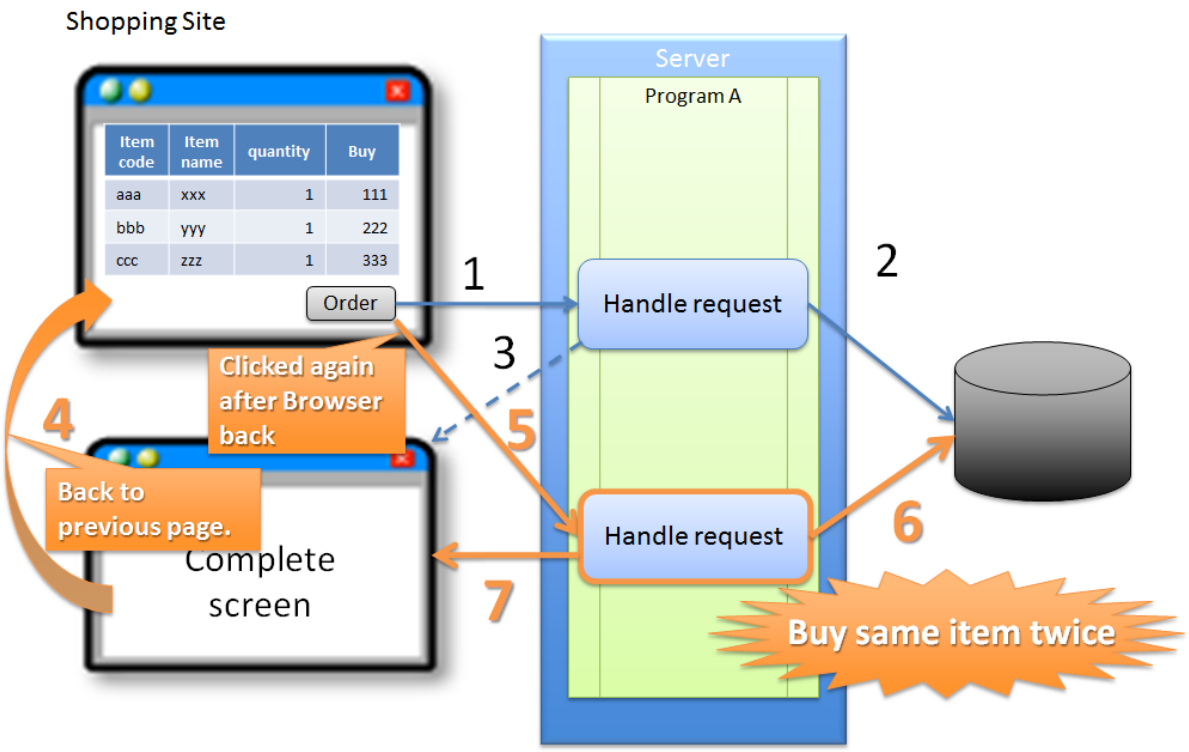
項番	説明
(1)	購買者が、商品購入画面で注文ボタンをクリックする。
(2)	サーバは、(1) のリクエストで受けた商品の購入処理を DB に対して反映する。
(3)	サーバは、(1) のリクエストで受けた商品の購入完了画面を応答する。
(4)	購買者が、誤ってブラウザのリロード機能を実行する。
(5)	サーバは、(4) のリクエストで受けた商品の購入処理を DB に対して反映する。
(6)	サーバは、(4) のリクエストで受けた商品の購入完了画面を応答する。

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。

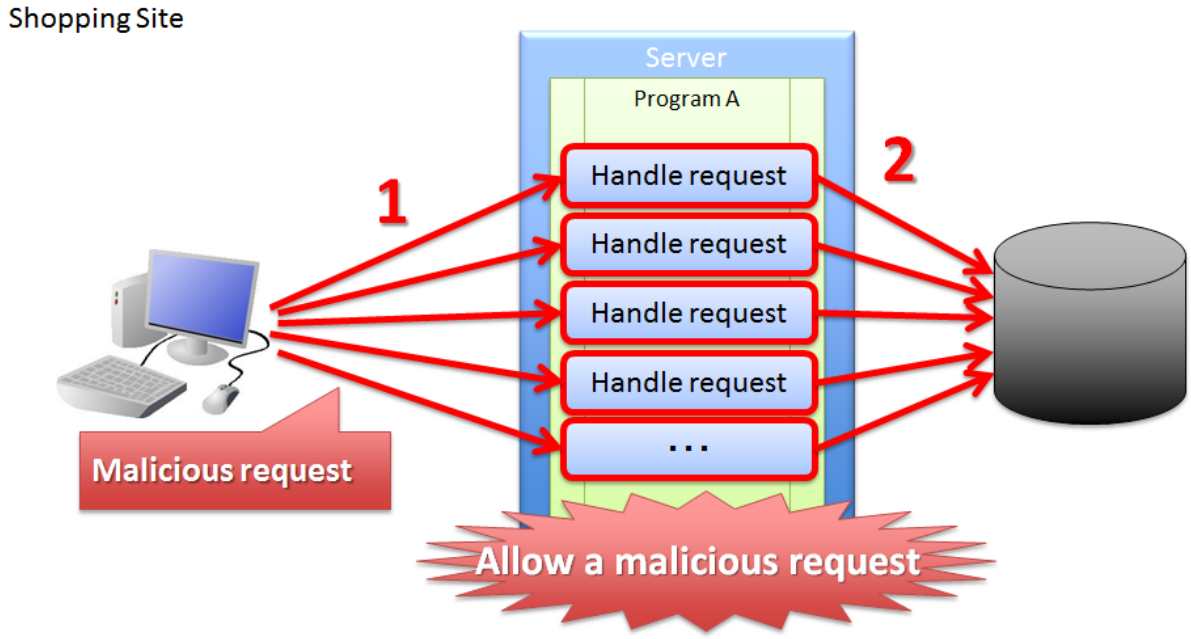
以下では、ショッピングサイトの商品購入を例として、対策を行わない場合にどのような問題が発生するのかを説明する。



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。
(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。

課題

TBD

Ajax と Web サービスでのチェック方法は、今後検討の余地あり。

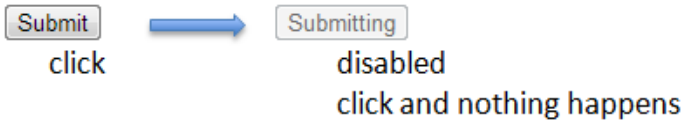
JavaScript によるボタンの 2 度押し防止について

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。

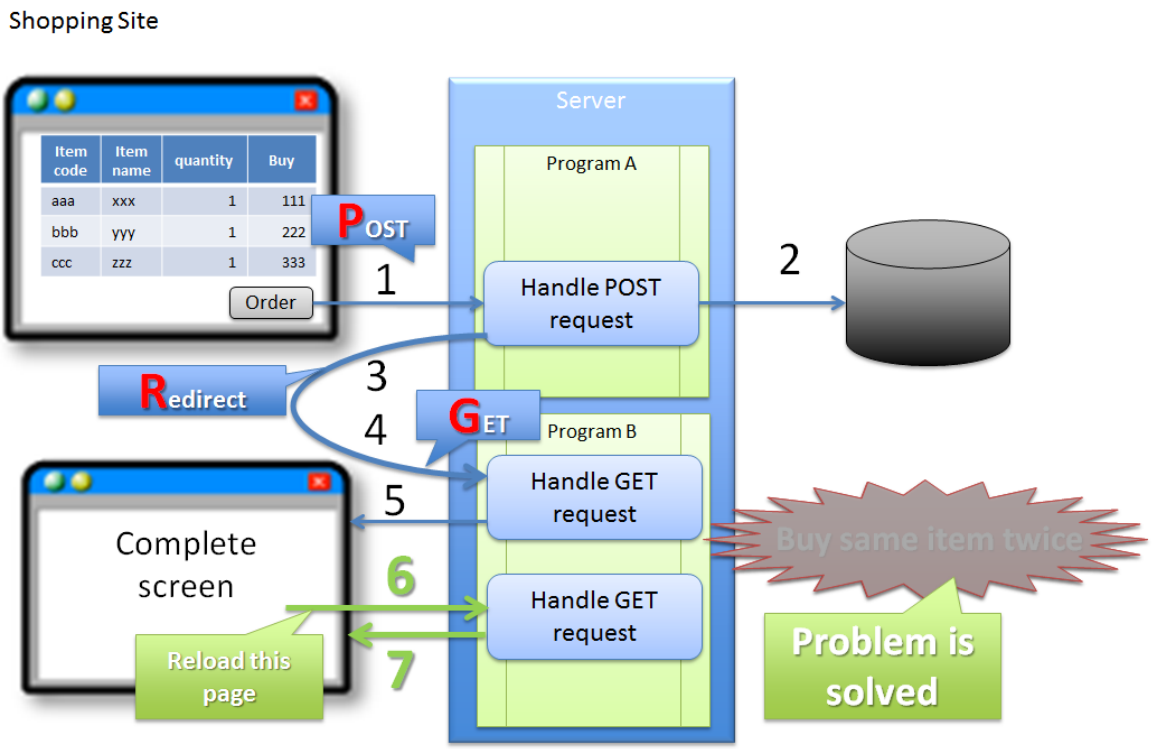
下記は、ボタンを非活性化した際のイメージとなる。



> **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った不正な画面遷移後の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、

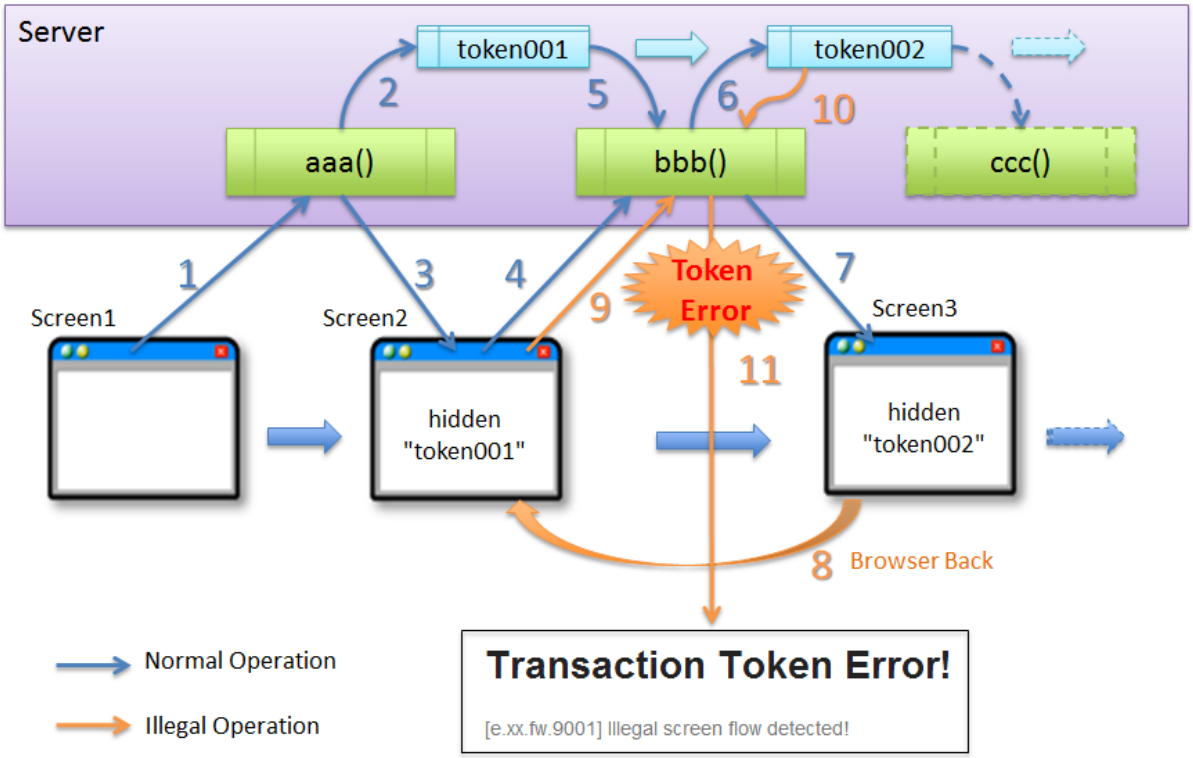
- サーバは、クライアントからクエストが来た際に、サーバ上にトランザクションを一意に識別するための値（以下、トランザクショントークン）を保持する。
- サーバは、クライアントへトランザクショントークンを引き渡す。画面を提供する Web アプリケーションの場合は、form の hidden タグを使用してクライアントにトランザクショントークンを引き渡す。
- クライアントは次のリクエストを送信する際に、サーバから引き渡されたトランザクショントークンを送る。サーバは、クライアントから受け取ったトランザクショントークンと、サーバ上で管理しているトランザクショントークンを比較する。

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。
(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。

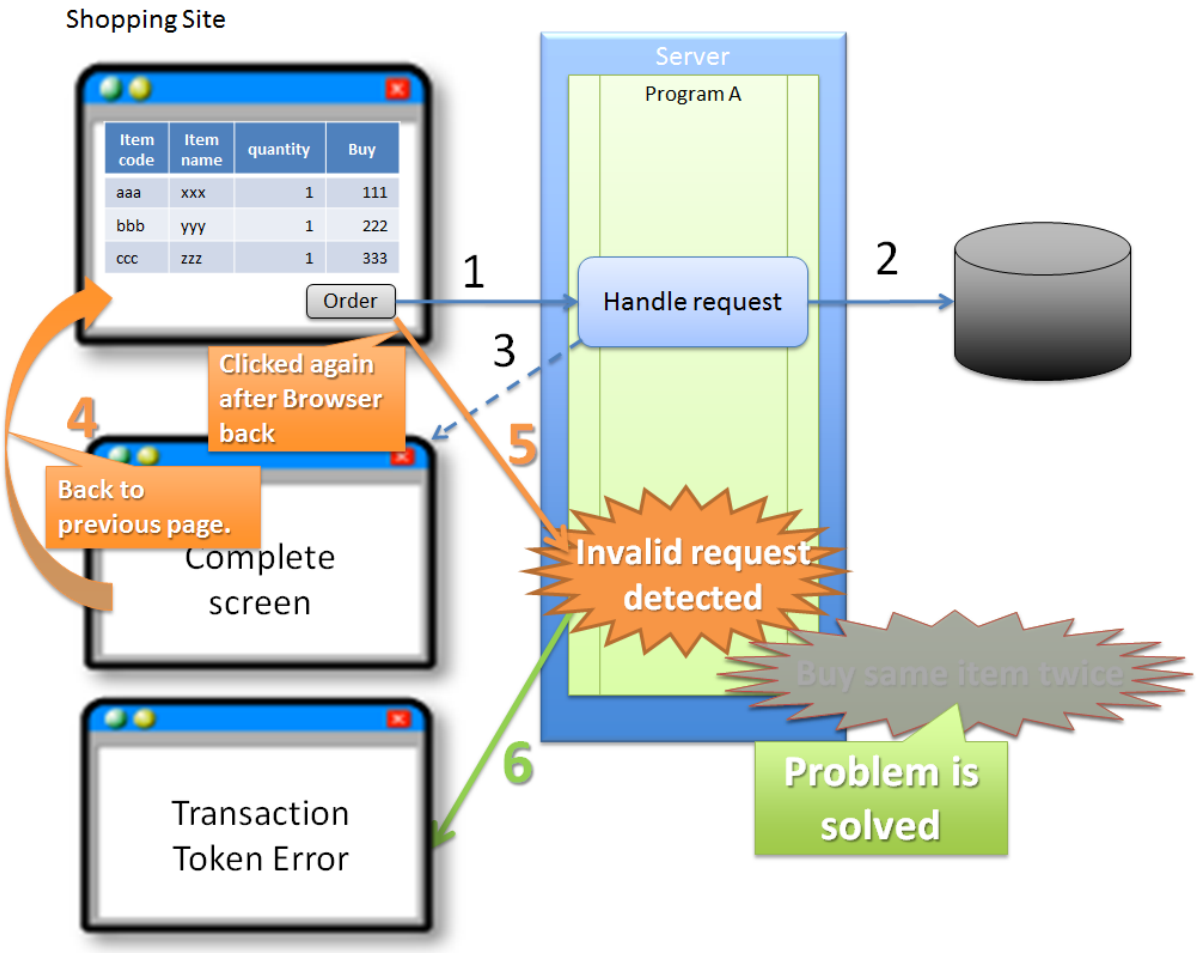
ここではブラウザの戻るボタンを例にしているが、ショートカットからの直接リクエストなどでも同様である。

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。

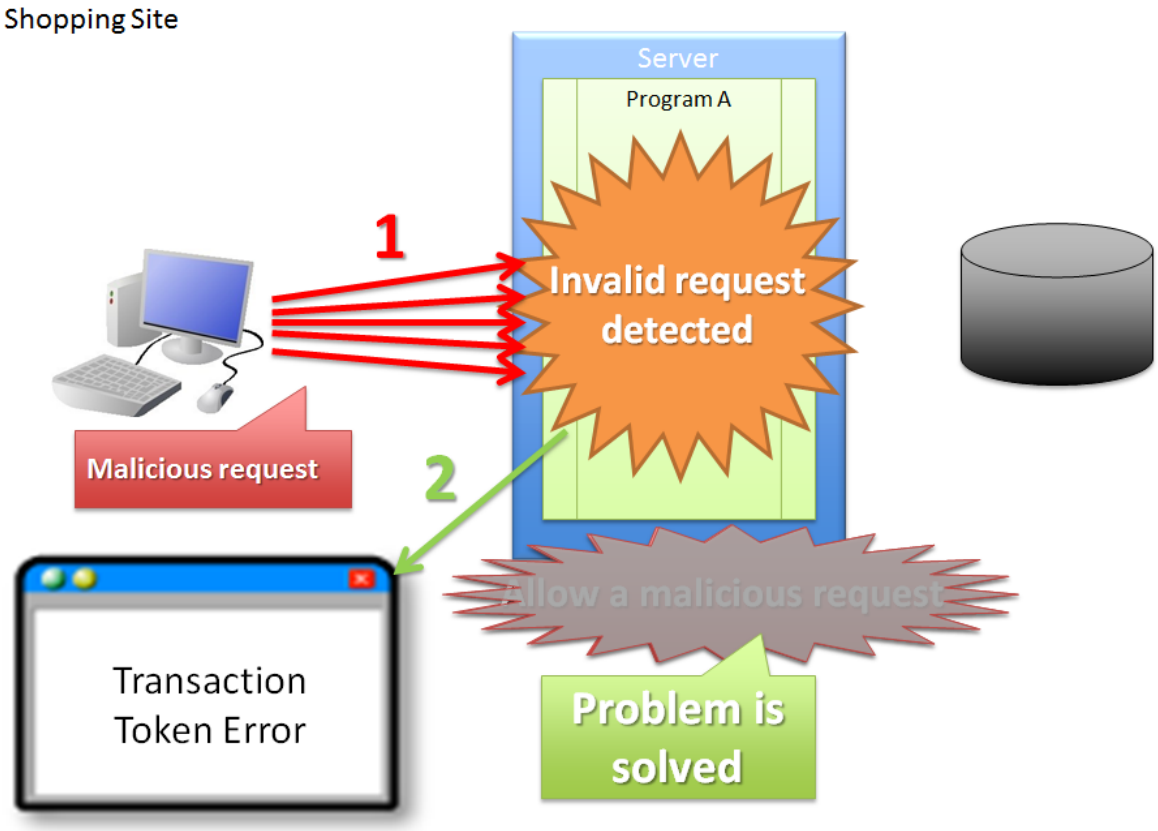
- 決められた画面遷移を行うことが求められる業務において、不正な画面遷移が行われる。
- 正規の画面遷移を伴わない不正なリクエストによって、データが更新される。
- 二重送信によって、更新処理が重複して実行される。

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。



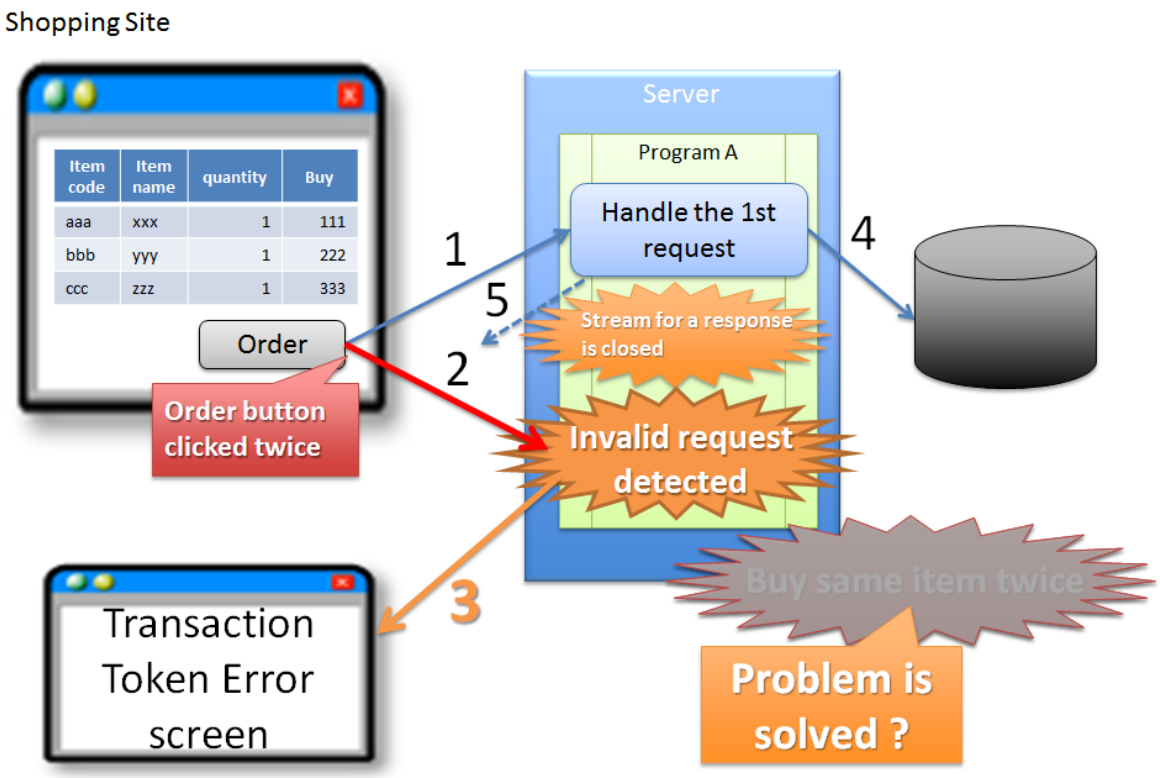
項番	説明
(1)	購買者が、商品購入画面で注文ボタンをクリックする。 サーバ上で保持しているトランザクショントークンと、クライアントから送信されたトランザクショントークンが一致するため、商品を購入する処理を実行する。 **このタイミングで、サーバ上で保持していたトランザクショントークの値が破棄され、新しいとトークン値に更新される。**
(2)	サーバは、(1) のリクエストで受けた商品の購入処理を DB に対して反映する。
(3)	サーバは、(1) のリクエストで受けた商品の購入完了画面を応答する。
(4)	購買者が、ブラウザの戻るボタンを使って購入画面を再度表示する。
(5)	購買者が、ブ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

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、新しいと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。

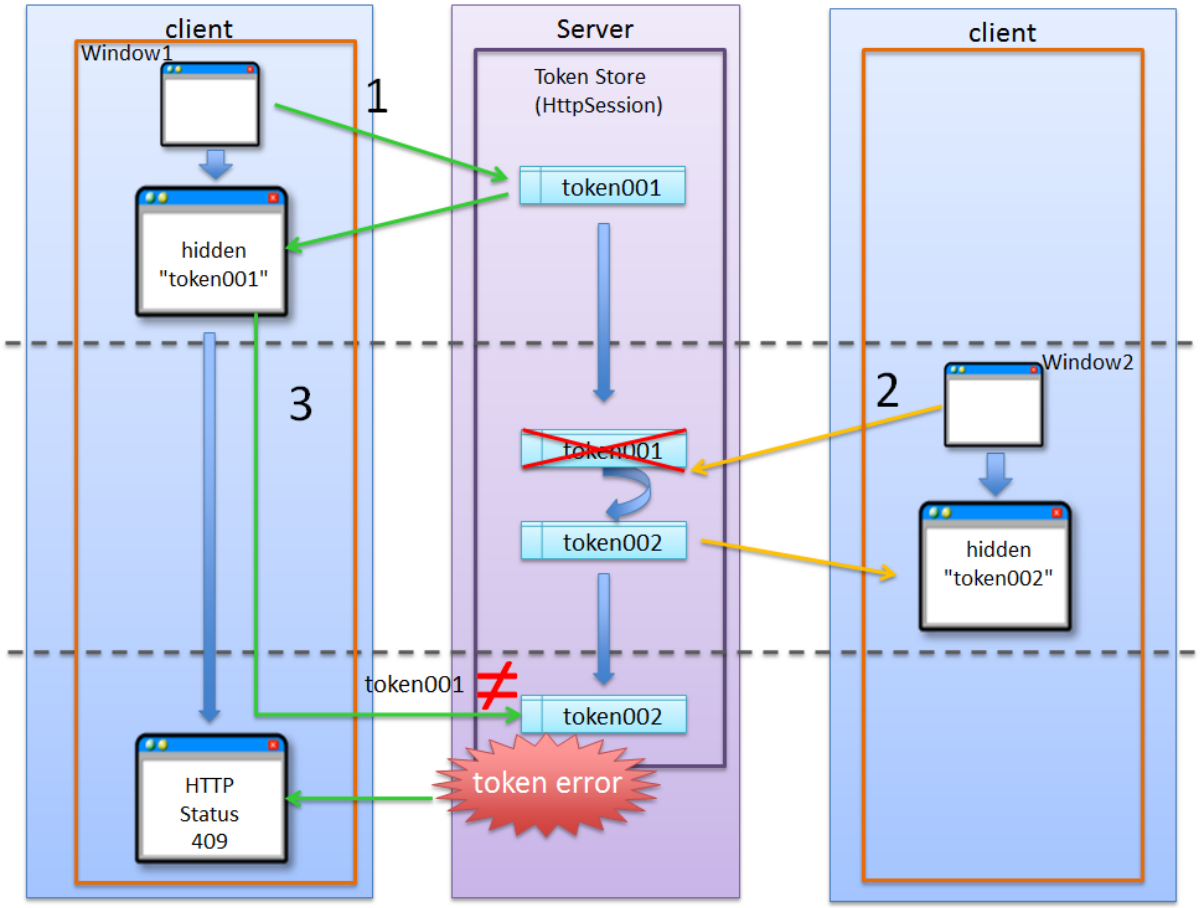
トランザクショントークンのネームスペースについて

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。



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。

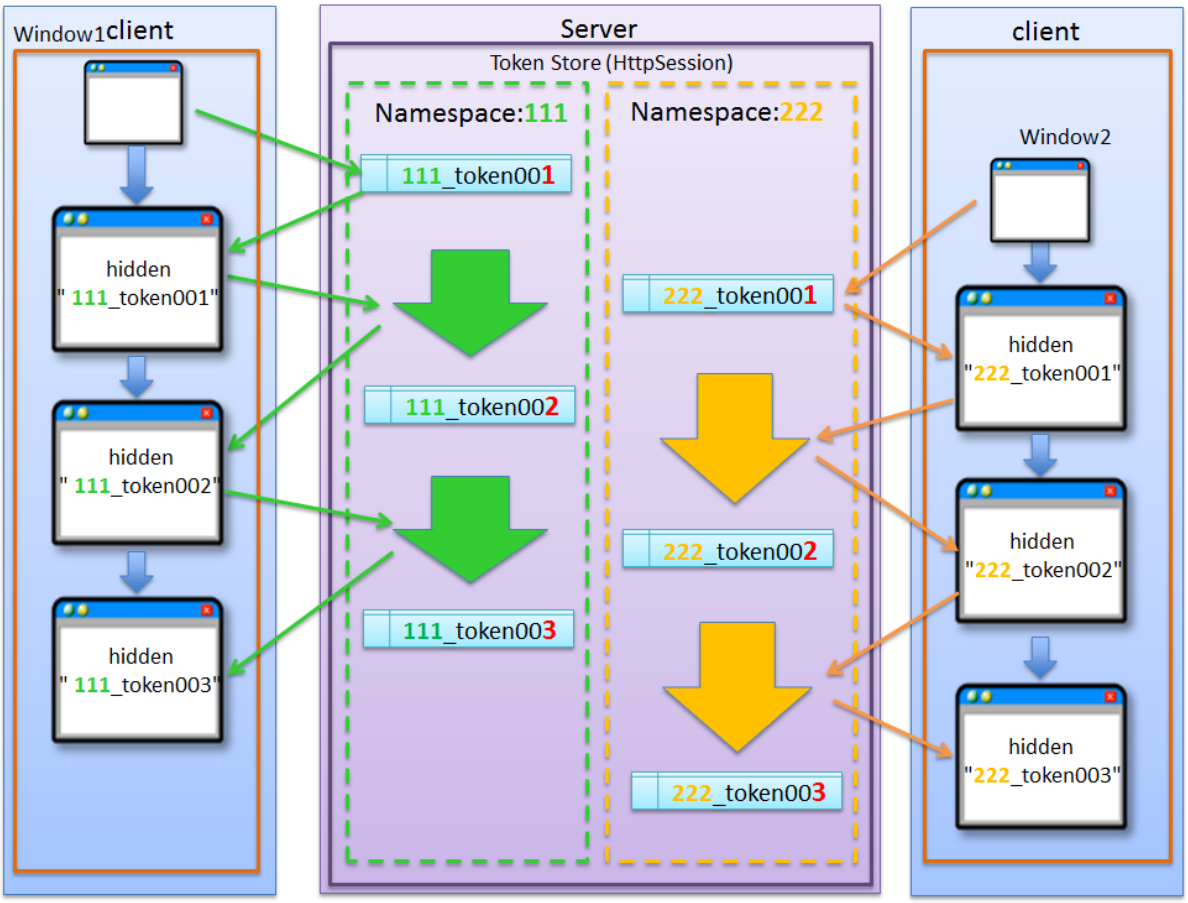
ネームスペース指定時の動作について

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。

5.12.2 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。

課題

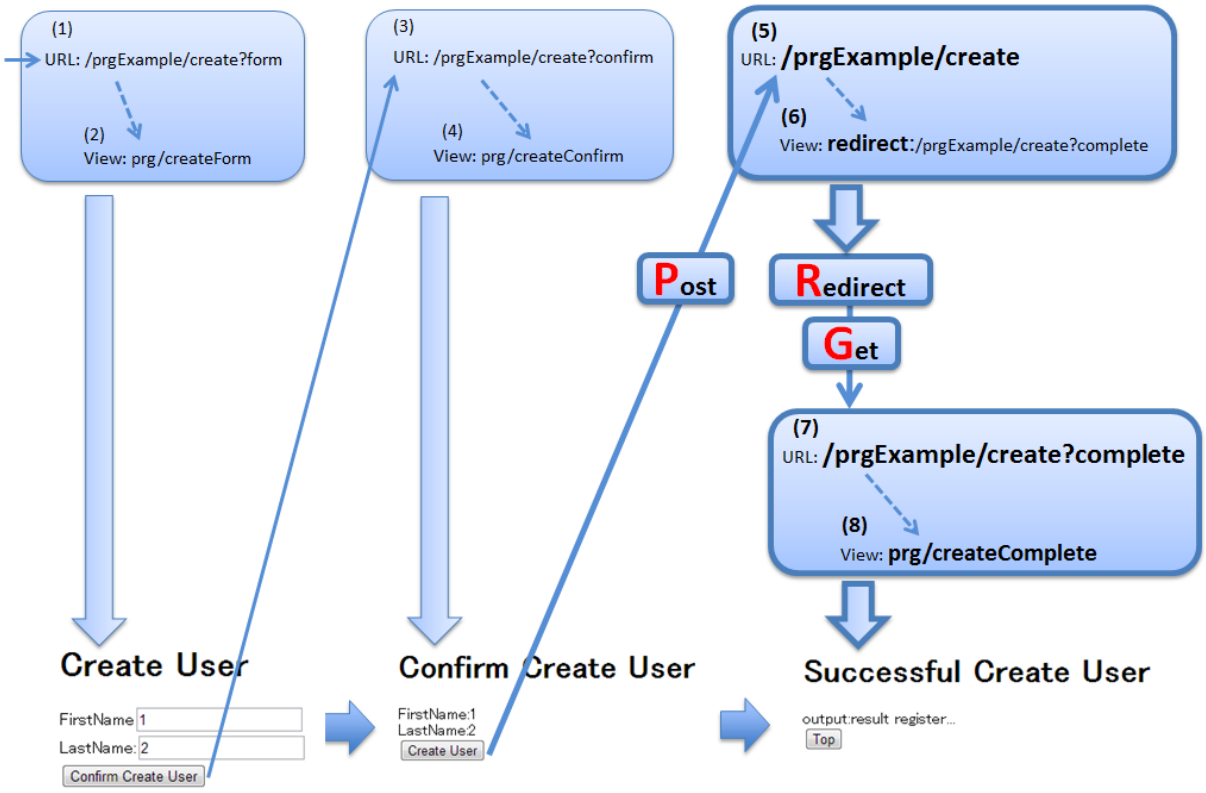
TBD

JavaScript でのチェック方法については、次版以降で詳細化する予定である。

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処理メソッド。 **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処理メソッド。 **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、処理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、処理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}/rpgExample/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="${pageContext.request.contextPath}/rpgExample/create"
        method="post"
        modelAttribute="postRedirectGetForm">
        FirstName:${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送る。**

- createComplete.jsp

```
    <h1>Successful Create User Completion</h1>
    <div id="prgForm">
      <form:form
        action="${pageContext.request.contextPath}/rpgExample/create"
        method="get" modelAttribute="postRedirectGetForm">
        output:${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 クラスの処理メソッドに対して、 `@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。

TABLE 5.22 `@TransactionTokenCheck` アノテーションパラメタ一覧

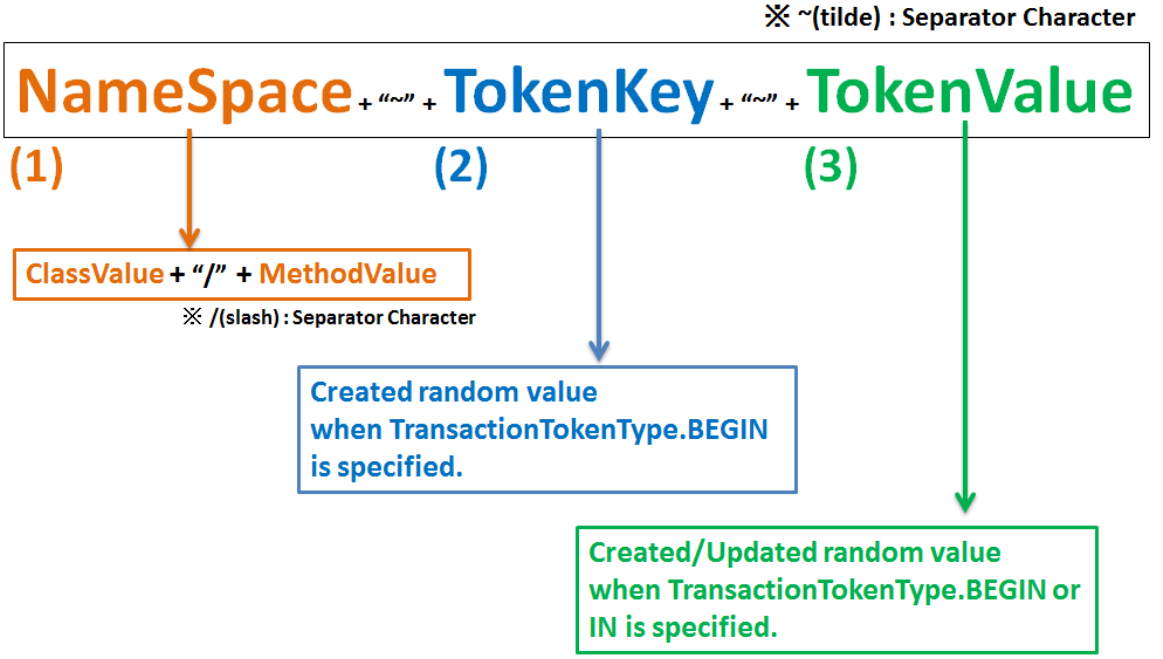
項番	属性名	内容	default	例
1.	value	任意文字列。NameSpace として使用される。	無	value = “create” 引数が 1 つのみの場合は、”value =” 部分は省略できる。
2.	type	**BEGIN** トランザクショントークンを作成し、新たなトランザクションを開始する。 **IN** トランザクショントークンの妥当性チェックを実施する。 リクエストされたトークン値とサーバ上で管理しているトークン値が一致している場合は、トランザクショントークンのトークン値を更新する。	IN	type = TransactionTokenType.BEGIN type = TransactionTokenType.IN

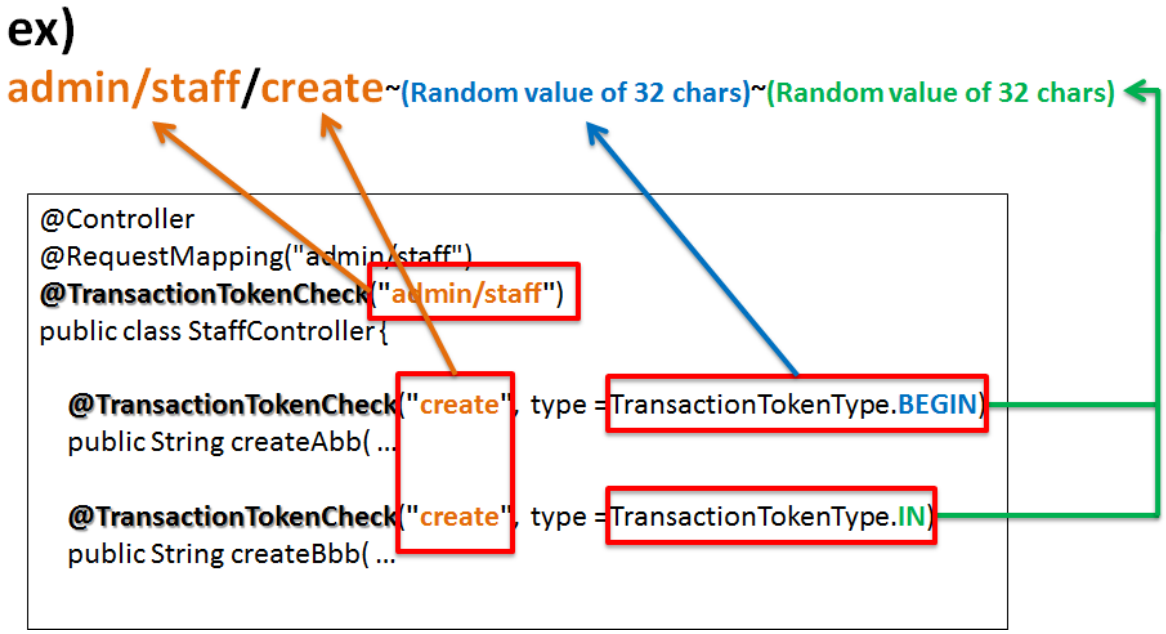
注釈: value 属性に設定する値は、`@RequestMapping` アノテーションの value 属性の設定値と、同じ値を設定することを推奨する。

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。
(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処理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処理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。

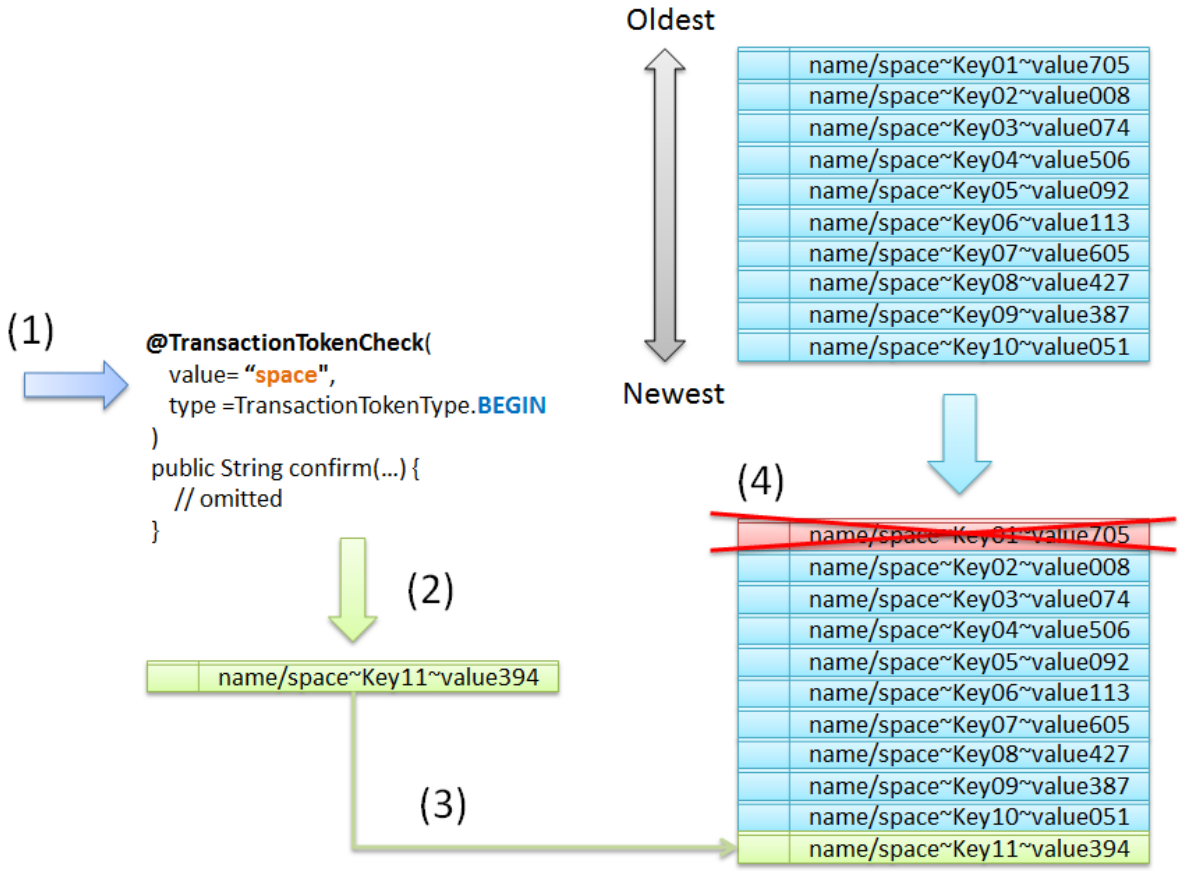
項番	ライフサイクル制御	説明
(1)	トークンの生成	`@TransactionTokenCheck` アノテーションの type 属性に `TransactionTokenType.BEGIN` が指定されたメソッドの処理が終了したタイミングで新たなトークンが生成され、トランザクションが開始される。
(2)	トークンの更新	`@TransactionTokenCheck` アノテーションの type 属性に `TransactionTokenType.IN` が指定されたメソッドの処理が終了したタイミングでトークン (TokenValue) が更新され、トランザクションが継続される。
(3)	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。

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のデフォルト 10 個である。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) -->
            <mvc:exclude-mapping path="/**/*.html" /> <!-- (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.TransactionTokenRequestDat
                <!-- 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と HTML 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<fomr: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処理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、*CSRF* **トークンを明示的に埋め込む方法**を参照されたい。

注釈: <form:form>タグでを使用すると、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、*CSRF* **トークンを自動で埋め込む方法**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: ユースケース毎に NameSpace を割り振ることで、各ユースケース毎にトランザクショントークンのチェックを行うことが出来る。

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処理メソッド。 **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処理メソッド。 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処理メソッド。 **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処理メソッド。 **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処理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

@SessionAttribute 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を 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: @SessionAttribute 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。

5.12.3 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

以下の設定を追加することで、Controller の処理メソッドの引数として org.terasoluna.gfw.web.token.transaction.TransactionTokenContext を受け取り、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ことができる。

- spring-mvc.xml

```
<mvc:annotation-driven>
  <mvc:argument-resolvers>
    <!-- (1) -->
    <bean
      class="org.terasoluna.gfw.web.token.transaction.TransactionTokenContextHandlerMethodAr
  </mvc:argument-resolvers>
</mvc:annotation-driven>
```

項番	説明
(1)	<mvc:argument-resolvers>要素に、Controller のメソッドの引数として、プログラマス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ためのオブジェクト (TransactionTokenContext) を引き渡すためのクラス (TransactionTokenContextHandlerMethodArgumentResolver) を設定をする。 プログラマス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必要がない場合は、本設定は不要である。

注釈: 使用されなくなったトランザクショントークンは、1 つの NameSpace で保持することが出来る上限値を超えた時点で自動的に破棄されていくため、基本的には、本設定は不要である。

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/**" />
        <mvc:exclude-mapping path="/**/*.html" />
        <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。

5.12.4 Appendix

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を 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。

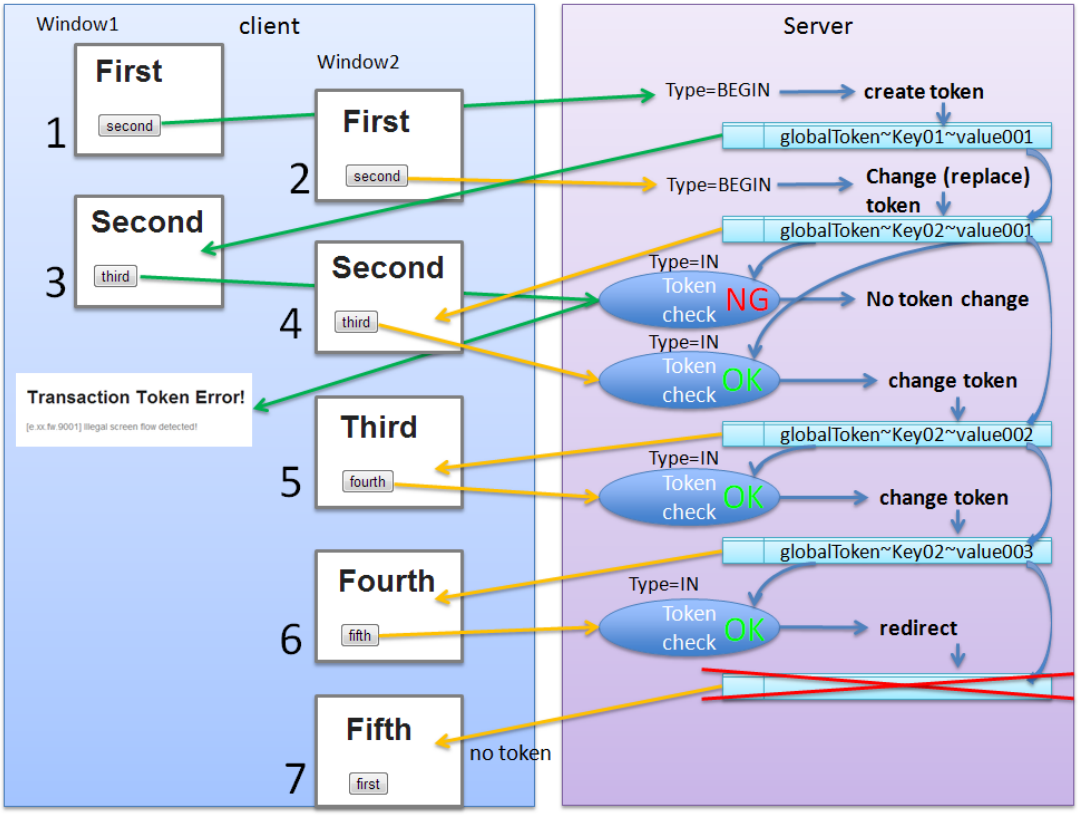


出力された HTML を確認すると、

- 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。
(5)	(4) と同様。

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。
(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

5.13 国際化

5.13.1 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、複数言語の切替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。

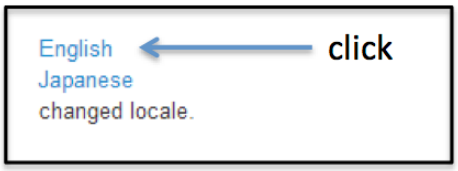
国際化するためには、以下の対応が必要となる。

- 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。

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。

5.13.2 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

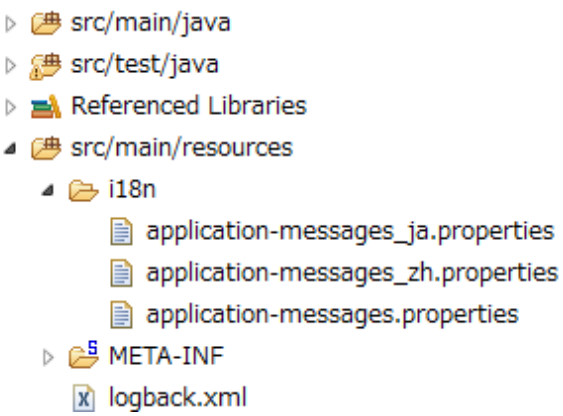
```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>i18n/application-messages</value>  <!-- (1) -->
        </list>
    </property>
</bean>
```

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) -->
```

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。

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。

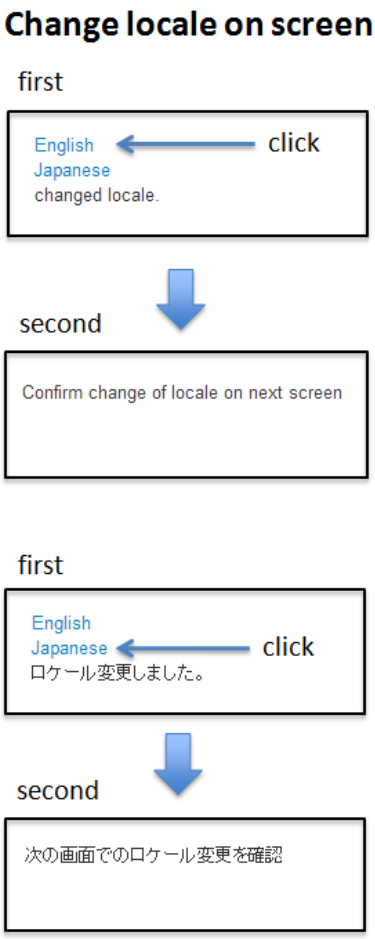
LocaleChangeInterceptor は、リクエストパラメータに指定された Locale の値を、org.springframework.web.servlet.LocaleResolver の API を使用してサーバ又はクライアントに保存するためのインタセプターである。

使用する LocaleResolver の実装クラスを、以下の表から選択する。

TABLE 5.23 **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/**" />
    <mvc:exclude-mapping path="/**/*.html" />
    <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。

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。 **paramName プロパティを省略した場合、”locale” が設定される。**“リクエスト URL?locale=xx” で **使用可能** となる。

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">  <!
    <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)	リクエストパラメータで Locale を指定しない場合、`defaultLocale` プロパティに指定された Locale が有効になる。この場合、`HttpServletRequest#getLocale` での取得値が有効になる。

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">  <!-
    <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)	Locale を指定しない場合、`defaultLocale` プロパティに指定された Locale が有効になる。この場合、`HttpServletRequest#getLocale` での取得値が有効になる。
(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。

5.14 コードリスト

5.14.1 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を行う機能

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 4 種類のコードリスト実装を提供している。

TABLE 5.24 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.i18n.JavaUtilLocaleCodeList	java.util.Localeに応じたコードリストを使用する。	NO

上記コードリストのインターフェースについて、共通ライブラリに org.terasoluna.gfw.common.codelist.CodeList を提供している。

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。

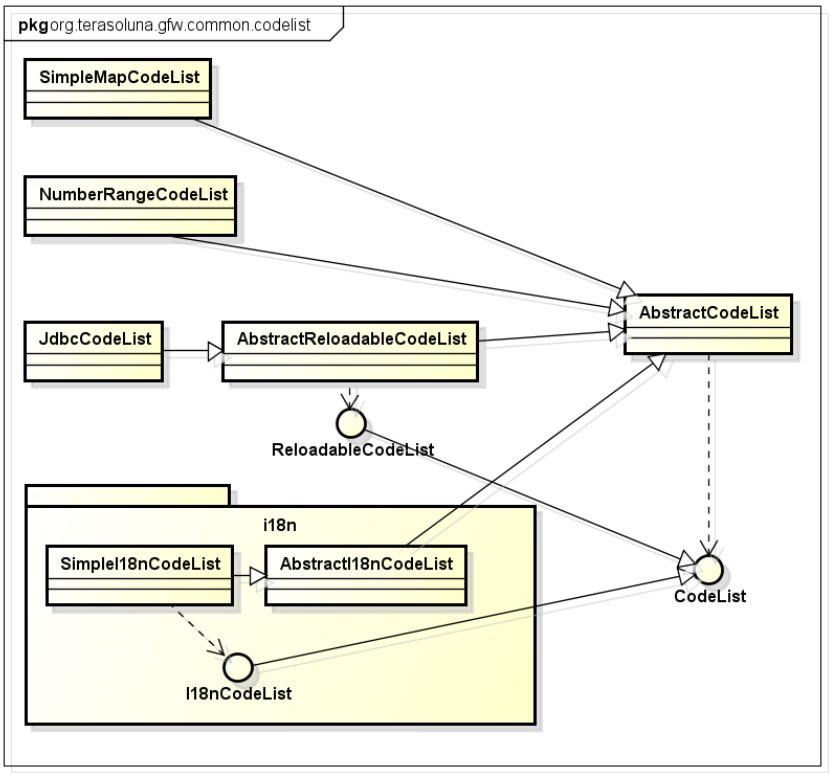


図 5.44 **Picture - Image of codelist class diagram**

5.14.2 How to use

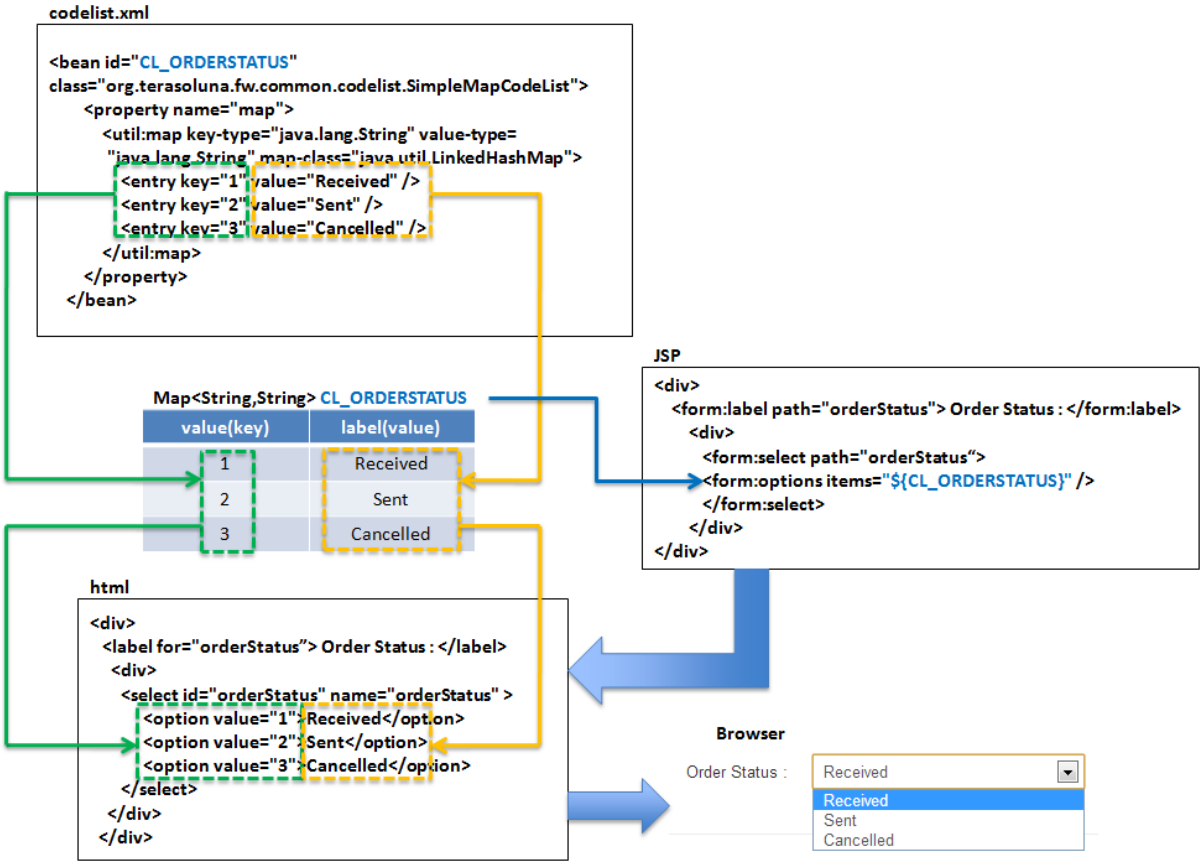
本項では、各種コードリストを使用する上での設定や実装方法を記述する。

- *SimpleMapCodeList* の使用方法
- *NumberRangeCodeList* の使用方法
- *JdbcCodeList* の使用方法
- *SimpleI18nCodeList* の使用方法
- 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 で使用可能となる。 `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 JSP で使用可能となる。

jsp の実装例

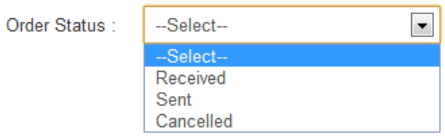
```
<form:select path="orderStatus">
  <form:option value="" label="--Select--" /> <!-- (4) -->
  <form:options items="${CL_ORDERSTATUS}" /> <!-- (5) -->
</form:select>
```

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。

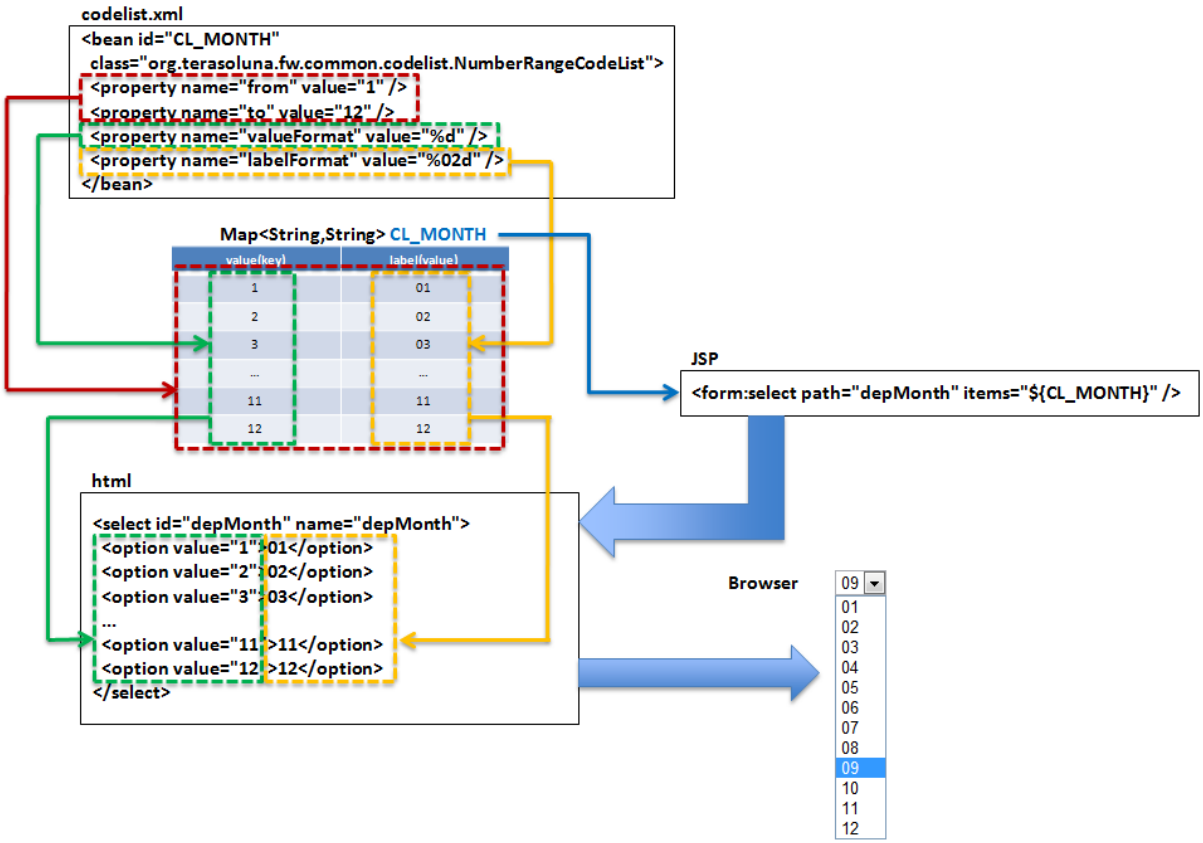
```
import javax.inject.Named;

import org.terasoluna.fw.common.codelist.CodeList;

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

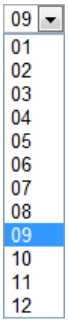
```
<form:select path="depMonth" items="${CL_MONTH}" />
```

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

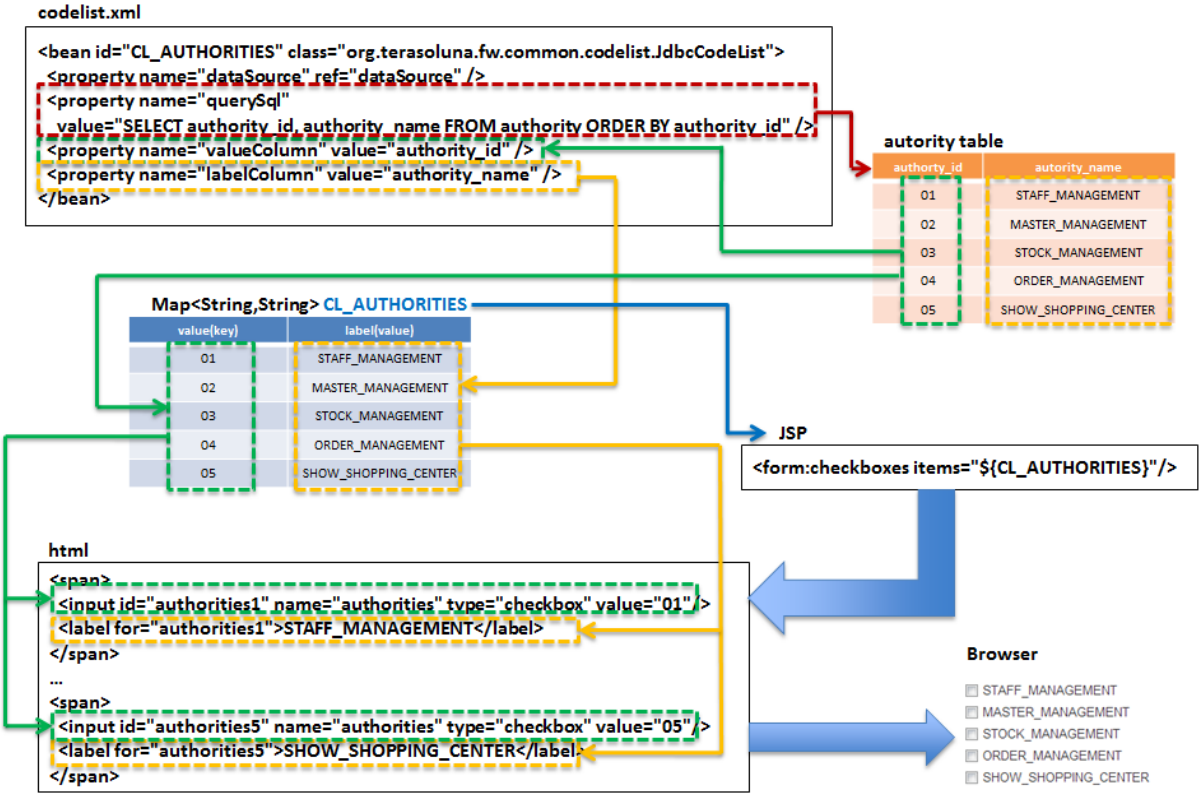
設定例の詳細は、前述した *Java* **クラスでのコードリスト使用** を参照されたい。

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。このリストはキャッシュされる。

また、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="CL_AUTHORITIES" class="org.terasoluna.gfw.common.codelist.JdbcCodeList"> <!-- (1) -->
    <property name="dataSource" ref="dataSource" />
    <property name="querySql"
        value="SELECT authority_id, authority_name FROM authority ORDER BY authority_id" /> <!--
    <property name="valueColumn" value="authority_id" /> <!-- (3) -->
    <property name="labelColumn" value="authority_name" /> <!-- (4) -->
</bean>
```

項番	説明
(1)	JdbcCodeList クラスを bean 定義する。
(2)	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。
(3)	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。
(4)	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

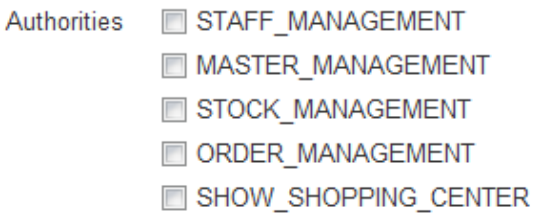
```
<form:checkboxes items="${CL_AUTHORITIES}"/>
```

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



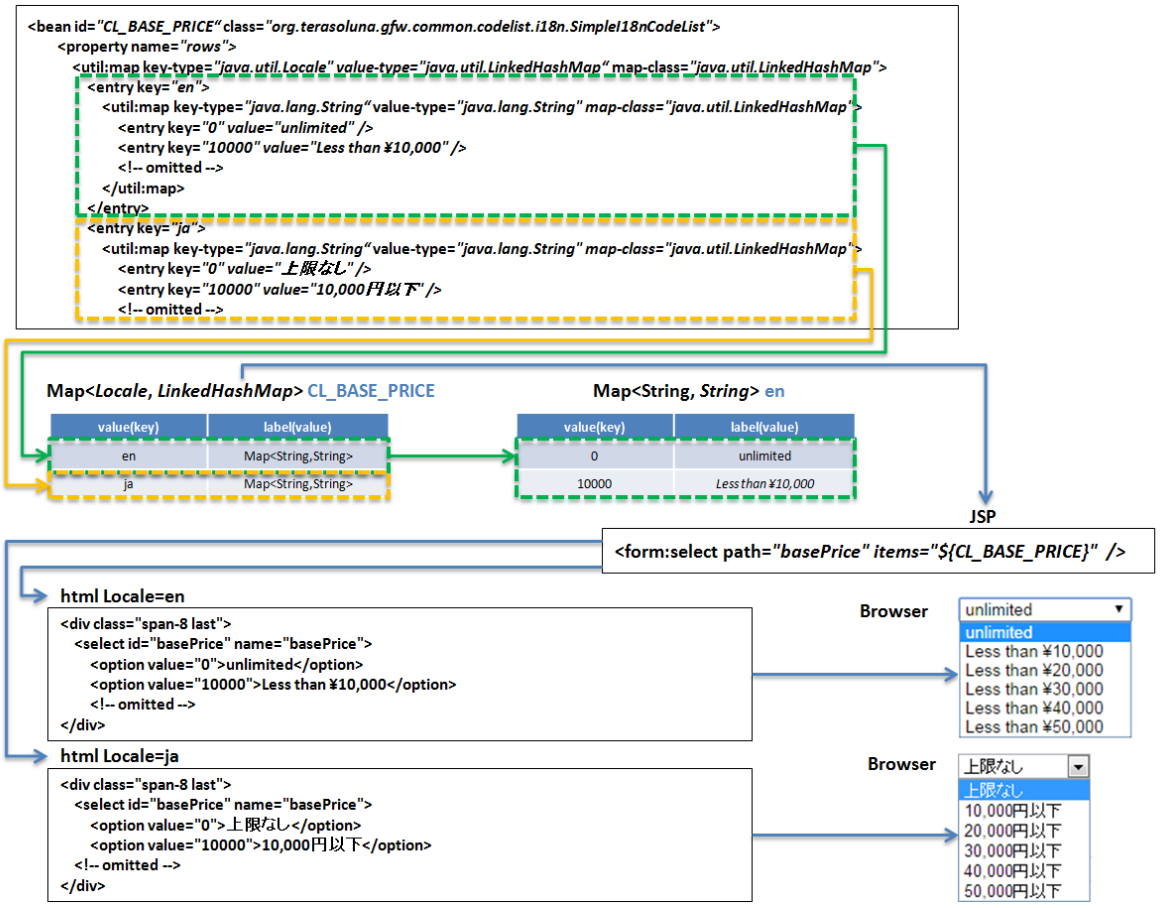
Java クラスでのコードリスト使用

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。

SimpleI18nCodeList の使用方法

`org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList` 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。

SimpleI18nCodeList のイメージ



コードリスト設定例

`SimpleI18nCodeList` 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の例に上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 th \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。

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.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.JdbcCodeList">  <!-- (4) -->
    <property name="dataSource" ref="dataSource" />
    <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" class="org.terasoluna.gfw.common.codelist.JdbcCodeList">  <!-- (5) -->
    <property name="dataSource" ref="dataSource" />
    <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)	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。
(5)	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

警告: 現時点で SimpleI18nCodeList は reloadable に対応していない。SimpleI18nCodeList が参照している JdbcCodeList (reloadable な CodeList) をリロードしても、 SimpleI18nCodeList には反映されないことに注意。もし、reloadable に対応したい場合は独自実装する必要がある。実装方法については、 **コードリストを独自カスタマイズする方法** を参照されたい。

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。

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}" />
```

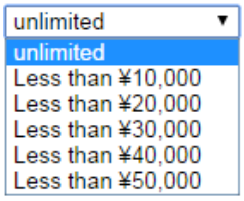
出力 HTML lang=en

```
<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>
```

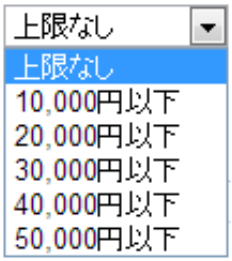
出力 HTML lang=ja

```
<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。

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 = Does not exist in {codeListId}
```

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。

ちなみに: @ExistInCodeList の入力チェックでサポートしている型は、 String または Character のみである。そのため、 @ExistInCodeList をつけるフィールドは意味的に整数型であっても、String で定義する必要がある。(年・月・日等)

5.14.3 How to extend

JdbcCodeList の読み込む件数が大きい場合

JdbcCodeList の読み込む件数が大きい (数百) 場合、Web アプリの起動に時間が掛かる。

原因は、DB 問い合わせ時に全件取得することがあり、DB からリストを取得する時間がかかってしまうためである。(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。)

この問題は、fetchSize を適切な値に指定することで解決できる。fetchSize を変更するには org.springframework.jdbc.core.JdbcTemplate の fetchSize を設定する必要がある。

以下に実装例を示す。

bean 定義ファイル (xxx-infra.xml) の定義

```
<bean id="jdbcTemplateForCodeList" class="org.springframework.jdbc.core.JdbcTemplate" > <!-- (1)
    <property name="dataSource" ref="dataSource" />
    <property name="fetchSize" value="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" />
    <property name="valueColumn" value="authority_id" />
    <property name="labelColumn" value="authority_name" />
</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。適切な値を設定すること。
(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。

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、 `org.terasoluna.gfw.common.codelist.ReloadableCodeList` インタフェースを用意している。上記インタフェースを実装したクラスは、refresh メソッドを実装しており、refresh メソッドを呼ぶことでコードリストの更新が可能となる。JdbcCodeList は、ReloadableCodeList インター

フェースを実装しているため、コードリストの更新ができる。

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 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}"/>
</task:scheduled-tasks>

<bean id="CL_AUTHORITIES" class="org.terasoluna.gfw.common.codelist.JdbcCodeList">
    <property name="dataSource" ref="dataSource" />
    <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。 http://static.springsource.org/spring/docs/3.2.16.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" class="org.terasoluna.gfw.common.codelist.JdbcCodeList">
    <property name="dataSource" ref="dataSource" />
    <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 CodeListContoller {

    @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。

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.Reloadab`
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

```
@Component("CL_YEAR") // (1)
public class DepYearCodeList extends AbstractCodeList { // (2)

    @Inject
    DateFactory dateFactory; // (3)

    @Override
    public Map<String, String> asMap() {  // (4)
        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)	`@Component` で、コードリストをコンポーネント登録する。 Value に `"CL_YEAR"` を指定することで、bean 定義で設定したコードリストインターセプトによりコードリストをコンポーネント登録する。
(2)	`org.terasoluna.gfw.common.codelist.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。
(3)	システム日付の Date クラスを作成する `org.terasoluna.gfw.common.date.DateFactory` をインジェクトしている。 `DateFactory` を利用して今年と来年の年を取得することができる。 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。
(4)	`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。

jsp の実装例

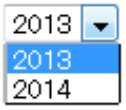
```
<form:select path="mostRecentYear" items="${CL_YEAR}" /> <!-- (5) -->
```

項番	説明
(5)	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。

5.14.4 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、 *SimpleI18nCodeList* **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

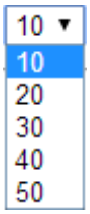
```
<form:select path="quantity" items="${CL_BULK_ORDER_QUANTITY_UNIT}" />
```

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。

5.15 Ajax

5.15.1 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。

課題

TBD

クライアント側の実装方法などについては、次版以降で詳細化する予定である。

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。

5.15.2 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 形式のデータを 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 の処理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.MappingJacksonHttpMessageConverter	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 1.x 系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。

警告: XXE(XML External Entity) Injection 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) Injection 対策を行う必要がある。terasoluna-gfw-web 1.0.1.RELEASE 以上では、XXE Injection 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。

terasoluna-gfw-web 1.0.0.RELEASE を使用している場合は、XXE Injection 対策が行われていない Spring MVC(3.2.4.RELEASE) に依存しているため、Spring-oxm から提供されているクラスを使用すること。

- spring-mvc.xml

```
<!-- (1) -->
<bean id="xmlMarshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <property name="packagesToScan" value="com.examples.app" /> <!-- (2) -->
</bean>

<!-- ... -->

<mvc:annotation-driven>

    <mvc:message-converters>
        <!-- (3) -->
        <bean class="org.springframework.http.converter.xml.MarshallingHttpMessageConv
            <property name="marshaller" ref="xmlMarshaller" /> <!-- (4) -->
            <property name="unmarshaller" ref="xmlMarshaller" /> <!-- (5) -->
        </bean>
    </mvc:message-converters>

    <!-- ... -->

</mvc:annotation-driven>

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

項番	説明
(1)	Spring-oxm から提供されている Jaxb2Marshaller の bean 定義を行う。 Jaxb2Marshaller はデフォルトの状態で XXE Injection 対策が行われている。
(2)	packagesToScan プロパティに JAXB 用の JavaBean(javax.xml.bind.annotation.XmlRootElement アノテーションなどが付与されている JavaBean) 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 JAXB 用の JavaBean がスキャンされ、marshal、unmarshal 対象の JavaBean として登録される。 <context:component-scan> の base-package 属性と同じ仕組みでスキャンされる。
(3)	<mvc:annotation-driven> の子要素である <mvc:message-converters> 要素に、

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。

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.
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。

- 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}" />

<meta name="_csrf" content="${_csrf.token}" />
<meta name="_csrf_header" content="${_csrf.headerName}" />

<!-- 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、ヘッダ名とトークン値は HTML の `<meta>` 要素に設定し、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。
(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。

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。

課題

TBD

クライアント側の実装方法については、次版以降で詳細化する予定である。

ちなみに: 上記例では CSRF トークン値 (`${_csrf.token}`) と CSRF トークンヘッダー名 (`${_csrf.headerName}`) 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「=」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.
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"
```

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.
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 の処理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 形式のデータを Jackson を使って JavaBean に変換する MappingJacksonHttpMessageConverter 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 を処理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。

処理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)	処理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、処理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 new ResponseEntity<CalculationResult>(result, HttpStatus.BAD_REQUEST);
    }
    // omitted

    // (2)
    return new ResponseEntity<CalculationResult>(result, HttpStatus.OK);
}
```

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 の処理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。
複数の処理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。

処理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 new ResponseEntity<CalculationResult>(result, HttpStatus.CONFLICT);
        }

        // omitted

        return new ResponseEntity<CalculationResult>(result, HttpStatus.OK);
    }
```

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。

5.16 RESTful Web Service

5.16.1 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

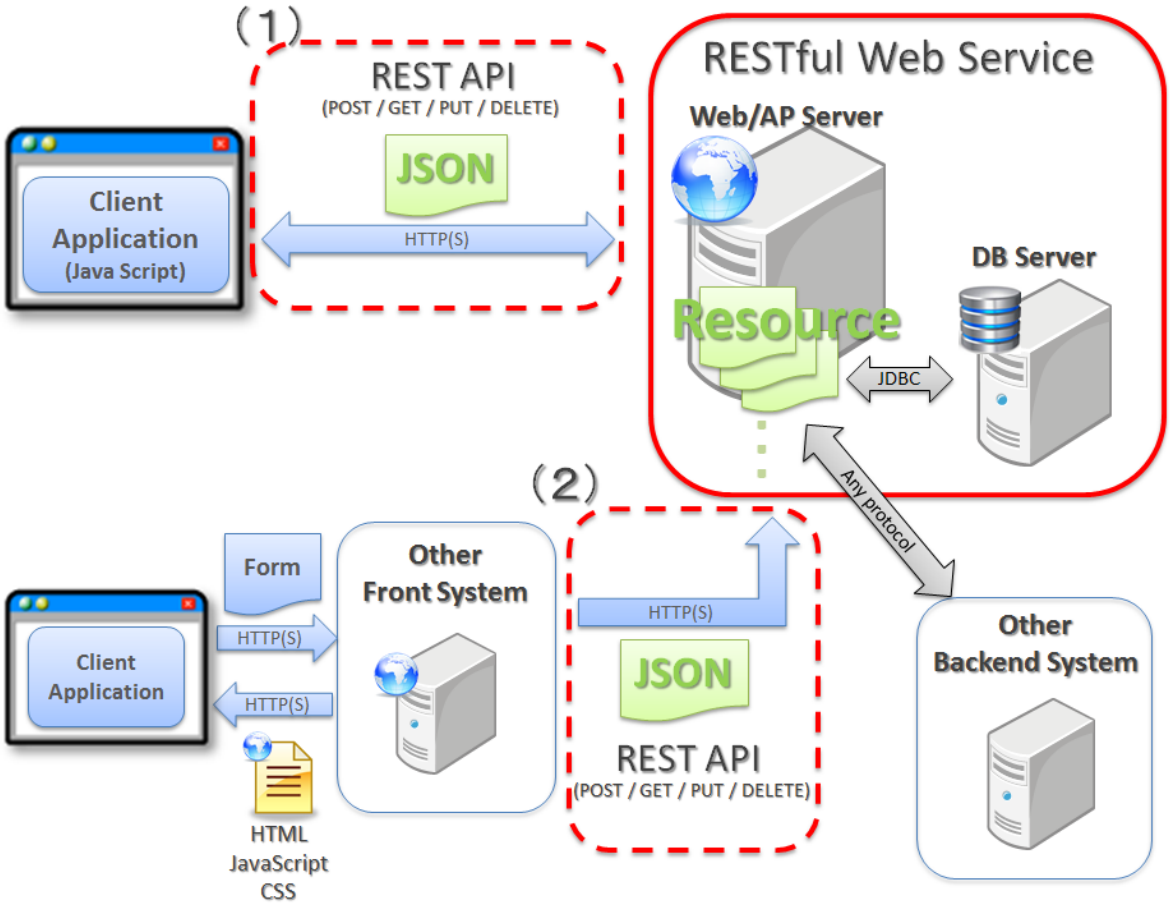
まず REST とは、「**RE**presentational **S**tate **T**ransfer」の略であり、
クライアントとサーバ間でデータをやりとりするアプリケーションを構築するための**アーキテクチャスタイル**の一つである。
REST のアーキテクチャスタイルには、いくつかの重要な原則があり、これらの原則に従っているもの (システムなど) は **RESTful** と表現される。
つまり、「RESTful Web Service」とは、REST の原則に従って構築されている Web Service という事になる。

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をやりとりを行う。 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 Global Framework 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。

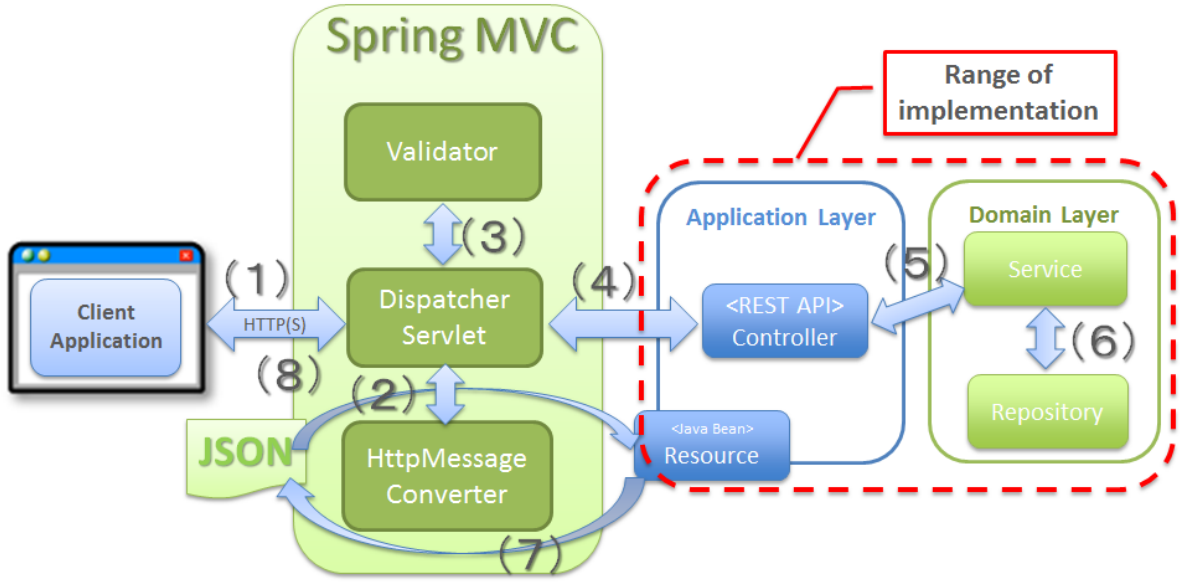
Spring MVC にデフォルトで組み込まれている主な共通機能は以下の通りである。
これらの機能は、REST API を提供する Controller のメソッドにアノテーションを指定だけで有効にする事ができる。

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。



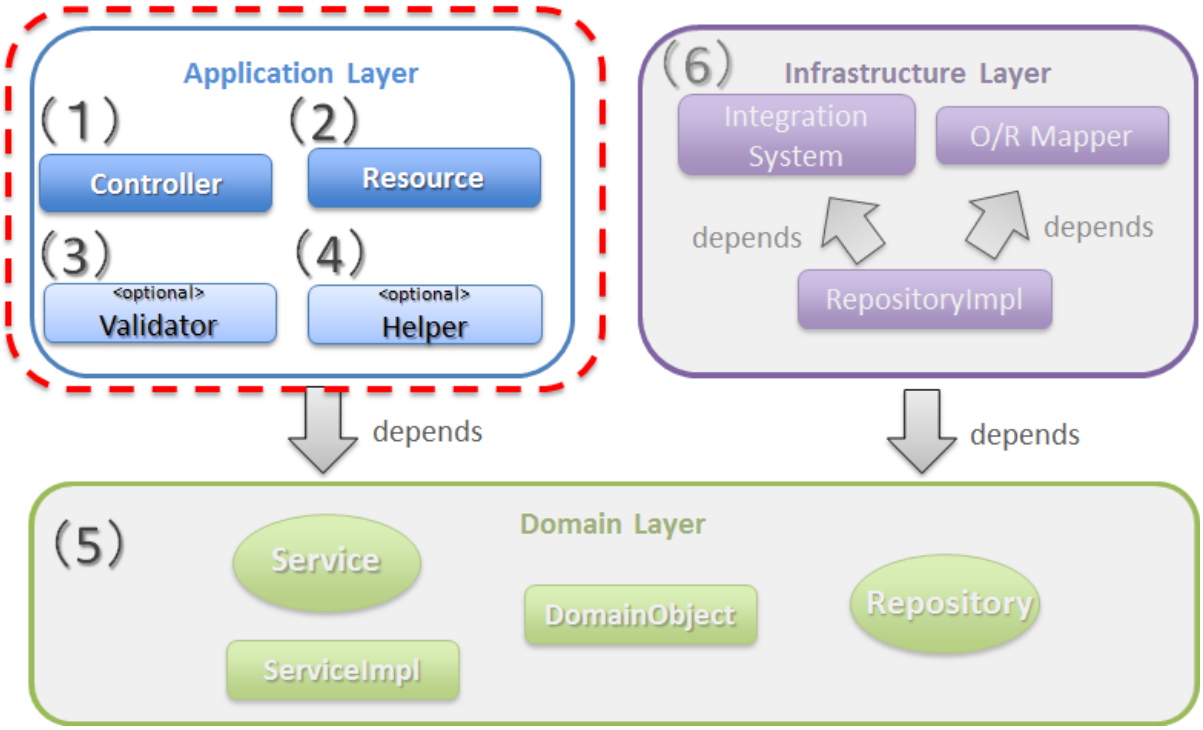
項番	処理レイヤ	説明
(1)	Spring MVC (Framework)	Spring MVC は、クライアントからのリクエストを受け取り、呼び出す REST API(Controller の処理メソッド) を決定する。
(2)		Spring MVC は、`HttpMessageConverter` を使用して、リクエスト BODY に指定されている JSON 形式の電文を Resource オブジェクトに変換する。
(3)		Spring MVC は、`Validator` を使用して、Resource オブジェクトに格納されて値に対して入力チェックを行う。
(4)		Spring MVC 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。
(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)		

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 Global Framework 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.util.Date;

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

public class TodoResource {

    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 の処理メソッド) 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.stereotype.Controller;
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.ResponseBody;
```

```
import org.springframework.web.bind.annotation.ResponseStatus;

import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Controller
@RequestMapping("todos")
public class TodoRestController {
    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    // (1)
    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    @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)
    @ResponseBody
    @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)
    @ResponseBody
    @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)
    @ResponseBody
    @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)
        @ResponseBody
        @ResponseStatus(HttpStatus.NO_CONTENT)
        public void deleteTodo(@PathVariable("todoId") String todoId) {
            todoService.delete(todoId);
        }

    }
```

5.16.2 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	リソースの作成する。	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。**

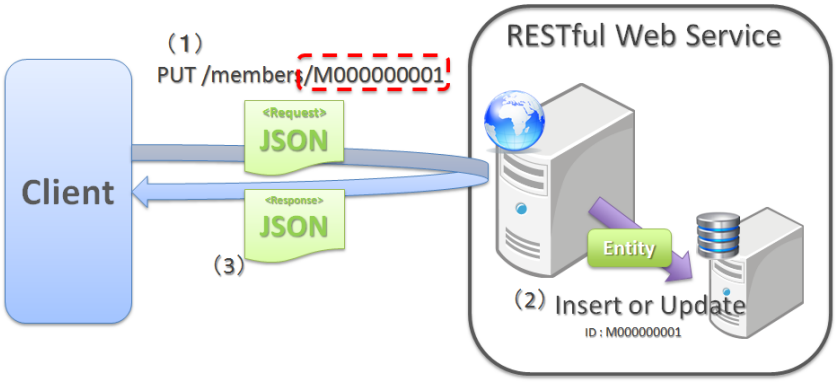
PUT メソッドを使用してリソースを作成する場合、

- 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。

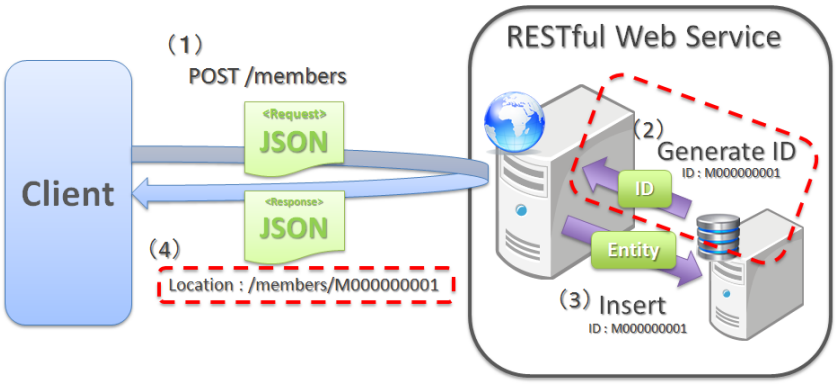
以下に、PUT と POST メソッドを使ってリソースを作成する際の処理イメージの違いについて説明する。

【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 2616(Hypertext Transfer Protocol – HTTP/1.1) の 6.1.1 Status Code and Reason Phrase 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。

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。

課題

TBD

高い可用性が求められる場合は、「CSRF 対策用のトークン値を AP サーバのメモリ (HTTP セッション) 以外に保存する」アーキテクチャを検討した方がよい。

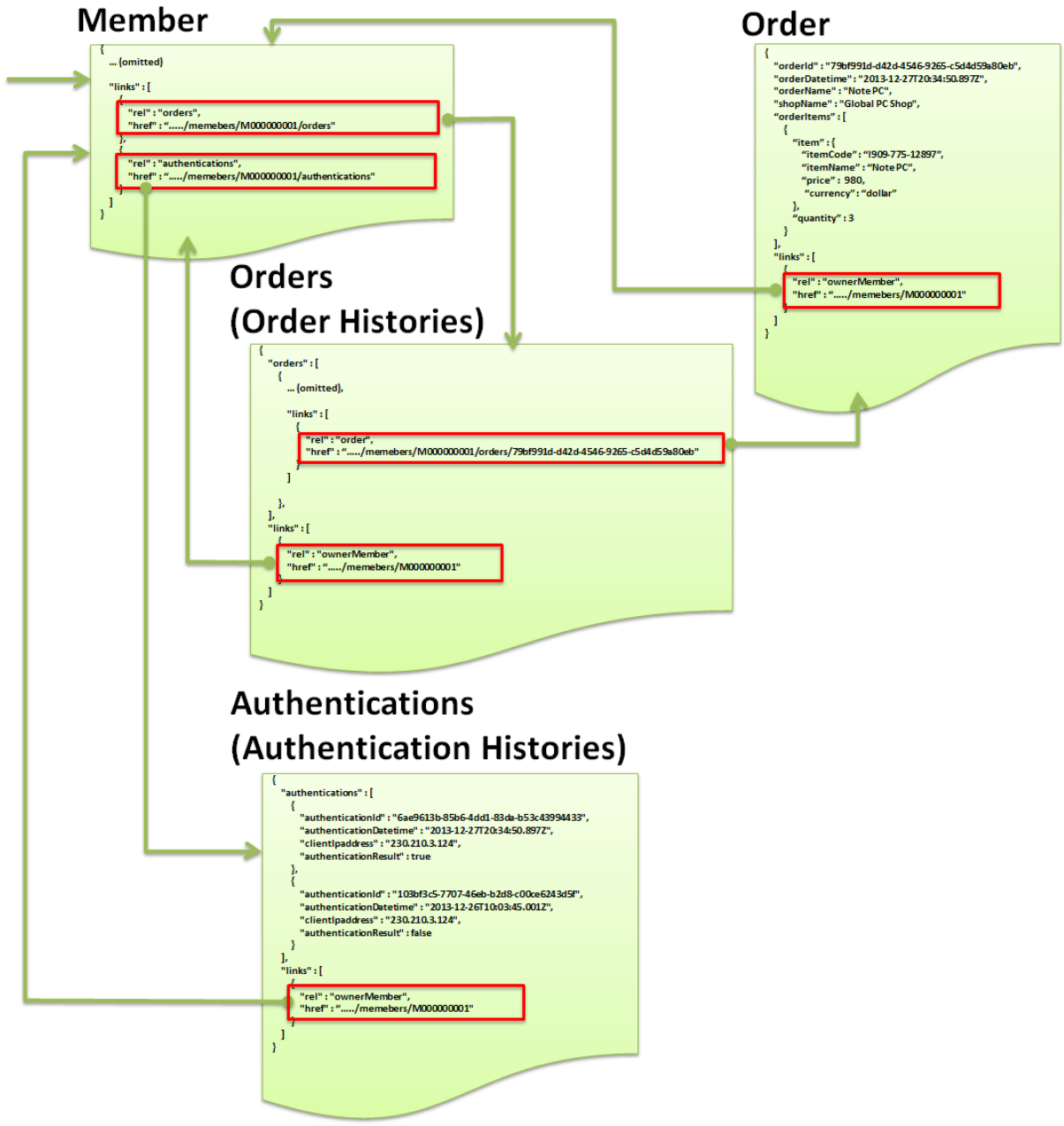
具体的なアーキテクチャについては、現在検討中であり、次版以降に記載する予定である。

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。



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"
	},

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。

5.16.3 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**`}`

課題

TBD

URI の中に API バージョンを含めるべきかは、現在検討中である。

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。

注釈: HEAD と OPTIONS メソッドについて

以降の説明では、HEAD と OPTIONS メソッドについても触れているが、REST API としての提供は任意とする。

HTTP の仕様に準拠した REST API を作成する場合は、HEAD 及び OPTIONS メソッドの提供も必要だが、実際に使われるケースは稀であり、必要ない事が多いためである。

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"
}
```

パイパーメディアリンクの形式

パ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

課題

TBD

認証及び認可制御をどのような指針で行うかについて記載する。

OAuth2 の仕組みを使って認証・認可を行う仕組みについて、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-Since 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。

バージョニング

課題

TBD

RESTful Web Service 自体のバージョン管理及び複数バージョンの並行稼働をどのように行うかについて、次版以降に記載する予定である。

5.16.4 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。

警告: StAX(Streaming API for XML) 使用時の DOS 攻撃対策について
XML 形式のデータを 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。

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。

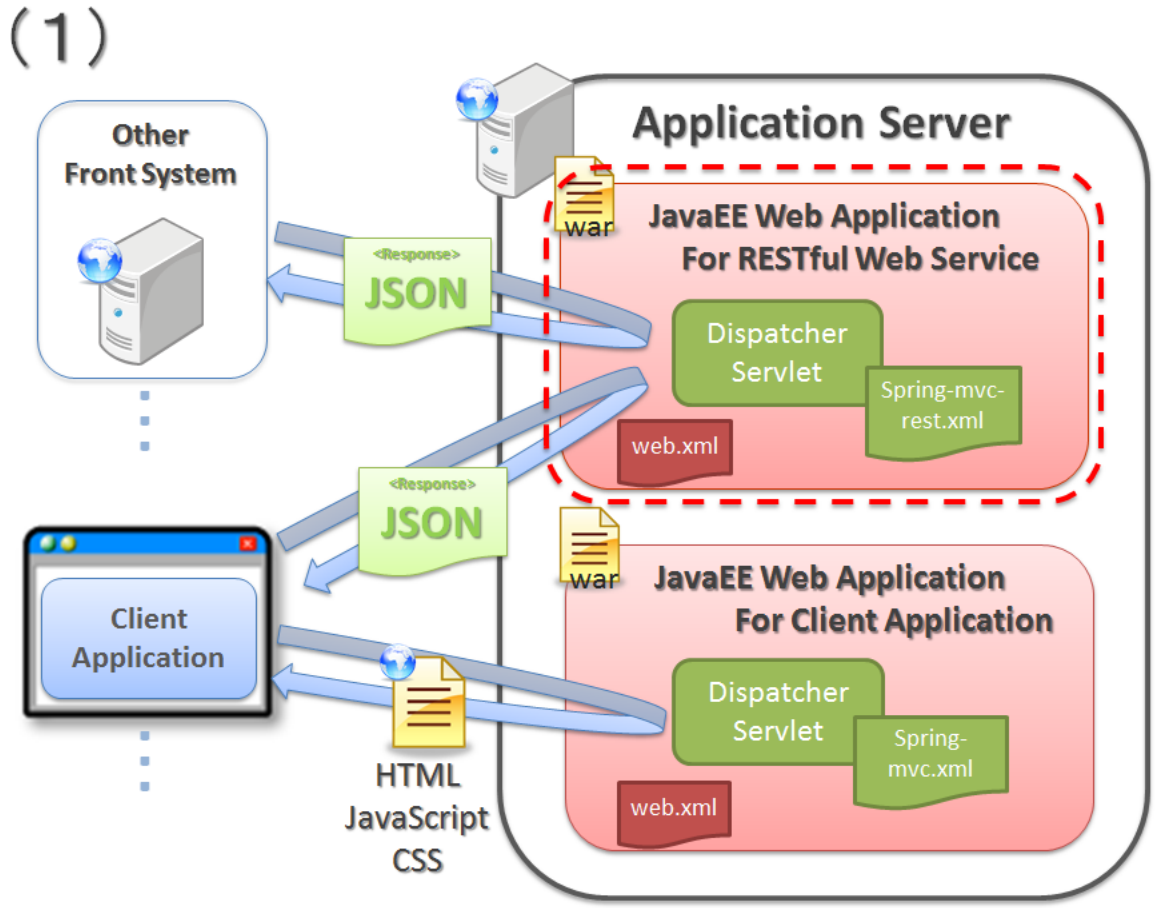
本ガイドラインでは、上記の様な問題が起こらないようにするために、RESTful Web Service をクライアントアプリケーションを同じ Web アプリケーション (war) として構築する場合は、`DispatcherServlet` を分割することを推奨している。

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。

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。



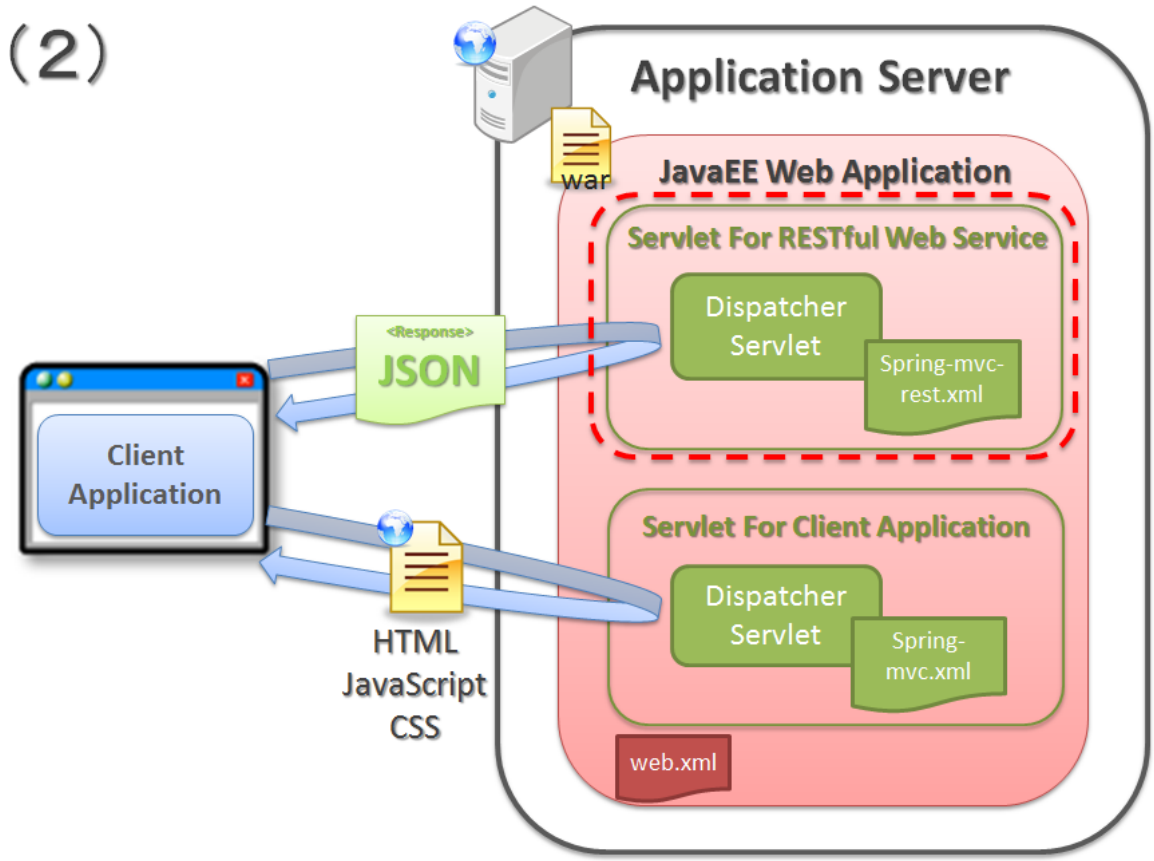
RESTful Web Service で必要となる Spring MVC のコンポーネントを有効化するための設定

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
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
```




```
            http://www.springframework.org/schema/util/spring-util.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd
            http://www.springframework.org/schema/aop
            http://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.MappingJacksonHttpMessageConverter">
            <property name="objectMapper">
                <bean id="objectMapper" class="org.codehaus.jackson.map.ObjectMapper">
                    <!-- (2) -->
                    <property name="dateFormat">
                        <bean class="org.codehaus.jackson.map.util.StdDateFormat" />
                    </property>
                </bean>
            </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.r
        </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。
(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 Injection 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 Injection* **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。
(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。

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)
(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。

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 リソースの一件取得する。
(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: 本節では、リソースの CRUD 操作の説明に注力するため、HEAD と OPTIONS メソッドの説明は行わない。HTTP の仕様に準拠した RESTful Web Service を作成する場合は、「*HTTP* **の仕様に準拠した** *RESTful Web Service* **の作成**」を参照されたい。

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。

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.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 org.codehaus.jackson.map.annotate.JsonSerialize;
    import org.codehaus.jackson.map.annotate.JsonSerialize.Inclusion;
```

```
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)
    @JsonSerialize(include = Inclusion.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では、Entity クラスと Resource クラスのコピーは、「Bean マッピング (Dozer)」を使って行う。

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、「Bean マッピング (Dozer)」を使ってコピーすると Joda-Time のオブジェクトは正しくコピーされない。

そのため、正しくコピーされるようにするためには、「***Dozer* を使って *Joda-Time* のクラスをコピーする方法**」を適用する必要がある。

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

@RequestMapping("members") // (1)
@Controller
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。

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 の作成する。

```
// (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")
@Controller
public class MemberRestController {
```

```
        // omitted

        @Inject
        MemberService memberService;

        @Inject
        Mapper beanMapper;

        // (3)
        @RequestMapping(method = RequestMethod.GET)
        // (4)
        @ResponseBody
        // (5)
        @ResponseStatus(HttpStatus.OK)
        public Page<MemberResource> getMembers(
                // (6)
                @Validated MembersSearchQuery query,
                // (7)
                Pageable pageable) {

            // (8)
            Page<Member> page = memberService.searchMembers(query.getName(), pageable);

            // (9)
            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());

            // (10)
            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。
(4)	メソッドアノテーションとして、`@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 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。
(5)	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 new ResponseEntity(responseResource, HttpStatus.OK); }` 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。
(6)	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。
(7)	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。

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"
  } ],
  "sort" : [ {
    "direction" : "DESC",
    "property" : "lastModifiedAt",
    "ignoreCase" : false,
    "ascending" : false
  } ],
  "lastPage" : false,
  "firstPage" : false,
  "totalElements" : 25,
  "numberOfElements" : 2,
```

```
    "totalPages" : 13,
    "size" : 2,
    "number" : 1
}
```

- 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-i
    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",
```

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)
@ResponseBody
@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")
@Controller
public class MemberRestController {

    // omitted

    // (1)
    @RequestMapping(method = RequestMethod.POST)
    @ResponseBody
    // (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。
(3)	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","emailAdd
```

- 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
```

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")
@Controller
public class MemberRestController {

    // omitted

    // (1)
    @RequestMapping(value = "{memberId}", method = RequestMethod.GET)
    @ResponseBody
    // (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
```

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")
@Controller
public class MemberRestController {

    // omitted

    // (1)
    @RequestMapping(value = "{memberId}", method = RequestMethod.PUT)
    @ResponseBody
    // (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
```

- 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
```

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")
@Controller
public class MemberRestController {

    // omitted

    // (1)
    @RequestMapping(value = "{memberId}", method = RequestMethod.DELETE)
```

```
        @ResponseBody
        // (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。

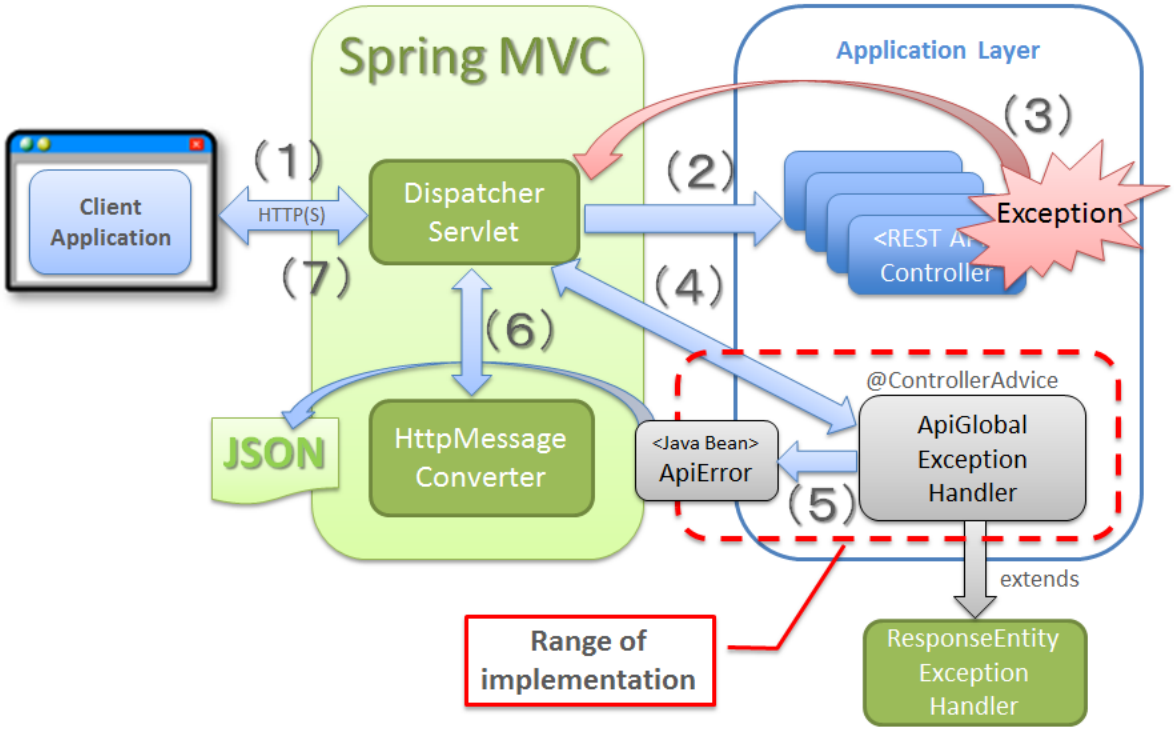
`ResponseEntityExceptionHandler` では、Spring MVC のフレームワーク内で発生する例外を`@ExceptionHandler` アノテーションを使ってハンドリングするメソッドが予め実装されている。
そのため、Spring MVC のフレームワーク内で発生する例外のハンドリングを個別に実装する必要がない。
また、`ResponseEntityExceptionHandler` でハンドリングされる例外に対応する HTTP ステータスコードは、`DefaultHandlerExceptionResolver` と同様の仕様で設定される。
ハンドリングされる例外と設定される 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 org.codehaus.jackson.map.annotate.JsonSerialize;
import org.codehaus.jackson.map.annotate.JsonSerialize.Inclusion;

// (1)
public class ApiError implements Serializable {

    private static final long serialVersionUID = 1L;

    private final String code;

    private final String message;

    @JsonSerialize(include = Inclusion.NON_EMPTY)
    private final String target; // (2)

    @JsonSerialize(include = Inclusion.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`@JsonSerialize(include = Inclusion.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、`@JsonSerialize(include = Inclusion.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 new ResponseEntity<>(apiError, headers, status);
    }

    // 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を変換している。 `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。

- 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.","d
```

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 の処理メソッドの引数 (`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、「*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。

- 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)	org.codehaus.jackson. JsonParseException	JSON として不正な構文が含まれる場合に発生する。
(2)	org.codehaus.jackson.map.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)	org.codehaus.jackson.map. 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 parameter
```

- 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, null,
                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, null,
                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, null,
                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, null,
                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。

```
# ---
# 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 or equal to {valu
javax.validation.constraints.DecimalMin.message  = "{0}" must be greater than or equal to {v
javax.validation.constraints.Digits.message      = "{0}" numeric value out of bounds. (<{int
javax.validation.constraints.Future.message      = "{0}" must be in the future.
javax.validation.constraints.Max.message         = "{0}" must be less than or equal to {valu
javax.validation.constraints.Min.message         = "{0}" must be greater than or equal to {v
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 numb
org.hibernate.validator.constraints.Email.message            = "{0}" not a well-formed email
org.hibernate.validator.constraints.Length.message           = "{0}" length must be between
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.Range.message            = "{0}" must be between {min} a
org.hibernate.validator.constraints.SafeHtml.message         = "{0}" may have unsafe html co
org.hibernate.validator.constraints.ScriptAssert.message     = "{0}" script expression "{scr
org.hibernate.validator.constraints.URL.message              = "{0}" must be a valid URL.

# for common library
org.terasoluna.gfw.common.codelist.ExistInCodeList = "{0}" must exist in code list of {codeL
```

- xxx-domain/src/main/resources/i18n/domain-messages.properties

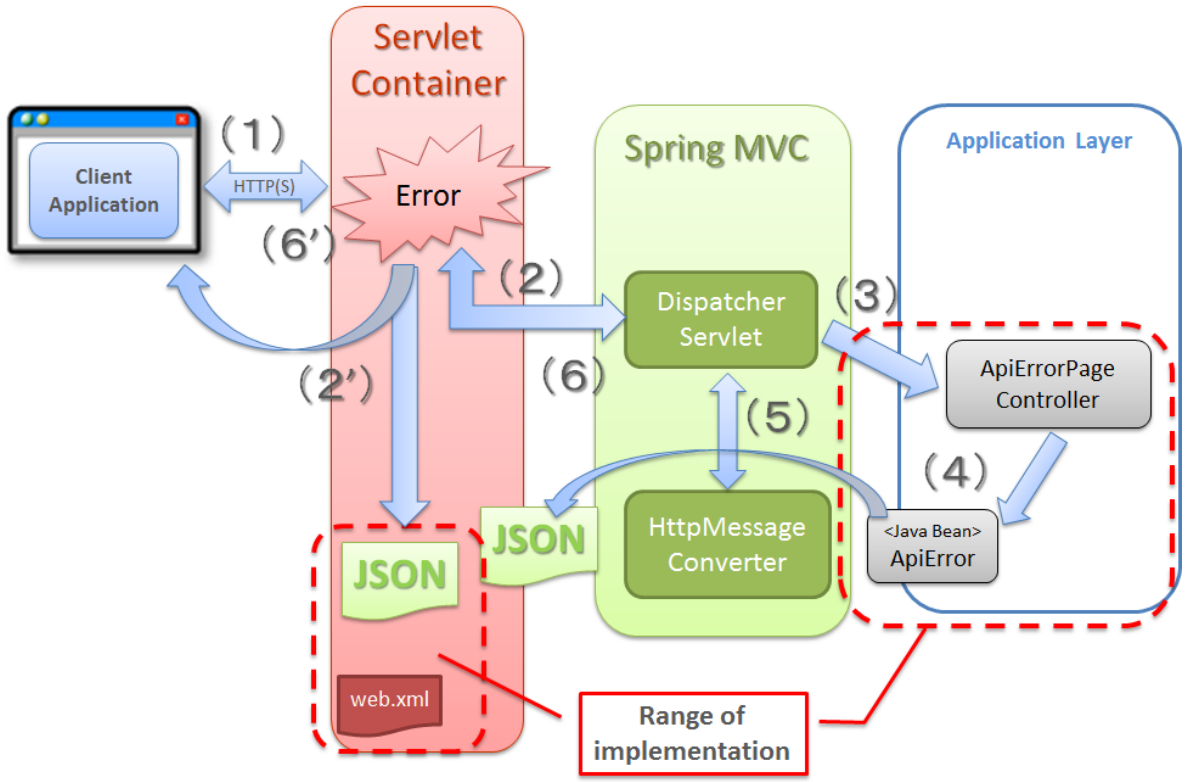
  ドメイン層で発生するエラーに対して、エラーコード (例外コード) に対応するメッセージの設定を行う。

```
# 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の実装

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 javax.inject.Inject;
import javax.servlet.RequestDispatcher;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.context.request.RequestAttributes;
import org.springframework.web.context.request.WebRequest;

// (1)
@RequestMapping("error")
@Controller
public class ApiErrorPageController {

    @Inject
    ApiErrorCreator apiErrorCreator; // (2)

    // (3)
    @RequestMapping
    @ResponseBody
    public ResponseEntity<ApiError> handleErrorPage(
            @RequestParam("errorCode") String errorCode, // (4)
            WebRequest request) {
        // (5)
        HttpStatus httpStatus = HttpStatus.valueOf((Integer) request
                .getAttribute(RequestDispatcher.ERROR_STATUS_CODE,
                        RequestAttributes.SCOPE_REQUEST));
        // (6)
        ApiError apiError = apiErrorCreator.createApiError(request, errorCode,
                httpStatus.getReasonPhrase());
        // (7)
        return new ResponseEntity<>(apiError, httpStatus);
    }

}
```

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)	エラー応答を行う処理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。 したがって、例外タイプ (<exception-type>) を使ってハンドリングしたエラーページの処理を本メソッドを使って行う場合は、別途考慮が必要である。
(4)	エラーコードをリクエストパラメータとして受け取る。
(5)	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。
(6)	リクエストパラメータで受け取ったエラーコードに対応するエラー情報を生成する。
(7)	(5)(6) で取得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?errorCode=e.ex.fw.5001</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?errorCode=e.ex.fw.5001」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。

- 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

課題

TBD

OAuth2(Spring Security OAuth2) を使用して認証・認可を実現する方法について、次版以降に記載する予定である。

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。

5.16.5 Appendix

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 org.codehaus.jackson.map.annotate.JsonSerialize;
import org.codehaus.jackson.map.annotate.JsonSerialize.Inclusion;

// (2)
public abstract class AbstractLinksSupportedResource {

    // (3)
    @JsonSerialize(include = Inclusion.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、パ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、`@JsonSerialize(include = Inclusion.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")
@Controller
public class MemberRestController {

    // omitted

    @RequestMapping(value = "{memberId}", method = RequestMethod.GET)
    @ResponseBody
    @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")
@Controller
public class MemberRestController {

    // omitted

    @RequestMapping(method = RequestMethod.POST)
    @ResponseBody
    @ResponseStatus(HttpStatus.CREATED)
    public HttpEntity<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)
        HttpHeaders responseHeaders = new HttpHeaders();
        responseHeaders.setLocation(createdUri);

        // (4)
        return new HttpEntity<>(responseResource, responseHeaders);
    }

    // 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。
(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)	作成したリソースの URI を Location ヘッダに設定する。
(4)	`org.springframework.http.HttpEntity` に、Resource オブジェクトとレスポンスヘッダに設定する情報を格納し、返却する。

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
```

OPTIONS メソッドのリクエストを Controller にディスパッチするための設定

HTTP の仕様に準拠する場合は、、リソース毎に呼び出しが許可されている HTTP メソッドの一覧を返却する必要がある。そのため、OPTIONS メソッドのリクエストを Controller へディスパッチするための設定を追加する必要となる。

DispatcherServlet のデフォルトの設定では、OPTIONS メソッドのリクエストは Controller にディスパッチされずに、DispatcherServlet が許可しているメソッドのリストが Allow ヘッダに設定されてしまう。

- web.xml

```
<!-- omitted -->

<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-rest.xml</param-value>
    </init-param>
    <!-- (1) -->
    <init-param>
        <param-name>dispatchOptionsRequest</param-name>
        <param-value>true</param-value>
```

```
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- omitted -->
```

項番	説明
(1)	RESTful Web Service のリクエストを受け付ける `DispatcherServlet` の初期化パラメータ (dispatchOptionsRequest) の値を、`true` に設定する。

OPTIONS メソッドの実装

HTTP の仕様に準拠する場合、リソース毎に呼び出しが許可されている HTTP メソッドの一覧を返却する必要がある。

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 (REST API) のリストを応答する API の実装例を、以下に示す。

- REST API の実装

  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 (REST API) のリストを応答する処理を実装する。

```
@RequestMapping("members")
@Controller
public class MembersRestController {

    // omitted

    @RequestMapping(value = "{memberId}", method = RequestMethod.OPTIONS)
    @ResponseBody
    // (1)
    @ResponseStatus(HttpStatus.OK)
    public HttpEntity<Void> optionsMember(
        @PathVariable("memberId") String memberId) {

        // (2)
        memberService.getMember(memberId);

        // (3)
```

```
            return RestResponseUtils.createEntityOfOptions(
                HttpMethod.GET, HttpMethod.HEAD, HttpMethod.PUT, HttpMethod.DELETE);
        }

        // omitted

    }
```

項番	説明
(1)	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。
(2)	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が存在するかチェックを行う。
(3)	**URI で指定されたリソースでサポートされている HTTP メソッドを、Allow ヘッダに設定する。** 上記例では、 ユーティリティメソッドを呼び出して、HTTP メソッドの一覧が設定されたレスポンスエンティティを生成している。

- ユーティリティメソッド

  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処理は、全てのリソースで同じようなロジックを組む事になるため、ユーティリティメソッド化した方がよい。

```
    public static HttpEntity<Void> createEntityOfOptions(
            HttpMethod... allowedMethods) {

        Set<HttpMethod> allowedMethodSet = new LinkedHashSet<>(
                Arrays.asList(allowedMethods));
        if (!allowedMethodSet.contains(HttpMethod.OPTIONS)) {
            allowedMethodSet.add(HttpMethod.OPTIONS);
        }

        HttpHeaders responseHeaders = new HttpHeaders();
        responseHeaders.setAllow(allowedMethodSet);
```

```
        return new HttpEntity<>(responseHeaders);
    }
```

- リクエスト例

```
    OPTIONS /rest-api-web/api/v1/members/M000000004 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: 6d7bbc818c7f44e7942c54bc0ddc15bb
    Allow: GET,HEAD,PUT,DELETE,OPTIONS
    Content-Length: 0
    Date: Mon, 17 Mar 2014 01:54:27 GMT
```

HEAD メソッドの実装

HTTP の仕様に準拠する場合、GET メソッドを実装する場合、HEAD メソッドも実装する必要がある。
URI で指定されたリソースのメタ情報を応答する API の実装例を、以下に示す。

- REST API の実装
  URI で指定されたリソースのメタ情報を取得する処理を実装する。

```
    @RequestMapping("members")
    @Controller
    public class MemberRestController {

        // omitted
```

```
    @RequestMapping(value = "{memberId}",
                    method = { RequestMethod.GET,
                               RequestMethod.HEAD }) // (1)
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public MemberResource getMember(
            @PathVariable("memberId") String memberId) {
        // omitted
    }

    // omitted

}
```

項番	説明
(1)	GET メソッドの処理を行う REST API の`@RequestMapping` アノテーションの method 属性に `RequestMethod.HEAD` を追加する。 HEAD メソッドは、GET メソッドと同じ処理を行いヘッダ情報のみレスポンスする必要があるため、`@RequestMapping` アノテーションの method 属性に、`RequestMethod.HEAD` も指定する。 レスポンス BODY を空にする処理は、Servlet API の標準機能で行われるため、Controller の処理としては GET メソッドと同じ処理を行えばよい。

- リクエスト例

```
HEAD /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
```

- レスポンス例

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: 71093a551e624c149867b6bfec486d2c
Content-Type: application/json;charset=UTF-8
```

```
Content-Length: 452
Date: Thu, 13 Mar 2014 13:25:23 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/**"
    auto-config="true" use-expressions="true" create-session="stateless">
    <!--<sec:custom-filter ref="csrfFilter" before="LOGOUT_FILTER"/>--> <!-- (2) -->
    <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
    <!--<sec:session-management session-authentication-strategy-ref="sessionAuthenticationSt
</sec:http>

<sec:http auto-config="true" use-expressions="true">
    <sec:custom-filter ref="csrfFilter" before="LOGOUT_FILTER"/>
    <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
    <sec:session-management session-authentication-strategy-ref="sessionAuthenticationStrate
</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。
(2)	CSRF 対策を無効化するために、CSRF 対策用の Servlet Filter の定義をコメントアウトしている。実際は、物理的に削除してよい。
(3)	セッションを使用しないため、セッション管理に関係するコンポーネントへの参照定義をコメントアウトしている。実際は、物理的に削除してよい。

XXE Injection 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) Injection 対策を行う必要がある。

terasoluna-gfw-web 1.0.1.RELEASE 以上では、XXE Injection 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。

> **警告: XXE(XML External Entity) Injection 対策について**
> terasoluna-gfw-web 1.0.0.RELEASE を使用している場合は、XXE Injection 対策が行われていない Spring MVC(3.2.4.RELEASE) に依存しているため、Spring-oxm から提供されているクラスを使用すること。

Spring-oxm を依存アーティファクトとして追加する。

- pom.xml

```
<!-- omitted -->

<!-- (1) -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
    <version>${org.springframework-version}</version> <!-- (2) -->
</dependency>

<!-- omitted -->
```

項番	説明
(1)	Spring-oxm を依存アーティファクトとして追加する。
(2)	Spring のバージョンは、terasoluna-gfw-parent の pom.xml に定義されている Spring のバージョン番号を管理するためのプレースフォルダ (${org.springframework-version}) から取得すること。

Spring-oxm から提供されているクラスを使用して XML とオブジェクトの相互変換を行うための bean 定義を行う。

- spring-mvc-rest.xml

```
<!-- omitted -->

<!-- (1) -->
<bean id="xmlMarshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <property name="packagesToScan" value="com.examples.app" /> <!-- (2) -->
</bean>

<!-- omitted -->

<mvc:annotation-driven>

    <mvc:message-converters>
        <!-- (3) -->
        <bean class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter"
```

```
                <property name="marshaller" ref="xmlMarshaller" /> <!-- (4) -->
                <property name="unmarshaller" ref="xmlMarshaller" /> <!-- (5) -->
            </bean>
        </mvc:message-converters>

        <!-- omitted -->

    </mvc:annotation-driven>

    <!-- omitted -->
```

項番	説明
(1)	Spring-oxm から提供されている `Jaxb2Marshaller` の bean 定義を行う。 `Jaxb2Marshaller` はデフォルトの状態で XXE Injection 対策が行われている。
(2)	`packagesToScan` プロパティに JAXB 用の JavaBean(`javax.xml.bind.annotation.XmlRootElement` アノテーションなどが付与されている JavaBean) 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 JAXB 用の JavaBean がスキャンされ、marshal、unmarshal 対象の JavaBean として登録される。 `<context:component-scan>` の base-package 属性と同じ仕組みでスキャンされる。
(3)	`<mvc:annotation-driven>` の子要素である `<mvc:message-converters>` 要素に、 `MarshallingHttpMessageConverter` の bean 定義を追加する。
(4)	`marshaller` プロパティに (1) で定義した `Jaxb2Marshaller` の bean を指定する。
(5)	`unmarshaller` プロパティに (1) で定義した `Jaxb2Marshaller` の bean を指定する。

Dozer を使って Joda-Time のクラスをコピーする方法

Dozer を使用して、Joda-Time のクラス (`org.joda.time.DateTime`、`org.joda.time.LocalDate` など) をコピーする方法について説明する。

Joda-Time のクラスを変換するためのカスタムコンバータを作成する。

カスタムコンバータの詳細については、「Bean マッピング (Dozer)」を参照されたい。

- `JodaDateTimeConverter.java`

```
package org.terasoluna.examples.rest.infra.dozer.converter;

import org.dozer.DozerConverter;
import org.joda.time.DateTime;

public class JodaDateTimeConverter extends DozerConverter<DateTime, DateTime> {

    public JodaDateTimeConverter() {
        super(DateTime.class, DateTime.class);
    }

    @Override
    public DateTime convertTo(DateTime source, DateTime destination) {
        // This method not called, because type of from/to is same.
        return convertFrom(source, destination);
    }

    @Override
    public DateTime convertFrom(DateTime source, DateTime destination) {
        return source;
    }

}
```

- `JodaLocalDateConverter.java`

```
package org.terasoluna.examples.rest.infra.dozer.converter;

import org.dozer.DozerConverter;
import org.joda.time.LocalDate;

public class JodaLocalDateConverter extends
                                    DozerConverter<LocalDate, LocalDate> {

    public JodaLocalDateConverter() {
        super(LocalDate.class, LocalDate.class);
    }
```

```
        @Override
        public LocalDate convertTo(LocalDate source, LocalDate destination) {
            // This method not called, because type of from/to is same.
            return convertFrom(source, destination);
        }

        @Override
        public LocalDate convertFrom(LocalDate source, LocalDate destination) {
            return source;
        }

    }
```

作成したカスタムコンバータを Dozer に適用する。

カスタムコンバータの詳細については、「Bean マッピング (Dozer)」を参照されたい。

```
    <!-- (1) -->
    <?xml version="1.0" encoding="UTF-8"?>
    <mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-i
        xsi:schemaLocation="
            http://dozer.sourceforge.net http://dozer.sourceforge.net/schema/beanmapping.xsd
        ">

        <configuration>
            <custom-converters>
                <!-- (2) -->
                <converter type="org.terasoluna.examples.rest.infra.dozer.converter.JodaDateTime
                    <class-a>org.joda.time.DateTime</class-a>
                    <class-b>org.joda.time.DateTime</class-b>
                </converter>
                <converter type="org.terasoluna.examples.rest.infra.dozer.converter.JodaLocalDat
                    <class-a>org.joda.time.LocalDate</class-a>
                    <class-b>org.joda.time.LocalDate</class-b>
                </converter>
            </custom-converters>
        </configuration>

    </mappings>
```

項番	説明
(1)	Dozer の動作設定を定義するファイルを作成する。 今回の実装例では、 `/xxx-domain/src/main/resources/META-INF/dozer/dozer-configration-mapping.` に格納する。
(2)	上記例では、Joda-Time のクラス (`org.joda.time.DateTime` と `org.joda.time.LocalDate`) に対するカスタムコンバータの定義を追加している。

注釈: ドメイン層でも Dozer を使用する場合は、Dozer の動作設定を定義するファイルは、ドメイン層用のプロジェクト (`xxx-domain`) に格納する事を推奨する。

アプリケーション層のみで Dozer を使う場合は、アプリケーション層用のプロジェクト (`xxx-web`) に格納してもよい。

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.stereotype.Controller;
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.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;
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")
@Controller
public class MemberRestController {

    @Inject
    MemberService memberService;

    @Inject
    Mapper beanMapper;

    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    @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.POST)
    @ResponseBody
    @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)
    @ResponseBody
    @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)
    @ResponseBody
    @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)
    @ResponseBody
    @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 new ResponseEntity<>(apiError, headers, status);
    }

    @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, null,
                HttpStatus.NOT_FOUND, request);
    }

    @ExceptionHandler(BusinessException.class)
    public ResponseEntity<Object> handleBusinessException(BusinessException ex,
            WebRequest request) {
        return handleResultMessagesNotificationException(ex, null,
                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, null, HttpStatus.CONFLICT,
                request);
    }
```

```
    @ExceptionHandler(Exception.class)
    public ResponseEntity<Object> handleSystemError(Exception ex,
            WebRequest request) {
        return handleExceptionInternal(ex, null, null,
                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、JPA(Spring Data JPA) 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)		*member-mapping.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 javax.persistence.Access;
import javax.persistence.AccessType;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;
import javax.persistence.Transient;
import javax.persistence.Version;

import org.joda.time.DateTime;
import org.joda.time.LocalDate;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;

@Table(name = "t_member")
@Entity
public class Member implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private String memberId;

    private String firstName;

    private String lastName;

    @Transient
    private Gender gender;

    private LocalDate dateOfBirth;

    private String emailAddress;

    private String telephoneNumber;

    private String zipCode;

    private String address;
```

```
    @CreatedDate
    private DateTime createdAt;

    @LastModifiedDate
    private DateTime lastModifiedAt;

    @Version
    private long version;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "member_id")
    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;
    }

    @Access(AccessType.PROPERTY)
    @Column(name = "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 javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;

import org.joda.time.DateTime;
```

```
import org.springframework.data.annotation.LastModifiedDate;

@Table(name = "t_member_credential")
@Entity
public class MemberCredential implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private String memberId;

    private String signId;

    private String password;

    private String previousPassword;

    private DateTime passwordLastChangedAt;

    @LastModifiedDate
    private DateTime lastModifiedAt;

    @Version
    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 org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.terasoluna.examples.rest.domain.model.Member;

public interface MemberRepository extends JpaRepository<Member, String> {
```

```
    @Query("SELECT m FROM Member m"
            + " WHERE m.firstName LIKE :name% ESCAPE '~'"
            + " OR m.lastName LIKE :name% ESCAPE '~'")
    Page<Member> findPageByContainsName(@Param("name") String name,
            Pageable pageable);

}
```

MemberService.java

java/org/terasoluna/examples/rest/domain/service/member/MemberService.java

```
package org.terasoluna.examples.rest.domain.service.member;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.terasoluna.examples.rest.domain.model.Member;

public interface MemberService {

    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 javax.inject.Inject;
```

```
import javax.inject.Named;

import org.dozer.Mapper;
import org.joda.time.DateTime;
import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
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.DateFactory;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.message.ResultMessages;
import org.terasoluna.gfw.common.query.QueryEscapeUtils;
import org.terasoluna.gfw.common.sequencer.Sequencer;

@Transactional
@Service
public class MemberServiceImpl implements MemberService {

    @Inject
    MemberRepository memberRepository;

    @Inject
    @Named("memberIdSequencer")
    Sequencer<String> sequencer;

    @Inject
    DateFactory dateFactory;

    @Inject
    PasswordEncoder passwordEncoder;

    @Inject
    Mapper beanMapper;

    @Transactional(readOnly = true)
    public Page<Member> searchMembers(String name, Pageable pageable) {

        // escape to like condition value
        String escapedName = QueryEscapeUtils.toLikeCondition(name);

        // find members that matches with search criteria
        return memberRepository.findPageByContainsName(escapedName, pageable);
    }
```

```
    @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;
    }

    public Member createMember(Member creatingMember) {

        MemberCredential creatingCredential = creatingMember.getCredential();

        // get processing current date time
        DateTime currentDateTime = dateFactory.newDateTime();

        // set id
        String newMemberId = sequencer.getNext();
        creatingMember.setMemberId(newMemberId);
        creatingCredential.setMemberId(newMemberId);

        // 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);

        // save member & member credential
        try {
            return memberRepository.saveAndFlush(creatingMember);
        } catch (DataIntegrityViolationException e) {
            // If sign id is already used
            throw new BusinessException(ResultMessages.error().add(
                    DomainMessageCodes.E_EX_MM_8001,
                    creatingCredential.getSignId()), e);
        }
    }

    public Member updateMember(String memberId, Member updatingMember) {
        // get member
        Member member = getMember(memberId);
```

```
        // override updating member attributes
        beanMapper.map(updatingMember, member, "member.update");

        // save updating member
        return memberRepository.save(member);
    }

    public void deleteMember(String memberId) {

        // delete member
        memberRepository.delete(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";
}
```

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-instanc
    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>
```

5.17 ファイルアップロード

5.17.1 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、Servlet3.0 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、Servlet3.0 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。

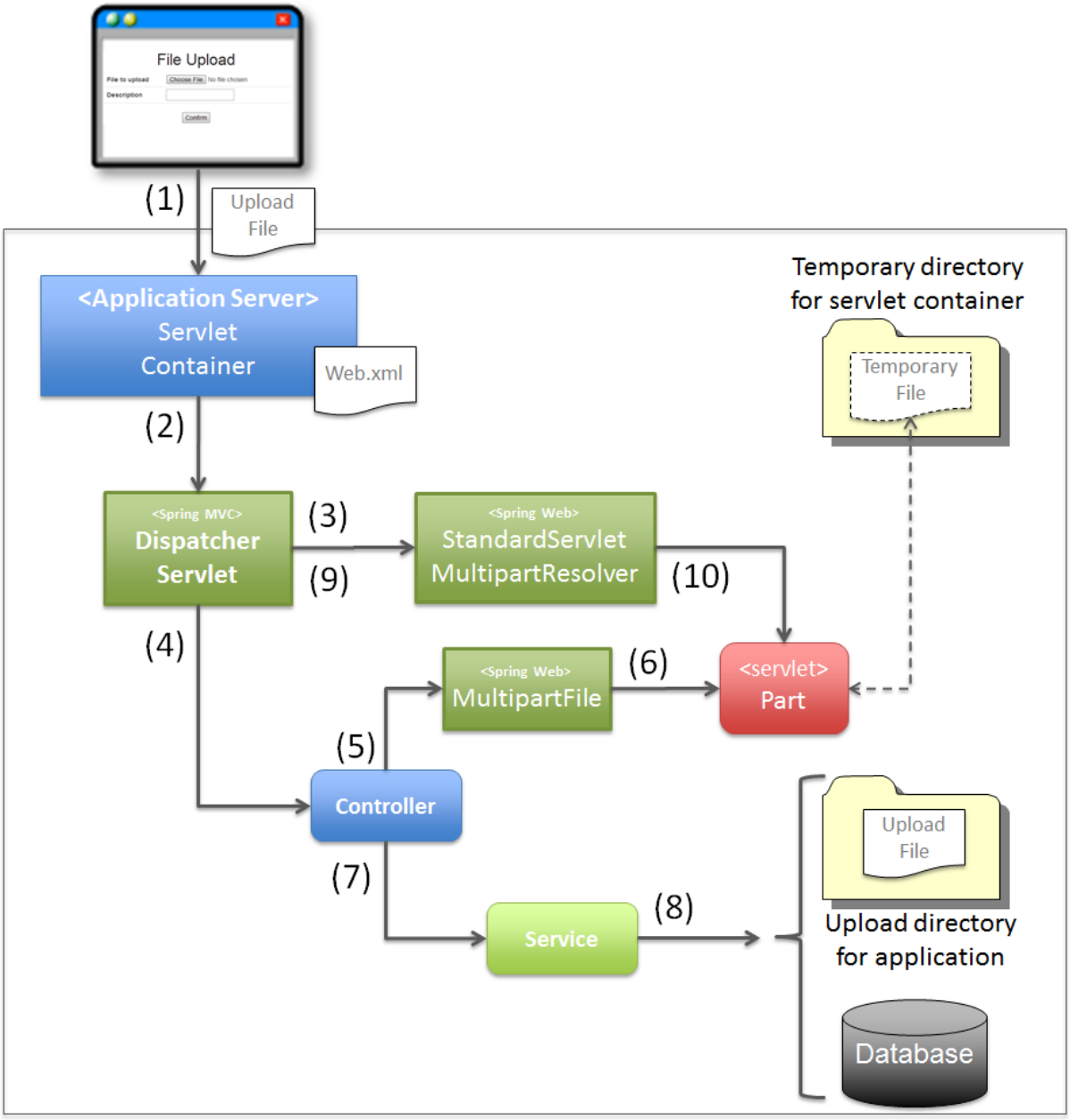
注釈: WebLogic など一部のアプリケーションサーバー上で Servlet 3.0 のファイルアップロード機能を使用すると、リクエストパラメータやファイル名のマルチバイト文字が文字化けすることがある。

問題が発生するアプリケーションサーバーを使用する場合は、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 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 7.0.52 以上、8.0 系は 8.0.3 以上を使用する必要がある。

アップロード処理の基本フロー

Servlet3.0 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.servlet.DispatcherServlet` を呼び出す。
(3)	`DispatcherServlet` は、`org.springframework.web.multipart.support.StandardServletMultipartResolve` のメソッドを呼び出し、Servlet3.0 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、Servlet3.0 から導入された

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、Servlet3.0 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	Servlet3.0 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。 Servlet3.0 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	Servlet3.0 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、Servlet Filter の処理内でリクエストパラメータを取得できるようにするためのクラス。 このクラスを使用しないと、Servlet Filter でリクエストパラメータの取得ができないため、Spring Security から提供されている CSRF トークンチェック機能が正しく動作しない。 具体的には、CSRF トークンが取得できないため、常に CSRF トークンエラーとなりファイルのアップロードが出来ない。

ちなみに: 本ガイドラインでは、Servlet3.0 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。

5.17.2 How to use

アプリケーションの設定

Servlet3.0 のアップロード機能を有効化するための設定

Servlet3.0 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
    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、Servlet3.0 以上の XSD ファイルを指定する。
(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、メモリを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 の処理内でリクエストパラメータを取得できるようにするための設定

multipart/form-data リクエストの時に、Servlet Filter の処理内でリクエストパラメータを取得できるようにするために、以下の設定を行う。

- 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: MultipartFilter は、リクエストパラメータにアクセスする Servlet Filter より前に定義する必要がある。
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。

Servlet3.0 のアップロード機能と Spring MVC を連携するための設定

Servlet3.0 のアップロード機能と、Spring MVC を連携するために、以下の設定を行う。

- `spring-mvc.xml`

```
    <bean id="multipartResolver"
        class="org.springframework.web.multipart.support.StandardServletMultipartResolver"> <!--
    </bean>
```

項番	説明
(1)	Servlet3.0 用の MultipartResolver である `StandardServletMultipartResolver` を、bean 定義する。 beanID は、`"multipartResolver"`とすること。 この設定を行うことで、アップロードされたファイルを `org.springframework.web.multipart.MultipartFile` として、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。

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。

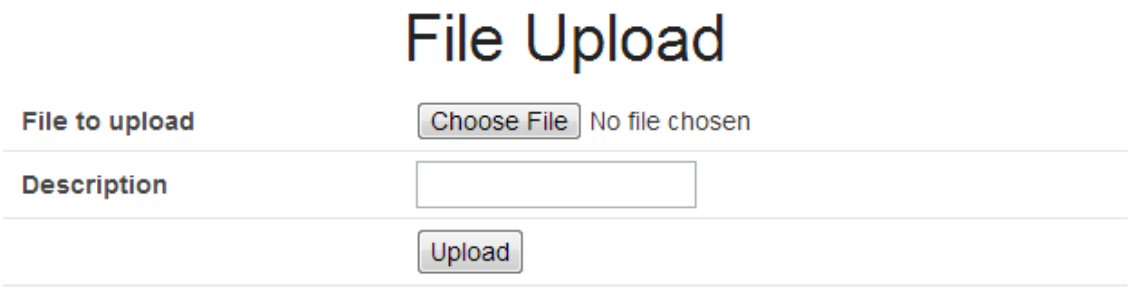
- applicationContext.xml

```
<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/uploadFile" 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処理メソッド。
(3)	ファイルをアップロードするための処理メソッド。
(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。
(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({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
    @Retention(RetentionPolicy.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({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
        @Retention(RetentionPolicy.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({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
    @Retention(RetentionPolicy.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({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
        @Retention(RetentionPolicy.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({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.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({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
    @Retention(RetentionPolicy.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

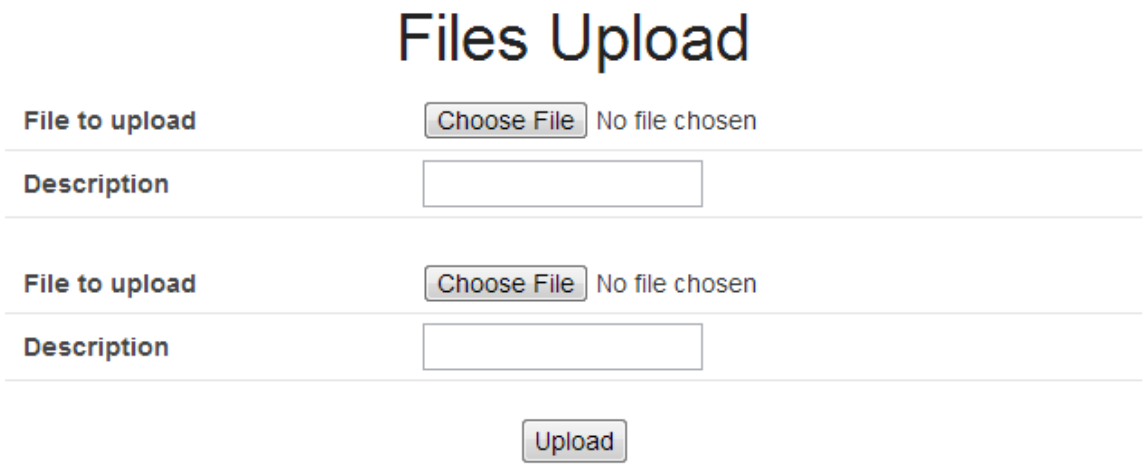
```
    @RequestMapping(value = "uploadFile", method = RequestMethod.POST)
    public String uploadFile(@Validated FileUploadForm form,
            BindingResult result, RedirectAttributes redirectAttributes) {

        // (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">
  <table>
    <tr>
      <th width="35%">File to upload</th>
      <td width="65%">
        <form:input type="file" path="fileUploadForms[0].file" /> <!-- (1) -->
        <form:errors path="fileUploadForms[0].file" />
      </td>
    </tr>
    <tr>
      <th width="35%">Description</th>
      <td width="65%">
        <form:input path="fileUploadForms[0].description" />
        <form:errors  path="fileUploadForms[0].description" />
      </td>
    </tr>
  </table>
  <table>
```

```
        <tr>
          <th width="35%">File to upload</th>
          <td width="65%">
            <form:input type="file" path="fileUploadForms[1].file" /> <!-- (1) -->
            <form:errors path="fileUploadForms[1].file" />
          </td>
        </tr>
        <tr>
          <th width="35%">Description</th>
          <td width="65%">
            <form:input path="fileUploadForms[1].description" />
            <form:errors path="fileUploadForms[1].description" />
          </td>
        </tr>
      </table>
      <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

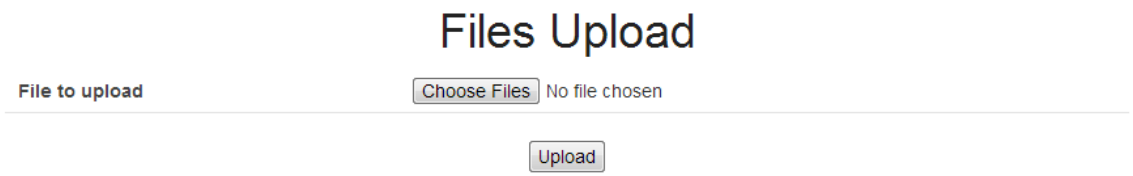
```
    @RequestMapping(value = "uploadFiles", method = RequestMethod.POST)
    public String uploadFiles(@Validated FilesUploadForm form,
            BindingResult result, RedirectAttributes redirectAttributes) {

        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<NotEmptyUploadFile, Collection<MultipartFile>> {

        // (2)
        private final UploadFileNotEmptyValidator validator =
            new UploadFileNotEmptyValidator();

        // (3)
        @Override
        public void initialize(NotEmptyUploadFile 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({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
    @Retention(RetentionPolicy.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、 `@NotEmptyUploadFile` 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="filesUploadForm2" 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";
    }
```

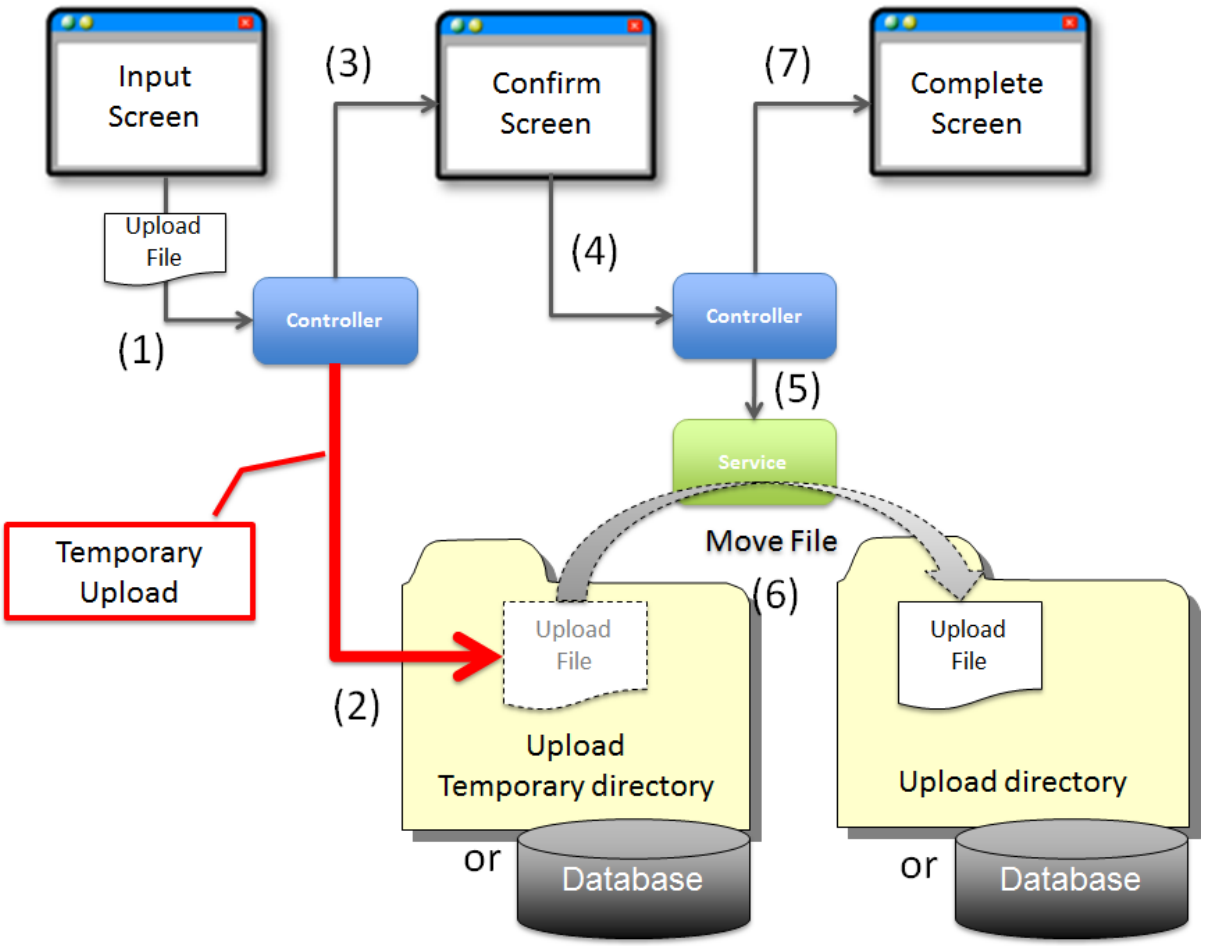
項番	説明
(1)	フォームオブジェクトから MultipartFile オブジェクトが格納されているリストを取得し、アップロード処理を実装する。 上記例では、具体的な実装は省略しているが、共有ディスクやデータベースへ保存する処理を行うことになる。

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、ファイルの中身を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。
(5)	Controller は、Service のメソ
(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

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の識別するための 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処理を Service のメソッドに委譲することになる。

5.17.3 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.DateFactory;

// (1)
public class UnnecessaryFilesCleaner {

    @Inject
    DateFactory 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。

5.17.4 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**

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。

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ー上で 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(検証バージョンは 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。アプリケーションサーバーの問題であると思われるが、アプリケーションサーバー側で修正されない限り、ファイルとマルチバイト文字を同時に送信する事ができない。

この問題は、Commons FileUpload を使用することで回避できるため、本ガイドラインでは、アプリケーションサーバーが修正されるまでの暫定対処として、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。

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>
    <version>1.3.1</version>
</dependency>
```

項番	説明
(1)	commons-fileupload への依存関係を追加する。

警告: Apache Commons FileUpload を使用する場合、CVE-2014-0050 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、1.3.1 以上を使用する必要がある。

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 に場合、最大値はヘッダ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-ap
    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、*CSRF* **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。

5.18 ファイルダウンロード

5.18.1 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.AbstractExcelView`

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。

5.18.2 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。

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>
        <version>${com.lowagie.itext.version}</version>
        <exclusions>
            <exclusion>
                <artifactId>xml-apis</artifactId>
                <groupId>xml-apis</groupId>
            </exclusion>
            <exclusion>
                <artifactId>bctsp-jdk14</artifactId>
                <groupId>org.bouncycastle</groupId>
            </exclusion>
            <exclusion>
                <artifactId>jfreechart</artifactId>
                <groupId>jfree</groupId>
            </exclusion>
            <exclusion>
                <artifactId>dom4j</artifactId>
                <groupId>dom4j</groupId>
            </exclusion>
            <exclusion>
                <groupId>org.swinglabs</groupId>
                <artifactId>pdf-renderer</artifactId>
            </exclusion>
```

```
        </exclusions>
    </dependency>
</dependencies>

<properties>
    <!-- omitted -->
    <com.lowagie.itext.version>4.2.1</com.lowagie.itext.version>
</properties>
```

> **注釈:** Spring 3.2 では、itext の 5 系のバージョンに対応していない。

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、`order` プロパティを設定して、通常使用する `InternalResourceViewResolver` より先に呼ばれるようにする必要がある。

> **注釈:** Spring はさまざまな View Resolver を提供している。複数の Resolver をチェーンすることができるため、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。それを防ぐために order プロパティを指定することで、優先順位を設定できる。order プロパティの指定値が低いほど、先に実行される。

bean 定義ファイル

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>

<bean class="org.springframework.web.servlet.view.BeanNameViewResolver">  <!-- (1) -->
    <property name="order" value="0"/>  <!-- (2) -->
</bean>
```

項番	説明
(1)	BeanNameViewResolver を定義する。
(2)	order プロパティに 0 を設定する。InternalResourceViewResolver より、優先度を高くすること。

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。

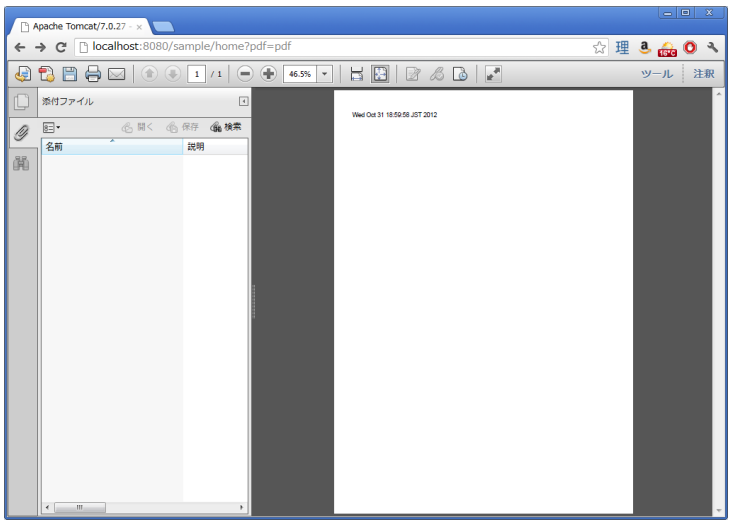


図 5.45　**Picture - FileDownload PDF**

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.AbstractExcelView` 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

AbstractExcelView を継承したクラスの実装例

```
@Component  // (1)
public class SampleExcelView extends AbstractExcelView {  // (2)

    @Override
    protected void buildExcelDocument(Map<String, Object> model,
            HSSFWorkbook workbook, HttpServletRequest request,
            HttpServletResponse response) throws Exception {  // (3)
        HSSFSheet sheet;
        HSSFCell 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)	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)	AbstractExcelView を継承する。
(3)	buildExcelDocument メソッドを実装する。

AbstractExcelView 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</artifactId>
        <version>${org.apache.poi.poi.version}</version>
    </dependency>
</dependencies>

<properties>
    <!-- omitted -->
    <org.apache.poi.poi.version>3.9</org.apache.poi.poi.version>
</properties>
```

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。

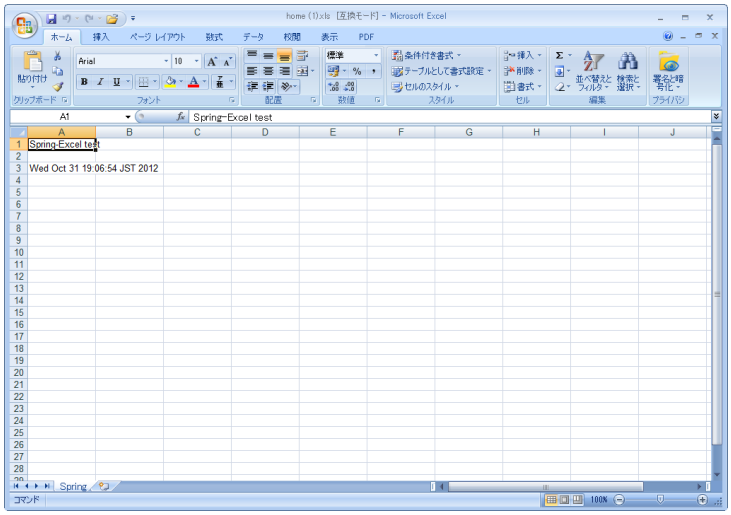


図 5.46　**Picture - FileDownload EXCEL**

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、InputStreame 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。

5.19 Tiles による画面レイアウト

5.19.1 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、

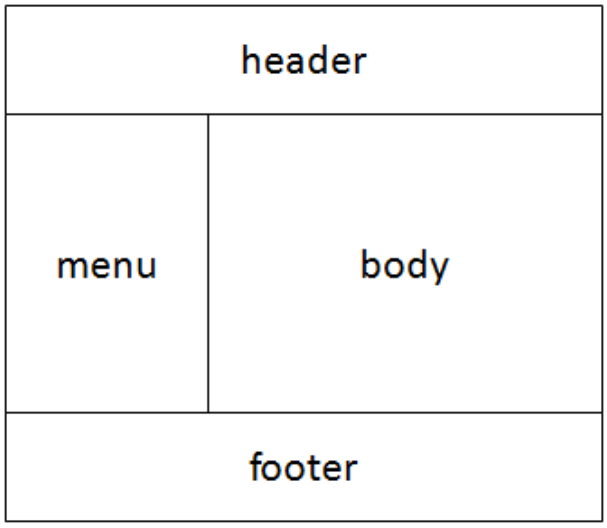


図 5.47 **Picture - Image of screen layout**

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。

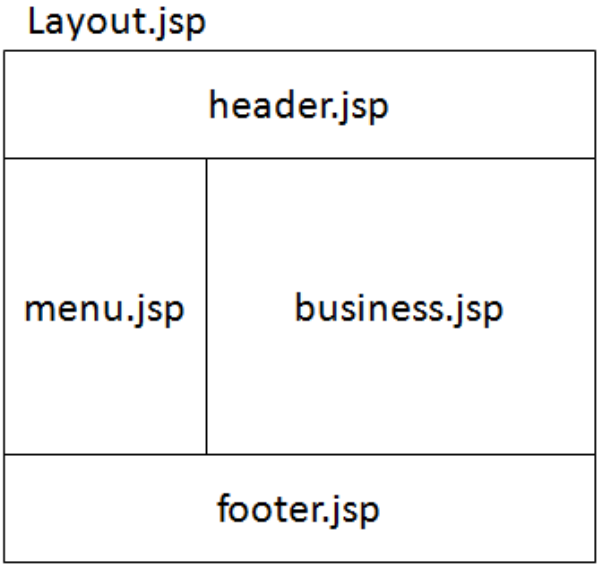


図 5.48 **Picture - Image of layout jsp**

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。

5.19.2 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: pom.xml は、以下のように terasoluna-gfw-parent の設定がされている前提である。

```
    <parent>
        <groupId>org.terasoluna.gfw</groupId>
        <artifactId>terasoluna-gfw-parent</artifactId>
        <version>x.y.z</version>
    </parent>
```

そのため、terasoluna-gfw-recommended-web-dependencies の <version> 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.tiles2.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

```
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver"> <!-- (1) -->
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
        <property name="order" value="2" />
    </bean>

    <bean id="tilesViewResolver"
        class="org.springframework.web.servlet.view.tiles2.TilesViewResolver"> <!-- (2) -->
        <property name="order" value="1" />
    </bean>

    <bean id="tilesConfigurer"
        class="org.springframework.web.servlet.view.tiles2.TilesConfigurer"> <!-- (3) -->
        <property name="definitions">
            <list>
                <value>/WEB-INF/tiles/tiles-definitions.xml</value>
            </list>
        </property>
    </bean>
```

項番	説明
(1)	InternalResourceViewResolver を定義する。 /WEB-INF/views/配下の jsp ファイルを対象にしている。 property 定義の order で”2” を指定することで、Tiles を定義した設定 tiles-definitions.xml に該当しない場合、 `org.springframework.web.servlet.view.InternalResourceViewResolver` を使うように優先順を定義する。 読み込んだ jsp の ファイル名が、Tiles 定義ファイル内 definition タグの name 属性のパターンに合致しない場合、 `org.springframework.web.servlet.view.InternalResourceViewResolver` が使用される。
(2)	TilesViewResovler を定義する。 property 定義の order で”1” を指定することで、最初に Tiles を定義した設定 tiles-definitions.xml を使うように優先順を定義する。 読み込んだ jsp のファイル名が、Tiles 定義ファイル内 definition タグの name 属性のパターンに合致する場合、 `org.springframework.web.servlet.view.tiles2.TilesViewResolver` が使用される。
(3)	Tiles の定義ファイルの場所を指定する。 /WEB-INF/tiles/tiles-definitions.xml を読み込む。

Tiles の定義

- tiles-definitions.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
   "-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
   "http://tiles.apache.org/dtds/tiles-config_2_1.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)	描画のリクエストの際に 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}はリクエストの”*/*” 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}にリクエストパス、{2}に JSP 名が一致するように設計する。 これにより、リクエスト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。

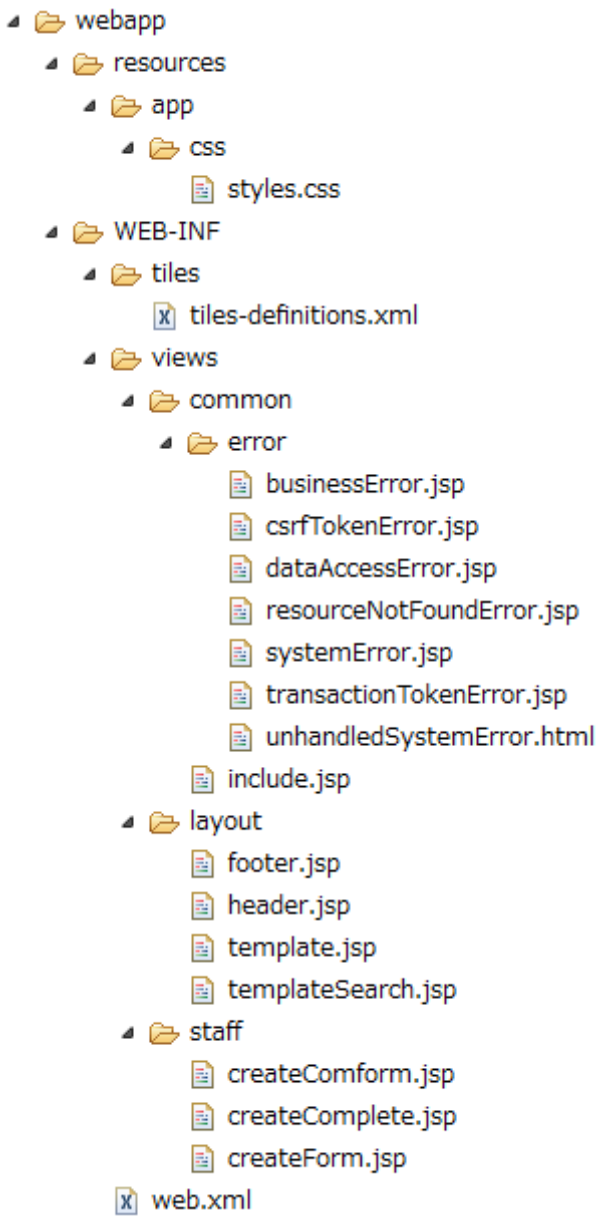
- /WEB-INF/views/common/include.jsp

```
<%@ 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) -->
```

項番	説明
(1)	Tiles 用のカスタムタグ (TLD) の定義を追加する。

- 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>
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>    <html class="no-js lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]>    <html class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!-->
<html class="no-js">
<!--<![endif]-->
<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"
    type="text/css" media="screen, projection">
<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)
</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>
```

Controller 作成

Controller を作成するとき、リクエストが <contextPath>/staff/create?form の場合、Controller からのリターンが”staff/createForm” となるように設定する。

- 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。

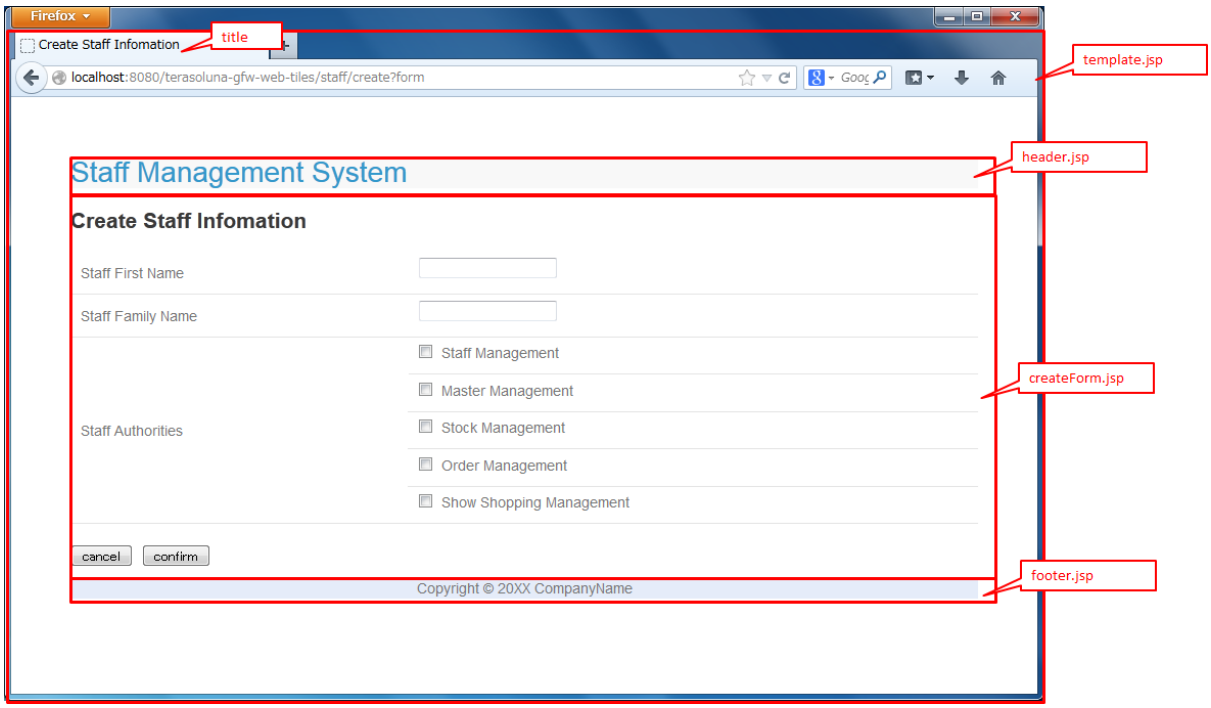
画面描画

リクエストに <contextPath>/staff/create?form が呼ばれると、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)	リクエストの時、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。



5.19.3 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 2.1//EN"
   "http://tiles.apache.org/dtds/tiles-config_2_1.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、描画のリクエストの際に name のパターンと同じ場合に呼ばれるレイアウト定義。 リクエストが<contextPath>/*/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}はリクエストの”*/search*” の「*」の 1 つ目。 {2}はリクエストの”*/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}にリクエストパス、{2}に先頭に”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: リクエストが 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、スタッフ検索画面へのリクエストが複数パターンに該当するため、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” はリクエストの”*/search*” の「*」の 1 つ目。 “searchStaff” はリクエストの”*/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>
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>    <html class="no-js lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]>    <html class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!-->
<html class="no-js">
<!--<![endif]-->
<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"
    type="text/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% で横幅 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% で横幅 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="${pageContext.request.contextPath}/staff/create?form">Create Staff Info
    </tr>
    <tr>
        <td><a href="${pageContext.request.contextPath}/staff/search">Search Staff Informati
    </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 を作成するとき、リクエストが <contextPath>/staff/search の場合、Controller からのリターンが”staff/searchStaff” となるように設定する。

- 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。

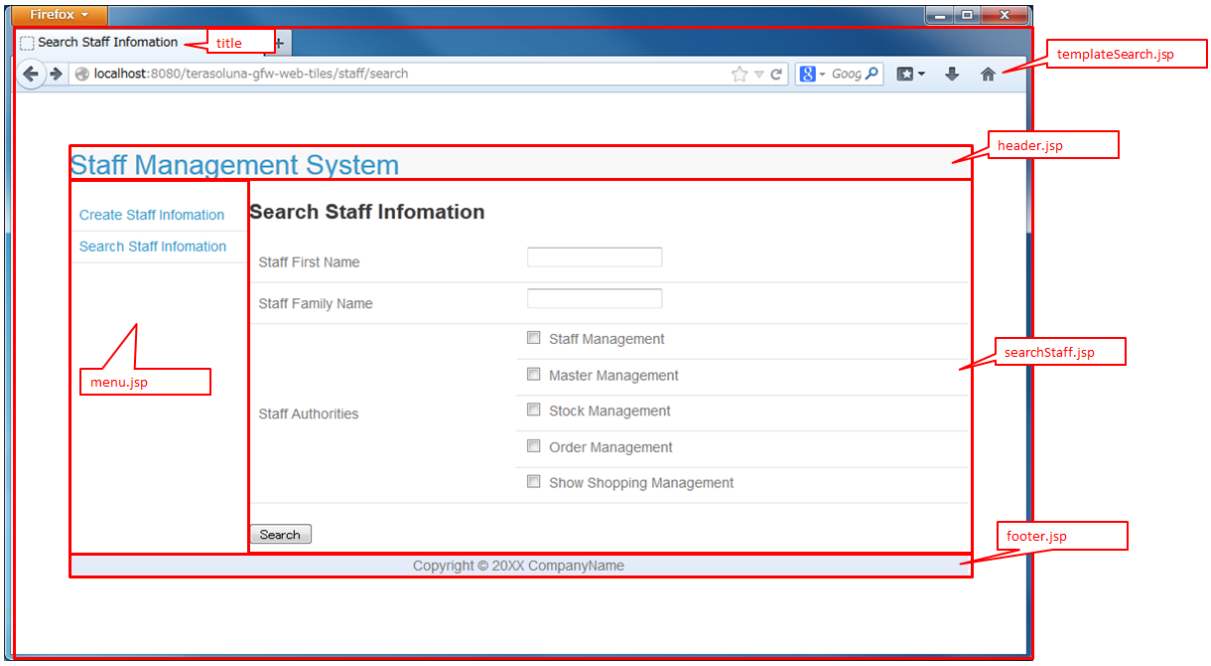
画面描画

リクエストに <contextPath>/staff/search が呼ばれると、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)	該当するリクエストの時、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)	リクエストが<contextPath>/*/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。



5.20 システム時刻

5.20.1 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。

共通ライブラリでは、返却する時刻を任意に変更できる `org.terasoluna.gfw.common.date.DateFactory` インタフェースを提供している。`DateFactory` により、以下 5 種類の型で日付を返却することができる。

method	type
newDateTime	org.joda.time.DateTime
newTimestamp	java.sql.Timestamp
newDate	java.util.Date
newSqlDate	java.sql.Date
newTime	java.sql.Time

注釈: Joda Time については、 日付操作 (Joda Time) を参照されたい。

ちなみに: JUnit などで日時を変更して試験を行いたい場合、Factory 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。

5.20.2 How to use

bean 定義ファイルに、`DateFactory` の実装クラスを定義し、Java クラスに `DateFactory` インタフェースでインジェクションする。
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.DefaultDateFactory	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.JdbcFixedDateFactory	DB に登録した固定の時刻を返却	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いる。 Performance Test 環境や、Production 環境では使用しない。 テーブルの定義が必要である。
org.terasoluna.gfw.common.date.JdbcAdjustedDateFactory	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。 差分値を 0 に設定することで Production 環境でも使用できる。 テーブルの定義が必要である。

注釈: 各クラスを設定する 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。DateFactory 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。

サーバーのシステム時刻を返却する

org.terasoluna.gfw.common.date.DefaultDateFactory を使用する。利用方法は、以下を参照されたい。

bean 定義ファイル ([projectname]-env.xml)

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.DefaultDateFactory" />  <!-- (1) -->
```

項番	説明
(1)	DefaultDateFactory クラスを bean 定義する。

Java クラス

```
@Inject
DateFactory dateFactory;  // (2)

public TourInfoSearchCriteria setUpTourInfoSearchCriteria() {

    DateTime dateTime = dateFactory.newDateTime();  // (3)

    // omitted
}
```

項番	説明
(2)	DateFactory 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.JdbcFixedDateFactory を使用する。利用方法は、以下を参照されたい。

bean 定義ファイル

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.JdbcFixedDateFactory" >  <!-- (1) --
    <property name="dataSource" ref="dataSource" />  <!-- (2) -->
    <property name="currentTimestampQuery" value="SELECT now FROM system_date" />  <!-- (3) -->
</bean>
```

項番	説明
(1)	org.terasoluna.gfw.common.date.JdbcFixedDateFactory を bean 定義する。
(2)	データソース (javax.sql.DataSource) の設定。
(3)	固定時刻取得 SQL(currentTimestampQuery) の設定。 テーブルに指定した、日時を返却する 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
DateFactory 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)	`JdbcFixedDateFactory.newDateTime()` の結果を画面に渡す。 DB に設定した固定の値が出力されている。
(5)	確認用に `new DateTime()` の結果を画面に渡す。 出力結果が毎回異なる値となっている。

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}
```

JdbcFixedDateFactory.newDateTime() メソッドを使用すると、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.JdbcAdjustedDateFactory を使用する。adjustedValueQuery プロパティに設定された SQL を実行して差分値を取得する。

利用方法は、以下を参照されたい。

bean 定義ファイル

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.JdbcAdjustedDateFactory" >
  <property name="dataSource" ref="dataSource" />
  <!-- <property name="adjustedValueQuery" value="SELECT diff FROM operation_date" /> --><!-- (1)
  <!-- <property name="adjustedValueQuery" value="SELECT diff * 1000 FROM operation_date" /> --><
  <property name="adjustedValueQuery" value="SELECT diff * 60 * 1000 FROM operation_date" /><!--
  <!-- <property name="adjustedValueQuery" value="SELECT diff * 60 * 60 * 1000 FROM operation_dat
  <!-- <property name="adjustedValueQuery" value="SELECT diff * 24 * 60 * 60 * 1000 FROM operatio
</bean>
```

項番	説明
(1)	operation_date テーブルに登録した差分値の単位を”milliseconds” として使用する場合の SQL
(2)	operation_date テーブルに登録した差分値の単位を”seconds” として使用する場合の SQL
(3)	operation_date テーブルに登録した差分値の単位を”minutes” として使用する場合の SQL
(4)	operation_date テーブルに登録した差分値の単位を”hours” として使用する場合の SQL
(5)	operation_date テーブルに登録した差分値の単位を”days” として使用する場合の SQL

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。

Java クラス

```
@Inject
DateFactory dateFactory;

@RequestMapping(value="datetime", method = RequestMethod.GET)
public String listConfirm(Model model) {

    model.addAttribute("firstExpectedDate", new DateTime());  // (6)
    model.addAttribute("serverTime", dateFactory.newDateTime());  // (7)
    model.addAttribute("lastExpectedDate", new DateTime());  // (8)

    return "date/dateTimeDisplay";
}
```

項番	説明
(6)	確認用に、dateFactory による DateTime 生成よりも前の時刻を画面に渡す。
(7)	JdbcAdjustedDateFactory.newDateTime() の結果を画面に渡す。 実行時から 1440 分を引いた時刻が取得されている。
(8)	確認用に、dateFactory による DateTime 生成よりも後の時刻を設定する

実行結果



SQL ログ

```
17. SELECT diff * 60 * 1000 FROM operation_date {executed in 1 msec}
```

dateFactory.newDateTime() による、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、JdbcAdjustedDateFactory では、取得結果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.JdbcAdjustedDateFactory" >
  <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 DateFactory 利用時に毎回 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、JdbcAdjustedDateFactory.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
    JdbcAdjustedDateFactory dateFactory;

    // omitted

    @RequestMapping(method = RequestMethod.GET)
    public String reload() {

        long adjustedValue = dateFactory.reload(); // (2)

        // omitted
    }
}
```

項番	説明
(2)	JdbcAdjustedDateFactory 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。

5.20.3 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 DateFactory	試験内容
Unit Test	DefaultDateFactory	日付に関わる試験は DataFactory を mock 化する。
Integration Test	DefaultDateFactory	日付に関わらない試験
	JdbcFixedDateFactory	特定の日付、時刻に固定して試験を実施する場合
	JdbcAdjustedDateFactory	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	JdbcAdjustedDateFactory	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	DefaultDateFactory	実際の時刻と変更する可能性が無い場合
	JdbcAdjustedDateFactory	**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

UnitTest 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、DateFacoty 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。dateFactory 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.DateFactory;

// omitted

@Inject
StaffRepository staffRepository;

@Inject
DateFactory 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.DateFactory;

public class StaffServiceTest {

    StaffService service;

    StaffRepository repository;

    DateFactory dateFactory;

    DateTime now;

    @Before
    public void setUp() {
        service = new StaffService();
        dateFactory = mock(DateFactory.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 DataFactory 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.DateFactory;

  // omitted

  @Inject
  DateFactory 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の判定する。
(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

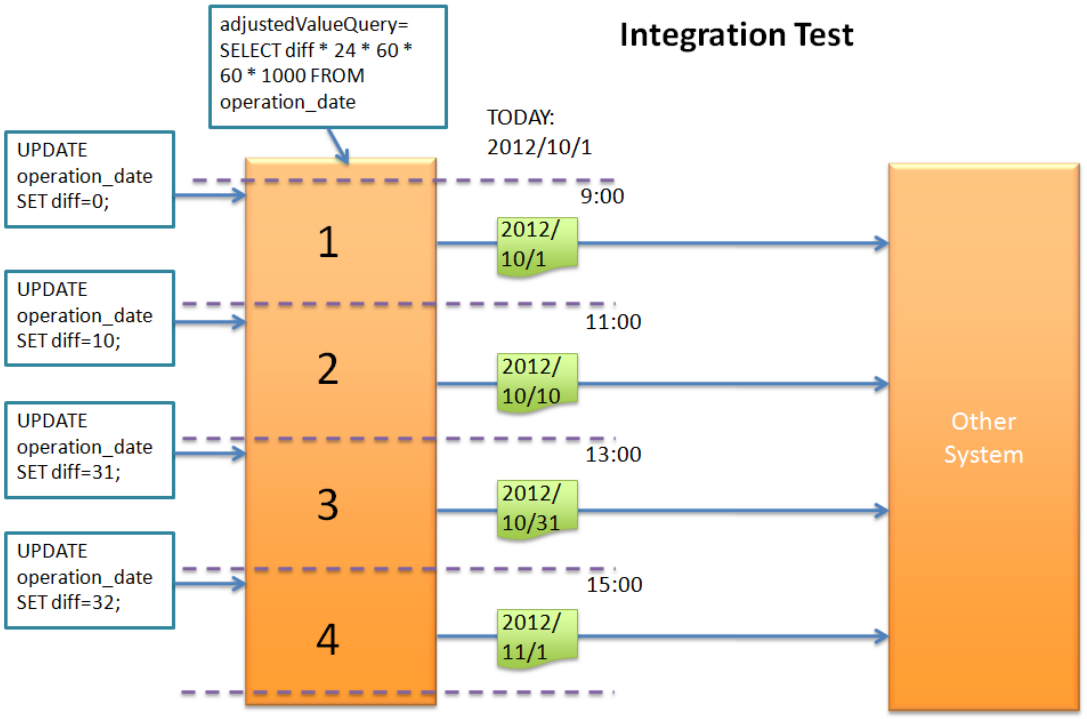
```
@Before
public void setUp() {
    service = new ReserveServiceImpl();

    // omitted

    Reserve reserveResult = new Reserve();
    reserveResult.setDepDay(new LocalDate(2012, 10, 10)); // (5)
    when(reserveRepository.findOne((String) anyObject())).thenReturn(
            reserveResult);
    dateFactory = mock(DateFactory.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。



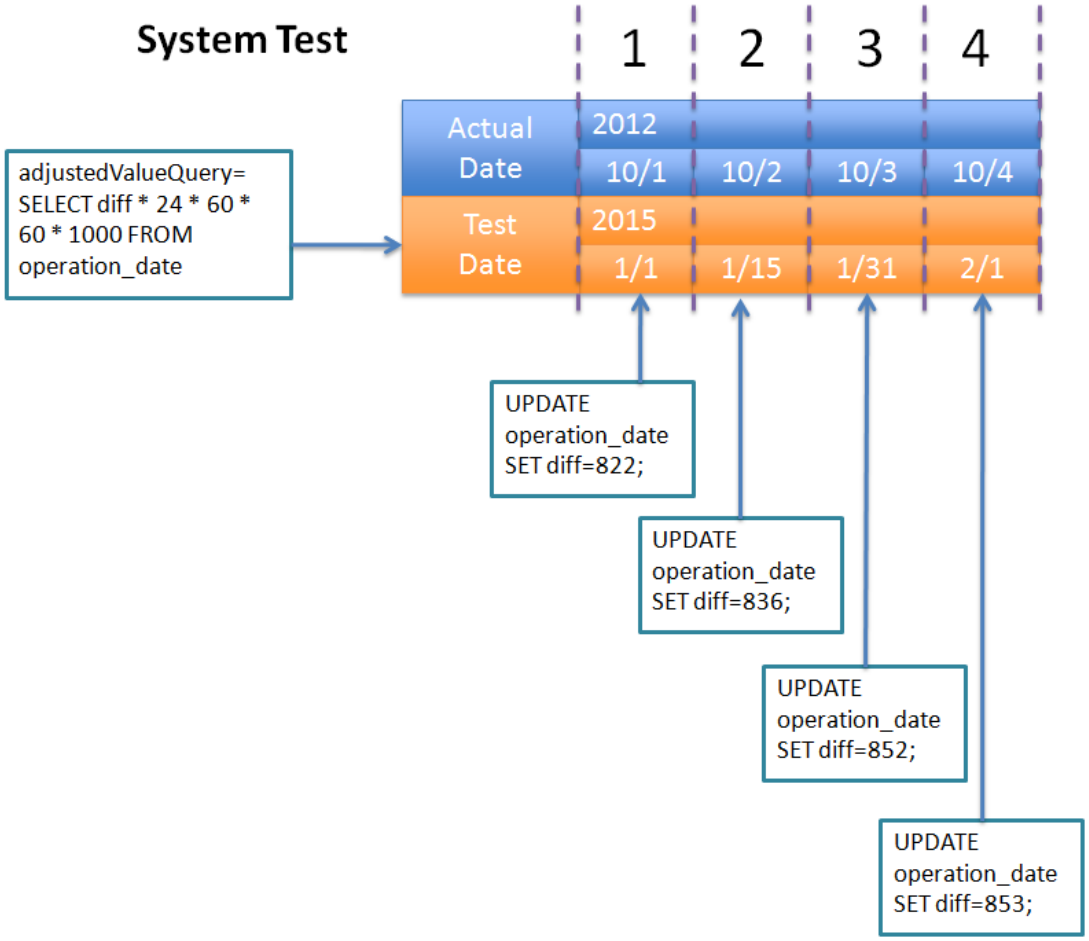
実際の日付が 2012/10/1 の場合、JdbcAdjustedDateFactory を使用し、試験対象の日付との差分を計算する SQL を設定する。

項番	説明
1	9:00-11:00 の間は差分値を”0 days” とし、dateFactory 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、dateFactory 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、dateFactory 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、dateFactory 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。



JdbcAdjustedDateFactory 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

JdbcAdjustedDateFactory を使用し、差分値を 0 とすることで、ソースを変更せず dateFactory 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、dateFactory 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、DefaultDateFactory に設定ファイルを変更することを推奨する。

5.21 ユーティリティ

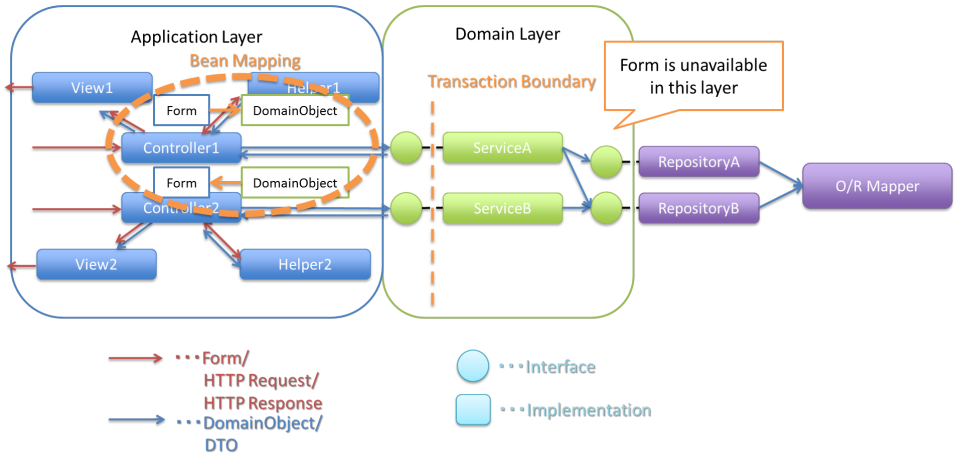
5.21.1 Bean マッピング (Dozer)

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。

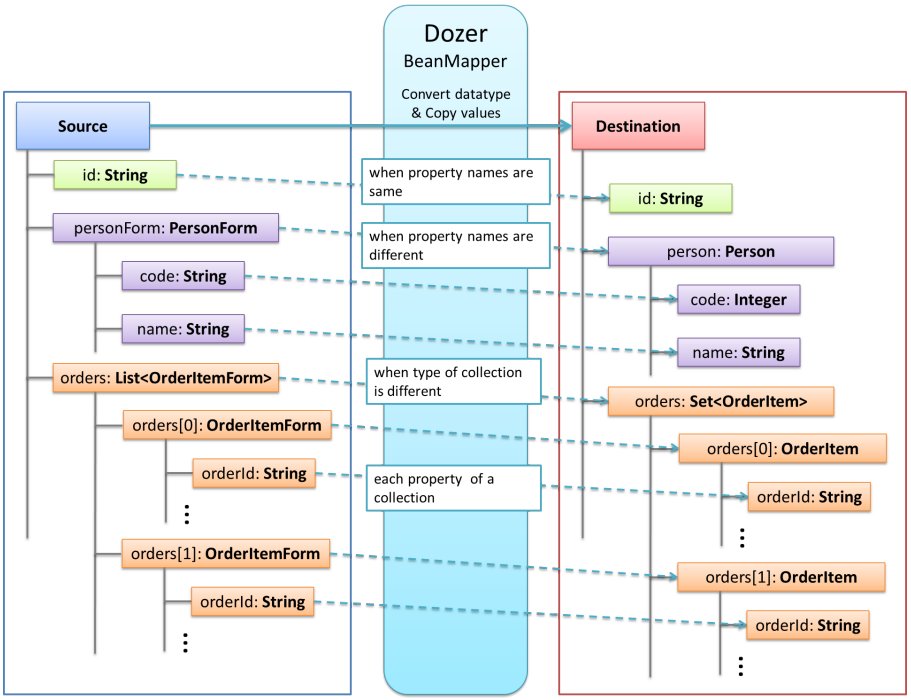
例として、アプリケーション層の `AccountForm` オブジェクトを、ドメイン層の `Account` オブジェクトに変換する場合を考える。
ドメイン層は、アプリケーション層に依存してはならないため、AccountForm オブジェクトをそのままドメイン層で使用できない。
そこで、`AccountForm` オブジェクトを、`Account` オブジェクトに Bean マッピングし、ドメイン層では、`Account` オブジェクトを使用する。
これによって、アプリケーション層と、ドメイン層の依存関係を一方向に保つことができる。



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。

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;
    // ommited setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int id;
    private String name;
    // ommited 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;
    // ommited setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int id;
    private String name;
    private String title;
    // ommited 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;
    // ommited setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private BigDecimal amount;
    // ommited 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;
    // ommited setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int destinationId;
    private String destinationName;
    // ommited 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-instanc
    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。

単方向を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-instanc
    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;
    // ommited setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int destinationId;
    private String destinationName;
```

```
    // ommited 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
```

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-instanc
    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-instanc
    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。

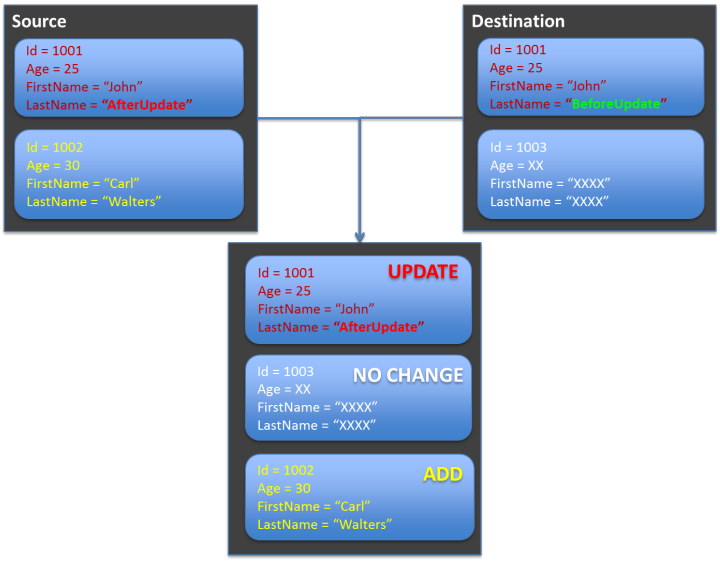
この設定のもと、前述のコードを実行すると以下のように出力される。

```
[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-instanc
    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。

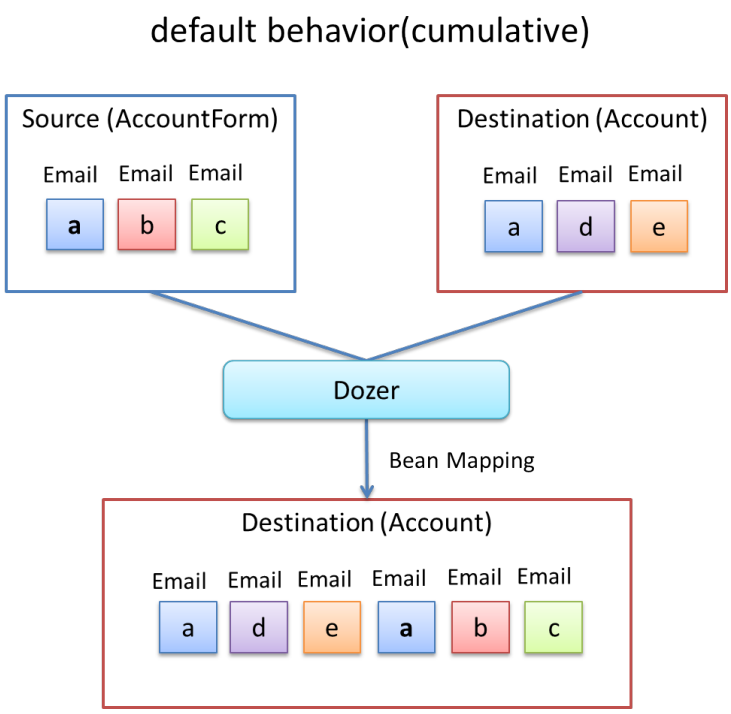
```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
    xsi:schemaLocation="http://dozer.sourceforge.net
          http://dozer.sourceforge.net/schema/beanmapping.xsd">
```

```
    <!-- 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。

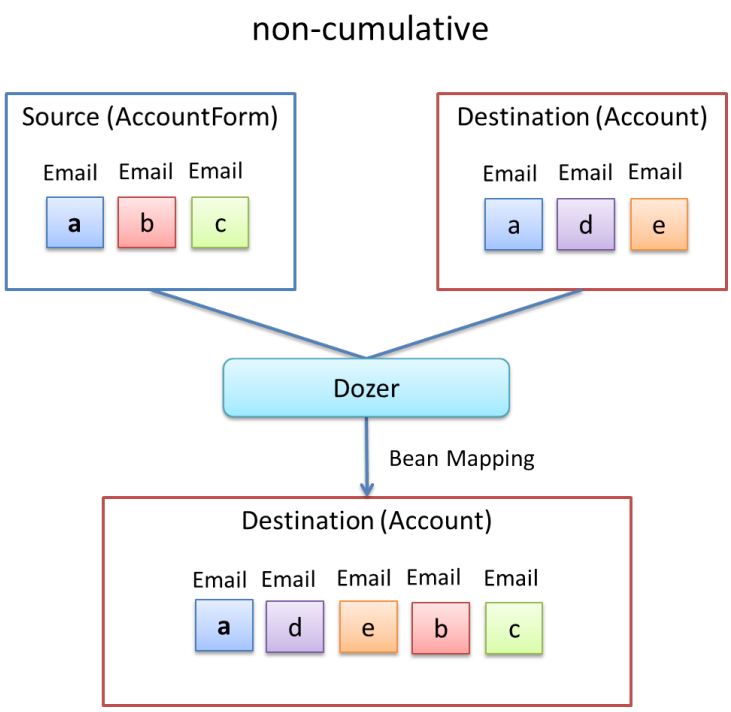
これまでの挙動を図で表現する。

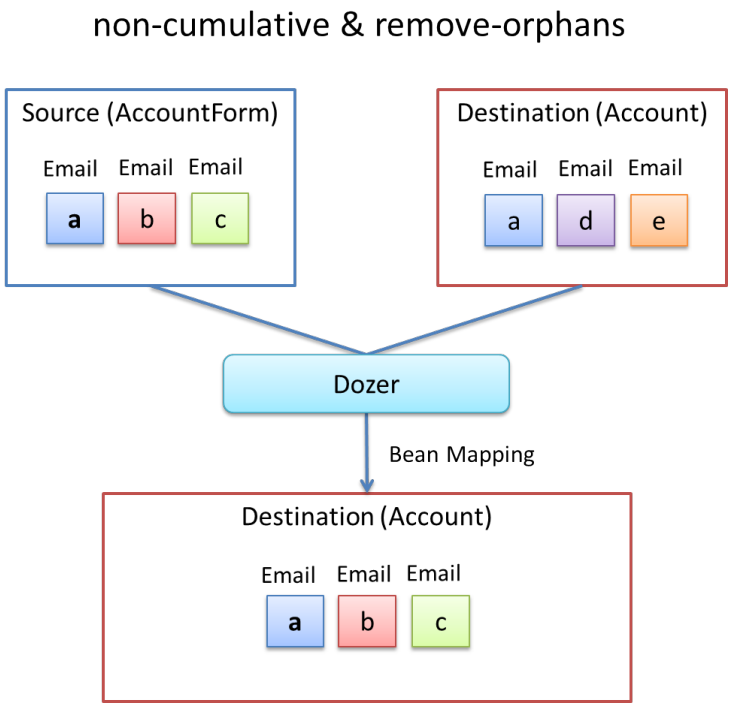
- デフォルトの挙動 (cumulative)



- non-cumulative

- non-cumulative かつ remove-orphans=true


non-cumulative
Source (AccountForm)
Email Email Email
a b c
Destination (Account)
Email Email Email
a d e
Dozer
Bean Mapping
Destination (Account)
Email Email Email Email Email
a d e b c

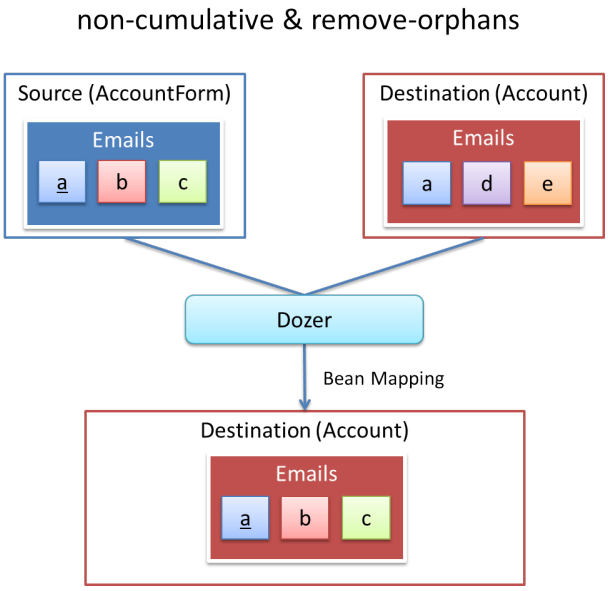

non-cumulative & remove-orphans
Source (AccountForm)
Email Email Email
a b c
Destination (Account)
Email Email Email
a d e
Dozer
Bean Mapping
Destination (Account)
Email Email Email
a b c

copy-by-reference もこのパターンである。

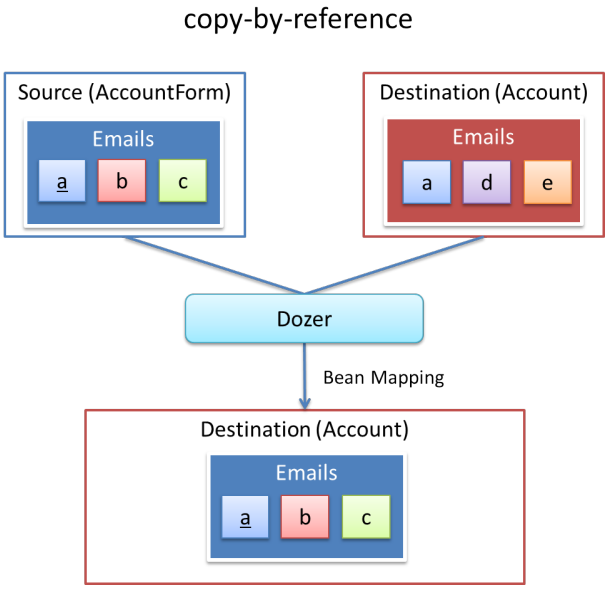
注釈: 「non-cumulative かつ remove-orphans=true」のパターンと「copy-by-reference」のパターンの違いは、Bean 変換後の Collection のコンテナがコピー先のものか、コピー元のものかで異なる点である。

「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、 マニュアル を参照されたい。

(Using Hints for Collection Mapping)

課題

Collection<T>を使用した Bean 間のマッピングは失敗することが確認されている。

例 :

```
public class ListNestedBean<T> {
   private List<T> nest;
   // omitted other declarations
}
```

実行結果 :

```
java.lang.ClassCastException: sun.reflect.generics.reflectiveObjects.TypeVariableImpl can
```

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の場合、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-instanc
    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
                <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;
```

```
    // ommited setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int id;
    private DateTime date;
    // ommited 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、 マニュアル 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。

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 のマニュアル 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;
    // ommited setter/getter
}
```

コピー先の Bean 定義

```
public class Destination {
    private int id;
    private String name;
    private String title;
    // ommited 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-instanc
    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-instanc
    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, "mapidSourceBeanFieldExclude"); // (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、 マニュアル 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-instanc
    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;
    // ommited setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private int id;
    private String name;
    private String title;
    // ommited 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-instanc
    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;
    // ommited setter/getter
}
```

変換先の Bean 定義

```
public class Destination {
    private Date date;
    // ommited 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("2013-1
```

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で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-instanc
    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、 マニュアル 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。

5.21.2 日付操作 (Joda Time)

Overview

java.util.Date 、 java.util.Calender 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 オブジェクトを用いて日付を表現する。

なお、 org.joda.time.DateTime 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)。

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 ,
org.joda.time.DateMidnight 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. 0 時に設定された現在日時を取得したい場合は、 org.joda.time.DateMidnight を使用する。

```
DateMidnight dateMidnight = new DateMidnight();
```

注釈: LocalDate と LocalTime は、TimeZone 情報を持たないが、DateMidnight は、TimeZone 情報を持つ。

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.DateFactory` 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、DateMidnight の生成は

```
        LocalDate localDate = dataFactory.newDateTime().toLocalDate();
        LocalTime localTime = dataFactory.newDateTime().toLocalTime();
        DateMidnight dateMidnight = dataFactory.newDateTime().toDateMidnight();
```

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.DateFactory` 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.DateFactory` 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

```
DateTime dateTime = DateTimeFormat.forPattern("yyyy-MM-dd").parseDateTime("2012-08-09");  // (1)
```

項番	説明
(1)	“yyyy-MM-dd” 形式の文字列を、DateTime 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

DateTime 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。

```
DateTime dateTime = new DateTime(); // dateTime is 2013-01-10T13:30:22.123Z
DateTime yesterday = dateTime.minusDays(1);  // (1)
DateTime tomorrow = dateTime.plusDays(1);  // (2)
DateTime afterThreeMonth = dateTime.plusMonths(3);  // (3)
DateTime nextYear = dateTime.plusYears(1);  // (4)
```

項番	説明
(1)	DateTime#minusDays 引数に、指定した値分の日付が減算される。本例では `2013-01-09T13:30:22.123Z` となる。
(2)	DateTime#plusDays 引数に、指定した値分の日付が加算される。本例では `2013-01-11T13:30:22.123Z` となる。
(3)	DateTime#plusMonths 引数に、指定した値分の月数が加算される。本例では `2013-04-10T13:30:22.123Z` となる。
(4)	DateTime#plusYears 引数に、指定した値分の年数が加算される。本例では `2014-01-10T13:30:22.123Z` となる。

上記で示したメソッド以外は、 DateTime 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。

下記の例では、時・分・秒・ミリ秒は、new DateTime() で取得した値のままとなる。

```
DateTime dateTime = new DateTime(); // dateTime is 2013-01-10T13:30:22.123Z
Property dayOfMonth = dateTime.dayOfMonth();  // (1)
DateTime firstDayOfMonth = dayOfMonth.withMinimumValue();  // (2)
DateTime 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 `2013-01-01T13:30:22.123Z` 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 `2013-01-31T13:30:22.123Z` 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。

下記の例では、時・分・秒・ミリ秒は、new DateTime() で取得した値のままとなる。

```
DateTime dateTime = new DateTime(); // dateTime is 2013-01-10T13:30:22.123Z
Property dayOfWeek = dateTime.dayOfWeek();  // (1)
DateTime firstDayOfWeek = dayOfWeek.withMinimumValue();  // (2)
DateTime 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 `2013-01-07T13:30:22.123Z` 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 `2013-01-13T13:30:22.123Z` 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。

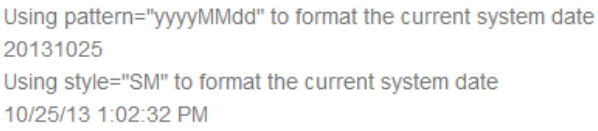
設定方法 タブライブラリを利用するには、以下の taglib 定義が必要である。

```
<%@ 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



joda:format タグの属性一覧は、以下の通りである。

TABLE 5.25 **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	処理メソッド
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) {
        DateTime today = new DateTime();
        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) {
        DateTime firstDayOfMonth = new DateTime(year, month, 1, 0, 0);
        DateTime lastDayOfMonth = firstDayOfMonth.dayOfMonth()
                .withMaximumValue();

        DateTime firstDayOfCalender = firstDayOfMonth.dayOfWeek()
                .withMinimumValue();
        DateTime lastDayOfCalender = lastDayOfMonth.dayOfWeek()
                .withMaximumValue();

        List<List<DateTime>> calendar = new ArrayList<List<DateTime>>();
        List<DateTime> weekList = null;
        for (int i = 0; i < 100; i++) {
            DateTime d = firstDayOfCalender.plusDays(i);
            if (d.isAfter(lastDayOfCalender)) {
                break;
            }
```

```
            if (weekList == null) {
                weekList = new ArrayList<DateTime>();
                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;
            }
        }

        DateTime nextMonth = firstDayOfMonth.plusMonths(1);
        DateTime 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<DateTime>> calendar;

    private DateTime firstDayOfMonth;

    private int yearOfNextMonth;

    private int monthOfNextMonth;

    private int yearOfPrevMonth;

    private int monthOfPrevMonth;

    // ommited getter/setter
}
```

警告: このサンプルコードは単純なため Controller の処理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.yearOfPrev
            Prev</a> <a
            href="${pageContext.request.contextPath}/calendar/month?year=${f:h(output.yearOfNext
            &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。

← Prev Next →
2012-11

Mon.	Tue.	Wed.	Thu.	Fri.	Sat.	Sun.
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

← Prev Next →
2012-12

Mon.	Tue.	Wed.	Thu.	Fri.	Sat.	Sun.
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

第 6 章

TERASOLUNA Global Framework によるセキュリティ対策

6.1 Spring Security 概要

6.1.1 Overview

Spring Security とは、アプリケーションのセキュリティを担う「認証」、「認可」の 2 つを
主な機能として提供している。
認証機能とは、なりすましによる不正アクセスに対抗するため、ユーザを識別する機能である。
認可機能とは、認証された（ログイン中の）ユーザの権限に応じて、
システムのリソースに対するアクセス制御を行う機能である。

Spring Security の概要図を、以下に示す。

Spring Security は、認証、認可のプロセスを何層にも連なる
ServletFilter の集まりで実現している。
また、パスワードハッシュ機能や、JSP の認可タグライブラリなども提供している。

認証

認証とは、正当性を確認する行為であり、ネットワークやサーバへ接続する際に
ユーザ名とパスワードの組み合わせを使って、利用ユーザにその権利があるかどうかや、
その人が利用ユーザ本人であるかどうかを確認することである。

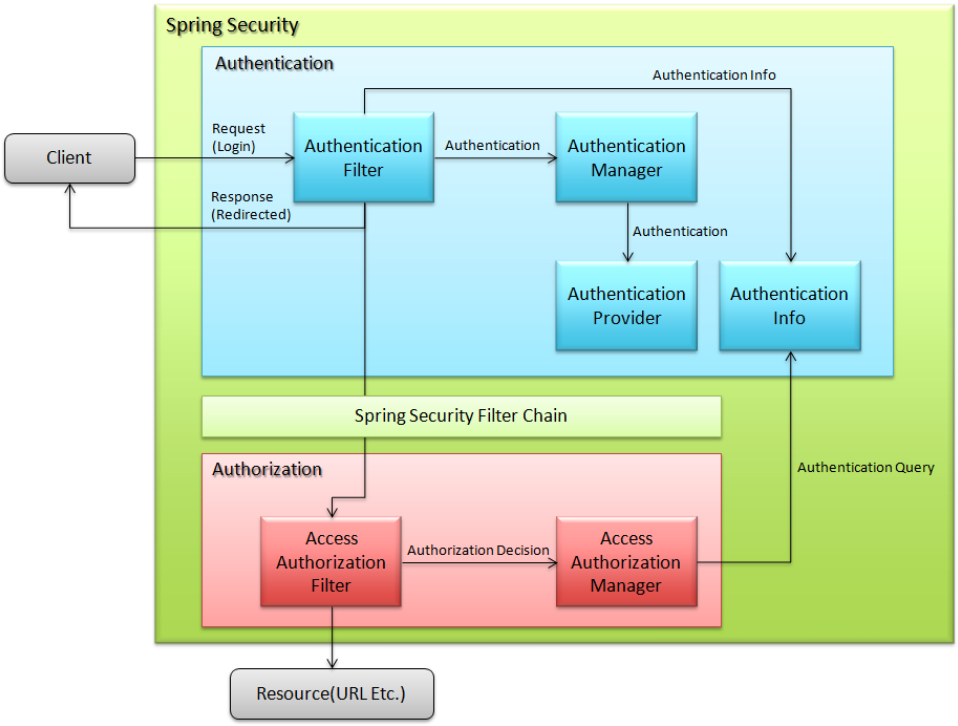


図 6.1 **Picture - Spring Security Overview**

Spring Security での使用方法は、認証を参照されたい。

パスワードハッシュ

平文のパスワードから、ハッシュ関数を用いて計算されたハッシュ値を、元のパスワードと置き換えることである。

Spring Security での使用方法は、パスワードハッシュ化を参照されたい。

認可

認可とは、認証された利用者がリソースにアクセスしようとしたとき、

アクセス制御処理でその利用者がそのリソースの使用を許可されていることを調べることである。

Spring Security での使用方法は、認可を参照されたい。

6.1.2 How to use

Spring Security を使用するために、以下の設定を定義する必要がある。

pom.xml の設定

Spring Security を使用する場合、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-security-core</artifactId>  <!-- (1) -->
</dependency>

<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-security-web</artifactId>  <!-- (2) -->
</dependency>
```

項番	説明
(1)	terasoluna-gfw-security-core は、web に依存しないため、ドメイン層のプロジェクトから使用する場合は、 terasoluna-gfw-security-core のみを dependency に追加すること。
(2)	terasoluan-gfw-web は web に関連する機能を提供する。terasoluna-gfw-security-core にも依存しているため、 Web プロジェクトは、terasoluna-gfw-security-web のみを dependency に追加すること。

Web.xml の設定

```
 <context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>  <!-- (1) -->
       classpath*:META-INF/spring/applicationContext.xml
       classpath*:META-INF/spring/spring-security.xml
   </param-value>
 </context-param>
 <listener>
   <listener-class>
     org.springframework.web.context.ContextLoaderListener
   </listener-class>
```

```
</listener>
<filter>
  <filter-name>springSecurityFilterChain</filter-name>  <!-- (2) -->
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  <!-- (3) -->
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>  <!-- (4) -->
</filter-mapping>
```

項番	説明
(1)	contextConfigLocation には、applicationContext.xml に加えて、 クラスパスに Spring Security 設定ファイルを追加する。本ガイドラインでは、 「spring-security.xml」とする。
(2)	filter-name には、Spring Security の内部で使用される Bean 名、「springSecurityFilterChain」 で定義すること。
(3)	各種機能を有効にするための、Spring Security のフィルタ設定。
(4)	全てのリクエストに対して設定を有効にする。

spring-security.xml の設定

web.xml において指定したパスに、spring-security.xml を配置する。

通常は src/main/resources/META-INF/spring/spring-security.xml に設定する。

以下の例は、雛形のみであるため、詳細な説明は、次章以降を参照されたい。

- spring-mvc.xml

```
    <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:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/security
            http://www.springframework.org/schema/security/spring-security.xsd
            http://www.springframework.org/schema/beans
```

```
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd">
        <sec:http  use-expressions="true">  <!-- (1) -->
          <!-- omitted -->
        </sec:http>
    </beans>
```

項番	説明
(1)	use-expressions=”true” と記載することで、アクセス属性の Spring EL 式を有効することができる。

注釈: use-expressions=”true” で有効になる Spring EL 式は、以下を参照されたい。

Expression-Based Access Control

6.2 Spring Security チュートリアル

6.2.1 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

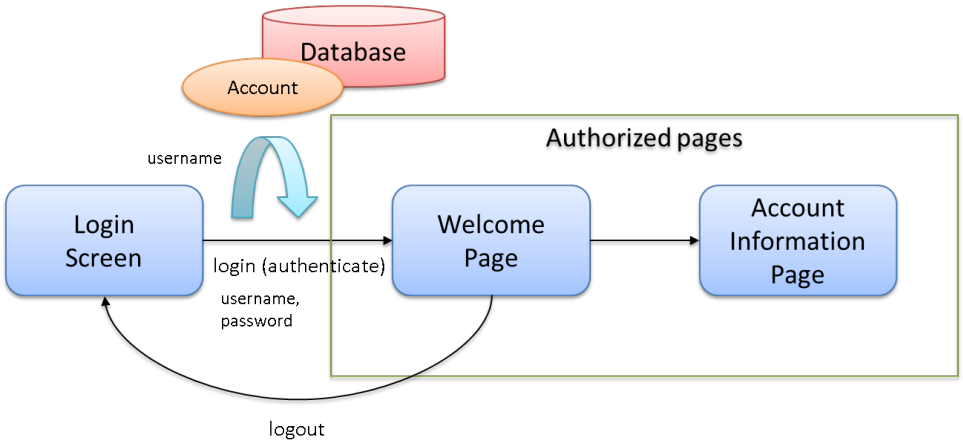
- チュートリアル (Todo アプリケーション) を実施ずみ
- Maven の基本的な操作を理解している

6.2.2 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

- ログインできる。ログインのためのアカウント情報はデータベース上に格納されている。
- Welcome 画面、アカウント情報表示画面があり、ログインしないと閲覧できない。
- ログアウトできる。

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.jsp	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	ログイン中のアカウント情報を表示する
5	ログアウト	GET	/logout	ログアウトする (Spring Security が行う)

6.2.3 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 Global Framework のブランクプロジェクトを作成する。

Spring tool suite 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は、Blank プロジェクトから新規プロジェクト作成を参照されたい。

```
mvn archetype:generate -B^
 -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-releases
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-mybatis2-archetype^
 -DarchetypeVersion=1.0.4.RELEASE^
 -DgroupId=com.example.security^
 -DartifactId=first-springsecurity^
 -Dversion=1.0-SNAPSHOT
```

コンソール上に以下のようなログが表示されれば、ブランクプロジェクトの作成は成功となる。

```
C:\work>mvn archetype:generate -B^
More?  -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-re
More?  -DarchetypeGroupId=org.terasoluna.gfw.blank^
More?  -DarchetypeArtifactId=terasoluna-gfw-web-blank-mybatis2-archetype^
More?  -DarchetypeVersion=1.0.4.RELEASE^
More?  -DgroupId=com.example.security^
More?  -DartifactId=first-springsecurity^
More?  -Dversion=1.0-SNAPSHOT
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom >>>
```

```
[INFO]
[INFO] <<< maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] Archetype repository missing. Using the one from [org.terasoluna.gfw.blank:terasoluna-gfw-w
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: terasoluna-gfw-web-blank-my
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.example.security
[INFO] Parameter: artifactId, Value: first-springsecurity
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.example.security
[INFO] Parameter: packageInPathFormat, Value: com/example/security
[INFO] Parameter: package, Value: com.example.security
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.example.security
[INFO] Parameter: artifactId, Value: first-springsecurity
[INFO] project created from Archetype in dir: C:\work\first-springsecurity
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 10.207s
[INFO] Finished at: Wed Aug 27 14:14:48 JST 2014
[INFO] Final Memory: 9M/23M
[INFO] ------------------------------------------------------------------------
C:\work>
```

6.2.4 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。以下の順に進める。

1. Domain Object(Account) の作成
2. AccountRepository の作成
3. AccountService の作成

Account テーブルは次のものを使用する (DDL スクリプトは後程作成する)。

```
CREATE TABLE account(
    username varchar(128),
    password varchar(128),
    first_name varchar(128),
    last_name varchar(128),
    constraint pk_tbl_account primary key (username)
);
```

Domain Object の作成

次の Account クラスを用意する。このクラスが認証情報 (ユーザー名とパスワード) を持つ。

- 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

AccountRepository にはユーザー名から Account オブジェクトを取得するデータアクセス処理を実装する。

- src/main/java/com/example/security/domain/repository/account/AccountRepository.java

  まずはインタフェースを定義する。ユーザー名から Account オブジェクトを取得する findOne(username) を定義する。

```
package com.example.security.domain.repository.account;

import com.example.security.domain.model.Account;

public interface AccountRepository {
    Account findOne(String username);
}
```

- src/main/java/com/example/security/domain/repository/account/AccountRepositoryImpl.java

  データアクセス処理を AccountRepositoryImpl に実装する。

```
package com.example.security.domain.repository.account;

import javax.inject.Inject;

import org.springframework.stereotype.Repository;

import jp.terasoluna.fw.dao.QueryDAO;

import com.example.security.domain.model.Account;
```

```
@Repository
public class AccountRepositoryImpl implements AccountRepository {
    @Inject
    QueryDAO queryDAO;

    @Override
    public Account findOne(String username) {
        Account account = queryDAO.executeForObject("account.findOne",
                username, Account.class);
        return account;
    }

}
```

- src/main/resources/META-INF/mybatis/sql/account-sqlmap.xml

  Account を 1 件取得するための SQLID"account.findOne"に対応する SQL を SQLMap ファイルに定義する。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
            PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
            "http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="account">
    <resultMap id="account"
        class="com.example.security.domain.model.Account">
        <result 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" parameterClass="java.lang.String"
        resultMap="account"><![CDATA[
SELECT username,
       password,
       first_name,
       last_name
FROM   account
WHERE  username = #value#
]]></select>
</sqlMap>
```

AccountService の作成

- src/main/java/com/example/security/domain/service/account/AccountService.java

AccountService にはユーザー名から Account オブジェクトを取得する業務処理を実装する。

この処理は後ほど Spring Security の認証サービスから利用されるので、クラス名は AccountSharedService とする。

```
package com.example.security.domain.service.account;

import com.example.security.domain.model.Account;

public interface AccountSharedService {
    Account findOne(String username);
}
```

- src/main/java/com/example/security/domain/service/account/AccountServiceImpl.java

データアクセスの結果、該当する Account が存在しない場合は、ResourceNotFoundException をスローする。

```
package com.example.security.domain.service.account;

import javax.inject.Inject;

import org.springframework.stereotype.Service;
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;

    @Override
    public Account findOne(String username) {
        Account account = accountRepository.findOne(username);
        if (account == null) {
            throw new ResourceNotFoundException("The given account is not found! username="
                    + username);
        }
        return account;
    }

}
```

認証サービスの作成

Spring Security で使用する認証ユーザー情報は org.springframework.security.core.userdetails.UserDetai
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。

- src/main/java/com/example/security/domain/service/userdetails/SampleUserDetails.java

```
package com.example.security.domain.service.userdetails;

import java.util.Collection;
import java.util.Collections;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.User;

import com.example.security.domain.model.Account;

public class SampleUserDetails extends User {
    private static final long serialVersionUID = 1L;

    private final Account account; // (1)

    public SampleUserDetails(Account account) {
        super(account.getUsername(), account.getPassword(), createRole(account)); // (2)
        this.account = account;

    }

    private static Collection<? extends GrantedAuthority> createRole(
            Account account) {
        // sample role
        return Collections
                .singletonList(new SimpleGrantedAuthority("ROLE_USER")); // (3)
    }

    public Account getAccount() { // (4)
        return account;
    }
}
```

項番	説明
(1)	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。
(2)	`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。
(3)	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。
(4)	アカウント情報の getter を用意する。これにより、ログイン中の `Account` オブジェクトを取得することができる。

- 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.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 {
    @Inject
    AccountSharedService accountSharedService; // (1)

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
        try {
            Account account = accountSharedService.findOne(username); // (2)
            return new SampleUserDetails(account); // (3)
        } catch (ResourceNotFoundException e) {
            throw new UsernameNotFoundException("user not found", e); // (4)
        }
    }
```

```
    }
```

項番	説明
(1)	`AccountSharedService` をインジェクションする。 本ガイドラインでは、Service からは Service の呼び出しは非推奨であり、 `AccountService` ではなく、`AccountSharedService` という名前にしている。
(2)	`username` から `Account` オブジェクトを取得する処理を `AccountSharedService` に委譲する。
(3)	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する。
(4)	`UserDetailsService` は対象のユーザーが見つからない場合には `UsernameNotFoundException` がスローされる仕様となっている。

ドメイン層の作成後のパッケージエクスプローラー

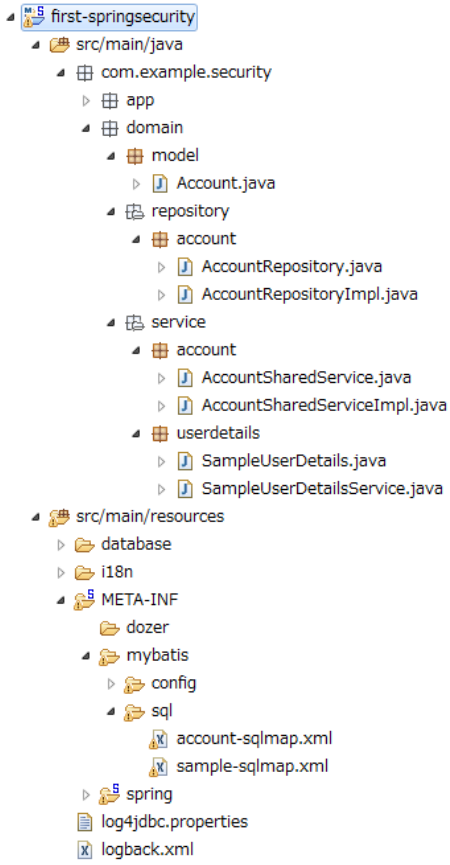
Package Explorer の Package Presentation は Hierarchical を使用している。

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。

特に URL に関する設定項目を以下に再掲する。



設定項目名	設定値
ログインフォームの URL	/login.jsp
認証失敗時に遷移する URL	/login.jsp?error=true
認証処理の URL	/authenticate
ログアウトの URL	/logout
ログアウト後に遷移する URL	/

blank プロジェクトからの差分のみ説明する。

- 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.springframe
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/security http://www.springfra
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/b
        http://www.springframework.org/schema/context http://www.springframework.org/schema

    <sec:http pattern="/resources/**" security="none" />
    <sec:http auto-config="true" use-expressions="true">
        <sec:form-login login-page="/login.jsp"
            authentication-failure-url="/login.jsp?error=true"
            login-processing-url="/authenticate" /><!-- (1) -->
        <sec:logout logout-url="/logout" logout-success-url="/"
            delete-cookies="JSESSIONID" /><!-- (2) -->

        <sec:intercept-url pattern="/login.jsp"
            access="permitAll" /><!-- (3) -->
        <sec:intercept-url pattern="/**" access="isAuthenticated()" /><!-- (4) -->

        <sec:custom-filter ref="csrfFilter" before="LOGOUT_FILTER" />
        <sec:custom-filter ref="userIdMDCPutFilter"
            after="ANONYMOUS_FILTER" />
        <sec:session-management
            session-authentication-strategy-ref="sessionAuthenticationStrategy" />
    </sec:http>


    <sec:authentication-manager>
        <!-- com.example.security.domain.service.userdetails.SampleUserDetails
            is scaned by component scan with @Service -->
        <sec:authentication-provider
            user-service-ref="sampleUserDetailsService"><!-- (5) -->
            <sec:password-encoder ref="passwordEncoder" /><!-- (6) -->
        </sec:authentication-provider>
    </sec:authentication-manager>

    <!-- CSRF Protection -->
    <bean id="csrfTokenRepository"
        class="org.springframework.security.web.csrf.HttpSessionCsrfTokenRepository" />

    <bean id="csrfFilter" class="org.springframework.security.web.csrf.CsrfFilter">
        <constructor-arg index="0" ref="csrfTokenRepository" />
        <property name="accessDeniedHandler">
            <bean
                class="org.springframework.security.web.access.AccessDeniedHandlerImpl">
                <property name="errorPage"
```

```
                        value="/WEB-INF/views/common/error/csrfTokenError.jsp" />
                </bean>
            </property>
        </bean>

        <bean id="sessionAuthenticationStrategy"
            class="org.springframework.security.web.authentication.session.CompositeSessionAuth
            <constructor-arg index="0">
                <list>
                    <bean
                        class="org.springframework.security.web.authentication.session.SessionF
                    <bean
                        class="org.springframework.security.web.csrf.CsrfAuthenticationStrategy
                        <constructor-arg index="0"
                            ref="csrfTokenRepository" />
                    </bean>
                </list>
            </constructor-arg>
        </bean>

        <!-- Put UserID into MDC -->
        <bean id="userIdMDCPutFilter"
            class="org.terasoluna.gfw.security.web.logging.UserIdMDCPutFilter">
            <property name="removeValue" value="true" />
        </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。 login-page 属性にログインフォームの URL、authentication-failure-url 属性に認証失敗時に遷移する URL、login-processing-url 属性に認証処理の 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。logout-url 属性にログアウトの URL、logout-success-url 属性にログアウト後に遷移する URL を設定する。 また delete-cookies 属性にログアウト時に削除する Cookie 名を指定できる。
(3)	<sec:intercept-url>タグで URL レベルでの認可設定を行う。ログインフォームは全ユーザーのアクセスを許可する permitAll を指定する。
(4)	この設定より上に設定されている/resources/**、/login.jsp を除くすべての URL に対し、認証済みユーザーのみアクセスを許可する isAuthenticated() を指定する。
(5)	<sec:authentication-provider>タグで認証処理を実施する 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、フォームから入力されたパスワードを org.springframework.security.crypto.password.PasswordEncoder を使用してハッシュ化したものを比較してユーザー認証を行う org.springframework.security.authentication.dao.DaoAuthenticationProvider が使用される。
(6)	PasswordEncoder の設定を行う。ここでは applicationContext.xml に定義されている org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder を参照する。

起動時に実行される SQL スクリプトの設定

- src/main/resources/META-INF/spring/first-springsecurity-env.xml

  SQL スクリプトの設定を追加する。

```
<?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:jdbc="http://www.springfram
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc http://www.springframewo
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/b

    <bean id="dateFactory" class="org.terasoluna.gfw.common.date.DefaultDateFactory" />

    <bean id="realDataSource" class="org.apache.commons.dbcp.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="maxActive" value="${cp.maxActive}" />
        <property name="maxIdle" value="${cp.maxIdle}" />
        <property name="minIdle" value="${cp.minIdle}" />
        <property name="maxWait" value="${cp.maxWait}" />
    </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>

    <jdbc:initialize-database data-source="dataSource"
        ignore-failures="ALL"><!-- (1) -->
        <jdbc:script location="classpath:/database/${database}-schema.sql" /><!-- (2) -->
        <jdbc:script location="classpath:/database/${database}-dataload.sql" /><!-- (3) -->
    </jdbc:initialize-database>
</beans>
```

項番	説明
(1)	<jdbc:initialize-database>タグで初期化 SQL スクリプトの設定を行う。 この設定は通常、開発中のみでしか使用しないため、xxx-env.xml に定義する。
(2)	DDL を設定する。雛形の設定では xxx-infra.properties に database=H2 と定義されているため、H2-schema.sql が実行される。
(3)	DML を設定する。雛形の設定では xxx-infra.properties に database=H2 と定義されているため、H2-dataload.sql が実行される。

今回はインメモリの H2 データベースを利用する。DDL と DML を以下のように用意する。

- src/main/resources/database/H2-schema.sql

```
CREATE TABLE account(
    username varchar(128),
    password varchar(128),
    first_name varchar(128),
    last_name varchar(128),
    constraint pk_tbl_account primary key (username)
);
```

- src/main/resources/database/H2-dataload.sql

  username=demo、passowrd=demo でログインできるテストユーザーを追加する。

```
INSERT INTO account(username, password, first_name, last_name) VALUES('demo', '$2a$10$oxSJl.
COMMIT;
```

項番	説明
(1)	雛形の設定では applicationContext.xml にパスワードハッシュ化のために org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder が設定されている。 テストデータとして、BCrypt アルゴリズムでハッシュ化された”demo” という文字列を投入する。

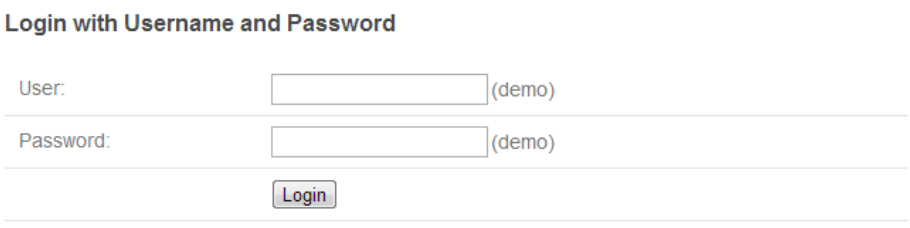
ログイン画面の作成

- src/main/webapp/login.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>

        <c:if test="${param.error}"><!-- (1) -->
            <t:messagesPanel messagesType="error"
                messagesAttributeName="SPRING_SECURITY_LAST_EXCEPTION" /><!-- (2) -->
        </c:if>

        <form action="${pageContext.request.contextPath}/authenticate"
            method="POST"><!-- (3) -->
            <table>
                <tr>
                    <td><label for="j_username">User:</label></td>
                    <td><input type="text" id="j_username"
                        name="j_username" value='demo'>(demo)</td><!-- (4) -->
                </tr>
                <tr>
                    <td><label for="j_password">Password:</label></td>
                    <td><input type="password" id="j_password"
                        name="j_password" value="demo" />(demo)</td><!-- (5) -->
                </tr>
                <tr>
                    <td> </td>
                    <td><input type="hidden"
                        name="${f:h(_csrf.parameterName)}"
                        value="${f:h(_csrf.token)}" /> <input
                        name="submit" type="submit" value="Login" /></td><!-- (6) -->
                </tr>
            </table>
        </form>
    </div>
</body>
</html>
```

項番	説明
(1)	認証が失敗した場合は”/login.jsp?error=true” が呼ばれるように設定した。この場合に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)	認証が失敗した場合はセッションスコープに例外オブジェクトが属性名 "SPRING_SECURITY_LAST_EXCEPTION"で格納される。 ここでは<t:messagesPanel>タグを使用してエラーメッセージを表示する。
(3)	認証処理の URL を”/authenticate” と設定した。認証するためのこの URL でユーザー名とパスワードを POST する。
(4)	ユーザー名のリクエストパラメータ名はデフォルトで j_username である。
(5)	パスワードのリクエストパラメータ名はデフォルトで j_password である。

ブラウザのアドレスバーに http://localhost:8080/first-springsecurity/を入力し、表示しようとすると未ログイン状態のため、 *Spring Security* **の設定** の (1) の定義により http://localhost:8080/first-springsecurity/login.jsp にアクセスとなり、下の画面が表示される。



JSP からログインアカウント情報にアクセスする

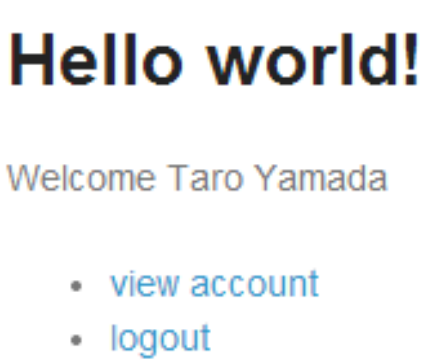
- src/main/webapp/WEB-INF/views/welcome/home.jsp

  以下のコードを追加する。

```
<!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">
        <sec:authentication property="principal.account" var="account" /><!-- (1) -->
        <h1>Hello world!</h1>
        <p>Welcome ${f:h(account.firstName)} ${f:h(account.lastName)}</p><!-- (2) -->

        <ul>
            <li><a href="${pageContext.request.contextPath}/account">view account</a></li>
            <li><a href="${pageContext.request.contextPath}/logout">logout</a></li>
        </ul>
    </div>
</body>
</html>
```

項番	説明
(1)	<sec:authentication>タグで、ログイン中の org.springframework.security.core.Authentication オブジェクトにアクセスできる。 property 属性で.Authentication オブジェクトの任意のプロパティにアクセスでき、var 属性で任意のスコープに設定できる。デフォルトでは page スコープの設定され、この JSP 内のみで参照可能である。 ここではログイン中の Account オブジェクトを変数名 account に格納する。
(2)	ログイン中の Account オブジェクトにアクセスして firstName と lastName を表示する。

Login ページの Login ボタンを押下することにより、Welcome ページが表示される。



ログインアカウント情報表示ページの作成

- src/main/java/com/example/security/app/account/AccountController.java

  ログイン済みの `UserDetails` オブジェクトは `java.security.Principal` オブジェクト内に格納されている。Controller の処理メソッドの引数で `Principal` オブジェクトを受け取ることにより、Controller 内でログイン済みの `UserDetails` オブジェクトにアクセスすることができる。

```
package com.example.security.app.account;

import java.security.Principal;

import org.springframework.security.core.Authentication;
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.security.domain.model.Account;
import com.example.security.domain.service.userdetails.SampleUserDetails;

@Controller
@RequestMapping("account")
public class AccountController {

    @RequestMapping(method = RequestMethod.GET)
    public String view(/* (1) */ Principal principal, Model model) {
        // get login user information
        Authentication authentication = (Authentication) principal; // (2)
        // get UserDetails
        SampleUserDetails userDetails = (SampleUserDetails) authentication
                .getPrincipal(); // (3)
```

```
            // get account object
            Account account = userDetails.getAccount(); // (4)
            model.addAttribute(account);
            return "account/view";
        }
    }
```

項番	説明
(1)	ログイン中の UserDetails オブジェクトを格納した Principal オブジェクトを受け取る。
(2)	org.springframework.security.core.Authentication は Principal インタフェースでもあり、コントローラに渡る Principal オブジェクトは実際は Authentication オブジェクトである。 UserDetails オブジェクトにアクセスするために、Authentication クラスにキャストする。
(3)	Authentication.getPrincipal() メソッドにより、ログイン中の UserDetails オブジェクトを取得できる。ここでは、本プロジェクト用の SampleUserDetails クラスにキャストする。
(4)	SampleUserDetails オブジェクトからログイン中の Account オブジェクトを取得する。

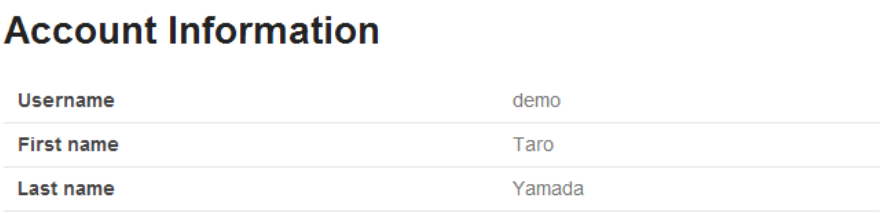
- src/main/webapp/WEB-INF/views/account/view.jsp

  Model に設定された Account オブジェクトの各プロパティを出力するだけであるので説明は省略する。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Home</title>
<link rel="stylesheet"
```

```
        href="${pageContext.request.contextPath}/resources/app/css/styles.css">
    </head>
    <body>
        <div id="wrapper">
            <h1>Account Information</h1>
            <table>
                <tr>
                    <th>Username</th>
                    <td>${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>
```

Welcome ページの view account リンクを押下することにより、Account Information ページが表示される。

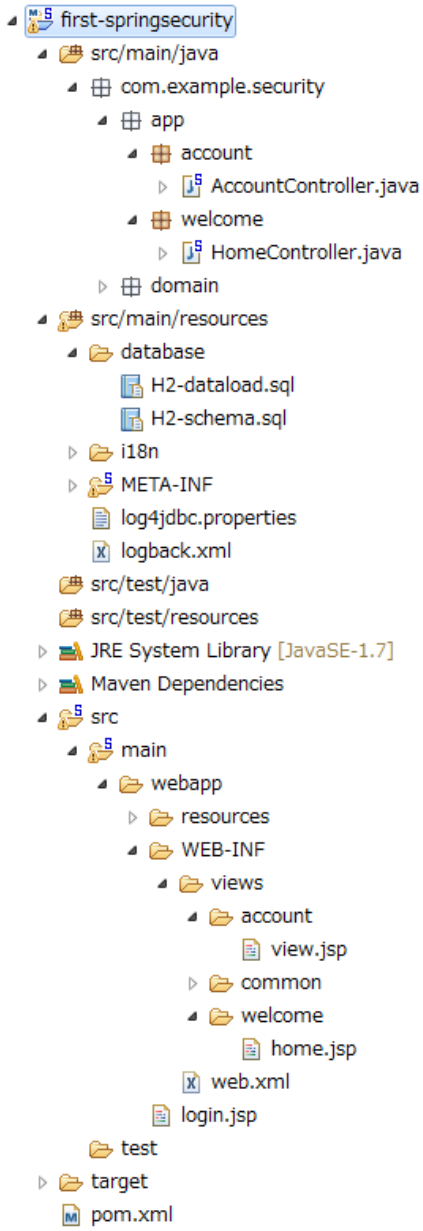


アプリケーション層の作成後のパッケージエクスプローラー

6.2.5 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

6.3 認証

6.3.1 Overview

本節では、Spring Security で提供している認証機能を説明する。

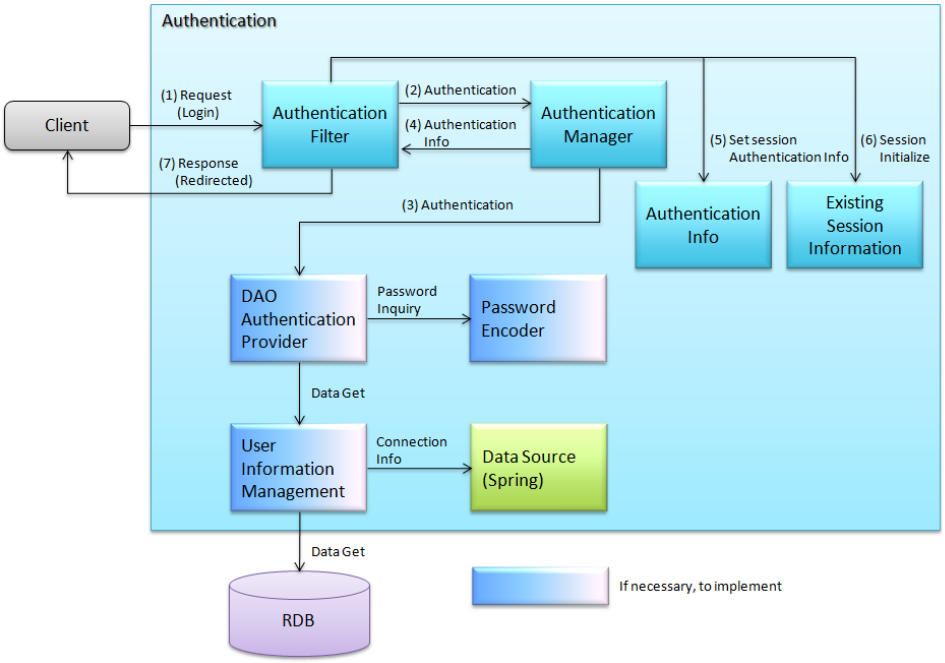
Spring Security では、設定ファイルの記述のみで、ユーザ認証を実装することができる。Spring Security で提供している認証方式として、DB 認証、LDAP 認証、CAS 認証、JAAS 認証、X509 認証、Basic 認証がサポートされているが、本ガイドラインでは、DB 認証についてのみ説明する。

ちなみに: DB 認証以外の詳細は、各認証方式の公式ドキュメントを参照されたい。

- LDAP Authentication
- CAS Authentication
- Java Authentication and Authorization Service (JAAS) Provider
- X.509 Authentication
- Basic and Digest Authentication

Login

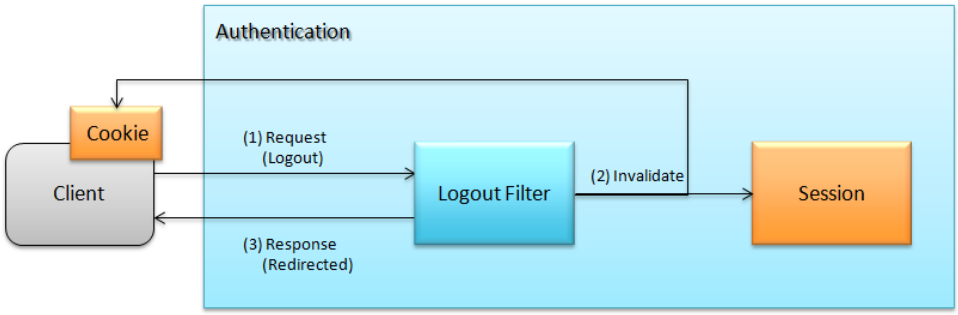
Spring Security によるログイン処理の流れを以下に示す。



1. 認証処理を指定したリクエストを受信すると、認証フィルタが起動する。
2. 認証フィルタは、リクエストからユーザ、パスワードを抽出し、認証情報を生成する。生成した認証情報をパラメータとし、認証マネージャの認証処理を実行する。
3. 認証マネージャは、指定された認証プロバイダの認証処理を実行する。認証プロバイダは、データソース（DB や LDAP）からユーザ情報を取得し、パスワード照合等のユーザ認証を行う。認証成功時には、認証済みの情報を保持する認証情報を作成し、認証マネージャに返す。認証失敗の場合は、認証失敗例外を送出する。
4. 認証マネージャは、受け取った認証情報を認証フィルタに返す。
5. 認証フィルタは、受け取った認証情報（認証済み）をセッションに格納する。
6. 認証成功時は、認証前のセッション情報を初期化し、新たにセッション情報を作成する。
7. 指定された認証成功/失敗時のパスへリダイレクトする。セッション ID をクライアントに返却する。

Logout

Spring Security によるログアウト処理の流れを以下に示す。



1. 指定されたログアウト処理へのリクエストを受信すると、ログアウトフィルタが起動する。
2. ログアウトフィルタはセッション情報を破棄する。また、クライアントのクッキー（図中の Cookie）を破棄するようなレスポンスを設定する。
3. 指定されたログアウト時のパスへ、リダイレクトする。

注釈: ログアウト後、残存するセッション情報が第三者に利用されることによるなりすましを防ぐため、セッション情報は、ログアウト時に `org.springframework.security.web.session.ConcurrentSessionFilter` で破棄される。

6.3.2 How to use

認証機能を使用するために、Spring Security の設定ファイルに記述する内容を以下に示す。

基本設定については、Spring Security 概要を参照されたい。

<sec:http>要素の設定

以下の設定例のように、spring-security.xml の<http>要素の auto-config 属性を true とすることで、Spring Security の認証機能の基本的な設定を、省略することができる。

```
<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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
    <sec:http auto-config="true" use-expressions="true">  <!-- (1) -->
      <!-- omitted -->
    </sec:http>
</beans>
```

項番	説明
(1)	auto-config="true"と設定することで、 <form-login>、<http-basic>、<logout>要素を設定しなくても有効になる。

注釈: <form-login>、<http-basic>、<logout>要素について説明する。

要素名	説明
<form-login>	org.springframework.security.web.authentication.UsernamePasswordAuthe が有効になる。 UsernamePasswordAuthenticationFilter は、ユーザ名、パスワードを POST 時に、リクエストから取り出し、認証を行う Filter である。 詳細は、*<sec:form-login>*要素の設定を参照されたい。
<http-basic>	org.springframework.security.web.authentication.www.BasicAuthenticati が有効になる。 BasicAuthenticationFilter は、Basic 認証の処理を実施する Filter であり、RFC1945 に準拠して実装されている。 詳細な利用方法は、BasicAuthenticationFilter JavaDoc を参照されたい。
<logout>	org.springframework.security.web.authentication.logout.LogoutFilter, org.springframework.security.web.authentication.logout.SecurityContex が有効になる。 LogoutFilter は、ログアウト時に呼ばれる Filter であり、 org.springframework.security.web.authentication.rememberme.TokenBased の削除) や、 SecurityContextLogoutHandler(セッションの無効化) を呼び出している。 詳細は、*<sec:logout>*要素の設定を参照されたい。

<sec:form-login>要素の設定

本節では、<sec:form-login>要素の設定方法を説明する。

form-login 要素の属性について、以下に示す。

spring-security.xml

```
<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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
  <sec:http auto-config="true" use-expressions="true">
    <sec:form-login login-page="/login"
        default-target-url="/"
        login-processing-url="/authentication"
        always-use-default-target="false"
        authentication-failure-url="/login?error=true"
        authentication-failure-handler-ref="authenticationFailureHandler"
        authentication-success-handler-ref="authenticationSuccessHandler" /> <!-- 属性の指定順番で (1)
  </sec:http>
</beans>
```

項番	説明
(1)	`login-page` 属性にログインフォーム画面のパスを指定する。 指定がない場合、「/spring_security_login」がデフォルトのパスになり、Spring Security が用意しているログイン画面が使用される。 「未認証ユーザ」が「認証ユーザ」しかアクセスできないページにアクセスした際に、本パスにリダイレクトされる。 **本ガイドラインでは、上記のデフォルト値「/spring_security_login」を使用せず、システム独自の値に変更することを推奨する。**この例では"/login" を指定している。
(2)	`default-target-url` 属性に認証成功時の遷移先パスを指定する。 指定がない場合、「/」が、デフォルトのパスになる。 `authentication-success-handler-ref` 属性の指定がある場合、本設定は使用されない。
(3)	`login-processing-url` 属性に認証処理を行うパスを指定する。 指定がない場合、「/j_spring_security_check」がデフォルトのパスになる。 **本ガイドラインでは、上記のデフォルト値「/j_spring_security_check」を使用せず、システム独自の値に変更することを推奨する。**この例では"/authentication" を指定している。
(4)	`always-use-default-target` 属性に、ログイン成功後に `default-target-url` に指定したパスに常に遷移するかどうかを設定する。 `true` が指定されている場合、`default-target-url` に指定したパスに常に遷移する。 `false`(デフォルト) が指定されている場合、「ログイン前にアクセスしようとした保護ページを表示するためのパス」又は「`default-target-url` に指定したパス」のいずれかに遷移する。 `authentication-success-handler-ref` 属性の指定がある場合、本設定は使用されない。
(5)	`authentication-failure-url` に認証失敗時の遷移先を設定する。 指定がない場合、`login-page` 属性に指定したパスが適用される。 `authentication-failure-handler-ref` 属性の指定がある場合、本設定は使用されない。

(6)	`authentication-failure-handler-ref` 属性に認証失敗時に呼ばれる、ハンドラクラスを指定する。 詳細は、**認証エラー時のハンドラクラスの設定**を参照されたい。

上記以外の属性については、Spring Security のマニュアルを参照されたい。

警告: Spring Security のデフォルト値「/spring_security_login, /j_spring_security_check」の使用を推奨しない理由

デフォルト値を使用している場合、そのアプリケーションが、Spring Security を使用していることについて、露見してしまう。そのため、Spring Security の脆弱性が発見された場合、脆弱性をついた攻撃を受けるリスクが高くなる。前述のリスクを避けるためにも、デフォルト値を使用しないことを推奨する。

ログインフォームの作成

認証時に使用するログインフォームを JSP で作成する。

- src/main/webapp/WEB-INF/views/login.jsp

```
<form:form action="${pageContext.request.contextPath}/authentication" method="post"><!-- (1)
    <!-- omitted -->
    <input type="text" id="username" name="j_username"><!-- (2) -->
    <input type="password" id="password" name="j_password"><!-- (5) -->
    <input type="submit" value="Login">
</form:form>
```

項番	説明
(1)	form の action 属性に認証処理を行うための遷移先を指定する。 遷移先のパスは login-processing-url 属性で指定した、/authentication を指定すること。 ${pageContext.request.contextPath}/authentication にアクセスすることで認証処理が実行される。 HTTP メソッドは、「POST」を指定すること。
(4)	認証処理において、「ユーザ ID」として扱われる要素。 name 属性には、Spring Security のデフォルト値である「j_username」を指定すること。
(5)	認証処理において、「パスワード」として扱われる要素。 name 属性には、Spring Security のデフォルト値である「j_password」を指定すること。

認証エラーメッセージを表示する場合は以下の追加する

```
<c:if test="${param.error}"><!-- (1) -->
    <t:messagesPanel
```

```
            messagesAttributeName="SPRING_SECURITY_LAST_EXCEPTION"/><!-- (2) -->
    </c:if>
```

項番	説明
(1)	リクエストパラメータに設定されたエラーメッセージの判定を行う。 form-login 要素の authentication-failure-url 属性に設定された値や、 認証エラーハンドラの”defaultFailureUrl” に設定された値によって、判定処理を変更する必要があるので注意すること。 本例では、authentication-failure-url=”/login?error=true” のような設定がある場合の、例を示している。
(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 `org.terasoluna.gfw.web.message.MessagesPanelTag` を指定して出力させることを推奨する。 「`<t:messagesPanel>`」タグの使用方法は、メッセージ管理を参照されたい。

注釈: 認証エラーの例外オブジェクトに JSP からアクセスする際に必要な設定について

認証エラーの例外オブジェクトは、セッションスコープに `"SPRING_SECURITY_LAST_EXCEPTION"` という属性名で格納されている。JSP からセッションスコープに格納されているオブジェクトにアクセスするためには、JSP の `page` ディレクティブの `session` 属性を `true` にする必要がある。

- `src/main/webapp/WEB-INF/views/common/include.jsp`

```
    <%@ page session="true"%>
```

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。

- spring-mvc.xml

  ログインフォームを表示する Controller を定義する。

```
    <mvc:view-controller path="/login" view-name="login" /><!-- (1) -->
```

項番	説明
(1)	“/login” にアクセスされたら、view 名として”login” を返却するだけの Controller を定義する。InternalResourceViewResolver によって src/main/webapp/WEB-INF/views/login.jsp が出力される。 この単純なコントローラは Java による実装が不要である。

ちなみに: 上記の設定は次の Controller と同義である。

```
    @Controller
    @RequestMapping("/login")
    public class LoginController {

        @RequestMapping
        public String index() {
            return "login";
        }
    }
```

単純に view 名を返すだけのメソッドが一つだけある Controller が必要であれば、<mvc:view-controller>を使用すればよい。

ログインフォームの属性名変更

「j_username」、「j_password」は、Spring Security のデフォルト値である。<form-login>要素の設定で、任意の値に変更することができる。

- spring-security.xml

  username、password の属性

```
    <sec:http auto-config="true" use-expressions="true">
      <sec:form-login
          username-parameter="username"
          password-parameter="password" /> <!-- 属性の指定順番で (1)～(2) -->
      <!-- omitted -->
    </sec:http>
```

項番	説明
(1)	username-parameter 属性で username の入力フィールドの name 属性を、「username」に変更している。
(2)	password-parameter 属性で password の入力フィールドの name 属性を、「password」に変更している。

認証処理の設定

Spring Security で認証処理を設定するために、AuthenticationProvider と UserDetailsService を定義する。

AuthenticationProvider は、次の役割を担う。

- 認証に成功した場合、認証ユーザー情報を返却する
- 認証に失敗した場合、例外をスローする

UserDetailsService は、認証ユーザー情報を永続化層から取得する役割を担う。

それぞれデフォルトで用意されているものを使用してもよいし、独自拡張して使用しても良い。組み合わせも自由である。

AuthenticationProvider クラスの設定

AuthenticationProvider の実装として、DB 認証を行うためのプロバイダ org.springframework.security.authentication.dao.DaoAuthenticationProvider を使用する方法を説明する。

- spring-security.xml

```
    <sec:authentication-manager><!-- (1) -->
        <sec:authentication-provider user-service-ref="userDetailsService"><!-- (2) -->
            <sec:password-encoder ref="passwordEncoder" /><!-- (3) -->
        </sec:authentication-provider>
    </sec:authentication-manager>
```

項番	説明
(1)	<sec:authentication-manager>要素内に <sec:authentication-provider>要素を定義する。複数指定して、認証方法を組み合わせることが可能であるが、ここでは説明しない。
(2)	<sec:authentication-provider>要素で AuthenticationProvider を定義する。デフォルトで、DaoAuthenticationProvider が有効になる。これ以外の AuthenticationProvider を指定する場合は ref 属性で、対象の AuthenticationProvider の Bean ID を指定する。 user-service-ref 属性に、認証ユーザ情報を取得する UserDetailsService の Bean Id を指定する。DaoAuthenticationProvider を使用する場合、この設定は必須である。 詳細は、*UserDetailsService* **クラスの設定**を参照されたい。
(3)	パスワード照合時に、フォームから入力されたパスワードのエンコードを行うクラスの Bean ID を指定する。 指定がない場合に、「平文」でパスワードが扱われる。詳細は、パスワードハッシュ化を参照されたい。

「ユーザー ID」と「パスワード」だけで永続化層からデータを取得し、認証するという要件であればこの DaoAuthenticationProvider を使用すれば良い。
永続化層からのデータ取得方法は次に説明する UserDetailsService で決める。

UserDetailsService クラスの設定

AuthenticationProvider の userDetailsService プロパティに指定した Bean を設定する。

UserDetailsService は次のメソッドをもつインタフェースである。

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
```

このインタフェースを実装すれば、任意の保存場所から認証ユーザー情報を取得することができる。

ここでは、JDBC を使用して、DB からユーザ情報を取得する org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl を説明する。

JdbcDaoImpl を使用するには spring-security.xml に以下の Bean 定義を行えば良い。

```
<!-- omitted -->

<bean id="userDetailsService"
  class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

JdbcDaoImpl は、認証ユーザー情報と認可情報を取得するためのデフォルト SQL を定義しており、これらに対応したテーブルが用意されていることが前提となっている。前提としているテーブル定義は Spring Security のマニュアルを参照されたい。

既存のテーブルからユーザー情報、認可情報を取得したい場合は、発行される SQL を既存のテーブルに合わせて修正すればよい。

使用する SQL は以下の 3 つである。

- ユーザ情報取得クエリ

  ユーザ情報取得クエリに合致するテーブルを作成することで、後述する設定ファイルへのクエリ指定が不要となる。

  「username」、「password」、「enabled」フィールドは必須であるが、

  後述する設定ファイルへのクエリ指定で、別名を付与することにより、テーブル名、カラム名が一致しなくても問題ない。

  例えば次のような SQL を設定すれば「email」カラムを「username」として使用することができ、「enabled」は常に true となる。

```
    SELECT email AS username, pwd AS password, true AS enabled FROM customer WHERE email = ?
```

  ログインフォームの作成で前述した、「ユーザ ID」がクエリのパラメータに指定される。

- ユーザ権限取得クエリ

  ユーザに対する認可情報を取得するクエリである。

- グループ権限取得クエリ

ユーザーが所属するグループの認可情報を取得するクエリである。グループ権限はデフォルトでは無効になっており、本ガイドラインでも扱わない。

以下に、DB の定義例、Spring Security の設定ファイル例を示す。

テーブルの定義について

DB 認証処理を実装するにあたり、必要となるテーブルを定義する。

前述した、デフォルトのユーザ情報取得クエリ合致するテーブルとなっている。

そのため、下記が最低限必要となるテーブルの定義となる（物理名は仮称）。

テーブル名: account

論理名	物理名	型	説明
ユーザ ID	username	文字列	ユーザを一意に識別するためのユーザ ID。
パスワード	password	文字列	ユーザパスワード。ハッシュ化された状態で格納する。
有効フラグ	enabled	真偽値	無効ユーザ、有効ユーザを表すフラグ。「false」に設定されたユーザは無効ユーザとして、認証エラーとなる。
権限名	authority	文字列	認可機能を必要としない場合は不要。

`JdbcDaoImpl` をカスタマイズして設定する例を以下に示す。

```
<!-- omitted -->

<bean id="userDetailsService"
  class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
  <property name="rolePrefix" value="ROLE_" /><!-- (1) -->
  <property name="dataSource" ref="dataSource" />
  <property name="enableGroups" value="false" /><!-- (2) -->
  <property name="usersByUsernameQuery"
    value="SELECT username, password, enabled FROM account WHERE username = ?" /><!-- (3) -->
  <property name="authoritiesByUsernameQuery"
    value="SELECT username, authority FROM account WHERE username = ?" /><!-- (4) -->
</bean>
```

項番	説明
(1)	権限名の prefix を指定する。DB 上に格納されている権限名が”USER” の場合、この認証ユーザーオブジェクトが持つ権限名は”ROLE_USER” になる。 認可機能と命名規則を合わせて設定する必要がある。認可機能の詳細は、認可を参照されたい。
(2)	認可機能において、「グループ権限」の概念を用いる場合に指定する。 本ガイドラインでは扱わない。
(3)	ユーザ情報を取得するクエリを設定する。取得するデータは、「ユーザ ID」、「パスワード」、「有効フラグ」の順とする。 「有効フラグ」による認証判定を行わない場合には、「有効フラグ」の SELECT 結果を「true」固定とする。 なお、ユーザを一意に取得できるクエリを記述すること。複数件数取得された場合には、１件目のレコードがユーザとして使われる。
(4)	ユーザの権限を取得するクエリを設定する。取得するデータは、「ユーザ ID」、「権限 ID」の順とする。 認可の機能を使用しない場合は、「権限 ID」は任意の固定値でよい。

注釈: クエリを変更するだけでは実現できない認証を行う場合、`UserDetailsService` を拡張して実現する必要がある。拡張方法については、*UserDetailsService* **の拡張**を参照されたい。

`UserDetails` クラスの利用方法

認証に成功した後に `UserDetailsService` が作成した `UserDetails` の利用方法について、説明する。

Java クラスで `UserDetails` オブジェクトを利用する

認証に成功した後、`UserDetails` クラスは
`org.springframework.security.core.context.SecurityContextHolder` に格納される。

SecurityContextHolder から UserDetails を取得する例を示す。

```
public static String getUsername() {
    Authentication authentication = SecurityContextHolder.getContext()
            .getAuthentication(); // (1)
    if (authentication != null) {
        Object principal = authentication.getPrincipal(); // (2)
        if (principal instanceof UserDetails) {
            return ((UserDetails) principal).getUsername(); // (3)
        }
        return (String) principal.toString();
    }
    return null;
}
```

項番	説明
(1)	SecurityContextHolder から org.springframework.security.core.Authentication オブジェクトを取得する。
(2)	Authentication オブジェクトから UserDetails オブジェクトを取得する。
(3)	UserDetails オブジェクトから、ユーザ名を取得する。

SecurityContextHolder から UserDetails オブジェクトを取得する方法は、どこからでも static メソッドで利用可能であり、便利な反面、モジュール結合度を高めてしまう。テストも実施しづらい。

UserDetails オブジェクトは java.security.Principal オブジェクトからも取得可能であるため、Spring MVC の Controller 内では以下のように SecurityContextHolder を使用せずに UserDetails オブジェクトを取得できる。

```
@RequestMapping(method = RequestMethod.GET)
public String view(Principal principal, Model model) {
    // get Authentication
    Authentication authentication = (Authentication) principal; // (1)
    // get UserDetails
    UserDetails userDetails = (UserDetails) authentication.getPrincipal(); // (2)
    // omitted ...
}
```

項番	説明
(1)	`java.security.Principal` オブジェクトを `org.springframework.security.core.Authentication` クラスにキャストする。
(2)	`Authentication` オブジェクトから `UserDetails` オブジェクトを取得する。

Controller 内で UserDetails オブジェクトにアクセスする場合はこちらの方法を推奨する。

注釈: Service クラスでは Controller が取得した `UserDetails` オブジェクトの情報を使用し、`SecurityContextHolder` は使用しないことを推奨する。

`SecurityContextHolder` は `java.security.Principal` オブジェクトにアクセスできないメソッド内でのみ利用することが望ましい。

JSP で `UserDetails` にアクセスする

Spring Security では、JSP で認証情報を利用するための仕組みとして、JSP taglib を提供している。この taglib を使うために以下の宣言が必要である。

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec"%>
```

注釈: TERASOLUNA Global Framework の雛形を使用している場合は WEB-INF/views/common/include.jsp に設定済みである。

認証ユーザ名を JSP で表示する場合を例に、使用方法を示す。

```
<sec:authentication property="principal.username" /><!-- (1) -->
```

項番	説明
(1)	<sec:authentication> タグで Authentication オブジェクトにアクセスでき、property 属性に指定したプロパティアクセスできる。この例では getPrincipal().getUsername() の結果を出力する。

```
<sec:authentication property="principal" var="userDetails" /> <!-- (1) -->

${f:h(userDetails.username)} <!-- (2) -->
```

項番	説明
(1)	property 属性に指定したプロパティを var 属性にした名前で変数に格納できる。
(2)	(1) で変数に格納した後は JSP 内で UserDetails にアクセスできる。

注釈: Controller 内で UserDetails を取得して Model に追加することもできるが、JSP に表示する際は JSP タグを使用すればよい。

注釈: *UserDetailsService* **クラスの設定**で説明した JdbcDaoImpl が生成する UserDetails は「ユーザーID」や「権限」といった最低限の情報しか保持していない。画面の表示項目として「ユーザー姓名」など他のユーザー情報が必要な場合は UserDetails と UserDetailsService を拡張する必要がある。拡張方法については、*UserDetailsService* **の拡張**を参照されたい。

Spring Security におけるセッション管理

ログイン時のセッション情報の生成方式や、例外発生時の設定を行う方法について説明する。

<session-management>タグを指定することで、セッションの管理方法をカスタマイズする事ができる。

以下に spring-security.xml の設定例を示す。

```
<sec:http auto-config="true" create-session="ifRequired" ><!-- (1) -->
  <!-- omitted -->
  <sec:session-management
    invalid-session-url="/"
    session-authentication-error-url="/"
    session-fixation-protection="migrateSession"
    session-authentication-strategy-ref="sessionStrategy" /><!-- 属性の指定順番で (2)～(5) -->
  <!-- omitted -->
</sec:http>
```

項番	説明
(1)	<http>タグの create-session 属性には、セッションの作成方針を指定する。 以下の値を指定することができる。 • always: Spring Security は、既存のセッションがない場合にセッションを新規作成する、セッションが存在すれば、再利用する。 • ifRequired: (デフォルト) Spring Security は、セッションが必要であれば作成する。セッションがすでにあれば、作成せずに再利用する。 • never: Spring Security は、セッションを作成しないが、セッションが存在すれば、再利用する。 • stateless: Spring Security は、セッションを作成しない、セッションが存在しても使用しない。そのため、毎回認証を行う必要がある。
(2)	invalid-session-url 属性には、無効なセッション ID がリクエストされた場合に遷移するパスを指定する。 指定しない場合、セッションの存在チェックは実行されずに後続処理が呼び出される。
(3)	session-authentication-error-url 属性には、org.springframework.security.web.authentication.session.SessionAuthentication で例外が発生した場合に遷移するパスを指定する。 指定しない場合、レスポンスコードに「401 Unauthorized」が設定され、エラー応答が行われる。 本設定は、<form-login>タグを使用して認証を行う場合は使用されない。 SessionAuthenticationStrategy で発生した例外は、<form-login>タグの authentication-failure-url 属性 又は authentication-failure-handler-ref 属性の定義に応じてハンドリングされる。
(4)	session-fixation-protection 属性には、認証成功時のセッション管理方式を指定する。 以下の値を指定することができる。 • none： ログイン前のセッションをそのまま利用する。 • migrateSession：(デフォルト) ログイン前のセッションを破棄して新しいセッションを新たに作成し、ログイン前のセッションに格納していた情報を引き継ぐ。 • newSession： ログイン前のセッションを破棄して新しいセッションを新たに作成し、ログイン前のセッ

SessionAuthenticationStrategy の設定

SessionAuthenticationStrategy の設定例を以下に示す。

```
<bean id="sessionStrategy"
    class="org.springframework.security.web.authentication.session.CompositeSessionAuthenticationS
    <constructor-arg index="0">
        <list>
            <bean
                class="org.springframework.security.web.authentication.session.SessionFixationProt
            <bean class="org.springframework.security.web.csrf.CsrfAuthenticationStrategy">  <!--
                <constructor-arg index="0" ref="csrfTokenRepository" />
            </bean>
        </list>
    </constructor-arg>
</bean>
<!-- omitted -->
```

項番	説明
(1)	org.springframework.security.web.authentication.session.CompositeSessionAuthe のコンストラクタの引数に、list で指定する。
(2)	本例では、CSRF チェックを行う org.springframework.security.web.csrf.CsrfAuthenticationStrategy を指定している。CSRF 対策については、CSRF 対策を参照されたい。

同時セッション数の制御

Spring Security では、1 ユーザが同時にログインできるセッション数を制御する機能 (Concurrent Session Control) を提供している。

ここでいうユーザとは、Authentication.getPrincipal() で取得される、認証ユーザーオブジェクトのことである。

注釈: この機能はアプリケーションサーバが 1 台構成、またはセッションサーバやクラスタによるセッションレプリケーションを実施している（つまり、全てのアプリケーションが同じセッション領域を利用している）場合に有効である。複数台または複数インスタンスで構成していて、セッション領域が別々に存在する場合は、本機能では同時ログインを制御できないので注意すること。

最大セッション数を超えた場合の制御方法は、次のパターンが存在する。業務要件によって使い分けること。

1. 1 ユーザの最大セッション数を超過した場合、最も使用されていないユーザを無効にする (後勝ち)
2. 1 ユーザの最大セッション数を超過した場合、新規ログインを受け付けない (先勝ち)

どちらの場合も、この機能を有効にするためには web.xml に以下の設定を追加する必要がある。

```
<listener>
  <listener-class>org.springframework.security.web.session.HttpSessionEventPublisher</listener-cla
</listener>
```

項番	説明
(1)	Concurrent Session Control を使用するに当たり、`org.springframework.security.web.session.HttpSessionEventPublisher` を、listener に定義する必要がある。

1. 最も使用されていないユーザを無効にする場合

   spring-security.xml に以下の設定を追加する。

```
    <sec:http auto-config="true" >
      <!-- omitted -->
      <sec:custom-filter position="CONCURRENT_SESSION_FILTER" ref="concurrencyFilter" />  <!-- (

      <sec:session-management
        session-authentication-strategy-ref="sessionStrategy" />
      <!-- omitted -->
    </sec:http>

    <bean id="concurrencyFilter"
       class="org.springframework.security.web.session.ConcurrentSessionFilter">  <!-- (2) -->
       <constructor-arg index="0" ref="sessionRegistry" />  <!-- (3) -->
       <constructor-arg index="1" value="/" />  <!-- (4) -->
    </bean>

    <bean id="sessionStrategy"
        class="org.springframework.security.web.authentication.session.CompositeSessionAuthentic
        <constructor-arg index="0">
            <list>
                <!-- omitted -->
                <bean class=
                    "org.springframework.security.web.authentication.session.ConcurrentSessionCo
                    <constructor-arg index="0" ref="sessionRegistry" />  <!-- (6) -->
```

```
                    <property name="maximumSessions" value="1" />  <!-- (7) -->
                </bean>
            </list>
        </constructor-arg>
    </bean>

    <bean id="sessionRegistry" class="org.springframework.security.core.session.SessionRegistryI
    <!-- omitted -->
```

項番	説明
(1)	<custom-filter>要素の、position 属性に CONCURRENT_SESSION_FILTER を指定する。
(2)	org.springframework.security.web.session.ConcurrentSessionFilter クラスを Bean 定義する。
(3)	コンストラクタの第 1 引数に、org.springframework.security.core.session.SessionRegistryImpl を参照指定する。
(4)	コンストラクタの第 2 引数に、期限切れになったセッションが遷移するパスを指定する。
(5)	CompositeSessionAuthenticationStrategy の第 1 引数に、 org.springframework.security.web.authentication.session.ConcurrentSession を指定する。
(6)	コンストラクタの第 1 引数に、org.springframework.security.core.session.SessionRegistryImpl を参照指定する。
(7)	maximumSessions 属性に、1 ユーザが許容する最大セッション数を定義することができる。 上記例では、1 を指定しているため、1 ユーザが許容するセッションは 1 つになる。 ユーザーが複数のブラウザでログインした場合、使用した日時が最も古いセッションに対して期限切れにする。 指定しない場合、1 が設定される。
(8)	org.springframework.security.core.session.SessionRegistry インタフェースを実装したクラスである、org.springframework.security.core.session.SessionRegistryImpl を指定する。

2. 新規ログインを受け付けない

   spring-security.xml に以下の設定を行う。

```
<bean id="concurrencyFilter"
   class="org.springframework.security.web.session.ConcurrentSessionFilter">
   <constructor-arg index="0" ref="sessionRegistry" />
   <constructor-arg index="1" value="/" />
</bean>

<bean id="sessionStrategy"
    class="org.springframework.security.web.authentication.session.CompositeSessionAuthentic
    <constructor-arg index="0">
        <list>
            <!-- omitted -->
            <bean class=
                "org.springframework.security.web.authentication.session.ConcurrentSessionCo
                <constructor-arg index="0" ref="sessionRegistry" />
                <property name="maximumSessions" value="1" />
                <property name="exceptionIfMaximumExceeded" value="true"/> <!-- (1) -->
            </bean>
        </list>
    </constructor-arg>
</bean>
<!-- omitted -->
```

項番	説明
(1)	exceptionIfMaximumExceeded 属性を true に設定することにより、最大セッション数を超過した場合、 org.springframework.security.web.authentication.session.SessionAuthentica がスローされる。 そのため、ConcurrentSessionFilter の第 2 引数で定義したパスには遷移しないので注意すること。 exceptionIfMaximumExceeded 属性の設定を省略した場合は、false が設定される。

ちなみに: <sec:session-management>要素の session-authentication-strategy-ref 属性を指定せず、<sec:session-management>要素の子要素として<sec:concurrency-control>要素を使用することもできる。

認証エラー時のハンドラクラスの設定

<sec:form-login>要素の authentication-failure-handler-ref 属性に

org.springframework.security.web.authentication.ExceptionMappingAuthenticationFailureHan
クラスの設定をし、
認証エラー時に送出される例外と、それに対応した遷移先を指定できる。
指定する遷移先は、未認証ユーザがアクセス可能であること。

spring-security.xml

```
<sec:http auto-config="true" use-expressions="true">
  <sec:form-login login-page="/login"
      authentication-failure-handler-ref="authenticationFailureHandler"
      authentication-success-handler-ref="authenticationSuccessHandler" />
</sec:http>

<bean id="authenticationFailureHandler"
class="org.springframework.security.web.authentication.ExceptionMappingAuthenticationFailureHandle
<property name="defaultFailureUrl" value="/login/defaultError" /><!-- (1) -->
  <property name="exceptionMappings"><!-- (2) -->
    <props>
      <prop key=
        "org.springframework.security.authentication.BadCredentialsException"><!-- (3) -->
          /login/badCredentials
      </prop>
      <prop key=
        "org.springframework.security.core.userdetails.UsernameNotFoundException"><!-- (4) -->
          /login/usernameNotFound
      </prop>
      <prop key=
        "org.springframework.security.authentication.DisabledException"><!-- (5) -->
          /login/disabled
      </prop>
      <prop key=
        "org.springframework.security.authentication.ProviderNotFoundException"><!-- (6) -->
          /login/providerNotFound
      </prop>
      <prop key=
        "org.springframework.security.authentication.AuthenticationServiceException"><!-- (7) -->
          /login/authenticationService
      </prop>
      <!-- omitted -->
    </props>
  </property>
</bean>
```

項番	説明
(1)	エラー時のデフォルトの遷移先パスを指定する。 後述する `exceptionMappings` プロパティに定義していない例外が発生した場合、本プロパティで指定した遷移先に遷移する。
(2)	catch する例外と、例外発生時の遷移先を、リスト形式で指定する。 key に例外クラスを指定し、値に遷移先を設定する。

Spring Security がスローする代表的な例外を、以下に記述する。

項番	エラーの種類	説明
(3)	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。
(4)	UsernameNotFoundException	不正ユーザ ID（存在しないユーザ 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 を false に変更しないと上記例外は、BadCredentialsException に変更される。
(5)	DisabledException	無効ユーザ ID による認証エラー時に、スローされる。
(6)	ProviderNotFoundException	認証プロバイダクラス未検出エラー時にスローされる。 設定誤り等の理由から、認証プロバイダクラスが不正な場合に発生する。
(7)	AuthenticationServiceException	認証サービスエラー時にスローされる。 DB 接続エラー等、認証サービス内で何らかのエラーが発生した際に発生する。

警告: 本例では、UsernameNotFoundException をハンドリングして遷移させているが、ユーザ ID が存在しないことを利用者に知らせると、特定の ID の存在有無が判明するため、セキュリティの観点上望ましくない。そのため、ユーザに通知するメッセージには、例外の種類によって区別をしない画面遷移、メッセージにした方がよい。

<sec:logout>要素の設定

本節では、<sec:logout>要素の設定方法を説明する。

spring-security.xml

```
<sec:http auto-config="true" use-expressions="true">
  <!-- omitted -->
  <sec:logout
      logout-url="/logout"
      logout-success-url="/"
      invalidate-session="true"
      delete-cookies="JSESSIONID"
      success-handler-ref="logoutSuccessHandler"
    /> <!-- 属性の指定順番で (1)～(5) -->
  <!-- omitted -->
</sec:http>
```

項番	説明
(1)	logout-url 属性に、ログアウト処理を実行するためのパスを指定する。 指定がない場合、「/j_spring_security_logout」がデフォルトのパスになる。 **本ガイドラインでは、上記のデフォルト値「/j_spring_security_logout」を使用せず、システム独自の値に変更することを推奨する。** この例では ” /logout ” を指定している。
(2)	logout-success-url 属性に、ログアウト後の遷移先パスを指定する。 指定がない場合、「/」がデフォルトのパスになる。 本属性を指定した場合、success-handler-ref 属性を指定すると起動時にエラーとなる。
(3)	invalidate-session 属性に、ログアウト時にセッションを破棄するかを設定する。 デフォルトは true である。 true の場合、ログアウト時にセッションが破棄される。
(4)	delete-cookies 属性に、ログアウト時に削除するクッキー名を列挙する。 複数記述する場合は「,」で区切る。
(5)	success-handler-ref 属性に、ログアウト成功後に呼び出すハンドラクラスを指定する。 本属性を指定した場合、logout-success-url 属性を指定すると起動時にエラーとなる。

警告: Spring Security のデフォルト値「/j_spring_security_logout」の使用を推奨しない理由
デフォルト値を使用している場合、そのアプリケーションが、Spring Security を使用していることについて、露見してしまう。そのため、Spring Security の脆弱性が発見された場合、脆弱性をついた攻撃を受けるリスクが高くなる。前述のリスクを避けるためにも、デフォルト値を使用しないことを推奨する。

<sec:remember-me>要素の設定

「Remeber Me」とは、website に頻繁にアクセスするユーザの利便性を、高めるための機能の一つとして、ログイン状態を保持する機能である。

本機能は、ユーザがログイン状態を保持することを許可していた場合、ブラウザを閉じた後も cookie にログイン情報を保持し、ユーザ名、パスワードを再入力しなくともログインすることができる機能である。

<sec:remember-me>要素の属性について、以下に示す。

spring-security.xml

```
<sec:http auto-config="true" use-expressions="true">
  <!-- omitted -->
  <sec:remember-me key="terasoluna-tourreservation-km/ylnHv"
          token-validity-seconds="#{30 * 24 * 60 * 60}" />  <!-- 属性の指定順番で (1)～(2) -->
  <!-- omitted -->
</sec:http>
```

項番	説明
(1)	key 属性に、Remeber Me 用の cookie を保持しておくためのユニークなキーを指定する。 指定が無い場合、ユニークなキーを起動時に生成するため、起動時間向上を考えた場合指定しておくことを推奨する。
(2)	「token-validity-seconds 属性に、Remeber Me 用の cookie の有効時間を秒単位で指定する。この例では 30 日間を設定している。 指定が無い場合、デフォルトで 14 日間が有効期限になる。

上記以外の属性については、Spring Security のマニュアルを参照されたい。

ログインフォームには以下のように「Remeber Me」機能を有効にするためのフラグを用意する必要がある。

```
<form method="post"
  action="${pageContext.request.contextPath}/authentication">
    <!-- omitted -->
    <label for="_spring_security_remember_me">Remember Me : </label>
    <input name="_spring_security_remember_me"
      id="_spring_security_remember_me" type="checkbox"
      checked="checked"> <!-- (1) -->
    <input type="submit" value="LOGIN">
    <!-- omitted -->
</form>
```

項番	説明
(1)	HTTP パラメータに、_spring_security_remember_me を設定することで、 true でリクエストされた場合、次回の認証を回避することができる。

6.3.3 How to extend

UserDetailsService の拡張

認証時にユーザ ID、パスワード以外の情報も取得したい場合、

- org.springframework.security.core.userdetails.UserDetails
- org.springframework.security.core.userdetails.userDetailsService

を実装する必要がある。

ログインユーザーの氏名や所属部署などの付属情報を常に画面のヘッダーに表示させる必要がある場合、毎リクエストで DB から取得するのは非効率的である。UserDetails オブジェクトに保持させて、SecurityContext や<sec:authentication>タグからアクセスできようにするにはこの拡張が必要である。

UserDetails の拡張

認証情報以外に顧客情報も保持する ReservationUserDetails クラスを作成する。

```
public class ReservationUserDetails extends User { // (1)
    // omitted

    private final Customer customer; // (2)

    private static final List<? extends GrantedAuthority> DEFAULT_AUTHORITIES = Collections
            .singletonList(new SimpleGrantedAuthority("ROLE_USER"));          // (3)

    public ReservationUserDetails(Customer customer) {
        super(customer.getCustomerCode(),
                customer.getCustomerPassword(), true, true, true, true, DEFAULT_AUTHORITIES); //
        this.customer = customer;
    }

    public Customer getCustomer() { // (5)
        return customer;
    }
}
```

項番	説明
(1)	`UserDetails` のデフォルトクラスである、`org.springframework.security.core.userdetails.User` クラスを継承する。
(2)	認証情報および顧客情報をもつ DomainObject クラスを保持する。
(3)	認可情報を、`org.springframework.security.core.authority.SimpleGrantedAuthority` のコンストラクタで作成する。ここでは”ROLE_USER” という権限を与える。 本実装は簡易実装であり、本来は認可情報は DB 上の別のテーブルから取得すべきである。
(4)	スーパークラスのコンストラクタに、DomainObject が持つユーザ ID、パスワードを設定する。
(5)	`UserDetails` 経由で顧客情報にアクセスするためのメソッド。

注釈: `User` クラスを継承するだけでは、業務要件を実現できない場合、`UserDetails` インタフェースを実装すればよい。

独自 `UserDetailsService` の実装

`UserDetailsService` を実装した ReservationUserDetailsService クラスを作成する。

本例では、`Customer` オブジェクトを取得する処理を実装した `CustomerSharedService` クラスをインジェクションして、DB から顧客情報を取得している。

```
public class ReservationUserDetailsService implements UserDetailsService {
    @Inject
    CustomerSharedService customerSharedService;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        Customer customer = customerSharedService.findOne(username);
```

```
        // omitted
        return new ReservationUserDetails(customer);
    }

}
```

使用方法

作成した ReservationUserDetailsService、ReservationUserDetails の使用方法を説明する。

- spring-security.xml

```
    <sec:authentication-manager>
        <sec:authentication-provider user-service-ref="userDetailsService"><!-- (1) -->
            <sec:password-encoder ref="passwordEncoder" />
        </sec:authentication-provider>
    </sec:authentication-manager>

    <bean id="userDetailsService"
        class="com.example.domain.service.userdetails.ReservationUserDetailsService"><!-- (2) --
    </bean>
    <!-- omitted -->
```

項番	説明
(1)	ReservationUserDetailsService の Bean ID を ref 属性に定義する。
(2)	ReservationUserDetailsService を Bean 定義する。

- JSP

  <sec:authentication>タグを使用して Customer オブジェクトにアクセスする。

```
    <sec:authentication property="principal.customer" var="customer"/><!-- (1) -->
    ${f:h(customer.customerName)}<!-- (1) -->
```

項番	説明
(1)	ReservationUserDetails がもつ Customer オブジェクトを変数に格納する。
(2)	変数に格納した Customer オブジェクトの任意のプロパティを表示する。 f:h() については、XSS 対策を参照されたい。

- Controller

```
@RequestMapping(method = RequestMethod.GET)
public String view(Principal principal, Model model) {
    // get Authentication
    Authentication authentication = (Authentication) principal;
    // get UserDetails
    ReservationUserDetails userDetails = (ReservationUserDetails) authentication.getPrincipa
    // get Customer
    Customer customer = userDetails.getCustomer(); // (1)
    // omitted ...
}
```

項番	説明
(1)	ReservationUserDetails から、ログイン中の Customer オブジェクトを取得する。 このオブジェクトを Service クラスに渡して業務処理を行う。

注釈: 顧客情報が変更された場合、一度ログアウトしないと ReservationUserDetails がもつ Customer オブジェクトは変更されない。

頻繁に変更されうる情報や、ログインユーザー以外のユーザー (管理者など) によって変更される情報は保持しない方がよい。

AuthenticationProvider 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業務要件がある場合、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)	ユーザ名、パスワード、会社識別子 (独自の認証パラメータ) 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.UsernamePasswordAuthentica クラスを継承して作成する。
(2)	ユーザ名、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.DaoAuthenticationProvi クラスを継承して作成する。
(3)	ユーザ名、パスワード、会社識別子 (独自の認証パラメータ) をリクエストパラメータから取得して、AuthenticationManager(AuthenticationProvider) に渡す Authentication を生成するためのサーブレットフィルタクラス。 ここでは、org.springframework.security.web.authentication.UsernamePasswordAuthen クラスを継承して作成する。

ちなみに: 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 を生成するためのサーブレットフィルタクラスの拡張が必要となる。

ユーザ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。

UsernamePasswordAuthenticationToken の拡張

ここでは、UsernamePasswordAuthenticationToken クラスを継承し、ユーザ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。

DaoAuthenticationProvide の拡張

ここでは、DaoAuthenticationProvider クラスを継承し、ユーザ名、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。 このタイミングでパスワード認証が行われる。
(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。

ちなみに: ユーザの存在チェック、ユーザの状態チェック (無効ユーザ、ロック中ユーザ、利用期限切れユーザなどのチェック) 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。

UsernamePasswordAuthenticationFilter の拡張

ここでは、UsernamePasswordAuthenticationFilter クラスを継承し、認証情報 (ユーザ名、パスワード、会社識別子) を AuthenticationProvider に引き渡すためのサーブレットフィルタクラスを作成する。

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.Authenticatio 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仕組みになっている。
(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。

注釈: 認証情報の入力チェックについて

DB サーバへの負荷軽減等で、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。その場合は、*UsernamePasswordAuthenticationFilter の拡張*のように、`UsernamePasswordAuthenticationFilter` を拡張することで、入力チェック処理を行うことができる。

なお、上記例では入力チェックは行っていない。

課題

認証情報の入力チェックは、Controller クラスでリクエストをハンドリングして、Bean Validation を使用して行う事も可能である。

Bean Validation を使用した入力チェックの方法については、今後追加する予定である。

拡張した認証処理の適用

ユーザ名、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`

```
<!-- omitted -->

<!-- (1) -->
<sec:http
    auto-config="false"
    use-expressions="true"
    entry-point-ref="loginUrlAuthenticationEntryPoint">

    <!-- omitted -->

    <!-- (2) -->
    <sec:custom-filter
        position="FORM_LOGIN_FILTER"
        ref="companyIdUsernamePasswordAuthenticationFilter" />

    <!-- omitted -->
```

```
    <sec:custom-filter before="LOGOUT_FILTER" ref="csrfFilter" />

    <sec:logout
        logout-url="/logout"
        logout-success-url="/login"
        delete-cookies="JSESSIONID" />

    <!-- 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="filterProcessesUrl" value="/authentication" />
    <!-- (6) -->
    <property name="authenticationManager" ref="authenticationManager" />
    <!-- (7) -->
    <property name="sessionAuthenticationStrategy" ref="sessionAuthenticationStrategy" />
    <!-- (8) -->
    <property name="authenticationFailureHandler">
        <bean class="org.springframework.security.web.authentication.SimpleUrlAuthenticationFailu
            <constructor-arg value="/login?error=true" />
        </bean>
    </property>
    <!-- (9) -->
    <property name="authenticationSuccessHandler">
        <bean class="org.springframework.security.web.authentication.SimpleUrlAuthenticationSucce
    </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.CompositeSessionAuthenticationS
    <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.SessionFixationPr
        </util:list>
    </constructor-arg>
</bean>

<bean id="csrfTokenRepository"
    class="org.springframework.security.web.csrf.HttpSessionCsrfTokenRepository" />

<bean id="csrfFilter" class="org.springframework.security.web.csrf.CsrfFilter">
    <constructor-arg ref="csrfTokenRepository" />
</bean>


<!-- omitted -->
```

項番	説明
(1)	custom-filter 要素を使用して”FORM_LOGIN_FILTER” を差し替える場合は、http 要素の属性に、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。 • form-login 要素を使用できないため、entry-point-ref 属性を使用して、使用する AuthenticationEntryPoint を明示的に指定する。
(2)	custom-filter 要素を使用して”FORM_LOGIN_FILTER” を差し替える。 custom-filter 要素の position 属性に"FORM_LOGIN_FILTER"を指定し、ref 属性に拡張したサーブレットフィルタの bean ID を指定する。
(3)	http 要素の entry-point-ref 属性に指定する AuthenticationEntryPoint の bean を定義する。 ここでは、form-login 要素を指定した際の使用される org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPo クラスの bean を定義している。
(4)	“FORM_LOGIN_FILTER” として使用するサーブレットフィルタの bean を定義する。 ここでは、拡張したサーブレットフィルタクラス (CompanyIdUsernamePasswordAuthenticationFilter) の bean を定義している。
(5)	filterProcessesUrl プロパティに、認証処理用のパスを指定する。 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、form-login 要素の 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、authentication-manager 要素の alias 属性に設定した値を指定する。 authentication-manager 要素の 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。
(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。

注釈: auto-config="false"を指定した場合、<sec:http-basic>要素と<sec:logout>要素は、明示的に定義しないと有効にならない。

ログインフォームの作成

ここでは、**ログインフォームの作成**で紹介した画面 (JSP) に対して、会社識別子を追加する。

```
<form:form action="${pageContext.request.contextPath}/authentication" method="post">
    <!-- omitted -->
    <span>User Id</span><br>
    <input type="text" id="username" name="j_username"><br>
    <span>Company Id</span><br>
    <input type="text" id="companyid" name="companyid"><br>  <!-- (1) -->
    <span>Password</span><br>
    <input type="password" id="password" name="j_password"><br>
    <!-- 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。

6.3.4 Appendix

遷移先の指定が可能な認証成功ハンドラ

Spring Security を使用した認証では、認証に成功した場合は、

- bean 定義ファイル (spring-security.xml) に記述したパス (<form-login>要素の default-target-url 属性に指定したパス)
- ログイン前にアクセスした「認証が必要な保護ページ」を表示するためのパス

に遷移する。

共通ライブラリでは、Spring Security が提供している機能に加えて、遷移先のパスをリクエストパラメータで指定できるクラス (`org.terasoluna.gfw.web.security.RedirectAuthenticationHandler`) を提供している。

`RedirectAuthenticationHandler` は、以下のような仕組みを実現するために作成されたクラスである。

- ページを表示するためにログインを行う必要がある
- ログイン後の遷移先のページを JSP 側 (遷移元の JSP) で指定したい

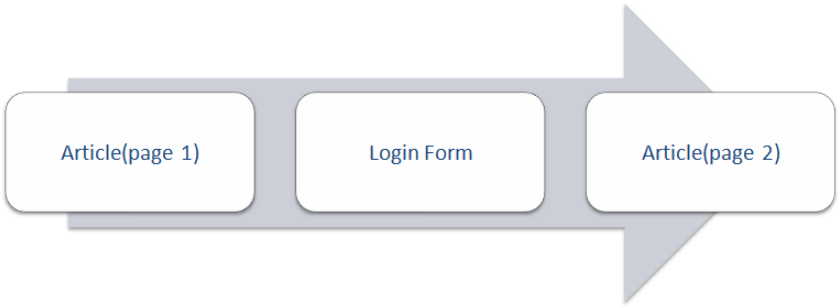


図 6.2　**Picture - Screen_Flow**

`RedirectAuthenticationHandler` の使用例を、下記に示す。

遷移元画面の JSP の記述例

```
<form:form action="${pageContext.request.contextPath}/login" method="get">
    <!-- omitted -->
  <input type="hidden" name="redirectTo"
    value="${pageContext.request.contextPath}/reservetour/read?
    ${f:query(reserveTourForm)}&page.page=${f:h(param['page.page'])}
    &page.size=${f:h(param['page.size'])}" />  <!-- (1) -->
</form:form>
```

項番	説明
(1)	hidden 項目として、「ログイン成功後に遷移するページの URL」を設定する。 hidden 項目のフィールド名 (リクエストパラメータ名) は「`redirectTo`」を指定する。 フィールド名 (リクエストパラメータ名) は、`RedirectAuthenticationHandler` の `targetUrlParameter` プロパティの値と一致させる必要がある。

ログイン画面の JSP の記述例

```
<form:form action="${pageContext.request.contextPath}/authentication" method="post">
    <!-- omitted -->
    <input type="submit"
      value="Login">
    <input type="hidden" name="redirectTo" value="${f:h(param.redirectTo)}" />  <!-- (1) -->
    <!-- omitted -->
</form:form>
```

項番	説明
(1)	hidden 項目として、遷移元画面からリクエストパラメータで渡された「ログイン成功後に遷移するページの URL」を設定する。 hidden 項目のフィールド名 (リクエストパラメータ名) は「`redirectTo`」を指定する。 フィールド名 (リクエストパラメータ名) は、`RedirectAuthenticationHandler` の `targetUrlParameter` プロパティの値と一致させる必要がある。

Spring Security 設定ファイル

```
<sec:http auto-config="true">
    <!-- omitted -->
    <!-- (1) -->
    <sec:form-login
        login-page="/login"
        login-processing-url="/authentication"
        authentication-failure-handler-ref="authenticationFailureHandler"
        authentication-success-handler-ref="authenticationSuccessHandler" />
    <!-- omitted -->
</sec:http>

<!-- (2) -->
<bean id="authenticationSuccessHandler"
    class="org.terasoluna.gfw.web.security.RedirectAuthenticationHandler">
</bean>

<!-- (3) -->
<bean id="authenticationFailureHandler"
    class="org.springframework.security.web.authentication.ExceptionMappingAuthenticationFailureHa
    <property name="defaultFailureUrl" value="/login?error=true"/> <!-- (4) -->
    <property name="useForward" value="true"/> <!-- (5) -->
</bean>
```

項番	説明
(1)	`authentication-failure-handler-ref`(認証エラー時のハンドラ設定) と `authentication-success-handler-ref`(認証成功時のハンドラ設定) の BeanId を指定する。
(2)	`authentication-success-handler-ref` から参照される bean として `org.terasoluna.gfw.web.security.RedirectAuthenticationHandler` を定義する。
(3)	`authentication-failure-handler-ref` から参照される bean として `org.springframework.security.web.authentication.ExceptionMappingAuthenticatio` を定義する。
(4)	認証失敗時の遷移先パスを指定する。 上記例では、ログイン画面のパスと認証エラー後の遷移であることを示すクエリ (`error=true`) を設定している。
(5)	**本機能を使用する場合は useForward を true に指定する必要がある。** `true` に指定することで、認証失敗時に表示する画面 (ログイン画面) に遷移する際に、Redirect ではなく Forward が使用される。 これは、認証処理を行うリクエストのリクエストパラメータの中に「ログイン成功後に遷移するページの URL」を含める必要があるためである。 Redirect を使用して認証エラー画面を表示してしまうと、「ログイン成功後に遷移するページの URL」がリクエストパラメータから引き継ぎことが出来ないため、ログインが成功しても指定した画面に遷移することが出来ない。 この事象を回避するためには、Forward を使用して「ログイン成功後に遷移するページの URL」をリクエストパラメータから引き継げるようにしておく必要がある。

ちなみに: `RedirectAuthenticationHandler` は、オープンリダイレクタ脆弱性対策が施されているため、「http://google.com」のような外部サイトへ遷移をすることはできない。外部サイトへ移動したい場合は、`org.springframework.security.web.RedirectStrategy` を実装したクラスを作成し、`RedirectAuthenticationHandler` の `targetUrlParameterRedirectStrategy` プロパティに

インジェクションすることで実現する事ができる。

拡張する際の注意点としては、`redirectTo` の値を改竄されても問題が発生しないようにする必要がある。たとえば、以下のような対策が考えられる。

- 遷移先の URL を直接指定するのではなく、ページ番号などの ID を指定して ID に対応する URL にリダイレクトする。
- 遷移先の URL をチェックし、ホワイトリストに一致する URL のみリダイレクトする。

6.4 パスワードハッシュ化

6.4.1 Overview

パスワードのハッシュ化は、セキュアなアプリケーションを設計する上で考慮しなければならない点の一つである。
通常のシステムでパスワードを平文で登録することはありえなく、ハッシュ化は必須であるが、
強度が弱いアルゴリズムを選択した場合は「オフライン総あたり攻撃」や「レインボークラック」などにより容易にハッシュ化元データを解析されてしまう。

Spring Security は、パスワードのハッシュ化の仕組みとして
`org.springframework.security.crypto.password.PasswordEncoder` インタフェースが用意している。
その実装クラスとして、

- `org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder`
- `org.springframework.security.crypto.password.StandardPasswordEncoder`

などが、提供されている。

`PasswordEncoder` の仕組みとして、`encode(String rawPassword)` メソッドでハッシュ化を行い、`matches(String rawPassword, String encodedPassword)` メソッドで照合を行う。

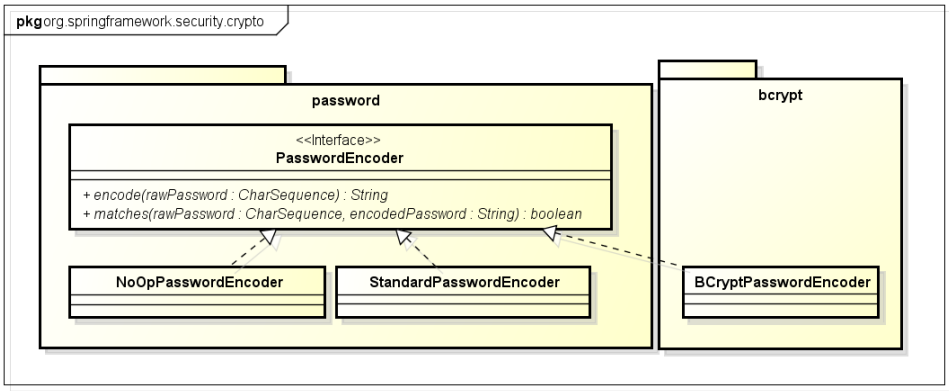


図 6.3 **Picture - PasswordEncoder Class Diagram**

6.4.2 How to use

本節では、Spring Security から提供されている、PasswordEncoder の実装クラスの使用方法について説明する。

PasswordEncoder の実装クラス一覧

PasswordEncoder の実装クラス	概要
`org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder`	“bcrypt” アルゴリズムでハッシュ化を行うエンコーダ
`org.springframework.security.crypto.password.StandardPasswordEncoder`	“SHA-256” アルゴリズム + 1024 回のストレッチでハッシュ化を行うエンコーダ
`org.springframework.security.crypto.password.NoOpPasswordEncoder`	ハッシュ化を行わないエンコーダ（テスト用）

ハッシュ化に関する要件がない場合は、`BCryptPasswordEncoder` を使用することを推奨する。ただし、`BCryptPasswordEncoder` は対攻撃性を高めるために計算時間が多いため、認証時の性能要件を満たせない場合は `StandardPasswordEncoder` を検討すること。

既存のシステムとの関係上、ハッシュ化するアルゴリズムや、ソルトに対して制限がある場合については、後述する `org.springframework.security.authentication.encoding.PasswordEncoder` インタフェースの実装クラスを使用すること。詳細は、*How to extend* を参照されたい。

BCryptPasswordEncoder

`BCryptPasswordEncoder` とは、`PasswordEncoder` を実装した、パスワードのハッシュ化を提供しているクラスである。

ランダムな 16 バイトのソルトを使用した、bcrypt アルゴリズムを使用したエンコーダーである。

注釈: Bcrypt アルゴリズムは、汎用的なアルゴリズムより意図的に計算量を増やしている。そのため、汎用アルゴリズム (SHA、MD5 など) より、「オフライン総あたり攻撃」に強い特性を持っている。

BCryptPasswordEncoder の設定例

- applicationContext.xml

```
    <bean id="passwordEncoder"
        class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" />    <!-- (1)
```

項番	説明
(1)	passwordEncoder のクラスに `BCryptPasswordEncoder` を指定する。 コンストラクタの引数に、ソルトのハッシュ化のラウンド数を指定できる。指定できる値は、4〜31 までである。 指定値を大きくすることにより、強度は増すが、計算数が指数関数的に増大するので、性能面に注意すること。 指定しない場合、「10」が設定される。

ちなみに: How to extend で後述するが、DaoAuthenticationProvider は、`org.springframework.security.crypto.password.PasswordEncoder` の実装クラス、`org.springframework.security.authentication.encoding.PasswordEncoder` の実装クラス両方を設定することができる。そのため、従来の PasswordEncoder(authentication パッケージ) から、新 PasswordEncoder に移行する際も、ユーザのパスワード移行が完了後、DaoAuthenticationProvider の passwordEncoder を変更するだけで対応できる。

警告: `DaoAuthenticationProvider` を認証プロバイダで設定している場合、`UsernameNotFoundException` がスローされた場合、利用者にユーザが存在しないことを悟らせないために、`UsernameNotFoundException` がスローされた後、意図的にパスワードをハッシュ化している。(サイドチャネル攻撃対策)
上記のハッシュ化に用いる値を作成するために、アプリケーション起動時に、`encode` メソッドを内部で 1 回実行している。

警告: Linux 環境で SecureRandom を使用している場合、処理の遅延や、タイムアウトが発生する場合がある。本問題の原因は乱数生成に関わるものであり、以下の Java Bug Database に説明がある。
http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6202721
JDK 7 の b20 以降のバージョンでは、修正されている。
http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6521844
本問題が発生する場合、JVM の起動引数に以下を設定することで、回避することができる。
-Djava.security.egd=file:///dev/urandom

- Java クラス

```
    @Inject
    PasswordEncoder passwordEncoder;  // (1)

    public String register(Customer customer, String rawPassword) {
        // omitted
        // Password Hashing
        String password = passwordEncoder.encode(rawPassword); // (2)
        customer.setPassword(password);
        // omitted
    }

    public boolean matches(Customer customer, String rawPassword) {
        return passwordEncoder.matches(rawPassword, customer.getPassword()); // (3)
    }
```

項番	説明
(1)	Bean 定義した、PasswordEncoder をインジェクションする。
(2)	パスワードをハッシュ化する例 encode メソッドの引数に平文のパスワードを指定することで、ハッシュ化されたパスワードが戻り値となる。
(3)	パスワードを照合する例 matches メソッドは、第 1 引数に平文のパスワード、第 2 引数にハッシュ化されたパスワードを指定することで、 一致しているかチェックできるメソッドである。

StandardPasswordEncoder

StandardPasswordEncoder はハッシュ化のアルゴリズムとして、SHA-256 を利用し、1024 回のストレッチを行う。
また、ランダムに生成される 8 バイトのソルトを付与している。

以下に、StandardPasswordEncoder の encode(String rawPassword) メソッド、
matches(String rawPassword, String encodedPassword) メソッドの仕組みを説明する。

encode(String rawPassword) メソッド

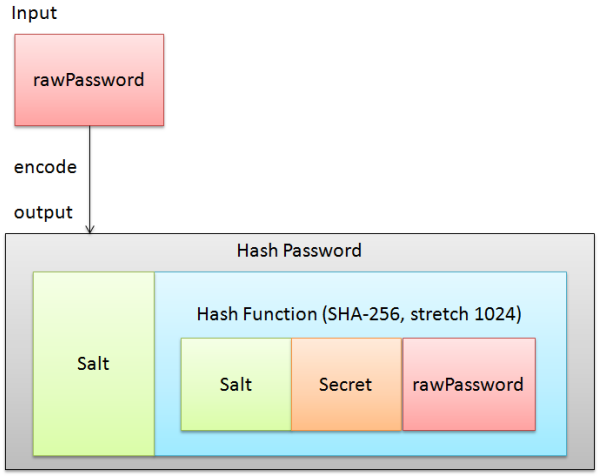


図 6.4　**Picture - encode method**

ランダムに生成される 8 バイトのソルト + 秘密鍵 + 引数に指定された、パスワードでハッシュ化される。
上記でハッシュ化された値に、ハッシュ化に用いたソルトを先頭に付与した値が、メソッドの戻り値となる。

matches(String rawPassword, String encodedPassword) メソッド

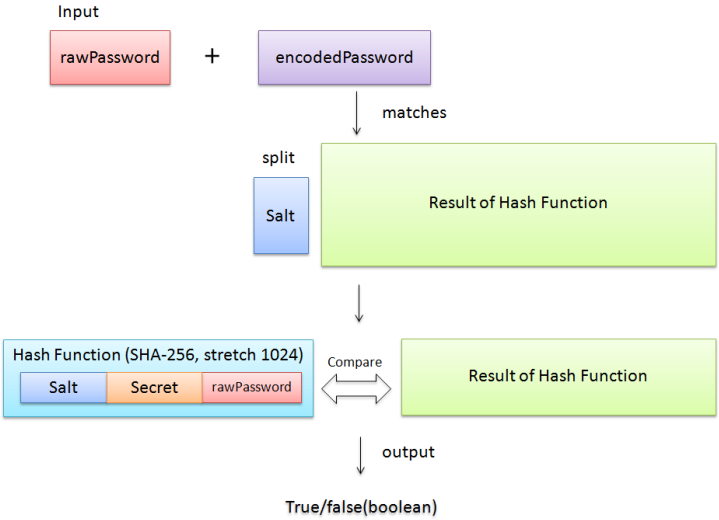


図 6.5　**Picture - matches method**

引数で渡された、encodedPassword の先頭の salt を split し、salt + secret + rawPassword でハッシュ化した値と

encodedPassword の先頭 salt を除いた値とで比較処理を行う。

StandardPasswordEncoder の設定例

- applicationContext.xml

```
<bean id="passwordEncoder"
  class="org.springframework.security.crypto.password.StandardPasswordEncoder">
  <!-- from properties file -->
  <constructor-arg value="${passoword.encoder.secret}"/> <!-- (1) -->
</bean>
```

項番	説明
(1)	ハッシュ化用の秘密鍵 (secret) を指定する。 指定した場合、ハッシュ化処理において、「内部で生成されるソルト」 + 「指定した秘密鍵」 +「パスワード」でハッシュ化される。 秘密鍵 (secret) を指定しない場合、レインボーテーブルを用いた攻撃方法に対する強度が下がるため、指定することを推奨する。 **秘密鍵 (secret) について** 秘密鍵 (secret) は、機密情報として扱うこと。 そのため、Spring Security の設定ファイルに直接指定せずプロパティファイルや、環境変数などから取得する。 本例では、プロパティファイルから取得する例が有効になっている。また本番環境ではプロパティファイルの格納場所にも注意する。

ちなみに: 秘密鍵 (secret) を環境変数から取得する場合

StandardPasswordEncoder の bean 定義の、<constructor-arg>に以下の設定を行うことで取得できる。

```
<bean id="passwordEncoder"
  class="org.springframework.security.crypto.password.StandardPasswordEncoder">
  <!-- from environment variable -->
  <constructor-arg value="#{systemEnvironment['PASSWORD_ENCODER_SECRET']}" /> <!-- (1) --
</bean>
```

項番	説明
(1)	環境変数:PASSWORD_ENCODER_SECRET から値を取得する。

Java クラス例は BCryptPasswordEncoder と同様のため、*BCryptPasswordEncoder* **の設定例**を参照されたい。

NoOpPasswordEncoder

NoOpPasswordEncoder は、指定した値をそのままの文字列で返却するエンコーダーである。
単体テスト時など、ハッシュ化されていない文字列を使用したい場合以外に使用してはいけない。

設定例は、BCryptPasswordEncoder と同様のため、省略する。

6.4.3 How to extend

業務要件によっては、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方式を踏襲したい場合などは、前述の PasswordEncoder では要件を満たせないことが多い。

たとえば、既存のハッシュ方式が、以下のような場合が考えられる。

- アルゴリズムが SHA-512 である。
- ストレッチ回数が 1000 回である。
- ソルトはアカウントテーブルのカラムに格納されており、PasswordEncoder の外から渡す必要がある。

その場合、org.springframework.security.crypto.password.PasswordEncoder を実装したクラスではなく、
異なるパッケージの
org.springframework.security.authentication.encoding.PasswordEncoder を実装したクラスの使用を推奨する。

> **警告:** Spring Security 3.1.4 以前では、org.springframework.security.authentication.encoding.Passw を実装したクラスをハッシュ化に使用していたが、3.1.4 以降では Deprecated となっている。そのため、Spring が推奨しているパターンとは異なる。

ShaPasswordEncoder を使用した例

業務要件が以下の場合、

アルゴリズムは SHA-512 を使用し、ストレッチを 1000 回を行う。

認証で説明した、

DaoAuthenticationProvider を使用した、認証処理を例に説明する。

- applicationContext.xml

```
    <bean id ="passwordEncoder"
        class="org.springframework.security.authentication.encoding.ShaPasswordEncoder"> <!-- (1
        <constructor-arg value="512" /> <!-- (2) -->
        <property name="iterations" value="1000" /> <!-- (3) -->
    </bean>
```

項番	説明
(1)	passwordEncoder には、 `org.springframework.security.authentication.encoding.ShaPasswordEncoder` を指定する。 passwordEncoder に指定する、クラスは使用するアルゴリズムに合わせて変更すること。
(2)	コンストラクタの引数に、SHA アルゴリズムの種類を設定する 指定可能な値は、「1、256、384、512」である。省略した場合は、「1」が設定される。
(3)	ハッシュ化時のストレッチングの回数を指定する。 省略した場合は、0 回となる。

- spring-mvc.xml

```
    <bean id="authenticationProvider"
        class="org.springframework.security.authentication.dao.DaoAuthenticationProvider">
        <!-- omitted -->
        <property name="saltSource" ref="saltSource" /> <!-- (1) -->
        <property name="userDetailsService" ref="userDetailsService" />
        <property name="passwordEncoder" ref="passwordEncoder" /> <!-- (2) -->
    </bean>

    <bean id="saltSource"
        class="org.springframework.security.authentication.dao.ReflectionSaltSource"> <!-- (3) -
        <property name="userPropertyToUse" value="username" /> <!-- (4) -->
```

```
    </bean>
```

項番	説明
(1)	ソルトを外部定義したい場合、 `org.springframework.security.authentication.dao.SaltSource` を 実装したクラスの BeanId を設定する。 本例では、 ユーザ情報クラスに設定された値をリフレクションで取得する、 `org.springframework.security.authentication.dao.ReflectionSaltSource` を定義している。
(2)	passwordEncoder には、 `org.springframework.security.authentication.encoding.ShaPasswordEncoder` を指定する。 passwordEncoder に指定する、クラスは使用するアルゴリズムに合わせて変更すること。
(3)	ソルトの作成方法を決める `org.springframework.security.authentication.dao.SaltSource` を指定 する。 ここでは `UserDetails` オブジェクトのプロパティをリフレクションで取得する `ReflectionSaltSource` を使用する。
(4)	`UserDetails` オブジェクトの `usernamte` プロパティを salt として使用する。

- Java クラス

```
    @Inject
    PasswordEncoder passwordEncoder;

    public String register(Customer customer, String rawPassword, String userSalt) {
        // omitted
        String password = passwordEncoder.encodePassword(rawPassword,
                userSalt); // (1)
        customer.setPassword(password);
        // omitted
    }
```

```
    public boolean matches(Customer customer, String rawPassword, String userSalt) {
        return passwordEncoder.isPasswordValid(customer.getPassword(),
                    rawPassword, userSalt); // (2)
    }
```

項番	説明
(1)	パスワードをハッシュ化する場合、 `org.springframework.security.authentication.encoding.PasswordEncoder` を実装したクラスでは、 `encodePassword` メソッドの引数にパスワードと、ソルト文字列を指定する。
(2)	パスワードを照合する場合、 `isPasswordValid` メソッドを使用し、引数にハッシュ化されたパスワード、 平文のパスワード、ソルト文字列を指定することで、ハッシュ化されたパスワードと平文のパスワードを比較する。

6.4.4 Appendix

注釈: ストレッチとは

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回数を決める必要がある。

注釈: ソルトとは

暗号化する元となるデータに追加する文字列である。ソルトをパスワードに付与することで、見かけ上、パスワード長を長くし、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ために利用する。なお、複数のユーザに対して同一のソルトを利用していると、同一パスワードを設定しているユーザが存在した時に、ハッシュ値から同一のパスワードである事が分かってしまう。そのため、ソルトはユーザごとに異なる値（ランダム値等）を設定することを推奨する。

6.5 認可

6.5.1 Overview

本節では、Spring Security で提供している認可機能を説明する。

Spring Security のアクセス認可機能を利用して実現するため、Spring Security の認証機能を用いることを前提とする。
Spring Security を利用した認証方法については、認証を参照されたい。

アクセス認可の対象のリソースは、以下の 3 項目である。

1. Web(リクエスト URL)
   - 特定の URL にアクセスするために必要な権限を設定できる
2. 画面項目 (JSP）
   - 画面中の特定の要素を表示するために必要な権限を設定できる
3. メソッド
   - 特定のメソッドを実行するために必要な権限を設定できる

Spring Security では、設定ファイルやアノテーションでアクセス認可情報を記述し、機能を実現している。

アクセス認可 (リクエスト URL)

1. ユーザのリクエストに対し、Spring Security のフィルタチェーンが割り込み処理を行う。
2. 認可制御の対象となる URL とリクエストのマッチングを行い、アクセス認可マネージャにアクセス認可の判断を問い合わせる。
3. アクセス認可マネージャが、ユーザの権限とアクセス認可情報をチェックし、必要なロールが割り当てられていない場合は、アクセス拒否例外をスローする。
4. 必要なロールが割り当てられている場合は、処理を継続する。

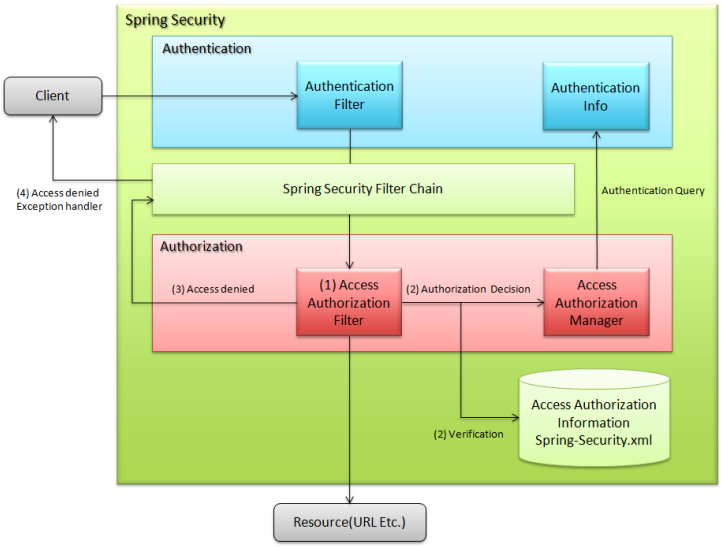


図 6.6　**Picture - Authorization(リクエスト URL)**

アクセス認可 (JSP)

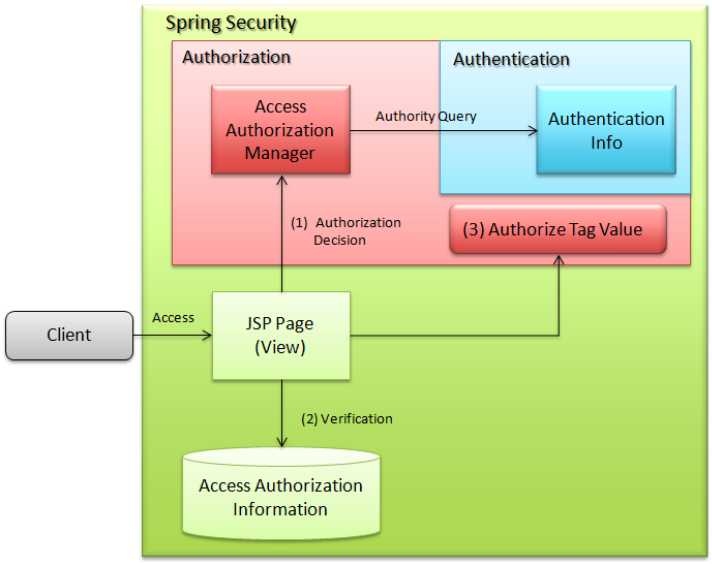


図 6.7　**Picture - Authorization(JSP)**

1. JSP から生成されたサーブレットが、アクセス認可マネージャに問い合わせる。
2. アクセス認可マネージャが、ユーザの権限とアクセス認可情報をチェックし、必要なロールが割り当てられていない場合は、タグの内部を評価しない。
3. 必要なロールが割り当てられている場合は、タグの内部を評価する。

アクセス認可 (Method)

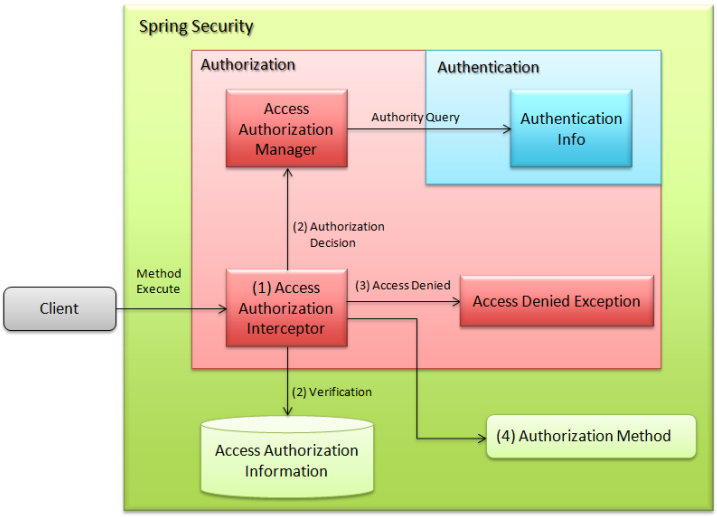


図 6.8　**Picture - Authorization(Method)**

1. Spring コンテナがアクセス認可情報をもとに、対象のオブジェクトに対してインターセプタを生成、割り込みさせる。
2. インターセプタは設定されたロールをもとにアクセス認可マネージャに問い合わせる。
3. アクセス認可マネージャが、ユーザが持つ権限とアクセス認可情報をチェックし、必要なロールが割り当てられていない場合はアクセス拒否例外をスローする。
4. 必要なロールが割り当てられている場合は、処理を継続する（設定により、処理を実行した後に権限をチェックすることもできる）。

6.5.2 How to use

アクセス認可 (リクエスト URL)、アクセス認可 (JSP)、アクセス認可 (Method) の使用方法について説明する。

アクセス認可 (リクエスト URL)

アクセス認可 (リクエスト URL) 機能を使用するために、Spring Security の設定ファイルに記述する内容を以下に示す。

基本設定については、Spring Security 概要を参照されたい。

<sec:intercept-url>要素の設定

<sec:http>要素の子要素である<sec:intercept-url>要素に制御対象とする URL、認可するロールを記述することで、
URL のパス単位で認可制御を行うことができる。

以下に、設定例を記載する。

- spring-security.xml

```
<sec:http auto-config="true" use-expressions="true">
    <sec:intercept-url pattern="/admin/*" access="hasRole('ROLE_ADMIN')"/>
    <!-- omitted -->
</sec:http>
```

属性名	説明
pattern	アクセス認可を行う対象の URL パターンを記述する。ワイルドカード「*」、「**」が使用できる。 「*」では、同一階層のみが対象であるのに対し、「**」では、指定階層以下の全 URL が、認可設定の対象となる。
access	Spring EL 式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。
method	HTTP メソッド（GET や POST 等）を指定する。指定したメソッドのみに関して、URL パターンとマッチングを行う。 指定しない場合は、任意の HTTP メソッドに適用される。主に REST を利用した Web サービスの利用時に活用できる。
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。 指定しない場合、どちらでもアクセスできる。

上記以外の属性については、B.1.10 <intercept-url>を参照されたい。

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。

- spring-security.xml

```
<sec:http auto-config="true" use-expressions="true">
    <sec:intercept-url pattern="/reserve/*" access="hasAnyRole('ROLE_USER','ROLE_ADMIN')" />
    <sec:intercept-url pattern="/admin/*" access="hasRole('ROLE_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。

`<sec:http>`属性に `use-expressions="true"`の設定をしたことで、Spring EL 式が有効になる。

access 属性に記述した Spring EL 式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

```
<sec:http auto-config="true" use-expressions="true">
    <sec:intercept-url pattern="/admin/*" access="hasRole('ROLE_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 Expression 一覧例

属性名	説明
hasRole('ロール名')	ユーザが指定したロールを保持していれば、真を返す。
hasAnyRole('ロール1','ロール2')	ユーザが指定したいずれかのロールを保持していれば、真を返す。
permitAll	常に真を返す。認証されていない場合も、アクセスできることに注意する。
denyAll	常に偽を返す。
isAnonymous()	匿名ユーザであれば、真を返す。
isAuthenticated()	認証されたユーザならば、真を返す。
isFullyAuthenticated()	匿名ユーザ、もしくは RememberMe 機能での認証であれば、偽を返す。
hasIpAddress('IP アドレス')	リクエスト URL、および JSP タグへのアクセス認可のみで、有効となる。 指定の IP アドレスからのリクエストであれば、真を返す。

その他、使用可能な Spring EL 式は、 Common built-in expressions を参照されたい。

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

```
<sec:http auto-config="true" use-expressions="true">
    <sec:intercept-url pattern="/admin/*" access="hasRole('ROLE_ADMIN') and hasIpAddress('19
    <!-- 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。

アクセス認可制御を行わない URL の設定

トップページやログイン画面、css ファイルへのパスなど、認証が必要のない URL に対しては、
http 要素の pattern 属性、および security 属性を利用する。

- spring-security.xml

```
<sec:http pattern="/css/*" security="none"/>  <!-- 属性の指定順番で (1)～(2) -->
<sec:http pattern="/login" security="none"/>
<sec:http auto-config="true" use-expressions="true">
    <!-- omitted -->
</sec:http>
```

項番	説明
(1)	pattern 属性に設定を行う対象の URL パターンを記述する。pattern 属性を記述しない場合、すべてのパターンにマッチする。
(2)	security 属性に none を指定することで、pattern 属性に記述されたパスは、Spring Security フィルタチェインを回避することができる。

URL パターンでの例外処理

認可されていない URL にアクセスした場合、
org.springframework.security.access.AccessDeniedException がスローされる。
デフォルトの設定では、
org.springframework.security.web.access.ExceptionTranslationFilter に設定された
org.springframework.security.web.access.AccessDeniedHandlerImpl が、エラーコード 403 を返却する。
http 要素に、アクセス拒否時のエラーページを設定することで、アクセス拒否時に指定のエラーページに遷移させることができる。

- spring-security.xml

```
<sec:http auto-config="true" use-expressions="true">
    <!-- omitted -->
    <sec:access-denied-handler error-page="/accessDeneidPage" />  <!-- (1) -->
</sec:http>
```

項番	説明
(1)	<sec:access-denied-handler>要素の error-page 属性に、遷移先のパスを指定する。

アクセス認可 (JSP)

画面表示項目を制御するには、Spring Security が提供しているカスタム JSP タグ<sec:authorize>を利用する。

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags"
%>
```

のタグライブラリの使用宣言設定をされていることが、前提条件である。

- <sec:authorize>タグの属性一覧

属性名	説明
access	アクセス制御式を記述する。真であれば、タグ内が評価される。
url	設定した URL に対して権限が与えられている場合に、タグ内が評価される。リンクの表示の制御等に利用する。
method	HTTP メソッド（GET や POST 等）を指定する。 url 属性と合わせて利用し、指定したメソッドのみに関して、 指定した URL パターンとマッチングを行う。指定しない場合、GET が適用される。
ifAllGranted	設定したロールが全て与えられている場合に、タグ内が評価される。ロール階層機能は効かない。
ifAnyGranted	設定したロールについて、いずれかが与えられている場合に、タグ内が評価される。 ロール階層機能は効かない。
ifNotGranted	設定されたロールが与えられていない場合、タグの中身が評価される。ロール階層機能は効かない。
var	タグの評価結果を格納する page スコープの変数を宣言する。同等の権限チェックをページ内で行う場合に利用する。

以下に、<sec:authorize>タグの使用例を示す。

- spring-security.xml

```
<div>
  <sec:authorize access="ROLE_USER">  <!-- (1) -->
      <p>This screen is for ROLE_USER</p>
  </sec:authorize>
  <sec:authorize url="/admin/menu">  <!-- (2) -->
      <p>
        <a href="/admin/menu">Go to admin screen</a>
      </p>
  </sec:authorize>
</div>
```

項番	説明
(1)	「ROLE_USER」を持つ場合のみ、タグ内が表示される。
(2)	「/admin/menu」に対してアクセスが認可されている場合、タグ内が表示される。

警告: <sec:authorize>タグによる認可処理は、**画面表示の制御でしかない**ため、特定の権限でリンクを表示されなくても、URL が推測されれば、直接リンク先の URL にアクセスできてしまう。そのため、必ず、前述の「アクセス認可 (リクエスト URL)」、もしくは、後述の「アクセス認可 (Method)」を併用して、本質的な認可制御をに行うこと。

アクセス認可 (Method)

メソッドに対して、認可制御ができる。

Spring の DI コンテナで管理されている Bean が、認可の対象となる。

前述の 2 つの認可方法はアプリケーション層での認可制御であったが、

メソッドレベルの認可制御はドメイン層 (Service クラス) に対して行う。

制御したいメソッドに対して

`org.springframework.security.access.prepost.PreAuthorize` アノテーションを設定すればよい。

- 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。 デフォルトは disabled である。

- Java コード

```
@Service
@Transactional
public class UserServiceImpl implements UserSerice
    // omitted

    @PreAuthorize("hasRole('ROLE_ADMIN')") // (1)
    @Override
    public User create(User user) {
        // omitted
    }


    @PreAuthorize("isAuthenticated()")
    @Override
    public User update(User user) {
        // omitted
    }
}
```

項番	説明
(1)	アクセス制御式を記述する。メソッドを実行する前に式が評価され、真であれば、メソッドが実行される。 偽であれば、org.springframework.security.access.AccessDeniedException がスローされる。 設定可能な値は、*<sec:intercept-url>要素の設定*で述べた Expression や、および Spring Expression Language (SpEL) で記述された式である。

ちなみに: 上記の設定では `org.springframework.security.access.prepost.PreAuthorize` 以外にも、以下のアノテーションを使用できる。

- `org.springframework.security.access.prepost.PostAuthorize`
- `org.springframework.security.access.prepost.PreFilter`
- `org.springframework.security.access.prepost.PostFilter`

これらの詳細は Spring Security マニュアルを参照されたい。

注 釈: Spring Security では Java 標準である JSR-250 の `javax.annotation.security.RolesAllowed` アノテーションによる認可制御も可能であるが、`@RolesAllowed` では SpEL による記述ができない。`@PreAuthorize` であれば SpEL を用いて、spring-security.xml の設定と同じ記法で認可制御

注釈: リクエストパスに対する認可制御は Controller のメソッドにアノテーションをつけるのではなく、spring-security.xml に設定を行うことを推奨する。

Service が Web 経由でしか実行されず、リクエストパスのすべてのパターンが認可制御されているのであれば Service の認可制御は行わなくても良い。Service がどこから実行されるか分からず、認可制御が必要な場合にアノテーションを使用するとよい。

6.5.3 How to extend

ロール階層機能

ロールに階層関係を設定することができる。
上位に設定したロールは、下位ロールに認可されたすべてのアクセスが可能となる。
ロールの関係が複雑な場合は、階層機能を検討されたい。

ROLE_ADMIN を上位ロール、ROLE_USER を下位ロールとして階層関係を設定する例で説明する。

このとき、下記のようにアクセス認可を設定すると、
「ROLE_ADMIN」のロールを持つユーザも、「/user/*」の URL にアクセスできる。

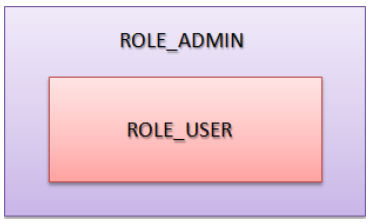


図 6.9 **Picture - RoleHierarchy**

Spring Security 設定ファイル

```
<sec:http auto-config="true" use-expressions="true">
    <sec:intercept-url pattern="/user/*" access="hasAnyRole('ROLE_USER')" />
    <!-- omitted -->
</sec:http>
```

アクセス認可 (リクエスト URL)、アクセス認可 (JSP)、アクセス認可 (Method) のそれぞれで設定方法が異なるため、
使用方法について、以降で説明する。

共通設定

共通で必要な設定について述べる。
階層関係を管理する
`org.springframework.security.access.hierarchicalroles.RoleHierarchy` クラスの Bean 定義を行う。

- spring-security.xml

```
    <bean id="roleHierarchy"
        class="org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl"> <!-- (1
        <property name="hierarchy">
            <value> <!-- (2) -->
                ROLE_ADMIN > ROLE_STAFF
                ROLE_STAFF > ROLE_USER
            </value>
        </property>
    </bean>
```

項番	説明
(1)	RoleHierarchy のデフォルト org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl クラスを指定する。
(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] 例では、STAFF は USER に認可されたすべてのリソースに、アクセスできる。 ADMIN は USER、STAFF に認可されたすべてのリソースに、アクセスできる。

アクセス認可 (リクエスト URL)、アクセス認可 (JSP) での使用方法

リクエスト URL、JSP に対するロール階層の設定について述べる。

- spring-security.xml

```
    <bean id="webExpressionHandler"
        class="org.springframework.security.web.access.expression.DefaultWebSecurityExpressionHa
        <property name="roleHierarchy" ref="roleHierarchy"/>  <!-- (2) -->
    </bean>

    <sec:http auto-config="true" use-expression="true">
        <!-- omitted -->
        <sec:expression-handler ref="webExpressionHandler" />  <!-- (3) -->
    </sec:http>
```

項番	説明
(1)	クラスに `org.springframework.security.web.access.expression.DefaultWebSecurityExpr` を指定する。
(2)	`roleHierarchy` プロパティに `RoleHierarchy` の Bean ID をプロパティに設定する。
(3)	`expression-handler` 要素に、`org.springframework.security.access.expression.SecurityExpressionHandler` を実装したハンドラクラスの Bean ID を指定する。

アクセス認可 (Method) での使用方法

Service のメソッドにアノテーションをつけて認可制御を行う場合のロール階層設定について説明する。

- spring-security.xml

```
<bean id="methodExpressionHandler"
    class="org.springframework.security.access.expression.method.DefaultMethodSecurityExpres
    <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)	クラスに `org.springframework.security.access.expression.method.DefaultMethodSecuri` を指定する。
(2)	`roleHierarchy` プロパティに `RoleHierarchy` の Bean ID をプロパティに設定する。
(3)	`expression-handler` 要素に、 `org.springframework.security.access.expression.SecurityExpressionHandler` を実装したハンドラクラスの Bean ID を指定する。

6.6 XSS 対策

6.6.1 Overview

クロスサイトスクリプティング (以下、XSS と略す) 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、大きく分けて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。

6.6.2 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。
エスケープが必要な HTML 上の特殊文字の例と、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()` を使用することを必須とする。
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 6.10 **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。

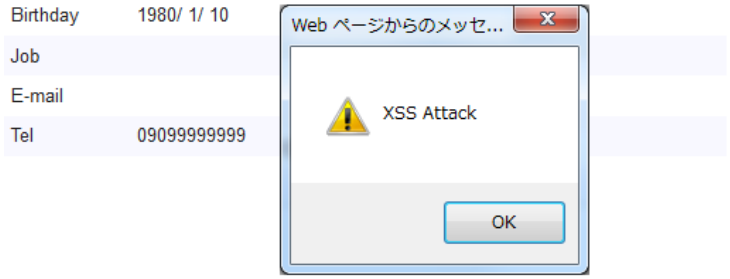


図 6.11 **Picture - No Escape Result**

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。

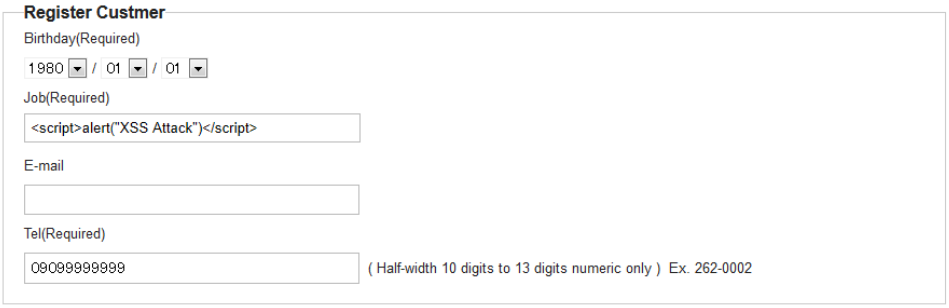


図 6.12 **Picture - Input HTML Tag**

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。

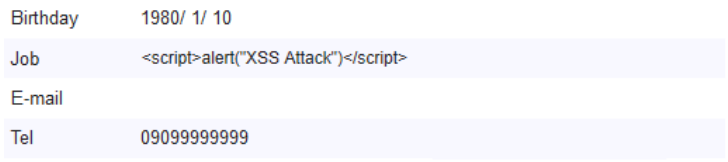


図 6.13 **Picture - Escape Result**

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。

エスケープが必要な JavaScript の特殊文字の例と、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。

```
<html>
  <script  type="text/javascript">
      var aaa = '<script>${warnCode}<\/script>';
      document.write(aaa);
  </script>
<html>
```

属性名	値
warnCode	<script></script><script>alert('XSS Attack!');</script><\/script>

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。

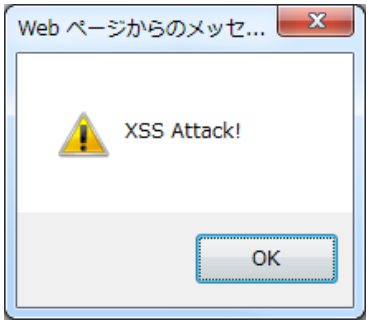


図 6.14 **Picture - No Escape Result**

出力結果

```
<script type="text/javascript">
    var aaa = '<script><\/script><script>alert('XSS Attack!');<\/script><\/script>';
    document.write(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 message = '<script>${f:js(message)}<\/script>';  // (1)
  <!-- omitted -->
</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。

出力結果

```
<script  type="text/javascript">
    var aaa = '<script>\x3c\/script\x3e\x3cscript\x3ealert(\'XSS Attack!\');\x3c\/script\x3e<\/sc
    document.write(aaa);
</script>
```

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。

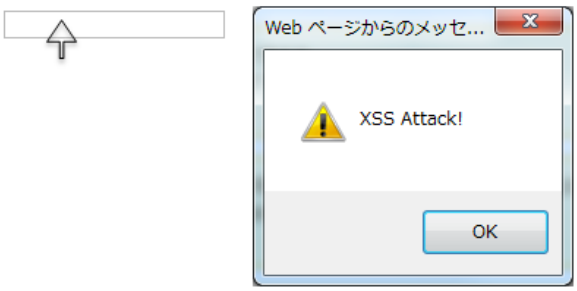


図 6.15　**Picture - No Escape Result**

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。



図 6.16　**Picture - Escape Result**

出力結果

```
<!-- omitted -->
<input type="text" onmouseover="alert('output is \&#39;); alert(\&#39;XSS Attack!\&#39;);\&quot;
<!-- omitted -->
```

6.7 CSRF 対策

6.7.1 Overview

Cross site request forgeries(以下、CSRF と略す）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、
ユーザが、ログイン済みの別の Web サイト上で、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
- Refer のチェック

OWASP 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。

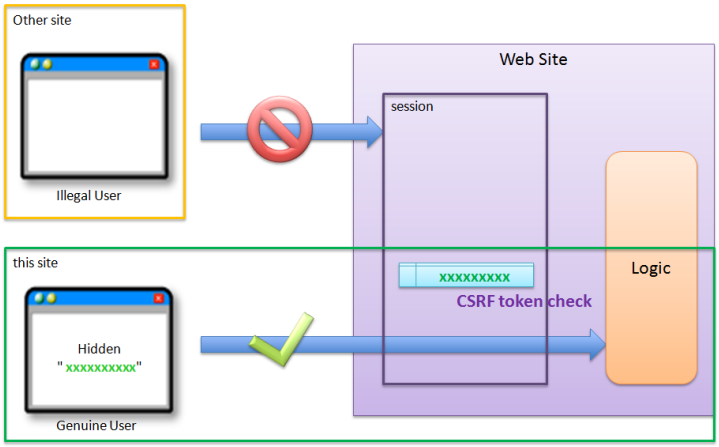


図 6.17　**Picture - csrf check other site**

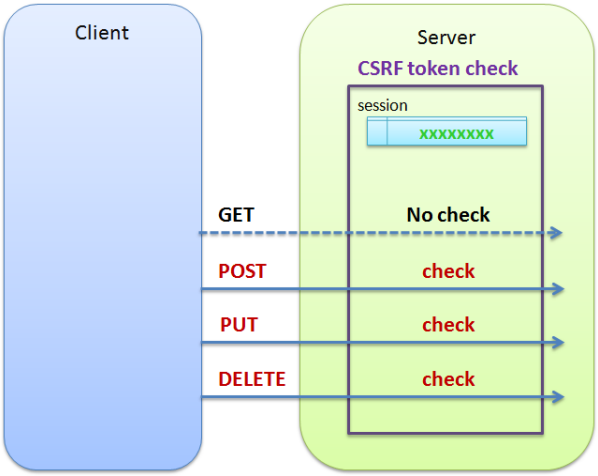
注釈: OWASP とは

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

CSRF を回避する方法は、前述したように複数あるが、固定トークンを使用するライブラリを、Spring Security が提供している。
セッション毎に 1 つの固定トークンを用い、すべてのリクエストについて、同じ値を使用している。

デフォルトでは HTTP メソッドが、GET,HEAD,TRACE,OPTIONS 以外の場合、
リクエストに含まれる CSRF トークンをチェックし、値が一致しない場合は、エラー (HTTP Status:403[Forbidden]) とする。



Picture - csrf check other kind

ちなみに: CSRF トークンチェックは、別サイトからの不正な更新リクエストをチェックし、エラーとするものである。ユーザに順序性（一連の業務フロー）を守らせ、チェックするためには、**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。

警告: CSRF 対策機能は、Spring Security3.2 から提供される機能であるが、共通ライブラリ (terasoluna-gfw-security-web) の 1.0.0.RELEASE 版が依存している Spring Security のバージョンは、3.1.4.RELEASE である (共通ライブラリの 1.0.0.RELEASE 版リリース時には、Spring Security の 3.2.0.RELEASE 版は未リリースであるため)。このため、terasoluna-gfw-security-web プロジェクト内に、1.0.0.RELEASE 版リリース時の Sprinng Security の CSRF 対策機能に関する以下のクラスが同梱されている。

- org.springframework.security.web.authentication.session.CompositeSessionAuthenticationStrategy
- org.springframework.security.web.csrf.CsrfAuthenticationStrategy
- org.springframework.security.web.csrf.CsrfFilter
- org.springframework.security.web.csrf.CsrfLogoutHandler
- org.springframework.security.web.csrf.CsrfToken
- org.springframework.security.web.csrf.CsrfTokenRepository
- org.springframework.security.web.csrf.DefaultCsrfToken
- org.springframework.security.web.csrf.HttpSessionCsrfTokenRepository
- org.springframework.security.web.csrf.InvalidCsrfTokenException
- org.springframework.security.web.servlet.support.csrf.CsrfRequestDataValueProcessor

共通ライブラリのバージョンアップのタイミングで、Spring Security のバージョンアップし、上記のクラスは、terasoluna-gfw-security-web プロジェクトからは取り除かれる予定である。

6.7.2 How to use

Spring Security の設定

Spring Security の CSRF 機能を使用するための設定を説明する。*Spring Security の How to use* で設定した web.xml を前提とする。

spring-security.xml の設定

追加で設定が必要な箇所を、ハイライトしている。

```
<sec:http auto-config="true" use-expressions="true" >
    <!-- omitted -->
    <sec:custom-filter ref="csrfFilter" before="LOGOUT_FILTER" />  <!-- (1) -->

    <sec:session-management
        session-authentication-strategy-ref="sessionAuthenticationStrategy" />  <!-- (2) -->
    <!-- omitted -->
</sec:http>

<bean id="csrfFilter" class="org.springframework.security.web.csrf.CsrfFilter">  <!-- (3) -->
    <constructor-arg index="0" ref="csrfTokenRepository" />  <!-- (4) -->
```

```
    <property name="accessDeniedHandler">
        <bean
            class="org.springframework.security.web.access.AccessDeniedHandlerImpl">  <!-- (5) --
            <property name="errorPage"
                value="/WEB-INF/views/common/error/csrfTokenError.jsp" />  <!-- (6) -->
        </bean>
    </property>
</bean>

<bean id="csrfTokenRepository"
    class="org.springframework.security.web.csrf.HttpSessionCsrfTokenRepository" />  <!-- (7) --

<bean id="sessionAuthenticationStrategy"
    class="org.springframework.security.web.authentication.session.CompositeSessionAuthenticatio
    <constructor-arg index="0">
        <list>
            <!-- omitted -->
            <bean
                class="org.springframework.security.web.csrf.CsrfAuthenticationStrategy">  <!--
                <constructor-arg index="0"
                    ref="csrfTokenRepository" />  <!-- (10) -->
            </bean>
        </list>
    </constructor-arg>
</bean>
```

項番	説明
(1)	<sec:custom-filter>要素を定義し、org.springframework.security.web.authentication.logout.LogoutFilter の前に CSRF の Filter 定義を行う。
(2)	<sec:session-management>要素の、session-authentication-strategy-ref 属性で、 org.springframework.security.web.authentication.session.SessionAuthentication を参照する。
(3)	org.springframework.security.web.csrf.CsrfFilter の bean 定義を行う。
(4)	コンストラクタの第 1 引数で、トークンの作成、保持を行う org.springframework.security.web.csrf.CsrfTokenRepository を参照する。
(5)	accessDeniedHandler プロパティに org.springframework.security.web.access.AccessDeniedHandlerImpl を bean 定義する。
(6)	AccessDeniedHandlerImpl の errorPage プロパティに、リクエストに含まれる CSRF トークンが、一致しない場合の遷移先パスを設定する。 設定を省略した場合、リクエストに含まれる CSRF トークンが一致しない場合、ステータスコード 403 でクライアントに返却する。
(7)	CsrfTokenRepository の実装として HTTP セッションに CSRF トークンを保存する、org.springframework.security.web.csrf.HttpSessionCsrfTokenRepository クラスを定義する。
(8)	SessionAuthenticationStrategy の実装として、複数の SessionAuthenticationStrategy を使用できる org.springframework.security.web.authentication.session.CompositeSessionAuthe
(9)	CompositeSessionAuthenticationStrategy に、

注釈: AccessDeniedHandlerImpl の errorPage プロパティを省略した場合の、エラーハンドリングについて

web.xml に、以下の設定を行うことで、任意のページに遷移させることができる。

web.xml

```
    <error-page>
        <error-code>403</error-code>  <!-- (1) -->
        <location>/WEB-INF/views/common/error/csrf-error.jsp</location>  <!-- (2) -->
    </error-page>
```

項番	説明
(1)	error-code 要素に、ステータスコード 403 を設定する。
(2)	location 要素に、遷移先のパスを設定する。

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、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。詳細は、Spring Security のレファレンスドキュメントを参照されたい。

課題

Spring Security のバージョンが、 3.2.0 以降の場合の設定

Spring Security 3.2 を使用する場合、`<sec:http>`要素に`<sec:csrf />`要素を設定することで、前述した設定を省略することができる。

Spring Security のレファレンスドキュメントを参照されたい。

spring-mvc.xml の設定

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 に埋め込むことができる。

```
<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.CsrfRequestDataValue
                factory-method="create" /> <!-- (2)  -->
            <bean
                class="org.terasoluna.gfw.web.token.transaction.TransactionTokenRequestDataValue
        </util:list>
    </constructor-arg>
</bean>
```

項番	説明
(1)	`org.terasoluna.gfw.web.mvc.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.CsrfRequestDataValuePro` の bean 定義を設定する。 factory-method に、create メソッドを指定する。

注釈: CSRF トークンの生成、チェックは、`CsrfFilter` が行うため、Controller では特に、CSRF 対策は意識しなくてよい。

フォームによる CSRF トークンの送信

JSP で、フォームから CSRF トークンを送信するには

- `<form:form>`タグを使用して CSRF トークンが埋め込まれた`<input type="hidden">`タグを自動的に追加する
- `<input type="hidden">`タグを作成し、明示的に CSRF トークンを埋め込む

のどちらかを行う必要がある。

CSRF トークンを自動で埋め込む方法

spring-mvc.xml **の設定**の通り、CsrfRequestDataValueProcessor が定義されている場合、<form:form>タグを使うことで、CSRF トークンが埋め込まれた<input type="hidden">タグが、自動的に追加される。

JSP で、CSRF トークンを意識する必要はない。

```
<form:form method="POST"
  action="${pageContext.request.contextPath}/csrfTokenCheckExample">
  <input type="submit" name="second" value="second" />
</form:form>
```

以下のような HTML が、出力される。

```
<form action="/terasoluna/csrfTokenCheckExample" method="POST">
  <input type="submit" name="second" value="second" />
  <input type="hidden" name="_csrf" value="dea86ae8-58ea-4310-bde1-59805352dec7" /> <!-- (1) -->
</form>
```

項番	説明
(1)	Spring Security のデフォルト実装では、name 属性に_csrf が設定されている <input type="hidden">タグが追加され、CSRF トークンが埋め込まれる。

CSRF トークンはログインのタイミングで生成される。

警告: CsrfRequestDataValueProcessor を設定している状態で <form:form method="GET" ...>...</form:form>とした場合 (GET メソッドを指定してフォームを送信した場合)、

- ブラウザのアドレスバーに CSRF トークンが表示される
- ブックマークした場合、ブックマークに CSRF トークンが記録される
- Web サーバのアクセスログに CSRF トークンが記録される

ため、攻撃者に CSRF トークンを悪用されるリスクが高くなる。
この事象を回避するためには、

```
<form:form method="GET" modelAttribute="xxxForm" action="...">
...
</form:form>
```

と書く代わりに、

```
<form method="GET" action="...">
  <spring:nestedPath path="xxxForm">
  ...
  </spring:nestedPath>
</form>`
```

と記述すればよい。
OWASP Top 10 では

> The unique token can also be included in the URL itself, or a URL parameter. However, such placement runs a greater risk that the URL will be exposed to an attacker, thus compromising the secret token.

と説明されており、必須ではないが対応することが推奨される。
Spring Security のデフォルト実装では、CSRF トークンの値としてランダムな UUID を生成しているため、仮に CSRF トークンが漏洩してもセッションハイジャックされる事はないという点を補足しておく。
また、Spring 4 を使用すると、この問題は解消される。(<form:form method="GET">を使用しても CSRF トークンは URL に現れない)。

CSRF トークンを明示的に埋め込む方法

<form:form>タグを使用しない場合は、明示的に、<input type="hidden">タグを追加する必要がある。

CsrfFilter により、org.springframework.security.web.csrf.CsrfToken オブジェクトが、リクエストスコープの_csrf 属性に設定されるため、jsp では、以下のように設定すればよい

```
<form method="POST"
  action="${pageContext.request.contextPath}/csrfTokenCheckExample">
    <input type="submit" name="second" value="second" />
    <input type="hidden" name="${f:h(_csrf.parameterName)}" value="${f:h(_csrf.token)}"/>  <!-- (
</form>
```

以下のような HTML が、出力される。

```
<form action="/terasoluna/csrfTokenCheckExample" method="POST">
  <input type="submit" name="second" value="second" />
  <input type="hidden" name="_csrf" value="dea86ae8-58ea-4310-bde1-59805352dec7"/>  <!-- (2) -->
</form>
```

項番	説明
(1)	_csrf.parameterName でリクエストパラメータ名を、_csrf.token で、CSRF トークンを設定する。
(2)	Spring Security のデフォルト実装では、name 属性に_csrf が設定されている <input type="hidden">タグが追加され、CSRF トークンが埋め込まれる。

注釈: CSRF トークンチェック対象のリクエスト (デフォルトでは、HTTP メソッドが、GET, HEAD, TRACE, OPTIONS 以外の場合) で、CSRF トークンがない、またはサーバー上に保存されているトークン値と、送信されたトークン値が異なる場合は、AccessDeniedHandler によりアクセス拒否処理が行われる。デフォルトでは 403 エラーとなり、AccessDeniedHandlerImpl の errorPage プロパティで指定したエラーページに遷移する。詳細は、*spring-security.xml* **の設定**を参照されたい。

Ajax による CSRF トークンの送信

CsrfFilter は、前述のようにリクエストパラメータから CSRF トークンを取得するだけでなく、
HTTP リクエストヘッダーからも CSRF トークンを取得する。
Ajax を利用する場合は HTTP ヘッダーに、CSRF トークンを設定することを推奨する。JSON 形式でリクエストを送る場合にも対応できるためである。

注釈: HTTP ヘッダ、リクエストパラメータの両方から CSRF トークンが送信する場合は、HTTP ヘッダの値が優先される。

Ajax で使用した例を用いて、説明を行う。追加で設定が必要な箇所を、ハイライトしている。

jsp の実装例

```
<!-- omitted -->
<head>
  <meta name="_csrf" content="${f:h(_csrf.token)}"/>  <!-- (1) -->
  <meta name="_csrf_header" content="${f:h(_csrf.headerName)}"/>  <!-- (2) -->
  <!-- omitted -->
</head>
<!-- omitted -->
```

```
<script type="text/javascript">
var contextPath = "${pageContext.request.contextPath}";
var token = $("meta[name='_csrf']").attr("content");  <!-- (3) -->
var header = $("meta[name='_csrf_header']").attr("content");  <!-- (4) -->
$(document).ajaxSend(function(e, xhr, options) {
    xhr.setRequestHeader(header, token);  <!-- (5) -->
});

$(function() {
    $('#calcButton').on('click', function() {
        var $form = $('#calcForm'),
             $result = $('#result');
        $.ajax({
            url : contextPath + '/sample/calc',
            type : 'POST',
            data: $form.serialize(),
        }).done(function(data) {
            $result.html('add: ' + data.addResult + '<br>'
                         + 'subtract: ' + data.subtractResult + '<br>'
                         + 'multipy: ' + data.multipyResult + '<br>'
                         + 'divide: ' + data.divideResult + '<br>'); // (6)
        }).fail(function(data) {
            // error handling
            alert(data.statusText);
        });
    });
});
</script>
```

項番	説明
(1)	<meta>タグに、${f:h(_csrf.token)}で取得した CSRF トークンを設定する。
(2)	<meta>タグに、${f:h(_csrf.headerName)}で取得したヘッダ名を設定する。
(3)	<meta>タグに、設定した CSRF トークンを取得する。
(4)	<meta>タグに、設定した CSRF ヘッダ名を取得する。
(5)	リクエストヘッダーに、<meta>タグで設定したヘッダ名 (デフォルト:X-CSRF-TOKEN)、CSRF トークンの値を設定する。
(6)	この書き方は XSS の可能性があるので、実際に JavaScript コードを書くときは気を付けること。 今回の例では data.addResult、data.subtractResult、data.multipyResult、data.divideResult の全てが数値型であるため、問題ない。

JSON でリクエストを送信する場合も、同様に HTTP ヘッダを設定すればよい。

課題

Ajax 対応する例がなくなっているため、例を直す。

マルチパートリクエスト (ファイルアップロード) 時の留意点

一般的に、ファイルアップロードなどマルチパートリクエストを送る場合、form から送信される値を Filter では取得できない。
そのため、これまでの説明だけでは、マルチパートリクエスト時に CsrfFileter が CSRF トークンを取得できず、不正なリクエストと見なされてしまう。

そのため、以下のどちらかの方法によって、対策する必要がある。

- org.springframework.web.multipart.support.MultipartFilter を使用する
- クエリのパラメータで CSRF トークンを送信する

注釈: それぞれメリット・デメリットが存在するため、システム要件を考慮して、採用する対策方法を決めて頂きたい。

ファイルアップロードの詳細については、FileUpload を参照されたい。

MultipartFilter を使用する方法

通常、マルチパートリクエストの場合、form から送信された値は Filter 内で取得できない。
org.springframework.web.multipart.support.MultipartFilter を使用することで、マルチパートリクエストでも、Filter 内で、
form から送信された値を取得することができる。

警告: MultipartFilter を使用した場合、springSecurityFilterChain による認証・認可処理が行われる前にアップロード処理が行われるため、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) を許容してしまう。

MultipartFilter を使用するには、以下のように設定すればよい。

web.xml の設定例

```
<filter>
    <filter-name>MultipartFilter</filter-name>
    <filter-class>org.springframework.web.multipart.support.MultipartFilter</filter-class> <!-- (
</filter>
<filter>
    <filter-name>springSecurityFilterChain</filter-name> <!-- (2) -->
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
    <filter-name>MultipartFilter</filter-name>
    <servlet-name>/*</servlet-name>
</filter-mapping>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

項番	説明
(1)	org.springframework.web.multipart.support.MultipartFilter を 定義する。
(2)	springSecurityFilterChain より前に、MultipartFilter を定義すること。

JSP の実装例

```
<form:form action="${pageContext.request.contextPath}/fileupload"
    method="post" modelAttribute="fileUploadForm" enctype="multipart/form-data">  <!-- (1) -->
    <table>
        <tr>
            <td width="65%"><form:input type="file" path="uploadFile" /></td>
        </tr>
        <tr>
            <td><input type="submit" value="Upload" /></td>
        </tr>
    </table>
</form:form>
```

項番	説明
(1)	spring-mvc.xml の設定の通り、CsrfRequestDataValueProcessor が定義されている場合、<form:form>タグを使うことで、CSRF トークンが埋め込まれた<input type="hidden">タグが自動的に追加されるため、 JSP の実装で、CSRF トークンを意識する必要はない。 **<form> タグを使用する場合** *CSRF* **トークンを明示的に埋め込む方法**で CSRF トークンを設定すること。

クエリパラメータで CSRF トークンを送る方法

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) を防ぎたい場合は、MultipartFilter は使用せず、クエリパラメータで CSRF トークンを送る必要がある。

警告: この方法で CSRF トークンを送った場合、<form:form method="GET">...</form:form>使用時と同様に、CSRF トークンが URL に現れるという問題がある。

以下に、CSRF トークンをクエリパラメータとして送る実装例を示す。

JSP の実装例

```
<form:form action="${pageContext.request.contextPath}/fileupload?${f:h(_csrf.parameterName)}=${f:h
    method="post" modelAttribute="fileUploadForm" enctype="multipart/form-data"> <!-- (1) -->
    <table>
        <tr>
            <td width="65%"><form:input type="file" path="uploadFile" /></td>
        </tr>
        <tr>
            <td><input type="submit" value="Upload" /></td>
        </tr>
    </table>
</form:form>
```

項番	説明
(1)	<form:form>タグの action 属性に、以下のクエリを付与する必要がある。 ?${f:h(_csrf.parameterName)}=${f:h(_csrf.token)} <form>タグを使用する場合も、同様の設定が必要である。

第 7 章

Appendix

7.1 チュートリアル (Todo アプリケーション REST 編)

7.1.1 はじめに

このチュートリアルで学ぶこと

- TERASOLUNA Global Framework 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

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。

ただし、REST Client として、Google Chrome の拡張機能である「Dev HTTP Client」を使用するため、Web Browser は Google Chrome を使用する必要がある。

種別	名前
OS	Windows7 64bit
JVM	Java 1.7
IDE	Spring Tool Suite Version: 3.4.0.RELEASE, Build Id: 201310051614 (以降「STS」と呼ぶ)
Build	Maven 3.0.4
Application Server	VMWare vFabric tc Server Developer Edition v2.9 (以降「ts Server」と呼ぶ)
Web Browser	Google Chrome 33.0.1750.154 m, Language is English(United States)
REST Client	Dev HTTP Client 0.6.9.16

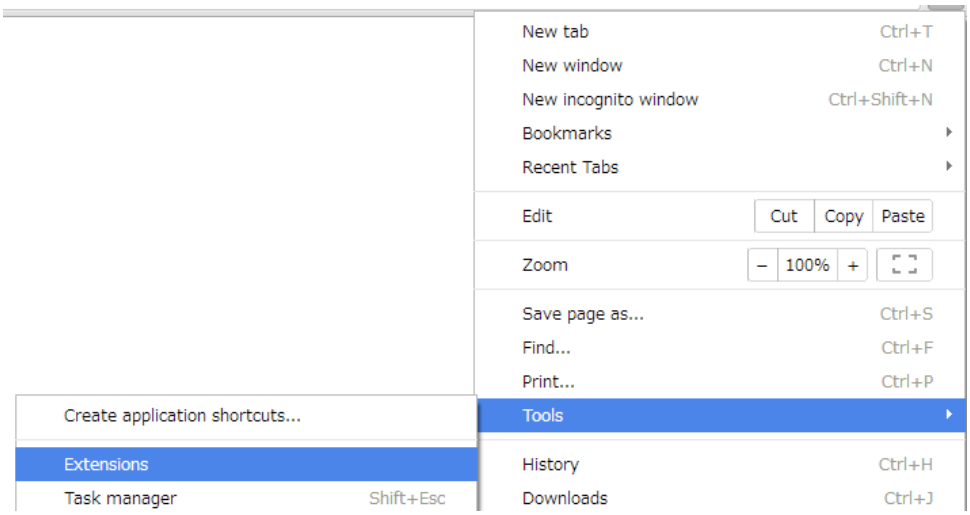
7.1.2 環境構築

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。

Dev HTTP Client のインストール

まず、REST クライアントして Chrome の拡張機能である「Dev HTTP Client」をインストールする。

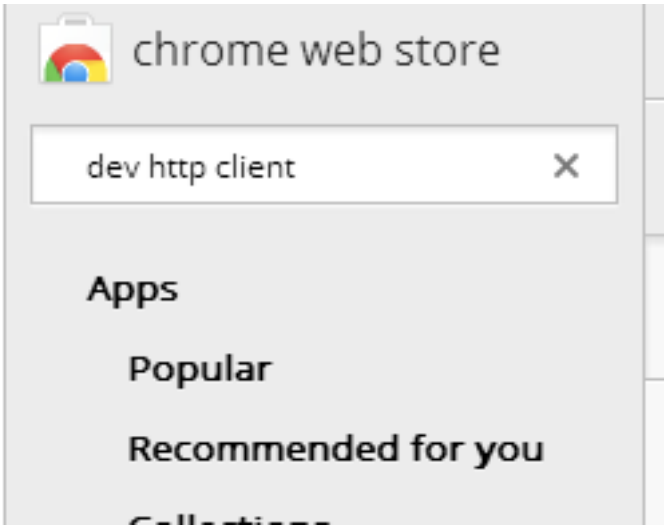
Chrome の「Tools」→「Extensions」を選択する。



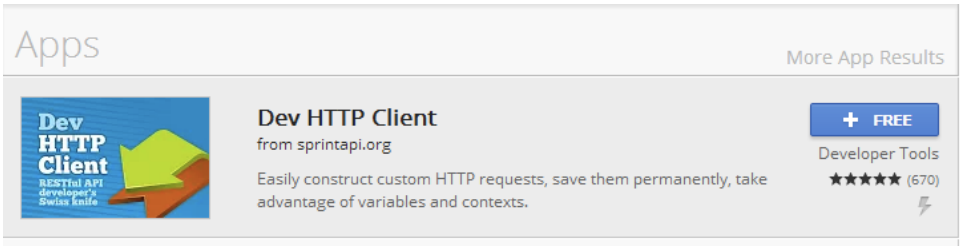
「Get more extensions」のリンクを押下する。



検索フォームに「dev http clinet」を入力して検索する。

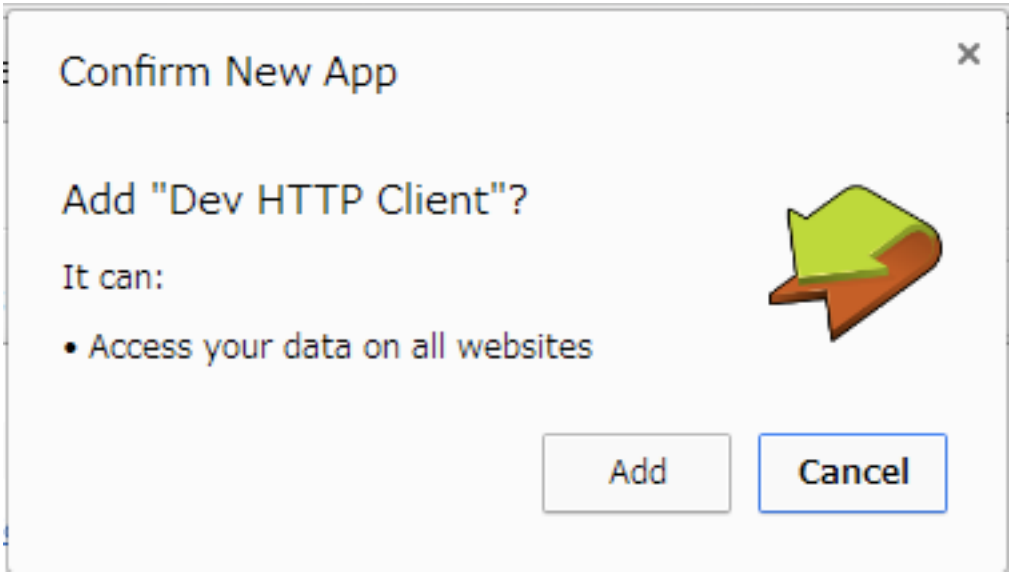


Dev HTTP Client の「+ FREE」ボタンを押下する。

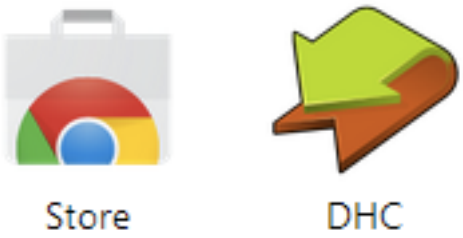


「Add」ボタンを押下する。

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(Dev HTTP Client) が追加されている。

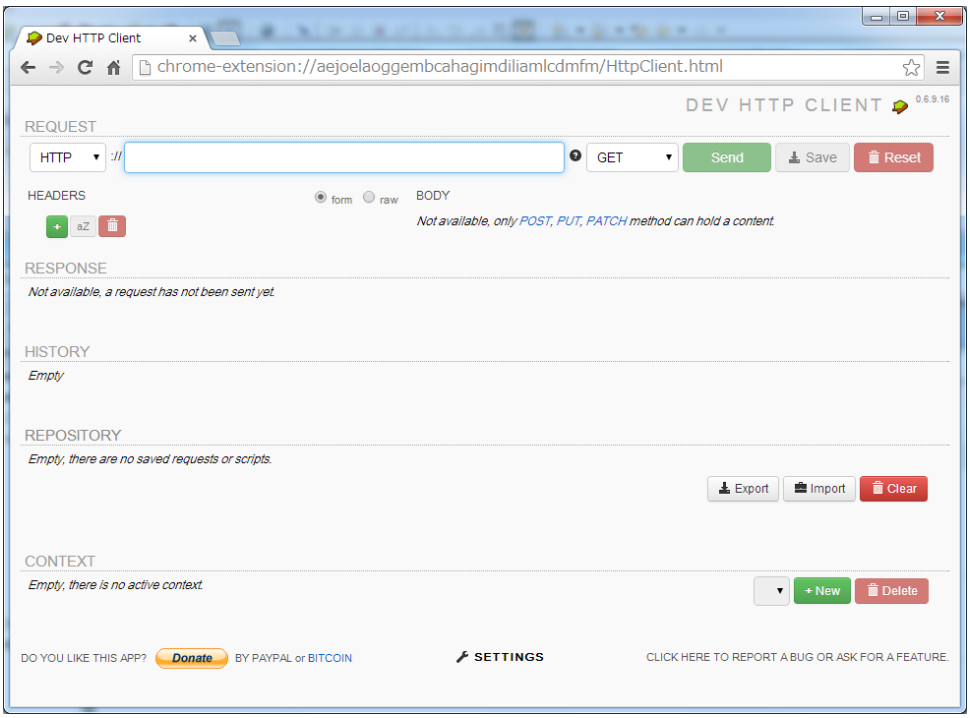


DHC(Dev HTTP Client) をクリックする。

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/HttpClient.html」を入力する事で開く事もできる。



プロジェクト作成

チュートリアル用のプロジェクトを作成する。

ドメイン層はチュートリアル (Todo アプリケーション) で作成した Todo アプリケーションのものを利用するが、今回はプロジェクトの雛形を Blank プロジェクトから生成する。

以下のコマンドを実行し、MyBatis2 版の Maven プロジェクトを作成する。

```
mvn archetype:generate -B^
 -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-rel
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-mybatis2-archetype^
 -DarchetypeVersion=1.0.4.RELEASE^
 -DgroupId=todo^
 -DartifactId=todo-api^
 -Dversion=1.0-SNAPSHOT
```

コンソール上に以下のようなログが表示されれば、ブランクプロジェクトの作成は成功となる。

```
C:\workspace>mvn archetype:generate -B^
More?  -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-g
More?  -DarchetypeGroupId=org.terasoluna.gfw.blank^
More?  -DarchetypeArtifactId=terasoluna-gfw-web-blank-mybatis2-archetype^
More?  -DarchetypeVersion=1.0.4.RELEASE^
```

```
More?  -DgroupId=todo^
More?  -DartifactId=todo-api^
More?  -Dversion=1.0-SNAPSHOT
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] Archetype repository missing. Using the one from [org.terasoluna.gfw.blank:terasoluna
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: terasoluna-gfw-web-bl
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: todo
[INFO] Parameter: artifactId, Value: todo-api
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: todo
[INFO] Parameter: packageInPathFormat, Value: todo
[INFO] Parameter: package, Value: todo
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: todo
[INFO] Parameter: artifactId, Value: todo-api
[INFO] project created from Archetype in dir: C:\workspace\todo-api
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.287s
[INFO] Finished at: Fri Aug 22 20:20:55 JST 2014
[INFO] Final Memory: 11M/231M
[INFO] ------------------------------------------------------------------------
C:\workspace>
```

プロジェクトのインポート

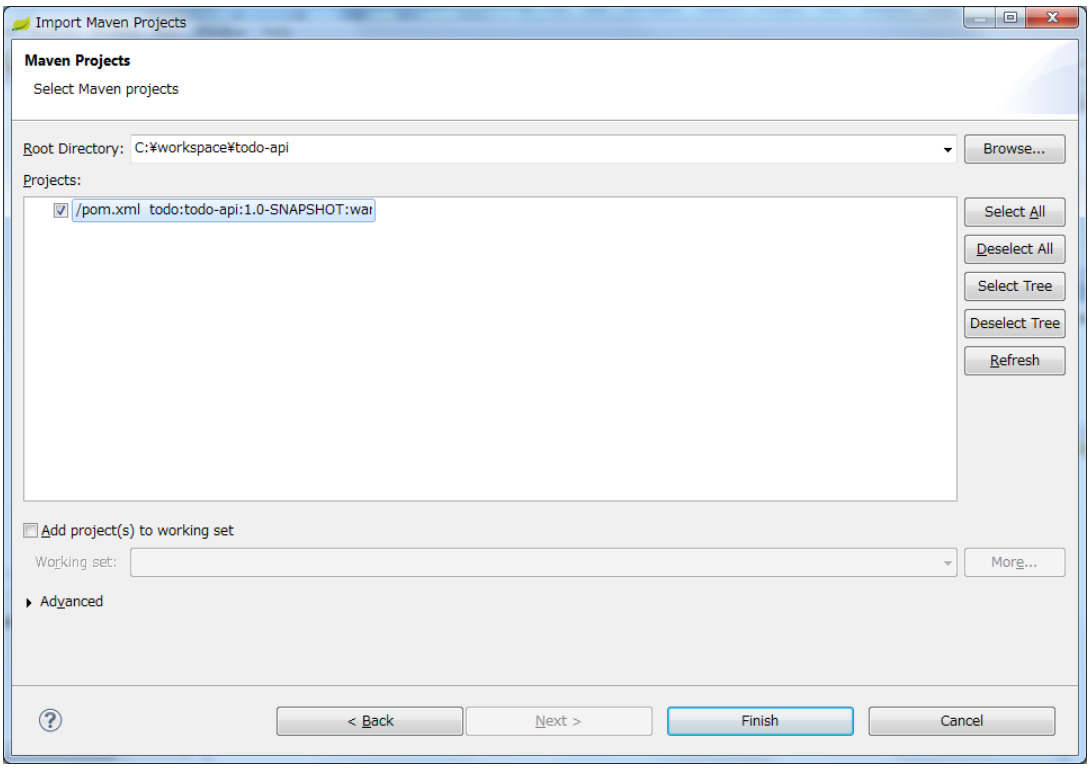
作成した Maven プロジェクトを STS の workspace にインポートする。

STS で「File」→「Import」→「Maven」→「Existing Maven Projects」を選択して「Next >」をクリックする。

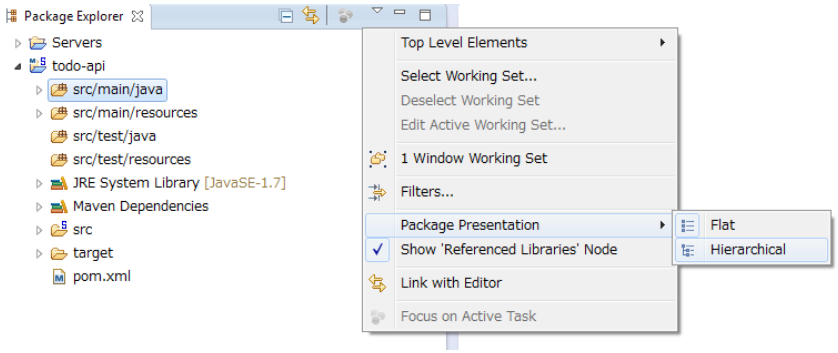
先ほど作成した「todo-api」フォルダを選択し、「Finish」をクリックする。

ちなみに: 「Pacakge Presentaion」は、以下のように「Hierarchical」に変更することを推奨する。

「Package Explorer」→「View Menu」→「Pacakge Presentaion」を選択して「Hierarchical」をクリック



する。



ドメイン層のファイルをコピー

チュートリアル (Todo アプリケーション) で作成したクラスのうち、

- `todo.domain.model.Todo`
- `todo.domain.repository.todo.TodoRepository`
- `todo.domain.repository.todo.TodoRepositoryImpl`(MyBatis2 版)
- `todo.domain.serivce.todo.TodoService`

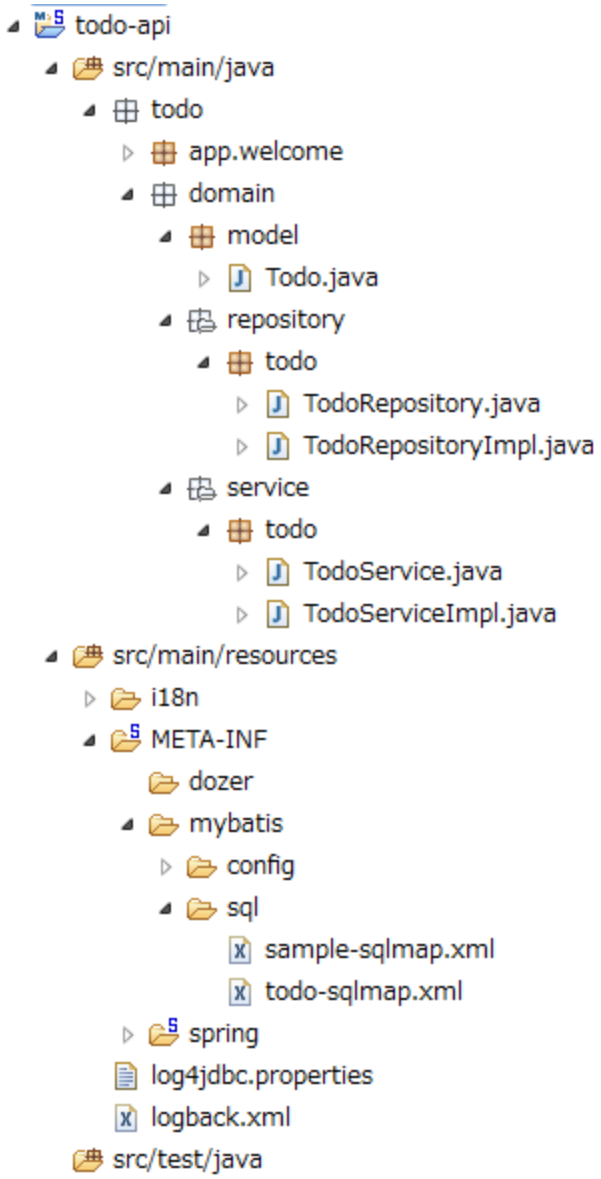
- `todo.domain.serivce.todo.TodoServiceImpl`(インフラストラクチャ層の変更反映版)

と設定ファイル

- `src/main/resources/META-INF/mybatis/sql/todo-sqlmap.xml`

を、STS にインポートしたプロジェクトにコピーする。

コピー後の STS の workspace は以下のようになる。



テーブル定義の追加

todo テーブルを生成するための DDL を追加する。

チュートリアル (Todo アプリケーション) で作成した、src/main/resources/META-INF/spring/todo-infra.prope の database.url に設定されているテーブル定義の設定内容を、今回作成した src/main/resources/META-INF/spring/todo-api-infra.properties の database.url にコピーする。

- src/main/resources/META-INF/spring/todo-api-infra.properties

  INIT パラメータにテーブルを作成する DDL を指定する事で、アプリケーションサーバ起動時に H2 のインメモリデータベースに todo テーブルが作成される。

```
database=H2
database.url=jdbc:h2:mem:todo-api;DB_CLOSE_DELAY=-1;INIT=create table if not exists todo(tod
database.username=sa
database.password=
database.driverClassName=org.h2.Driver
# connection pool
cp.maxActive=96
cp.maxIdle=16
cp.minIdle=0
cp.maxWait=60000
```

7.1.3 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。

下記に示す具体例では、Web アプリケーションのコンテキストパスは「<contextPath>」で表現している。
本チュートリアルで作成する Web アプリケーションのコンテキストパスは、「todo-api」となる。

GET Todos

- リクエスト

```
> GET /<contextPath>/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":fals
```

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 /<contextPath>/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
```

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 /<contextPath>/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
```

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 /<contextPath>/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,
```

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/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。

- 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":[{"c
```

- 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-4
```

- 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-4696-9b4
```

- 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。

追加で設定する箇所をハイライトしている。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSche
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
    version="3.0">
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-clas
    </listener>
    <listener>
        <listener-class>org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener</list
    </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>

    <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>
        </session-config>

    </web-app>
```

項番	説明
(1)	初期化パラメータ「contextConfigLocation」に、REST 用の SpringMVC 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

src/main/resources/META-INF/spring/spring-mvc.xml をコピーして、REST 用の SpringMVC 設定ファイルを作成する。

REST 用の SpringMVC 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:mvc="http://www.springfra
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframew
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
    http://www.springframework.org/schema/context http://www.springframework.org/schema/cont
        http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/

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

    <!-- (1) -->
    <context:component-scan base-package="todo.api"/>

    <mvc:annotation-driven>
        <mvc:message-converters>
            <!-- (2) -->
            <bean
                    class="org.springframework.http.converter.json.MappingJacksonHttpMessage
                <property name="objectMapper" ref="objectMapper"/>
```

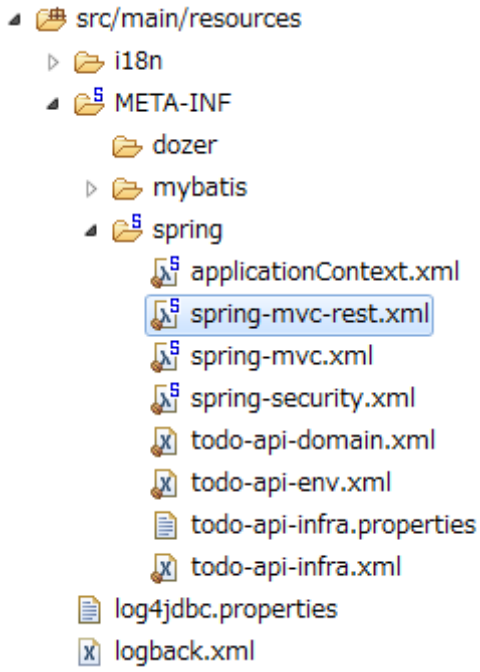
```
                </bean>
            </mvc:message-converters>
            <mvc:argument-resolvers>
                <bean class="org.springframework.data.web.PageableHandlerMethodArgumentResolver"
            </mvc:argument-resolvers>
        </mvc:annotation-driven>

        <bean id="objectMapper" class="org.codehaus.jackson.map.ObjectMapper">
            <property name="dateFormat">
                <!-- (3) -->
                <bean class="org.codehaus.jackson.map.util.StdDateFormat"/>
            </property>
        </bean>

        <mvc:interceptors>
            <mvc:interceptor>
                <mvc:mapping path="/**"/>
                <mvc:exclude-mapping path="/resources/**"/>
                <mvc:exclude-mapping path="/**/*.html"/>
                <bean
                        class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor"/>
            </mvc:interceptor>
            <!-- (4) -->
            <!--  REMOVE THIS LINE IF YOU USE JPA
            <mvc:interceptor>
                <mvc:mapping path="/**"/>
                <mvc:exclude-mapping path="/resources/**"/>
                <mvc:exclude-mapping path="/**/*.html"/>
                <bean
                        class="org.springframework.orm.jpa.support.OpenEntityManagerInViewInterc
            </mvc:interceptor>
                  REMOVE THIS LINE IF YOU USE JPA  -->
        </mvc:interceptors>

        <!-- 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.r
        </aop:config>
    </beans>
```

項番	説明
(1)	REST API 用のパッケージ配下に対してコンポーネントスキャンの設定をする。 本チュートリアルでは、REST API 用のパッケージを `todo.api` にしている。 画面遷移用の Contoller は、`app` パッケージ配下に格納していたが、REST API 用の Contoller は、`api` パッケージ配下に格納する事を推奨する。
(2)	`<mvc:message-converters>`に、Contrtoller の引数及び返り値で扱う JavaBean をシリアライズ/デシリアライズするためのクラスを設定する。 複数設定する事ができるが、今回のチュートリアルでは JSON しか使用しないため、`MappingJacksonHttpMessageConverter` のみ指定している。 `MappingJacksonHttpMessageConverter` の `objectMapper` プロパティには、Jackson より提供されている「JSON <-> JavaBean」の変換を行うためのコンポーネント「`ObjectMapper`」を指定する。 本チュートリアルでは、日時型の変換をカスタマイズした `ObjectMapper` を指定している。
(3)	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、`ObjectMapper` の `dateFormat` プロパティに `org.codehaus.jackson.map.util.StdDateFormat` を設定する事で実現する事ができる。
(4)	JPA を使用する場合は `OpenEntityManagerInViewInterceptor` を使用するが、今回のチュートリアルでは MyBatis2 を使用するので、コメントアウトのままにするか、この `<mvc:interceptor>`タグを削除する。

REST API 用の Spring Security の定義追加

本チュートリアルで作成する REST API では、CSRF 対策を無効にする。

REST API を使って構築する Web アプリケーションでも、CSRF 対策は必要である。ただし、本チュートリアルの目的として CSRF 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。



CSRF 対策を無効化すると、セッションを使用する必要がなくなる。

そのため、本チュートリアルではセッションを使用しないアーキテクチャ（ステートレスなアーキテクチャ）を採用する。

以下の設定を追加する事で、CSRF 対策の無効化及びセッションを使用しないようにする事ができる。

- 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.springframew
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/security http://www.springfram
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/be
        http://www.springframework.org/schema/context http://www.springframework.org/schema/

    <sec:http pattern="/resources/**" security="none"/>

    <!-- (1) -->
    <sec:http pattern="/api/v1/**" auto-config="true" use-expressions="true" create-session=
        <!--<sec:custom-filter ref="csrfFilter" before="LOGOUT_FILTER"/>--> <!-- (2) -->
        <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
        <!--<sec:session-management session-authentication-strategy-ref="sessionAuthenticati
    </sec:http>

    <sec:http auto-config="true" use-expressions="true">
        <sec:custom-filter ref="csrfFilter" before="LOGOUT_FILTER"/>
```

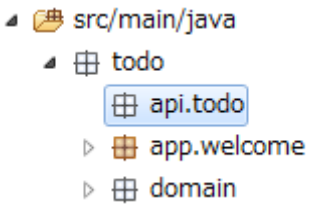
```
            <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/>
            <sec:session-management session-authentication-strategy-ref="sessionAuthenticationSt
        </sec:http>
        <sec:authentication-manager></sec:authentication-manager>

        <!-- CSRF Protection -->
        <bean id="csrfTokenRepository"
            class="org.springframework.security.web.csrf.HttpSessionCsrfTokenRepository" />

        <bean id="csrfFilter" class="org.springframework.security.web.csrf.CsrfFilter">
            <constructor-arg index="0" ref="csrfTokenRepository" />
            <property name="accessDeniedHandler">
                <bean
                    class="org.springframework.security.web.access.AccessDeniedHandlerImpl">
                    <property name="errorPage" value="/WEB-INF/views/common/error/csrfTokenError
                </bean>
            </property>
        </bean>

        <bean id="sessionAuthenticationStrategy"
            class="org.springframework.security.web.authentication.session.CompositeSessionAuthe
            <constructor-arg index="0">
                <list>
                    <bean
                        class="org.springframework.security.web.authentication.session.SessionFi
                    <bean
                        class="org.springframework.security.web.csrf.CsrfAuthenticationStrategy"
                        <constructor-arg index="0"
                            ref="csrfTokenRepository" />
                    </bean>
                </list>
            </constructor-arg>
        </bean>

        <!-- Put UserID into MDC -->
        <bean id="userIdMDCPutFilter" class="org.terasoluna.gfw.security.web.logging.UserIdMDCPu
            <property name="removeValue" value="true" />
        </bean>

    </beans>
```

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。 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。 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。
(2)	CSRF 対策を無効化するために、CSRF 対策用の Servlet Filter の定義をコメントアウトしている。
(3)	セッションを使用しないアーキテクチャを採用するために、セッション管理に関係するコンポーネントへの参照定義をコメントアウトしている。

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、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

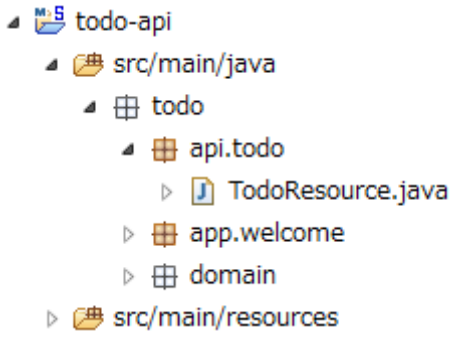
Todo リソースを表現する TodoResource クラスを作成する。

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。

- TodoResource.java

```
package todo.api.todo;

import java.util.Date;

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

public class TodoResource {

    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。

- TodoRestController.java

```
package todo.api.todo;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("todos") // (1)
public class TodoRestController {

}
```

項番	説明
(1)	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。

```
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.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;
```

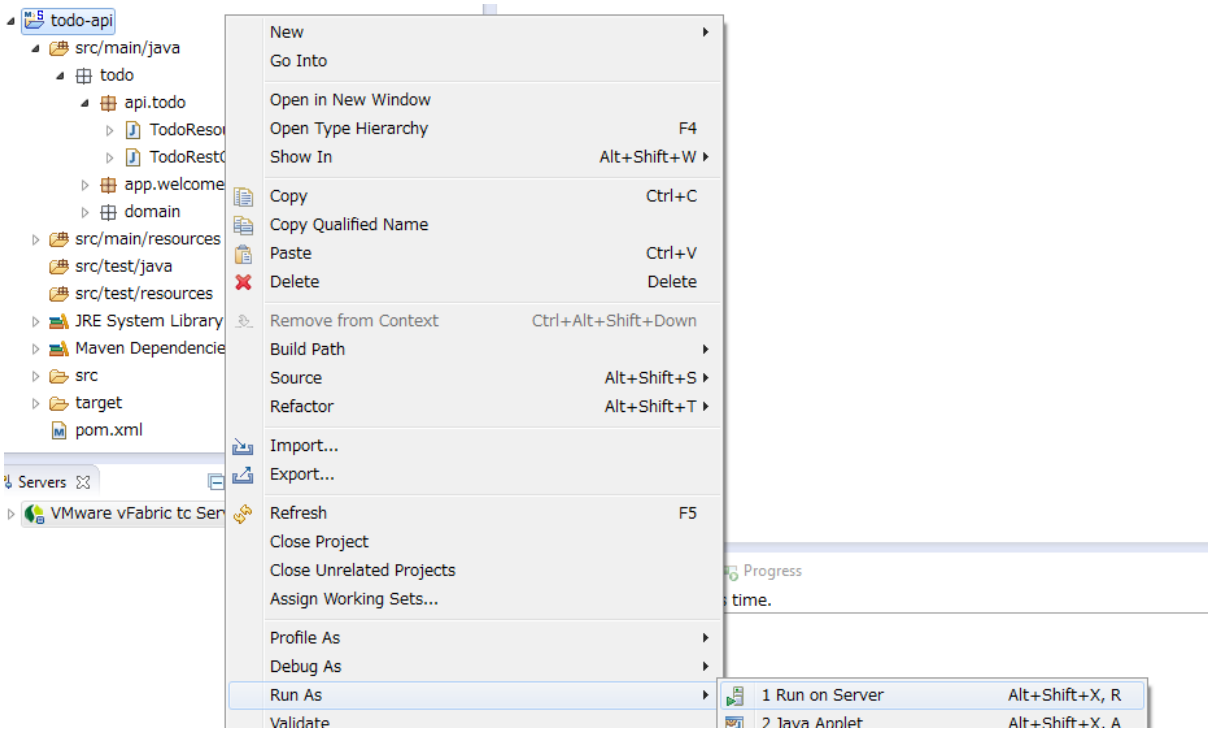
```
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Controller
@RequestMapping("todos")
public class TodoRestController {
    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    @RequestMapping(method = RequestMethod.GET) // (1)
    @ResponseBody // (2)
    @ResponseStatus(HttpStatus.OK) // (3)
    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)); // (4)
        }
        return todoResources; // (5)
    }

}
```

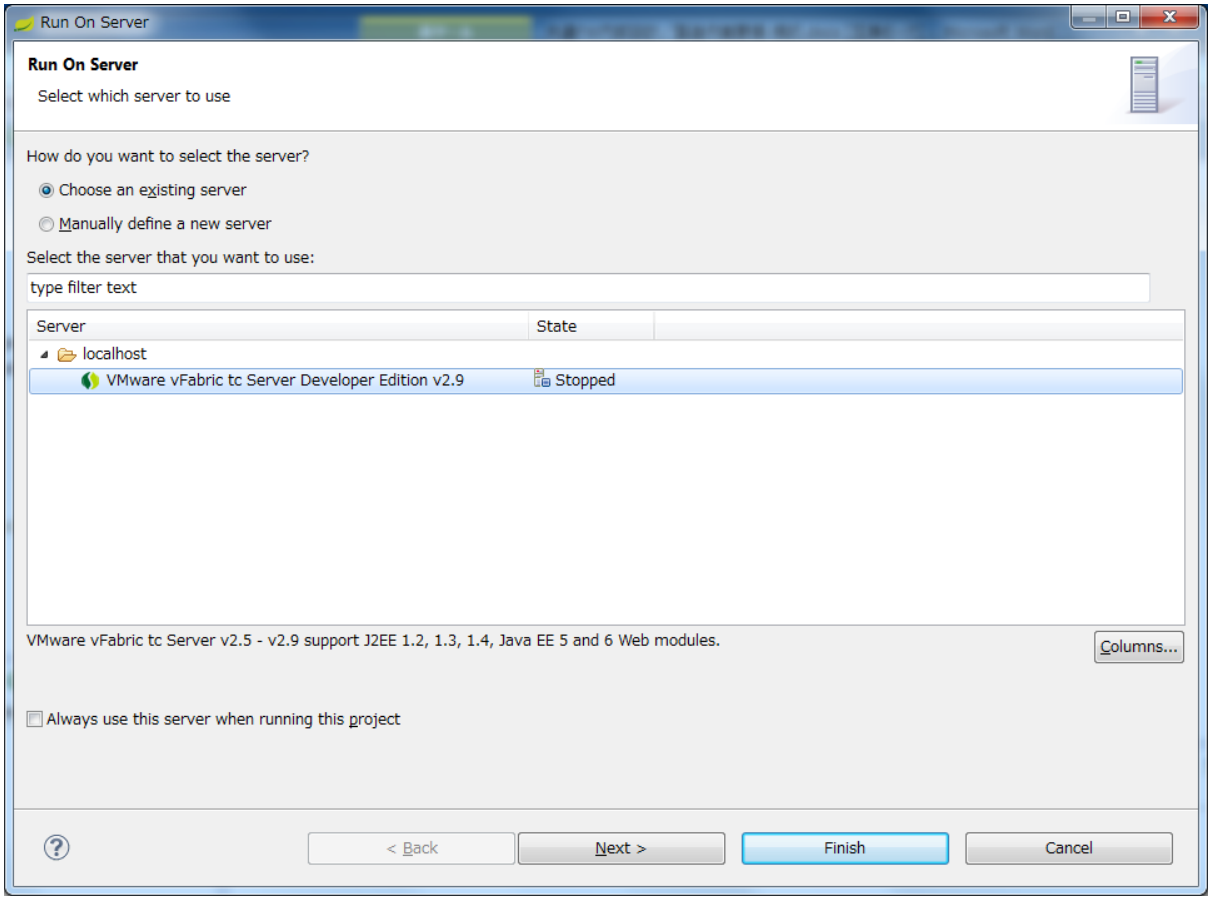
項番	説明
(1)	メソッドが GET のリクエストを処理するために、`method` 属性に `RequestMethod.GET` を設定する。
(2)	メソッドの返り値として返却したオブジェクトを、レスポンス BODY のメッセージとしてクライアントに返答することを示す`@ResponseBody` アノテーションを付与する。 本チュートリアルでは、JSON 形式のデータにシリアライズする。
(3)	応答する HTTP ステータスコードを`@ResponseStatus` アノテーションに指定する。 HTTP ステータスとして、”200 OK” を設定するため、value 属性には `HttpStatus.OK` を設定する。
(4)	`TodoService.findAll` で取得した `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。 `Todo` と `TodoResource` の変換処理は、Dozer の `org.dozer.Mapper` を使うと便利である。
(5)	`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 `MappingJacksonHttpMessageConverter` により、`[{"todoId" : "xxx", "todoTitle": "yyy", ...}, {"todoId" : "xxx", "todoTitle": "yyy", ...}, ...]` 形式の JSON にシリアライズされる。

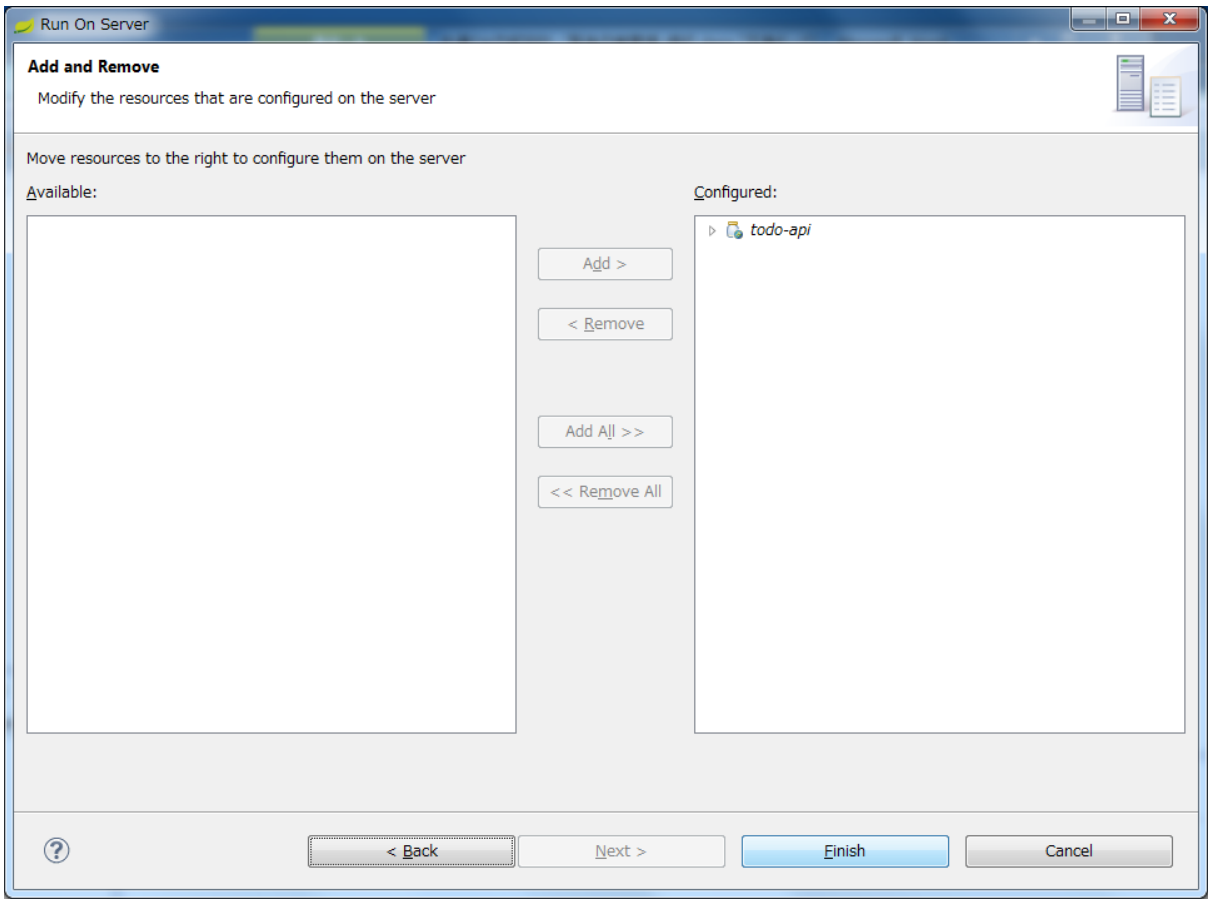
tc Server を起動し、実装した API の動作確認を行う。
プロジェクト名を右クリックして、「Run As」→「Run on Server」を選択する。

「VMWare vFabric tc Server Developer Edition v2.9」を選択して、「Next >」をクリックする。

「Configured」の欄に「todo-api」が含まれていることを確認して、「Finish」をクリックする。


todo-api
src/main/java
todo
api.todo
app.welcome
domain
src/main/resources
src/test/java
src/test/resources
JRE System Library
Maven Dependencie
src
target
pom.xml
Servers
VMware vFabric tc Ser
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Profile As
Debug As
Run As
Validate
Progress
time.
1 Run on Server Alt+Shift+X, R
2 Java Applet Alt+Shift+X, A


Run On Server
Run On Server
Select which server to use
How do you want to select the server?
Choose an existing server
Manually define a new server
Select the server that you want to use:
type filter text
Server
State
localhost
VMware vFabric tc Server Developer Edition v2.9
Stopped
VMware vFabric tc Server v2.5 - v2.9 support J2EE 1.2, 1.3, 1.4, Java EE 5 and 6 Web modules.
Columns...
Always use this server when running this project
< Back
Next >
Finish
Cancel

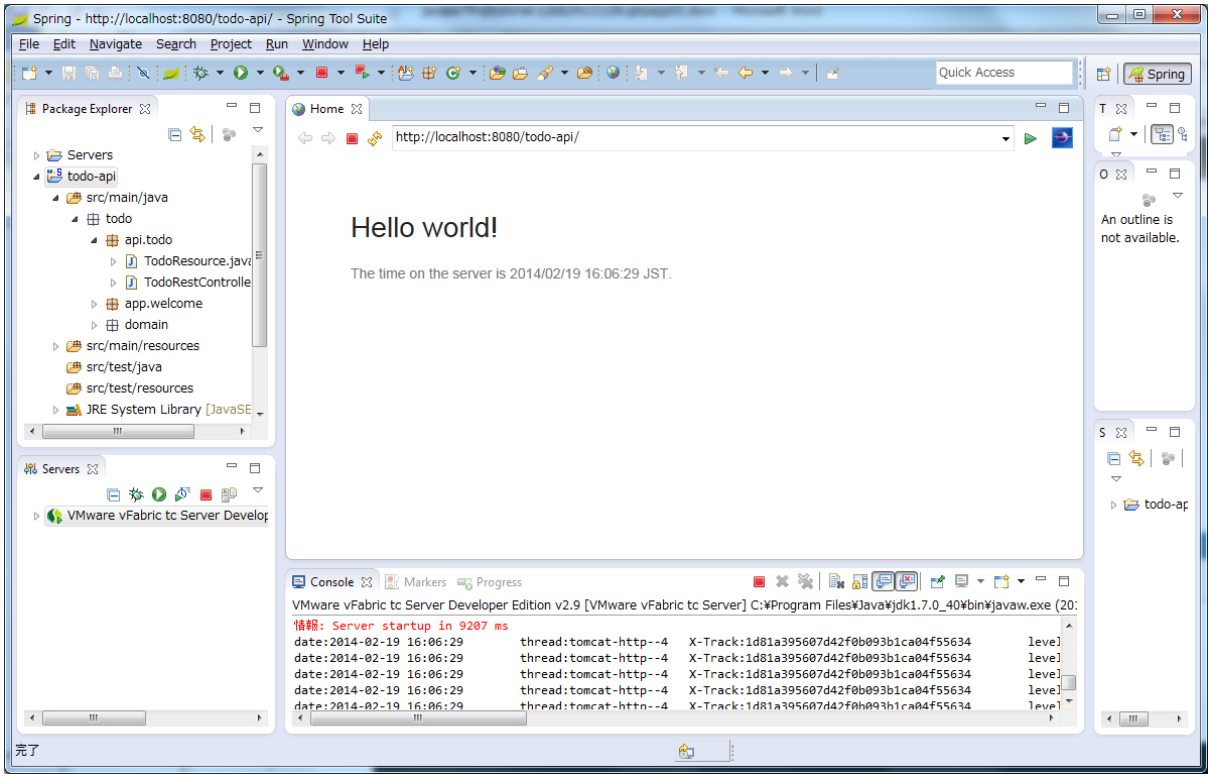


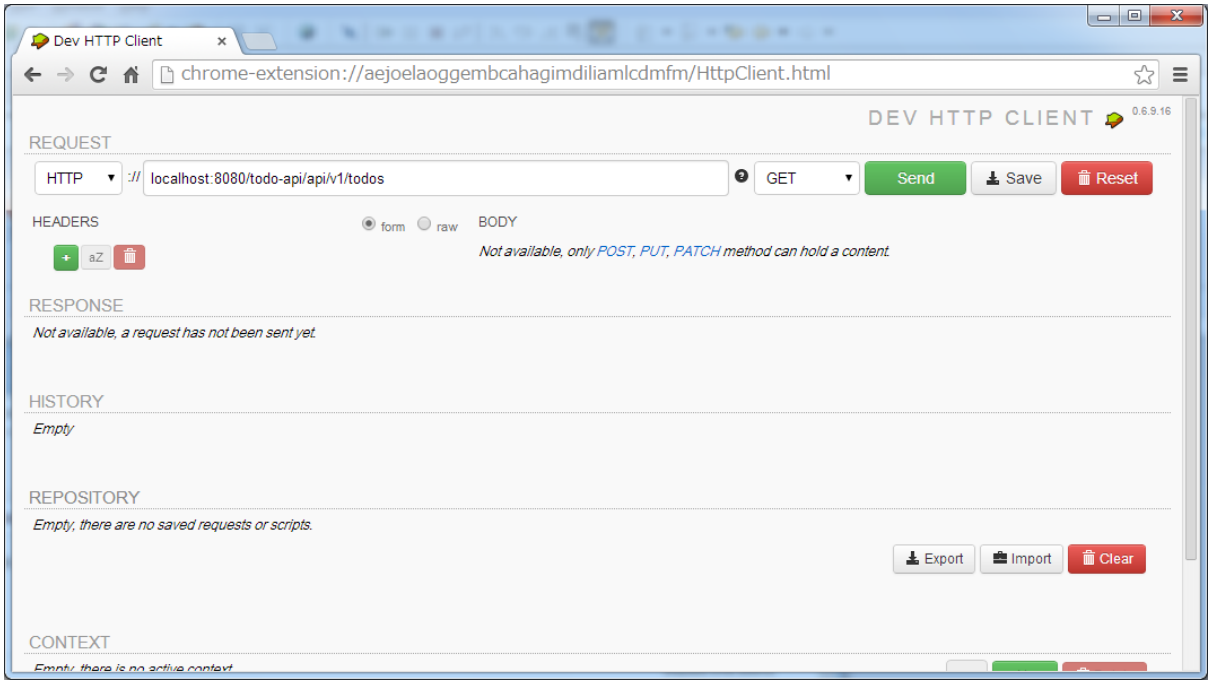
正常に起動して"localhost:8080/todo-api/"にアクセスすると、以下のようにデフォルトの welcome ページが表示される。

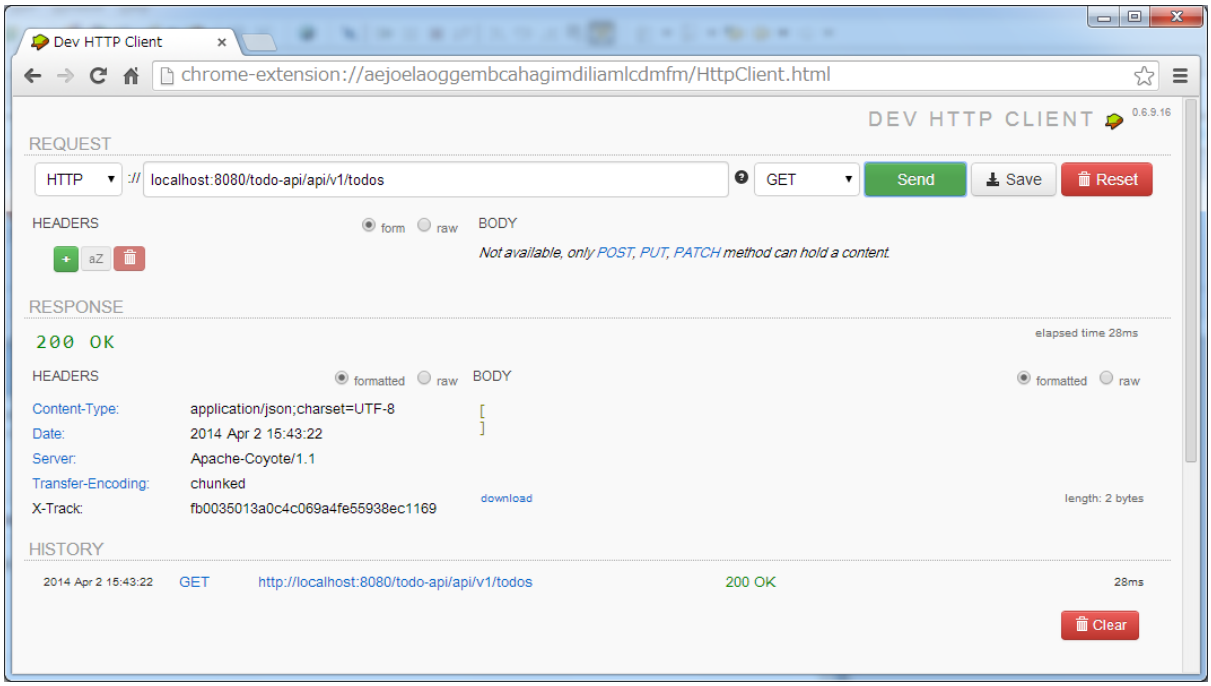
次に、REST API にアクセスする。Dev HTTP Client を開いて URL に"localhost:8080/todo-api/api/v1/todos"を入力し、メソッドに GET を指定して、”Send”ボタンをクリックする。

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。

> Spring Security の設定を、セッションを使用しない設定に変更しているため、「RESPONSE」の「HEADERS」に"Set-Cookie: JSESSIONID=xxxx"がないという点にも着目してほしい。


Spring - http://localhost:8080/todo-api/ - Spring Tool Suite
File Edit Navigate Search Project Run Window Help
Quick Access
Spring
Package Explorer
Servers
todo-api
src/main/java
todo
api.todo
TodoResource.java
TodoRestControlle
app.welcome
domain
src/main/resources
src/test/java
src/test/resources
JRE System Library [JavaSE
Home
http://localhost:8080/todo-api/
Hello world!
The time on the server is 2014/02/19 16:06:29 JST.
An outline is not available.
Servers
VMware vFabric tc Server Develop
Console
Markers
Progress
VMware vFabric tc Server Developer Edition v2.9 [VMware vFabric tc Server] C:¥Program Files¥Java¥jdk1.7.0_40¥bin¥javaw.exe (20
情報: Server startup in 9207 ms
date:2014-02-19 16:06:29 thread:tomcat-http--4 X-Track:1d81a395607d42f0b093b1ca04f55634 level
date:2014-02-19 16:06:29 thread:tomcat-http--4 X-Track:1d81a395607d42f0b093b1ca04f55634 level
date:2014-02-19 16:06:29 thread:tomcat-http--4 X-Track:1d81a395607d42f0b093b1ca04f55634 level
date:2014-02-19 16:06:29 thread:tomcat-http--4 X-Track:1d81a395607d42f0b093b1ca04f55634 level
todo-ap
完了


Dev HTTP Client
chrome-extension://aejoelaoggembcahagimdiliamlcdmfm/HttpClient.html
DEV HTTP CLIENT 0.6.9.16
REQUEST
HTTP
://
localhost:8080/todo-api/api/v1/todos
GET
Send
Save
Reset
HEADERS
form
raw
BODY
Not available, only POST, PUT, PATCH method hold a content.
RESPONSE
Not available, a request has not been sent yet.
HISTORY
Empty
REPOSITORY
Empty, there are no saved requests or scripts.
Export
Import
Clear
CONTEXT



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。

```
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.stereotype.Controller;
import org.springframework.validation.annotation.Validated;
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.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Controller
@RequestMapping("todos")
public class TodoRestController {
```

```
        @Inject
        TodoService todoService;
        @Inject
        Mapper beanMapper;

        @RequestMapping(method = RequestMethod.GET)
        @ResponseBody
        @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;
        }

        @RequestMapping(method = RequestMethod.POST) // (1)
        @ResponseBody
        @ResponseStatus(HttpStatus.CREATED) // (2)
        public TodoResource postTodos(@RequestBody @Validated TodoResource todoResource) { // (3
            Todo createdTodo = todoService.create(beanMapper.map(todoResource, Todo.class)); //
            TodoResource createdTodoResponse = beanMapper.map(createdTodo, TodoResource.class);
            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、`method` 属性に `RequestMethod.POST` 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.create` を実行し、Todo リソースを新規作成する。
(5)	`TodoService.create` 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 `MappingJacksonHttpMessageConverter` により、`{"todoId" : "xxx", "todoTitle": "yyy", ...}`形式の JSON にシリアライズされる。

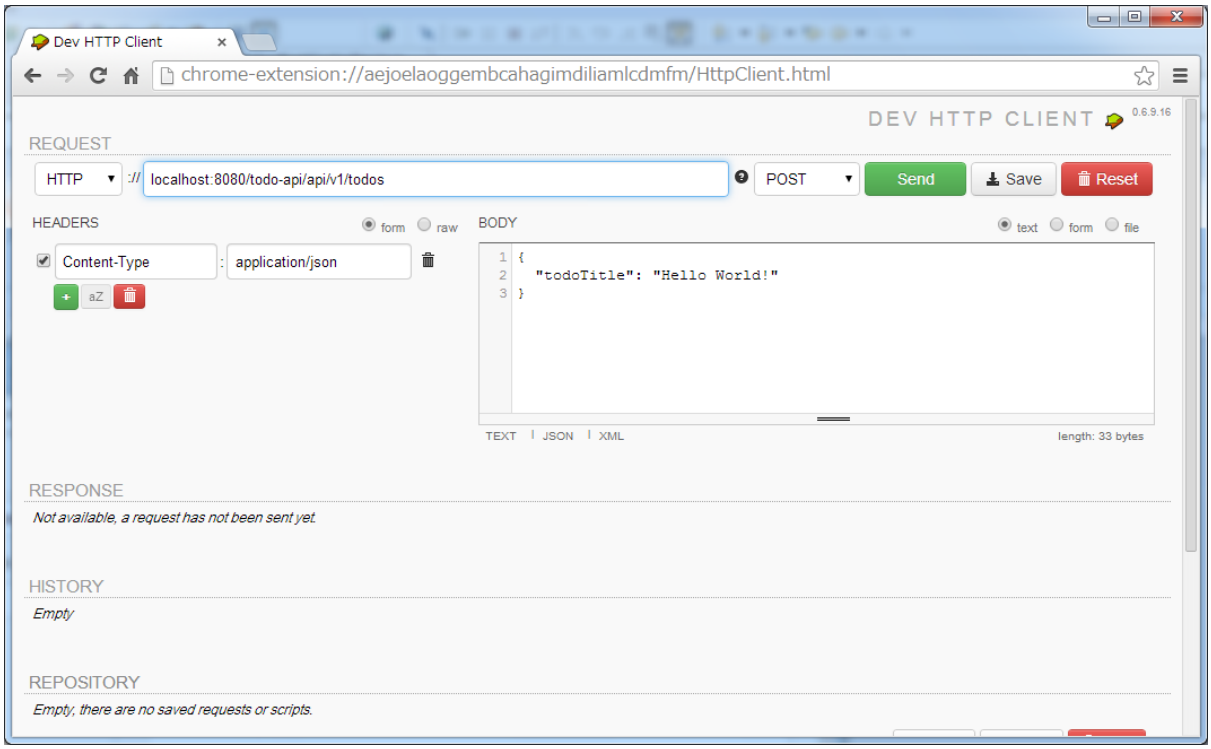
HTTP Dev Client を使用して、実装した API の動作確認を行う。

Dev HTTP Client を開いて URL に`"localhost:8080/todo-api/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。

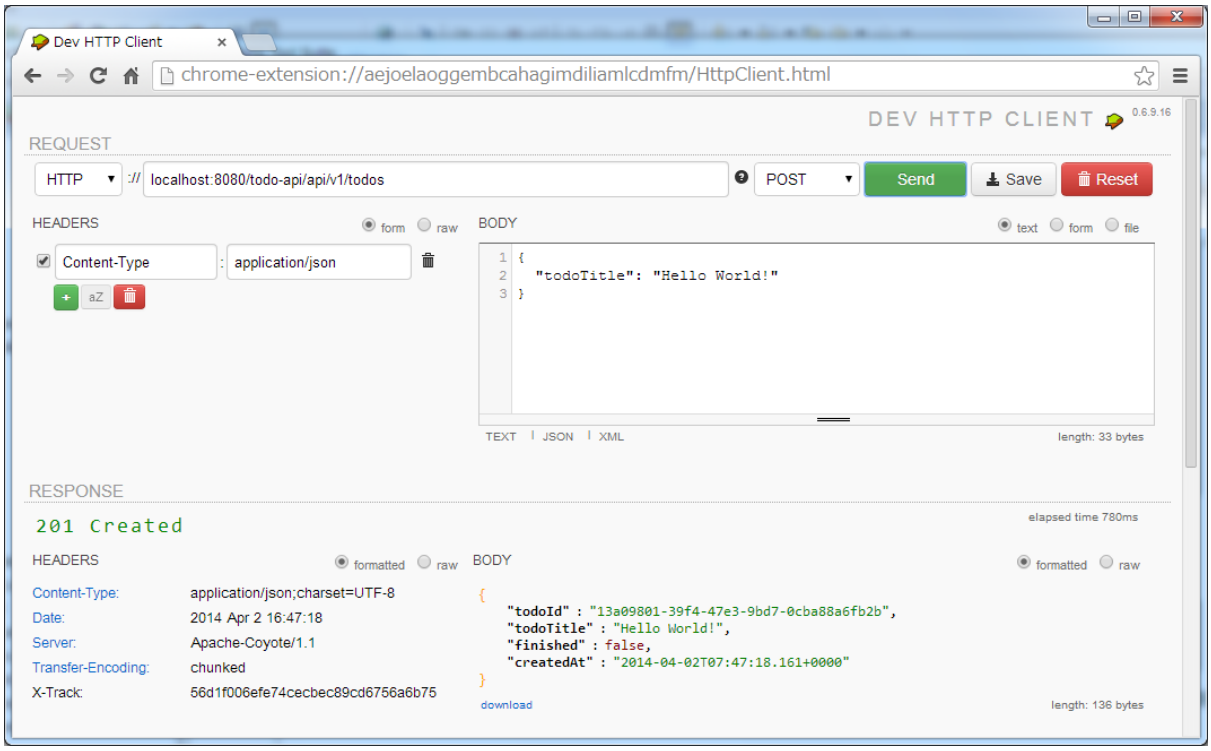
この状態で再び GET Todos を実行すると、作成した Todo リソースを含む配列が返却される。

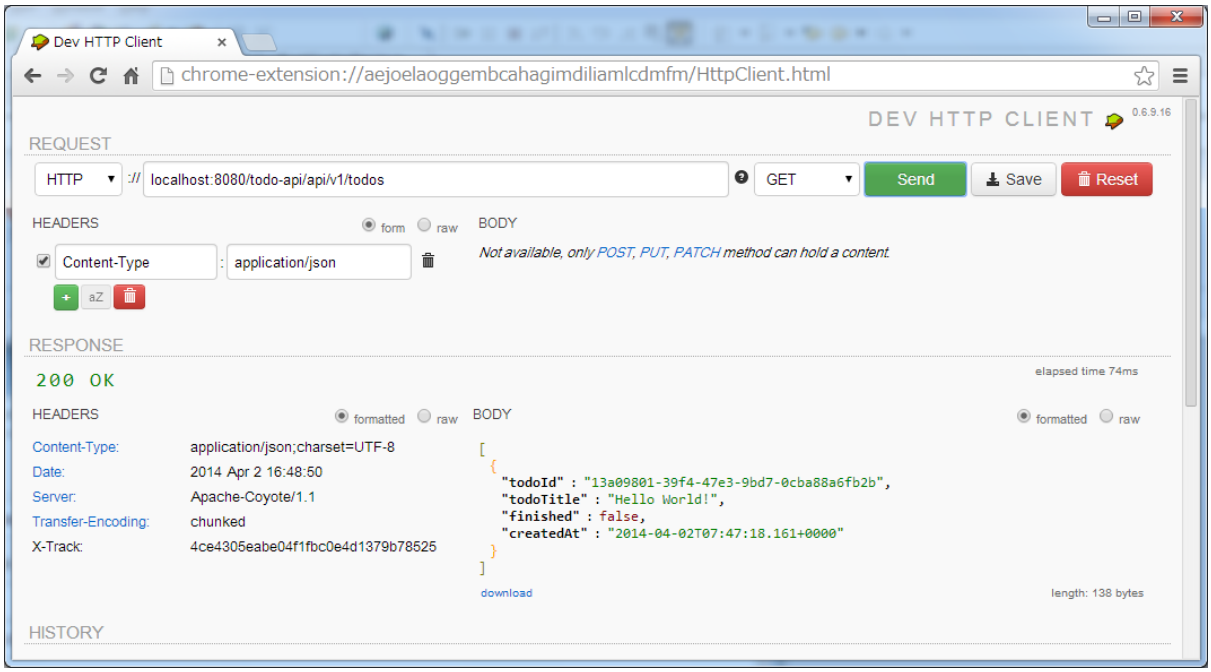
GET Todo の実装

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追加する。

- TodoService.java

  findOne メソッドの定義を追加する。


Dev HTTP Client
chrome-extension://aejoelaoggembcahagimdiliamlcdmfm/HttpClient.html
DEV HTTP CLIENT
0.6.9.16
REQUEST
HTTP
localhost:8080/todo-api/api/v1/todos
POST
Send
Save
Reset
HEADERS
form
raw
BODY
text
form
file
Content-Type
application/json
{
"todoTitle": "Hello World!"
}
TEXT
JSON
XML
length: 33 bytes
RESPONSE
201 Created
elapsed time 780ms
HEADERS
formatted
raw
BODY
formatted
raw
Content-Type:
application/json;charset=UTF-8
Date:
2014 Apr 2 16:47:18
Server:
Apache-Coyote/1.1
Transfer-Encoding:
chunked
X-Track:
56d1f006efe74cecbec89cd6756a6b75
{
"todoId" : "13a09801-39f4-47e3-9bd7-0cba88a6fb2b",
"todoTitle" : "Hello World!",
"finished" : false,
"createdAt" : "2014-04-02T07:47:18.161+0000"
}
download
length: 136 bytes


Dev HTTP Client
chrome-extension://aejoelaoggembcahagimdiliamlcdmfm/HttpClient.html
DEV HTTP CLIENT
0.6.9.16
REQUEST
HTTP
localhost:8080/todo-api/api/v1/todos
GET
Send
Save
Reset
HEADERS
form
raw
BODY
Content-Type
application/json
Not available, only POST, PUT, PATCH method can hold a content.
RESPONSE
200 OK
elapsed time 74ms
HEADERS
formatted
raw
BODY
formatted
raw
Content-Type:
application/json;charset=UTF-8
Date:
2014 Apr 2 16:48:50
Server:
Apache-Coyote/1.1
Transfer-Encoding:
chunked
X-Track:
4ce4305eabe04f1fbc0e4d1379b78525
[
{
"todoId" : "13a09801-39f4-47e3-9bd7-0cba88a6fb2b",
"todoTitle" : "Hello World!",
"finished" : false,
"createdAt" : "2014-04-02T07:47:18.161+0000"
}
]
download
length: 138 bytes
HISTORY

```
package todo.domain.service.todo;

import java.util.Collection;

import 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);
}
```

- TodoServiceImpl.java

  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する。

```
package 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 todo.domain.model.Todo;
import todo.domain.repository.todo.TodoRepository;

@Service
@Transactional
public class TodoServiceImpl implements TodoService {
    @Inject
    TodoRepository todoRepository;

    private static final long MAX_UNFINISHED_COUNT = 5;

    @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.save(todo);

            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.save(todo);
            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。

```
    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.stereotype.Controller;
    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.ResponseBody;
    import org.springframework.web.bind.annotation.ResponseStatus;

    import todo.domain.model.Todo;
    import todo.domain.service.todo.TodoService;

    @Controller
    @RequestMapping("todos")
    public class TodoRestController {
        @Inject
        TodoService todoService;
        @Inject
        Mapper beanMapper;

        @RequestMapping(method = RequestMethod.GET)
        @ResponseBody
        @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;
        }

        @RequestMapping(method = RequestMethod.POST)
        @ResponseBody
        @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;
        }

        @RequestMapping(value="{todoId}", method = RequestMethod.GET) // (1)
        @ResponseBody
        @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;
        }

    }
```

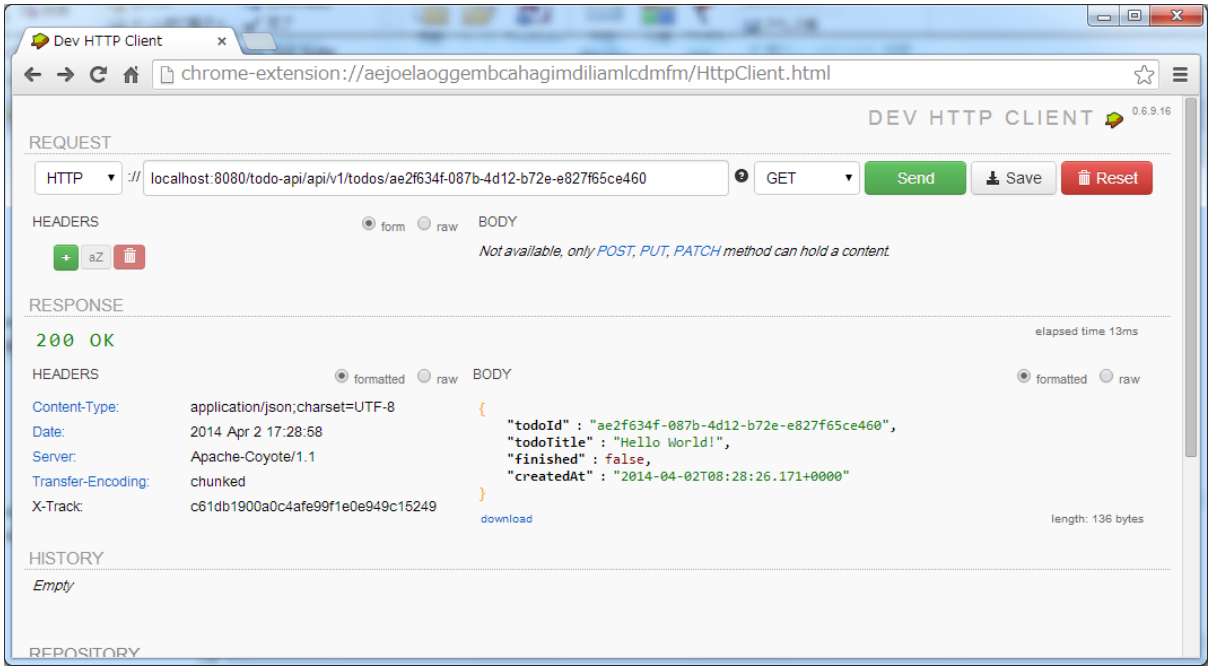
項番	説明
(1)	パスから todoId を取得するために、@RequestMapping アノテーションの value 属性にパス変数を指定する。 メソッドが GET のリクエストを処理するために、method 属性に RequestMethod.GET を設定する。
(2)	@PathVariable アノテーションの value 属性に、todoId を取得するためのパス変数名を指定する。
(3)	パス変数から取得した todoId を使用して、Todo リソースを一件を取得する。

HTTP Dev Client を使用して、実装した API の動作確認を行う。

Dev HTTP Client を開いて URL に"localhost:8080/todo-api/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。

```
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.stereotype.Controller;
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.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Controller
@RequestMapping("todos")
public class TodoRestController {
    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    @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;
    }

    @RequestMapping(method = RequestMethod.POST)
    @ResponseBody
    @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;
    }

    @RequestMapping(value="{todoId}", method = RequestMethod.GET)
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public TodoResource getTodo(@PathVariable("todoId") String todoId) {
        Todo todo = todoService.findOne(todoId);
        TodoResource todoResource = beanMapper.map(todo, TodoResource.class);
        return todoResource;
    }

    @RequestMapping(value="{todoId}", method = RequestMethod.PUT) // (1)
    @ResponseBody
    @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)	パスから `todoId` を取得するために、`@RequestMapping` アノテーションの `value` 属性にパス変数を指定する。 メソッドが GET のリクエストを処理するために、`method` 属性に `RequestMethod.GET` を設定する。
(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。

HTTP Dev Client を使用して、実装した API の動作確認を行う。

Dev HTTP Client を開いて URL に`"localhost:8080/todo-api/api/v1/todos/{todoId}"`を入力し、メソッドに PUT を指定する。

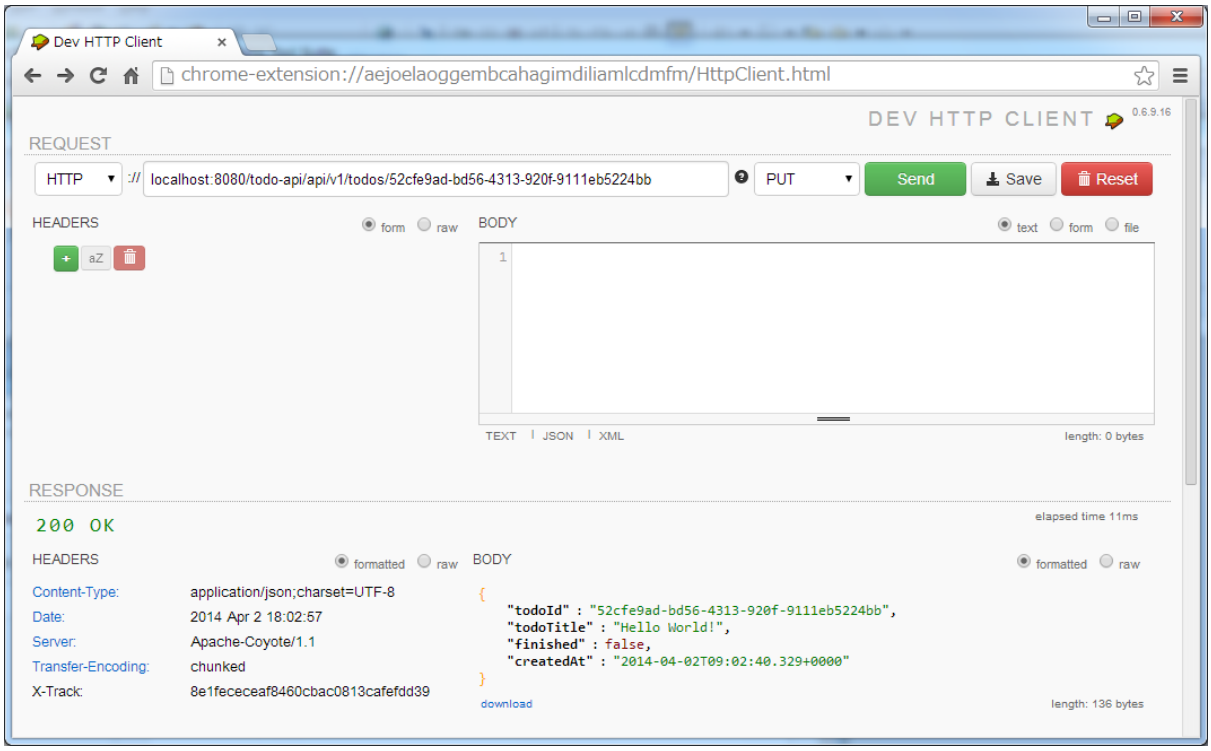
`{todoId}`の部分は実際の ID を入れる必要があるので、POST Todos または GET Todos を実行して Response 中の `todoId` をコピーして貼り付けてから、”Send” ボタンをクリックする。

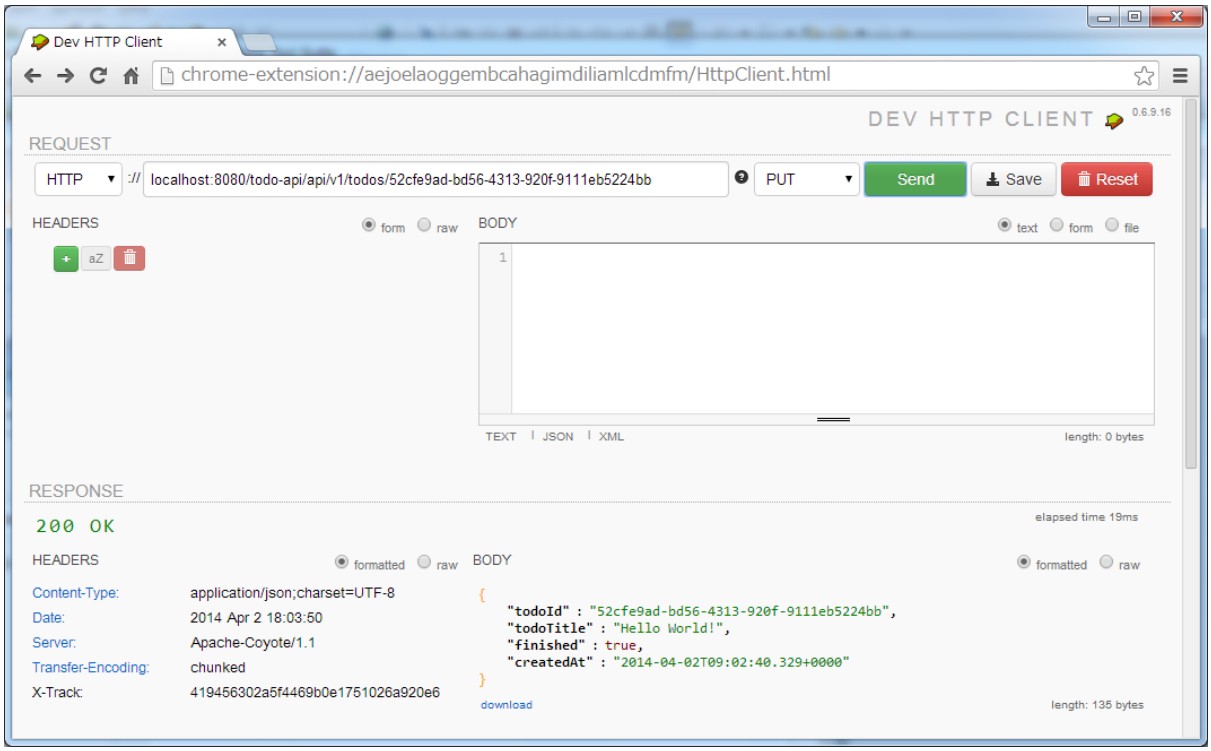
“200 OK” の HTTP ステータスが返却され、「RESPONSE」の「Body」に更新された Todo リソースの JSON が表示される。

`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。


Dev HTTP Client
chrome-extension://aejoelaoggembcahagimdiliamlcdmfm/HttpClient.html
DEV HTTP CLIENT
0.6.9.16
REQUEST
HTTP
localhost:8080/todo-api/api/v1/todos/52cfe9ad-bd56-4313-920f-9111eb5224bb
PUT
Send
Save
Reset
HEADERS
form
raw
BODY
text
form
file
TEXT
JSON
XML
length: 0 bytes
RESPONSE
200 OK
elapsed time 11ms
HEADERS
formatted
raw
BODY
formatted
raw
Content-Type: application/json;charset=UTF-8
Date: 2014 Apr 2 18:02:57
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked
X-Track: 8e1fececeaf8460cbac0813cafefdd39
{
"todoId" : "52cfe9ad-bd56-4313-920f-9111eb5224bb",
"todoTitle" : "Hello World!",
"finished" : false,
"createdAt" : "2014-04-02T09:02:40.329+0000"
}
download
length: 136 bytes


Dev HTTP Client
chrome-extension://aejoelaoggembcahagimdiliamlcdmfm/HttpClient.html
DEV HTTP CLIENT
0.6.9.16
REQUEST
HTTP
localhost:8080/todo-api/api/v1/todos/52cfe9ad-bd56-4313-920f-9111eb5224bb
PUT
Send
Save
Reset
HEADERS
form
raw
BODY
text
form
file
TEXT
JSON
XML
length: 0 bytes
RESPONSE
200 OK
elapsed time 19ms
HEADERS
formatted
raw
BODY
formatted
raw
Content-Type: application/json;charset=UTF-8
Date: 2014 Apr 2 18:03:50
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked
X-Track: 419456302a5f4469b0e1751026a920e6
{
"todoId" : "52cfe9ad-bd56-4313-920f-9111eb5224bb",
"todoTitle" : "Hello World!",
"finished" : true,
"createdAt" : "2014-04-02T09:02:40.329+0000"
}
download
length: 135 bytes

```
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.stereotype.Controller;
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.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Controller
@RequestMapping("todos")
public class TodoRestController {
    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    @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;
    }

    @RequestMapping(method = RequestMethod.POST)
    @ResponseBody
    @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;
    }
```

```
        @RequestMapping(value="{todoId}", method = RequestMethod.GET)
        @ResponseBody
        @ResponseStatus(HttpStatus.OK)
        public TodoResource getTodo(@PathVariable("todoId") String todoId) {
            Todo todo = todoService.findOne(todoId);
            TodoResource todoResource = beanMapper.map(todo, TodoResource.class);
            return todoResource;
        }

        @RequestMapping(value="{todoId}", method = RequestMethod.PUT)
        @ResponseBody
        @ResponseStatus(HttpStatus.OK)
        public TodoResource putTodo(@PathVariable("todoId") String todoId) {
            Todo finishedTodo = todoService.finish(todoId);
            TodoResource finishedTodoResource = beanMapper.map(finishedTodo, TodoResource.class)
            return finishedTodoResource;
        }

        @RequestMapping(value="{todoId}", method = RequestMethod.DELETE) // (1)
        @ResponseBody
        @ResponseStatus(HttpStatus.NO_CONTENT) // (2)
        public void deleteTodo(@PathVariable("todoId") String todoId) { // (3)
            todoService.delete(todoId); // (4)
        }

    }
```

項番	説明
(1)	パスから `todoId` を取得するために、`@RequestMapping` アノテーションの `value` 属性にパス変数を指定する。 メソッドが DELETE のリクエストを処理するために、`method` 属性に `RequestMethod.DELETE` 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、”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。

HTTP Dev Client を使用して、実装した API の動作確認を行う。

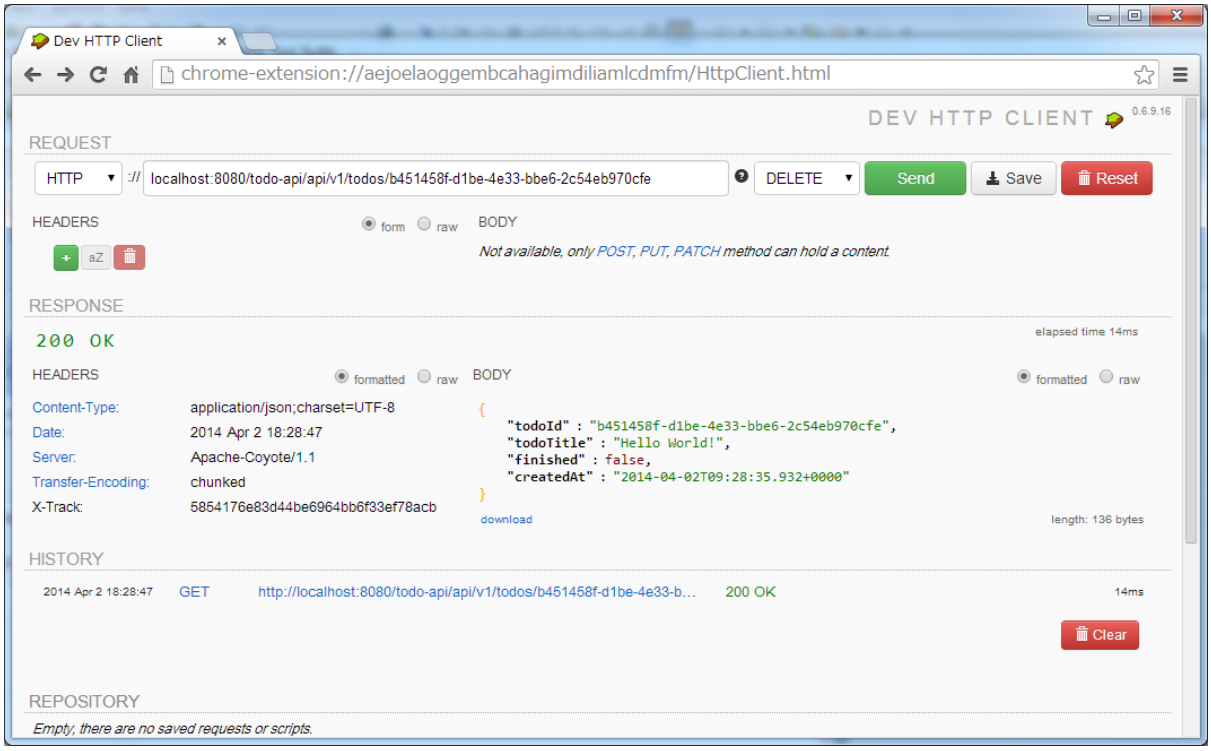
Dev HTTP Client を開いて URL に`"localhost:8080/todo-api/api/v1/todos/{todoId}"`を入力し、メソッドに DELETE を指定する。

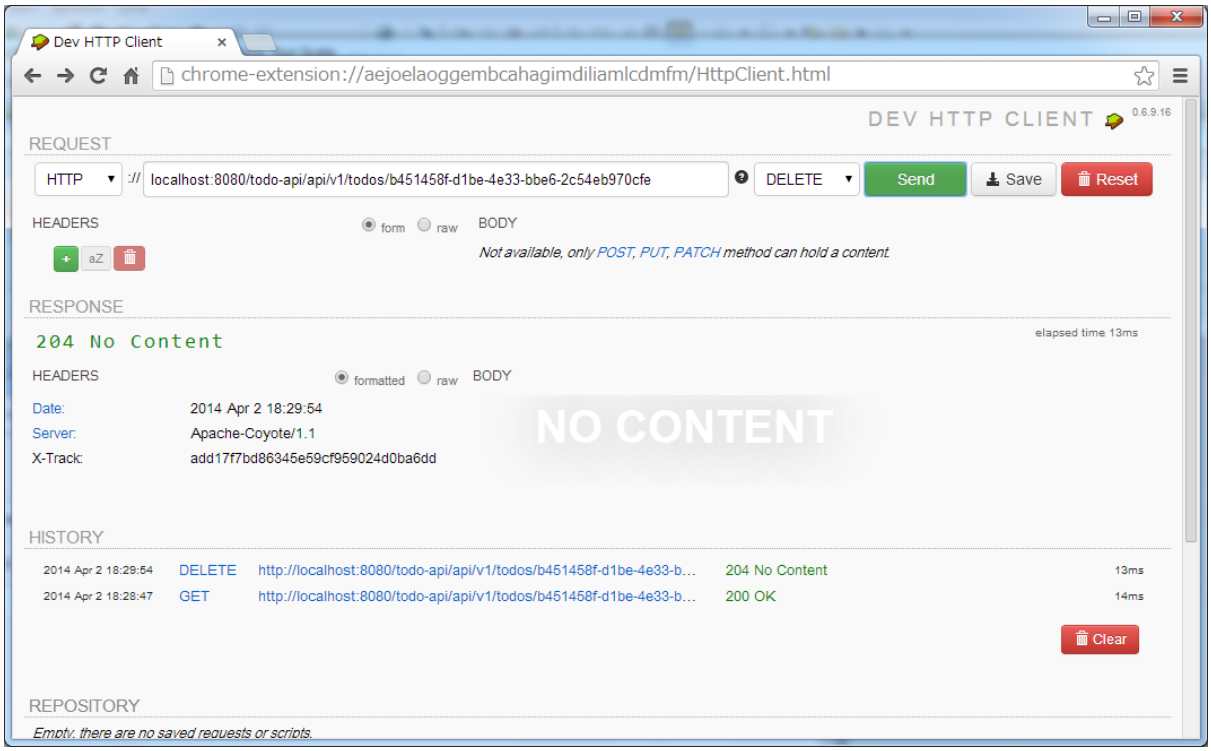
`{todoId}`の部分は実際の ID を入れる必要があるので、POST Todos または GET Todos を実行して Response 中の `todoId` をコピーして貼り付けてから、”Send” ボタンをクリックする。

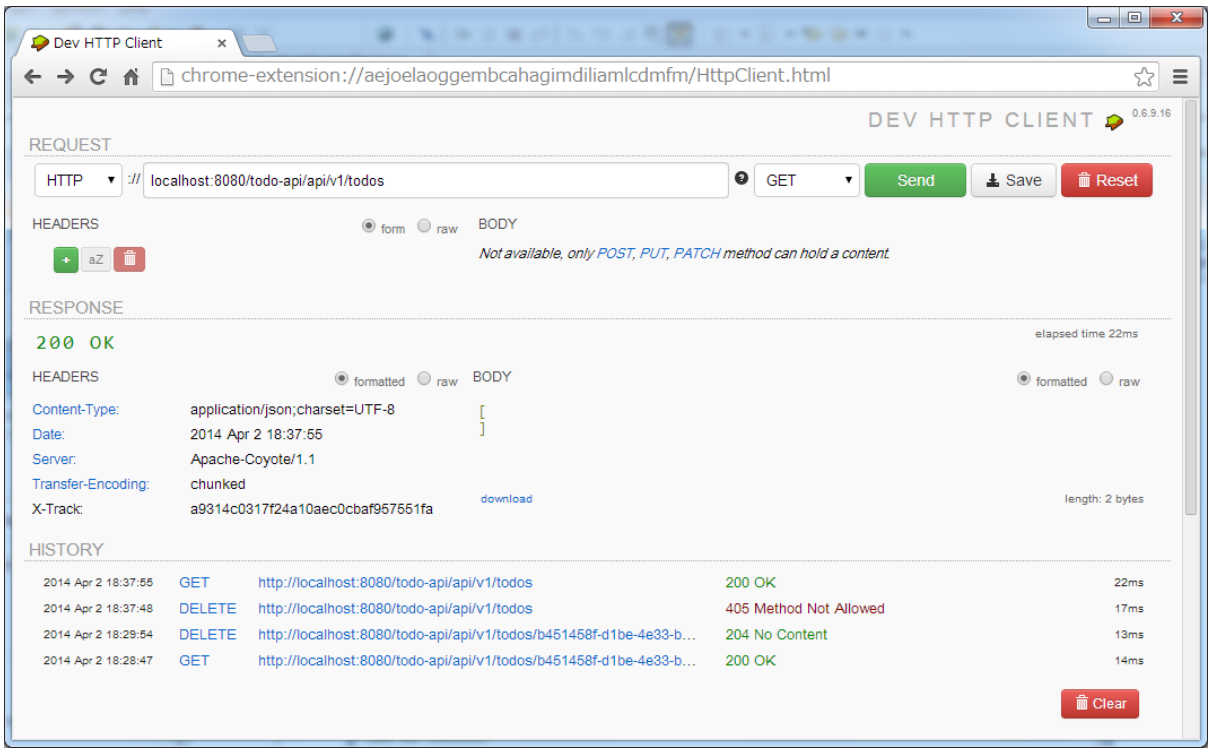
“204 No Content” の HTTP ステータスが返却され、「RESPONSE」の「Body」は空である。

Dev HTTP Client の URL に`"localhost:8080/todo-api/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。


Dev HTTP Client
chrome-extension://aejoelaoggembcahagimdiliamlcdmfm/HttpClient.html
DEV HTTP CLIENT
0.6.9.16
REQUEST
HTTP
localhost:8080/todo-api/api/v1/todos/b451458f-d1be-4e33-bbe6-2c54eb970cfe
DELETE
Send
Save
Reset
HEADERS
form
raw
BODY
Not available, only POST, PUT, PATCH method can hold a content.
RESPONSE
200 OK
elapsed time 14ms
HEADERS
formatted
raw
BODY
formatted
raw
Content-Type: application/json;charset=UTF-8
Date: 2014 Apr 2 18:28:47
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked
X-Track: 5854176e83d44be6964bb6f33ef78acb
{
"todoId" : "b451458f-d1be-4e33-bbe6-2c54eb970cfe",
"todoTitle" : "Hello World!",
"finished" : false,
"createdAt" : "2014-04-02T09:28:35.932+0000"
}
download
length: 136 bytes
HISTORY
2014 Apr 2 18:28:47 GET http://localhost:8080/todo-api/api/v1/todos/b451458f-d1be-4e33-b... 200 OK 14ms
Clear
REPOSITORY
Empty, there are no saved requests or scripts.


Dev HTTP Client
chrome-extension://aejoelaoggembcahagimdiliamlcdmfm/HttpClient.html
DEV HTTP CLIENT
0.6.9.16
REQUEST
HTTP
localhost:8080/todo-api/api/v1/todos/b451458f-d1be-4e33-bbe6-2c54eb970cfe
DELETE
Send
Save
Reset
HEADERS
form
raw
BODY
Not available, only POST, PUT, PATCH method can hold a content.
RESPONSE
204 No Content
elapsed time 13ms
HEADERS
formatted
raw
BODY
Date: 2014 Apr 2 18:29:54
Server: Apache-Coyote/1.1
X-Track: add17f7bd86345e59cf959024d0ba6dd
NO CONTENT
HISTORY
2014 Apr 2 18:29:54 DELETE http://localhost:8080/todo-api/api/v1/todos/b451458f-d1be-4e33-b... 204 No Content 13ms
2014 Apr 2 18:28:47 GET http://localhost:8080/todo-api/api/v1/todos/b451458f-d1be-4e33-b... 200 OK 14ms
Clear
REPOSITORY
Empty, there are no saved requests or scripts.



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。

- `TodoServiceImpl.java`

  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。

```
package 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 todo.domain.model.Todo;
import todo.domain.repository.todo.TodoRepository;

@Service
@Transactional
public class TodoServiceImpl implements TodoService {
    @Inject
    TodoRepository todoRepository;

    private static final long MAX_UNFINISHED_COUNT = 5;

    @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.save(todo);

            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.save(todo);
            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。

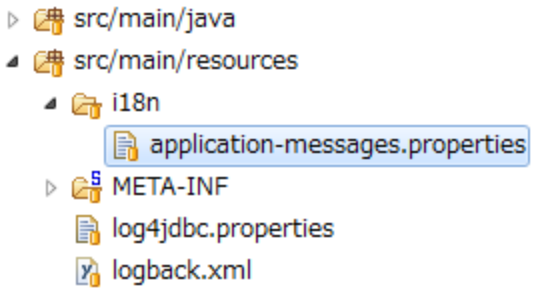
処理結果用のエラーコードに対応するエラーメッセージを、メッセージ用のプロパティファイル (`src/main/resources/i18n/application-messages.properties`) に定義する。

- `application-messages.properties`

  ハイライトした部分のメッセージ定義を追加する。

```
    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.8001 = Business error occurred!

    e.xx.fw.9001 = System error occurred!
```

```
    e.xx.fw.9002 = Data Access error!

    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}
```



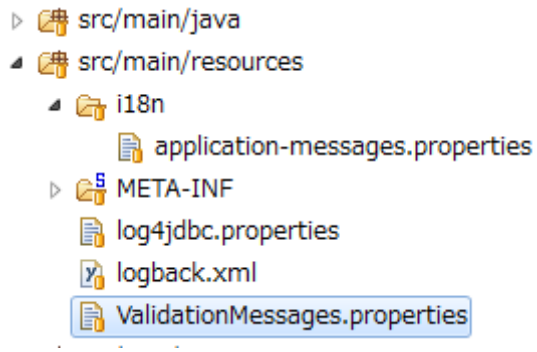
入力チェック用のエラーコードに対応するエラーメッセージを、Bean Validation のメッセージ用のプロパティファイル (`src/main/resources/ValidationMessages.properties`) に定義する。

- `ValidationMessages.properties`

  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。

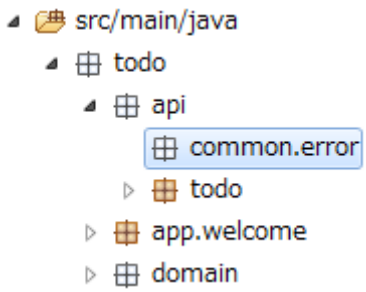
```
    javax.validation.constraints.NotNull.message = {0} may not be null.
    javax.validation.constraints.Size.message    = {0} size must be between {min} and {max}.
```



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、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事を推奨する。

以下に、ResponseEntityExceptionHandler を継承した todo.api.common.error.RestGlobalExceptionHandler クラスを作成する。

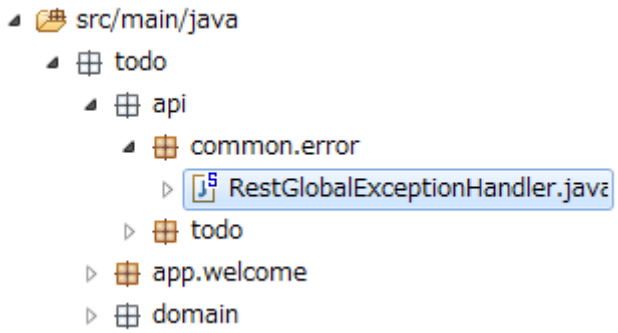
- RestGlobalExceptionHandler.java

```
package 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 todo.api.common.error パッケージに作成する。

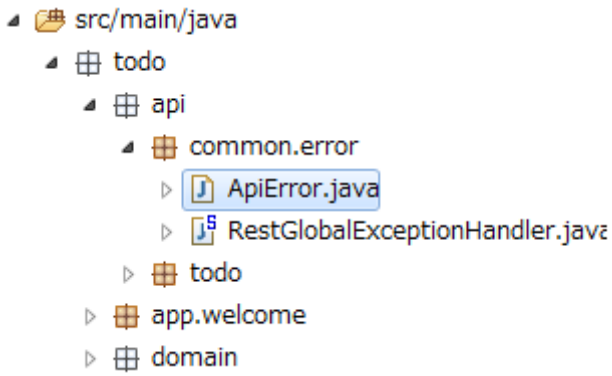
ApiError クラスが JSON に変換されて、クライアントに応答される。



- ApiError.java

```
package todo.api.common.error;

import java.util.ArrayList;
import java.util.List;

import org.codehaus.jackson.map.annotate.JsonSerialize;
import org.codehaus.jackson.map.annotate.JsonSerialize.Inclusion;

public class ApiError {

    private final String code;

    private final String message;

    @JsonSerialize(include = Inclusion.NON_EMPTY)
    private final String target;

    @JsonSerialize(include = Inclusion.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。

- `RestGlobalExceptionHandler.java`

```
    package 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 new ResponseEntity<Object>(responseBody, headers, status);
        }

        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。

HTTP Dev Client 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。

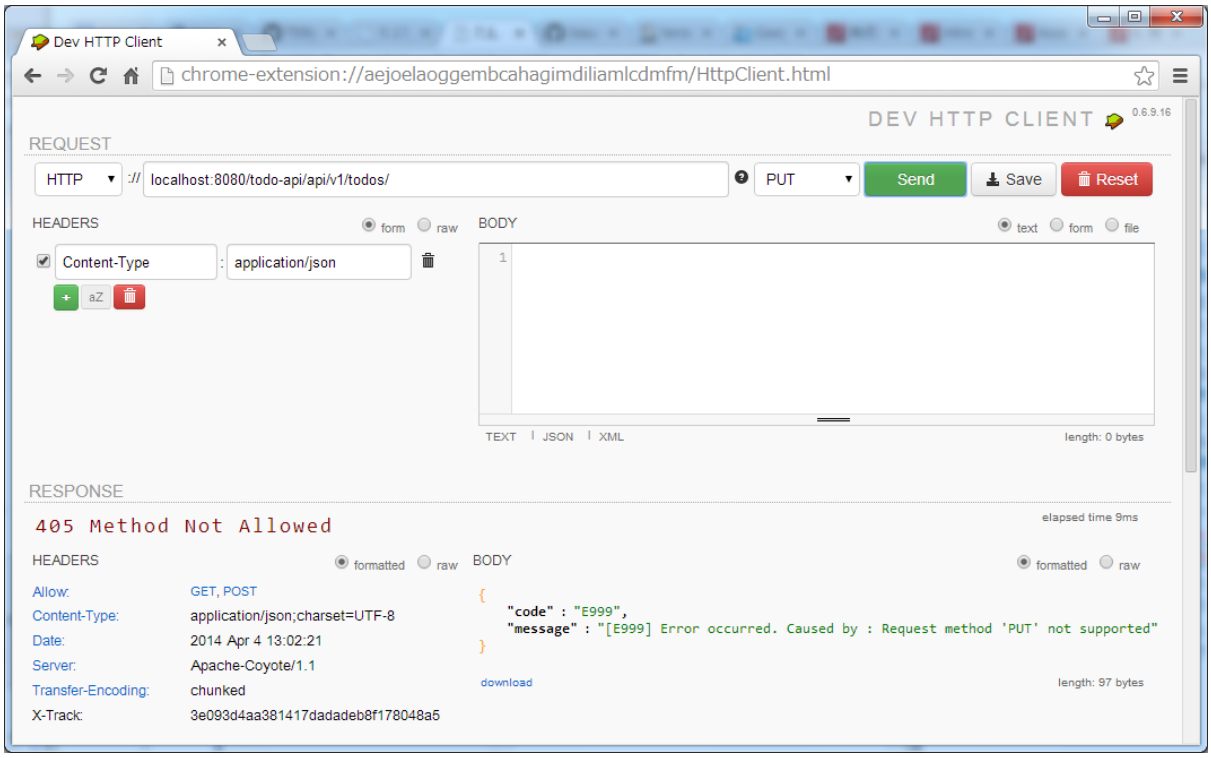
Dev HTTP Client を開いて URL に"localhost:8080/todo-api/api/v1/todos"を入力し、メソッドに PUT を指定してから、”Send” ボタンをクリックする。

“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。

- `RestGlobalExceptionHandler.java`

```
package 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 new ResponseEntity<Object>(responseBody, headers, status);
    }

    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);
    }

}
```

HTTP Dev Client 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。

Dev HTTP Client を開いて URL に"localhost:8080/todo-api/api/v1/todos"を入力し、メソッドに POST を指定する。

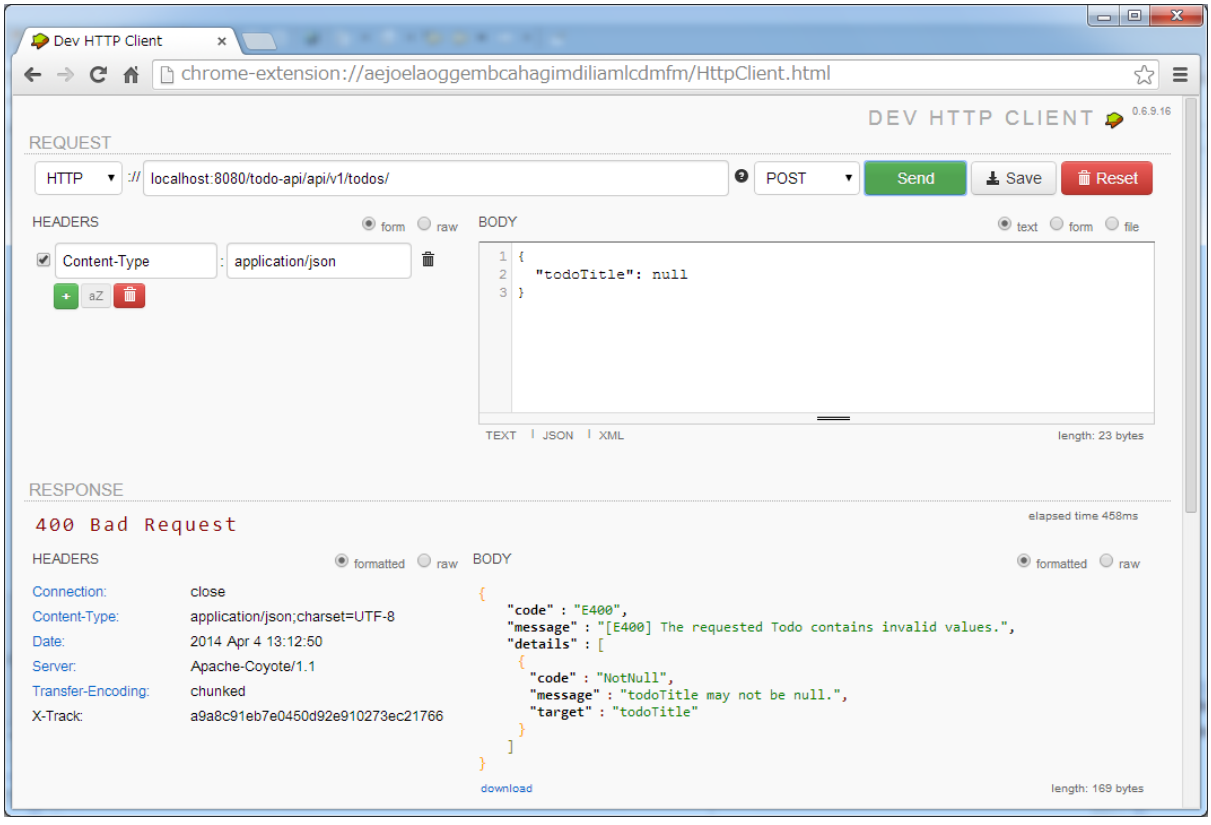
「REQUEST」の「BODY」に以下の JSON を入力する。

```
{
  "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。

- RestGlobalExceptionHandler.java

```
package 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 new ResponseEntity<Object>(responseBody, headers, status);
    }

    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, null,
                    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);
        }

    }
```

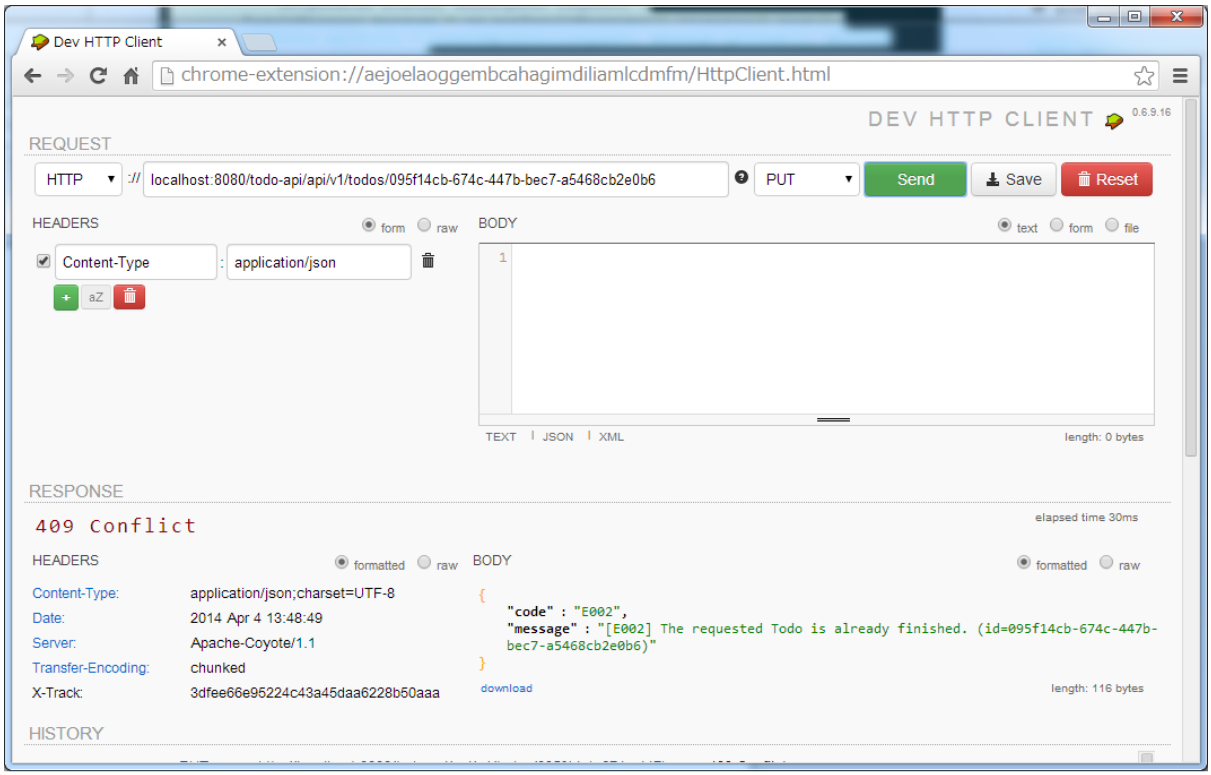
HTTP Dev Client を使用して、実装したエラーハンドリングの動作確認を行う。

Dev HTTP Client を開いて URL に"localhost:8080/todo-api/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.ResourceNotFoundExceptio` 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。

- `RestGlobalExceptionHandler.java`

```
package 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 new ResponseEntity<Object>(responseBody, headers, status);
    }

    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, null,
                    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, null,
                    HttpStatus.NOT_FOUND, request);
        }

    }
```

HTTP Dev Client 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。

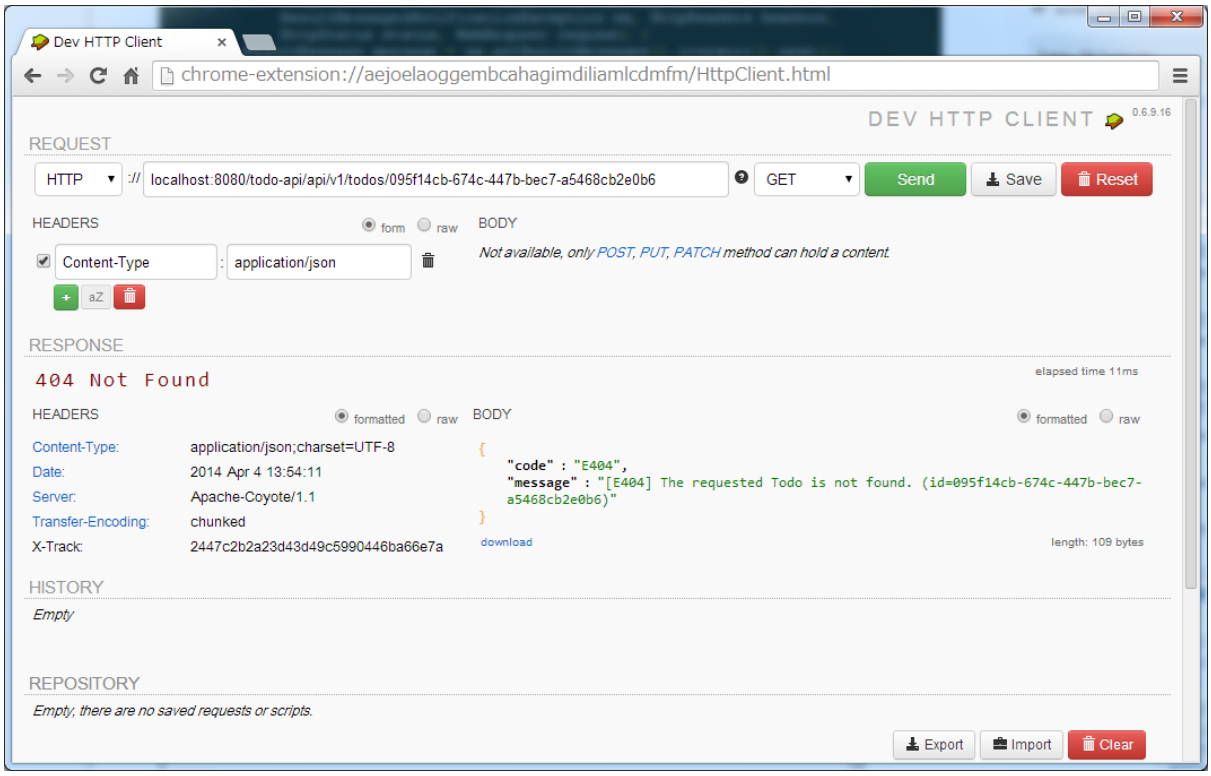
Dev HTTP Client を開いて URL に"localhost:8080/todo-api/api/v1/todos/{todoId}"を入力し、メソッドに GET を指定する。

{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。

- `RestGlobalExceptionHandler.java`

```
package 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 new ResponseEntity<Object>(responseBody, headers, status);
        }

        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, null,
                    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, null,
                    HttpStatus.NOT_FOUND, request);
        }

        @ExceptionHandler(Exception.class)
        public ResponseEntity<Object> handleSystemError(Exception ex,
                WebRequest request) {
            ApiError apiError = createApiError(request, "E500");
            return handleExceptionInternal(ex, apiError, null,
                    HttpStatus.INTERNAL_SERVER_ERROR, request);
        }

    }
```

HTTP Dev Client 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。

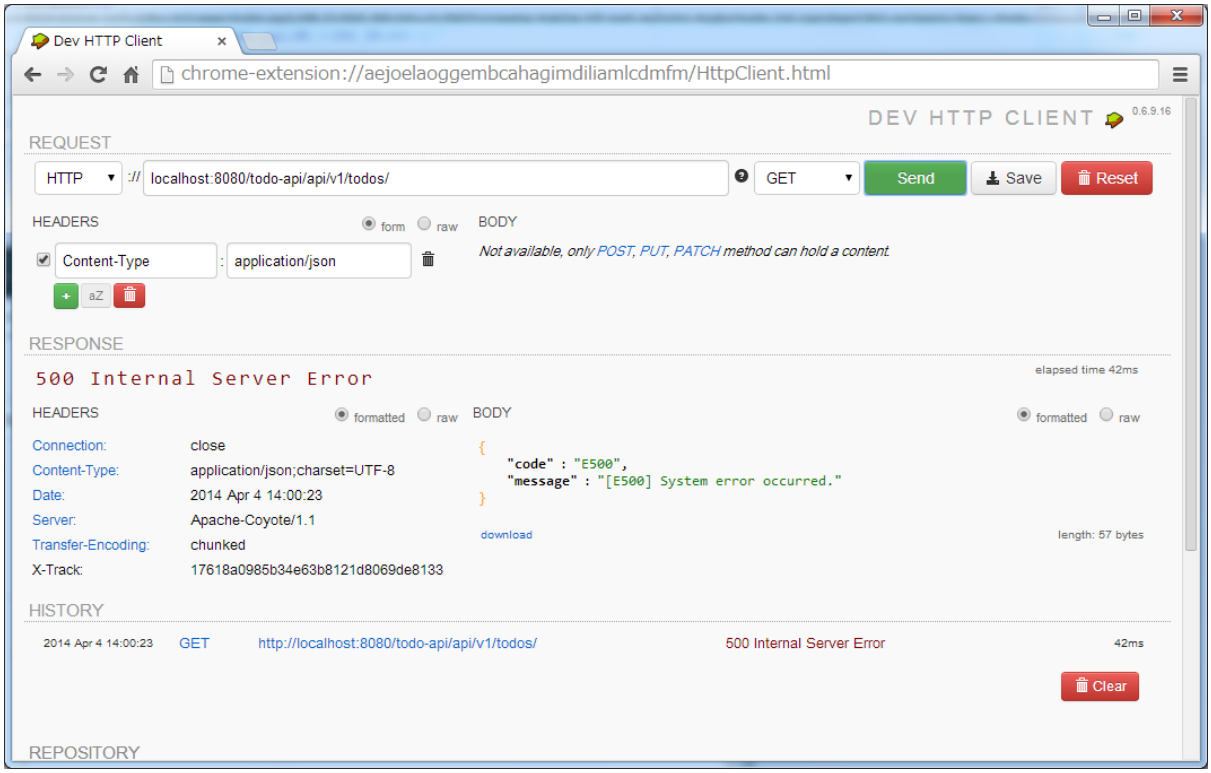
システムエラーを発生させるために、テーブルを未作成の状態でアプリケーションを起動させる。

- todo-api-infra.properties

```
database=H2
#database.url=jdbc:h2:mem:todo-api;DB_CLOSE_DELAY=-1;INIT=create table if not exists todo(to
database.url=jdbc:h2:mem:todo-api;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
```

Dev HTTP Client を開いて URL に"localhost:8080/todo-api/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:2014-04-04 15:15:07    thread:tomcat-http--3   X-Track:09689dcae9d04ad3b25650cba2ebb
        --- The error occurred while executing query.
        --- Check the  SELECT todo_id,        todo_title,        finished,        created_at FROM
        --- Check the SQL Statement (preparation failed).
        --- Cause: org.h2.jdbc.JdbcSQLException: Table "TODO" not found; SQL statement:
         SELECT todo_id,        todo_title,        finished,        created_at FROM   todo  [4210
        org.springframework.jdbc.BadSqlGrammarException: SqlMapClient operation; bad SQL grammar
        --- The error occurred while executing query.
        --- Check the  SELECT todo_id,        todo_title,        finished,        created_at FROM
        --- Check the SQL Statement (preparation failed).
        --- Cause: org.h2.jdbc.JdbcSQLException: Table "TODO" not found; SQL statement:
```

```
          SELECT todo_id,        todo_title,        finished,        created_at FROM   todo  [4210
             at org.springframework.jdbc.support.SQLErrorCodeSQLExceptionTranslator.doTranslate(SQ
             at org.springframework.jdbc.support.AbstractFallbackSQLExceptionTranslator.translate(
             at org.springframework.orm.ibatis.SqlMapClientTemplate.execute(SqlMapClientTemplate.j
             at org.springframework.orm.ibatis.SqlMapClientTemplate.queryForList(SqlMapClientTempl
             at jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl.executeForObjectList(QueryDAOiBatis
             at todo.domain.repository.todo.TodoRepositoryImpl.findAll(TodoRepositoryImpl.java:33)
             at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.7.0_51]
             ...

             (omitted)
```

7.1.4 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 Global Framework 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。

7.2 Blank プロジェクトから新規プロジェクト作成

Blank プロジェクトからの新規プロジェクトを作成する方法を説明する。

- 前提
  - Spring Tool Suite が動作可能なこと。
  - Maven をコマンドラインで実行可能なこと。
  - インターネットに接続できること。

この節の内容は以下のバージョンで動作確認している。

Product	Version
JDK	1.7.0_40
Spring Tool Suite (STS)	3.4.0
Maven	3.0.4

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。

7.2.1 新規プロジェクト作成

1. プロジェクトを作成するフォルダにマンドプロンプトで移動する。

```
cd C:\work
```

2. Maven archetype を利用してプロジェクトの雛形を生成する

```
mvn archetype:generate -B^
 -DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-
 -DarchetypeGroupId=org.terasoluna.gfw.blank^
 -DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype^
 -DarchetypeVersion=1.0.4.RELEASE^
 -DgroupId=example.first.application^
 -DartifactId=example-first-application^
 -Dversion=1.0.0-SNAPSHOT
```

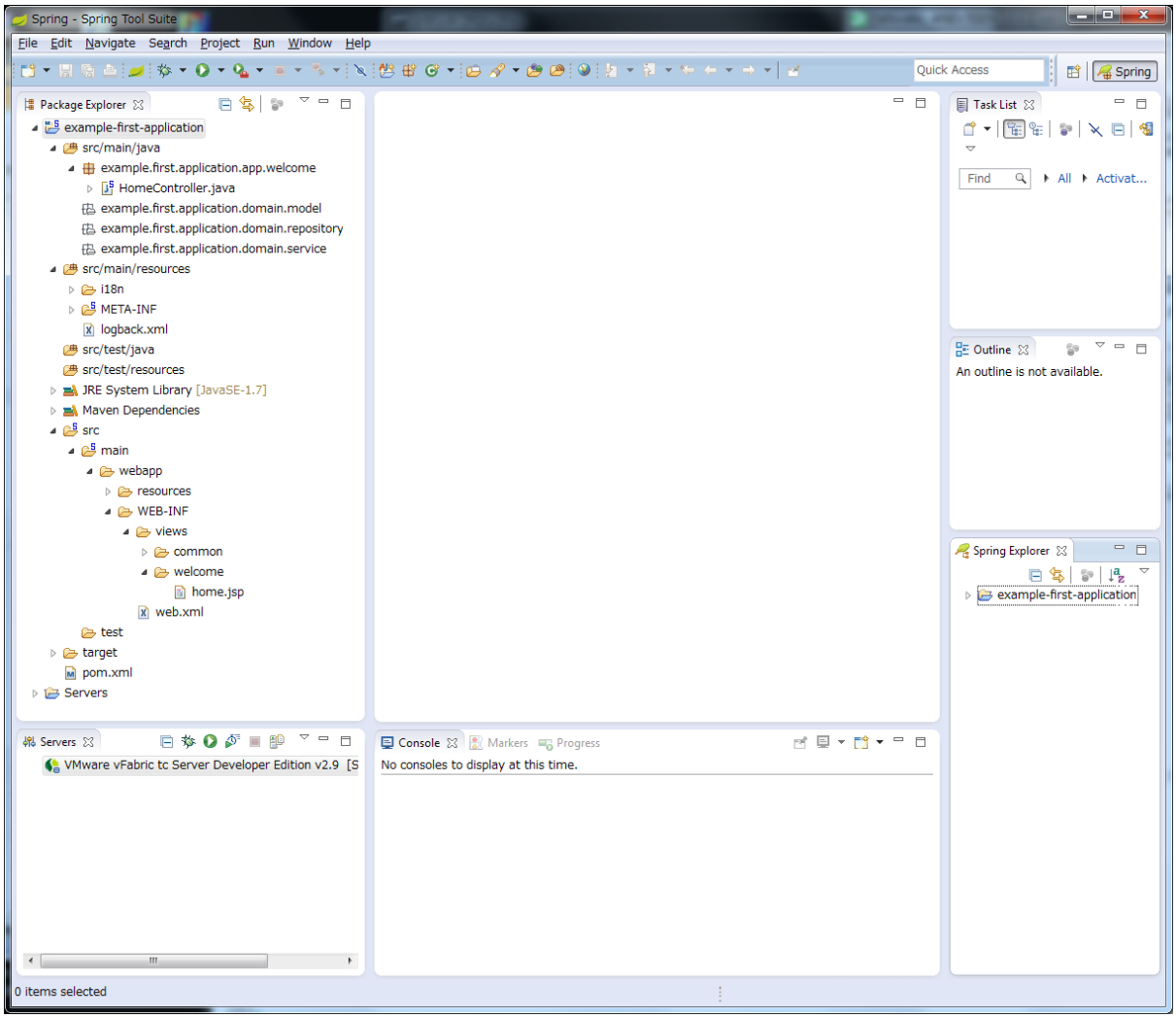
パラメータ	説明
-B	batch mode (対話を省略)
-DarchetypeCatalog	Maven archetype でプロジェクトを作成する。TERASOLUNA Global Framework のレポジトリを指定する。
-DarchetypeGroupId	tarasoluna.gfw のブランクプロジェクトの GroupID = org.terasoluna.gfw.blank 固定
-DarchetypeArtifactId	terasoluna-gfw-web-blank-archetype = JPA,MyBatis2 は使用せずにプロジェクトを作成する場合 terasoluna-gfw-web-blank-jpa-archetype = JPA を使用するプロジェクトを作成する場合 terasoluna-gfw-web-blank-mybatis2-archetype = MyBatis2 を使用するプロジェクトを作成する場合
-DarchetypeVersion	terasoluna.gfw のバージョンを指定する。
-DgroupId	作成プロジェクトの groupId を指定する。
-DartifactId	作成プロジェクトの artifactId を指定する。
-Dversion	作成プロジェクトのバージョンを指定する。

3. Spring Tool Suite にプロジェクトをインポートする。

   [STS] -> [File] -> [Import] -> [Maven] -> [Exsiting Maven Projects] ->[Browse...] で Maven archetype で作成したプロジェクトを指定 -> 1 つ表示される pom.xml にチェックが入ってい

ることを確認して [Finish]

以下のような状態になる。



4. アプリケーションサーバに作成したプロジェクトを追加する。

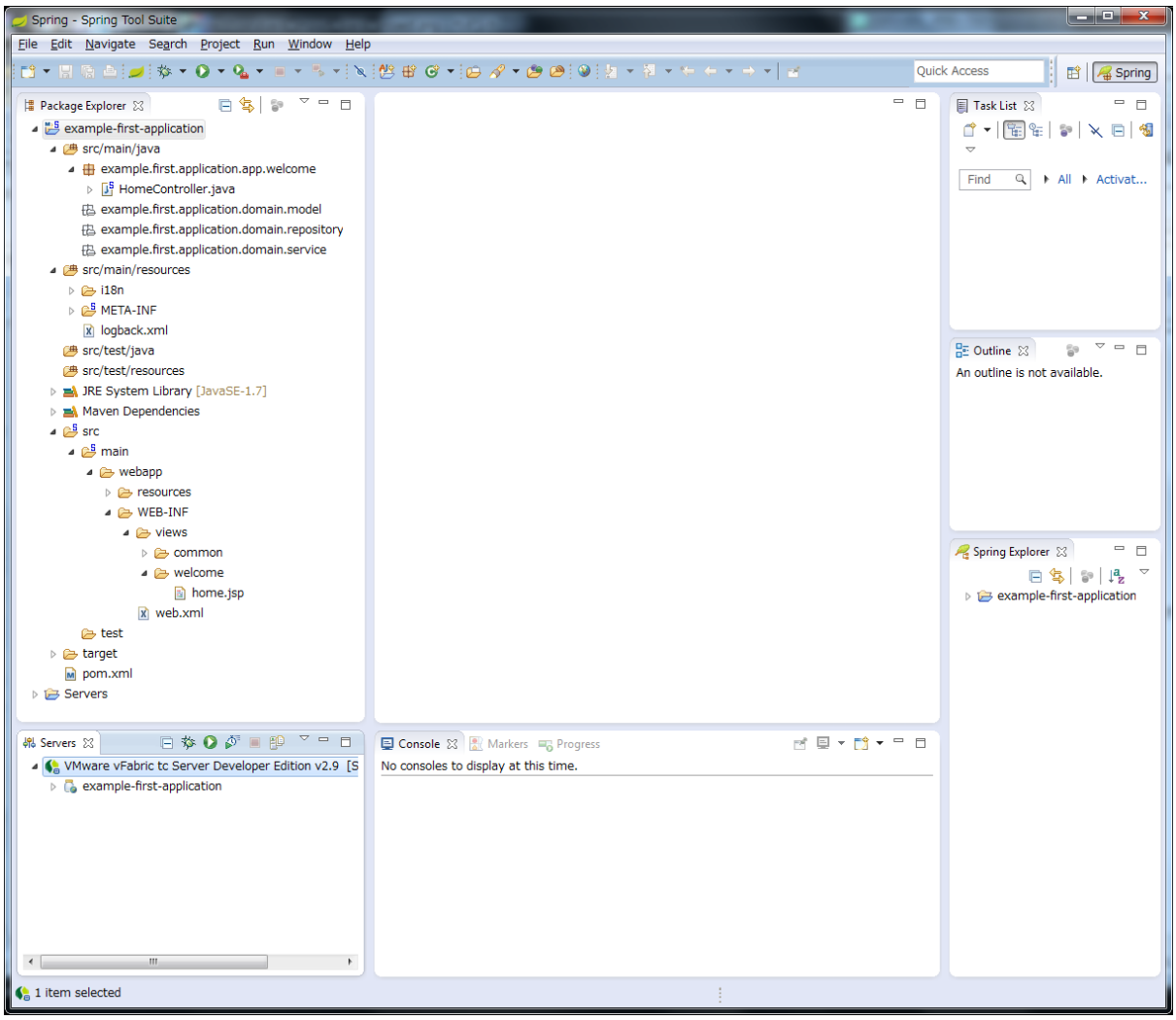
   ここではデフォルトで STS に付属する VMware vFabric tc Server Developer Edition v2.9 を使用する。

   [VMware vFabric tc Server Developer Edition v2.9] を右クリック -> [Add and Remove] -> 作成したプロジェクトを選択して [Add] -> [Finish]

   以下のような状態になる。

5. アプリケーションサーバを起動する。

   Server のスタートボタンを押して起動する。Console に以下が出力されれば起動成功となる。




```
        FrameworkServlet 'appServlet': initialization completed
```

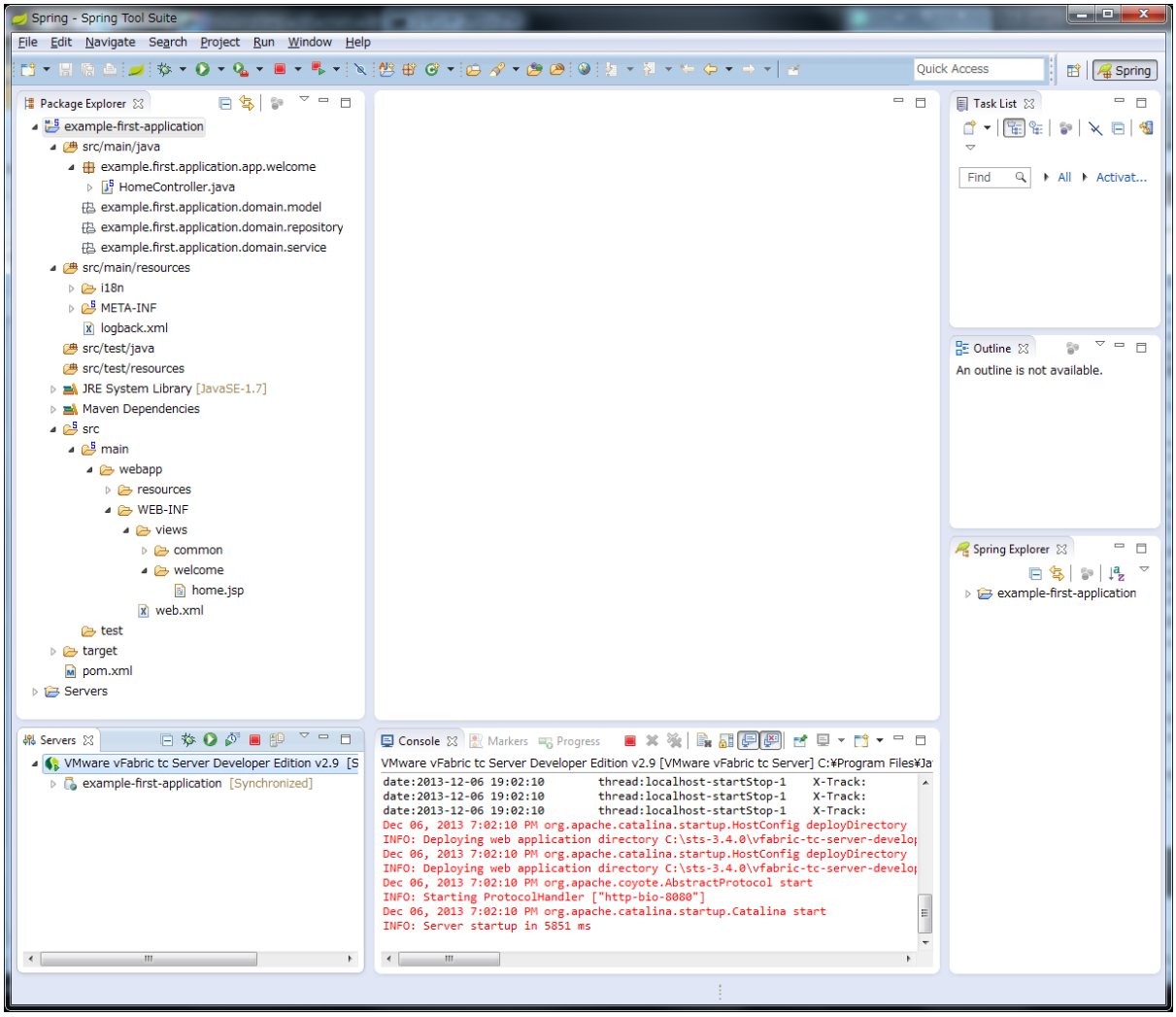
下記の出力例を見ると、Console にログが出力されるが、赤文字の一行上の行に"`FrameworkServlet 'appServlet': initialization completed`"が出力される (スクリーンキャプチャ上には表示されていない)。

6. 起動したアプリケーションにアクセスする。

   ブラウザで http://localhost:8080/example-first-application/にアクセスする。

   以下のような画面が表示される。

   「Hellow world!」が表示されれば、プロジェクトの作成準備は完了である。ガイドラインに沿って、必要な機能を追加していくこと。


Spring - Spring Tool Suite
File Edit Navigate Search Project Run Window Help
Quick Access
Spring
Package Explorer
example-first-application
src/main/java
example.first.application.app.welcome
HomeController.java
example.first.application.domain.model
example.first.application.domain.repository
example.first.application.domain.service
src/main/resources
i18n
META-INF
logback.xml
src/test/java
src/test/resources
JRE System Library [JavaSE-1.7]
Maven Dependencies
src
main
webapp
resources
WEB-INF
views
common
welcome
home.jsp
web.xml
test
target
pom.xml
Servers
Task List
Find
All
Activat...
Outline
An outline is not available.
Spring Explorer
example-first-application
Servers
VMware vFabric tc Server Developer Edition v2.9 [S
example-first-application [Synchronized]
Console
Markers
Progress
VMware vFabric tc Server Developer Edition v2.9 [VMware vFabric tc Server] C:¥Program Files¥Ja
date:2013-12-06 19:02:10 thread:localhost-startStop-1 X-Track:
date:2013-12-06 19:02:10 thread:localhost-startStop-1 X-Track:
date:2013-12-06 19:02:10 thread:localhost-startStop-1 X-Track:
Dec 06, 2013 7:02:10 PM org.apache.catalina.startup.HostConfig deployDirectory
INFO: Deploying web application directory C:\sts-3.4.0\vfabric-tc-server-develop
Dec 06, 2013 7:02:10 PM org.apache.catalina.startup.HostConfig deployDirectory
INFO: Deploying web application directory C:\sts-3.4.0\vfabric-tc-server-develop
Dec 06, 2013 7:02:10 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8080"]
Dec 06, 2013 7:02:10 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 5851 ms

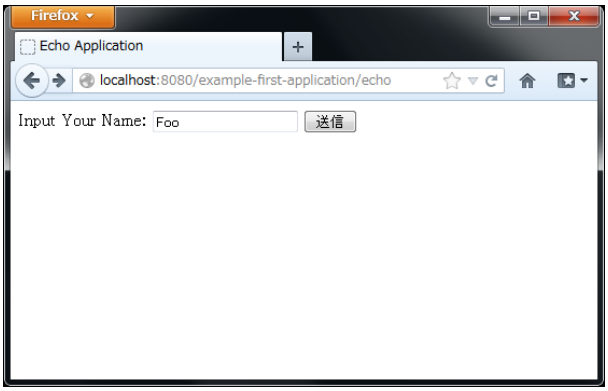

Firefox
Home
localhost:8080/example-first-application/
Hello world!
The time on the server is 2013/12/06 19:05:36 JST.

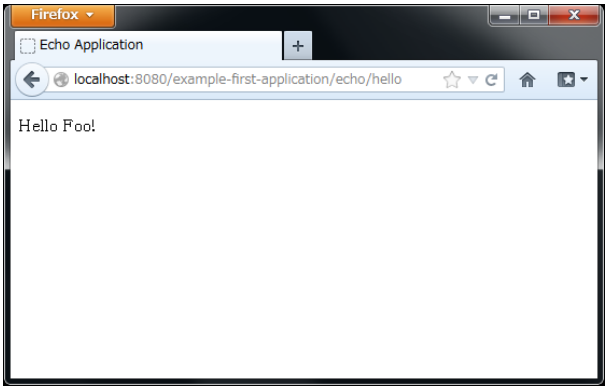
7.2.2 簡単な Echo プロジェクトの作成

はじめての Spring MVC アプリケーションで説明した**エコーアプリケーションの作成**と基本的には同じ手順となるため、ソースは割愛する。

はじめての Spring MVC アプリケーションでは spring-mvc.xml の中で、`<context:component-scan base-package="com.example.helloworld" />`を設定しているが、が Blank Project から作成した場合は`<context:component-scan base-package="example.first.application.app" />`と設定される。

`EchoController` は `example.first.application.app.echo` パッケージで作成すること。





課題

TBD

今回はシングルプロジェクト構成での作成方法の説明をしているが、これは主に検証目的で使用されることを想定している。実際には**マルチプロジェクト構成**でプロジェクトを構築する必要がある。マルチプロジェクト構成での作成方法は今後追記する。

警告: Blank プロジェクトの pom.xml に定義されている、以下の設定用はサンプルアプリケーションを簡単試すためのものであり、実際の開発で使用されることを想定していない。実際のプロジェクトでは削除すること。

```
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>1.3.172</version>
        <scope>compile</scope>
    </dependency>
```

7.3 共通ライブラリが提供する JSP Tag Library と EL Functions

7.3.1 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

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 内の HTML 特殊文字をエスケープ
2.	*f:js()*	指定された文字列内の JavaScript 特
3.	*f:hjs()*	指定された文字列内の JavaScript 特 HTML 特殊文字をエスケープする 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、UT 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

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 を生成する。
7.	*f:br()*	指定された文字列内の改行コードを

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

7.3.2 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 引数で指定さらたオブジェクトが `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 引数で指定さらた文字列が `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 引数で指定さらた文字列が `null` の場合は、空文字 (`""`) を返却する。

f:hjs() 使用方法

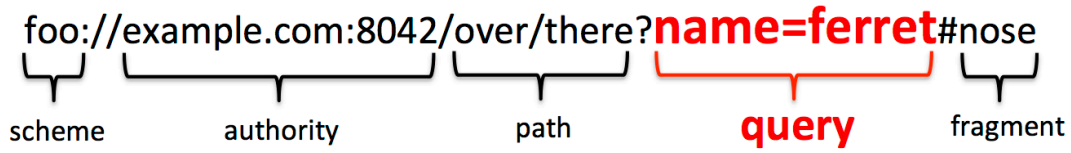
`f:hjs()` の使用方法については、「**出力値を *f:hjs()* 関数でエスケープする例**」を参照されたい。

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。

注釈: 指定されたオブジェクトのフィールド値は、`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。

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 引数で指定さらた文字列が `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、 • 引数で指定さらた文字列が `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`<br />`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`<br />`タグに変換する必要がある。

例えば、入力画面のテキストアリア (`<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ら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 引数で指定さらた文字列が `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、"<br />"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) 引数で指定さらた文字列が `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。

7.4 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 版と商用版がありますが、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。

7.4.1 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。

7.4.2 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。

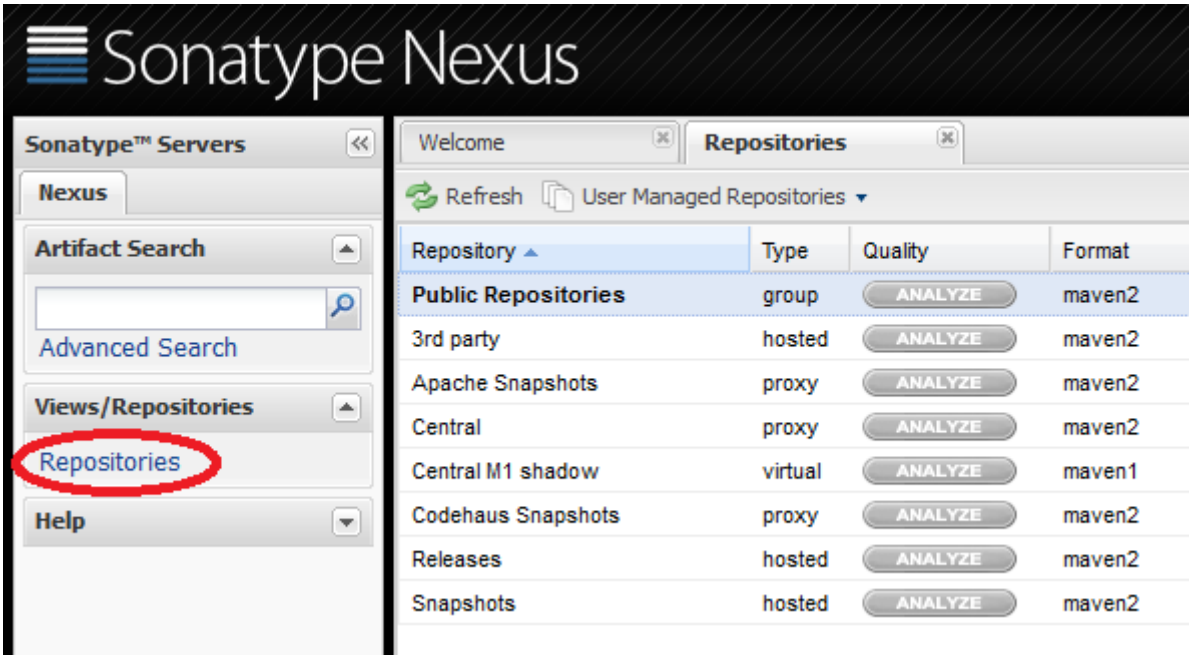
- JRE6 以上がインストール済みであること
- インターネット上の下記の URL に http アクセス可能であること
- 先頭が http://repo1.maven.org/ で始まる URL（セントラルリポジトリ）
- 先頭が http://repo.terasoluna.org/ で始まる URL（Terasoluna リポジトリ）

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。

7.4.3 Add TERASOLUNA Global Framework repository

TERASOLUNA Global Framework を用いて開発する場合、上記で説明したリポジトリに加えて、TERASOLUNA Global Framework のリポジトリを追加する必要がある。

課題

http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-releases/と http://repo.terasoluna.org/nexus/content/repositorie gfw-3rdparty/への proxy リポジトリの追加と、public リポジトリグループへの追加方法をキャプチャつきで書く。

7.4.4 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

7.4.5 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。

課題

Jenkins のビルドジョブのキャプチャ画像

7.4.6 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>
```

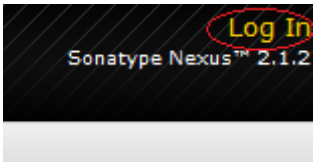
前述の mvn deploy コマンドは、<distributionManagement>タグで指定された URL に対して HTTP PUT で artifact をアップロードする。

7.4.7 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。

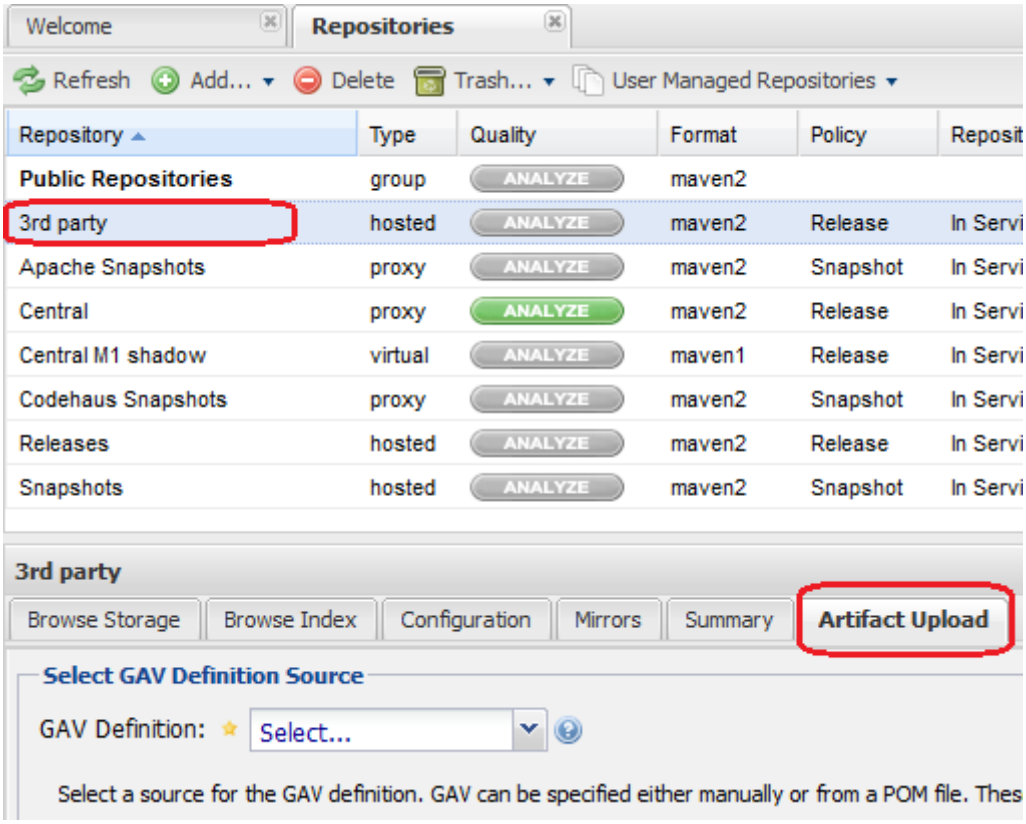
典型的な例が、oracle の JDBC ドライバ (ojdbc*.jar) である。RDBMS として oracle を使用する場合に必須だが、セントラルリポジトリはもちろん、インターネット上の公開リポジトリに格納されていることはほとんどない。そのため、組織内のパッケージリポジトリに格納しておく必要がある。

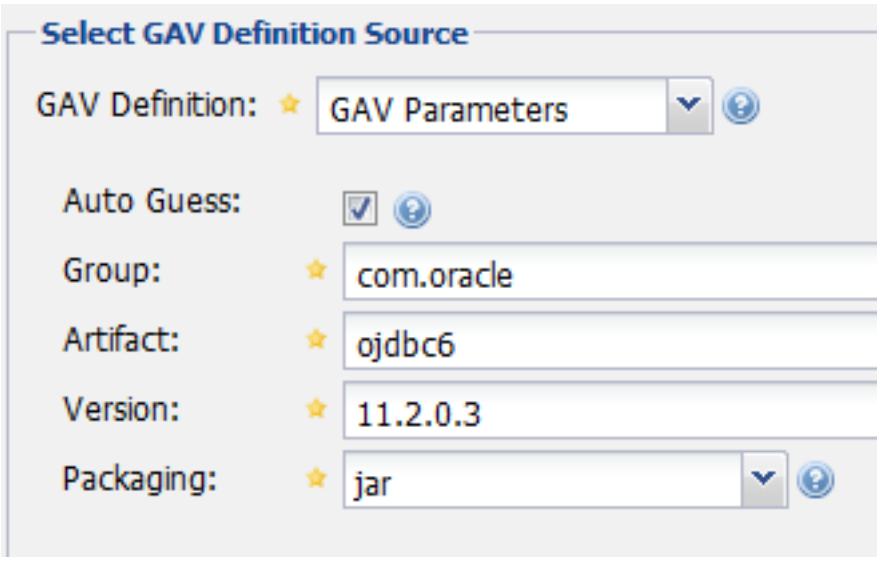
1. admin ユーザーでログインします。（デフォルトのパスワードは admin123）



2. 3rdParty リポジトリを選択し、 **Artifact Upload** タブを選択する。

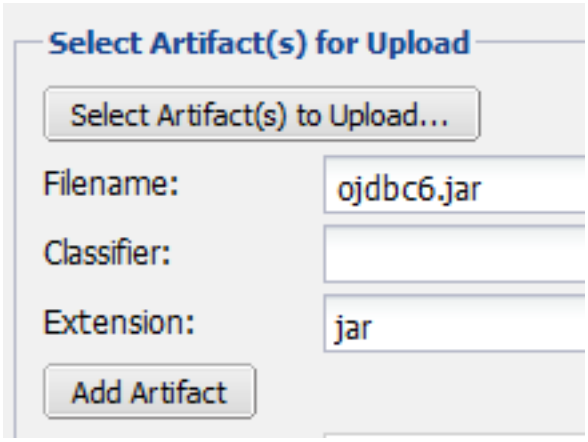
3. GAV 情報を入力します。(GAV = groupId, artifactId, version)

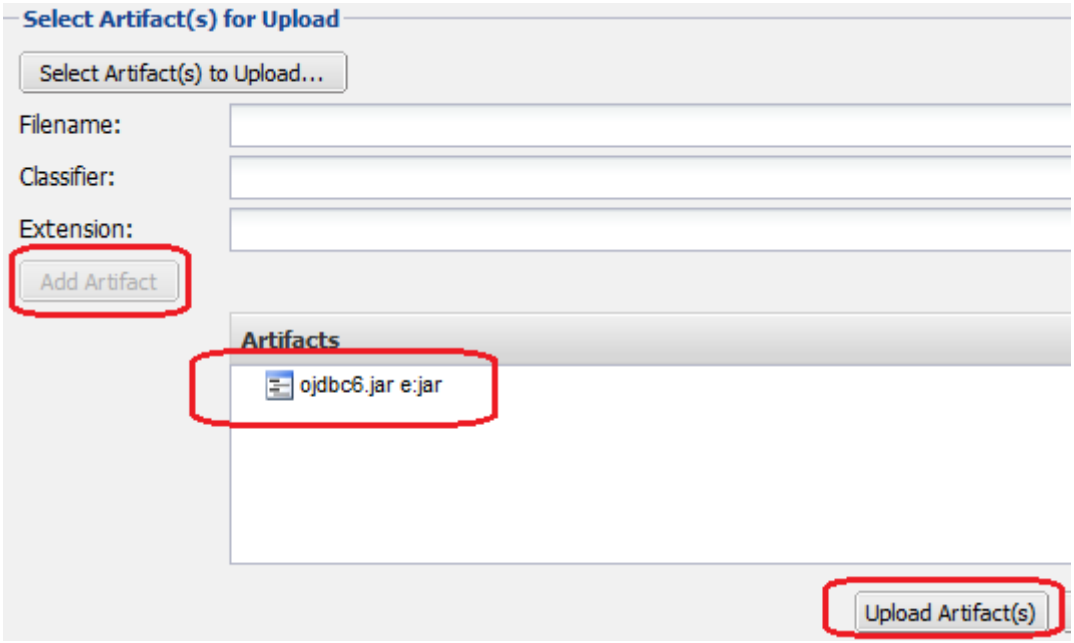




4. ローカル PC 上の ojdbc6.jar ファイルを選択し、 **Add Artifact** ボタンを押す。

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。





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。

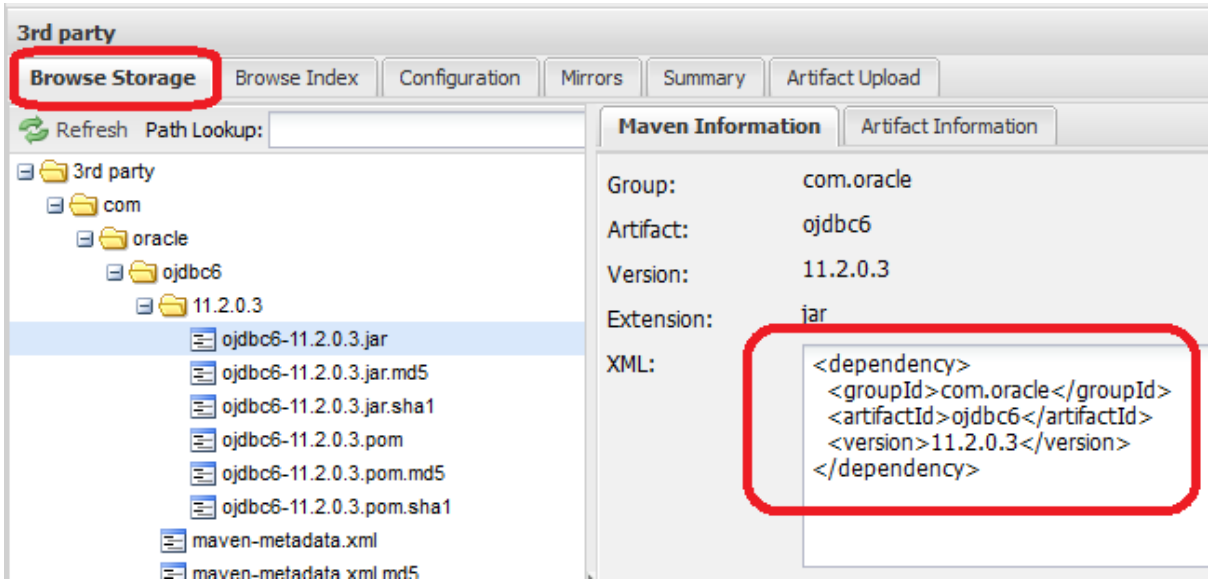
---

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。



7.5 環境依存性の排除

課題

書き直し。

Web アプリケーションの開発プロジェクトでは、必ず環境依存性の問題が発生する。

もしも、datasource.xml ファイルに jdbcurl=hdbc:mysql:127.0.0.1... と書いて、あるいはもしも、logback.xml ファイルに level="DEBUG" と書いて、それらすべてが war ファイルに同梱されてしまったら、その Web アプリケーションは、あなたのローカル PC でしか正常に作動せず、試験用サーバにはリリースできない。

過去、意外と多くの開発プロジェクトが、この単純な問題を軽視していた。そして結合テストの直前になってから、開発した Web アプリケーションを試験用サーバで作動させることが難しいことに気づき、問題の解決のために膨大な時間を費やすことになった。

この章では、環境依存性の問題を解決するための原則と具体的な方法を解説する。

7.5.1 目的

あなたのチームがこれから開発する全てのソースコード、あるいはそのバイナリは、以下のすべてのシチュエーションでシームレスに動作可能でなければならない。

- 全ての開発者の PC の IDE(eclipse) 上で設定された AP サーバ上でのアプリケーションの実行
- 全ての開発者の PC の IDE 上の JUnit プラグインによるテストの実行
- 全ての開発者の PC 上のビルドツール（maven/ant）によるテストの実行
- CI サーバ上でのテストの実行
- CI サーバ上でのパッケージング（jar/war ファイルの生成）
- 試験サーバ上でのアプリケーションの実行
- 本番サーバ上でのアプリケーションの実行

7.5.2 原則

前述の目的を実現するために、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。

7.5.3 デプロイ

Tomcat へのデプロイ

Web アプリケーションを Tomcat 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 VirtualWebappLoader 機能を使って、 /etc/foo/bar/*.jar をクラスパスに追加する。

- [CATALINA_HOME]/conf/[contextPath].xml ファイルに上記の設定を記述すればよい。
- 詳しくは terasoluna-shopping-env サンプルの configs フォルダと、 http://tomcat.apache.org/tomcat-7.0-doc/api/org/apache/catalina/loader/VirtualWebappLoader.html を参考のこと。
- なお、VirtualWebappLoader は Tomcat6.x でも使用可能。

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。

他のアプリケーションサーバーへのデプロイ

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：Resin,Websphere,Weblogic）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。

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 上で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。

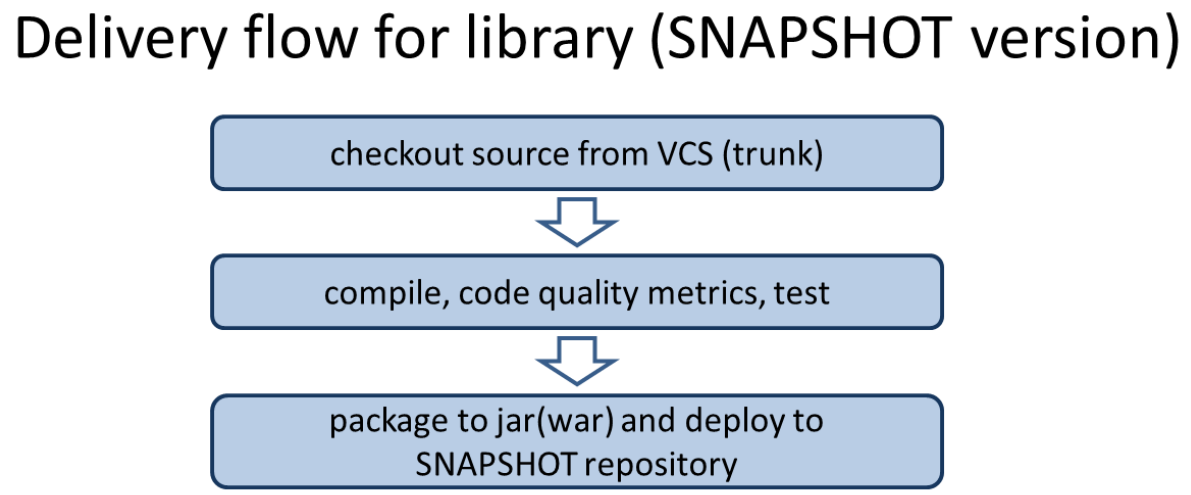
- 末尾が -SNAPSHOT である場合に SNAPSHOT とみなされる。例：<version>1.0-SNAPSHOT</version>
- 末尾が -SNAPSHOT ではない場合は RELEASE とみなされる。例：<version>1.0</version>

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。



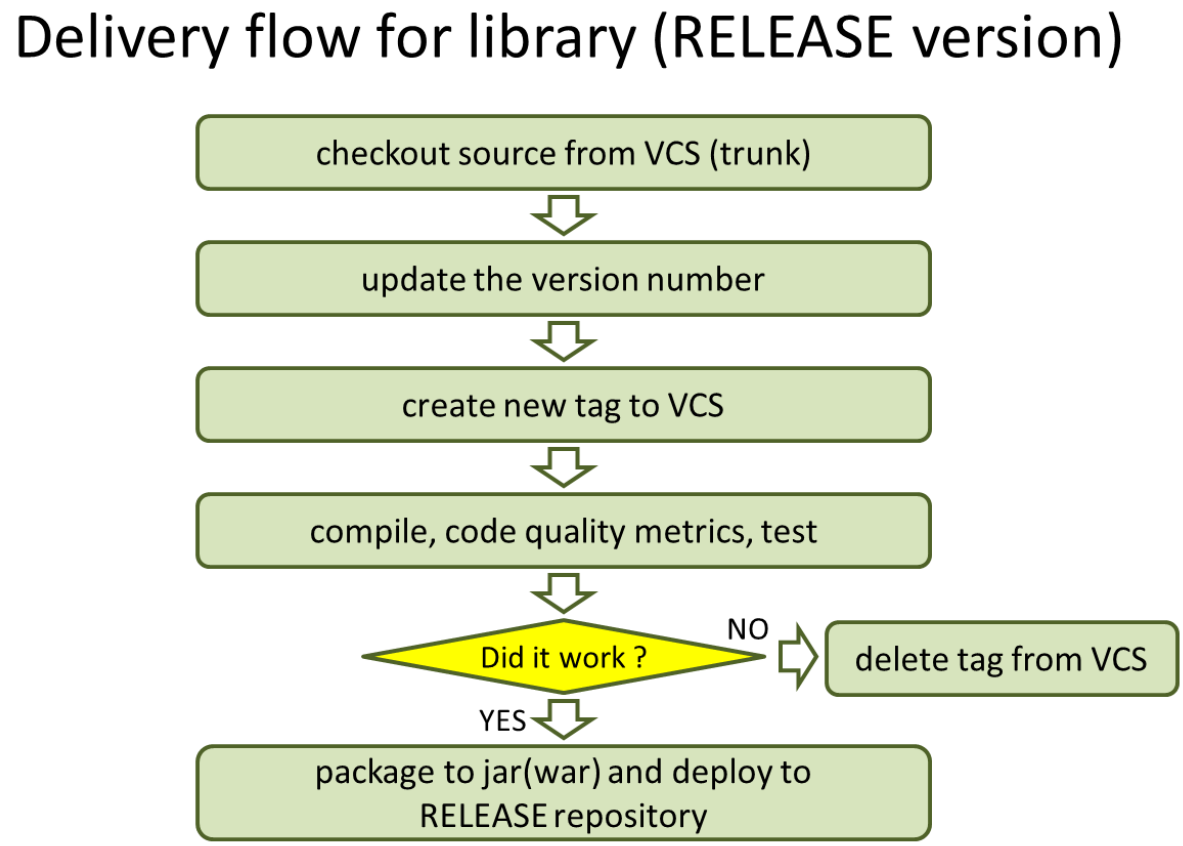
1. 開発用 trunk からソースコードをチェックアウトする。
2. コンパイル、コードメトリクスの測定、テストを実行する。
   - コンパイルエラー、コードメトリクスでの一定以上の violation の発生、テストの失敗の場合、以降の作業を中止する。
3. maven パッケージリポジトリサーバに artifact(jar,war ファイル) をアップロード (mvn deploy) する。

課題

後でキャプチャをはる

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。

課題

ここで最後に trunk のソースツリーの pom.xml の version タグを、次の SNAPSHOT バージョンに書き変えてコミットするところまで書くべきか？！

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。

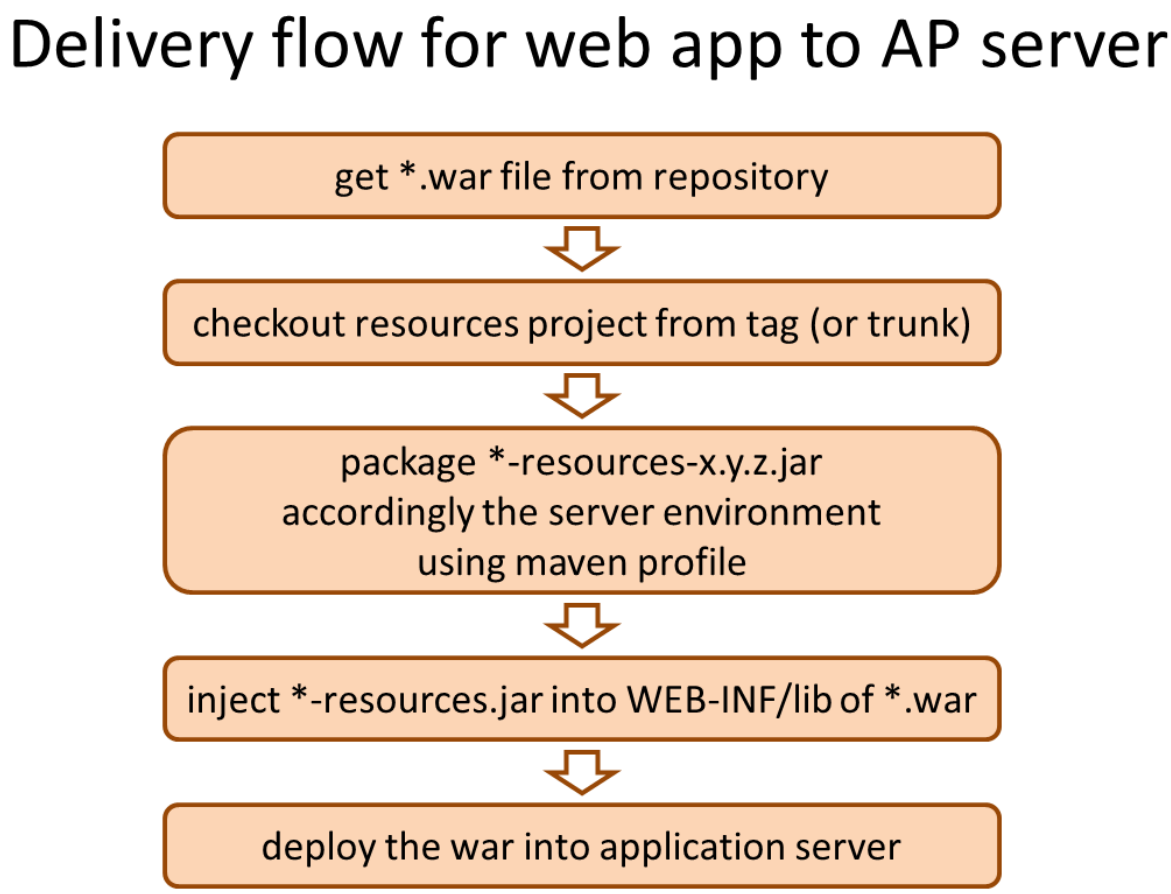
課題

後でキャプチャをはる

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。詳細は 環境依存性の排除 を参照。

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
```

課題

あとでキャプチャをはる

7.6 Project Structure Standard

課題

書き直し。

ソフトウェアのソースコードツリーは、複数のプロジェクトに分かれる、いわゆるマルチプロジェクト構成にすることを推奨する。

注釈: ここではビルドツールとして maven を想定している。maven を使う場合の標準的なマルチプロジェクト構成は、 階層型レイアウト（ Hierachical project layout）である。しかし、開発環境として使用する eclipse は階層型レイアウトではなくフラットレイアウト (flat layout) にしか対応していないため、あえてフラットレイアウトを使用する。

7.6.1 Simple pattern

Web アプリケーション開発プロジェクト「foo」の、最もシンプルなプロジェクト構成は下記のようになる。

- foo-parent
- foo-initdb
- foo-domain
- foo-web
- foo-env
- foo-selenium

それぞれのプロジェクトの内容は下記のようになる。

- foo-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 foo-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。

- foo-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。

- foo-domain

  サービスクラスやリポジトリクラスなど、ドメイン層として使われるクラスを格納する。このドメイン層のクラスを使って foo-web 内のアプリケーション層のクラスを組み立てる。

- foo-web

  アプリケーション層のクラス、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。

- foo-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。foo-web は foo-env への依存性を持つ。詳細は 環境依存性の排除 を参照のこと。

- foo-selenium

  Selenium WebDriver によるテストケースを格納するプロジェクト。

7.6.2 Complex pattern

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「bar」のプロジェクト構成は下記のようになる。

- 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

（foo-parent と同じ）

- bar-initdb

  （foo-initdb と同じ）

- 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 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 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。 詳細は 環境依存性の排除 を参照のこと。

- 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。

課題

JSP も分割したい場合について記述する。

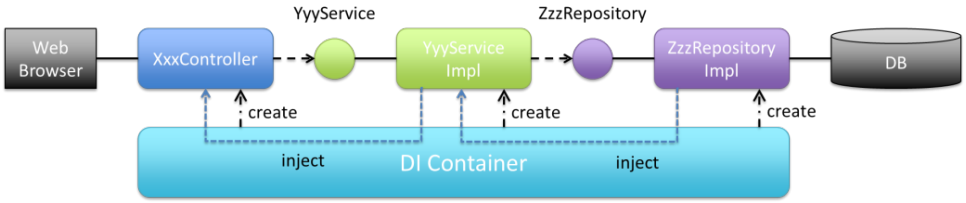
7.7 参考書籍

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
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

7.8 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。




```
@Contoller
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 {
  @(2)("${emails.min.count}")
  protected int emailsMinCount;
  @(2)("${emails.max.count}")
  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。
