TERASOLUNA Global Framework Development Guideline Documentation

Release 1.0.1.RELEASE

NTT DATA

August 28, 2014

CONTENTS

1 In the Beginning 3
1.1 Terms of Use . . . 3
1.2 This document covers the following . . . 5
1.3 Target readers of this document . . . 5
1.4 Structure of this document . . . 5
1.5 Reading this document . . . 6
1.6 Criterion-based mapping of guideline . . . 8
1.6.1 Mapping based on security measures . . . 8
1.7 Change Log . . . 11

2 Summary - Architecture of TERASOLUNA Global Framework 13
2.1 Stack of TERASOLUNA Global Framework . . . 13
2.1.1 Summary of Software Framework of TERASOLUNA Global Framework . . . 13
2.1.2 Main Structural Elements of Software Framework . . . 13
DI Container . . . 13
MVC Framework . . . 14
O/R Mapper . . . 14
View . . . 15
Security . . . 15
Validation . . . 15
Logging . . . 15
Common Library . . . 15
2.1.3 OSS Versions . . . 15
2.1.4 Building blocks of Common Library . . . 17
terasoluna-gfw-common . . . 18
terasoluna-gfw-web . . . 19
terasoluna-gfw-security-web . . . 19
2.2 Overview of Spring MVC Architecture . . . 19
2.2.1 Overview of Spring MVC Processing Sequence . . . 20
2.2.2 Implementation of each component . . . 21
Implementation of HandlerMapping . . . 21
Implementation of HandlerAdapter . . . 21
Implementation of ViewResolver . . . 22

Implementation of View . . . 23
2.3 First application based on Spring MVC . . . 25
2.3.1 Prerequisites . . . 25
2.3.2 Create a New Project . . . 25
2.3.3 Run on Server . . . 29
2.3.4 Create an Echo Application . . . 30
Creating a form object . . . 31
Create a Controller . . . 31
Create JSP Files . . . 33
Implement Input Validation . . . 34
Summary . . . 38
2.4 Application Layering . . . 39
2.4.1 Layer definition . . . 39
Application layer . . . 39
Controller . . . 40
View . . . 40
Form . . . 40
Helper . . . 40
Domain layer . . . 41
DomainObject . . . 41
Repository . . . 41
Service . . . 41
Infrastructure layer . . . 42
RepositoryImpl . . . 42
O/R Mapper . . . 42
Integration System Connector . . . 42
2.4.2 Dependency between layers . . . 42
2.4.3 Project structure . . . 45
[projectname]-domain . . . 46
[projectname]-web . . . 47
[projectname]-env . . . 49

3 Tutorial (Todo Application) 51
3.1 Introduction . . . 51
3.1.1 Points to study in this tutorial . . . 51
3.1.2 Target readers . . . 51
3.1.3 Verification environment . . . 51
3.2 Description of application to be created . . . 52
3.2.1 Overview of application . . . 52
3.2.2 Business requirements of application . . . 52
3.2.3 Screen transition of application . . . 53
Show all TODO . . . 53

Create TODO . . . 53
Finish TODO . . . 53
Delete TODO . . . 54
3.2.4 Error message list . . . 54
3.3 Environment creation . . . 54
3.3.1 Project creation . . . 54
3.3.2 Maven settings . . . 56
3.3.3 Project configuration . . . 60
3.3.4 Creation of configuration file . . . 60
web.xml settings . . . 60
Settings of common JSP . . . 64
Settings of Bean definition file . . . 64
applicationContext.xml . . . 65
todo-domain.xml . . . 67
todo-infra.xml . . . 68
spring-mvc.xml . . . 69
logback.xml settings . . . 71
3.3.5 Operation verification . . . 74
3.4 Creation of Todo application . . . 77
3.4.1 Creation of Domain layer . . . 78
Creation of Domain Object . . . 78
Repository creation . . . 80
Creation of RepositoryImpl (Infrastructure layer) . . . 81
Service creation . . . 83
Creation of JUnit for Service . . . 88
3.4.2 Creation of application layer . . . 88
Creation of Controller . . . 88
Show all TODO . . . 89
Create TODO . . . 93
Finish TODO . . . 100
Delete TODO . . . 106
3.5 Change of infrastructure layer . . . 113
3.5.1 Common settings . . . 113
Modifications in pom.xml . . . 113
Definition of data source . . . 114
Modifications in todo-infra.xml . . . 114
Creation of todo-env.xml . . . 114
todo-infra.properties . . . 115
Modifications in todo-domain.xml . . . 115
Modifications in TodoServiceImpl . . . 116
3.5.2 Use Spring Data JPA . . . 118
Modifications in configuration file for using Spring Data JPA . . . 118

Modifications in pom.xml . . . 118
Modifications in todo-infra.xml . . . 118
Modifications in todo-env.xml . . . 120
Modifications in spring-mvc.xml . . . 121
Modifications in logback.xml . . . 122
Settings of Entity . . . 123
Modifications in TodoRepository . . . 125
Modifications in TodoRepositoryImpl . . . 126
3.5.3 Use TERASOLUNA DAO . . . 126
Setting for using the TERASOLUNA DAO . . . 127
Modifications in pom.xml . . . 127
Modifications in todo-infra.xml . . . 127
Modifications in todo-env.xml . . . 128
Modifications in logback.xml . . . 129
Creation of sqlMapConfig . . . 130
Modifications in RepositoryImpl . . . 130
Create SQLMap file . . . 133
3.6 In the end... . . . 135

4 Application Development using TERASOLUNA Global Framework 137
4.1 Domain Layer Implementation . . . 137
4.1.1 Roles of domain layer . . . 137
4.1.2 Flow of development of domain layer . . . 138
4.1.3 Implementation of Entity . . . 140
Policy of creating Entity class . . . 140
Example of creating Entity class . . . 141
Table structure . . . 141
Entity structure . . . 143
4.1.4 Implementation of Repository . . . 147
Roles of Repository . . . 147
Structure of Repository . . . 147
Creation of Repository . . . 150
Example of creating Repository . . . 150
Structure of Repository . . . 151
Definition of Repository interface . . . 151
Creation of Repository interface . . . 151
Method definition of Repository interface . . . 153
Creation of RepositoryImpl . . . 156
4.1.5 Implementation of Service . . . 156
Roles of Service . . . 156
Structure of Service class . . . 158
Reason for separating Service and SharedServic . . . 160

Reason for prohibiting the calling of other Service classes from Service class . . . 160
Regarding interface and base classes to limit signature of method . . . 161
Patterns of creating service class . . . 161
Image of Application development - Creating Service for each Entity - . . . 163
Image of Application development - Creating Service for each use case . . . 164
Image of Application development - Creating Service for each use event . . . 167
Creation of Service class . . . 170
Methods of creating Service class . . . 170
Creation of methods of Service class . . . 172
Regarding arguments and return values of methods of Service class . . . 173
Implementation of SharedService class . . . 175
Creation of SharedService class . . . 175
Creation of SharedService class method . . . 175
Regarding arguments and return values of SharedService class method . . . 175
Implementation of logic . . . 175
Operate on business data . . . 176
Returning messages . . . 176
Returning warning message . . . 177
Notifying business error . . . 178
Notifying system error . . . 179
4.1.6 Regarding transaction management . . . 181
Method of transaction management . . . 181
Declarative transaction management . . . 181
Information required for “Declarative transaction management” . . . 181
Propagation of transaction . . . 183
Way of calling the method which is under transaction control . . . 186
Settings for using transaction management . . . 186
PlatformTransactionManager settings . . . 187
Settings for enabling @Transactional . . . 188
Regarding attributes of <tx:annotation-driven> element . . . 189
4.1.7 Appendix . . . 189
Regarding drawbacks of transaction management . . . 189
Programmatic transaction management . . . 190
Sample of implementation of interface and base classes to limit signature . . . 190
4.1.8 Tips . . . 193
Method of dealing with violation of business rules as field error . . . 193
4.2 Implementation of Infrastructure Layer . . . 194
4.2.1 Implementation of RepositoryImpl . . . 194
Implementing Repository using JPA . . . 194
Implementing Repository using MyBatis2 . . . 195
Implementing Repository to link with external system using RestTemplate . . . 201
4.3 Implementation of Application Layer . . . 202

4.3.1 Implementing Controller . . . 202
Creating Controller class . . . 204
Mapping request and processing method . . . 204
Mapping with request path . . . 206
Mapping by HTTP method . . . 207
Mapping by request parameter . . . 208
Mapping using request header . . . 208
Mapping using Content-Type header . . . 209
Mapping using Accept header . . . 209
Mapping request and processing method . . . 209
Overview of sample application . . . 210
Request URL . . . 210
Mapping request and processing method . . . 212
Implementing form display . . . 215
Implementing the display of user input confirmation screen . . . 217
Implementing ‘redisplay of form’ . . . 220
Implementing ‘create new user’ business logic . . . 223
Implementing notification of create new user process completion . . . 224
Placing multiple buttons on HTML form . . . 226
Source code of controller of sample application . . . 227
Regarding arguments of processing method . . . 228
Passing data to screen (View) . . . 230
Retrieving values from URL path . . . 232
Retrieving request parameters individually . . . 233
Retrieving request parameters collectively . . . 235
Performing input validation . . . 237
Passing data while redirecting request . . . 238
Passing request parameters to redirect destination . . . 241
Inserting values in redirect destination URL path . . . 242
Acquiring values from Cookie . . . 243
Writing values in Cookie . . . 243
Retrieving pagination information . . . 244
Retrieving uploaded file . . . 244
Displaying result message on the screen . . . 245
Regarding return value of processing method . . . 245
HTML response . . . 245
Responding to downloaded data . . . 247
Implementing the process . . . 249
Correlation check of input value . . . 250
Calling business logic . . . 251
Reflecting values to domain object . . . 252
Reflecting values to form object . . . 254

4.3.2 Implementing form object . . . 255
Creating form object . . . 256
Number format conversion of fields . . . 257
Date and time format conversion of fields . . . 258
DataType conversion in controller . . . 259
Specifying annotation for input validation . . . 260
Initializing form object . . . 260
Binding to HTML form . . . 263
Binding request parameters . . . 263
Determining binding result . . . 264
4.3.3 Implementing View . . . 265
Implementing JSP . . . 266
Creating common JSP for include . . . 267
Displaying value stored in model . . . 269
Displaying numbers stored in model . . . 270
Displaying date and time stored in model . . . 270
Binding form object to HTML form . . . 271
Displaying input validation errors . . . 272
Displaying message of processing result . . . 272
Displaying codelist . . . 273
Displaying fixed text content . . . 273
Switching display according to conditions . . . 274
Repeated display of collection elements . . . 275
Displaying link for pagination . . . 276
Switching display according to authority . . . 276
Implementing JavaScript . . . 277
Implementing style sheet . . . 277
4.3.4 Implementing common logic . . . 277
Implementing common logic to be executed before and after calling controller . . . 278
Implementing Servlet Filter . . . 278
Implementing HandlerInterceptor . . . 280
Implementing common processes of controller . . . 281
Implementing HandlerMethodArgumentResolver . . . 281
Implementing “@ControllerAdvice” . . . 283
4.3.5 Prevention of double submission . . . 286
4.3.6 Usage of session . . . 286

5 Architecture in Detail - TERASOLUNA Global Framework 289
5.1 [coming soon] Database Access (Common) . . . 289
5.2 [coming soon] Database Access (JPA) . . . 290
5.3 [coming soon] Database Access (MyBatis2) . . . 291
5.4 [coming soon] Exclusive Control . . . 292

5.5 Input Validation . . . 293
5.5.1 Overview . . . 293
Classification of input validation . . . 293
5.5.2 How to use . . . 294
Single item check . . . 294
Basic single item check . . . 294
Single item check of nested Bean . . . 305
Grouped validation . . . 316
Correlation item check . . . 327
Correlation item check implementation using Spring Validator . . . 328
implementation of input check of correlated items using Bean Validation . . . 333
Definition of error messages . . . 333
Messages to be defined in ValidationMessages.properties . . . 335
Messages to be defined in application-messages.properties . . . 337
5.5.3 How to extend . . . 338
Creation of Bean Validation annotation by combining existing rules . . . 339
Creation of Bean Validation annotation by implementing new rules . . . 343
Rules that cannot be implemented by combining the existing rules . . . 344
Check rules for correlated items . . . 346
Business logic check . . . 351
5.5.4 Appendix . . . 353
Input validation rules provided by Hibernate Validator . . . 353
JSR-303 check rules . . . 353
Hibernate Validator check rules . . . 355
Default messages provided by Hibernate Validator . . . 355
Type mismatch . . . 356
Binding null to blank string field . . . 358
5.6 [coming soon] Logging . . . 360
5.7 Exception Handling . . . 361
5.7.1 Overview . . . 361
Classification of exceptions . . . 361
Exception handling methods . . . 362
5.7.2 Detail . . . 366
Types of exceptions . . . 366
Business exception . . . 367
Library exceptions that occurs during normal operation . . . 368
System Exception . . . 369
Unexpected System Exception . . . 370
Fatal Errors . . . 371
Framework exception in case of invalid requests . . . 371
Exception Handling Patterns . . . 372
When notifying partial redo of a use case (from middle) . . . 374

When notifying redo of a use case (from beginning) . . . 374
When notifying that the system or application is not in a normal state . . . 375
When notifying that the request contents are invalid . . . 376
When a fatal error has been detected . . . 376
When notifying that an exception has occurred in the presentation layer (JSP etc.) . . . 377
Basic Flow of Exception Handling . . . 377
Basic flow when the Controller class handles the exception at request level . . . 378
Basic flow when the Controller class handles the exception at use case level . . . 379
Basic flow when the framework handles the exception at servlet level . . . 381
Basic flow when the servlet container handles the exception at web application level 382
5.7.3 How to use . . . 383
Application Settings . . . 383
Common Settings . . . 383
Domain Layer Settings . . . 390
Application Layer Settings . . . 390
Servlet Container Settings . . . 396
Coding Points (Service) . . . 398
Generating Business Exception . . . 398
Generating System Exception . . . 401
Catch the exception to continue the execution . . . 403
Coding Points (Controller) . . . 405
Method to handle exceptions at request level . . . 405
Method to handle exception at use case level . . . 406
Coding points (JSP) . . . 408
Method to display messages on screen using MessagesPanelTag . . . 409
Method to display system exception code on screen . . . 409
5.7.4 How to use (Ajax) . . . 410
5.7.5 Appendix . . . 411
Exception handling classes provided by the common library . . . 411
About SystemExceptionResolver settings . . . 414
Attribute name of result message . . . 416
Attribute name of exception code (message ID) . . . 417
Header name of exception code (message ID) . . . 418
Attribute name of exception object . . . 419
Cache control flag of HTTP response . . . 420
About HandlerExceptionResolverLoggingInterceptor settings . . . 420
List of exception classes to be excluded from scope of logging . . . 421
HTTP response code set by DefaultHandlerExceptionResolver . . . 422
5.8 [coming soon] Session Management . . . 424
5.9 Message Management . . . 425
5.9.1 Overview . . . 425
Types of messages . . . 425

Types of messages depending on patterns . . . 426
Message ID . . . 428
Title . . . 428
Labels . . . 429
Result messages . . . 431
Input validation error message . . . 435
5.9.2 How to use . . . 435
Display of messages set in properties file . . . 435
Settings at the time of using properties . . . 435
Display of messages set in properties . . . 436
Display of result messages . . . 438
Using basic result messages . . . 438
Specifying attribute name of result messages . . . 444
Displaying business exception messages . . . 446
5.9.3 How to extend . . . 448
Creating independent message types . . . 448
5.9.4 Appendix . . . 449
Changing attribute of <t:messagesPanel> tag . . . 449
Display of result message wherein ResultMessages is not used . . . 452
Auto-generation tool of message key constant class . . . 455
5.10 [coming soon] Property Management . . . 458
5.11 [coming soon] Pagination . . . 459
5.12 [coming soon] Double Submit Protection . . . 460
5.13 [coming soon] Internationalization . . . 461
5.14 [coming soon] CodeList . . . 462
5.15 [coming soon] Ajax . . . 463
5.16 [coming soon] RESTful Web Service . . . 464
5.17 [coming soon] File Upload . . . 465
5.18 [coming soon] File Download . . . 466
5.19 [coming soon] Screen Layout using Tiles . . . 467
5.20 [coming soon] System Date . . . 468
5.21 Utilities . . . 469
5.21.1 [coming soon] Bean Mapping (Dozer) . . . 469
5.21.2 Date Operations (Joda Time) . . . 470
Overview . . . 470
How to use . . . 470
Fetching date . . . 470
Type conversion . . . 474
Date operations . . . 475
Fetching the duration . . . 478
JSP Tag Library . . . 480
Example (display of calendar) . . . 483

6 Security for TERASOLUNA Global Framework 487
6.1 [coming soon] Spring Security Overview . . . 487
6.2 [coming soon] Spring Security Tutorial . . . 488
6.3 [coming soon] Authentication . . . 489
6.4 [coming soon] Password Hashing . . . 490
6.5 [coming soon] Authorization . . . 491
6.6 XSS Countermeasures . . . 492
6.6.1 Overview . . . 492
Stored & Reflected XSS Attacks . . . 492
6.6.2 How to use . . . 493
Output Escaping . . . 493
Example of vulnerability when output values are not escaped . . . 493
Example of escaping output value using f:h() function . . . 495
JavaScript Escaping . . . 496
Example of vulnerability when output values are not escaped . . . 497
Example of escaping output value using f:js() function . . . 498
Event handler Escaping . . . 498
Example of vulnerability when output values are not escaped . . . 499
Example of escaping output value using f:hjs() function . . . 499
6.7 [coming soon] CSRF(Cross Site Request Forgeries) Countermeasures . . . 501
7 Appendix 503
7.1 [coming soon] Create Project from Blank Project . . . 503
7.2 [coming soon] Maven Repository Management using Nexus . . . 504
7.3 [coming soon] Exclude Environmental Dependencies . . . 505
7.4 [coming soon] Project Structure Standard . . . 506
7.5 Reference Books . . . 507
7.6 [coming soon] Spring Comprehension Check . . . 508

Note: This guideline has been released for Public Review. Be noted that the contents and structure might change in the coming versions.

If there are any mistakes in the contents or any comments related to the contents, please raise them at Issues on Github.

1

In the Beginning

1.1 Terms of Use

In order to use this document, you are required to agree to abide by the following terms. If you do not agree with the terms, you must immediately delete or destroy this document and all its duplicate copies.

1. Copyrights and all other rights of this document shall belong to NTT DATA or third party possessing such rights.
2. This document may be reproduced, translated or adapted, in whole or in part for personal use. However, deletion of the terms given on this page and copyright notice of NTT DATA is prohibited.
3. This document may be changed, in whole or in part for personal use. Creation of secondary work using this document is allowed. However, “ Reference document: TERASOLUNA Global Framework Development Guideline ” or equivalent documents may be mentioned in created document and its duplicate copies.
4. Document and its duplicate copies created according to Clause 2 may be provided to third party only if these are free of cost.
5. Use of this document and its duplicate copies, and transfer of rights of this contract to a third party, in whole or in part, beyond the conditions specified in this contract, are prohibited without the written consent of NTT Data.
6. NTT DATA shall not bear any responsibility regarding correctness of contents of this document, warranty of fitness for usage purpose, assurance for accuracy and reliability of usage result, liability for defect warranty, and any damage incurred directly or indirectly.
7. NTT DATA does not guarantee the infringement of copyrights and any other rights of third party through this document. In addition to this, NTT DATA shall not bear any responsibility regarding any claim (Including the claims occurred due to dispute with third party) occurred directly or indirectly due to infringement of copyright and other rights.

Registered trademarks or trademarks of company name and service name, and product name of their respective companies used in this document are as follows.

1.2 This document covers the following

This guideline provides best practices to develop highly maintainable Web applications using full stack framework focussing on Spring, Spring MVC and JPA, MyBatis.

This guideline helps to proceed with the software development (mainly coding) smoothly.

1.3 Target readers of this document

This guideline is written for the architects and programmers having software development experience and knowledge of the following.

- Basic knowledge of DI and AOP of Spring Framework
- Web development experience using Servlet/JSP
- Knowledge of SQL
- Experience of configuration management with Maven

This guideline is not for beginners.

In order to check whether the readers have basic knowledge of Spring Framework, refer to *[coming soon] Spring Comprehension Check*. It is recommended to study the following books if one is not able to answer 40% of the comprehension test.

- 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）[Japanese]
- Pro Spring 3 (Apress)

1.4 Structure of this document

- *Summary - Architecture of TERASOLUNA Global Framework* Overview of Spring MVC and basic concepts of TERASOLUNA Global Framework is explained.
- *Tutorial (Todo Application)* Experience in application development using TERASOLUNA Global Framework through simple application development.
- *Application Development using TERASOLUNA Global Framework* Knowledge and methods for application development using TERASOLUNA Global Framework are explained.
- *Architecture in Detail - TERASOLUNA Global Framework* Method to implement the functions required for general application development using TERASOLUNA Global Framework or features of each function is explained.

- *Security for TERASOLUNA Global Framework* Security measures are explained focusing on Spring Security.
- *Appendix* Describing the additional information when TERASOLUNA Global Framework is being used.

1.5 Reading this document

Firstly read "*Summary - Architecture of TERASOLUNA Global Framework*".

Implement "*First application based on Spring MVC*" for beginners of Spring MVC.

Read "*Application Layering*" as the terminology and concepts used in this guideline are explained here.

Then continue with "*Tutorial (Todo Application)*".

Get a feel of application development using TERASOLUNA Global

Framework by firstly trying it as per the proverb "Practice makes perfect".

Use this tutorial to study the details of application development in "*Application Development using TERASOLUNA Global Framework*".

Since the knowhow for development is explained using Spring MVC in "*Implementation of Application Layer*", it is recommended to read it again and again several times.

One can get a better understanding by studying "*Tutorial (Todo Application)*" once again after reading this chapter.

It is strongly recommended that all the developers who use TERASOLUNA Global Framework read it.

Refer to "*Architecture in Detail - TERASOLUNA Global Framework*" and "*Security for TERASOLUNA Global Framework*"

as per the requirement. However, read "*Input Validation*" since it is normally required for application development.

The technical leader understands all the contents and checks the type of policy to be decided in the project.

Note: If you do not have sufficient time, first go through the following.

1. *First application based on Spring MVC*
2. *Application Layering*
3. *Tutorial (Todo Application)*

4. *Application Development using TERASOLUNA Global Framework*
5. *Tutorial (Todo Application)*
6. *Input Validation*

1.6 Criterion-based mapping of guideline

The chapter 4 of this guideline is structured functionality wise. This section shows a mapping from a point of view other than functionality. It indicates which part of guideline contains which type of content.

1.6.1 Mapping based on security measures

Using OWASP Top 10 for 2013 as an axis, links to explanation of functionalities related to security have been given

No.	Item Name	Correcsponding Guideline
A1	Injection SQL Injection	• *[coming soon] Database Access (JPA)* • *[coming soon] Database Access (MyBatis2)* (Details about using bind variable at the time of placeholders for query parameters)
A1	Injection XXE(XML External Entity) Injection	• *[coming soon] Ajax*
A2	Broken Authentication and Session Management	• *[coming soon] Authentication*
A3	Cross-Site Scripting (XSS)	• *XSS Countermeasures*
A4	Insecure Direct Object References	No mention in particular
A5	Security Misconfiguration	• *[coming soon] Logging*(Mention about message contents of log) • *Method to display system exception code on screen*(Mention about message output at the time of system exception)
A6	Sensitive Data Exposure	• *[coming soon] Property Management* • *[coming soon] Password Hashing* (Mention about password hash only)
A7	Missing Function Level Access Control	• *[coming soon] Authorization*
A8	Cross-Site Request Forgery (CSRF)	• *[coming soon] CSRF(Cross Site Request Forgeries) Countermeasures*
A9	Using Components with Known Vulnerabilities	No mention in particular
A10	Unvalidated Redirects and Forwards	• *[coming soon] Authentication*(Mention about Open Redirect Vulnerability measures)

1.7 Change Log

Modified on	Modified locations	Modification details
2014-08-27	-	Released "1.0.1 RELEASE" version For details, refer to Issue list of 1.0.1.
	Overall modifications	Fixed guideline errors (corrected typos, mistakes in description etc.) For details, refer to Issue list of 1.0.1.
	Japanese version	Added Japanese version of the following. • *Criterion-based mapping of guideline* • *[coming soon] RESTful Web Service* • `Tutorial (Todo Application for REST)`
	English version	Added English version of the following. • *In the Beginning* • *Summary - Architecture of TERASOLUNA Global Framework* • *Tutorial (Todo Application)* • *Application Development using TERASOLUNA Global Framework* • *Input Validation* • *Exception Handling* • *Message Management* • `../ArchitectureInDetail/utilities/JodaTime` • *XSS Countermeasures* • *Reference Books*
	Stack of TERASOLUNA Global Framework	Updated the OSS version in accordance with bug fixes. • GroupId (`org.springframework`) updated to 3.2.10.RELEASE from 3.2.4.RELEASE • GroupId (`org.springframework.data`)/ArtifactId(`spring-da` updated to 1.6.4.RELEASE from 1.6.1.RELEASE • GroupId (`org.springframework.data`)/ArtifactId(`spring-da` updated to 1.4.3.RELEASE from 1.4.1.RELEASE • GroupId (`org.aspectj`) updated to 1.7.4 from 1.7.3 • Deleted GroupId (`javax.transaction`)/ArtifactId(`jta`)
	Implementation of Application Layer Japanese version *Message Management*	Added a warning about CVE-2014-1904(XSS Vulnerability of `action` attribute in `<form:form>` tag) Added description about bug fix • Fixed bugs of `<t:messagesPanel>` tag of common library (terasoluna-gfw#10)
	Japanese version *[coming soon] Pagination*	Updated description about bug fix • Fixed bugs of `<t:pagination>` tag of common library (terasoluna-gfw#12) • Fixed bugs of Spring Data Commons (terasoluna-gfw#22)

2

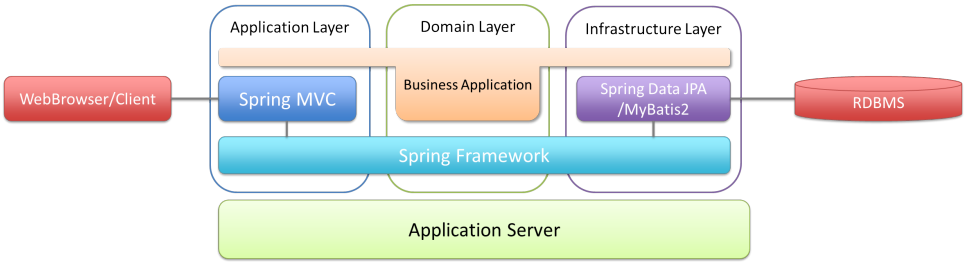
Summary - Architecture of TERASOLUNA Global Framework

The architecture adopted in this guideline is explained here.

2.1 Stack of TERASOLUNA Global Framework

2.1.1 Summary of Software Framework of TERASOLUNA Global Framework

Software Framework being used in TERASOLUNA Global Framework is not a proprietory Framework but a combination of various OSS technologies around Spring Framework.



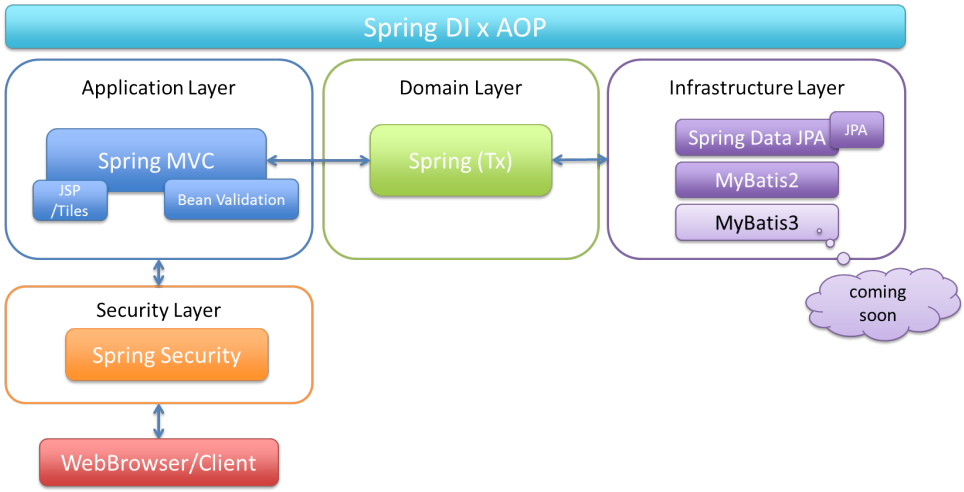
2.1.2 Main Structural Elements of Software Framework

Libraries which constitute TERASOLUNA Global Framework are as follows:

DI Container

Spring is used as DI Container.

- Spring Framework 3.2



MVC Framework

Spring MVC is used as Web MVC Framework.

- Spring MVC 3.2

O/R Mapper

This guideline assumes the use of **any one of the below**.

- JPA2.0
  - Hibernate 4.2 is used as provider.
- MyBatis 2.3.5
  - DAO(TERASOLUNA DAO) of TERASOLUNA Framework is used as wrapper.

Todo

MyBatis 3 is planned to be included here.

Note: To be precise MyBatis is "SQL Mapper", but it is classified as "O/R Mapper" in this guidelines.

Warning: Not every project must adopt JPA. For situations in which table design has been done and "Most of the tables are not normalized", "The number of columns in the table is too large" etc, use of JPA is difficult. Further, this guideline does not explain the basic usage of JPA. Hence, it is pre-requisite to have JPA experience people in the team.

View

JSP is used as View.

For using Tiled JSP, use the following.

- Apache Tiles 2.2

Security

Spring Security is used as the framework for Authentication and Authorization.

- Spring Security 3.1

Todo

Update to Spring Security 3.2 is planned in future.

Validation

- For Single item input check, BeanValidation 1.0 is used.
  - For implementation, Hibernate Validator 4.3is used.
- For correlated items check, BeanValidation or Spring Validation
  - Refer to *Input Validation* for determining which of the two is to be used in which sitation.

Logging

- for Logger API, SLF4J is used.
  - For implementation of Logger, Logback is used.

Common Library

- https://github.com/terasolunaorg/terasoluna-gfw
- Refer to *Building blocks of Common Library* for details.

2.1.3 OSS Versions

List of OSS being used in version 1.0.1.RELEASE.

Type	GroupId	ArtifactId	Version	Remarks
Spring	org.springframework	spring-aop	3.2.10.RELEASE	
Spring	org.springframework	spring-aspects	3.2.10.RELEASE	
Spring	org.springframework	spring-beans	3.2.10.RELEASE	
Spring	org.springframework	spring-context	3.2.10.RELEASE	
Spring	org.springframework	spring-context-support	3.2.10.RELEASE	
Spring	org.springframework	spring-core	3.2.10.RELEASE	
Spring	org.springframework	spring-expression	3.2.10.RELEASE	
Spring	org.springframework	spring-jdbc	3.2.10.RELEASE	
Spring	org.springframework	spring-orm	3.2.10.RELEASE	
Spring	org.springframework	spring-tx	3.2.10.RELEASE	
Spring	org.springframework	spring-web	3.2.10.RELEASE	
Spring	org.springframework	spring-webmvc	3.2.10.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	
AOP	aopalliance	aopalliance	1	
AOP	org.aspectj	aspectjrt	1.7.4	
AOP	org.aspectj	aspectjweaver	1.7.4	
Log Output	ch.qos.logback	logback-classic	1.0.13	
Log Output	ch.qos.logback	logback-core	1.0.13	
Log Output	org.lazyluke	log4jdbc-remix	0.2.7	

Continued on Next page

Table. 2.1 – continued from previous page

Type	GroupId	ArtifactId	Version	Remarks
Log Output	org.slf4j	jcl-over-slf4j	1.7.5	
Log Output	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	
Input check	javax.validation	validation-api	1.0.0.GA	
Input check	org.hibernate	hibernate-validator	4.3.1.Final	
Input check	org.jboss.logging	jboss-logging	3.1.0.GA	
Bean conversion	commons-beanutils	commons-beanutils	1.8.3	*3
Bean conversion	net.sf.dozer	dozer	5.4.0	*3
Bean conversion	org.apache.commons	commons-lang3	3.1	*3
Date conversion	joda-time	joda-time	2.2	
Date conversion	joda-time	joda-time-jsptags	1.1.1	*3
Date conversion	org.jadira.usertype	usertype.core	3.0.0.GA	*1
Date conversion	org.jadira.usertype	usertype.spi	3.0.0.GA	*1
Connection pool	commons-dbcp	commons-dbcp	1.2.2.patch_DBCP264_DBCP372	*2
Connection pool	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
Utility	com.google.guava	guava	13.0.1	
Utility	commons-collections	commons-collections	3.2.1	*3
Utility	commons-io	commons-io	2.4	*3
Servlet	javax.servlet	jstl	1.2	

1. Dependent libraries, when JPA is used for data access.
2. Dependent libraries, when MyBatis2 is used for data access.
3. Libraries which are not dependent on Common Library, but recommended in case of application development using TERASOLUNA Global Framework.

2.1.4 Building blocks of Common Library

Common Library includes `+ alpha` functionalities which are not available in Spring Ecosystem or other dependent libraries included in TERASOLUNA Global Framework. Basically, application development is possible using TERASOLUNA Global Framework even without this library. It is a “nice to have” kind of existence.

No.	Project Name	Summary	Java source-code availability
(1)	terasoluna-gfw-common	general-purpose functionality irrespective of Web	Yes
(2)	terasoluna-gfw-web	Group of functionalities for creating web application	Yes
(3)	terasoluna-gfw-jpa	Dependency definition for using JPA	No
(4)	terasoluna-gfw-mybatis2	Dependency definition for using MyBatis2	No
(5)	terasoluna-gfw-security-core	Dependency definition for using Spring Security (other than Web).	No
(6)	terasoluna-gfw-security-web	Dependency definition for using Spring Security (related to Web) and extended classes of Spring Security.	Yes

The project which does not contain the Java source code, only defines library dependencies.

terasoluna-gfw-common

- Common exception mechanism
  - Exception Class
  - Exception Logger
  - Exception Code
  - Exception Logging Mechanism
- System Date
- CodeList

- Message containing processing result
- Query (SQL, JPQL) Escape
- Sequencer

terasoluna-gfw-web

- Transaction token mechanism
- Common exception handler
- Populate CodeList interceptor
- General Download View
- group of servlet filters for MDC log output
  - Parent servlet filter
  - Servlet filter for Tracking ID log output
  - Servlet filter for MDC clear
- EL function group
  - XSS counter-measures
  - URL Encoding
  - converting JavaBeans properties to query string
- JSP Tag for pagination
- JSP Tag to display result message

terasoluna-gfw-security-web

- Servlet filter for logging authenticated username
- Redirect handler for open redirect vulnerability
- CSRF counter measures (Interim measure until the introduction of Spring Security 3.2)

2.2 Overview of Spring MVC Architecture

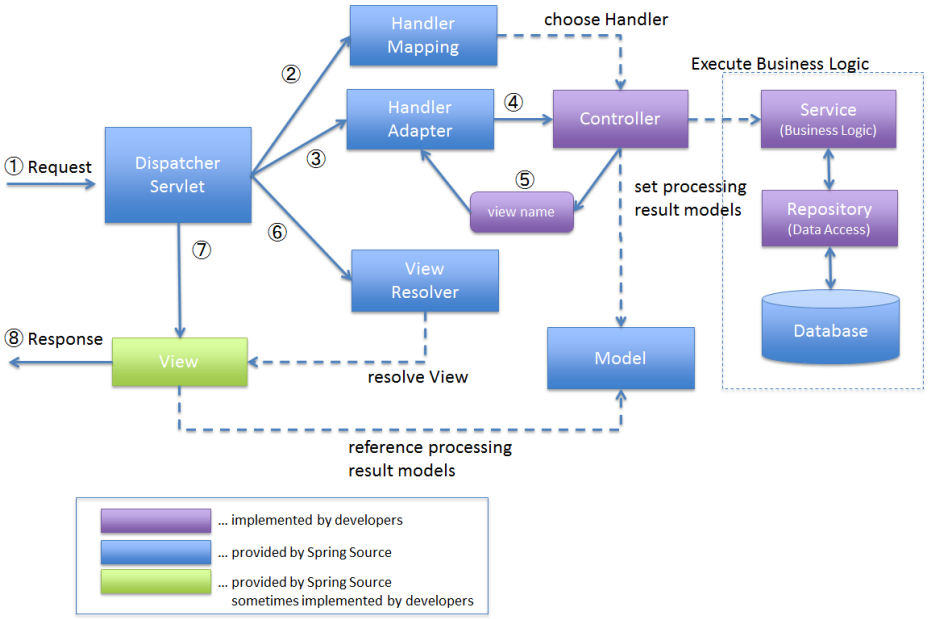
Official website of Spring MVC says the following

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

The processing flow of Spring MVC from receiving the request till the response is returned is shown in the following diagram.



1. `DispatcherServlet` receives the request.
2. `DispatcherServlet` dispatches the task of selecting an appropriate `controller` to `HandlerMapping`. `HandlerMapping` selects the controller which is mapped to the incoming request URL and returns the `(selected Handler)` and `Controller` to `DispatcherServlet`.
3. `DispatcherServlet` dispatches the task of executing of business logic of `Controller` to `HandlerAdapter`.
4. `HandlerAdapter` calls the business logic process of `Controller`.
5. `Controller` executes the business logic, sets the processing result in `Model` and returns the logical name of view to `HandlerAdapter`.
6. `DispatcherServlet` dispatches the task of resolving the `View` corresponding to the View name to `ViewResolver`. `ViewResolver` returns the `View` mapped to View name.

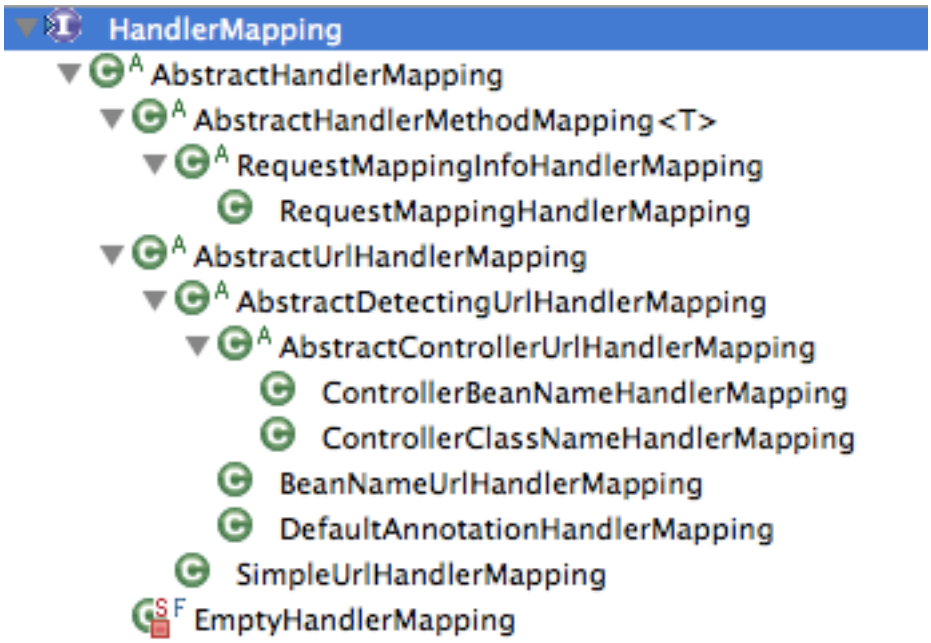
7. `DispatcherServlet` dispatches the rendering process to returned `View`.
8. `View` renders `Model` data and returns the response.

2.2.2 Implementation of each component

Among the components explained previously, the extendable components are implemented.

Implementation of HandlerMapping

Class hierarchy of `HandlerMapping` provided by Spring framework is shown below.



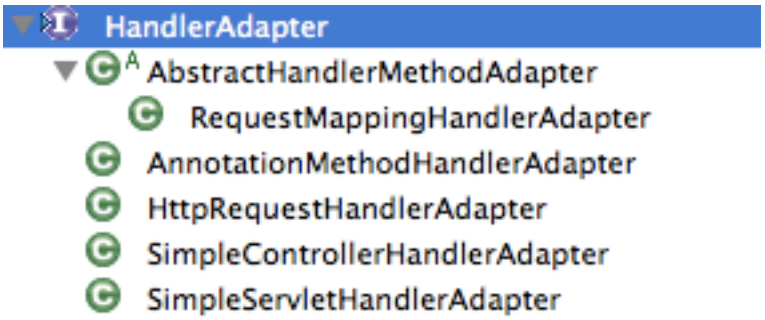
Usually `org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping` is used. This class reads `@RequestMapping` annotation from the `Controller` and uses the method of `Controller` that matches with URL as Handler class.

In Spring3.1, `RequestMappingHandlerMapping` is enabled by default when `<mvc:annotation-driven>` is set in Bean definition file read by `DispatcherServlet`. (For the settings which get enabled with the use of `<mvc:annotation-driven>` annotation, refer Web MVC framework.)

Implementation of HandlerAdapter

Class hierarchy of `HandlerAdapter` provided by Spring framework is shown below.

Usually `org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter`



is used. `RequestMappingHandlerAdapter` class calls the method of handler class (`Controller`) selected by `HandlerMapping`. In Spring 3.1, this class is also configured by default throught `<mvc:annotation-driven>`.

Implementation of ViewResolver

Classes that implement `ViewResolver` provided by Spring framework and dependent libraries are shown below.

Normally (When JSP is used),

- `org.springframework.web.servlet.view.InternalResourceViewResolver` is used,

however, when template engine Tiles is to be used

- `org.springframework.web.servlet.view.tiles2.TilesViewResolver`

and when stream is to be returned for file download

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

Thereby, it is required to use different viewResolver based on the type of the View that needs to be returned.

When `View` of multiple types is to be handled, multiple definitions of `ViewResolver` are required.

A typical example of using multiple `ViewResolver` is the screen application for which file download process exists.

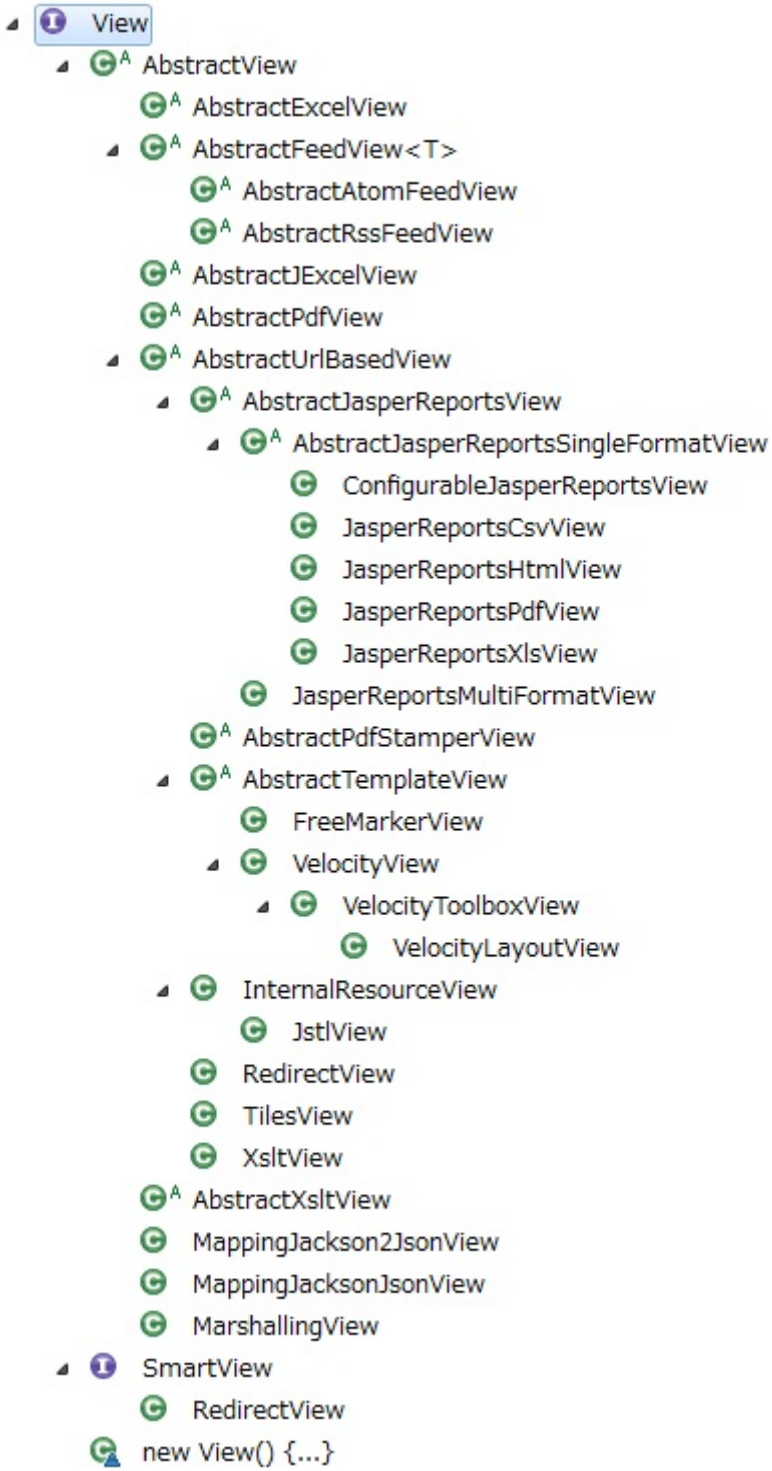
For screen (JSP), `View` is resolved using `InternalResourceViewResolver` and for File download `View` is resolved using `BeanNameViewResolver`.

For details, refer *[coming soon] File Download*.

Implementation of View

Classes that implement `View` provided by Spring framework and its dependent libraries are shown below.

`View` changes with the type of response to be returned. When JSP is to be returned, `org.springframework.web.servlet.view.JstlView` is used. When `View` not provided by Spring framework and its dependent libraries are to be handled, it is necessary to extend the class in which `View` interface is implemented. For details, refer *[coming soon] File Download*.


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() {...}

2.3 First application based on Spring MVC

Before entering into the advanced usage of Spring MVC, it is better to understand Spring MVC by actually trying handson web application development using Spring MVC. This purpose of this chapter to understand the overall picture. **Note that it does not follow the recommendations given from the next chapter onwards. (for the ease of understanding)**.

2.3.1 Prerequisites

The description of this chapter has been verified on the following environment. (For other environments, replace the contents mentioned here appropriately)

Product	Version
JDK	1.6.0_33
Spring Tool Suite (STS)	3.2.0
VMware vFabric tc Server Developer Edition	2.8
Fire Fox	21.0

Note: To connect to the internet via a proxy server, STS Proxy settings and Maven Proxy settings are required for the following operations.

Warning: "Spring Template Project" used in this chapter has been removed from Spiring Tool Suite 3.4; hence, the contents of this chapter can only be confirmed on "Spring Tool Suite 3.3" or before.
Refer to *[coming soon] Create Project from Blank Project* in case using Spring Tool Suite 3.4.
We plan to update the contents of this chapter in future.

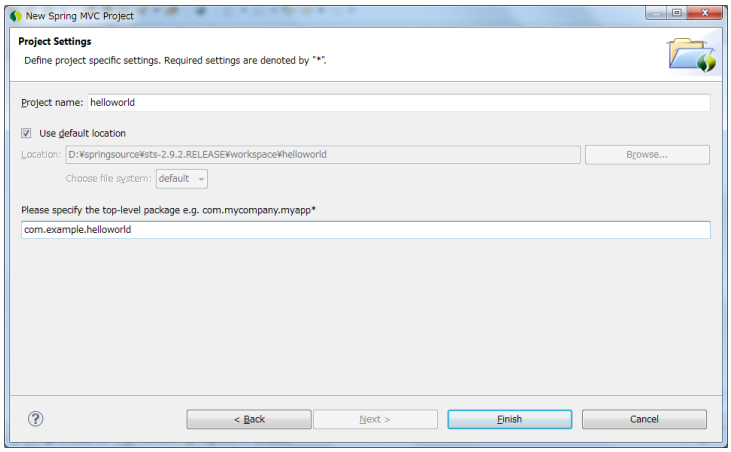
2.3.2 Create a New Project

From the menu of SpringSource Tool Suite, select [File] -> [New] -> [Spring Template Project] -> [Spring MVC Project] -> [Next], and create Spring MVC project.

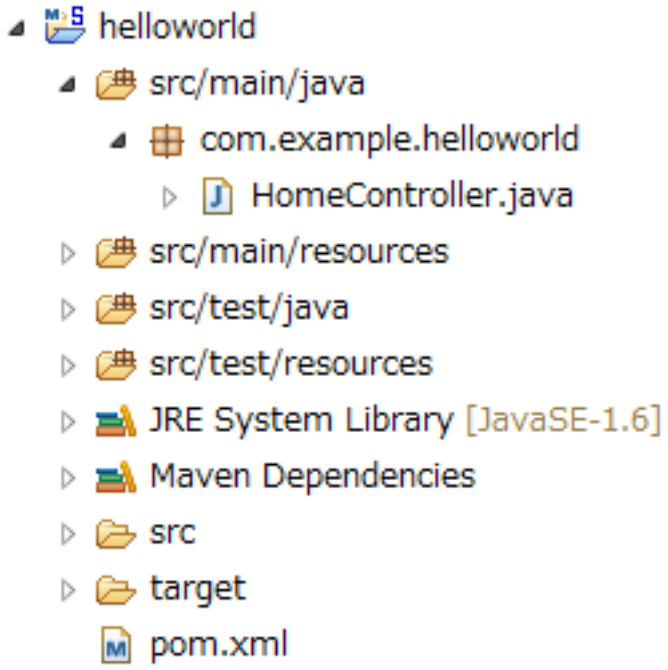
Note: If proxy server is being used, there is a possibility of not being able to select Spring Template Project. In order to resolve this, do the following.

- First select [window] -> [Preferences] -> [Spring] -> [Template Projects] and remove all items except "spring-defaults".
- Then click Apply.
- After this, when [File] -> [New] -> [Spring Project] is clicked, [Spring MVC Project] can be selected from Templetes.

Enter “helloworld” in [Project Name], “com.example.helloworld” in [Please specify the top-level package] and click [Finish] on the next screen.



Following project is generated in the Package Explorer (**Internet access is required**)



The generated configuration file (src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml) of Spring MVC is described briefly to understand the configuration of Spring MVC.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:beans="http://www.springframework
    xmlns:context="http://www.springframework.org/schema/context"
    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/context http://www.springframework.org/schema/contex
```

```
    <!-- DispatcherServlet Context: defines this servlet's request-processing
        infrastructure -->

    <!-- (1) Enables the Spring MVC @Controller programming model -->
    <annotation-driven />

    <!-- Handles HTTP GET requests for /resources/** by efficiently serving
        up static resources in the ${webappRoot}/resources directory -->
    <resources mapping="/resources/**" location="/resources/" />

    <!-- (2) Resolves views selected for rendering by @Controllers to .jsp resources
        in the /WEB-INF/views directory -->
    <beans:bean
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jsp" />
    </beans:bean>

    <!-- (3) -->
    <context:component-scan base-package="com.example.helloworld" />
</beans:beans>
```

Item number	Description
(1)	Default settings of Spring MVC are configured by defining <annotation-driven />. Refer to the official website Enabling the MVC Java Config or the MVC XML Namespace of Spring framework for default configuration.
(2)	Define the location of View by specifying Resolver of View.
(3)	Define the package which will be target of searching components used in Spring MVC.

Following is the `com.example.helloworld.HomeController` (However, it is modified to simple form for explanation).

```
package com.example.helloworld;

import java.util.Date;

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// (1)
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) // (2)
    public String home(Model model) { // (3)
        logger.info("Welcome home!");

        Date date = new Date();
        model.addAttribute("serverTime", date);

        return "home"; // (4)
    }
}
```

It is explained briefly.

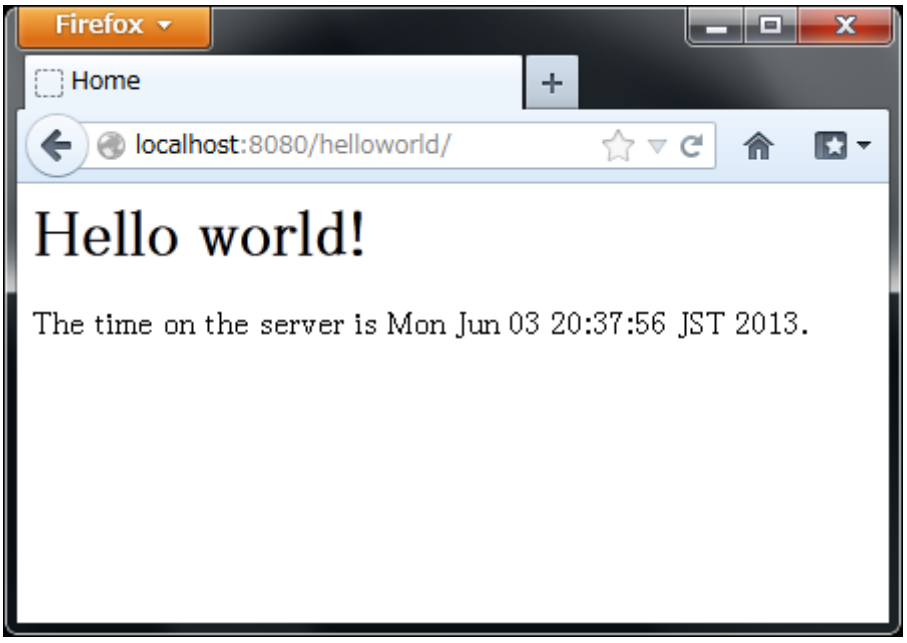
Item number	Description
(1)	It can be read automatically by DI container if `@Controller` annotation is used. As stated earlier in "explanation of Spring MVC configuration files (3)", it is the target of component-scan.
(2)	It gets executed when the HTTP method is GET and when the Resource is (or request URL) is "/".
(3)	Set objects to be delivered to View.
(4)	Return View name. Rendering is performed by `WEB-INF/views/home.jsp` as per the configuration in "Explanation of Spring MVC configuration files (2)".

The object set in Model is set in HttpServletRequest. The value passed from Controller can be output by mentioning `${serverTime}` in home.jsp as shown below. (**However, as described below, it is necessary to perform HTML escaping since there is possibility of ${XXX} becoming a target of XSS**)

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
        <title>Home</title>
</head>
<body>
<h1>
        Hello world!
</h1>

<P>  The time on the server is ${serverTime}. </P>
</body>
</html>
```

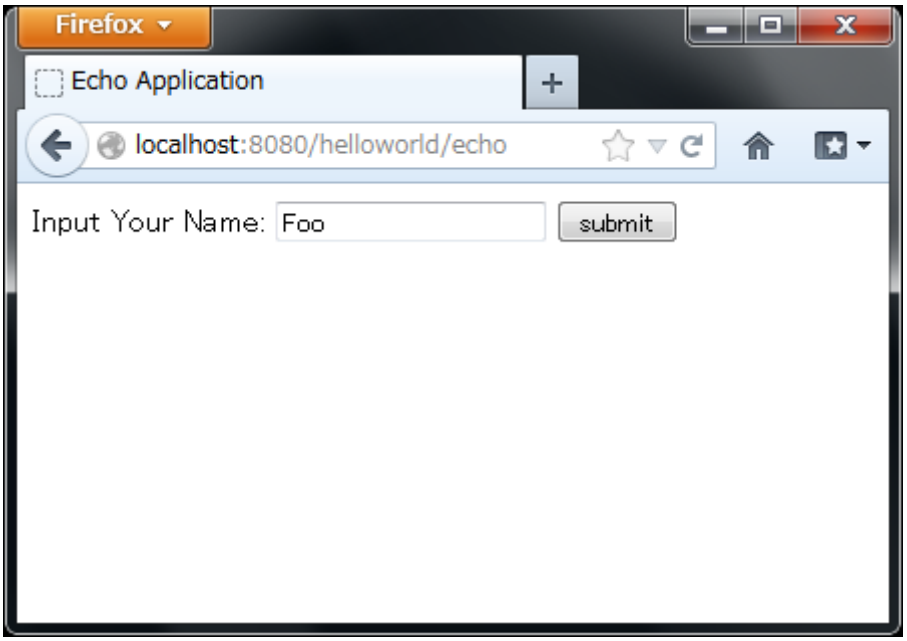
2.3.3 Run on Server

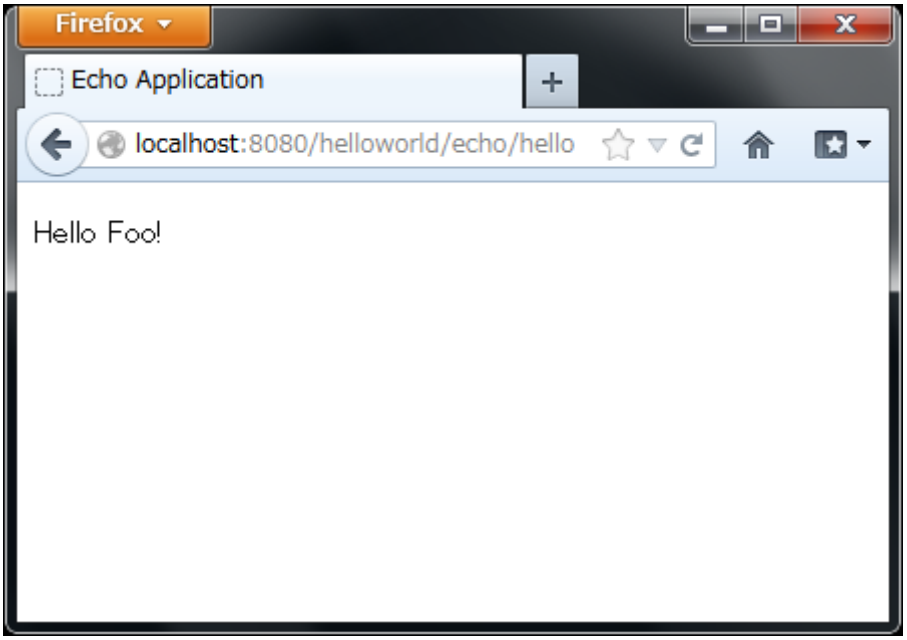
Right click “helloworld” project in STS, and start helloworld project by executing “Run As” -> “Run On Server” -> “localhost” -> “VMware vFabric tc Server Developer Edition v2.8” -> “Finish”. Enter “http://localhost:8080/helloworld/” in browser to display the following screen.



2.3.4 Create an Echo Application

Lets go ahead and reate a simple application. It is a typical eco application in which message will be displayed if name is entered in the text field as given below.





Creating a form object

First create a form object to accept the value of text field. Create `EchoForm` class in `com.example.helloworld.echo` package. It is a simple JavaBean that has only 1 property.

```
package com.example.helloworld.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;
    }
}
```

Create a Controller

Next, create the Controller class. create the `EchoController` class in "com.example.helloworld.echo" package.

```
package com.example.helloworld.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;

@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"; // (4)
    }

    @RequestMapping("hello") // (5)
    public String hello(EchoForm form, Model model) {// (3)
        model.addAttribute("name", form.getName()); // (6)
        return "echo/hello";
    }
}
```

Item number	Description
(1)	Add `@ModelAttribute` annotation to the method. Return value of such a method is automatically added to the Model. Attribute name can be specified in `@ModelAttribute`, but the class name with the first letter in lower case is the default attribute name. In this case it will be `echoForm`. This attribute name must match with the value of `modelAttribute` of `form:form` tag.
(2)	When nothing is specified in `value` attribute of `@RequestMapping` annotation at the method level, it is mapped to `@RequestMapping` added at class level. In this case, `index` method is called, if `<contextPath>/echo` is accessed. When nothing is set in `method` attribute, mapping is done for any HTTP method.
(3)	EchoForm object added to the model in (1) is passed as argument.
(4)	Since `echo/index` is returned as View name, `WEB-INF/views/echo/index.jsp` is rendered by ViewResolver.
(5)	Since `hello` is specified in `@RequestMapping` annotation at method level, if `<contextPath>/echo/hello` is accessed, `hello` method is called.
(6)	`name` entered in form is passed as it is to the View.

Create JSP Files

Finally create JSP of input screen and output screen. Each file path must match with View name as follows.

- Input Screen src/main/webapp/WEB-INF/views/echo/index.jsp

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ page pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<!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>
```

Item number	Description
(1)	HTML form is constructed by using tag library. Specify the name of form object created by Controller in `modelAttribute`. Refer hereforfor tag library.

The generated HTML is as follows

```
<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" />
  </form>
</body>
</html>
```

- Output Screen src/main/webapp/WEB-INF/views/echo/hello.jsp

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ page pageEncoding="UTF-8"%>
```

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<!DOCTYPE html>
<html>
<head>
<title>Echo Application</title>
</head>
<body>
  <p>
    Hello <c:out value="${name}" />!
  </p>
</body>
</html>
```

Output `name` passed from Controller. Take countermeasures for XSS using `c:out` tag.

Note: Countermeasure for XSS are taken using `c:out` standard tag here. However, `f:h()` function that can be used easily is provided in common library. Refer to *XSS Countermeasures* for details.

Implementation of Eco application is completed here. Start the server, access the link `http://localhost:8080/helloworld/echo` to access the application.

Implement Input Validation

Input validation is not performed in this application till this point. In Spring MVC, Bean Validation (JSR-303) and annotation based input validation can be easily implemented. For example input validation of name is performed in Eco Application.

Following dependency is added to pom.xml for using Bean Validation.

```
        <!-- Bean Validation (JSR-303) -->
        <dependency>
            <artifactId>hibernate-validator</artifactId>
            <groupId>org.hibernate</groupId>
            <version>4.3.1.Final</version>
        </dependency>
```

In `EchoForm`, add `@NotNull` annotation and `@Size` annotation to `name` property as follows (can be added to getter method).

```
package com.example.helloworld.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 = 2557725707095364446L;

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

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

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

Item number	Description
(1)	By adding `@NotNull` annotation, whether `name` parameter exists in HTTP request is checked.
(2)	By adding `@Size(min=1, max=5)` annotation, whether the size of `name` is more than or equal to 1 and less than or equal to 5 is checked.

In `EchoController` class, add `@Valid` annotation as shown below and also use of `hasErrors` method.

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

import javax.validation.Valid;

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;

@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("hello")
    public String hello(@Valid EchoForm form, BindingResult result, Model model) { // (1)
        if (result.hasErrors()) { // (2)
            return "echo/index";
        }
        model.addAttribute("name", form.getName());
        return "echo/hello";
    }
}
```

Item number	Description
(1)	In Controller, add `@Valid` annotation to the argument on which validation needs to be executed. Also add `BindingResult` object to arguments. Input validation is automatically performed using Bean Validation and the result is stored in `BindingResult` object.
(2)	It can be checked whether there is error by using `hasErrors` method.

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ page pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<!DOCTYPE html>
<html>
<head>
<title>Echo Application</title>
</head>
<body>
  <form:form modelAttribute="echoForm" action="${action}">
```

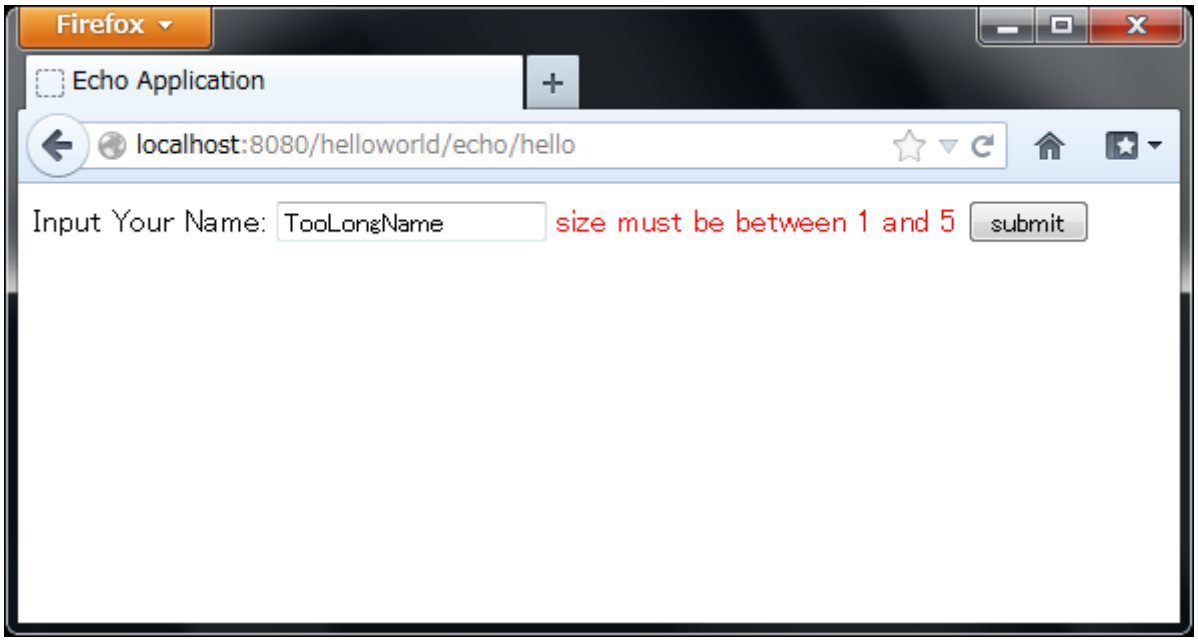
```
    <form:label path="name">Input Your Name:</form:label>
    <form:input path="name" />
    <form:errors path="name" cssStyle="color:red" /><!-- (1) -->
    <input type="submit" />
  </form:form>
</body>
</html>
```

Item number	Description
(1)	Add `form:errors` tag for displaying error message when an error occurs on input screen.

Implementation of input validation is completed.

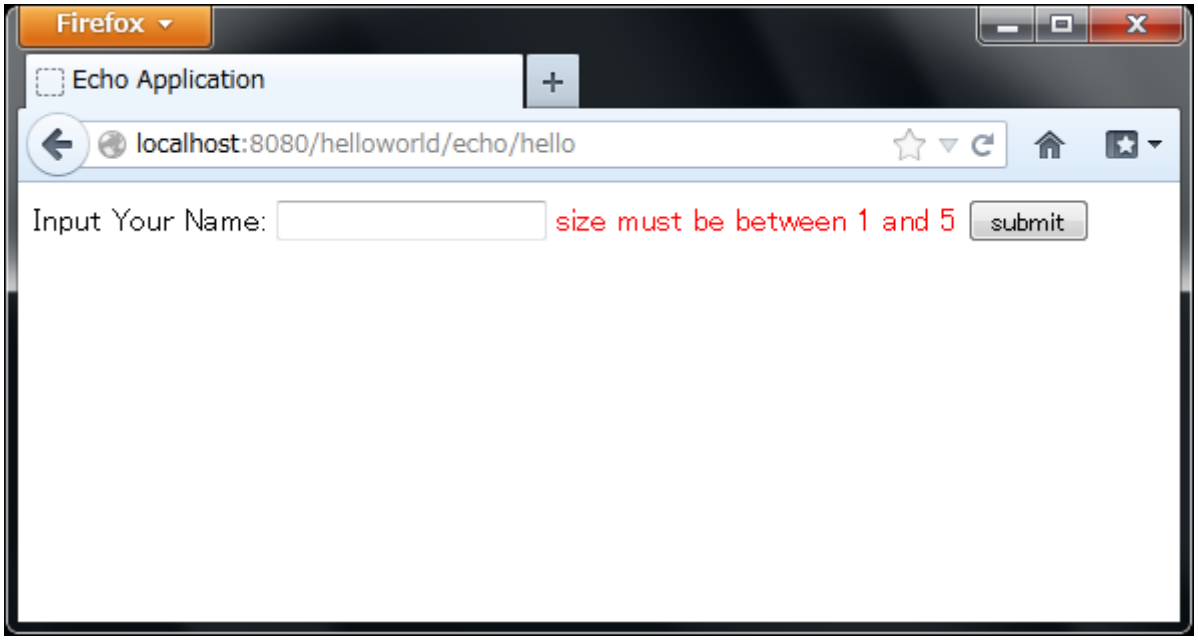
Error message is displayed in the following conditions:

- When an empty name is sent
- Size is more than 5 characters.



The generated HTML is as follows

```
<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" />
  </form>
</body>
</html>
```

Summary

The following are the learnings from this chapter.

1. Setting up configuration file of Spring MVC (simplified level)
2. How to do Screen Transition (simplified level)
3. Way to pass values between screens.
4. Simple input validation

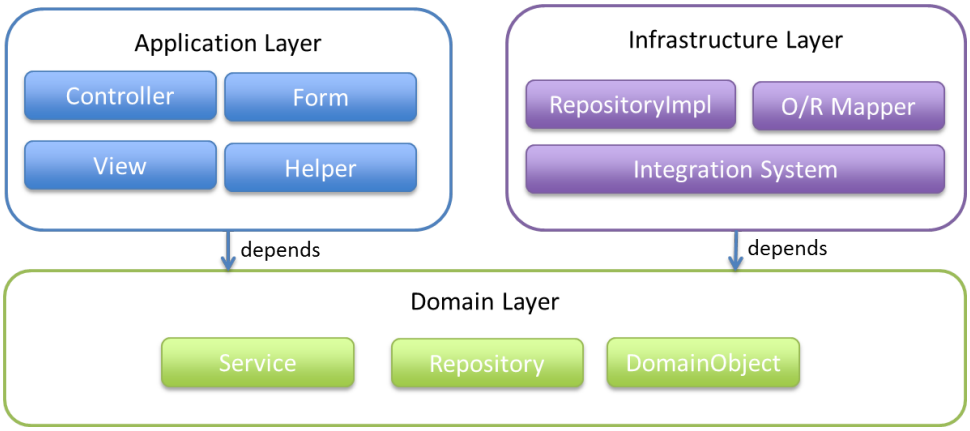
If above points are still not understood, it is recommended to read this chapter again and start again from building the environment. This will imporve the understanding of above concepts.

2.4 Application Layering

In this guideline, the application is classified into the following 3 layers.

- Application layer
- Domain layer
- Infrastructure layer

Each layer has the following components.



Application layer as well as infrastructure layer depends on domain layer, however **domain layer should not depend on other layers.** There can be changes in the application layer with the changes in domain layer, However, there should be no changes in domain layer with the changes in application layer.

Each layer is explained.

Note: Application layer, domain layer, infrastructure layer are the terms explained in "Domain-Driven Design (2004, Addison-Wesley)" of Eric Evans. However, though the terms are used, they are not necessarily as per the concept used in Domain Driven Design.

2.4.1 Layer definition

Flow of data from input to output is Application layer → Domain layer → Infrastructure layer, hence explanation is given in this sequence.

Application layer

Application layer does the wiring part of the application. It provides a UI to input/output information, calls the domain layer by converting request information into model information and also constructs the output from the processing result. **This layer should be as thin as possible and should not contain any business rules.**

Controller

Controller is responsible for mapping the requests to the process and passing the results to the view as well as session management. Main logic processing must be done in Service of domain layer without performing them in the Controller.

In Spring MVC, POJO class having `@Controller` annotation becomes the Controller class. The output of Controller is (logical name of) the View.

View

View generates output to the Client. Returns output results in various formats such as JSP/PDF/Excel/JSON. In Spring MVC, all this is done by `View` class.

Form

Represents the form of the screen. Used for sending form information to the Controller and data output from Controller to form of the screen. Conversion from Form to DomainObject(such as Entity) and conversion from DomainObject to Form has to be done in application layer, so that domain layer does not dependent on the application layer.

Note: When conversion is done in controller, source code of controller can get too long and visibility of Controller's main responsibility (like screen transition) becomes poor. In that case, it is recommended to create helper class and delegate conversion logic to helper classes.

In Spring MVC, form object are the POJO class that store request parameters. It is called as form backing bean.

Helper

It plays the role of assisting the Controller and performs the processes other than original duties of Controller such as mutual conversion of model between Application layer and Domain layer. It can be considered as a part of Controller.

Helper is an option, and should be created as POJO class if required.

Note: Helper class exists to improve the visibility of Controller; hence, it is OK to think of it as a part of Controller.

The main duty of Controller is routing (URL mapping and sepcifying the destination). If there is any processing required (converting JavaBean etc), then that part must be cut-off from controller and must be delegated to helper classes.

The purpose is to keep the controller class clean; hence, helper class is similar to private methods of Controller.

Domain layer

Domain layer is the core of the application. Represents the business issues to be resolved and has business object and business rules (checks like whether there is sufficient balance when crediting amount into the account). Domain layer is not so thick as compared to other layers and is reusable.

DomainObject

DomainObject is a model that represents resources required for business and items generated in the business process.
Models are broadly classified into following 3 categories.

- Resource models such as Employee, Customer, Product etc. (generally indicated by a noun),
- Event models such as Order, Payment (generally indicated by a verb),
- Summary model such as YearlySales, MonthlySales.

Domain Object is the Entity that represents an object which indicates 1 record of database table.

Note: Mainly Models holding state onlyare handled in this guideline.

In "Patterns of Enterprise Application Architecture (2002, Addison-Wesley)" of Martin Fowler, Domain Model is defined as Item having state and behavior. We will not be touching such models in detail.

In this guideline, Rich domain model proposed by Eric Evans is also not included. However, it is mentioned here for classification.

Repository

It can be thought of as a collection of DomainObjects and is responsible for CRUD operations such as create, read, update and delete of DomainObject. In this layer, only interface is defined. It is implemented by RepositoryImpl of infrastructure layer. Hence, in this layer, there is no information about how data access is implemented.

Service

Provides the business logic. This is also the transaction boundary.

Information related to Web such as Form and HttpRequest should not be handled in service. This information should be converted to object of domain layer in Application layer before calling Service.

Infrastructure layer

Implementation (Repository interface) of domain layer is provided in infrastructure layer. It is responsible for storing the data permanently (location where data is stored such as RDBMS, NoSQL etc.) as well as transmission of messages.

RepositoryImpl

Represents implementation of Repository interface of domain layer. It covers life cycle management of Domain-Object. With the help of this structure, it is possible to hide implementation details from domain layer. When using Spring Data JPA, a few RepositoryImpls are created by Spring Data JPA automatically.

O/R Mapper

It is responsible for mapping database with Entity. This function is provided by JPA, MyBatis and Spring JDBC. When using JPA specifically, EntityManager can be used. In case of MyBatis2 (TERASOLUNA DAO), Query-DAO and UpdateDAO are applicable are used.

Note: It is more correct to classify MyBatis and Spring JDBC as "SQL Mapper" and not "O/R Mapper"; however, in this guideline it is treated as "O/R Mapper".

Integration System Connector

It links a data store other than the database; such as messaging system, Key-Value-Store, Web service, existing legacy system etc. It also links with some external systems. Used for implementation of Repository.

2.4.2 Dependency between layers

As explained earlier, domain layer is the core of the application, and application layer and infrastructure layer are dependent on it.

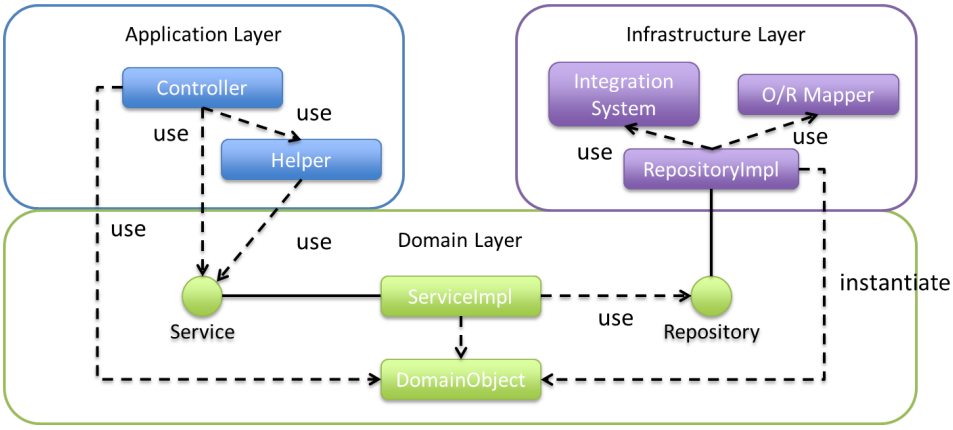
In this guideline, it is assumed that,

- Spring MVC is used in application layer
- Spring Data JPA and MyBatis are used in infrastructure layer

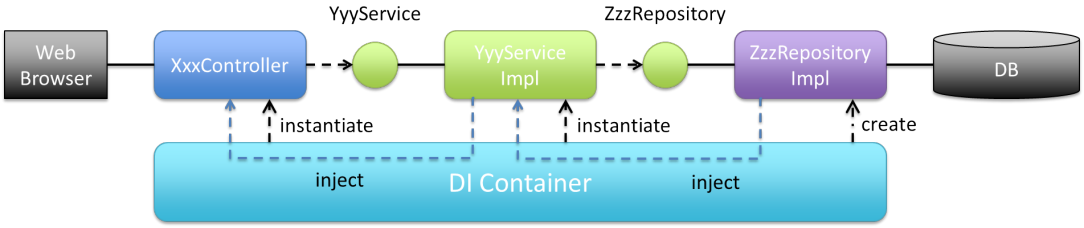
Even if there is change in implementation technology, the differences can be absorbed in respective layers and there should not be any impact on domain layer.

Coupling between layers is done by using interfaces and hence they can be made independent of implementation technology being used in each layer.

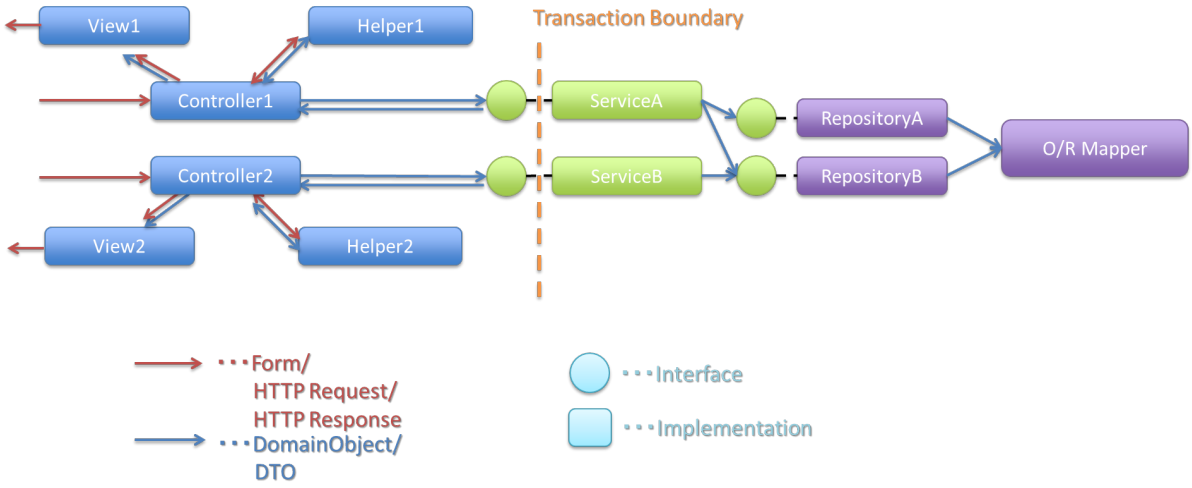
It is recommended to make loosely-coupled design by understanding the layering.



Object dependency in each layer can be resolved by DI container.



The flow from input to output is given in the following figure.



Sequence is explained using the example of update operation.

1. Controller receives the Request.
2. (Optional) Controller calls Helper and converts the Form information to DomainObject or DTO.
3. Controller calls the Service by using DomainObject or DTO.
4. Service calls the Repository and executes business logic.
5. Repository calls the O/R Mapper and persists DomainObject or DTO.
6. (Implementation dependency) O/R Mapper stores DomainObject or DTO information in DB.
7. Service returns DomainObject or DTO which is the result of business logic execution to Controller.
8. (Optional) Controller calls the Helper and converts DomainObject or DTO to Form.
9. Controller returns View name of transition destination.
10. View outputs Response.

Please refer to the below table to determine whether it is OK to call a component from another component.

Table.2.2 Possibility of calling between components

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

Note that **calling a Service from another Service is basically prohibited**. If services that can be used even from other services are required, SharedService should be created in order to clarify such a possibility. Refer to *Domain Layer Implementation* for the details.

Note: It may be difficult to follow the above rules at the initial phase of application development. If looking at a very small application, it can be created quickly by directly calling the Repository from Controller. However, when rules are not followed, there will be many maintainability issues when development scope becoms larger. It may be difficult to understand the impact of modifications and to add common logic which is cross-cutting in nature. It is strongly recommended to pay attention to dependency from the beginning of development so that there will be no problem later on.

Hiding the implementation details and sharing of data access logic are the merits of creating a Repository. However, it is difficult to share data access logic depending on the project team structure (multiple companies may separately implement business processes and control on sharing may be difficult etc.) In such cases, if abstraction of data access is not required, O/R Mapper can be called directly from Service as shown in the following diagram, without creating the repository.

Inter-dependency between components in this case must be as shown below:

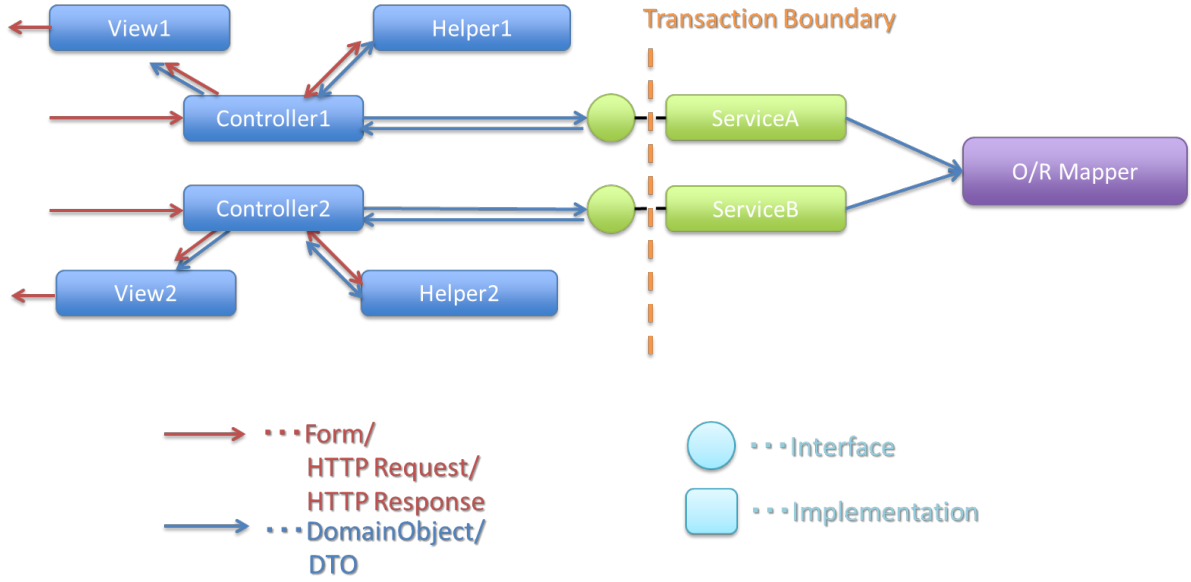


Table.2.3 Inter-dependency between components (without Repository)

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

2.4.3 Project structure

Here, recommended project structure is explained when application layering is done as described earlier.

The standard maven directory structure is pre-requisite.

Basically, it is recommended to create the multiple projects with the following configuration.

- [projectname]-domain ... Project for storing classes/configuration files related to domain layer
- [projectname]-web ... Project for storing classes/configuration files related to application layer
- [projectname]-env ... Project for storing files dependent on environment

([projectname] is the target project name)

Note: Classes of infrastructure layer such as RepositoryImpl are also included in project-domain. Originally, [projectname]-infra project should be created separately; however, normally there is no need to conceal the implementation details in the infra project and development becomes easy if implementation is also stored in domain project. Moreover, when required, [projectname]-infra project can be created.

Tip: For the example of multi-project structure, refer to Sample Application or Test Application of Common Library.

[projectname]-domain

Recommended structure of [projectname]-domain project is as below:

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

No.	Details
(1)	Store DomainObjects
(2)	Store Repository interfaces. Create a separate package for each Entity. If there are associated Entities to the main entity, then Repository interfaces of associated Entities must also be placed in the same package as main Entity. (For example, Order and OrderLine). If DTO is also required, it too must be placed in this package. RepositoryImpl belongs to Infrastructure layer; however, there is no problem in keeping it in this project. In case of using different data stores or existance of multiple persistence platforms, RepositoryImpl class must be kept in separate project or separate package so that implementation related details are concealed.
(3)	Service classes are kept here. Package must be created based on Entity Model or other functional unit. Interface and Implementation class must be kept at the same level of package. If input/output classes are also required, then they must be placed in this package.
(4)	Bean definition pertaining to domain layer.
(5)	Bean definition pertaining to infrastructure layer.

[projectname]-web

Recommended structure of [projectname]-web

```
[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
   └WEB-INF
      ├views ...(7)
      |  ├abc
      |  | ├list.jsp
      |  | └createForm.jsp
      |  └def
      |    ├list.jsp
      |    └createForm.jsp
      └web.xml
```

No,	Details
(1)	Package to store configuration elements of application layer.
(2)	Bean definited related to the entire application.
(3)	Define the properties to be used in the application.
(4)	Bean definitions related to Spring MVC.
(5)	Bean definitions related to Spring Security
(6)	Define the messages and other contents to be used for screen display (internationalization).
(7)	Store View(jsp) files.

[projectname]-env

The recommended structure of [projectname]-env project is below:

```
[projectName]-env
  └src
      └main
          └resources
              └META-INF
                  └spring
                      ├[projectname]-env.xml ...(1)
                      └[projectname]-infra.properties ...(2)
```

No.	Details
(1)	Bean definitions that depend on the environment (like dataSource etc).
(2)	Property definitions which depend on the environment.

Note: The purpose of separating [projectname]-domain and [projectname]-web into different projects is to prevent reverse dependency among them.

It is natural that [projectname]-web uses [projectname]-domain; however, [projectname]-domain must not refer projectname]-web.

If configuration elements of both, [projectname]-web and [projectname]-domain, are kept in a single project, it may lead to an incorrect reference by mistake. It is strongly recommended to prevent reference to [projectname]-web from [projectname]-domain by separating the project and setting order of reference.

Note: The reason for creating [projectname]-env is to separate the information depending on the environment and thereby enable switching of environment.

For example, set local development environment by default and at the time of building the application, create war file without including [projectname]-env. By creating a separate jar for integration test environment or system test environment, deployment becomes possible just by replacing the jar of corresponding environment.

Further, it is also possible to minimize the changes in case of project where RDBMS being used changes.

If the above point is not considered, contents of configuration files have to changed according to the target environment and the project has to be re-build.

For further details regarding significance of creating a separate project for environment dependent files, refer to *[coming soon] Exclude Environmental Dependencies*.

3

Tutorial (Todo Application)

3.1 Introduction

3.1.1 Points to study in this tutorial

- Basic application development and configuration of Eclipse project using TERASOLUNA Global Framework
- Development in accordance with application layering of this TERASOLUNA Global Framework

3.1.2 Target readers

- Basic knowledge of DI and AOP of Spring is required
- Web application development using Servlet/JSP
- SQL knowledge

3.1.3 Verification environment

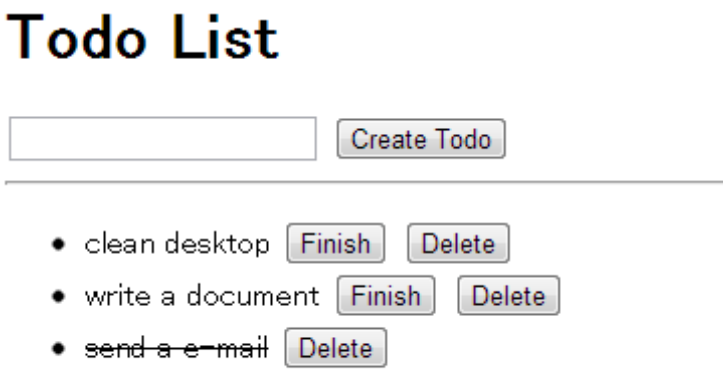
In this tutorial, operations are verified in the following environment.

Type	Name
OS	Windows7 64bit
JVM	Java 1.6
IDE	Spring Tool Suite Version: 3.2.0.RELEASE, Build Id: 201303060821 (henceforth referred to STS) Build Maven 3.0.4 (STS attached)
Application Server	VMWare vFabric tc Server Developer Edition v2.8 (STS attached)
Web Browser	Google Chrome 27.0.1453.94 m

3.2 Description of application to be created

3.2.1 Overview of application

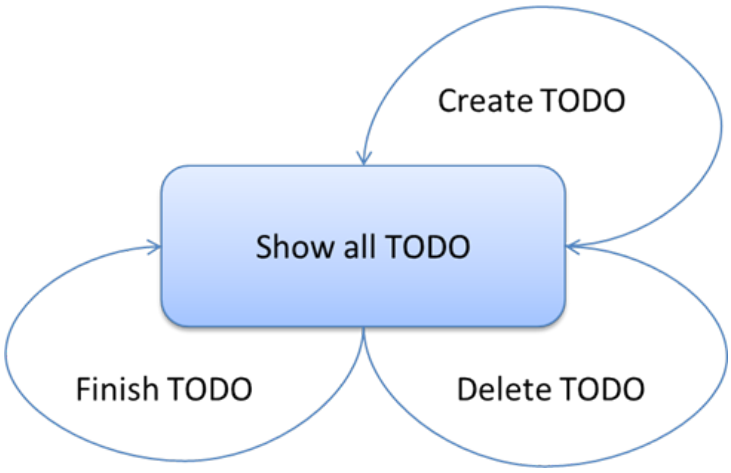
Application to manage TODO list is to be developed. TODO list display, TODO registration, TODO completion and TODO deletion can be performed.



3.2.2 Business requirements of application

RuleID	Description
B01	Only up to 5 incomplete TODO records can be registered
B02	For TODOs which are already completed, “TODO Complete” processing cannot be done.

Note: This application is for learning purpose only. It is not suitable as a real todo management application.



3.2.3 Screen transition of application

Sr.No.	Process name	HTTP method	URL	Description
1	Show all TODO	GET	/todo/list	
2	Create TODO	POST	/todo/create	Redirect to 1 after creation is completed
3	Finish TODO	POST	/todo/finish	Redirect to 1 after creation is completed
4	Delete TODO	POST	/todo/delete	Redirect to 1 after creation is completed

Show all TODO

- Display all records of TODO
- Provide `Finish` and `Delete` buttons for incomplete TODO
- Strike-through the completed records of TODO
- Only record name of TODO

Create TODO

- Save TODO sent from the form
- Record name of TODO should be between 1 - 30 characters
- When *Business requirements of application* B01 is not fulfilled, business exception with error code E001 is thrown

Finish TODO

- For the TODOs corresponding to todoId which is received from the form object, change the status to `completed`.

- When *Business requirements of application* B02 is not fulfilled, business exception with error code E002 is thrown
- When the corresponding TODO does not exist, business exception with error code E404 is thrown

Delete TODO

- Delete TODO corresponding to todoId sent from the form
- When the corresponding TODO does not exist, business exception with error code E404 is thrown

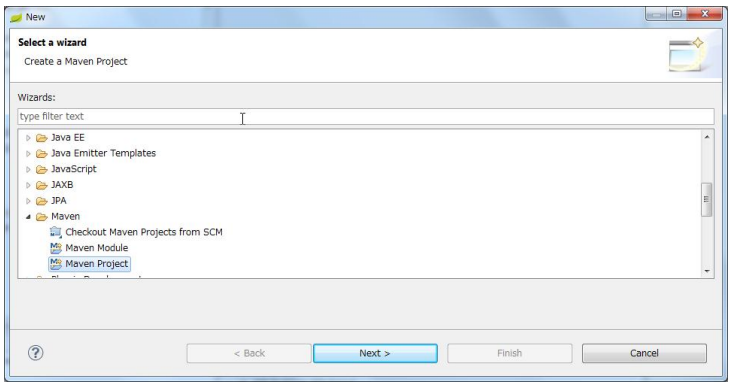
3.2.4 Error message list

Error code	Message	Parameter to be replaced
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 Environment creation

3.3.1 Project creation

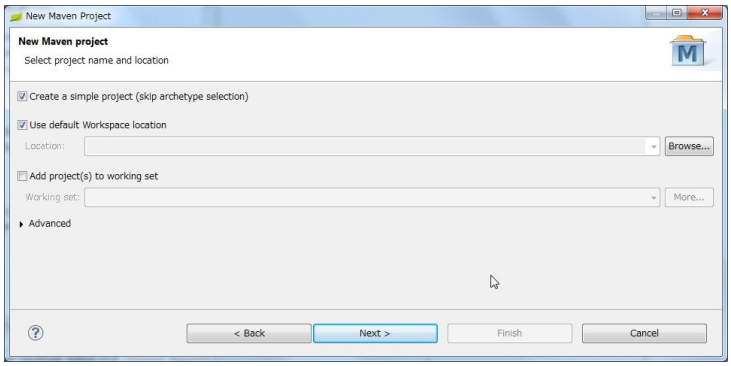
Select `File` -> `Other` -> `Maven` -> `Maven Project` and proceed to `Next`

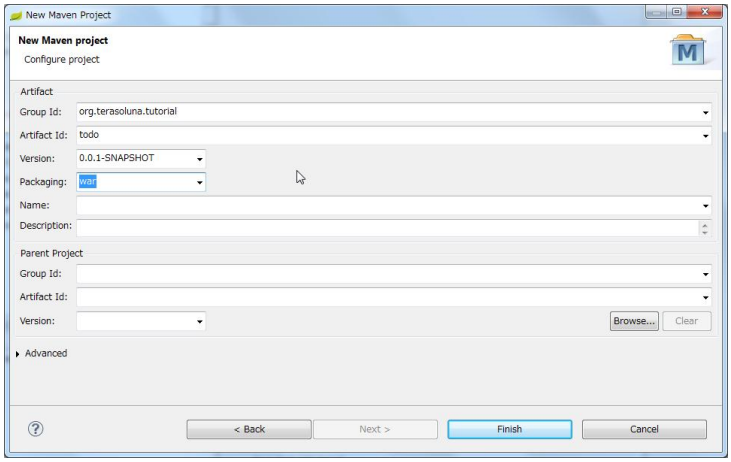


Insert check-mark to `Create a simple project` and proceed to `Next`

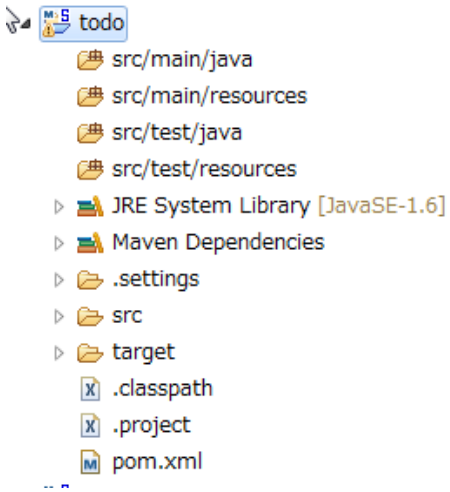
Group Id:	org.terasoluna.tutorial
Artifact Id:	todo
Packaging:	war

`Finish`

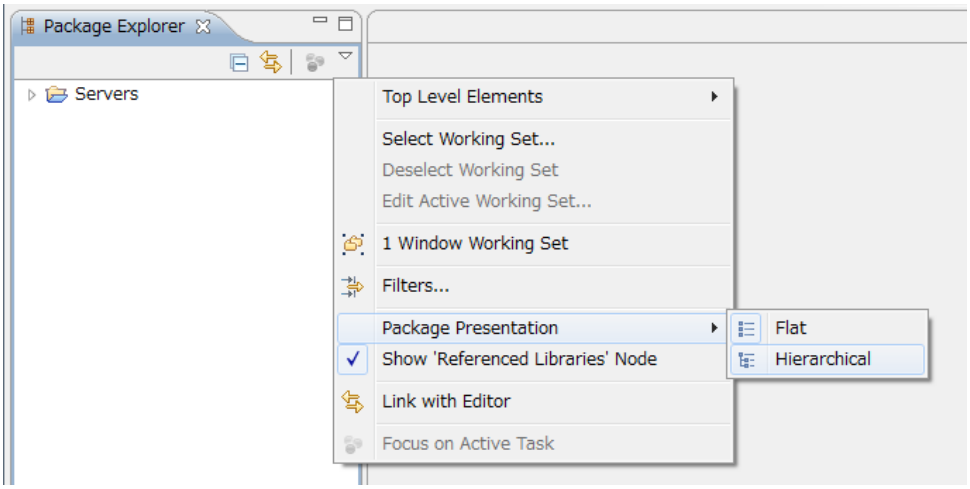




Project as shown below is created.



Note: For better visibility, Package Presentation must be changed to Hierarchical.



3.3.2 Maven settings

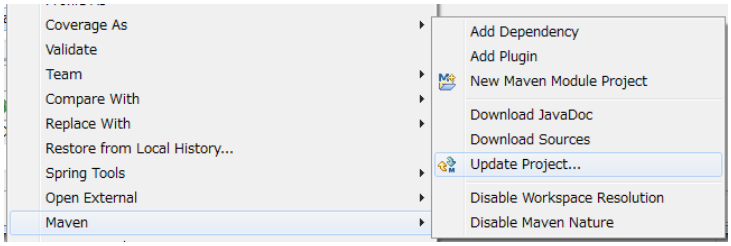
Change pom.xml as follows. If basic knowledge of Maven is not there, then just copy the pom.xml and skip the below explanation.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-i
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.terasoluna.tutorial</groupId>
    <artifactId>todo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>
    <!-- (1) -->
    <parent>
        <groupId>org.terasoluna.gfw</groupId>
        <artifactId>terasoluna-gfw-parent</artifactId>
        <version>1.0.0.RELEASE</version>
    </parent>

    <!-- (2) -->
    <repositories>
        <repository>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
            <id>terasoluna-gfw-releases</id>
            <url>http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-releases/<
        </repository>
        <repository>
            <releases>
                <enabled>false</enabled>
```

```
            </releases>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
            <id>terasoluna-gfw-snapshots</id>
            <url>http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-snapshots/
        </repository>
        <repository>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
            <id>terasoluna-gfw-3rdparty</id>
            <url>http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-3rdparty/<
        </repository>
    </repositories>

    <dependencies>
        <!-- (3) -->
        <!-- TERASOLUNA -->
        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-web</artifactId>
        </dependency>
        <!-- (4) -->
        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-security-web</artifactId>
            </dependency>
        <!-- (5) -->
        <dependency>
            <groupId>org.terasoluna.gfw</groupId>
            <artifactId>terasoluna-gfw-recommended-dependencies</artifactId>
            <type>pom</type>
        </dependency>

        <!-- (6) -->
        <!-- Servlet API/ JSP API -->
        <dependency>
            <groupId>org.apache.tomcat</groupId>
            <artifactId>tomcat-servlet-api</artifactId>
            <version>7.0.40</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.tomcat</groupId>
            <artifactId>tomcat-jsp-api</artifactId>
            <version>7.0.40</version>
            <scope>provided</scope>
```

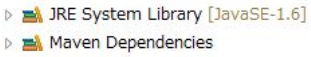
```
        </dependency>
    </dependencies>
</project>
```

Right click on the project name in “Package Explorer” and select [Maven] -> [Update Project]



Press “OK” button.

Confirm that the version of “JRE System Library” is “[JavaSE-1.6]”.



Note: In order to update the version of JDK to 7, set `<java-version>1.7</java-version>` in `<properties>` of pom.xml. After that, execute “Update Project”

```
    <project>
        <!-- omitted -->

        <properties>
            <java-version>1.7</java-version>
        </properties>
    </project>
```

If you are familiar with the Maven, and to make sure the following discussion.

Sr.No.	Description
(1)	Specify parent pom file of TERASOLUNA Global Framework. In this way, even without specifying the version, the library defined in terasoluna-parent can be added to dependency.
(2)	Specify URL of Maven repository for using TERASOLUNA Global Framework.
(3)	Add common library of TERASOLUNA Global Framework to dependency.
(4)	Add the library group recommended by TERASOLUNA Global Framework. Since terasoluna-gfw-recommended-dependencies is just pom file, `<type>pom</type>` should be mentioned.
(5)	Add the libraries which are recommended by TERASOLUNA Global Framework terasoluna-gfw-recommended-dependencies is just a pom file; hence `<type>pom</type>` must be specified.
(6)	Add Servlet/JSP API to dependency. Compatiblity with Servlet3 is necessary. These API have scope=provided (provided by the original AP server), they are not included in war, but it should be added explicitly to dependency for compiling on eclipse. （Further, though dependency name is tomcat-xxx, but the package of embedded class is javax.servlet, there is no dependency on tomcat）

Note:
When proxy server is used to access the internet, perform the following settings in <HOME>/.m2/settings.xml.
(In case of Windows7 C:\Users\<YourName>\.m2settings.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>
```

3.3.3 Project configuration

Below is the structure of the project to be created.

```
  src
└main
    ├java
    |  └todo
    |    ├ app ... Application layer
    |    |   └todo ... Classes related todo management business process
    |    └domain ... Domain layer
    |        ├model ... Domain Object
    |        ├repository ... Repository
    |        |   └todo ... Repository related to Todo
    |        └service ... Services
    |            └todo ... Service related to Todo
    ├resources
    |  └META-INF
    |      └spring ... configuration files related to Spring
    └wepapp
        └WEB-INF
            └views ... jsp
```

Since above will be created in order, there is no need to provide prepare the above structure beforehand.

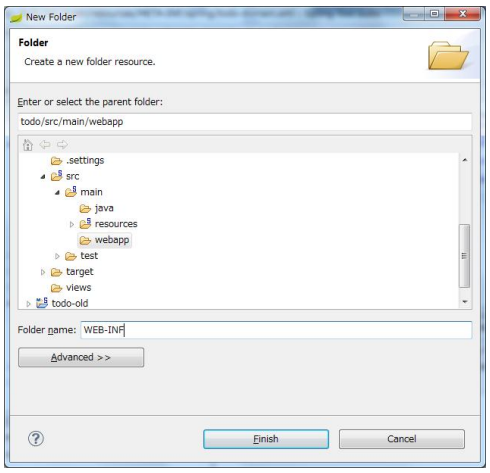
Note: It had been recommended to use a multi-project structure in *"Project Structure" section of previous chapter* . In this tutorial, a single project configuration is used because it focuses on ease of learning. However, when in a real project, multi project configuration is strongly recommended.

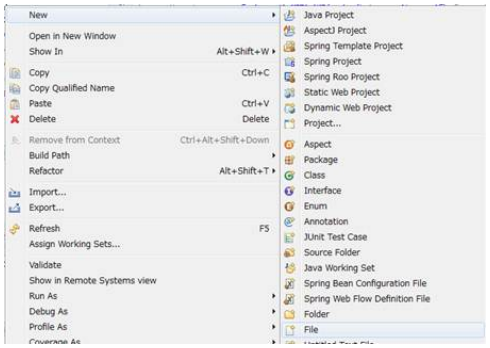
3.3.4 Creation of configuration file

web.xml settings

Create `src/main/webapp/WEB-INF/web.xml` and define servlet and filters. New WEB-INF folder should be created by `New` -> `Folder`.

Create web.xml by `New` -> `File`,



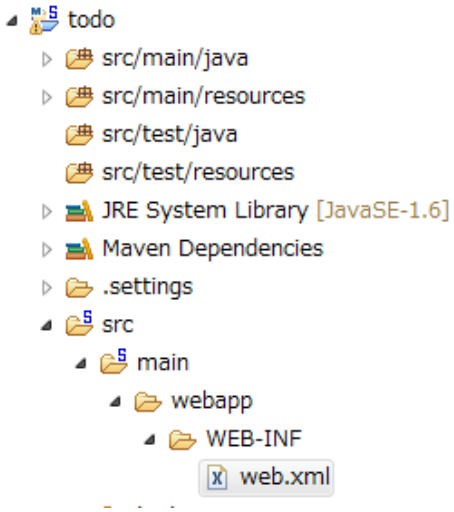


and describe the contents as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (1) -->
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-ap
    version="3.0">
    <!-- (2) -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <!-- Root ApplicationContext -->
        <param-value>
            classpath*:META-INF/spring/applicationContext.xml
        </param-value>
    </context-param>

    <!-- (3) -->
    <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>

    <!-- (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>
</web-app>
```

Sr.No.	Description
(1)	Declaration for using Servlet3.0.
(2)	Define `ContextLoaderListener`. `ApplicationContext` created by this listener is the root context. Path of Bean definition file is `META-INF/spring/applicationContext.xml` just under the classpath.
(3)	Define `CharacterEncodingFilter`. This is done for changing the character encoding of request and response to UTF-8.
(4)	Define `DispatcherServlet` that is the entry point of Spring MVC. Path of Bean definition file to be used in Spring MVC is `META-INF/spring/spring-mvc.xml` just under the classpath. `ApplicationContext` created here is the child of `ApplicationContext` created in step (2).
(5)	Define common JSP to be included. Include `/WEB-INF/views/common/include.jsp` for any JSP(*.JSP)



Settings of common JSP

Describe the contents to be included in each common JSP in src/main/webapp/WEB-INF/views/common/include.jsp. Also define taglib in common area. Create views/common folder and include.jsp file and describe as follows.

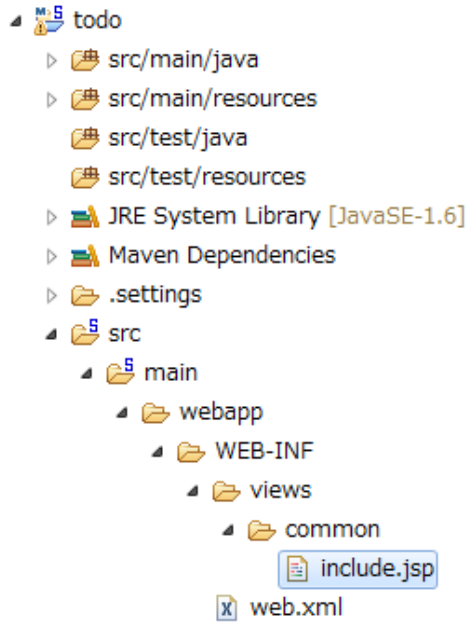
```
<%@ page session="false"%>
<!-- (1) -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<!-- (2)  -->
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<!-- (3) -->
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec"%>
<!-- (4) -->
<%@ taglib uri="http://terasoluna.org/functions" prefix="f"%>
<%@ taglib uri="http://terasoluna.org/tags" prefix="t"%>
```

Sr.No.	Description
(1)	Define standard tag library.
(2)	Define tag library for Spring MVC.
(3)	Define tag library for Spring Security.(However, it is not used in this tutorial)
(4)	Define EL function and tag library provided in common library.

Settings of Bean definition file

Create 4 types of Bean definition files in the following order.

- applicationContext.xml
- todo-domain.xml
- todo-infra.xml

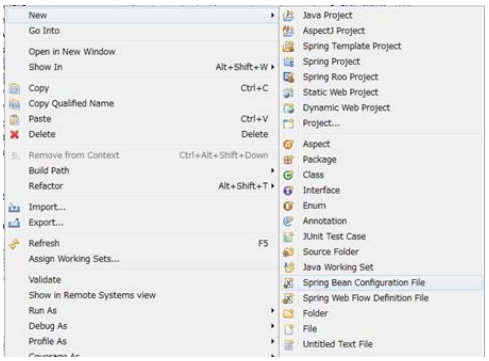


- spring-mvc.xml

applicationContext.xml

Carry out settings related to entire Todo application in src/main/resources/META-INF/spring/applicationContext.xml.

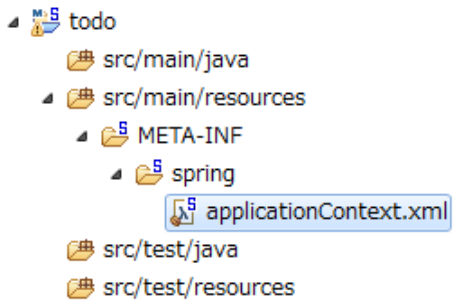
Create META-INF/spring folder and create applicationContext.xml using `New -> Spring Bean Configuration File`.




```
<?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
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.or
        http://www.springframework.org/schema/context http://www.springframework.org/schema/conte

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

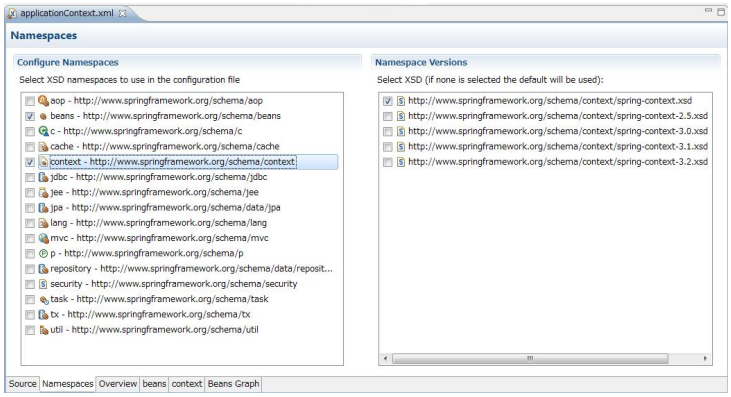
```
    <!-- (2) -->
    <context:property-placeholder
        location="classpath*:/META-INF/spring/*.properties" />

    <!-- (3) -->
    <bean class="org.dozer.spring.DozerBeanMapperFactoryBean">
        <property name="mappingFiles"
            value="classpath*:/META-INF/dozer/**/*-mapping.xml" />
    </bean>

</beans>
```

Sr.No.	Description
(1)	Import Bean definition file related to domain layer.
(2)	Read the settings of property file. Read any property file under `src/main/resources/META-INF/spring`. Using this setting, it is possible to insert property file value in ${propertyName} format in Bean definition file and to inject using @Value("${propertyName}") in Java class.
(3)	Define Mapper of library Dozer for Bean conversion. (Not used in this tutorial, but while defining XML file for mapping, it should be created in `src/main/resources/META-INF/dozer/xxx-mapping.xml` format. For the mapping file, refer to Dozer manual .)

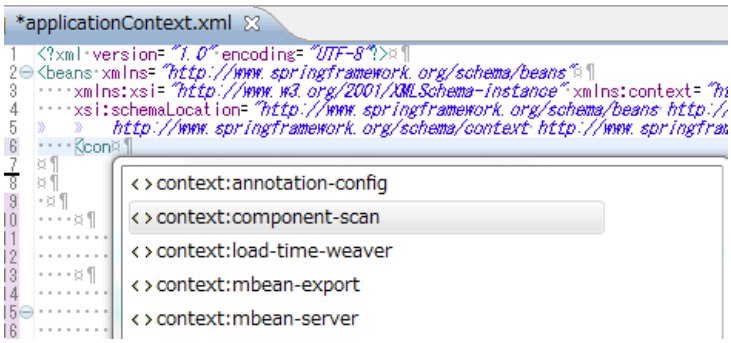


Note: While entering the above contents manually without copying them, open `namespace` tab and insert

check-mark in `beans` and `context` in `Configure Namespace`. It is recommended to select xsd file without version in `Namespace Versions`.



Thus, at the time of editing XML, it is possible to supplement the input using Ctrl+Space.



Moreover, by not specifying the version, latest xsd included in the jar is used.

---

todo-domain.xml

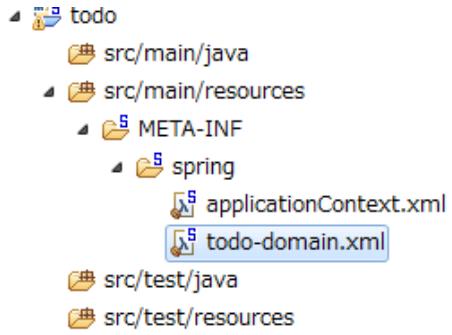
Carry out settings related to domain layer in `src/main/resources/META-INF/spring/todo-domain.xml`.

Create todo-domain.xml using `New` -> `Spring Bean Configuration File` under `META-INF/spring`.

```
<?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
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.or
        http://www.springframework.org/schema/context http://www.springframework.org/schema/conte
    <!-- (1) -->
    <import resource="classpath:META-INF/spring/todo-infra.xml"/>
```

```
    <!-- (2) -->
    <context:component-scan base-package="todo.domain" />
</beans>
```

Sr.No.	Description
(1)	Import Bean definition file related to infrastructure layer (explained later).
(2)	Components under todo.domain package are target of component scan. Thus, it is possible to make DI target by attaching annotations like `@Repository` , `@Service` , `@Controller`, `@Component` to the class under todo.domain package.

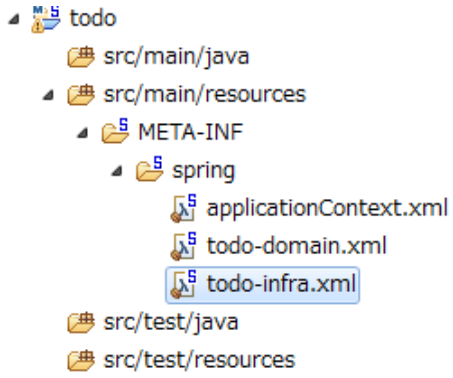


todo-infra.xml

Define Beans related to infrastructure layer in `src/main/resources/META-INF/spring/todo-infra.xml`. Here, DB setting are carried out, but DB is not used in this section, hence the definition may be blank as follows. Bean will be defined in next section.

Create todo-infra.xml using `New` -> `Spring Bean Configuration File` under `META-INF/spring`.

```
<?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.springframework.org
</beans>
```



Note: All the contents of todo-domain.xml, todo-infra.xml may likely be described in applicationContext.xml, however it is recommended to split the file for each layer. It will be easy to understand the definitions at various locations and to improve maintainability. There is no effect on a small application like the current tutorial, but larger the scope more the effect.

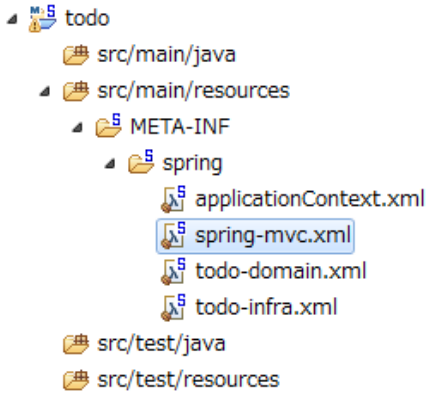
spring-mvc.xml

Define Spring MVC related definitions in `src/main/resources/META-INF/spring/spring-mvc.xml`.

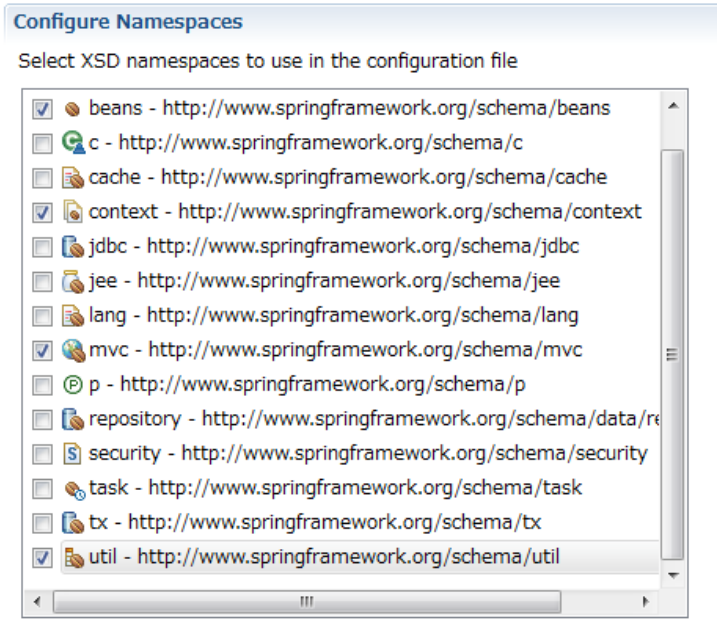
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframewo
    xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:util="http://www.springframework.
  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

    <!-- (1) -->
    <mvc:annotation-driven></mvc:annotation-driven>

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

    <!-- (3) -->
    <mvc:resources mapping="/resources/**"
        location="/resources/,classpath:META-INF/resources/"
        cache-period="#{60 * 60}" />

    <mvc:interceptors>
        <!-- (4) -->
        <mvc:interceptor>
            <mvc:mapping path="/**" />
            <mvc:exclude-mapping path="/resources/**" />
            <bean
```

```
                class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor" />
        </mvc:interceptor>
    </mvc:interceptors>

    <!-- (5) -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

Sr.No.	Description
(1)	Carry out annotation based default settings of Spring MVC.
(2)	Components under todo.app package that holds classes of application layer are made target of component-scan.
(3)	Carry out the settings for accessing the static resource (css, images, js etc.). Set URL path to `mapping` attribute and physical path to `location` attribute. In case of this setting, when there is a request for `<contextPath>/resources/css/styles.css`, `WEB-INF/resources/css/styles.css` is searched. If not found, `resources/css/style.css` is searched in classpath (src/main/resources and jar). If not found again, 404 error is returned. Cache period (3600 seconds = 60 minutes) of static resources is set in `cache-period` attribute. Further, static resources are not used in this tutorial. `cache-period="3600"` is also correct, however, in order to demostrate that it is 60 minutes, it is better to write as `cache-period="#{60 * 60}"` which uses SpEL .
(4)	Set interceptor that outputs trace log of controller processing. Set so that it excludes the path under `/resources` from mapping.
(5)	Carry out the settings of ViewResolver. Using these settings, for example, when view name `hello` is returned from controller, `/WEB-INF/views/hello.jsp` is executed.



Note: While entering the above contents manually without copying them, in addition to the operations described in todo-domain.xml, check mark should be put also to “mvc” and “util”.



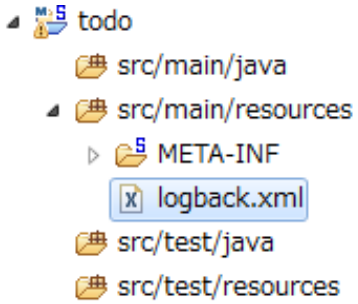
logback.xml settings

Carry out log output settings using logback in `src/main/resources/logback.xml`.

Create logback.xml by `New` -> `File` just under `src/main/resources/`.

```
<!DOCTYPE logback>
<configuration>
    <!-- (1) -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern><![CDATA[%d{yyyy-MM-dd HH:mm:ss} [%thread] [%-5level] [%-48logger{48}] - %ms
        </encoder>
    </appender>

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

    <!-- TERASOLUNA -->
    <!-- (3) -->
    <logger name="org.terasoluna.gfw">
        <level value="info" />
    </logger>
    <!-- (4) -->
    <logger name="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor">
        <level value="trace" />
    </logger>

    <!-- 3rdparty Loggers -->
    <!-- (5) -->
    <logger name="org.springframework">
        <level value="warn" />
    </logger>

    <!-- (6) -->
    <logger name="org.springframework.web.servlet">
        <level value="info" />
    </logger>

    <!-- (7) -->
    <root level="WARN">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

Sr.No.	Description
(1)	Set appender that outputs the log in standard output.
(2)	Set so that log of level 'debug' and above is output under todo package.
(3)	Change the log level of common library to info.
(4)	Set log level to 'trace' for `TraceLoggingInterceptor` which is defined in spring-mvc.xml.
(5)	Set so that log of level 'warn' and above is output for Spring Framework.
(6)	For log of Spring Framework, set the log level to 'info' and above for `org.springframework.web.servlet` so that logs valueable to development activity gets output.
(7)	Set so that log level of 'warn' and above is output by default.

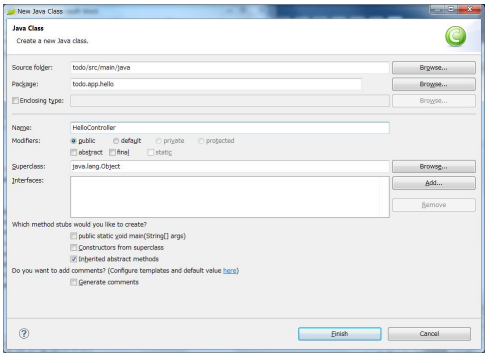


3.3.5 Operation verification

Before starting development of Todo application, create SpringMVC HelloWorld application and verify the operation.

Package:	todo.app.hello
Name:	HelloController

Create todo.app.hello.HelloController using `New` -> `Class`.



Edit HelloController as shown below.

```
package todo.app.hello;

import java.util.Date;

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;

// (1)
@Controller
public class HelloController {
    // (2)
    private static final Logger logger = LoggerFactory
            .getLogger(HelloController.class);

    // (3)
    @RequestMapping("/")
    public String hello(Model model) {
        Date now = new Date();
        // (4)
        logger.debug("hello {}", now);
        // (5)
        model.addAttribute("now", now);
        // (6)
        return "hello";
```

```
    }
}
```

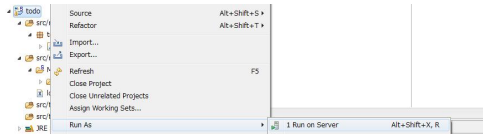
Sr.No.	Description
(1)	In order to make the Controller as component-scan target, attach `@Controller` annotation to class level.
(2)	Generate logger. Since logback implements logger and API is SLF4J, `org.slf4j.Logger` should be used.
(3)	Set mapping of methods for accessing / (root) using `@RequestMapping`.
(4)	Output debug log. `{}` is the placeholder.
(5)	For passing date to the screen, add Date object with name `now` to Model.
(6)	Return hello as view name. Using ViewResolver settings, WEB-INF/views/hello.jsp is output.

Next, create view(jsp). Create src/main/webapp/WEB-INF/views/hello.jsp as follows.

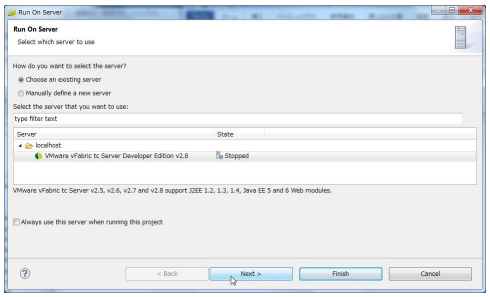
```
<!DOCTYPE html>
<html>
<head>
<title>Hello World!</title>
</head>
<body>
    <h1>Hello World!</h1>
    <p>
        Today is
        <!-- (1) -->
        <fmt:formatDate value="${now}" pattern="yyyy-MM-dd HH:mm:ss" />
    </p>
</body>
</html>
```

Sr.No.	Description
(1)	Display `now` passed from Controller. Here, `<fmt:formatDate>` tag is used for date formating.

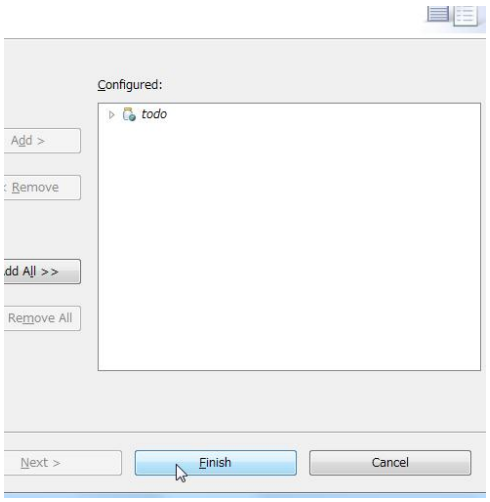
Right click package project name `todo` and click `Run As` -> `Run on Server`



Select AP server (here, VMWare vFabric tc Server Developer Edition v2.8) to be executed Click `Next`



Verify that todo is included in `Configured`, click `Finish` to start the server.



When started, log shown as below will be output. For / path, it is understood that hello method of `todo.app.hello.HelloController` is mapped.

```
2013-06-14 14:26:54 [localhost-startStop-1] [WARN ] [org.dozer.config.GlobalSettings
2013-06-14 14:26:54 [localhost-startStop-1] [INFO ] [o.springframework.web.servlet.DispatcherServ
2013-06-14 14:26:54 [localhost-startStop-1] [INFO ] [o.s.w.s.m.m.a.RequestMappingHandlerMapping
2013-06-14 14:26:55 [localhost-startStop-1] [INFO ] [o.s.web.servlet.handler.SimpleUrlHandlerMapp
2013-06-14 14:26:55 [localhost-startStop-1] [INFO ] [o.springframework.web.servlet.DispatcherServ
```

Note: WARN log of the first row may be ignored. In order to prevent, a blank dozer.properties should be created in src/main/resources.

If http://localhost:8080/todo is accessed in browser, following is displayed.

If you see console, you will understand that TRACE log using `TraceLoggingInterceptor` and debug log implemented by Controller is output.

```
2013-06-14 15:40:59 [tomcat-http--3] [TRACE] [o.t.gfw.web.logging.TraceLoggingInterceptor     ] -
2013-06-14 15:40:59 [tomcat-http--3] [DEBUG] [todo.app.hello.HelloController                  ] -
2013-06-14 15:40:59 [tomcat-http--3] [TRACE] [o.t.gfw.web.logging.TraceLoggingInterceptor     ] -
2013-06-14 15:40:59 [tomcat-http--3] [TRACE] [o.t.gfw.web.logging.TraceLoggingInterceptor     ] -
```

Note: `TraceLoggingInterceptor` outputs start and end of Controller in log. While ending, View and Model information and processing time are output.

After verification of log, one can delete HelloController and hello.jsp.

3.4 Creation of Todo application

Create Todo application. Order in which it must be created is as follows

- Domain layer (+ Infrastructure layer)

- Domain Object creation
- Repository creation
- Service creation
- Application layer
- Controller creation
- Form creation
- View creation

Further, do not use DB for saving Todo in this section. Creation of Repository in which DB is used, is carried out in *Change of infrastructure layer*.

3.4.1 Creation of Domain layer

Creation of Domain Object

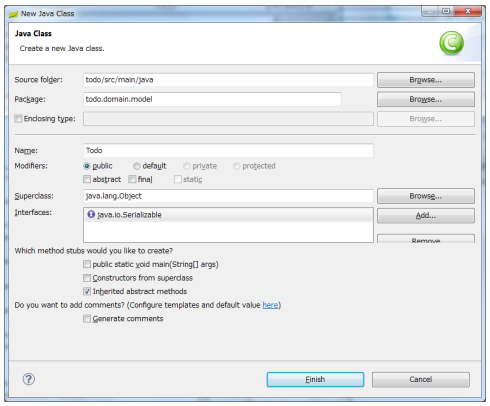
Following properties are required in domain object.

1. ID
2. Title
3. Completion flag
4. Created on

Create the following Domain objects. FQCN should be `todo.domain.model.Todo`. Implement as JavaBean.

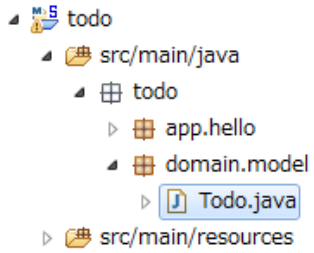
Package:	todo.domain.model
Name:	Todo
Interfaces:	java.io.Serializable

```
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;
```

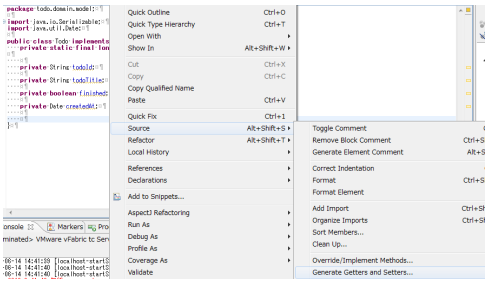



```
private String todoTitle;

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;
}
```

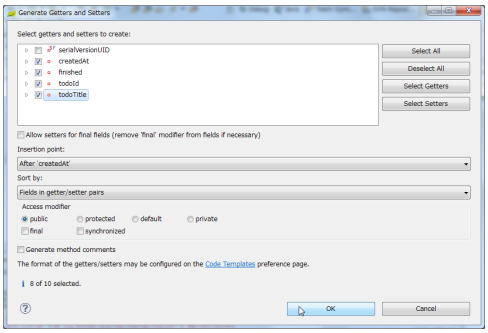
```
}
```



Note: Getter/Setter can be generated automatically. After defining fields, right click `Source` -> `Generate Getter and Setters`...



Click `OK` after selecting all other than serialVersionUID



Repository creation

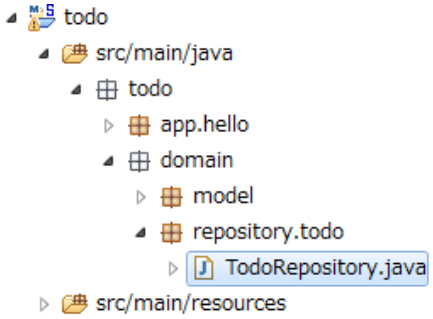
Following are the steps of CRUD operations pertaining to TODO object required in the current application.

- Fetch 1 record of TODO
- Fetch all records of TODO

- Delete 1 record of TODO
- Update 1 record of TODO
- Fetch record count of completed TODO

Create interface TodoRepository that defines these operations. FQCN should be `todo.domain.repository.todo.TodoRepository`.

```
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);
}
```



Note: Here, to improve versatility of TodoRepository, method is defined to fetch `record count having x completion status` and not `Fetch completed record count`.

Creation of RepositoryImpl (Infrastructure layer)

For simplification, in-memory implementation that uses Map as the implementation of Repository is used.

Repository implementation using DB is described in *Change of infrastructure layer*.

FQCN should be `todo.domain.repository.todo.TodoRepositoryImpl`.

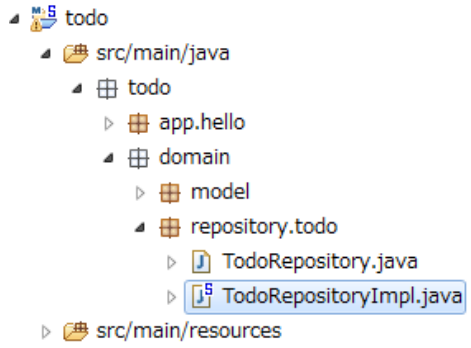
`@Repository`annotation must be used at class level.

```
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 (Map.Entry<String, Todo> e : TODO_MAP.entrySet()) {
            Todo todo = e.getValue();
            if (finished == todo.isFinished()) {
                count++;
            }
        }
        return count;
    }
```

```
}
```

Sr.No.	Description
(1)	To consider Repository as component scan target, add `@Repository`annotation at class level.

Since the business rules must not be included in Repository, it should focus only on inserting and removing information from the persistence store (here, it is Map).



Note: If package is divided completed on the basis of layers, it is better create classes of infrastructure layer under `todo.infrastructure`. However, in a normal project, infrastructure layer rarely changes. Hence, in order to improve the work efficiency, RepositoryImpl can be created in the layer same as the repository of domain layer.

Service creation

Implement business logic. The required processes are as follows.

- Fetch all records of Todo
- New creation of Todo
- Todo completion
- Todo deletion

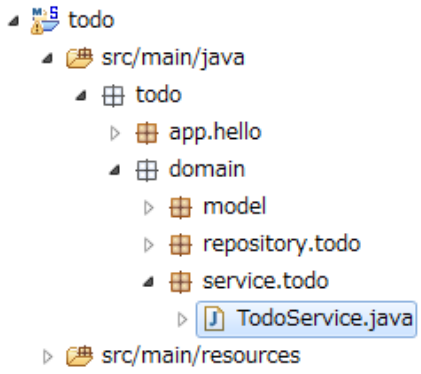
First, create TodoService interface and then define the above. FQCN shold be `todo.domain.serivce.todo.TodoService`.

```
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);
}
```

The required processes and the corresponding implementation methods are as follows

- Fetch all records of Todo→ findAll method
- New creation of Todo→create method
- Todo completion→finish method
- Todo deletion→delete method



FQCN of implementation class should be `todo.domain.service.TodoServiceImpl`.

```
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;
```

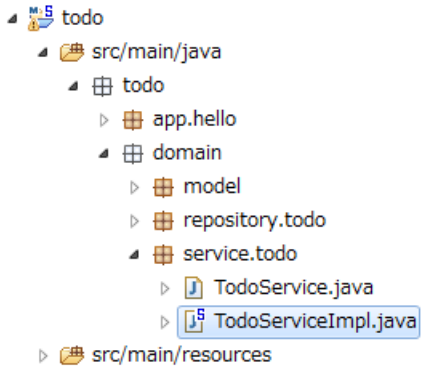
```
import todo.domain.repository.todo.TodoRepository;

@Service// (1)
// @Transactional // (2)
public class TodoServiceImpl implements TodoService {
    @Inject// (3)
    protected TodoRepository todoRepository;

    private static final long MAX_UNFINISHED_COUNT = 5;

    // (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
    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 + "."));
            // (7)
            throw new BusinessException(messages);
        }

        // (8)
        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);
    }
}
```

Sr.No.	Description
(1)	To consider Service as component-scan target, add `@Service` at class level.
(2)	DB is not used in the current implementation, hence transaction management is not required, but when DB is to be used, `@Transactional` should be added at class level. It is described in *Change of infrastructure layer*.
(3)	Inject TodoRepository implementation using `@Inject`.
(4)	Logic fetch a single record is used in both, delete and finish method. Hence it should be implemented in a method (OK to make it public by declaring in the interface).
(5)	Use `org.terasoluna.gfw.common.message.ResultMessage` provided in common library, as a class that stores result messages. Currently, for throwing error message, ResultMessage is added by specifying message type using `ResultMessages.error()`.
(6)	When target data is not found, `org.terasoluna.gfw.common.exception.ResourceNotFoundException` provided in common library is thrown.
(7)	When business error occurs, `org.terasoluna.gfw.common.exception.BusinessException` provided in common library is thrown.
(8)	UUID is used to generate a unique value. DB sequence may be used.

Note: In this chapter, error message is hard coded for simplification, but in reality it is not preferred

from maintenance viewpoint. Usually, it is recommended to create message externally in property file. The method for creating the message externally in property file is described in *[coming soon] Property Management*.



Creation of JUnit for Service

TBD

3.4.2 Creation of application layer

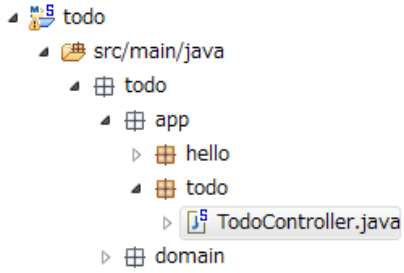
Since domain layer implementation is completed, use the domain layer to create application layer.

Creation of Controller

First create TodoController that controls screen transition. FQCN should be `todo.app.todo.TodoController`. It should be noted that the higher level package is different from the domain layer.

```
package todo.app.todo;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller // (1)
@RequestMapping("todo") // (2)
public class TodoController {

}
```

Sr.No.	Description
(1)	In order to make Controller as component-scan target, add `@Controller` at class level.
(2)	In order to bring all screen transitions handled by TodoController, under `<contextPath>/todo`, set `@RequestMapping(“todo”)` at class level.



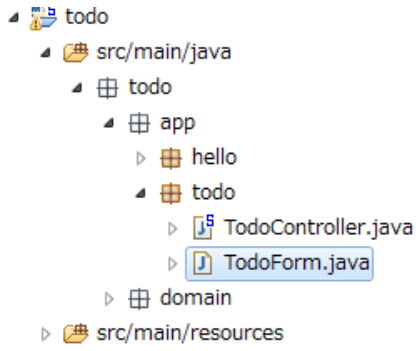
Show all TODO

Following is performed on this screen.

- Display of new form
- Display of all records of TODO

Form creation Form must contain title information. It should be implemented as JavaBean as shown below. FQCN should be `todo.app.todo.TodoForm`.

```
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;
    }

}
```



Implementation of Controller　Implement `setUpForm` method and `list` method in TodoController.

```
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)
    protected 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)
    }
}
```

Sr.No.	Description
(3)	Add `@Inject` annotation for injecting `TodoService` using DI container. Since instance of type `TodoService` managed by DI container is injected, as a result, `TodoServiceImpl` instance is injected.
(4)	Initialize Form. Adding `@ModelAttribute` annotation, form object of the return value of this method is added to Model with name `todoForm`. It is same as executing model.addAttribute(" todoForm " , form) in each method of TodoController.
(5)	Map list method to `<contextPath>/todo/list`. Since @RequestMapping("todo") is being set at class level, only @RequestMapping(value = "list") is required to be set here.
(6)	Add Todo list to Model and pass to View.
(7)	If `todo/list` is returned as View name, `WEB-INF/views/todo/list.jsp` will be rendered using `InternalResourceViewResolver` defined in spring-mvc.xml.

JSP creation Display Model passed from Controller in `WEB-INF/views/todo/list.jsp`. First, create buttons except "Finish"', "Delete".

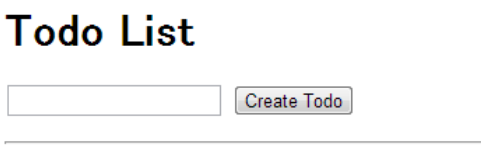
```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Todo List</title>
<style type="text/css">
.strike {
    text-decoration: line-through;
}
</style>
</head>
<body>
    <h1>Todo List</h1>
    <div id="todoForm">
        <!-- (1) -->
        <form:form
           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>
```

Sr.No.	Description
(1)	Display form object using <form:form> tag. Specify name of the form object added to Model by Controller in `modelAttribute` attribute. `contextPath` to be specified in `action` attribute can be fetched in `${pageContext.request.contextPath}`
(2)	Bind form property using <form:input> tag. Property name of form which is specified in `modelAttribute` should match with the value of `path` attribute.
(3)	Display entire list of Todo using `<c:forEach>` tag.
(4)	Determine whether to decorate text using strikethrough(text-decoration: line-through;) to display if it is completed (finished).
(5)	**To take XSS countermeasures at the time of output of character string, HTML escape should be performed using f:h() function.** Regarding XSS measures, refer to *XSS Countermeasures*.

Right click `todo` project in STS and start Web application by `Run As` → `Run on Server`.

If `http://localhost:8080/todo/todo/list` is accessed in browser, the following screen gets displayed.



Create TODO

Next, implement a new business logic after clicking `Create TODO` button on List display screen.

Modifications in Controller　Add `create` method to TodoController.

```
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
    protected TodoService todoService;

    // (8)
    @Inject
    protected 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";
    }

}
```

Sr.No.	Description
(8)	At the time of converting form object into domain object. Inject useful Mapper.
(9)	Set `@RequestMapping` such that HTTP method corresponds to POST with path `/todo/create`.
(10)	For performing input validation of form, add `@Valid` to form argument. Input validation result is stored in the immediate next argument `BindingResult`.
(11)	Return to list screen by redirecting after it is created normally. Add `RedirectAttributes` to argument for storing the information to be redirected.
(12)	Return to list screen in case of input error. Re-execute `list` method as it is necessary to fetch all records of Todo again.
(13)	Create Todo object from TodoForm using Mapper. No need to set if the property name of conversion source and destination is the same. There is no merit in using Mapper to convert only todoTitle property, but it is very convenient in case of multiple properties.
(14)	Execute business logic and in case of `BusinessException`, add the result message to Model and return to list screen.
(15)	Since it is created normally, add the result message to flash scope and redirect to list screen. Since redirect is used, there is no case of browser being read again and a new registration process being launched. Since this time 'Created successfully' message is displayed, ResultMessages.success() is used.

Modifications in Form To define input validation rules, add annotations in form class.

```
package todo.app.todo;

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

public class TodoForm {

    @NotNull // (1)
    @Size(min = 1, max = 30) // (2)
    private String todoTitle;

    public String getTodoTitle() {
        return todoTitle;
    }

    public void setTodoTitle(String todoTitle) {
        this.todoTitle = todoTitle;
    }
}
```

Sr.No.	Description
(1)	Since it is a mandatory item, add `@NotNull`.
(2)	Specify the range for `@Size` between 1 - 30 characters.

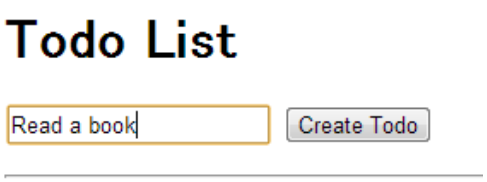
Modifications in JSP　Add the tag for dislaying the result message.

```
<!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>
```

Sr.No.	Description
(6)	Display result message using <t:messagesPanel> tag.
(7)	Display errors in case of input error using <form:errors> tag. Match value of path attribute of <form:errors> with path attribute of <form:input> tag.

If form is submitted by entering appropriate value in the form, 'Created successfully' message is displayed as given below.



When 6 or more records are registered and business error occurs, error message is displayed.

If form is submitted by entering null character, the following error message is displayed.

Customize message display `<t:messagesPanel>` result is output by default as follows.

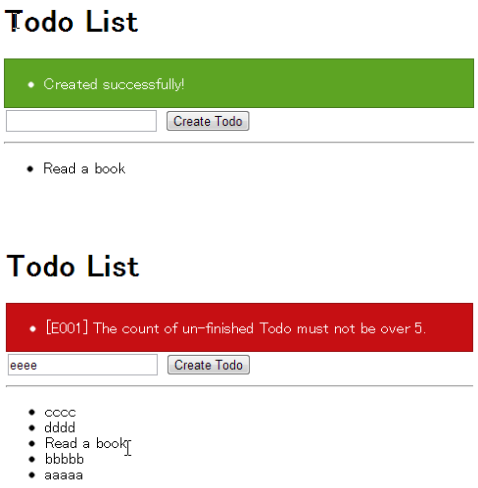
```
<div class="alert alert-success"><ul><li>Created successfully!</li></ul></div>
```

With the following modifications in style sheet (in <style> tag of `list.jsp`), customize appearance of the result message.

```
.alert {
    border: 1px solid;
}

.alert-error {
    background-color: #c60f13;
    border-color: #970b0e;
    color: white;
}

.alert-success {
    background-color: #5da423;
    border-color: #457a1a;
    color: white;
}
```

The message is as follows.

Moreover, input error message class can be specified to `cssClass` attribute of <form:errors> tag. Modify JSP as follows,




```
<form:errors path="todoTitle" cssClass="text-error" />
```

and add the following to style sheet.

```
.text-error {
    color: #c60f13;
}
```

Input error is as follows.

Finish TODO

Add `Finish` button to List display screen. If the form is submited, then hidden todoId target will be sent and the corresponding Todo will be completed.

Modifications in JSP　Add form in the JSP for completion of Todo.

```
<!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>
```

Sr.No.	Description
(8)	Display the form only if there are incomplete Todo. Send todoId by POST to `<contextPath>/todo/finish`.
(9)	Pass todoId using `<form:hidden>` tag. Also while setting the value in `value` attribute, HTML escaping should always be performed using **f:h() function.**

Modifications in Form　Form for completion process flow uses TodoForm. When todoId property needs to be added to TodoForm, input validation rules for new creation are applied as it is. For specifying separate rules for new creation and completion in a single Form, set `group` attribute.

```
package todo.app.todo;

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

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

    public static interface TodoFinish {
    };

    // (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;
    }

}
```

Sr.No.	Description
(3)	Create the class which will be the group name for performing group validation. Since class may be blank, define the interface here. Refer to *Input Validation* for group validation.
(4)	todoId is mandatory for completion process, hence add @NotNull. It is the rule required only at the time of completion, set TodoFinish.class in group attribute.
(5)	Rule for new creation is not required for completion process, hence set TodoCreate.class in group attribute of respective `@NotNull` and `@Size`.

Modifications in Controller Add completion processing logic to TodoController. Take precaution of using **@Validated instead of @Valid** for executing the group validation.

```
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
    protected TodoService todoService;

    @Inject
    protected 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";
    }
}
```

Sr.No.	Description
(16)	Change @Valid to @Validated for executing group validation. Multiple group classes can be specified in value. `Default.class` is the group when group is not specified in validation rules. At the time of using `@Validated`, `Default.class` can also be specified.
(17)	Set `@RequestMapping` to `/todo/finish` and HTTP method to POST.
(18)	Specify TodoFinish.class as the group for Finish.
(19)	In case of input error, return to list screen.
(20)	Execute business logic, add result message to Model and return to list screen in case when `BusinessException` occurs.
(21)	Since it is created normally, add result message to flash scope and redirect to list screen.

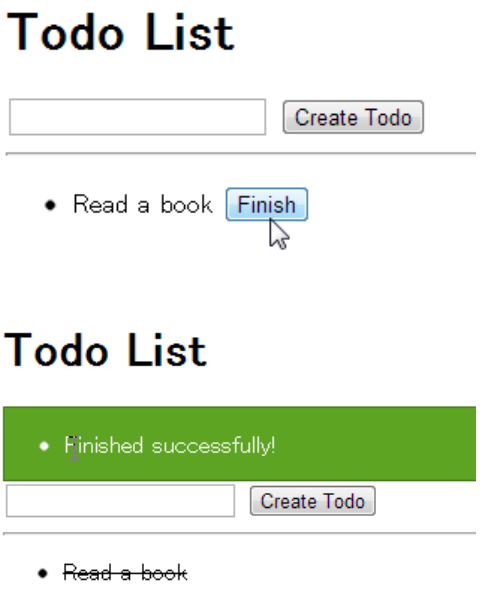
Note: Separate Form can also be created for Create and Finish. In that case, only the required parameters will be the properties of Form. However, as the number of classes increase, duplicate properties also increase, and when the specifications change, the modification cost will also be more. Moreover, if multiple Form objects in the same Controller are initialized by `@ModelAttribute` method, unnecessary instance gets generated because every time all Forms are being initialized. Therefore, it is recommended to basically consolidate the Form as much as possible to be used in a single Controller and carry out the group validation settings.

After creating new Todo, if submit is performed by Finish button, then strike-through is shown as below and it can be understood that the operation is completed.

Delete TODO

Add `Delete` button to list display screen. If the form is submitted, the hidden todoId target will be sent and corresponding Todo will be deleted.

Modifications in JSP Add form to the JSP for deletion business logic.




```
<!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>
```

Sr.No.	Description
(10)	Display form in the JSP for deletion request. Send todoId by POST to `<contextPath>/todo/delete`.
(11)	Pass todoId using <form:hidden> tag. Also while setting the value in `value` attribute, HTML escaping should always be performed using **f:h() function.**

Modifications in Form Add group for Delete to TodoForm. Rules are almost same as for Finish.

```
package todo.app.todo;

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

public class TodoForm {
    public static interface TodoCreate {
    };

    public static interface TodoFinish {
    };

    // (6)
    public static interface TodoDelete {
    }

    // (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;
    }

}
```

Sr.No.	Description
(6)	Define group TodoDelete for Delete.
(7)	Set so that validation of TodoDelete group will be carried out for todoId property.

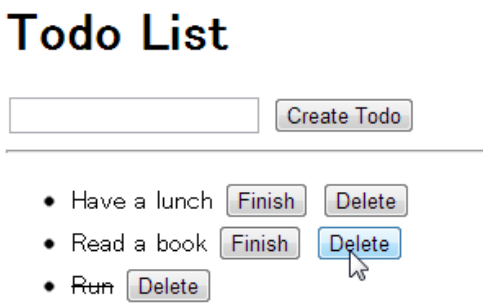
Modifications in Controller Add the logic for delete processing to TodoController. It is almost same as the completion process.

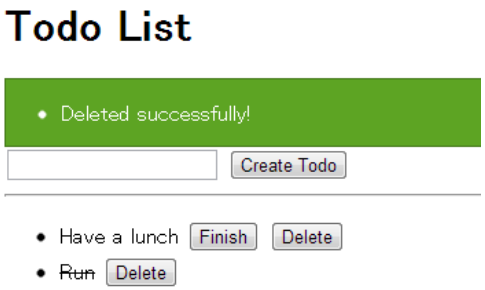
```
package todo.app.todo;

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
```

```
    protected TodoService todoService;

    @Inject
    protected 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)
    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";
    }

}
```

If submit is performed for Todo using `Delete` button, the following target TODO gets deleted.





3.5 Change of infrastructure layer

Till the last section, infrastructure layer is implemented using memory, but in this chapter, it is implemented using DB. O/R Mapper is used for accessing DB, here 2 methods are described: one using Spring Data JPA and second using TERASOLUNA DAO.

3.5.1 Common settings

First, apply settings common to the both, Spring Data JPA version and TERASOLUNA Dao version. Currently, H2Database is used for reducing the effort of setting up a DB.

Modifications in pom.xml

Define dependency for using H2Database in pom.xml.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.3.172</version>
    <scope>runtime</scope>
</dependency>
```

Warning: The above settings are **for easy trial of the application** and is not to be used in actual application development. Dependency settings of H2Database must be deleted in actual application development.
Further, in actual application development, usually use a data source that is provided by the application server. If your application are using a data source that is provided by the application server, `<scope>` of JDBC driver must be `provided`.

Definition of data source

Modifications in todo-infra.xml

Since data source definition is related to infrastructure layer, it should be defined in todo-infra.xml, but it is recommended to define the information depending on the environment like user name, password etc. of database in a separate Bean definition file (todo-env.xml).

Here, only import todo-env.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframewo

    <import resource="classpath:/META-INF/spring/todo-env.xml" />
</beans>
```

Note: By saving xxx-env.xml in another file and replacing only this file with build tools like Maven, configurations values that differs with each environment (development environment, test environment etc.) can be managed. Also the configuration file, in which data source of only specific environment is fetched from JNDI, can be managed.

Creation of todo-env.xml

Create `src/main/resources/META-INF/spring/todo-env.xml` and perform the following settings. Include the environment dependent settings (here, DataSource) in this file.

```
<?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.springframework.or

    <bean id="dataSource" 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>
</beans>
```

For improving maintainability define the properties in external property file.

Note: DataSource should be fetched using JNDI depending on the environment (Application Server). In that case, define `<jee:jndi-lookup id="dataSource" jndi-name="JNDI name" />` env file is created in such a way that, switch-over becomes possible at the time of build, between commons-dbcp in development environment and JNDI in test environment.

todo-infra.properties

Define property value related to infrastructure layer in `src/main/resources/META-INF/spring/todo-infra.proper`

```
database=H2
## (1)
database.url=jdbc:h2:mem:todo;DB_CLOSE_DELAY=-1;INIT=create table if not exists todo(todo_id varc
database.username=sa
database.password=
database.driverClassName=org.h2.Driver
# connection pool
## (2)
cp.maxActive=96
cp.maxIdle=16
cp.minIdle=0
cp.maxWait=60000
```

Sr.No.	Description
(1)	Perform settings related to database. Set H2 URL and driver. For simplification, in-memory DB is used here and settings are made such that initialization DDL is executed whenever AP server starts.
(2)	Perform settings related to connection pool. Here sample values are being set. Take note that the actual value differ according to server performance.

Modifications in todo-domain.xml

In order to enable transaction management using @Transactional annotation, set `<tx:annotation-driven>` tag.

```
<?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:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.o
        http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring
```

```
        http://www.springframework.org/schema/context http://www.springframework.org/schema/conte
    <context:component-scan base-package="todo.domain" />
    <import resource="classpath:META-INF/spring/todo-infra.xml"/>
    <tx:annotation-driven/>
</beans>
```

Modifications in TodoServiceImpl

```
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 // (9)
public class TodoServiceImpl implements TodoService {
    @Inject
    protected TodoRepository todoRepository;

    private static final long MAX_UNFINISHED_COUNT = 5;

    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) // (10)
    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);
    }
}
```

Sr.No.	Description
(9)	Add `@Transactional` at class level and manage all public methods as transactions. This will enable to start transaction while starting the method and to commit when the method ends normally. When unchecked exception occurs in between, transaction is rolled-back.
(10)	Add `readOnly=true` for the methods which only read data from the db. Depending on O/R Mapper, optimization is done at the time of reference queries by using this setting (not effective while using JPA).

3.5.2 Use Spring Data JPA

In this section, configuration is explained when Spring Data JPA is to be used in infrastructure layer. Proceed to *Use TERASOLUNA DAO* by skipping this section when TERASOLUNA DAO is to be used.

Modifications in configuration file for using Spring Data JPA

Modifications in pom.xml

Add the following to pom.xml for adding Spring Data JPA dependent library.

```
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-jpa</artifactId>
</dependency>
```

Modifications in todo-infra.xml

This setting in todo-infra.xml is for using JPA and Spring Data JPA . Define EntityManagerFactory of JPA here.

```
<?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jpa="http://www.springframework.org/schema/data/jpa"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.o
            http://www.springframework.org/schema/util http://www.springframework.org/schema/uti
            http://www.springframework.org/schema/data/jpa http://www.springframework.org/schema

    <import resource="classpath:/META-INF/spring/todo-env.xml" />
```

```
    <!-- (1) -->
    <jpa:repositories base-package="todo.domain.repository"></jpa:repositories>

    <!-- (2) -->
    <bean id="jpaVendorAdapter"
        class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <property name="showSql" value="false" />
        <property name="database" value="${database}" />
    </bean>

    <!-- (3) -->
    <bean
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
        id="entityManagerFactory">
        <!-- (4) -->
        <property name="packagesToScan" value="todo.domain.model" />
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
        <!-- (5) -->
        <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>
```

Sr.No.	Description
(1)	If Spring Data JPA is used, the class that implements `Repository` interface gets generated automatically. Specify the package that includes target repositories in `base-package` attribute of `<jpa:repository>` tag.
(2)	Set JPA implementation vendor. Define HibernateJpaVendorAdapter for using Hibernate for JPA implementation.
(3)	Define EntityManager.
(4)	Specify package name of entity.
(5)	Perform detailed settings related to Hibernate.

Modifications in todo-env.xml

Add Bean definition related to transaction manager.

```
<?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.springframework.o

    <bean id="dataSource" 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>
```

```
    <!-- (6) -->
    <bean class="org.springframework.orm.jpa.JpaTransactionManager"
        id="transactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>


</beans>
```

Sr.No.	Description
(1)	Set transaction manager. `id` should set to `transactionManager`. When another name is to be specified, transaction manager name should be specified in <tx:annotation-driven> tag in todo-domain.xml and <jpa:repository> tag in todo-infra.xml.

Note: Transaction manager should use JtaTransactionManager in case of JavaEE container. In this case, define transaction manager using <tx:jta-transaction-manager />.

These settings can be defined in todo-infra.xml for the project (while using Tomcat) which does not change with environment.

Modifications in spring-mvc.xml

Add `OpenEntityManagerInViewFilter` to spring-mvc.xml. Start and end of EntityManager lifecycle is carried out using Interceptor. By adding this configuration, Lazy Load support gets enabled at application layer (Controller or View class).

```
    <mvc:interceptors>

    <!-- ... -->

    <!-- (6) -->
    <mvc:interceptor>
        <mvc:mapping path="/**" />
        <mvc:exclude-mapping path="/resources/**" />
        <bean
            class="org.springframework.orm.jpa.support.OpenEntityManagerInViewInterceptor" />
    </mvc:interceptor>

</mvc:interceptors>
```

No.	Details
(6)	During the access (`/resources/**`) to static resources (css, js, image etc), database access is not going to happen for sure. Hence, intercept has been exempted.

Modifications in logback.xml

```
<!DOCTYPE logback>
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern><![CDATA[%d{yyyy-MM-dd HH:mm:ss} [%thread] [%-5level] [%-48logger{48}] - %m
        </encoder>
    </appender>

    <!-- Application Loggers -->
    <logger name="todo">
        <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>

    <!-- 3rdparty Loggers -->
    <logger name="org.springframework">
        <level value="warn" />
    </logger>

    <logger name="org.springframework.web.servlet">
        <level value="info" />
    </logger>

    <!-- (8) -->
    <logger name="org.hibernate.SQL">
        <level value="debug" />
    </logger>

    <!-- (9) -->
    <logger name="org.hibernate.type.descriptor.sql.BasicBinder">
        <level value="trace" />
    </logger>
```

```
    <!-- (10) -->
    <logger name="org.hibernate.engine.transaction">
        <level value="debug" />
    </logger>

    <root level="WARN">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

Sr.No.	Description
(8)	This setting is to output SQL log using Hibernate.
(9)	This setting is to output SQL bind variable using Hibernate.
(10)	This setting is to output transaction log using Hibernate.

Settings of Entity

JPA annotation must be used to map Todo class with database.

```
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;
    }
}
```

Sr.No.	Description
(1)	Add `@Entity` showing that it is JPA entity and set the corresponding table name using `@Table`.
(2)	Add `@Id` to the field corresponding to primary key column.
(3)	Set the corresponding column name by `@Column`.
(4)	It has to be clearly specified that `Date` type corresponds to which of `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`. Here `Timestamp` is specified.

Modifications in TodoRepository

```
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)
}
```

Sr.No.	Description
(1)	Set interface that extends JpaRepository. Specify Entity class (Todo) and primary key type (String) sequentially in Generics parameter. Since basic CRUD operations (findOne, findAll, save, delete etc.) are defined in upper level interface, only `countByFinished` can be defined in `TodoRepository`.
(2)	Specify JPQL executed at the time of calling `countByFinished` by @Query.
(3)	Set bind variable of JPQL specified in (2) by `@Param`. Here, add `@Param("finished")` to method argument for inserting `':finished"` in JPQL.

Modifications in TodoRepositoryImpl

When Spring Data JPA is used, RepositoryImpl gets generated automatically from interface. Hence, delete the unnecessary TodoRepositoryImpl.

Usage of Spring Data JPA is completed here. Start AP server to output SQL log and transaction log as follows for Todo display and new creation.

3.5.3 Use TERASOLUNA DAO

Configuration method using TERASOLUNA DAO in infrastructure layer, is explained in this section.

Note: TERASOLUNA DAO is the library providing a simple SQL mapper which is extended from `org.springframework.orm.ibatis.support.SqlMapClientDaoSupport` which is a linkage class of MyBatis2.3.5 and Spring. DAO having the following 4 interfaces is provided.

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

jp.terasoluna.fw.dao.ibatis.XxxDAOiBatisImpl is implemented for each interface.

Setting for using the TERASOLUNA DAO

Modifications in pom.xml

Add the following in pom.xml in order to add dependent library related to TERASOLUNA DAO.

```
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-mybatis2</artifactId>
</dependency>
```

Modifications in todo-infra.xml

The following setting is to be done in todo-infra.xml to use TERASOLUNA DAO.

```
<?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.springframework.o

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

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

    <!-- (4) -->
    <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>
```

Sr.No.	Description
(1)	Define `SqlMapClient`.
(2)	Set the path of SqlMap configuration file. Read *sqlMapConfig.xml under `META-INF/mybatis/config`.
(3)	Set the path of SqlMap file. Read *-sqlmap.xml of any folder under `META-INF/mybatis/sql`.
(4)	Define TERASOLUNA DAO.

Modifications in todo-env.xml

Add definition of transaction manager. Define transaction manager in environment dependent configuration file since JtaTransactionManager can also be used in JavaEE container.

```
<?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.springframework.o

    <bean id="dataSource" 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>

    <!-- (1) -->
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
```

```
        <property name="dataSource" ref="dataSource" />
    </bean>

</beans>
```

Sr.No.	Description
(1)	Set transaction manager. `id` should be set to `transactionManager`. If a separate name is to be specified, transaction manager name should be specified even in <tx:annotation-driven> tag.

Modifications in logback.xml

```
<!DOCTYPE logback>
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern><![CDATA[%d{yyyy-MM-dd HH:mm:ss} [%thread] [%-5level] [%-48logger{48}] - %m
        </encoder>
    </appender>

    <!-- Application Loggers -->
    <logger name="todo">
        <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>

    <!-- 3rdparty Loggers -->
    <logger name="org.springframework">
        <level value="warn" />
    </logger>

    <logger name="org.springframework.web.servlet">
        <level value="info" />
    </logger>

    <!-- (8) -->
    <logger name="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <level value="debug" />
```

```
    </logger>

    <!-- (9) -->
    <logger name="java.sql.Connection">
        <level value="trace" />
    </logger>
    <logger name="java.sql.PreparedStatement">
        <level value="debug" />
    </logger>

    <root level="WARN">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

Sr.No.	Description
(8)	Set to output transaction log for `DataSourceTransactionManager`.
(9)	Set to output SQL log.

Creation of sqlMapConfig

Create `src/main/resources/META-INF/mybatis/config/sqlMapConfig.xml` as given below.

```
<?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>
    <!-- (1) -->
    <settings useStatementNamespaces="true" />
</sqlMapConfig>
```

Sr.No.	Description
(1)	This setting is to give namespace to SQLID.

Modifications in RepositoryImpl

```
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
public class TodoRepositoryImpl implements TodoRepository {
    // (2)
    @Inject
    protected QueryDAO queryDAO;

    @Inject
    protected UpdateDAO updateDAO;

    // (3)
    @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) {
        // (4)
        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);
    }
}
```

Sr.No.	Description
(1)	Manage all the transactions of public methods by adding `@Transactional` at class level. Since settings are made even at Service side that calls the Repository, the transactions can be managed even without adding `@Transactional`. However, since the `propagation` attribute is `REQUIRED` by default, internal (Repository side) transactions take part in external (Service side) transactions when the transactions are nested.
(2)	Inject QueryDAO and UpdateDAO using `@Inject`.
(3)	Implementing repository methods involves passing of `SQLID` and parameters to TERASOLUNA DAO. Use `QueryDAO` for reference and `UpdateDAO` for update. Set SQL corresponding to SQLID as follows.
(4)	Create new and update are executed using `save` method. In order to determine which process is to be executed, create 'exists' method. In this method, the record count of todoId is acquired and whether the record count is greater than 0 is checked.

Note: It is convenient to use save method as it can be used to create new and to update. On the other hand,

the disadvantage of using save method from performance perspective is that SQL gets executed twice. If performance is higher priority, then create `create` method for new creation and `update` method for updating the data.

Create SQLMap file

Create `src/main/resources/META-INF/mybatis/sql/todo-sqlmap.xml` and mention sql corresponding to `SQLID` used in TodoRepositoryImpl as shown below.

```
<?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>
```

With this, usage of TERASOLUNA DAO is completed. Start AP server, SQL log and transaction log as shown below are output for the display of Todo and for creating a new one.

```
 2013-11-28 14:15:37 [tomcat-http--3] [TRACE] [o.t.gfw.web.logging.TraceLoggingInterceptor     ]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [o.s.jdbc.datasource.DataSourceTransactionManager]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [o.s.jdbc.datasource.DataSourceTransactionManager]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [o.s.jdbc.datasource.DataSourceTransactionManager]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [java.sql.Connection                             ]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [java.sql.Connection                             ]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [java.sql.PreparedStatement                      ]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [java.sql.PreparedStatement                      ]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [java.sql.PreparedStatement                      ]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [o.s.jdbc.datasource.DataSourceTransactionManager]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [o.s.jdbc.datasource.DataSourceTransactionManager]
 2013-11-28 14:15:37 [tomcat-http--3] [DEBUG] [o.s.jdbc.datasource.DataSourceTransactionManager]
 2013-11-28 14:15:37 [tomcat-http--3] [TRACE] [o.t.gfw.web.logging.TraceLoggingInterceptor     ]
 2013-11-28 14:15:37 [tomcat-http--3] [TRACE] [o.t.gfw.web.logging.TraceLoggingInterceptor     ]
```

3.6 In the end...

In this tutorial, following contents have been learnt

- How to develop basic applications by TERASOLUNA Global Framework, and how to build Eclipse project
- Using the STS
- How to use TERASOLUNA Global Framework with Maven
- Way of development using application layering of TERASOLUNA Global Framework.
- Implementation of domain layer with POJO(+ Spring)
- Implementation of application layer with the use of JSP tag libraries and Sring MVC
- Development of Infrastructure layer with the use of Spring Data JPA
- Development of Infrastructure layer with the use of MyBatis2

The following improvement can be done in the TODO management application.

- To externalize the property → *[coming soon] Property Management*
- To externalize the messages → *Message Management*
- To add paging → *[coming soon] Pagination*
- To add exception handling → *Exception Handling*
- To add a CSRF measures → *[coming soon] CSRF(Cross Site Request Forgeries) Countermeasures*

4

Application Development using TERASOLUNA Global Framework

Description of various rules as well as recommended implementation on the use of TERASOLUNA Global Framework.

The flow of development in this guideline with be as follows.

4.1 Domain Layer Implementation

4.1.1 Roles of domain layer

Domain layer **implements business logic**to be provided to the application layer.

Implementation of domain layer is classified into the following.

S.No.	Classification	Description
1.	*Implementation of Entity*	Creation of classes (Entity class) to hold business data.
2.	*Implementation of Repository*	Implementation of the methods to operate on business data. These methods are provided to Service classes. These are in particular the CRUD operations on Entity object.
3.	*Implementation of Service*	Implementation of the methods for executing business logic. These methods are provided to the application layer. Business data required by the business logic is fetched as the Entity object through the Repository.

This guideline recommends the structure of creating Entity classes and Repository for the following reasons.

1. **By splitting the overall logic into business logic (Service) and the logic to access business data, the implementation scope of business logic gets limited to the implementation of business rules**,
2. **Access logic to business data is standardized** by consolidating the operations of business data in the Repository.

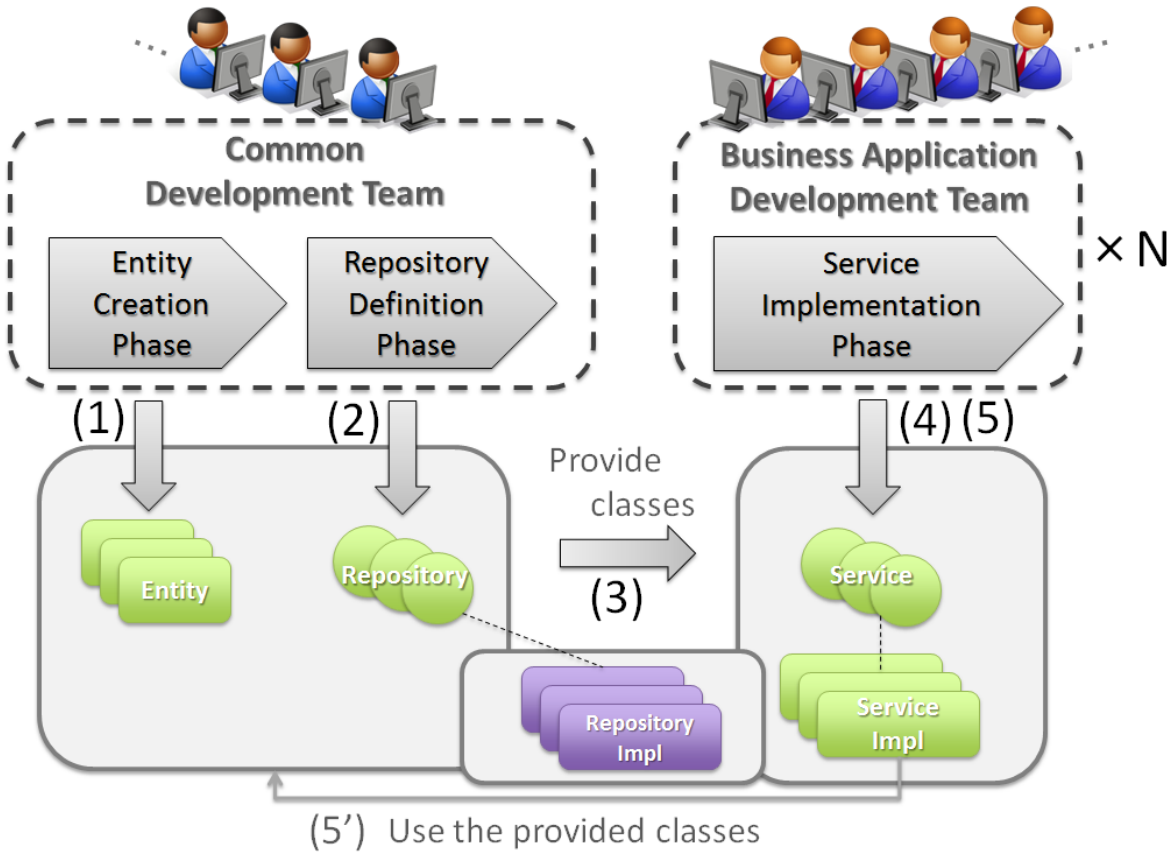
Note: Though this guideline recommends a structure to create Entity classes and Repository, it is not mandatory to perform development in this structure.

Decide a structure by taking into account the characteristics of the application as well as the project (structure of development team and development methodology).

4.1.2 Flow of development of domain layer

Flow of development of domain layer and allocation of roles is explained here.

A case where application is created by multiple development teams is assumed; however, the flow itself remains same even if developed by a single team.



S.No.	Team in-charge	Description
(1)	Common development team	Common development team designs and creates Entity classes.
(2)	Common development team	Common development team works out access pattern for the Entity classes extracted in (1) and designs methods of Repository interface. Common development team should implement the methods to be shared by multiple development teams.
(3)	Common development team	Common development team provides Entity classes and Repository created in (1) and (2) to the business application development team. At this time, it requests each business application development team to implement the Repository interface.
(4)	Business application development team	Business application development team takes charge of the implementation of Repository interface.
(5)	Business application	Business application development team develops Service interface and Service class

> **Warning:** A system having a large development scope is often developed by assigning the application to multiple teams. In that case, it is strongly recommended to provide a common team to design Entity classes and Repository.
> When there is no common team, O/R Mapper(Mybatis) should be called from Service directly without creating Entity classes and Repository.

4.1.3 Implementation of Entity

Policy of creating Entity class

Create an Entity using the following method.

Specific creation method is shown in *Example of creating Entity class*.

S.No.	Method	Supplementary
1.	Create Entity class for each table.	However, Entity class is not required for mapping tables which represent the the relationship between the tables. Further, when the tables are not normalized, Entity class for each table rule may not be applible. Refer to the *Warning as well as Note outside this table* for the approach related to not-normalized tables.
2.	When there is a FK (Foreign Key) in the table, the Entity class of FK destination table must be defined as one of the properties of this Entity.	When there is 1:N relationship with FK destination table, use either `java.util.List<E>` or `java.util.Set<E>`. The Entity corresponding to the FK destination table is called as the related Entity in this guideline.
3.	Treat the code related tables as `java.lang.String` rather than as an Entity.	Code related tables are to manage the pairs of code value and name. When there is a need to bifurcate the process as per code values, `enum` class corresponding to code value should be created and it must be defined as property.

Warning: When table is not normalized, **check whether to use the method of creating the Entity classes and Repository** by considering the following points. Since the unnormalized tables do not have good compatibility with JPA, it is better not to use JPA.

- Creating an appropriate Entity class may often not be possible because of increased difficulty in creating entities if the tables are not normalized.

  In addition, efforts to create an Entity classes also increases.

  Two viewpoints must be taken into consideration here. Firstly "Can we assign an engineer who can perform normalization properly?" and secondly "Is it worth taking efforts for creating normalized Entity classes?".
- If the tables are not normalized, the logic to fill the gap of differences between the Entity class and structure of table is required in data access.

  Here the viewpoint to be considered is, "Is it worth taking efforts to fill the gap of differences between the Entity class and structure of table ?".

The method of creating Entity classes and Repository is recommended; however, the characteristics of the application as well as the project (structure of development team and development methodology) must also be taken into account.

Note: If you want to operate on business data with normalized Entity classes - even if the tables are not normalized - it is recommended to use Mybatis as an implementation of RepositoryImpl of the infrastructure layer.

Mybatis is the O/R Mapper developed to map the SQL with object and not to map the database table record with object. So depending on the implementation of SQL, mapping to the object independent of table structure is possible.

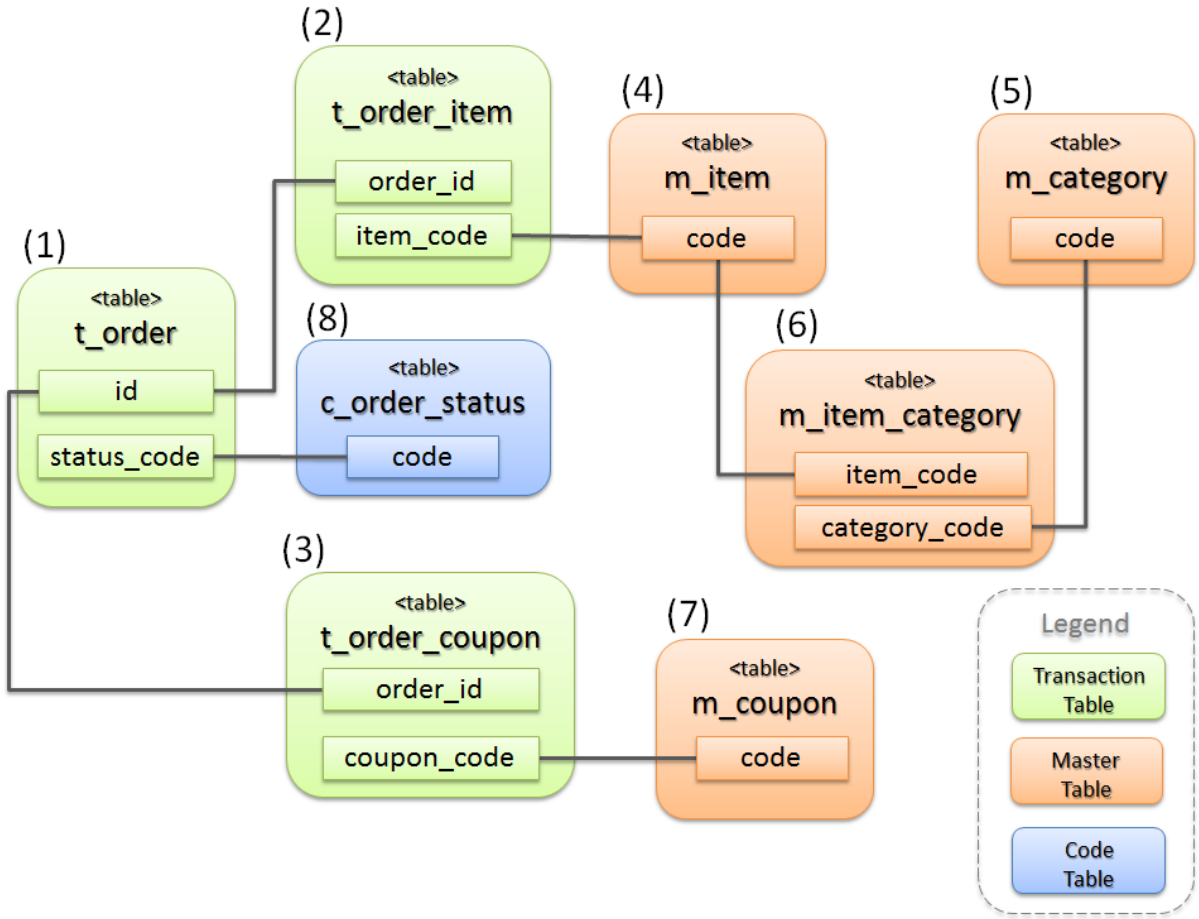
Example of creating Entity class

The creation of Entity class is explained using specific examples.

Following is an example of creating the business data of Entity classes required for purchasing a product on some shopping site.

Table structure

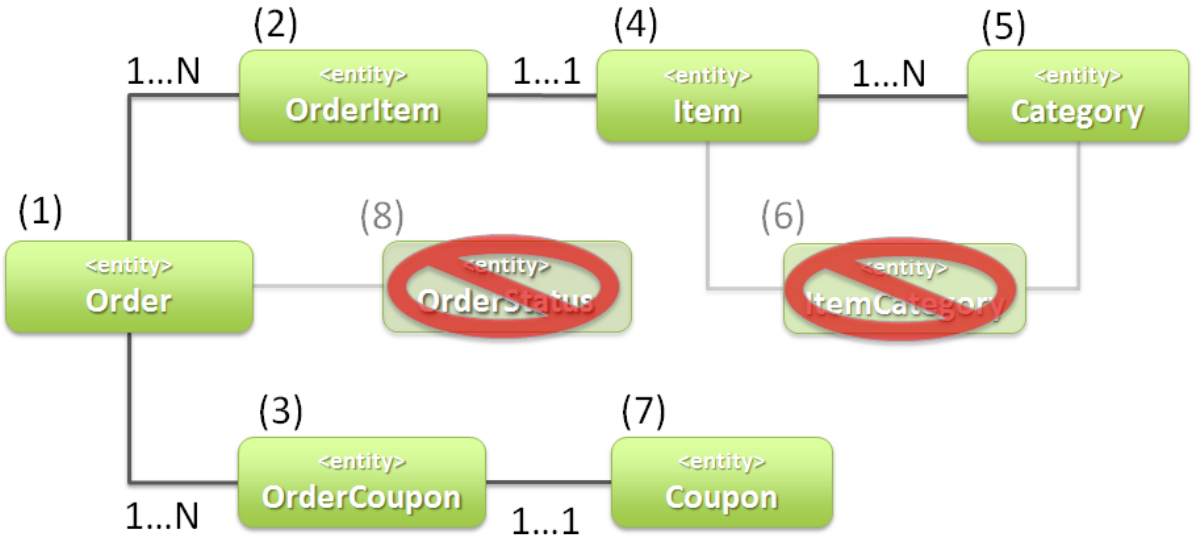
The table structure is as given below:



S.No.	Classification	Table name	Description
(1)	Transaction related	t_order	Table to store orders. 1 record is stored for 1 order.
(2)		t_order_item	Table to store the products purchased in 1 order. Record of each product is stored when multiple products are purchased in 1 order.
(3)		t_order_coupon	Table to store the coupon used in a single order. Record of each coupon is stored when multiple coupons are used in 1 order. No record is stored when coupon is not used.
(4)	Master related	m_item	Master table to define products.
(5)		m_category	Master table to define product category.
(6)		m_item_category	Master table to define the category of the product.

Entity structure

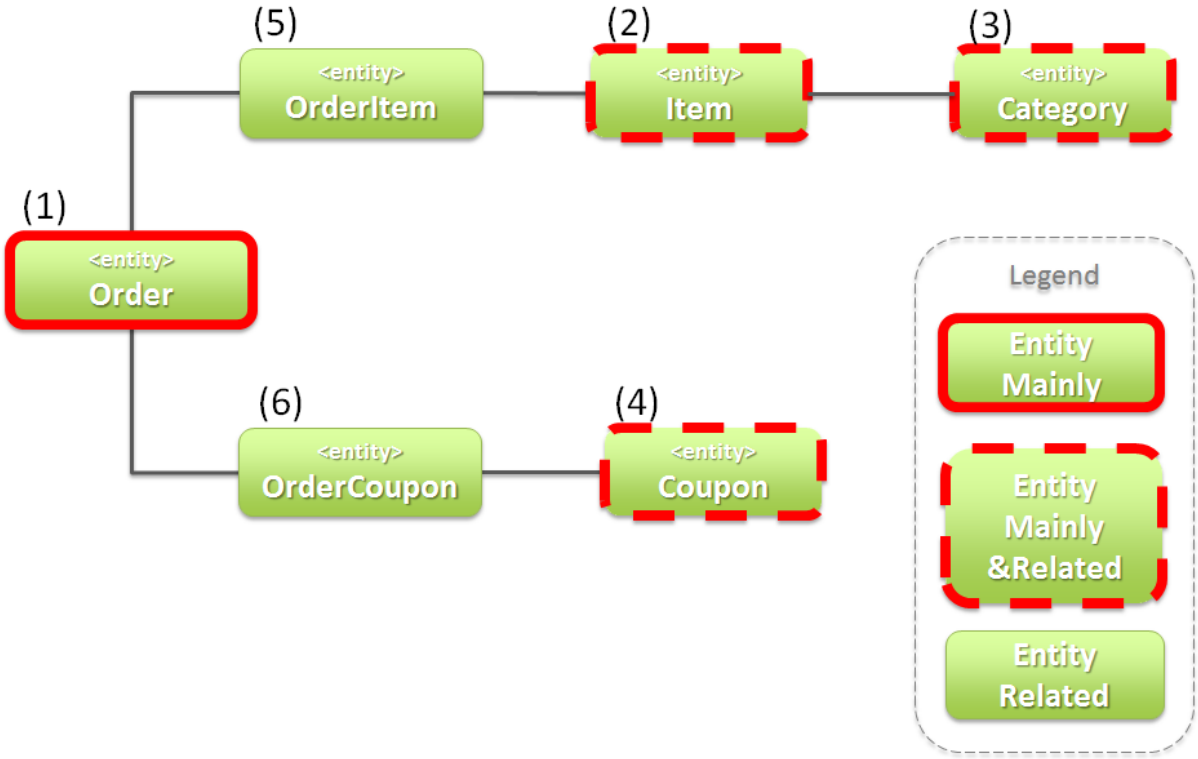
If Entity classes are created with the help of policy defined by the above table, it results into the following structure.



S.No.	Class name	Description
(1)	Order	Entity class indicating 1 record of t_order table. Multiple `OrderItem`and `OrderCoupon`are stored as the related Entity.
(2)	OrderItem	Entity class indicating 1 record of t_order_item table. `Item` is stored as the related Entity.
(3)	OrderCoupon	Entity class indicating 1 record of t_order_coupon table. `Coupon` is stored as the related Entity.
(4)	Item	Entity class indicating 1 record of m_item table. Multiple `Category`are stored as the related Entity. The association between `Item` and `Category`is done using m_item_category table.
(5)	Category	Entity class indicating 1 record of m_category table.
(6)	ItemCategory	Entity class is not created since m_item_category table is the mapping table to store the relationship between m_item table and m_category table.
(7)	Coupon	Entity class indicating 1 record of m_coupon table.
(8)	OrderStatus	Entity class is not created since c_order_status table is code table.

As it can be observed from the above entity diagram, it might first seem that Order class is the only main entity class in the shopping site application; however, there are other main entity class as well other than Order class.

Below is the classification of main Entity classes as well as Entity class which are not main.



The following 4 Entities are treated as the main Entity for creating shopping site application.

S.No.	Entity class	Reasons for treating as the main Entity.
(1)	Order class	It is one of the most important Entity class in the shopping site. Order class is the Entity indicating the order itself and a shopping site cannot be created without the Order class.
(2)	Item class	It is one of the most important Entity class in the shopping site. Item class is the Entity indicating the products handled in the shopping site and a shopping site cannot be created without Item class.
(3)	Category class	Product categories are displayed usually on the top page or as a common menu in shopping sites. In such shopping sites, Category becomes a main entity. Usually operations like 'search category list' can be expected.
(4)	Coupon class	Often discounts through coupons are offered in the shopping sites as a measure of promoting sales of the products. In such shopping sites, Coupon becomes a main entity. Usually operations like 'search coupon list' can be expected.

The following are not main Entities for creating shopping site application.

S.No.	Entity class	Reason of not treating Entity as main Entity
(5)	OrderItem class	This class indicates 1 product purchased in 1 order and exists only as the related Entity of Order class. So OrderItem class should not be considered as main Entity.
(6)	OrderCoupon	This class indicates 1 coupon used in 1 order and exists only as the related Entity of Order class. So, OrderCoupon class should not be considered as main Entity.

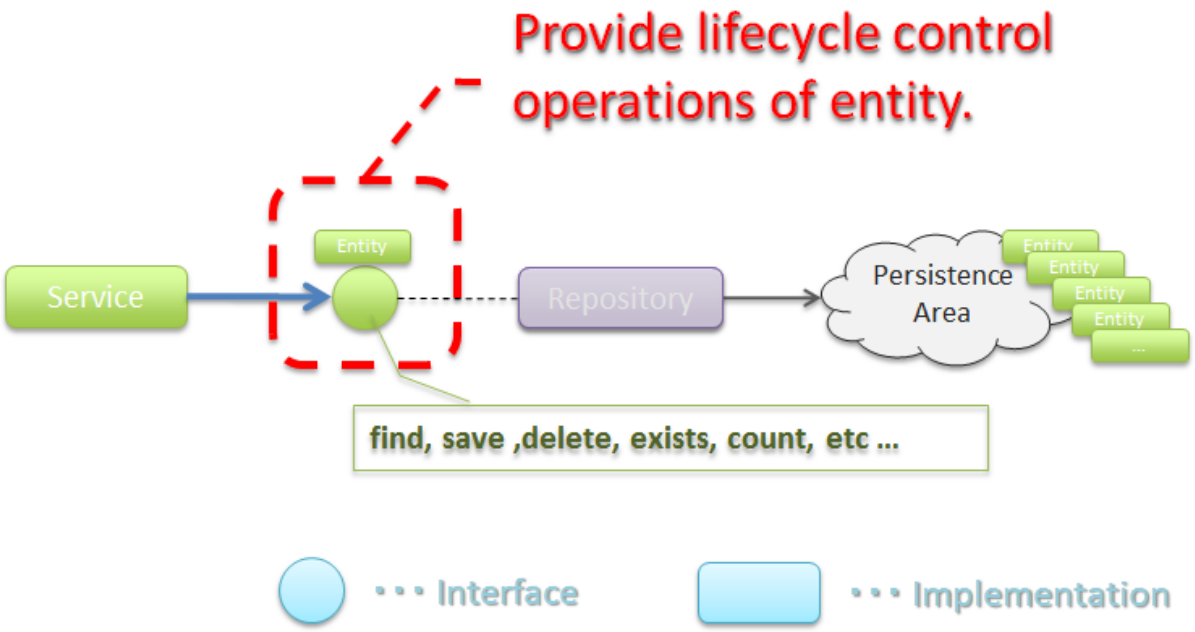
4.1.4 Implementation of Repository

Roles of Repository

Repository has following 2 roles.

1. **To provide to Service, the operations necessary to control Entity lifecycle (Repository interface).**

   The operations for controlling Entity lifecycle are CRUD operations.



2. **To provide persistence logic for Entity (implementation class of Repository interface).**

   Entity object should persist irrespective of the lifecycle (start and stop of server) of application.

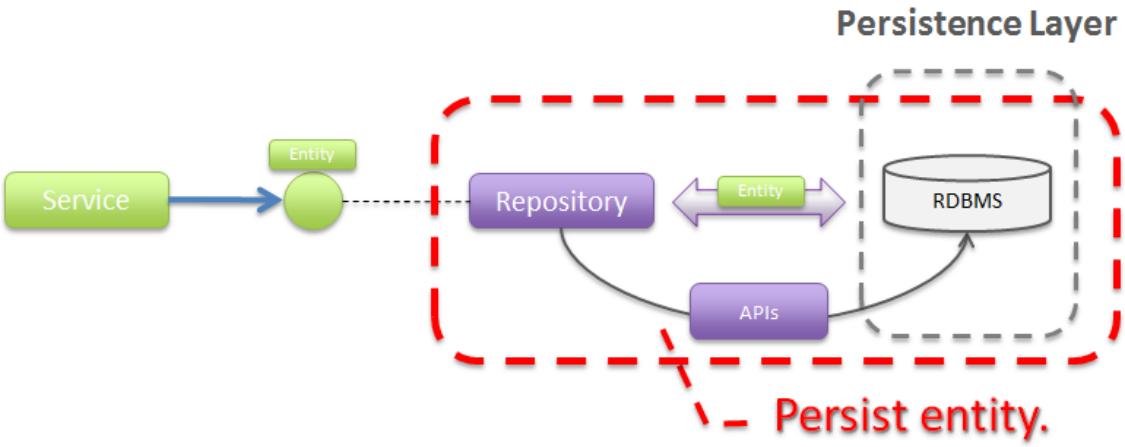
   Mostly relational database is the permanent destination of Entity. However, NoSQL database, cache server, external system and file (shared disk) can also be the permanent destination.

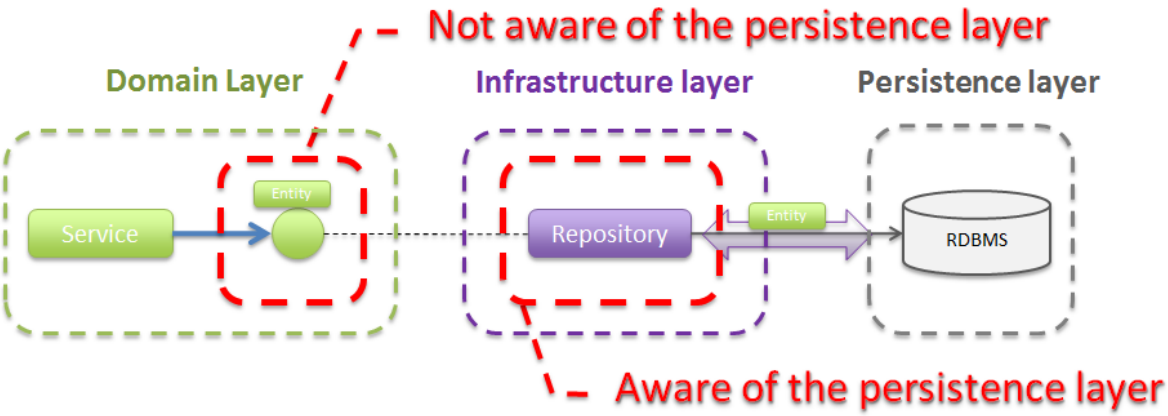
   The actual persistence processing is done using O/R Mapper API.

   This role is implemented in the RepositoryImpl of the infrastructure layer. Refer to *Implementation of Infrastructure Layer*for the details.

Structure of Repository

Repository consists of Repository interface and RepositoryImpl and performs the following roles.

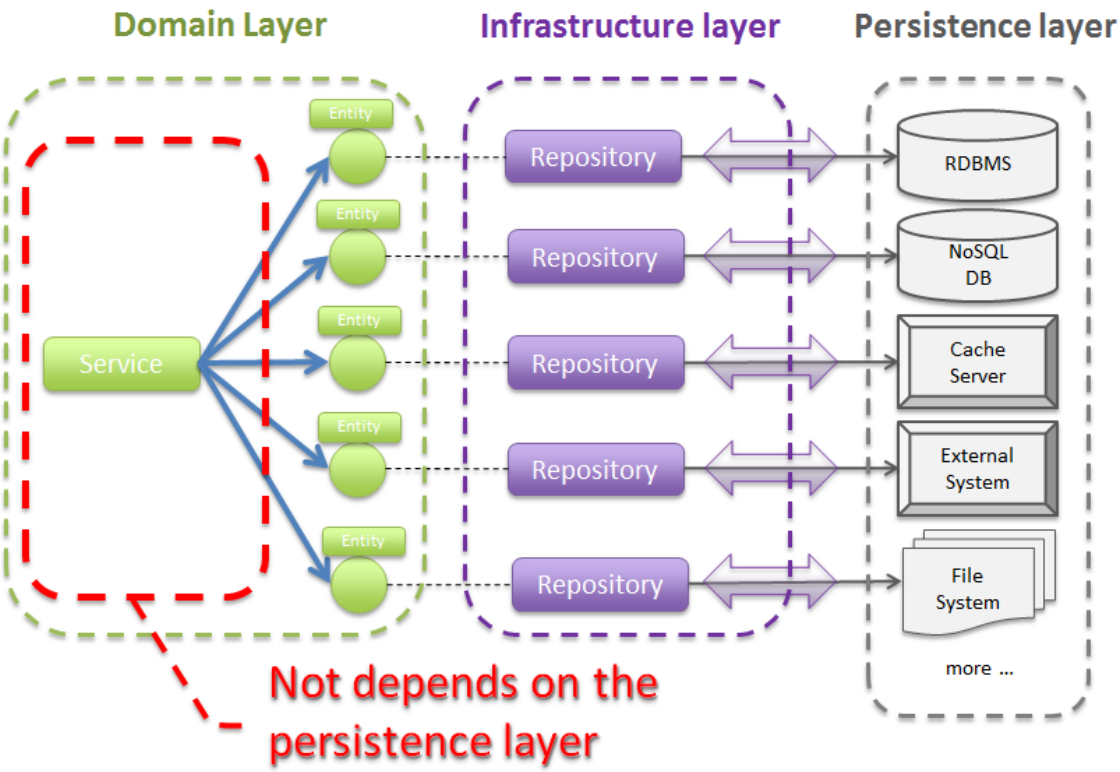




S.No.	Class(Interface)	Role	Description
(1)	Repository interface	Defines methods to control Entity lifecycle required for implementing business logic (Service).	Defines methods for CRUD operations of the Entity and is not dependent on persistence layer. Repository interface belongs to the domain layer since it plays the roles of defining the operations on Entity required for implementing business logic (Service).
(2)	RepositoryImpl	Implements the methods defined in Repository interface.	Implements CRUD operations of the Entity and is dependent on persistence layer. Performs actual CRUD processes using API that performs persistence provided by Spring Framework, O/R Mapper and middleware. RepositoryImpl belongs to infrastructure layer since it plays the role of implementing the operations defined in Repository interface. Refer to *Implementation of Infrastructure*

In case of multiple destinations in persistence layer, the resulting configuration as follows.

due to this, the logic depending on persistence platform of Entity is hidden from business logic (Service).



Note: Is it possible to hide 100% of persistence platform dependent logic from the Service class ?

In some cases it cannot be hidden completely due to constraints of persistence platform and the libraries used to access the platform. As much as possible, platform dependent logic should be implemented in RepositoryImpl instead of Service class. When it is difficult to exclude the platform dependent logic and merits of doing so are less, persistence platform dependent logic can be implemented as a part of business logic (Service) process.

A specific example of this is given here. There are cases when unique constraints violation error is needed to be handled when save method of `org.springframework.data.jpa.repository.JpaRepository` interface provided by Spring Data JPA is called. In case of JPA, there is a mechanism of cache entity operations and SQL is executed when transactions are committed. Therefore, since SQL is not executed even if save method of JpaRepository is called, unique constraints violation error cannot be handled in logic. There is a method (flush method) to reflect cached operations as means to explicitly issue SQLs in JPA. saveAndFlush and flush methods are also provided in JpaRepository for the same purpose. Therefore, when unique constraints violation error needs to be handled using JpaRepository of Spring Data JPA, JPA dependent method (saveAndFlush or flush) must be called.

> **Warning:** The most important purpose of creating Repository is not to exclude the persistence platform dependent logic from business logic. The most important purpose is to limit the implementation scope of business logic (Service) to the implementation of business rules. This is done by separating the operations to access business data in Repository. As an outcome of this, persistence platform dependent logic gets implemented in Repository instead of business logic (Service).

Creation of Repository

Repository must be created using the following policy only.

S.No.	Method	Supplementary
1.	Create Repository for the main Entity only.	This means separate Repository for operations of related Entity is not required. However, there are case when it is better to provide Repository for the related Entity in specific applications (for example, application having high performance requirements etc).
2.	Place Repository interface and RepositoryImpl in the same package of domain layer.	Repository interface belongs to domain layer and RepositoryImpl belongs to infrastructure layer. However, Java package of RepositoryImpl can be same as the Repository interface of domain layer.
3.	Place DTO used in Repository in the same package as Repository interface.	For example, DTO to store search criteria or summary DTO for that defines only a few items of Entity.

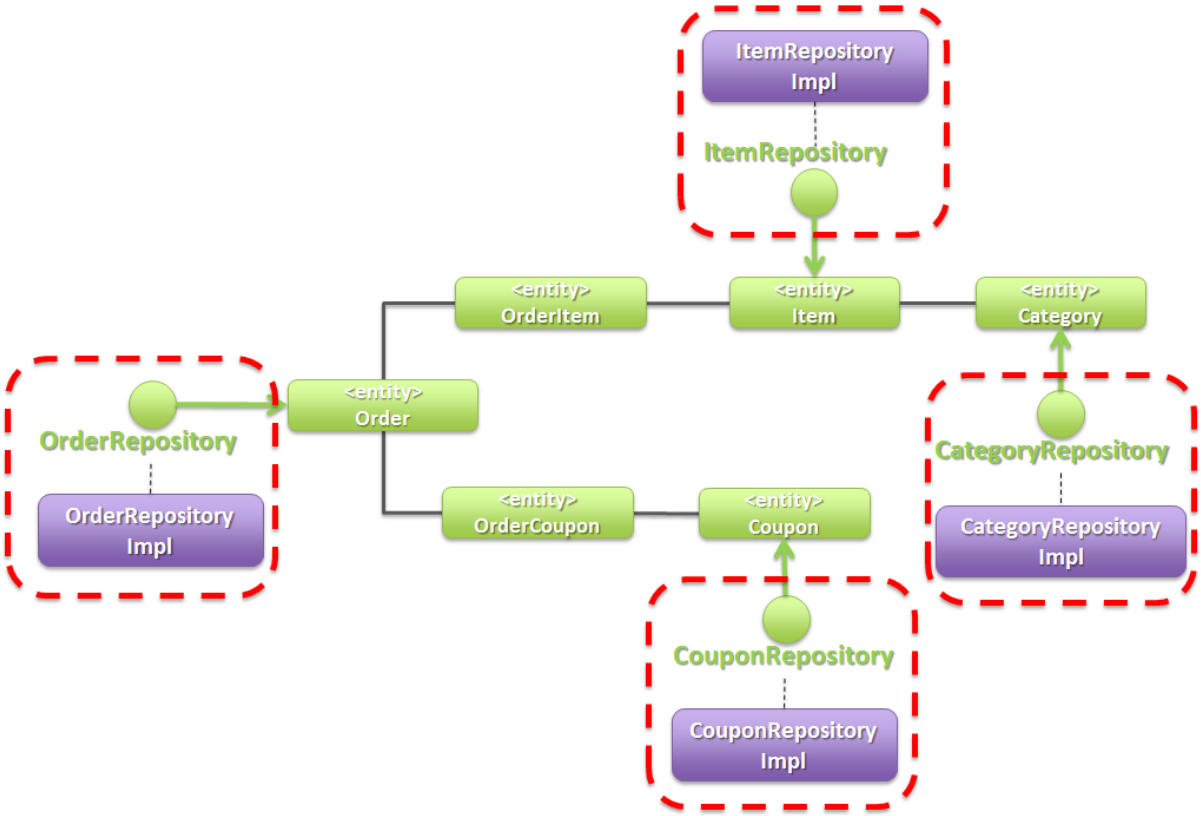
Example of creating Repository

An example of creating Repository is explained here.

An example of creating Repository of Entity class used in the explanation of *Example of creating Entity class*is as follows.

Structure of Repository

Entity class used in the explanation of *Example of creating Entity class*is used as an example, the resulting configuration is as follows:



Repository is created for the main Entity class.

Refer to *Project structure* for the recommended package structure.

Definition of Repository interface

Creation of Repository interface

An example of creating Repository interface is introduced below.

- `SimpleCrudRepository.java`

  This interface provides only simple CRUD operations.

  Method signature is created by referring to `CrudRepository`interface and `PagingAndSortingRepository`provided by Spring Data.

```
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);
}
```

S.No.	Description
(1)	Method to fetch the Entity object of specified ID.
(2)	Method to determine if the Entity of specified ID exists or not.
(3)	Method to retrieve the list of all Entities. In Spring Data, it was `java.util.Iterable`. Here as a sample, it is set to `java.util.List`.
(4)	Method to fetch collection of Entity objects corresponding to the specified pagination information (start position, record count, sort information). `Pageable` and `Page`are the interfaces provided by Spring Data.
(5)	Method to fetch total number of Entity objects.
(6)	Method to save (create, update) the specified Entity collection.
(7)	Method to delete the specified Entity.

- `TodoRepository.java`

  An example of creating Repository of Todo Entity, which was created in tutorial, on the basis of `SimpleCrudRepository` interface created above is shown below.

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

S.No.	Description
(1)	TodoRepository interface is created by specifying Todo entity in the generic type parameter “T” and String class in the generic type parameter “ID”.
(2)	Methods not provided by `SimpleCrudRepository`interface are added in this interface. In this case, “Method for acquiring count of Todo entity objects for which specified tasks have been finished” is added.

Method definition of Repository interface

It is recommended to have the same signature as `CrudRepository`and `PagingAndSortingRepository`provided by Spring Data for the methods performing general CRUD operations.

However, in case of returning collection, (`java.util.Collection`or `java.util.List`) interfaces which can be handled in a better way in logic are better than `java.lang.Iterable`.

In real development environment, it is difficult to develop an application using only general CRUD operations. Hence additional methods are required.

It is recommended to add the methods as per the following rules.

S.No.	Types of methods	Rules
1.	Method for searching a single record	1. Method name beginning with **findOneBy**to indicate that this method fetches a single record that matches with the condition. 2. In the method name after "findOneBy", physical or logical name of the field used as search condition must be specified. Hence, the method name must be such that it becomes possible to estimate "the kind of entity that can be fetched using this method". 3. There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided. 4. Return value must be Entity class.
2.	Method for searching multiple records	1. Method name beginning with **findAllBy**to indicate that this method fetches all the records that matches with the condition. 2. In the method name after "findAllBy", physical or logical name of the field used as search condition must be specified. Hence, the method name must be such that it becomes possible to estimate "the kind of entity that can be fetched using this method". 3. There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided. 4. Return value must be collection of Entity class.
3.	Method for searching multiple records with pagination	1. Method name beginning with **findPageBy**to indicate that this method fetches pages that matches with the condition. 2. In the method name after "findPageBy", physical or logical name of the field used as search condition must be specified. Hence, the method name must be such that it becomes possible to estimate "the kind of entity that can be fetched using this method". 3. There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided. `Pageable` provided by Spring Data should be the interface for pagination information (start position, record count, sort information). 4. Return value should be `Page` interface provided by Spring Data.
4.	Count related method	1. Method name beginning with **countBy** to indicate that this method fetches count of Entities which matches with the condition. 2. Return value must be long type. 3. In the method name after "countBy", physical or logical name of the field used as search condition must be specified. Hence, the method name must be such that it becomes possible to estimate "the kind of entity that can be fetched using this method". 4. There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided.

Note: In case of methods related to update processing, it is recommended to construct methods in the same way as shown above. “find” in the method name above can be replaced by “update” or “delete”.

- `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);
}
```

S.No.	Description
(1)	Example of method that fetches TODO objects whose title matches with specified value (TODO in which todoTitle=[argument value]). Physical name(todoTitle) of condition field is specified after findOneBy.
(2)	Example of method that fetches unfinished TODO objects (TODO objects where finished=false). Logical condition name is specified after findAllBy.
(3)	Example of method that fetches pages of unfinished TODOs (TODO objects where finished=false). Logical condition name is specified after findPageBy.
(4)	Example of method that fetches count of TODO objects for which the finish deadline has already passed (TODO for which createdAt < sysdate - [finish deadline in days] && finished=false). Logical condition name is specified after countBy.
(5)	Example of method that checks whether a TODO is created on a specific date (createdAt=specified date). Physical name (createdAt) is specified after existsBy.

Creation of RepositoryImpl

Refer to *Implementation of Infrastructure Layer*for the implementation of RepositoryImpl.

4.1.5 Implementation of Service

Roles of Service

Service plays the following 2 roles.

1. **Provides business logic to Controller.**

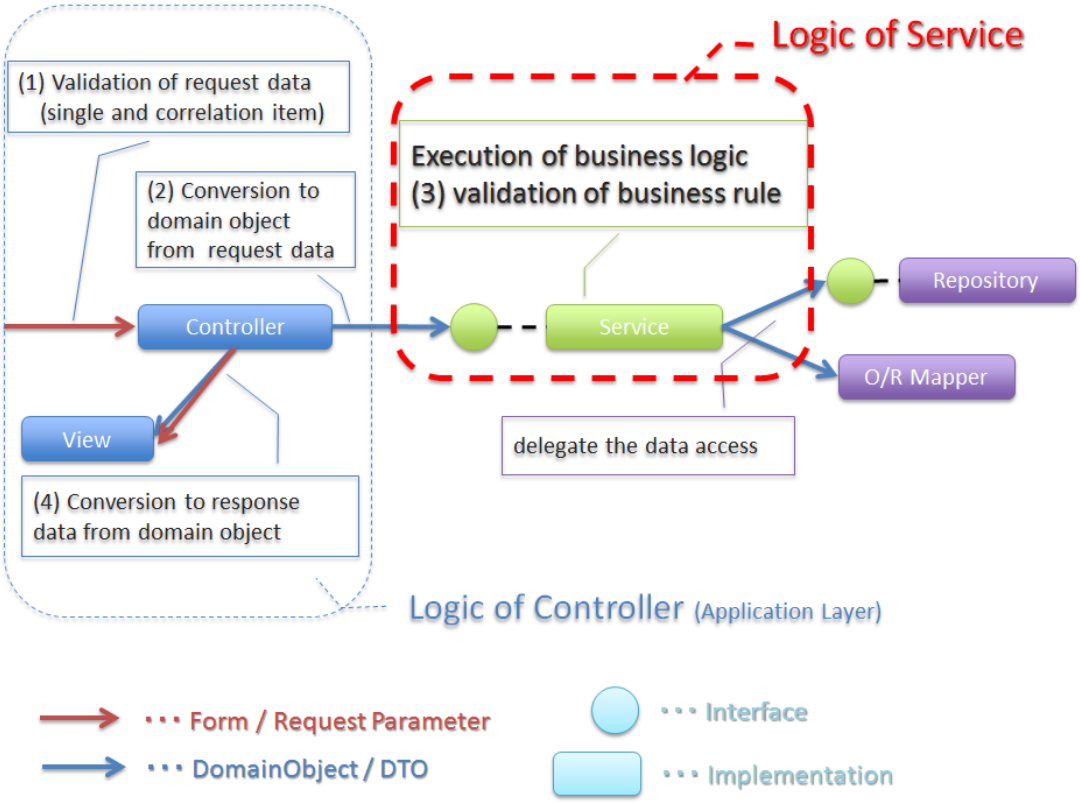
Business logic consists of create, update, consistency check etc of business data as well as all the processes related to business logic.

Create and update process of business data should be delegated to Repository(or O/R Mapper) and **service should be limited to implementation of business rules.**

Note: Regarding distribution of logic between Controller and Service

In this guideline, the logic to be implemented by Controller and Service should be as per the rules given below.

1. For the data requested from the client, single item check and correlated item check is to be performed in Controller (Bean Validation or Spring Validator).
2. Conversion processes (Bean conversion, Type conversion and Format conversion) for the data to be passed to Service, must be performed in Controller instead of Service.
3. **Business rules should be implemented in Service.**Access to business data is to be delegated to Repository or O/R Mapper.
4. Conversion processes (Type conversion and Format conversion) for the data received from Service (data to respond to the client), must be performed in Controller (View class etc).

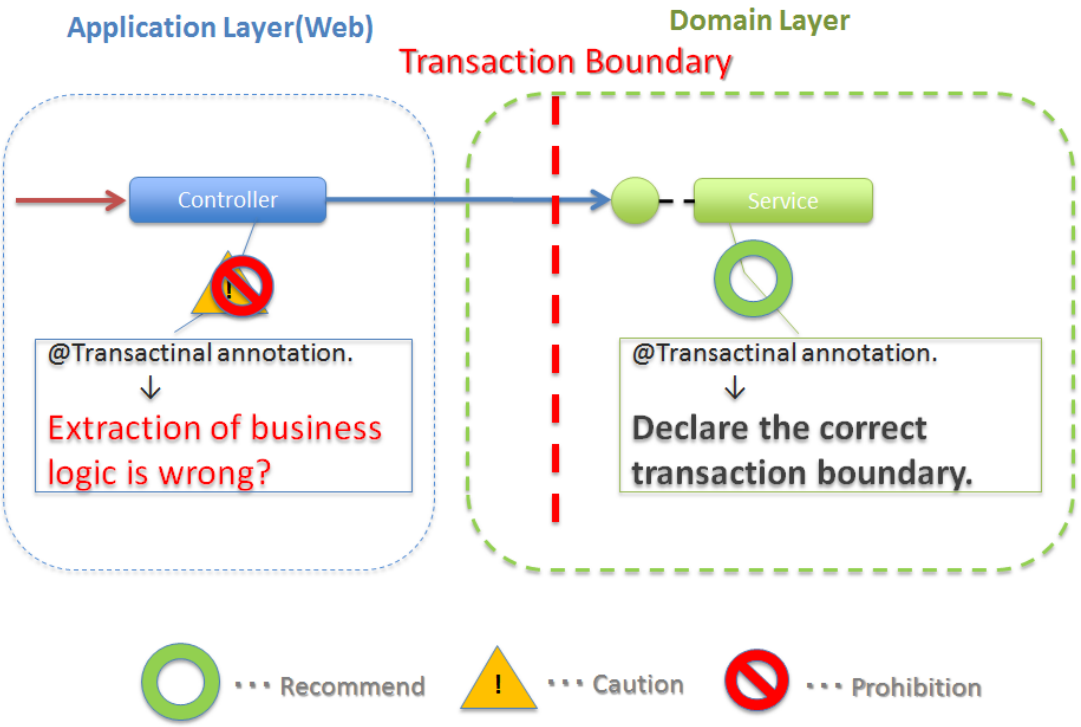


2. **Declare transaction boundary.**

Declare transaction boundary when business logic is performing any operation which requires ensuring data consistency (mainly data update process).

Even in case of logic that just read the data, often there are cases where transaction management is required due to the nature of business requirements. In such cases, declare transaction boundary.

Transaction boundary must be set in Service layer as a principle rule. If it is found to be set in application layer (Web layer), there is a possibility that the extraction of business logic has not been performed correctly.



Refer to *Regarding transaction management*for details.

Structure of Service class

Service consists of Service classes and SharedService classes and plays the following role.

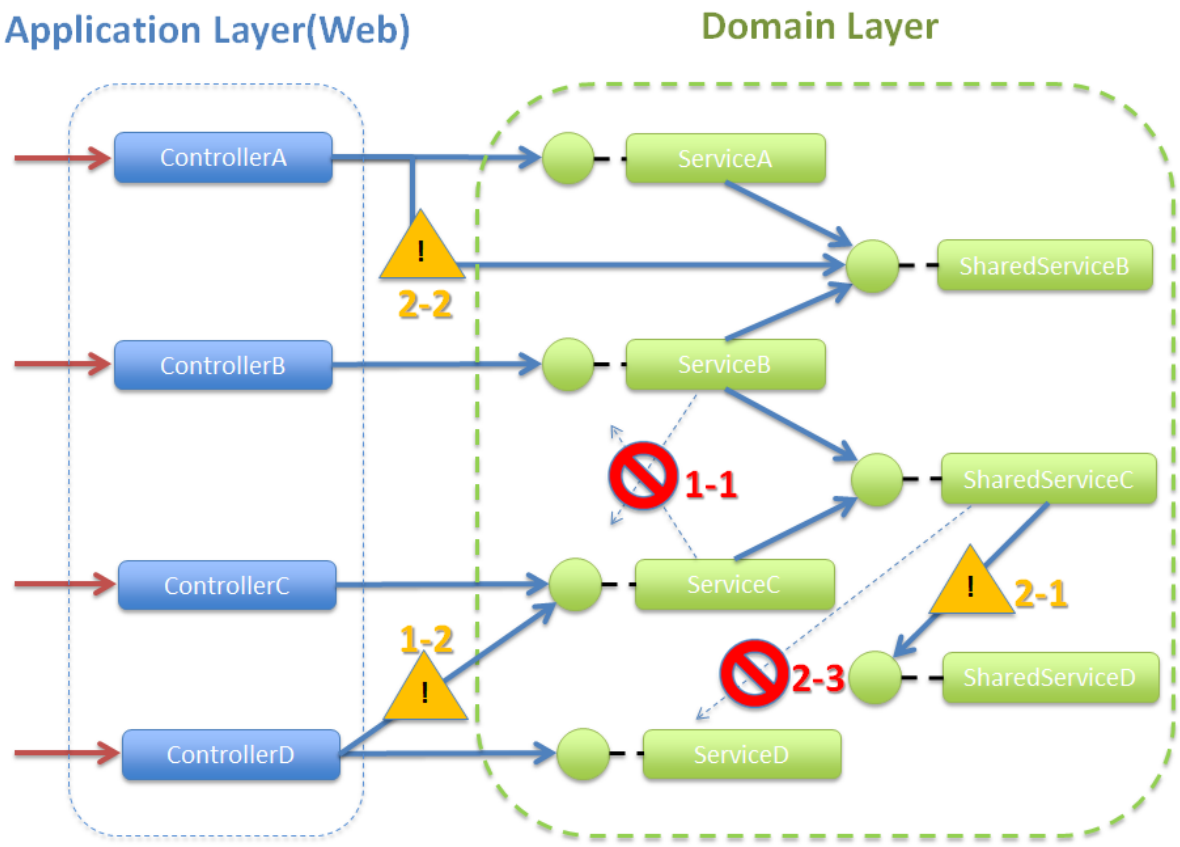
In this guideline, POJO (Plain Old Java Object) having `@Service`annotation is defined as Service or SharedService class.

We are not preventing the creation of interface and base classes that limit the signature of methods.

S.No.	Class	Role	Notes related to dependency relationship
1.	Service class	**Provides business logic to the specific Controller.** Service class methods **must not implement logic that need to be reused.**	1. **It is prohibited to call a method of Service class from another Service class method (Figure 1-1).**For shared logic, create SharedService class. 2. Method of Service class can be called from multiple Controllers (Figure 1-2). However, **it must be created for each controller when processing is to be branched based on the calling controller.**In such a scenario, create a method in SharedService class and call that method from the individual Service class methods.
2	SharedService class	**Provides shared (reusable) logic**for multiple Controllers and Service classes.	1. Methods of other SharedService classes can be called from a SharedService (Figure 2-1). However, **Calling hierarchy should not become complicated.** If calling hierarchy becomes complicated, there is a risk of reduction in maintainability. 2. Methods of SharedService classes can be called from Controller (Figure 2-2). However, **it can be only be done if there is no problem from transaction management perspective.**If there is a problem from transaction management perspective, first create a method in Service class and implement transaction management in this method. 3. **It is prohibited to call methods of Service class from SharedService (Figure 2-3).**

Dependency relationship of Service class and SharedService class is shown below.

The numbers inside the diagram are related to the numbering in “Notes related to dependency relationship” column of the above table.



Reason for separating Service and SharedServic

Logic that cannot be (should not be) reused and logic that can be (should be) reused exist in the business logic.

To implement these 2 logics in the same class, it is difficult to decide whether a method can be re-used or not.

To avoid this problem, **it is strongly recommended to implement the method to be re-used in the SharedService class**in this guideline.

Reason for prohibiting the calling of other Service classes from Service class

In this guideline, calling methods of other Service classes from a Service class is prohibited.

Service provides business logic to a specific controller and is not created with the assumption of using it from other services.

If it is called directly from other Service classes, the following situations can easily occur **and there is a risk of reduced maintainability.**

S.No.	Situations that can occur
1.	The logic that must be implemented in the calling service class, gets implemented in the called service class for reasons like “having the logic at a single location” etc. As a result, **arguments for identifying the caller, get added to the method easily; Ultimately, the logic is incorrectly abstracted out as shared logic (like utilities). It results into a modular structure without much insight.**
2.	If the stack patterns or stack of services calling each other is large in number, **understanding the impact of modifications in source-code due to change in specifications or bug fixes, becomes difficult.**

Regarding interface and base classes to limit signature of method

In order to bring consistency in development of business logic, interfaces and base classes are created which limit the signature of the methods.

The purpose is also to prevent the injection of diferences due to development style of each developer by limiting the signature through interfaces and base classes.

Note: In large scale development, there are situations where not every single developer is highly skilled or situations like having consistency in developemnt of business logic considering maintainability after servicing. In such situations, limiting the signature through interfaces can be an appropriate decision.

In this guideline, we do not specifically recommended to create interface to limit signature; however, type of architecture must be selected on the basis of characteristics of the project. decide the type of architecture taking into account the project properties.

Appendix has a sample of creating interface and base classes to limit the signature.

Refer to *Sample of implementation of interface and base classes to limit signature*for the details.

Patterns of creating service class

There are mainly 3 patterns for creating Service.

S.No.	Unit	Creation method	Description
1.	For each Entity	Create Service paired with the main Entity.	Main Entity is in other words, business data. **If the application is to be designed and implemented with focus on business data, then Service classes should be created in this way.** If service is created in this way, business logic will also be created for each entity and it will become to extract shared logic. However, if Service is created using this pattern, its affinity is not so good with the type of application which has to be developed by introducing a large number of developers at the same time. It can be said that the pattern is suitable when for the developing small or medium sized application.
2.	For each use-case	Create Service paired with the use-case.	**If the application is to be designed and implemented with focus on events on the screen, Service should be created in this way.** If the Service is created using this pattern, it is possible to assign a person to each use case; hence, its affinity is good with the type of application which has to be developed by introducing a large number of developer at the same time. On the other hand, if Service is created using this pattern, shared logic within use case can be extracted to a single location; however, shared logic which spans across multiple use-case might not get extracted to a single location. When it is very important to have extract shared logic out to a single location, it becomes necessary devise measures like having a separate team to look after designing shared components of business logic that span across multiple use cases.
3	For each event	Create Service paired with the events generated from	**If the application is to be designed and implemented with focus on events on the screen**

Warning: The pattern of Service creation must be decided by taking into account the features of application to be developed and the structure of development team.
It is not necessary to narrow down to any one pattern out of the 3 indicated patterns. Creating Services using different patterns randomly should be avoided for sure; however, **patterns can be used in combinations, if policy of usage of patterns in certain specific conditions has been well-thought decision and has been directed by the architect.** For example, the following combinations are possible.
[Example of usage of patterns in combination]

- For the business logic very important to the whole application, create as SharedService class for each Entity.
- For the business logic to be processed for the events from the screen, create as Service class for each Controller.
- In the Service class for each controller, implement business logic by calling the sharedService as and when required.

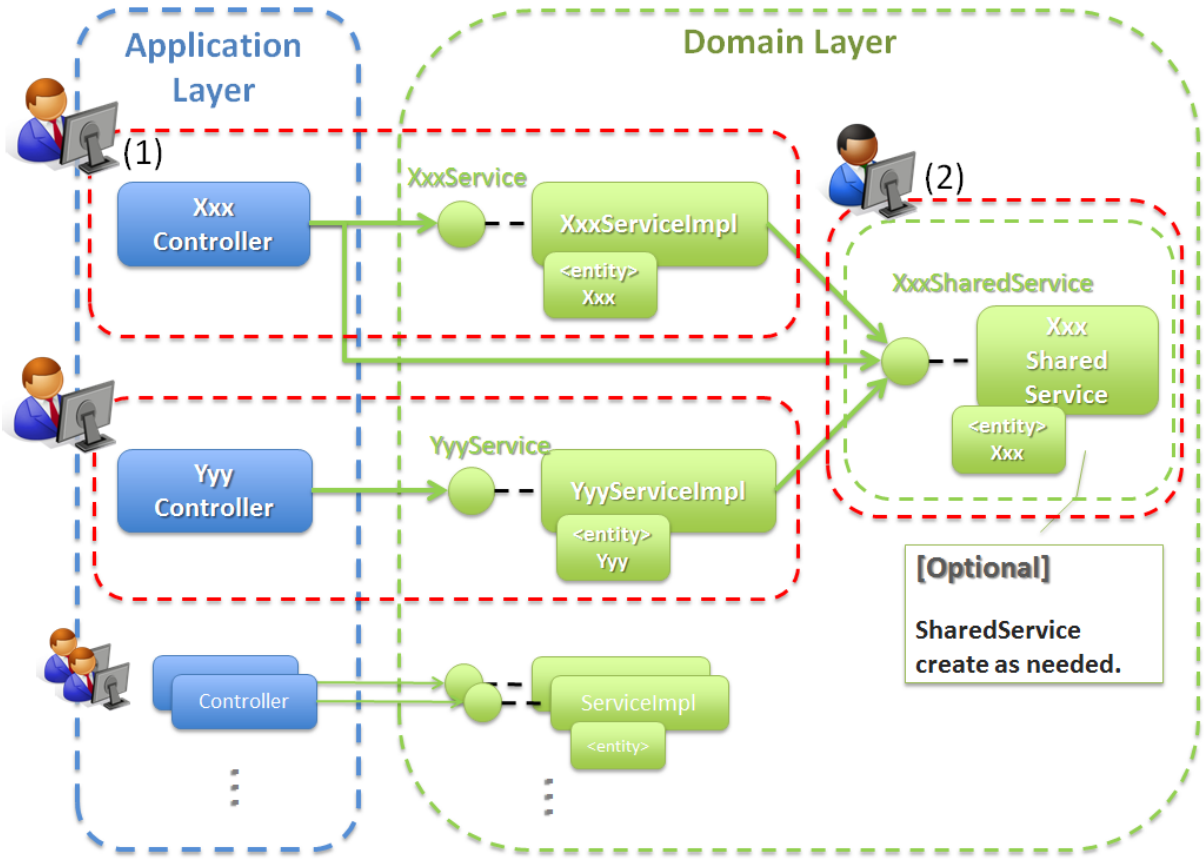
Tip: BLogic is generated directly from design documents when using “TERASOLUNA ViSC”.

Image of Application development - Creating Service for each Entity -

Following is the iamge of application development when creating a Service for each Entity.

Note: An example of a typical application in which a Service is created for each Entity is a REST application. REST application provides CRUD operations (POST, GET, PUT, DELETE of HTTP) for published resources on HTTP. Most of the times, the resources published on HTTP are business data (Entity) or part of business data (Entity), they have good compatibility with the pattern of creating Service for each Entity.

In case of REST application, most of the times, use-cases are also extracted on a “per Entity” basis. Hence, the structure is similar to the case when Service is created for on a “per use-case” basis.



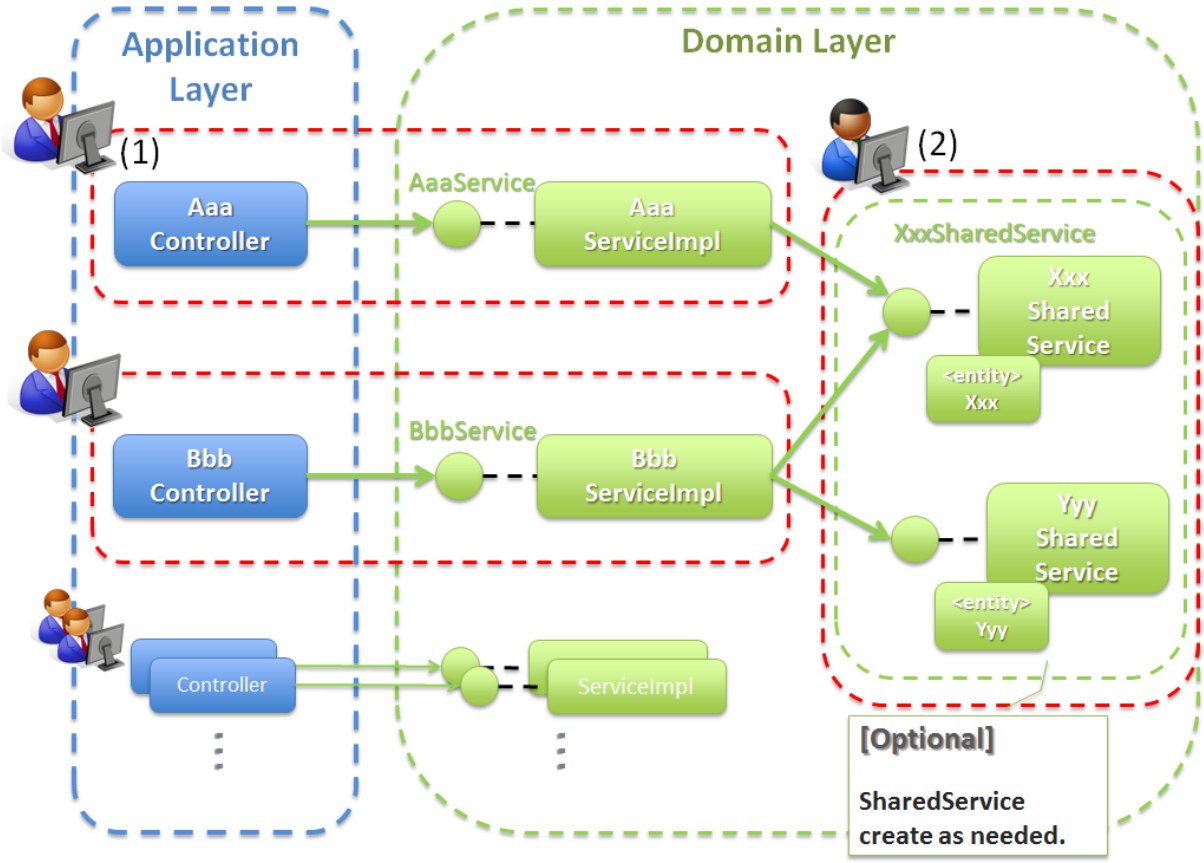
S.No.	Description
(1)	Implement Service by assigning a person for each Entity. If there is no specific reason, it is desirable that Controller must also be created for each Entity and must be developed by the same developer who created the Service class.
(2)	Implement SharedService if there is shared logic between multiple business logics. In the above figure, different person is assigned as the incharge. However, he may be the same person as (1) depending per the project structure.

Image of Application development - Creating Service for each use case

Following is the iamge of application development when creating a Service for each use-case.

In case of use-case which performs CRUD operations on the Entity, structure is same as in case of creating

Service for each Entity.

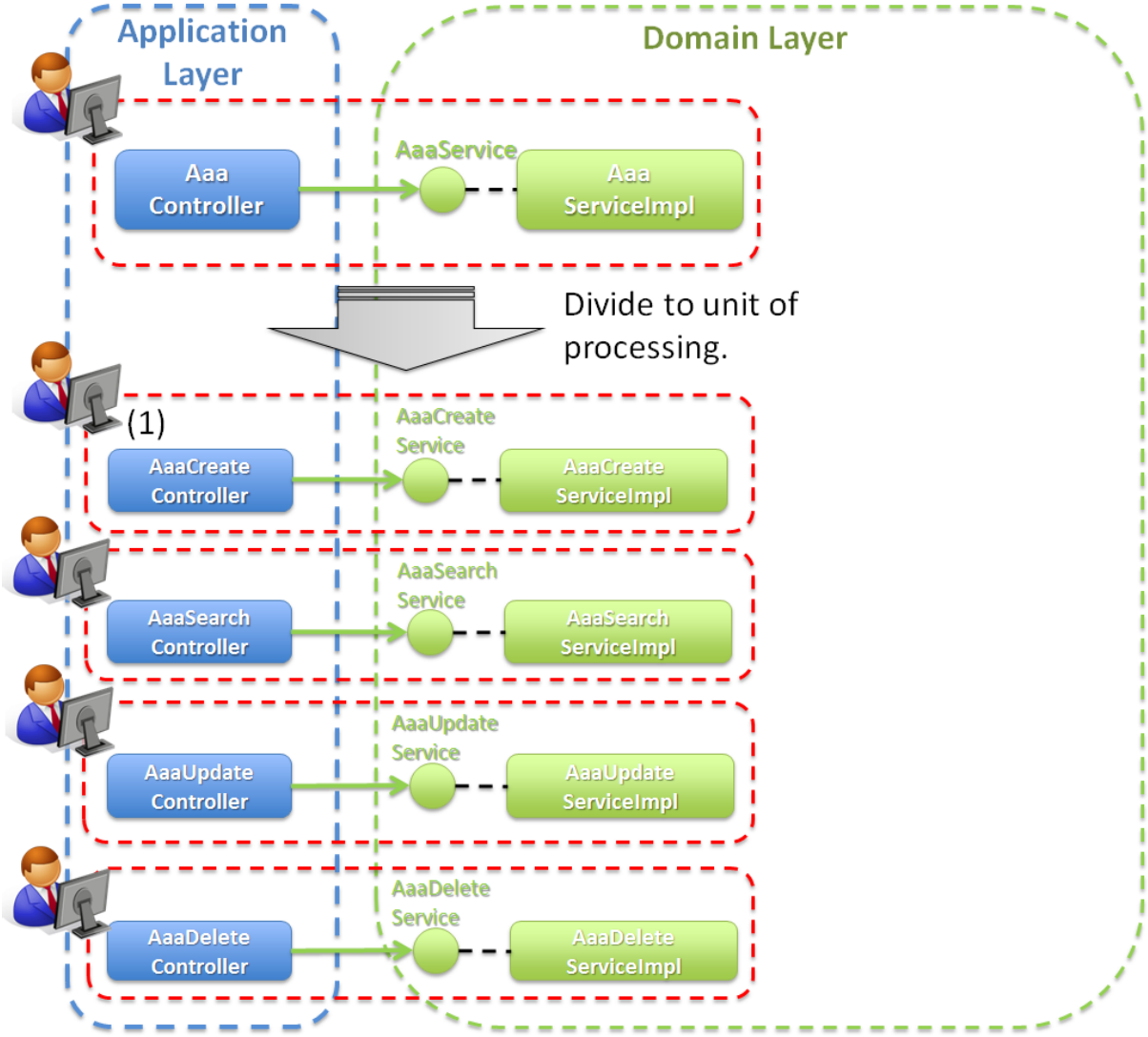


S.No.	Description
(1)	Implement Service by assigning a person for each use-case. If there is no specific reason, it is desirable that Controller must also be created for each use-case and must be developed by the same developer who created the Service class.
(2)	Implement SharedService if there is shared logic between multiple business logics. In the above figure, different person is assigned as the incharge. However, he may be the same person as (1) depending per the project structure.

Note: With an increase in the size of the use-cases, the development scope of a person increases. At such a point of time, it becomes difficult to divide the work of this use-case with other developers. In case of application which has to be developed by introducing a large number of developer at the same time, the use-case can be further split into finer use-cases and which can then be allocated to more number of developers.

Below is the image of application development when the use-case is further split.

Splitting a use-case has no impact on SharedService. Hence, the explanation is omitted here.

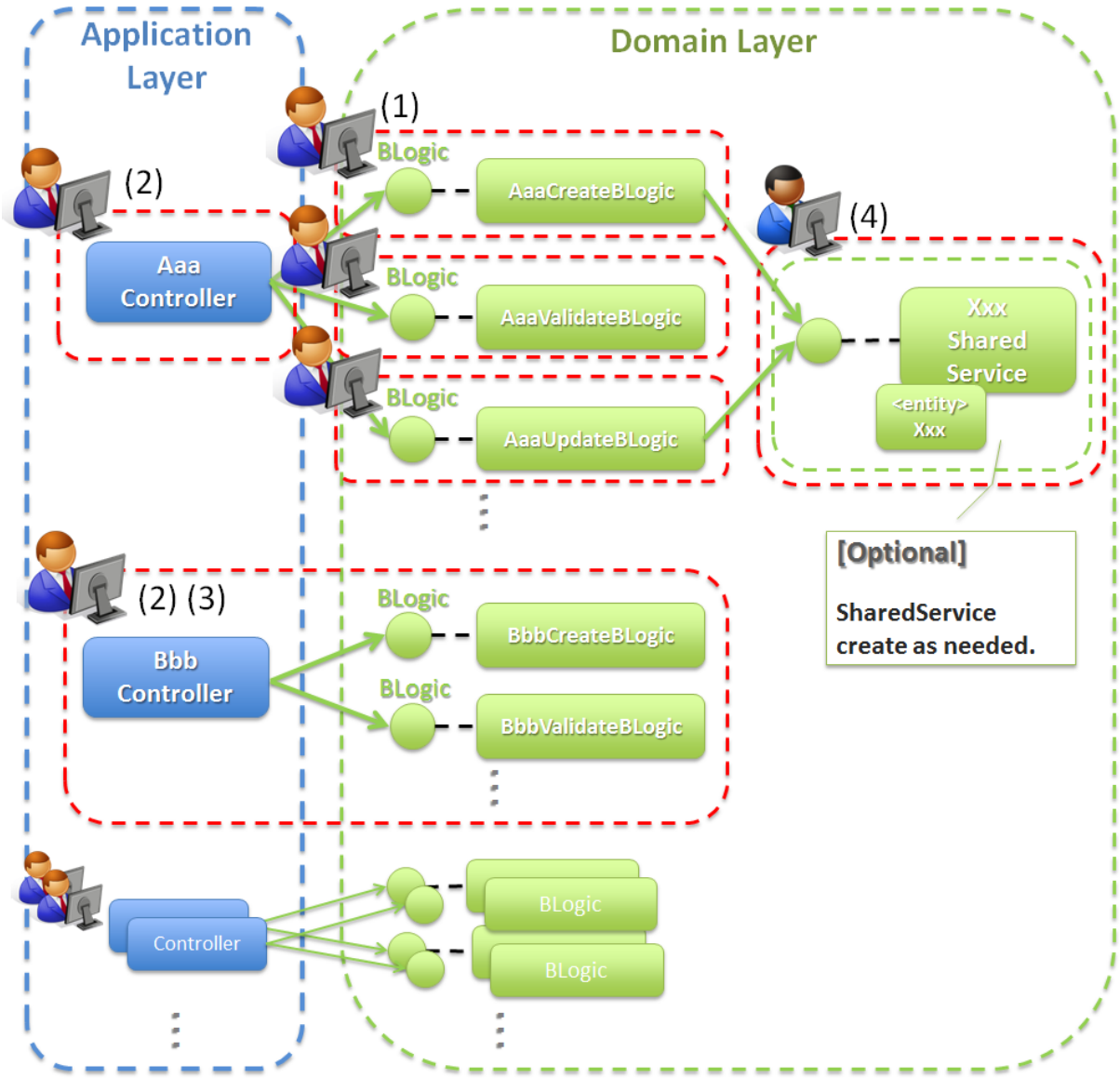


S.No.	Description
(1)	Divide the use-case into finer processes which make-up the complete use-case. Assign each fine process to a developer. Each developer creates the Service for assigned process. Note that the processes here are operations like search, create, update, delete etc. and these processes do not have a direct mapping to the processing required to be done for each event generated on screen. For example, if it the event generated on screen is “Update”, it includes multiple finer processes such as “Fetching the data to be updated”, “Compatibility check of update contents” etc. If there is no specific reason, it is desirable that Controller must also be created for each of these finer processes and must be developed by the same developer who creates the Service class.

Tip: In some projects, “group of use-cases” and “use-cases” are used in place of “use-case” and “processes” used in this guideline.

Image of Application development - Creating Service for each use event

Following is the iamge of application development when creating a Service(BLogic) for each event.



S.No.	Description
(1)	Implement Service(BLogic) by assigning a person for each event. Above example is an extreme case where separate developer is assigned for each service(BLogic). In reality, a single person must be assigned for a use-case.
(2)	If there is no specific reason, controller also should be created on “per use-case” basis.
(3)	Even if the separate Service(BLogic) is created for each event, it is recommended that same person is the in-charge of the complete use-case.

(4)	Implement in SharedService to share the logic with multiple business logics. In the above figure, different person is assigned as the incharge. However, he may be the same

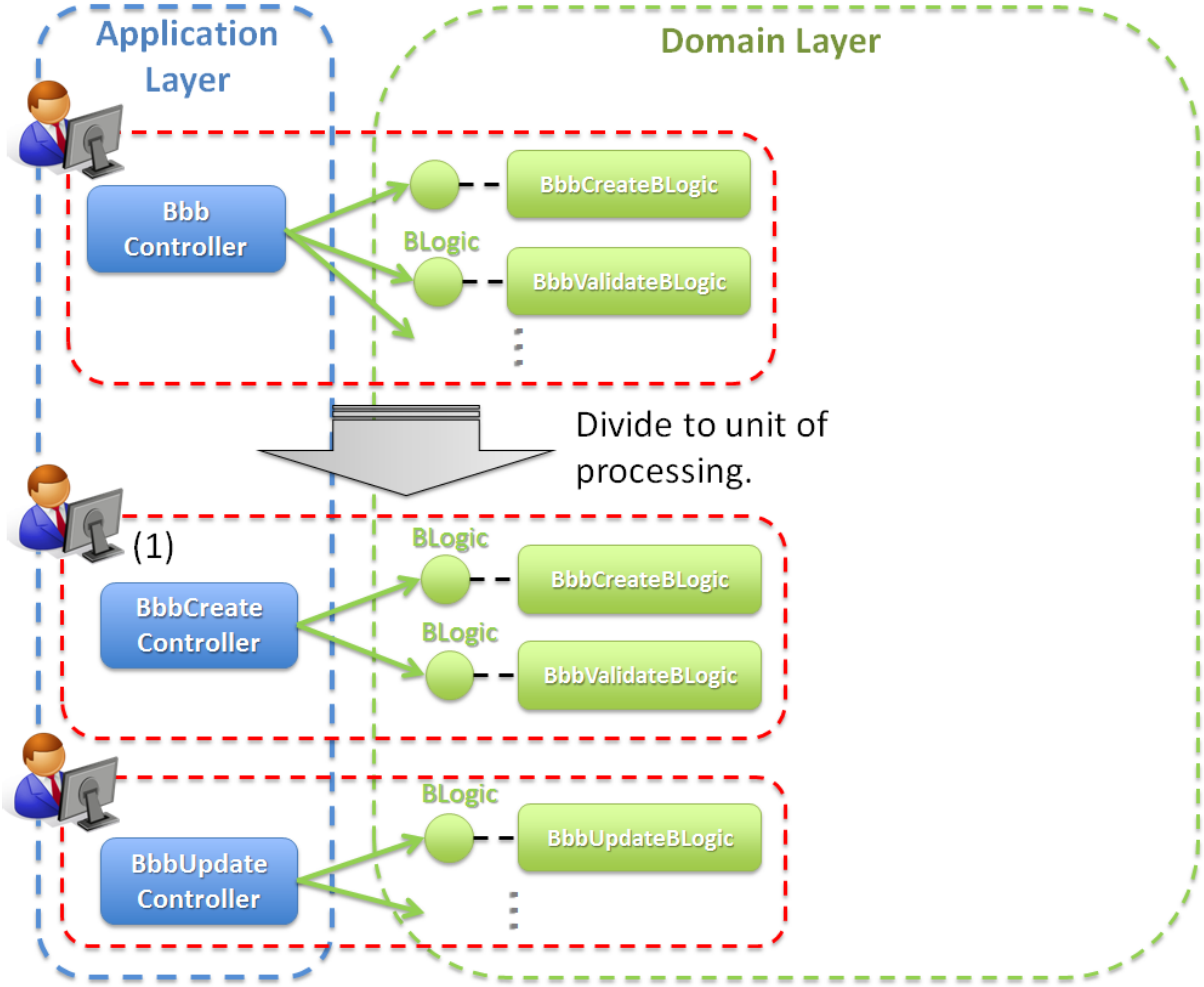
Note:

With an increase in the size of the use-cases, the development scope of a person increases. At such a point of time, it becomes difficult to divide the work of this use-case

with other developers. In case of application which has to be developed by introducing a large number of developer at the same time, the use-case can be further split into finer use-cases and which can then be allocated to more number of developers.

Below is the image of application development when the use-case is further split.

Splitting a use-case has no impact on SharedService. Hence, the explanation is omitted here.



S.No.	Description
(1)	Divide the use-case into finer processes which make-up the complete use-case. Assign each fine process to a developer. Each developer creates the Service for assigned process. Note that the processes here are operations like search, create, update, delete etc. and these processes do not have a direct mapping to the processing required to be done for each event generated on screen. For example, if it the event generated on screen is “Update”, it includes multiple finer processes such as “Fetching the data to be updated”, “Compatibility check of update contents” etc. If there is no specific reason, it is desirable that Controller must also be created for each of these finer processes and must be developed by the same developer who creates the Service class.

Creation of Service class

Methods of creating Service class

Below are the points to be taken care of while creating Service class.

- Creation of Service interface

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

S.No.	Description
(1)	**It is recommended to create Service interface.** By providing an interface, it is possible to execute the method published as Service explicitly.

Note: Merits from architecture perspective

1. If interface is there, When using AOP, Dynamic proxies functionality of standard JDK is used. In case of no interface, CGLIB included in Spring Framework is used. In case of CGLIB there are certain restrictions like “Advice can not be applied on final methods” etc. Refer to Spring Reference Documentfor details.

2. It becomes easier to create a stub of business logic. When application layer and domain layer are developed in parallel using different development teams, stubs of Service are required. When there is a need to create stubs, it is recommended to have interface .

- Creation of Service class

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

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

S.No.	Description
(1)	**Add @Service annotation to class.** By adding the above annotation, bean definition in configuration file is not required. Specify package for component scanning in `base-package` attribute of <context:component-scan> element. In case of this example, all the classes in "xxx.yyy.zzz.domain" is registered in container.
(2)	**Add @Transactional annotation to class.** By adding the above annotation, transaction boundary is set for to the all the methods of the Service class. `value` attribute should be specified as required. Refer to *Information required for "Declarative transaction management"*for details.
(3)	**Consider interface name as XxxService and class name as XxxServiceImpl.** Any naming conventions can be used. However, it is recommended to use distinguishable naming conventions for Service class and SharedService class.
(4)	**Service class must not maintain state. Register it in container as bean of singleton scope .** Objects (POJO such as Entity/DTO/VO) and values (primitive type, primitive wrapper class) where state changes in each thread should not be maintained in class level fields. Setting scope to any value other than singleton (prototype, request, session) using `@Scope` annotation is also prohibited.

Note: Reason for adding @Transactional annotation to class

Transaction boundary is required only for the business logic that updates the database. However, it is recommended to apply the annotation at class level to prevent bugs due to skipped annotation. However, defining `@Transactional`annotation only at required places (methods which update the database) is

also fine.

Note: Reason to prohibit non-singleton scopes

1. prototype, request, session are the scopes for registering bean that maintains state. Hence they must not be used in Service class.
2. When scope is set to request or prototype, performance is affected as the bean generation frequency is high in DI container.
3. When scope is set to request or session, it cannot be used in non Web applications (for example, Batch application).

Creation of methods of Service class

Below are the points to be taken care of while writing methods of Service class.

- Creation of method of Service interface

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

- Creation of methods of Service class

```
@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;
    }

}
```

S.No.	Description
(1)	**Create a method of Service class for each business logic.**
(2)	**Define methods in Service interface and implement business logic in its implementation class.**
(3)	**Add @Transactional annotation for changes to default transaction definition (class level annotation).** Attributes should be specified as per the requirement. Refer to *Information required for "Declarative transaction management"* for details.

Warning: Regarding transaction definition of business logic that just reads the database and does not update values

Transaction management for reference queries can be applied by through `@Transactional(readOnly = true)`. However, for JPA, there is no need to specify "readOnly = true". Refer to "Listing 7. Using read-only with REQUIRED propagation mode - JPA" of IBM DeveloperWorks articlefor details.

Note: Defining transaction when a new transaction is required to be started

Set `@Transactional(propagation = Propagation.REQUIRES_NEW)` to start a new transaction without participating in the transaction of the caller method.

Regarding arguments and return values of methods of Service class

The below points must be considered for arguments and return values of methods of Service class.

Serializable classes (class implementing `java.io.Serializable`) must be used for arguments and return values of Service class.

Since there is possibility of Service class getting deployed as distributed application, it is recommended to allow only Serializable class.

Typical arguments and return values of the methods are as follows.

- Primitive types (`int`, `long`)
- Primitive wrapper classes (`java.lang.Integer`, `java.lang.Long`)
- java standard classes (`java.lang.String`, `java.util.Date`)
- Domain objects (Entity, DTO)
- Input/output objects (DTO)
- Collection (implementation class of `java.util.Collection`) of above types
- void
- etc ...

Note: Input/Output objects

1. Input object indicates the object that has all the input values required for executing Service method.
2. Output object indicates the object that has all the execution results (output values) of Service method.

   If business logic(BLogic class) is generated using "TERASOLUNA ViSC" then, input and output objects are used as argument and return value of the of BLogic class.

Values that are forbidden as arguments and return values are as follows.

- Objects (`javax.servlet.http.HttpServletRequest`, `javax.servlet.http.HttpServletResponse`, `javax.servlet.http.HttpSession`, `org.springframework.http.server.ServletServerHttpRequest`) which are dependent on implementation architecture of application layer (Servlet API or web layer API of Spring).
- Model(Form, DTO) classes of application layer
- Implementation classes of `java.util.Map`

Note: Reason for prohibition

1. If objects depending on implementation architecture of application layer are allowed, then application layer and domain layer get tightly coupled.
2. `java.util.Map`is too generalized. Using it for method arguments and return values makes it difficult to understand what type of object is stored inside it. Further, since the values are managed using keys, the following problems may occur.

- Values are mapped to a unique key and hence cannot be retrieved by specifying a key name which is different from the one specified at the time of inserting the value.
- When key name has to be changed, it becomes difficult to determine the impacted area.

How to sharing the same DTO between the application layer and domain layer is shown below.

- DTO belonging to the package of domain layer can be used in application layer.

> **Warning:** Form and DTO of application layer should not be used in domain layer.

Implementation of SharedService class

Creation of SharedService class

Below are the points to be taken care of while creating SharedService class.

Only the points which are different from Service class are explained here.

1. **Add @Transactional annotation to class as and when required.**

   `@Transactional`annotation is not required when data access is not involved.

2. **Interface name should be XxxSharedService and class name should be XxxSharedServiceImpl.**

   Any other naming conventions can also be used. However, it is recommended to use distinguishable naming conventions for Service class and SharedService class.

Creation of SharedService class method

Below are the points to be taken care of while writing methods of SharedService class.

Only the points which are different from Service class are explained here.

1. **Methods in SharedService class must be created for each logic which is shared between multiple business logics.**

2. **Add @Transactional annotation to class as and when required.**

   Annotation is not required when data access is not involved.

Regarding arguments and return values of SharedService class method

Points are same as *Regarding arguments and return values of methods of Service class*.

Implementation of logic

Implementation in Service and SharedService is explained here.

Service and SharedService has implementation of logic related to operations such as data fetch, update, consistency check of business data and implementation related to business rules.

Example of a typical logic is explained below.

Operate on business data

Refer to the following for the examples of data (Entity) fetch and update.

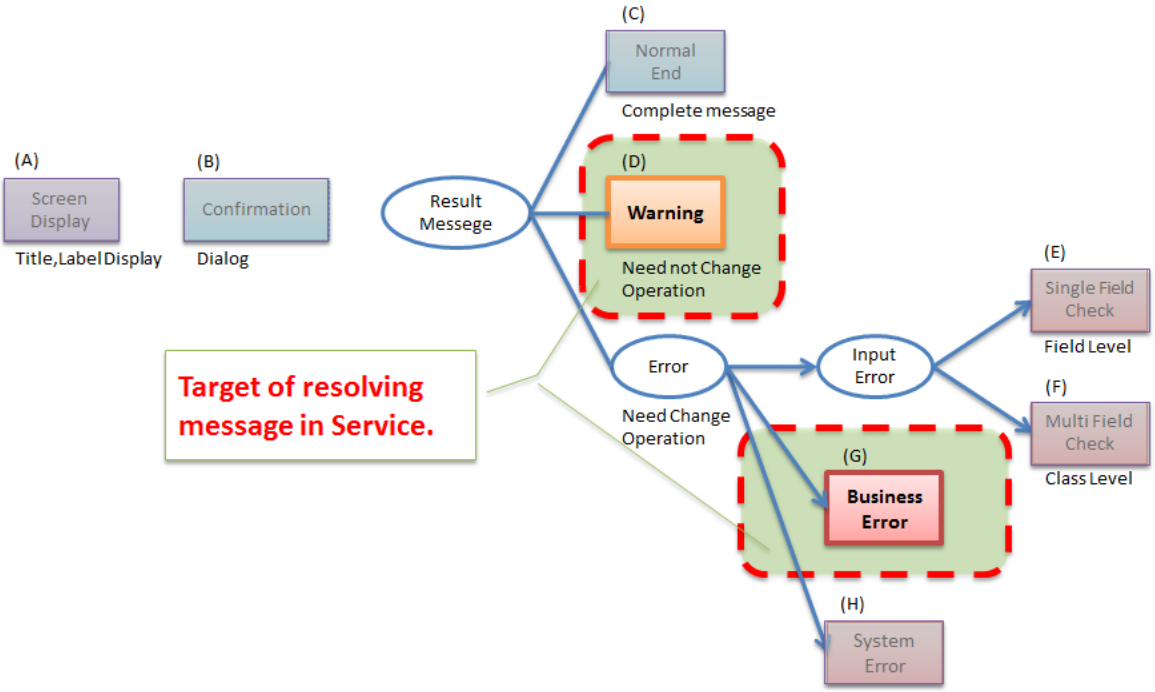
- When using JPA, *[coming soon] Database Access (JPA)*
- When using Mybatis2, *[coming soon] Database Access (MyBatis2)*

Returning messages

Warning message and business error message are the two type of messages which must be resolved in Service (refer to the figure in red broken line below).

Other messages should be resolved in application layer.

Refer to *Message Management*for message types and message pattern.



Note: Regarding resolving message

In service, instead of the actual message **the information required for building the message (message code, message insert value) is resolved**.

Refer to the following for detailed implementation method.

- *Returning warning message*
- *Notifying business error*

Returning warning message

Message object must be returned for warning message. If domain object such as Entity needs to be returned with it,
message object and domain object should inserted into output object (DTO) and this output object must be returned.

Message object (`org.terasoluna.fw.common.message.ResultMessages`) is provided as common library. When the class provided in common library
does not fulfill the requirements, message object should be created for each project.

- Creation of DTO

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

    // omitted

}
```

- Implementation of method of Service class

  Following is an example of implementation of displaying a warning message. The message is “Products may not be delivered together since the order includes products which are not available right now”.

```
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;
    }
```

S.No.	Description
(1)	When the order includes products which are not available right now, set `hasOrderProduct`to `true`.
(2)	In the above example, when the order includes products which are not available right now, a warning message occurs.
(3)	In the above example, the registered `Order` object and warning message are returned by storing objects in a DTO called `OrderResult`.

Notifying business error

Business exception is thrown when business rules are violated while executing business logic.

The following can be the cases.

- When reservation date exceeds deadline while making tour reservation
- When the product is out of stock at the time of placing an order
- etc ...

Business exception (`org.terasoluna.fw.common.exception.BusinessException`) is provided as common library.

When business exception class provided in common library does not fulfill the requirements, business exception class should be created in the project.

It is recommended to create business exception class as subclass of java.lang.RuntimeException.

> **Note: Reason for considering business exception as an unchecked exception**
>
> Since business exceptions need to be handled in controller class, they can be configured as checked exception. However in this guideline, it is recommended that business exception be subclass of unchecked exception (`java.lang.RuntimeException`). By default, if there is a RuntimeException, transaction will be rollbacked. Hence, doing this will prevent leaving a bug in the source-code due to inadequate settings of @Transactional annotation. Obviously, if settings are changed such that transaction rollbacks even in case checked exceptions, business exception can be configured as subclass of checked exceptions.

Example of throwing business exception.

Below example notifies that reservation is past the deadline and a business error.

```
// omitted

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

// omitted
```

S.No.	Description
(1)	Business exception is thrown since reservation date is past the deadline at the time of making reservation.

Refer to *Exception Handling*for the details of entire exception handling.

Notifying system error

System exception is thrown when error occurs in system while executing business logic.

The following can be the cases.

- When master data, directories and files that should already exist, do not exist
- When a checked exception generated by a library method is caught and this exception indicates abnoraml system state.
- etc ...

System exception (`org.terasoluna.fw.common.exception.SystemException`) is provided as common library.

When system exception class provided in common library does not fulfill the requirements, system exception class should be created in the project.

It is recommended to create system exception class as subclass of java.lang.RuntimeException.

The reason is system exception should not be handled by application code and rollback target of `@Transactinal`annotation is set to `java.lang.RuntimeException`by default.

Example of throwing system exception.

Example notifying the non-existence of the specific product in product master as system error is shown below.

```
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 + ".");
}
```

S.No.	Description
(1)	System exception is thrown since master data that should already exist does not exist. Example of case when system error is detected in logic）

Example that throws system exception while catching IO exception while copying the file is shown below.

```
// ...

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);
}
```

S.No.	Description
(1)	System exception that is classified into invalid system state is thrown by the library method. **The exception generated by library must be passed to system exception class as cause exception.** If cause exception is lost, error occurrence location and basic error cause can not be traced from the stacktrace.

Note: Regarding handling of data access error

When data access error occurs in Repository and O/R Mapper while executing business logic, it is converted to subclass of `org.springframework.dao.DataAccessException`and thrown. Error can be handled in application layer instead of catching in business logic. However, some errors like unique constraints violation error should be handled in business logic as per business requirements. Refer to *[coming soon] Database Access (Common)*for details.

4.1.6 Regarding transaction management

Transaction management is required in the logic where data consistency must be ensured.

Method of transaction management

There are various transaction management methods. However, in this guideline, **it is recommended to use "Declarative Transaction Management" provided by Spring Framework.**

Declarative transaction management

In "Declarative transaction management", the information required for transaction management can be declared by the following 2 methods.

- Declaration in XML(bean definition file).
- **Declaration using annotation (@Transactional) (Recommended).**

Refer to Spring Reference Documentfor the details of "Declarative type transaction management" provided by Spring Framework.

Note: Reason for recommending annotation method

1. The transaction management to be performed can be understood by just looking at the source code.
2. AOP settings for transaction management is not required if annotations are used and so XML becomes simple.

Information required for "Declarative transaction management"

Specify `@Transactional` annotation for at class level or method level which are considered as target of transaction management and specify the information required for
transaction control in attributes of `@Transactional` annotation.

S.No.	Attribute name	Description
1	propagation	Specify transaction propagation method. **[REQUIRE]** Starts transaction if not started. (default when omitted) **[REQUIRES_NEW]** Always starts a new transaction. **[SUPPORTS]** Uses transaction if started. Does not use if not started. **[NOT_SUPPORTED]** Does not use transaction. **[MANDATORY]** Transaction should start. An exception occurs if not started. **[NEVER]** Does not use transaction (never start). An exception occurs if started. **[NESTED]** save points are set. They are valid only in JDBC.
2	isolation	Specify isolation level of transaction. Since this setting depends on DB specifications, settings should be decided by checking DB specifications. **[DEFAULT]** Isolation level provided by DB by default.(default when omitted) **[READ_UNCOMMITTED]** Reads (uncommitted) data modified in other transactions. **[READ_COMMITTED]** Does not read (uncommitted) data modified in other transactions. **[REPEATABLE_READ]** Data read by other transactions cannot be updated. **[SERIALIZABLE]** Isolates transactions completely. Isolation level of transaction is considered as the parameter related to exclusion control. Refer to *[coming soon] Exclusive Control* for exclusion control.
3	timeout	Specify timeout of transaction (seconds).

Note: Location to specify the @Transactional annotation

It is recommended to specify the annotation at the class level or method level of the class. Must be noted that it should not interface or method of interface. Refer to 2nd Tip on Spring Reference Document for reason.

Warning: Default operations of rollback and commit when exception occurs

When rollbackFor and noRollbackFor is not specified, Spring Framework performs the following operations.

- Rollback when unchecked exception of (java.lang.RuntimeException and java.lang.Error) class or its subclass occurs.
- Commit when checked exception of (java.lang.Exception) class or its subclass occurs. **(Necessary to note)**

Note: Regarding value attributes of @Transactional annotation

There is a value attribute in `@Transactional`annotation. However, this attribute specifies which Transaction Manager to be used when multiple Transaction Managers are declared. It is not required to specify when there is only one Transaction Manager. When it is required to use multiple Transaction Managers, refer to Spring Reference Document.

Note: Default isolation levels of main DB are given below.

Default isolation levels of main DB are given below.

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

Propagation of transaction

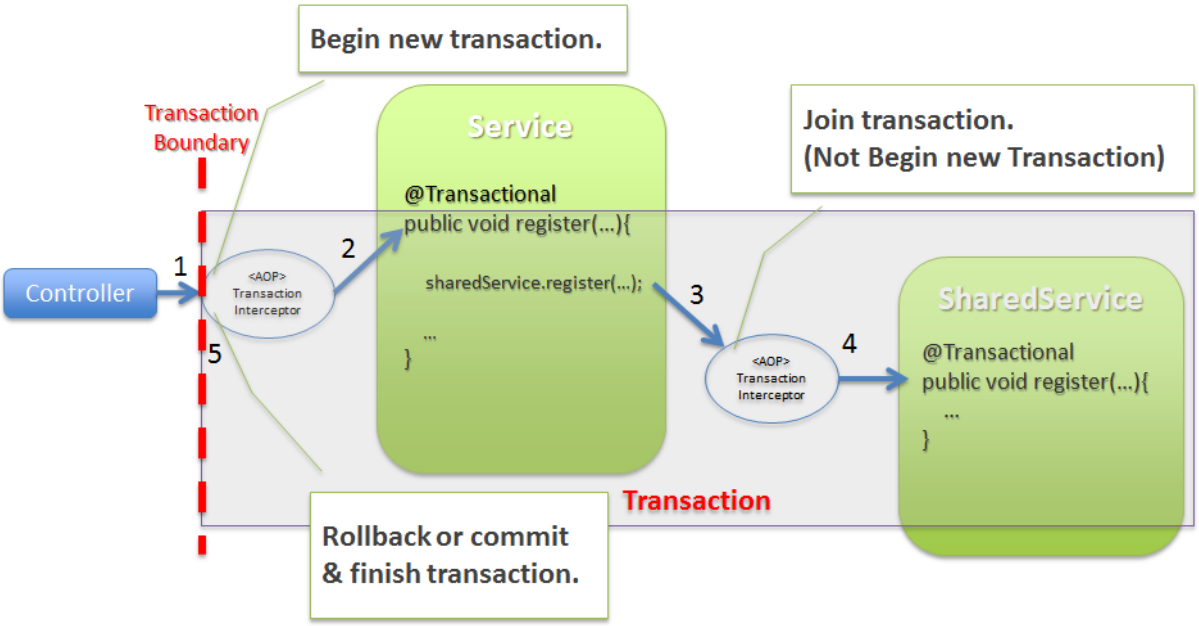
In most of the cases, Propagation method of transaction is “REQUIRED”.

However, since **“REQUIRES_NEW” is also used according to the requirements of Application**, transaction control flow in case of “REQUIRED” and “REQUIRES_NEW” is explained below.

The explanation of other propagation methods is omitted in this guideline since their usage frequency is very low.

Transaction control flow when propagation method of transaction is set to "REQUIRED"

When propagation method of transaction is set to "REQUIRED", all sequential processes called from controller are processed in the same transaction.



1. Controller calls a method of Service class. At this time, since started transaction does not exist, transaction is started using `TransactionInterceptor`.
2. `TransactionInterceptor`calls the method of service class after starting the transaction.
3. Service class calls a method of `SharedService`. This method is also under transaction control. At this time, though started transaction exists, `TransactionInterceptor`participates in the started transaction without starting a new transaction.
4. `TransactionInterceptor`calls the method under transaction control after participating in the started transaction.
5. `TransactionInterceptor`performs commit or rollback according to result of processing and ends the transaction.

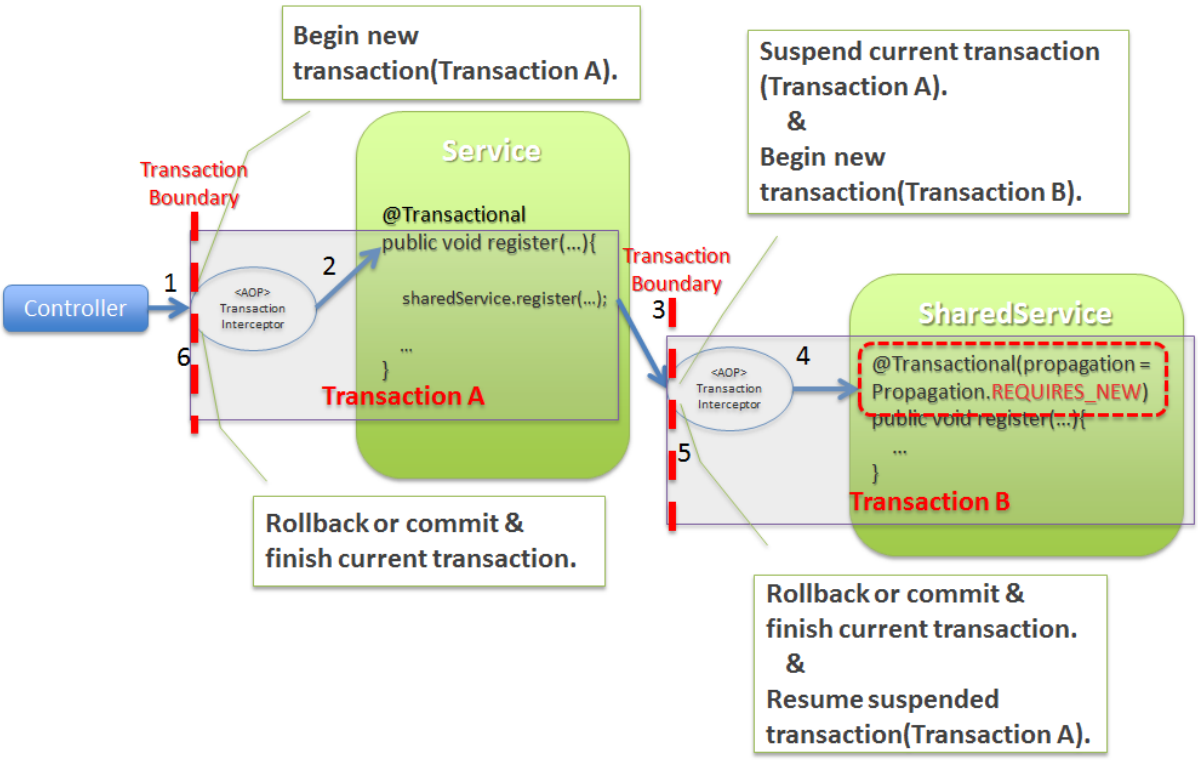
Note: Reason for occurrence of org.springframework.transaction.UnexpectedRollbackException

When propagation method of transaction is set to "REQUIRED", though there is only one physical transaction, internally Spring Framework creates transaction boundaries. In case of above example, when a method of SharedService is called, a TransactionInterceptor is started which internally provides transaction control boundary at SharedService level. Therefore, when an exception (which is set as target of rollback) occurs in `SharedService`method, status of transaction is set to rollback (rollback-only) by `TransactionInterceptor`. This transaction now cannot be committed. Going further, if the Service method

tries to commit this transaction due to conflicting settings of rollback target exception between Service method and SharedShared method, `UnexpectedRollbackException`is generated by Spring Framework notifying that there is inconsistency in transaction control settings. When UnexpectedRollbackException is generated, it should be checked that there is no inconsistency in rollbackFor and noRollbackFor settings.

Transaction management flow when propagation method of transaction is set to "REQUIRES_NEW"

When propagation method of transaction is set to "REQUIRES_NEW", a part of the sequence of processing (Processing done in SharedService) are processed in another transaction when called from Controller.



1. Controller calls a method of Service class. This method is under transaction control. At this time, since started transaction does not exist, transaction is started by `TransactionInterceptor` (Hereafter, the started transaction is referred as "Transaction A").

2. `TransactionInterceptor` calls the method of service class after transaction (Transaction A) is started.

3. Service class calls a method of `SharedService` class. At this time, though started transaction (Transaction A) exists, since propagation method of transaction is "REQUIRES_NEW", new transaction is started by `TransactionInterceptor`. (Hereafter, started transaction is referred as "Transaction B"). At this time, "Transaction A" is interrupted and the status changes to 'Awaiting to resume'.

4. `TransactionInterceptor`calls the method of SharedService class after Transaction B is started.
5. `TransactionInterceptor`performs commit or rollback according to the process result and ends the Transaction B. At this time, “Transaction A” is resumed and status is changed to Active.
6. `TransactionInterceptor`performs commit or rollback according to the process result and ends the Transaction A.

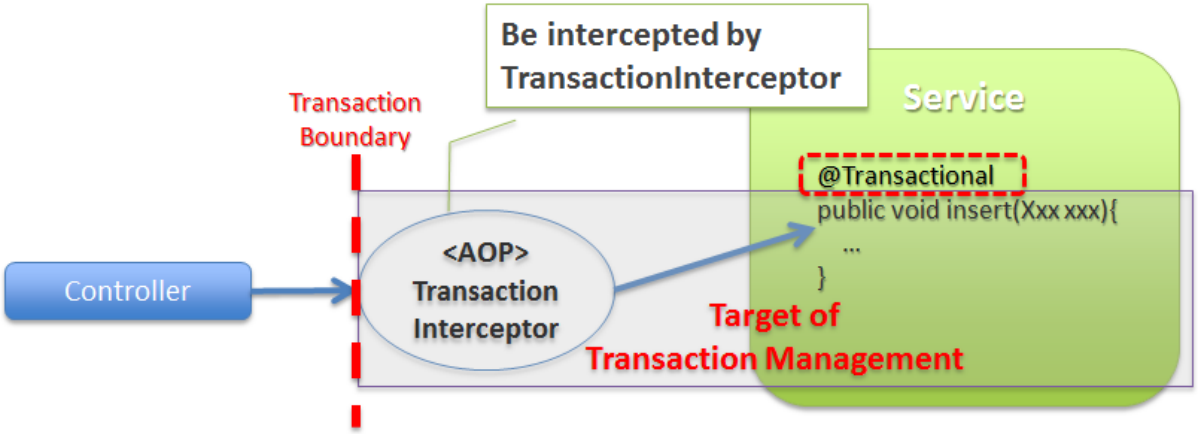
Way of calling the method which is under transaction control

Since “Declarative transaction management” provided by Spring Framework is implemented using AOP, transaction management is applied only for method calls for which AOP is enabled.

Since the default mode of AOP is **“proxy” mode, transaction control will be applied only when public method is called from another class.**

Note that **transaction control is not applied if the target method is called from an internal method even if the target method is a public method**.

- **Way of calling the method which is under transaction control**



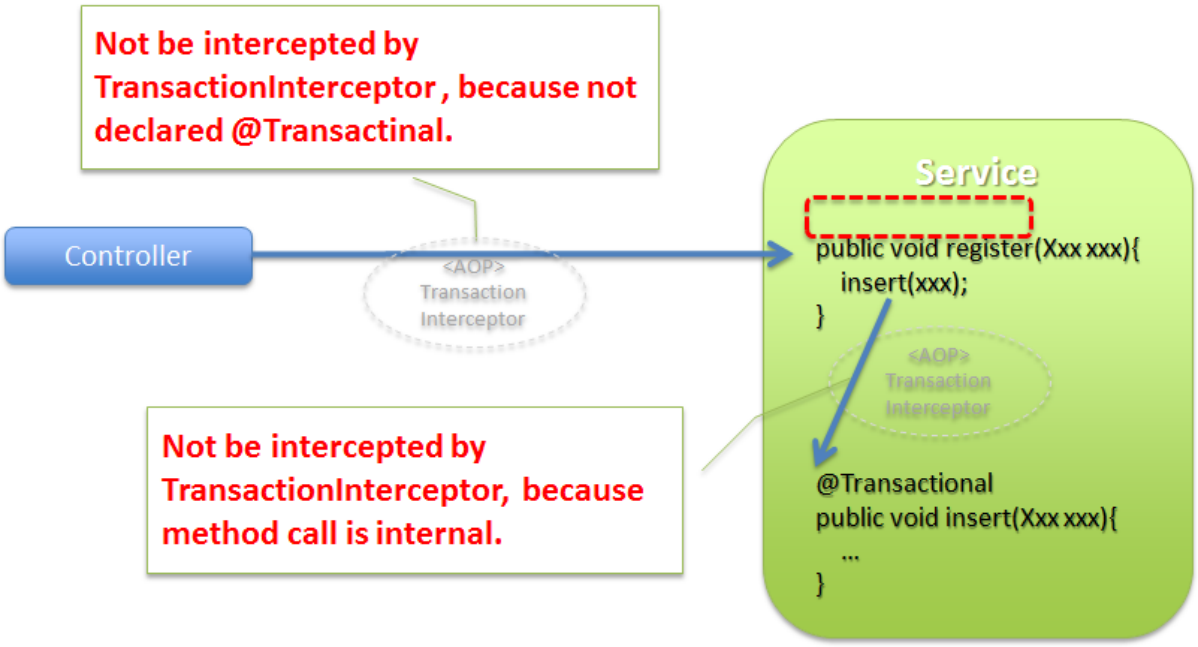
- **Way of calling the method which is not under transaction control**

Note: In order to bring internal method calls under transaction control

It is possible to enable transaction control for internal method calls as well by setting the AOP mode to “aspectj”. However, if internal method call of transaction management is enabled, the route of transaction management may become complicated; hence it is recommended to use the default “proxy” for AOP mode.

Settings for using transaction management

The settings required for using transaction management are explained.



PlatformTransactionManager settings

In order to have transaction management done, it is necessary to define the bean of `PlatformTransactionManager`.

Spring Framework has provided couple of classes based on the purpose; any class can be specified that meets the requirement of the application.

- `xxx-env.xml`

  Example of settings for managing the transaction using JDBC connection which is fetched from DataSource is given below.

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

S.No.	Description
(1)	Specify the implementation class of PlatformTransactionManager. It is recommended to set id as “transactionManager”.

Note: When transaction management (Global transaction management) is required for multiple DBs (Multiple resources)

- It is necessary to use `org.springframework.transaction.jta.JtaTransactionManager`and

manage transactions by using JTA functionality provided by application server.

- When JTA is to be used in WebSphere, Oracle WebLogic Server and Oracle OC4J, a `JtaTransactionManager` which is extended for the application server is automatically set by specifying <tx:jta-transaction-manager/>.

Table.4.1 Implementation class of PlatformTransactionManager provided by Spring Framework

S.No.	Class name	Description
1.	org.springframework.jdbc.datasource.DataSourceTransactionManager	Implementation class for managing the transaction by calling API of JDBC(`java.sql.Connection`). Use this class when Mybatis or `JdbcTemplate`is to be used.
2.	org.springframework.orm.jpa.JpaTransactionManager	Implementation class for managing the transaction by calling API of JPA(`javax.persistence.EntityTransaction`). Use this class when JPA is to be used.
3.	org.springframework.transaction.jta.JtaTransactionManager	Implementation class for managing the transaction by calling API of JTA(`javax.transaction.UserTransaction`). Use this class to manage transaction with resources (Database/Messaging service/General-purpose EIS(Enterprise Information System) etc.) using JTS (Java Transaction Service) provided by application server. When it is necessary to execute the operations with multiple resources in a single transaction, it is necessary to use JTA for managing transactions.

Settings for enabling @Transactional

In this guideline, it is recommended to manage transaction by using “Declarative transaction management” where `@Transactional`annotation is used.

Here, the settings required for using `@Transactional`annotation are explained.

- xxx-domain.xml

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

S.No.	Description
(1)	With use of <tx:annotation-driven> element in XML (bean definition file), transaction control gets enabled at the locations where `@Transactional`annotation is used.

Regarding attributes of <tx:annotation-driven> element

Various attributes can be specified in <tx:annotation-driven> and default behavior can be customized.

- xxx-domain.xml

```
<tx:annotation-driven
    transaction-manager="txManager"
    mode="aspectj"
    proxy-target-class="true"
    order="0" />
```

S.No.	Attribute	Description
1	transaction-manager	Specify `PlatformTransactionManager`bean. When omitted, bean registered with the name “transactionManager” is used.
2	mode	Specify AOP mode. When omitted, `"proxy"`is the default value. `"aspectj"`can be also be specified. It is recommended to use `"proxy"`.
3	proxy-target-class	Flag to specify whether proxy target is limited to class (this is valid only in case of mode=”proxy”). When omitted, it will be “false”. • In case of false, if the target class has an interface, proxy is done using dynamic proxies functionality of standard JDK. When there is no interface, proxy is done using GCLIB functionality. • In case of true, proxy is done using GCLIB function irrespective of whether interface is available or not.
4	order	Order of Advice of AOP (Priority). When omitted, it will be “Last (Lowest priority)”.

4.1.7 Appendix

Regarding drawbacks of transaction management

There is a description regarding “Understanding drawbacks of transaction” in IBM DeveloperWorks.

Read this article about pitfalls of transaction management and pitfalls while using @Transactional of Spring Framework. Refer to Article on IBM DeveloperWorksfor details.

Programmatic transaction management

In this guideline, “Declarative transaction management” is recommended. However, programmatic transaction management is also possible. Refer to Spring Reference Documentfor details.

Sample of implementation of interface and base classes to limit signature

- Interface to limit signature

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

S.No.	Description
(1)	Interface to limit signature of implementation method of business logic. In the above example, it is defined as generic type of input (I) and output (O) information having one method (execute) for executing business logic. In this guideline, the above interface is called BLogic interface.

- 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";
}
```

S.No.	Description
(2)	Controller injects calling BLogic interface.
(3)	Controller calls execute method of BLogic interface and executes business logic.

To standardize process flow of business logic when a fixed common process is included in Service, base classes are created to limit signature of method.

- Base classes to limit signature

```
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);

}
```

S.No.	Description
(4)	Call the method to perform pre-processing before executing business logic from base classes. In the preExecute method, business rules are checked.
(5)	Call the method executing business logic from the base classes.

Sample of extending base classes to limit signature is shown below.

- BLogic class (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;
    }

}
```

S.No.	Description
(6)	Implement pre-process before executing business logic. Business rules are checked.
(7)	Implement business logic. Logic is implemented to satisfy business rules.

4.1.8 Tips

Method of dealing with violation of business rules as field error

When it is necessary to output the error of business rules for each field, the mechanism of (Bean Validation or Spring Validator) on the Controller side should be used.

In this case, It is recommended to implement check logic as Service class and then to call the method of Service class from Bean Validation or Spring Validator.

Refer to *Input Validation*business logic approach for details.

4.2 Implementation of Infrastructure Layer

*Implementation of RepositoryImpl*is carried out in infrastructure layer.

RepositoryImpl implements the Repository interface.

4.2.1 Implementation of RepositoryImpl

The method to create Repository for relational database using JPA and MyBatis2 is introduced below.

- *Implementing Repository using JPA*
- *Implementing Repository using MyBatis2*

Implementing Repository using JPA

When JPA is to be used as persistence API with relational database, Repository can be very easily created if `org.springframework.data.jpa.repository.JpaRepository` of Spring Data JPA is used.
Refer to *[coming soon] Database Access (JPA)*for details regarding usage of Spring Data JPA.

When Spring Data JPA is used, only an interface, extending JpaRepository, is required to be created for basic CRUD operations. That means, RepositoryImpl is not required.
However, RepositoryImpl is needed for using dynamic query (JPQL).
Refer to *[coming soon] Database Access (JPA)*for implementing RepositoryImpl while using Spring Data JPA

- TodoRepository.java

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

S.No.	Description
(1)	Only by defining the interface that extends JpaRepository, basic CRUD operations for Todo entity can be performed without implementing the interface.

Procedure to add an operation not provided by JpaRepository is explained.

When Spring Data JPA is used, if it is a static query, the method must be added to the interface and the query (JPQL) to be executed must be specified in the annotation to the method in the interface.

- 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);
    // ...
}
```

S.No.	Description
(1)	Specify Query (JPQL) using @Query annotation.

Implementing Repository using MyBatis2

When MyBatis is used as persistence API, RepositoryImpl must be created as follows

Refer to *[coming soon] Database Access (MyBatis2)* for details regarding usage of MyBatis2

Furthermore, in this guideline, it is assumed that TERASOLUNA DAO which is a wrapper for MyBatis API is used instead of using MyBatis directly.

When MyBatis is to be used, **only the required method must be defined** in the Repository interface.

CrudRepository and PagingAndSortingRepository provided by Spring Data can also be used. However, using all the methods is very rare. Hence,

implementation of unnnecessary methods have to be done if these interfaces are used.

When MyBatis is to be used, RepositoryImpl and SQL definition file should be created in addition to defining Repository interface.

Example of implementation of PagingAndSortingRepository which is super interface of JpaRepository, is explained below.

1. Sample while implementing general purpose CRUD operation in MyBatis is shown.
2. There is a comparision to the case when Repository is implemented using mechanism of Spring Data JPA.

- TodoRepository.java

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

S.No.	Description
(1)	The basic methods required for Repository interface is defined by inheriting `org.springframework.data.repository.PagingAndSortingRepository` (Sub interface of CrudRepository) provided by Spring Data. In case of MyBatis, in addition to defining the interface, RepositoryImpl should also be implemented.

- TodoRepositoryImpl.java

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

    @Inject
    protected 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 = new ArrayList<Todo>();
        if(0 < count){
            todos = queryDAO.executeForObjectList("todo.findAllSort",
                pageable.getSort(),pageable.getOffset(),pageable.getPageSize());
        } else {
            todos = new ArrayList<Todo>();
        }
        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.clas
            return count.longValue();
        }

    }
```

S.No.	Description
(1)	Assign `@Repository` as class annotation. By assigning annotation, it becomes target of component-scan and bean definition in the configuration file is not required.
(2)	Assign `@Transactional` as class annotation. Transaction boundary is controlled by Service, but this annotation should also be assigned to Repository as well.
(3)	Inject `jp.terasoluna.fw.dao.QueryDAO` for executing query processing.
(4)	Inject `jp.terasoluna.fw.dao.UpdateDAO` for executing update processing.
(5)	Assign `@Transactional(readOnly = true)` to query method.
(6)	The method defined in `CrudRepository` is implemented.
(7)	The method defined in `PagingAndSortingRepository` is implemented.
(8)	The method added in `TodoRepository` is implemented.

- 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>
```

S.No.	Description
(1)	Specify namespace. Assign name that can uniquely identify Entity.
(2)	Specify the type of Entity and execute mapping of field with column.
(3)	Implement SQL for each SQLID.

Implementing Repository to link with external system using RestTemplate

Todo

TBD

Plan to provide details in the coming versions.

4.3 Implementation of Application Layer

This chapter explains implementation of application layer of a web application that uses HTML forms.

Note: Refer to the following page for the implementation required for Ajax development and REST API.

- *[coming soon] Ajax*

Implementation of application layer is given in the following 3 steps.

1. *Implementing Controller*

   Controller receives the request, calls business logic, updates model, decides View. Thereby, controls one complete round of operations after receiving the request.

   It is the most important part in the implementation of application layer.

2. *Implementing form object*

   Form object transfers the values between HTML form and application.

3. *Implementing View*

   View (JSP) acquires the data from model (form object, domain object etc.) and generates screen (HTML).

4.3.1 Implementing Controller

Implementation of Controller is explained below.

The Controller performs the following roles.

1. **Provides a method to receive the request.**

   Receives requests by implementing methods to which `@RequestMapping`annotation is assigned.

2. **Performs input validation of request parameter.**

   For request which requires input validation of request parameters, it can be performed by specifying `@Validated` annotation to form object argument of method which receives the request.

   Performs single item check using Bean Validation (JSR-303) and correlation check by Spring Validator or Bean Validation (JSR-303).

3. **Calls business logic.**

   Controller does not implement business logic but delegates by calling Service method.

4. **Reflects the output of business logic to Model.**

   The output can be referred in View by reflecting domain object returned from Service method to `Model`.

5. **Returns View name corresponding to business logic output.**

   Controller does not implement the logic of rendering the view. Rendering is done in View technologies like JSP etc.

   Controller only returns the name of the view in which the rendering logic is implemented.

   The View corresponding to view name is resolved by `ViewResolver`provided by Spring Framework.

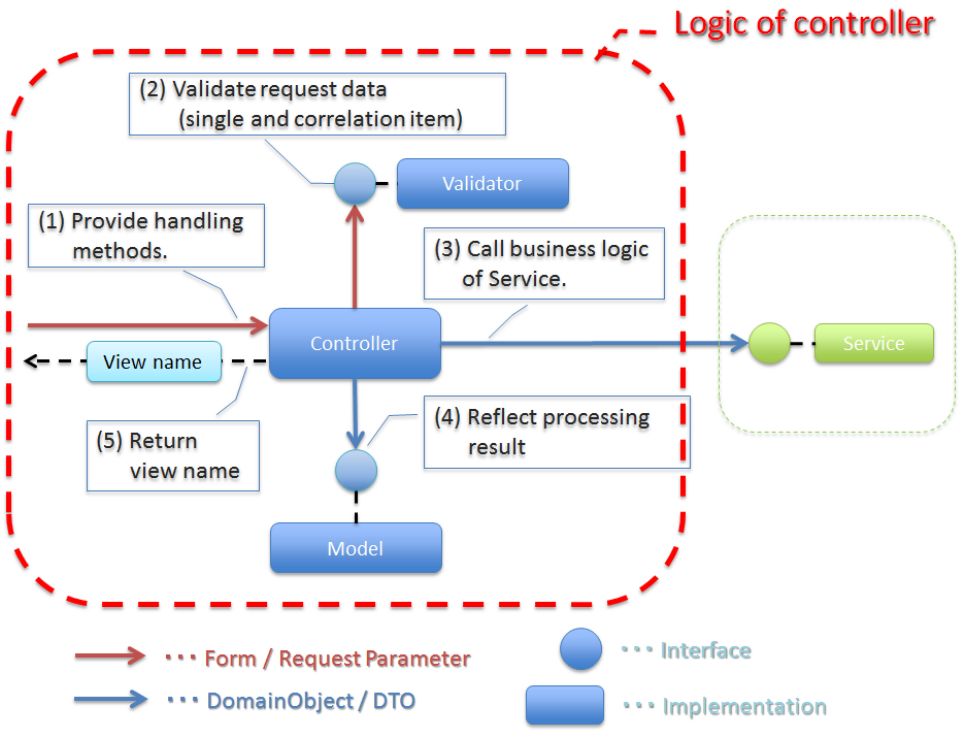


Figure.4.1 **Picture - Logic of controller**

Note: **It is recommended that controller implements only the routing logic** such as calling business logic, reflecting output of the business logic to `Model`, deciding the View name is implemented in the Controller.

The implementation of Controller is explained by focusing on the following points.

- *Creating Controller class*
- *Mapping request and processing method*

- *Regarding arguments of processing method*
- *Regarding return value of processing method*

Creating Controller class

**Controller class is created with @Controller annotation added to POJO class (Annotation-based Controller).
**

Controller in Spring MVC can also be created by implementing `org.springframework.web.servlet.mvc.Controller` interface (Interface-based Controller). However, it is preferred to avoid using it as it is Deprecated from Spring 3 onwards.

```
@Controller
public class SampleController {
    // ...
}
```

Mapping request and processing method

`@RequestMapping`annotation is assigned to the method that receives request.

In this document, the method to which `@RequestMapping`is added, is called as “processing method”.

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

The rules for mapping the incoming request with a processing method can be specifying as attributes of `@RequestMapping`annotation.

Sr.No.	Attribute name	Description
1.	value	Specify request path which needs to be mapped (multiple values allowed).
2.	method	Specify HTTP method (`RequestMethod`type) which needs to be mapped (multiple methods allowed). GET/POST are mainly used for mapping requests from HTML form, while other HTTP methods (such as PUT/DELETE) are used for mapping requests from REST APIs as well.
3.	params	Specify request parameters which needs to be mapped (multiple parameters allowed). Request parameters are mainly used for mapping request from HTML form. If this mapping method is used, the case of mapping multiple buttons on HTML page can be implemented easily.
4.	headers	Specify request headers which needs to be mapped (multiple headers allowed). Mainly used while mapping REST API and Ajax requests.
5.	consumes	Mapping can be performed using Content-Type header of request. Specify media type which needs to be mapped (multiple types allowed). Mainly used while mapping REST API and Ajax requests.
6.	produces	Mapping can be performed using Accept header of request. Specify media type which needs to be mapped (multiple types allowed). Mainly used while mapping REST API and Ajax requests.

Note: Combination of mapping

Complex mapping can be performed by combining multiple attributes, but considering maintainability, mapping should be defined and designed in the simplest way possible . It is recommended to consider combining 2 attributes (value attribute and any other 1 attribute).

6 examples of mapping are shown below.

- *Mapping with request path*
- *Mapping by HTTP method*
- *Mapping by request parameter*
- *Mapping using request header*
- *Mapping using Content-Type header*
- *Mapping using Accept header*

In the following explanation, it is prerequisite to define the processing method in the Controller class.

```
@Controller // (1)
@RequestMapping("sample") // (2)
public class SampleController {
    // ...
}
```

Sr.No.	Description
(1)	With `@Controller`, it is recognized as Annotation-based controller class and becomes the target of component scan.
(2)	All the processing methods in this class are mapped to URLs with "sample" by adding `@RequestMapping("sample")` annotation at class level.

Mapping with request path

In case of the following definition, if the URL `"sample/hello"` is accessed, then `hello()` method is executed.

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

When multiple values are specified, it is handled by 'OR' condition.

In case of following definition, if "sample/hello" or "sample/bonjour" is accessed, then hello() method is executed.

```
    @RequestMapping(value = {"hello", "bonjour"})
    public String hello() {
```

Pattern can be specified instead of a specific value for request path. For details of specifying patterns, refer to reference documention of Spring Framework.

- URI Template Patterns
- URI Template Patterns with Regular Expressions
- Path Patterns
- Patterns with Placeholders

Mapping by HTTP method

In case of the following definition, if the URL "sample/hello" is accessed with POST method, then hello() method is executed. For the list of supported HTTP methods, refer to Javadoc of Spring framework. When not specified, all supported HTTP methods are mapped.

```
    @RequestMapping(value = "hello", method = RequestMethod.POST)
    public String hello() {
```

When multiple values are specified, it is handled by 'OR' condition.

In case of following definition, if "sample/hello" is accessed with GET or HEAD method, then hello() method is executed.

```
    @RequestMapping(value = "hello", method = {RequestMethod.GET, RequestMethod.HEAD})
    public String hello() {
```

Mapping by request parameter

In case of following definition, if the URL `sample/hello?form` is accessed, then `hello()` method is executed.

When request is sent as a POST request, request parameters may exist in request body even if they do not exist in URL.

```
    @RequestMapping(value = "hello", params = "form")
    public String hello() {
```

When multiple values are specified, it is handled by 'AND' condition.

In case of following definition, if the URL `"sample/hello?form&formType=foo"` is accessed, then `hello()` method is executed.

```
    @RequestMapping(value = "hello", params = {"form", "formType=foo"})
    public String hello(@RequestParam("formType") String formType) {
```

Supported formats are as follows.

Sr.No.	Format	Explanation
1.	paramName	Mapping is performed when request parameter of the specified parameName exists.
2.	!paramName	Mapping is performed when request parameter of the specified parameName does not exist.
3.	paramName=paramValue	Mapping is performed when value of the specified parameName is paramValue.
4.	paramName!=paramValue	Mapping is performed when value of the specified parameName is not paramValue.

Mapping using request header

Refer to the details on the following page to mainly use the controller to map REST API and Ajax requests.

- *[coming soon] Ajax*

Mapping using Content-Type header

Refer to the details on the following page to mainly use the controller to map REST API and Ajax requests.

- *[coming soon] Ajax*

Mapping using Accept header

Refer to the details on the following page to mainly use the controller to map REST API and Ajax requests.

- *[coming soon] Ajax*

Mapping request and processing method

Mapping by the following method is recommended.

- **Grouping of URL of request is done for each unit of business flow or functional flow.**

  URL grouping means defining `@RequestMapping(value = "xxx")` as class level annotation.

- **Use the same URL for requests for screen transitions within same functional flow**

  The same URL means the value of 'value' attribute of `@RequestMapping(value = "xxx")` must be same.

  Determining which processing method is used for a particular request with same functional flow is performed using HTTP method and HTTP parameters.

The following is an example of mapping between incoming request and processing method by a sample application with basic screen flow.

- *Overview of sample application*
- *Request URL*
- *Mapping request and processing method*
- *Implementing form display*
- *Implementing the display of user input confirmation screen*
- *Implementing 'redisplay of form'*
- *Implementing 'create new user' business logic*

Overview of sample application

Functional overview of sample application is as follows.

- Provides functionality of performing CRUD operations of Entity.
- Following 5 operations are provided.

Sr.No.	Operation name	Overview
1.	Fetching list of Entities	Fetch list of all the created Entities to be displayed on the list screen.
2.	Create Entity	Create a new Entity with the specified contents. Screen flow (form screen, confirmation screen, completion screen) exists for this process.
3.	Fetching details of Entity	Fetch Entity of specified ID to be displayed on the details screen.
4.	Entity update	Update Entity of specified ID. Screen flow (form screen, confirmation screen, completion screen) exists for this process.
5.	Entity delete	Delete Entity of specified ID.

- Screen flow of all functions is as follows.

  It is not mentioned in screen flow diagram however, when input validation error occurs, form screen is displayed again.

Request URL

Design the URL of the required requests.

- Request URLs of all the requests required by the process flow are grouped.

  This functionality performs CRUD operations of Entity called ABC, therefore URL that starts with `"/abc/"` is considered.

- Design request URL for each operation of the functionality.

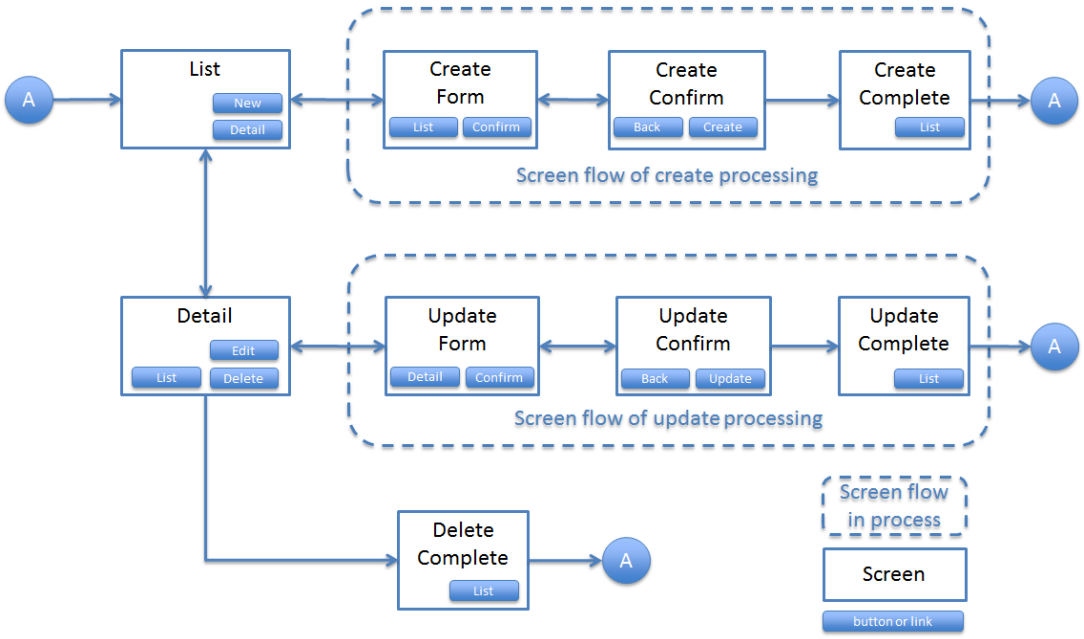


Figure.4.2 **Picture - Screen flow of entity management function**

Sr.No.	Operation name	URL for each operation (pattern)
1.	Fetching list of Entities	/abc/list
2.	Create Entity	/abc/create
3.	Fetching details of Entity	/abc/{id}
4.	Entity update	/abc/{id}/update
5.	Entity delete	/abc/{id}/delete

Note: "`{id}`" specified in URL of 'Fetching details of Entity', 'Entity update', 'Entity delete' operations is called as, URI Template Patternand any value can be specified. In this sample application, Entity ID is specified.

Assigned URL of each operation of screen flow diagram is mapped as shown below:

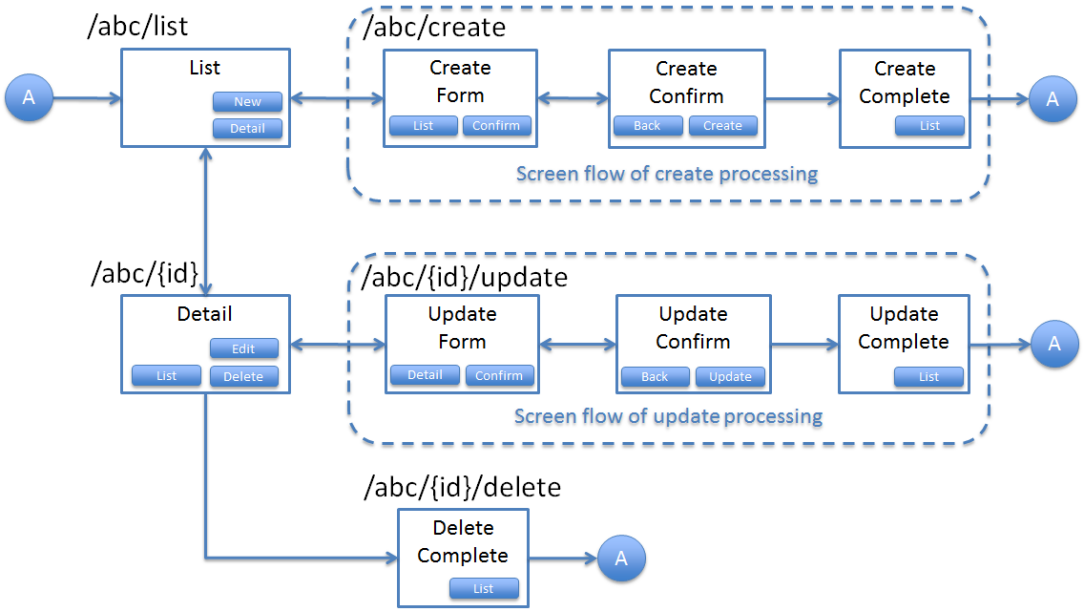


Figure.4.3 **Picture - Screen flow of entity management function and corresponding assigned URL**

Mapping request and processing method

Design the mapping between incoming request and processing method.

The following is the mapping design which is designed according to mapping policy.

Sr.No.	Operation name	URL	Request name	HTTP Method	HTTP Parameter	Processing method
1.	Fetching list of Entities	/abc/list	List display	GET	-	list
2.	Create New Entity	/abc/create	Form display	-	form	createForm
3.			Displaying input confirmation	POST	confirm	createConfirm
4.			Form re-display	POST	redo	createRedo
5.			Entity Creation	POST	-	create
6.			Displaying completion of Entity Creation	GET	complete	createComple
7.	Fetching details of Entity	/abc/{id}	Display details of Entity	GET	-	read
8.	Entity update	/abc/{id}/update	Displaying Form	-	form	updateForm
9.			Displaying confirmation of user input	POST	confirm	updateConfir
10.			Form re-display	POST	redo	updateRedo
11.			Update	POST	-	update
12.			Displaying completion of update process	GET	complete	updateCompl
13.	Entity delete	/abc/{id}/delete	Delete	POST	-	delete
14.			Displaying completion of delete process	GET	complete	deleteComple

Multiple requests exist for each of Create Entity, Entity Update and Entity Delete functions. Therefore switching of processing methods is done using HTTP method and HTTP parameters.

The following is the flow of requests in case of multiple requests in a function like “Create New Entity”.

All URLs are `"/abc/create"` and determining the processing method is done based on combination of HTTP method and HTTP parameters.

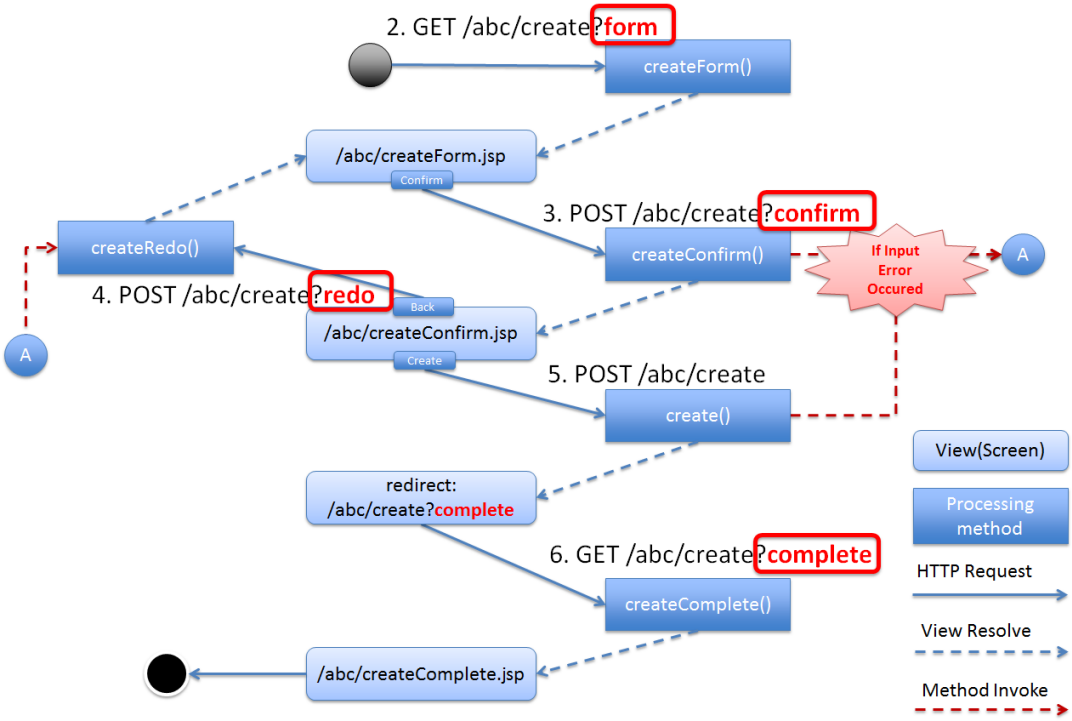


Figure.4.4 **Picture - Request flow of entity create processing**

Implementation of processing method for “Create New Entity” is shown below.

Here, the purpose is to understand mapping between request and processing method and therefore focus must on `@RequestMapping`.

The details of argument and return value (view name and view) of processing method are explained in the next chapter.

- *Implementing form display*
- *Implementing the display of user input confirmation screen*
- *Implementing ‘redisplay of form’*
- *Implementing ‘create new user’ business logic*

- *Implementing notification of create new user process completion*
- *Placing multiple buttons on HTML form*

Implementing form display

In order to display the form, `form` is specified as HTTP parameter.

```
@RequestMapping(value = "create", params = "form") // (1)
public String createForm(AbcForm form, Model model) {
    // omitted
    return "abc/createForm"; // (2)
}
```

Sr.No.	Description
(1)	Specify "`form`" as value of `params` attribute.
(2)	Return view name of JSP to render form screen.

Note: In this processing method, `method` attribute is not specified since it is not required for HTTP GET method.

Example of implementation of sections other than processing method is explained below.

Besides implementing the processing method for form display, points mentioned below are required:

- Implement generation process of form object. Refer to *Implementing form object* for the details of form object.
- Implement View of form screen. Refer to *Implementing View* for the details of View.

Use the following form object.

```
public class AbcForm implements Serializable {
    private static final long serialVersionUID = 1L;

    @NotEmpty
    private String input1;

    @NotNull
    @Min(1)
    @Max(10)
    private Integer input2;

    // omitted setter&getter
}
```

Creating an object of AbcForm.

```
@ModelAttribute
public AbcForm setUpAbcForm() {
    return new AbcForm();
}
```

Create view(JSP) of form screen.

```
<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>
```

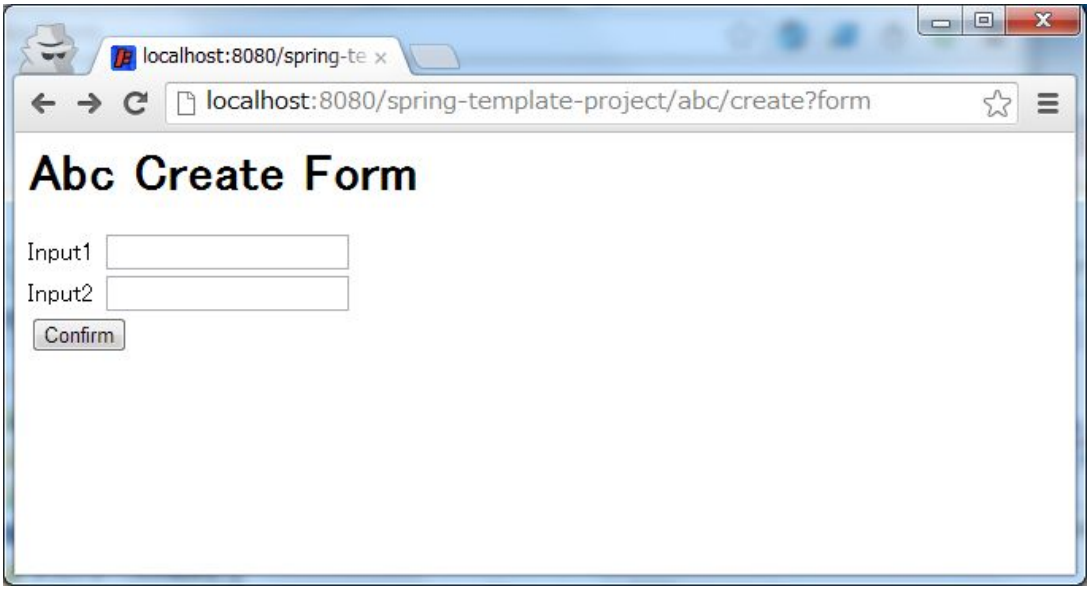
Sr.No.	Description
(1)	Specify `name="confirm"`parameter for submit button to transit to confirmation screen.

The operations are explained below.

Sending the request for form display.

Access "`abc/create?form`" URL.

Since `form` is specified in the URL as an HTTP parameter, `createForm` method of controller is called and form screen is displayed.



Implementing the display of user input confirmation screen

To check user input in the form, data is sent by POST method and `confirm` is specified as HTTP parameter.

```
    @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)
    }
```

Sr.No.	Description
(1)	Specify “RequestMethod.POST” in `method` attribute and “confirm” in `params` attribute.
(2)	In case of input validation errors, it is recommended to call the processing method of form re-display.
(3)	Return view-name of JSP to render the screen for user input confirmation.

Note: POST method is specified to prevent displaying confidential information such as password and other personal information etc. in the address bar. (Needless to say that these security measures not sufficient and needs more secure measures such as SSL etc.)

Example of implementation of sections other than processing method is explained below.

Besides implementing processing method for user input confirmation screen, points mentioned below are required.

- Implement view of user-input confirmation screen. Refer to *Implementing View* for the details of view.

Create the view (JSP) for user input confirmation screen.

```
<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>
```

Sr.No.	Description
(1)	The values entered on form screen is set as the hidden fields of HTML form since they must be sent back to the server when Create or Back buttons are clicked.
(2)	Specify ‘‘name=”redo”‘‘parameter for submit button to return to form screen.
(3)	Parameter name need not be specified for submit button. Submit button will do the actual create operation.

Note: In the above example, HTML escaping is performed as an XSS countermeasure using `f:h()` function while displaying the user input values. For details, refer to `Cross Site Scripting`.

The operations are explained below.

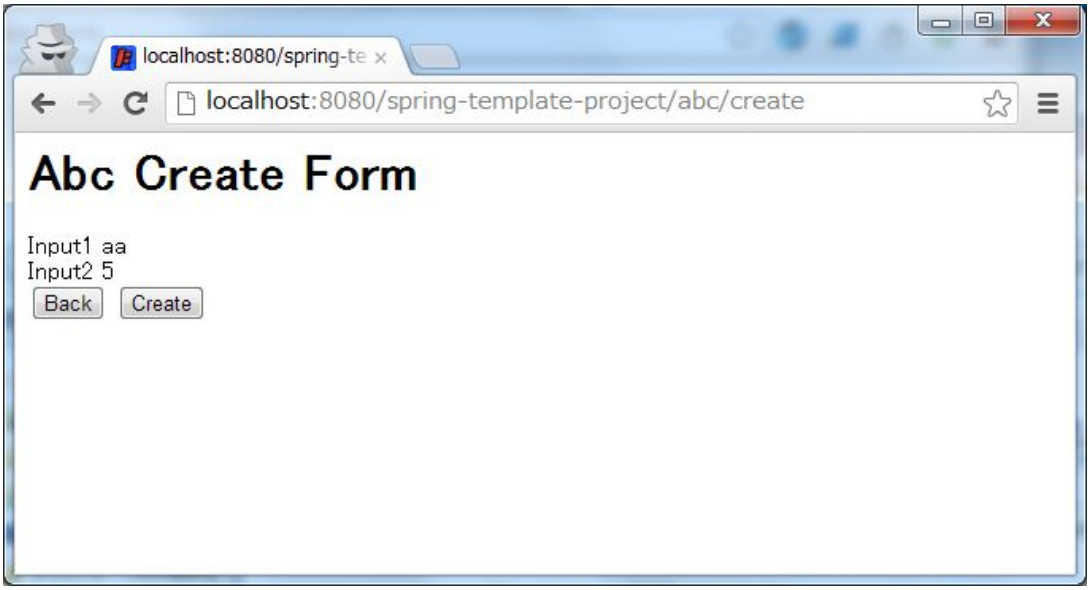
Send the request for displaying user input confirmation.

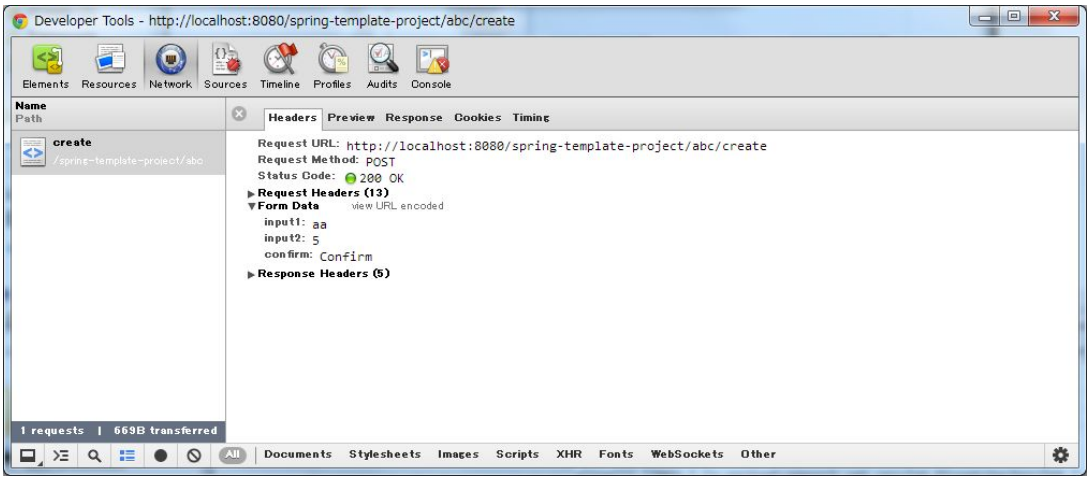
Enter `"aa"` in Input1 and `"5"` in Input2 and click Confirm button on form screen.

After clicking Confirm button, `"abc/create?confirm"` URI gets accessed using POST method.

Since HTTP parameter `confirm` is present in the URI, `createConfirm` method of controller is called and user input confirmation screen is displayed.

Since HTTP parameters are sent across through HTTP POST method after clicking the Confirm button, it does not appear in URI. However, “confirm” is included as HTTP parameter.





Implementing 'redisplay of form'

"redo" is specified as HTTP parameter to indicate that form needs to be redisplayed.

```
@RequestMapping(value = "create", method = RequestMethod.POST, params = "redo") // (1)
public String createRedo(AbcForm form, Model model) {
    // ommited
    return "abc/createForm"; // (2)
}
```

Sr.No.	Description
(1)	Specify “RequestMethod.POST” in `method` attribute and “redo” in `params` attribute.
(2)	Return view name of JSP to render the form screen.

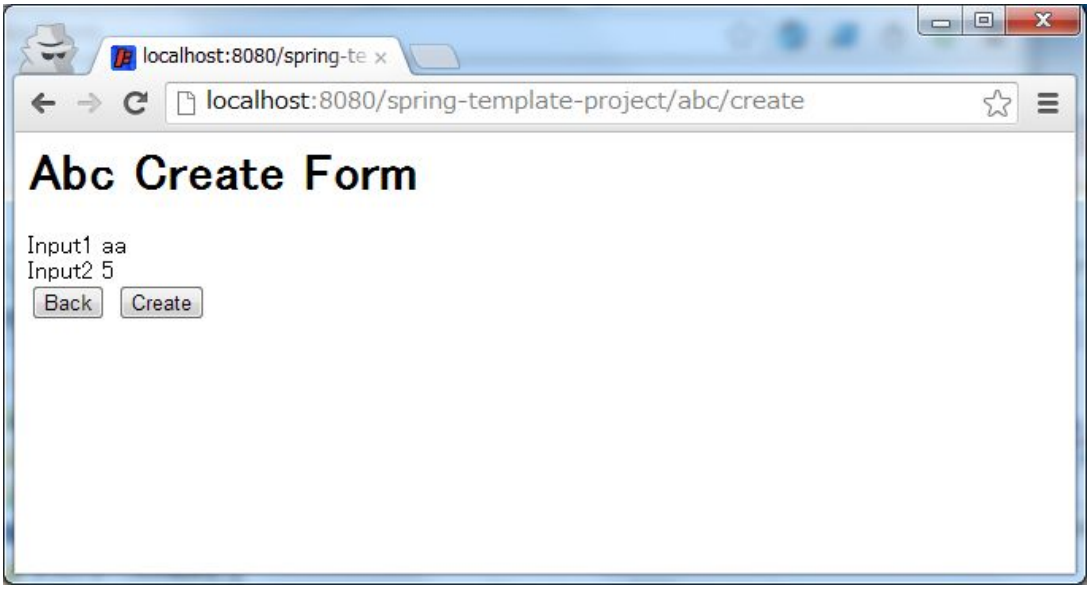
Operation is described below.

Send the request to redisplay the form screen.

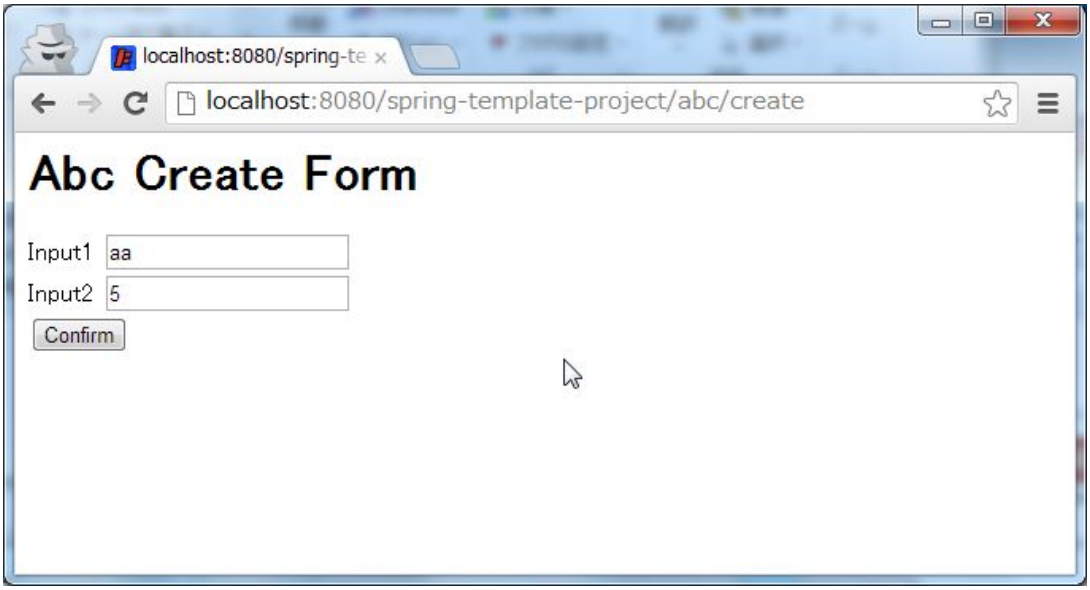
Click Back button on user input confirmation screen.

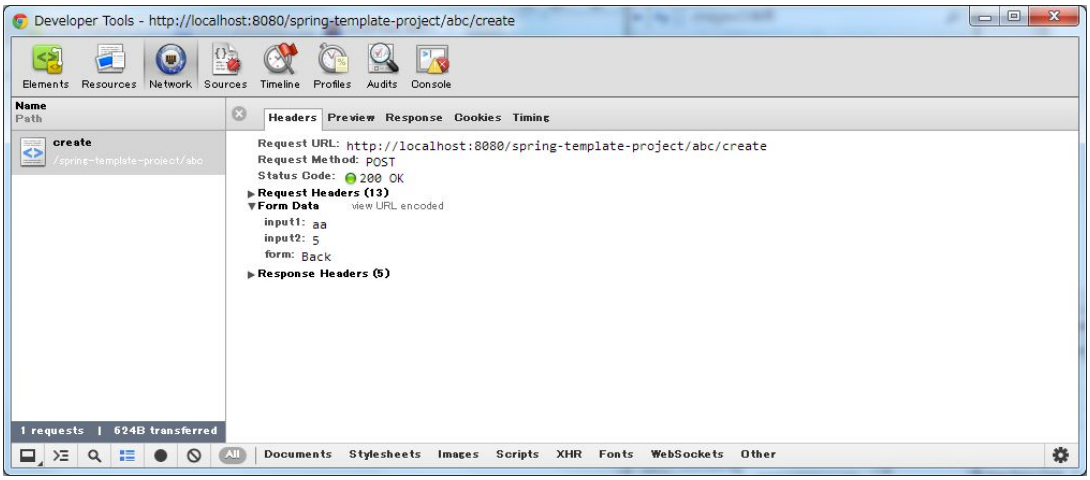
When Back button is clicked, “abc/create?redo” URI gets accessed through HTTP POST method.

Since “redo” HTTP parameter is present in the URI, `createRedo` method of controller is invoked and form screen is redisplayed.



Since HTTP parameters are sent across through HTTP POST method after clicking the Back button, it does not appear in URI. However, “redo” is included as HTTP parameter. Moreover, since input values of form had been sent as hidden fields, input values can be restored on redisplayed form screen.





Note: In order to implement back button functionality, setting `onclick="javascript:history.back()"` is also one of the ways. Both the methods differ in the following ways. Appropriate method must be selected as per requirement.

- Behavior when “Back button on browser” is clicked.
- Behavior when page having Back button is accessed and Back button is clicked.
- History of browser

Implementing 'create new user' business logic

To register input contents of form, the data (hidden parameters) to be registered is sent with HTTP POST method.

Sorting is not carried out using HTTP parameters since new request will be the main request of this operation.

Since the state of database changes in this process, it should not be executed multiple times due to double submission.

Therefore, it is 'redirected' to the next screen (create complete screen) instead of directly displaying View (screen) after

completing this process. This pattern is called as POST-Redirect-GET(PRG) pattern. For the details of PRG (Post-Redirect-Get) pattern

refer to *[coming soon] Double Submit Protection* .

```
@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)
}
```

Sr.No.	Description
(1)	Specify `RequestMethod.POST` in `method` attribute. Do not specify `params` attribute.
(2)	Return URL to the request needs to be redirected as view name in order to use PRG pattern.

Note: It can be redirected to "/xxx" by returning "redirect:/xxx" as view name.

Warning: PRG pattern is used to avoid double submission when the browser gets reloaded by clicking F5 button. However, as a countermeasure for double submission, it is necessary to use TransactionTokenCheck functionality. For details of TransactionTokenCheck, refer to *[coming soon] Double Submit Protection* .

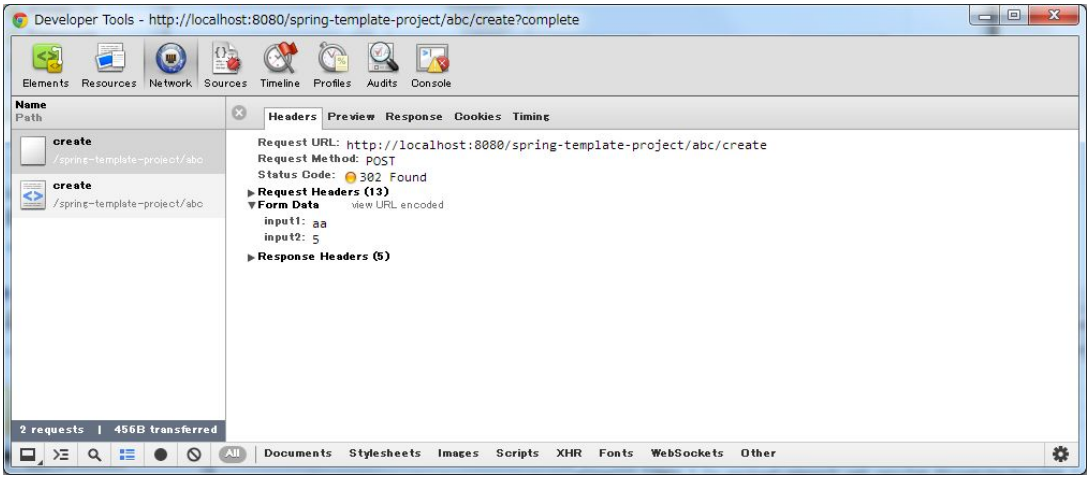
Operation is described below.

Click ‘Create’ button on input confirmation screen.

After clicking ‘Create’ button, "abc/create" URL is accessed through POST method.

Since HTTP parameters are not sent for identifying a button, it is considered as main request of Entity create process and ‘create’ method of Controller is invoked.

‘Create’ request does not return to the screen directly, but it is redirected to create complete display ("/abc/create?complete"). Hence HTTP status is changed to 302.



Implementing notification of create new user process completion

In order to notify the completion of create process, complete must be present in the request as HTTP parameter.

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

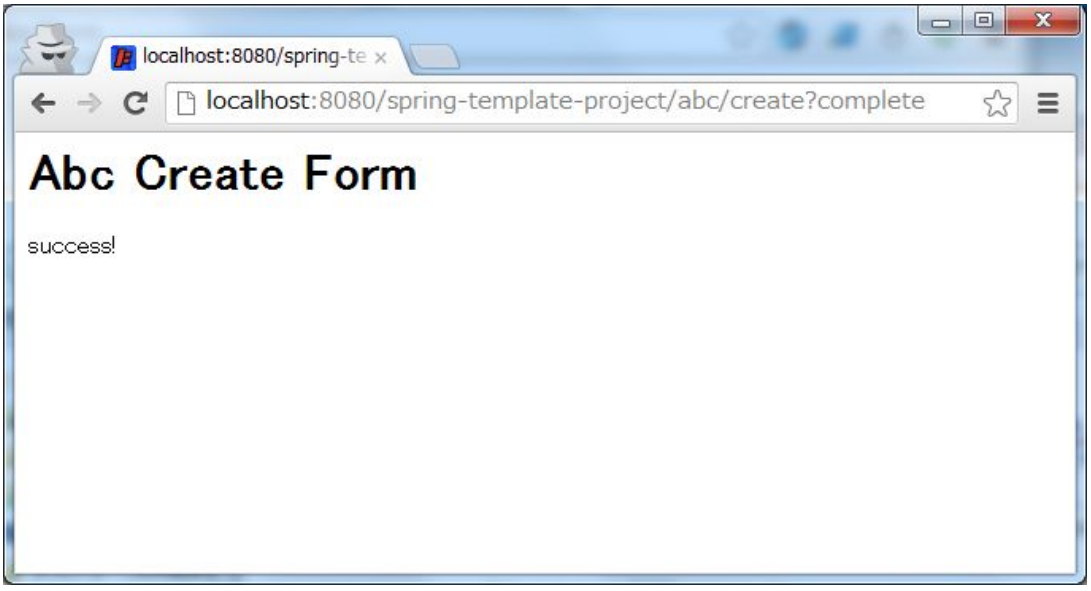
Sr.No.	Description
(1)	Specify `"complete"` in `params` attribute.
(2)	Return View name of JSP to render the create completion screen.

Note: In this processing method, `method` attribute is not specified since it is not required for HTTP GET method.

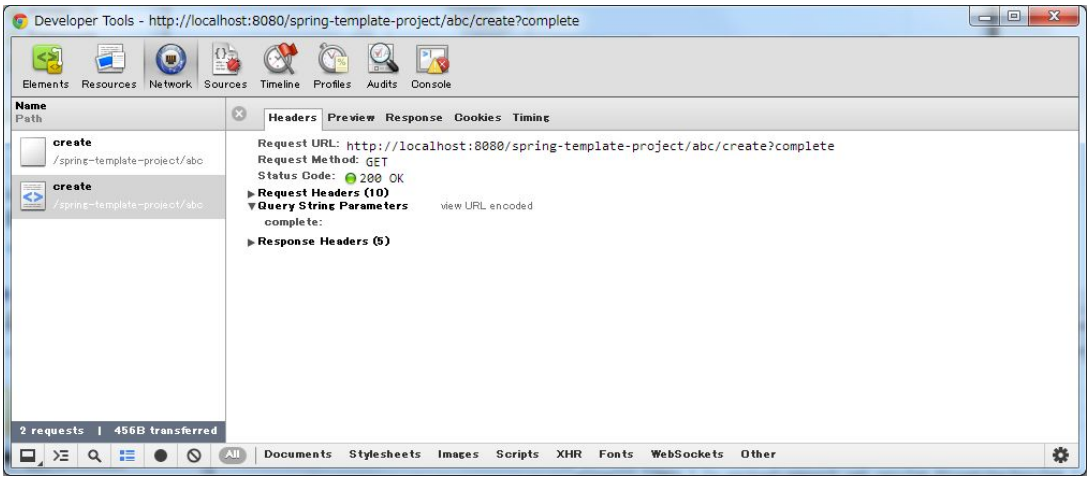
Operation is described below.

After completing creation of user, access URI (`"/abc/create?complete"`) is specified as redirect destination.

Since HTTP parameter is `complete`, createComplete() method of controller is called and create completion screen is displayed.



Note: Since PRG pattern is used, even if browser is reloaded, create completion screen is only re-displayed without re-executing create process.



Placing multiple buttons on HTML form

To place multiple buttons on a single form, send HTTP parameter to identify the corresponding button and and so that the processing method of controller can be switched. An example of Create button and Back button on input confirmation screen of sample application is explained here.

'Create' button to perform 'user creation' and 'Back' button to redisplay 'create form' exists on the form of input confirmation screen as shown below.

To redisplay 'create form' using request (`"/abc/create?redo"`) when Back button is clicked, the following code is required in HTML form.

```
<input type="submit" name="redo" value="Back" /> <!-- (1) -->
<input type="submit" value="Create" />
```

Sr.No.	Description
(1)	In input confirmation screen (`"abc/createConfirm.jsp"`), specify `name="redo"`parameter for Back button.

For the operations when Back button is clicked, refer to *Implementing 'redisplay of form'*.

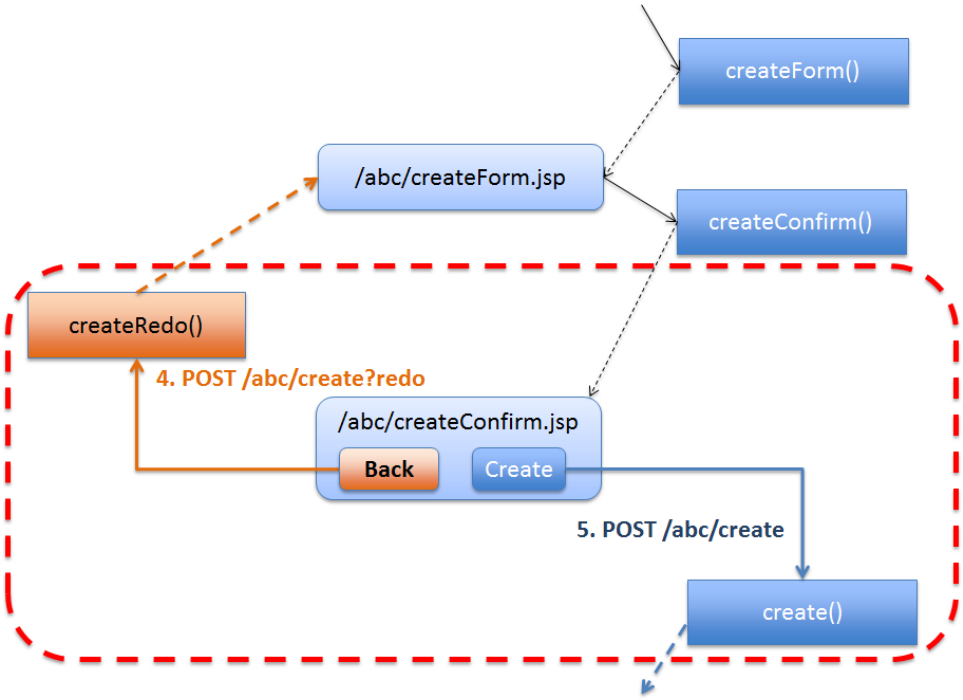


Figure.4.5 **Picture - Multiple button in the HTML form**

Source code of controller of sample application

Source-code of controller after implementing create process of sample application are shown below.

Fetching list of Entities, Fetching detail of Entity, Entity update, Entity delete are implemented using the same guidelines.

```
@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";
        }

    }
```

Regarding arguments of processing method

The arguments of processing method can be used to fetch various values http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-ann-arguments However, as a principle rule, the following must be not be fetched using arguments of processing method of controller.

- 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

Note: When generalized values like HttpServletRequest, getAttribute/setAttribute of HttpSession and get/put of Map are allowed, liberal use of these can degrade the maintainability of the project with an increase in project size.

When common parameters (request parameters) need to be stored in JavaBean and passed as an argument to a method of controller, it can be implemented using *Implementing HandlerMethodArgumentResolver* as described later.

Arguments depending on the purpose of usage are described below.

- *Passing data to screen (View)*
- *Retrieving values from URL path*
- *Retrieving request parameters individually*
- *Retrieving request parameters collectively*
- *Performing input validation*
- *Passing data while redirecting request*
- *Passing request parameters to redirect destination*
- *Inserting values in redirect destination URL path*
- *Acquiring values from Cookie*
- *Writing values in Cookie*

- *Retrieving pagination information*
- *Retrieving uploaded file*
- *Displaying result message on the screen*

Passing data to screen (View)

To pass data to be displayed on screen (View), fetch `org.springframework.ui.Model`(Hereafter called as `Model`) as argument of processing method and add the data (Object) to `Model`object.

- 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) -->
```

Sr.No.	Description
(1)	Fetch `Model`object as argument.
(2)	Call `addAttribute`method of `Model`object received as argument, and add the data to `Model`object. For example, `"HelloWorld!"` string is added to the attribute name `"hello"`.
(3)	If first argument of `addAttribute`method is omitted, the class name beginning with lower case letter will become the attribute name. For example, the result of `model.addAttribute("helloBean", new HelloBean());` is same as the result of `model.addAttribute(new HelloBean());`
(4)	In View (JSP), it is possible to acquire the data added to `Model`object by specifying "${Attribute name}". For example, HTML escaping is performed using "${f:h(Attribute name)}" function of EL expression. For details of functions of EL expression that perform HTML escaping, refer to *Cross Site Scripting*.
(5)	The values of JavaBean stored in `Model`can be acquired by specifying "${Attribute name.property name}".
(6)	JSP is output in HTML format.

Note: Even though the `Model`is not used, it can be specified as an argument. Even if it is not required at the initial stage of implementation, it can be used later (so that the signature of methods need not be changed in future).

Note: The value can also be referred from the module which is not managed under Spring MVC (for example, ServletFilter, etc.) since `addAttribute` in `Model` performs a `setAttribute` in

`HttpServletRequest`.

Retrieving values from URL path

To retrieve values from URL path, add `@PathVariable`annotation to argument of processing method of controller.

In order to retrieve values from the path using `@PathVariable`annotation, value of `@RequestMapping`annotation must contain those values in the form of variables (for example, {id}).

```
@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
}
```

Sr.No.	Description
(1)	Specify the portion to be extracted as path variable in the value of `@RequestMapping`annotation. Specify path variable in “{variable name}” format. For example, 2 path variables such as `"id"` and `"version"` are specified.
(2)	Specify variable name of path variable in `@PathVariable`annotation. For example, when the URL `"sample/hello/aaaa/1"` is accessed, the string `"aaaa"` is passed to argument “id”.
(3)	Value attribute of `@PathVariable`annotation can be omitted. When it is omitted, the argument name is considered as the request parameter name. For example, when the URL `"sample/hello/aaaa/1"` is accessed, value `"1"` is passed to argument “version”. However, if value attribute is omitted, the compilation must be done in debug mode.

Note: Binding argument can be of any data type other than string. In case of different data type, `org.springframework.beans.TypeMismatchException`is thrown and default response is 400 (Bad Request). For example, when the URL `"sample/hello/aaaa/v1"` is accessed, an exception is thrown since `"v1"` cannot be converted into Integer type.

Warning: When omitting the value attribute of `@PathVariable`annotation, the application to be deployed must be compiled in debug mode. Compiling in debug mode has an impact on memory and performance, since information or processing required for debugging is inserted to the class after compiling. Basically, explicitly specifying the value attribute is recommended.

Retrieving request parameters individually

To retrieve request parameters individually, add `@RequestParam`annotation to argument.

```
@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
}
```

Sr.No.	Description
(1)	Specify request parameter name in the value attribute of `@RequestParam`annotation. For example, when the URL `"sample/hello?id=aaaa"` is accessed, the string `"aaaa"` is passed to argument “id”.
(2)	value attribute of `@RequestParam`annotation can be omitted. When it is omitted, the argument name becomes the request parameter name. For example, when the URL `"sample/hello?name=bbbb&...."` is accessed, string `"bbbb"` is passed to argument “name”. However, if value attribute is omitted, the compilation should be done in debug mode.
(3)	By default, an error occurs if the specified request parameter does not exist. When request parameter is not required, specify `false` in the `required` attribute. For example, when it is accessed where request parameter `age` does not exist, `null`is passed to argument “age”.
(4)	When default value is to be used if the specified request parameter does not exist, specify the default value in defaultValue attribute. For example, when it is accessed where request parameter `genderCode` does not exist, `"unknown"` is passed to argument “genderCode”.

Note: When it is accessed without specifying mandatory parameters, `org.springframework.web.bind.MissingServletRequestParameterException`is thrown and default operation is responded with 400 (Bad Request). However, when defaultValue attribute is specified, the value specified in defaultValue attribute is passed without throwing exception.

Note: Binding argument can be of any data type. In case the data type do not match, `org.springframework.beans.TypeMismatchException`is thrown and default response is 400 (Bad Request). For example, when `"sample/hello?age=aaaa&..."` URL is accessed, exception is thrown since `aaaa` cannot be converted into Integer.

Binding to form object must be done only when any of the following conditions are met.

- If request parameter is an item in the HTML form.
- If request parameter is not an item in HTML form, however, input validation other than mandatory check needs to be performed.
- If error details of input validation error needs to be output for each parameter.
- If there are 3 or more request parameters. (maintenance and readability point of view)

Retrieving request parameters collectively

Use form object to collectively fetch all the request parameters.

Form object is JavaBean representing HTML form. For the details of form object, refer to *Implementing form object*.

Following is an example that shows the difference between processing method that fetches each request parameter using `@RequestParam` and the same processing method when fetching request parameters in a form object

Processing method that receives request parameter separately using `@RequestParam` is as shown below.

```
@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
}
```

Create form object class

For jsp of HTML form corresponding to this form object, refer to *Binding to HTML form*.

```
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

}
```

Note: Request parameter name should match with form object property name.

When parameters "id=aaa&name=bbbb&age=19&genderCode=men?tel=01234567" are sent to the above f

the values of `id` , `name` , `age` , `genderCode` matching with the property name, are stored, however `tel` is not included in form object, as it does not have matching property name.

Make changes such that request parameters which were being fetched individually using `@RequestParam` now get fetched as form object.

```
@RequestMapping("bindRequestParams")
public String bindRequestParams(@Validated SampleForm form, // (1)
        BindingResult result,
        Model model) {
    // do something
    return "sample/hello"; // returns view name
}
```

Sr.No.	Description
(1)	Receive `SampleForm`object as argument.

Note: When form object is used as argument, unlike `@RequestParam`, mandatory check is not performed. ** When using form object, ** *Performing input validation* ** should be performed as described below **.

Warning: Domain objects such as Entity, etc. can also be used as form object without any changes required. However, the parameters such as password for confirmation, agreement confirmation checkbox, etc. should exist only on WEB screen. Since the fields depending on such screen items should not be added to domain objects, it is recommended to create class for form object separate from domain object. When a domain object needs to be created from request parameters, values must first be received in form object and then copied to domain object from form object.

Performing input validation

When performing input validation for the form object, add `@Validated`annotation to form object argument, and specify `org.springframework.validation.BindingResult`(Hereafter called as `BindingResult`) to argument immediately after form object argument.

Refer to *Input Validation* for the details of input validation.

Add annotations required in input validation to the fields of form object class.

```
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
}
```

Add `@Validated`annotation to form object argument.

Input validation is performed for the argument with `@Validated` annotation before the processing method of controller is executed. The check result is stored in the argument `BindingResult` which immediately follows form object argument.

The type conversion error that occurs when a data-type other than String is specified in form object, is also stored in `BindingResult`.

```
@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
    }
```

Sr.No.	Description
(1)	Add `@Validated`annotation to `SampleForm`argument, and mark it as target for input validation.
(2)	Specify `BindingResult` in the argument where input validation result is stored.
(3)	Check if input validation error exists. If there is an error, `true` is returned.

Passing data while redirecting request

To redirect after executing a processing method ofcontroller and to pass data along with it, fetch `org.springframework.web.servlet.mvc.support.RedirectAttributes`(Henceforth called as `RedirectAttributes`) as an argument of processing method, and add the data to `RedirectAttributes` object.

- 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) -->
```

Sr.No.	Description
(1)	Fetch `RedirectAttributes`object as argument of the processing method of controller.
(2)	Call `addFlashAttribute`method of `RedirectAttributes`and add the data to `RedirectAttributes`object. For example, the string data `"HelloWorld!"` is added to attribute name `"hello"`.
(3)	If first argument of `addFlashAttribute`method is omitted, the class name beginning with lower case letter becomes the attribute name. For example, the result of `model.addFlashAttribute("helloBean", new HelloBean());` is same as `model.addFlashAttribute(new HelloBean());`.
(4)	Send a redirect request to another URL which will display the next screen instead of displaying screen (View) directly.
(5)	In the processing method after redirection, return view name of the screen that displays the data added in (2) and (3).
(6)	In the View (JSP) side, the data added to `RedirectAttributes`object can be obtained by specifying “${attribute name}”. For example, HTML escaping is performed using “${f:h(attribute name)}” function of EL expression. For the details of functions of EL expression that performs HTML escaping, refer to *Cross Site Scripting*.
(7)	The value stored in `RedirectAttributes`can be obtained from JavaBean by using “${Attribute name.Property name}”.
(8)	HTML output.

Warning: The data cannot be passed to redirect destination even though it is added to `Model`.

Note: It is similar to the `addAttribute`method of `Model`. However survival time of data differs. In `addFlashAttribute`of `RedirectAttributes`, the data is stored in a scope called flash scope. Data of only 1 request (G in PRG pattern) can be referred after redirect. The data from the second request onwards is deleted.

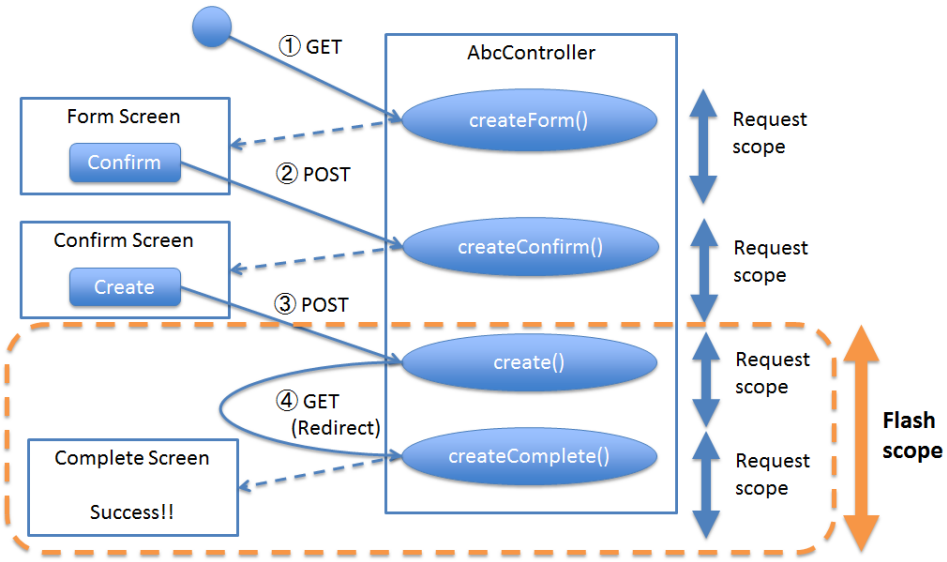


Figure.4.6 **Picture - Survival time of flush scope**

Passing request parameters to redirect destination

When request parameters are to be set dynamically to redirect destination, add the values to be passed to `RedirectAttributes`object of argument.

```
@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";
}
```

Sr.No.	Description
(1)	Specify request parameter name in argument `name and request parameter value in argument ``value` and call `addAttribute`method of `RedirectAttributes`object. In the above example, it is redirected to `"/sample/hello?complete&id=aaaa"`.

Warning: In the above example, the result is the same as of `return "redirect:/sample/hello?complete&id=" + id;`(as shown in the commented out line in the above example). However, since URL encoding is also performed if `addAttribute` method of `RedirectAttributes`object is used, the request parameters that needs to be inserted dynamically **should be set to the request parameter using addAttribute method and should not be set to redirect URL specified as return value.** The request parameters which are not to be inserted dynamically ("complete" as in the above example), can be directly specified in the redirect URL specified as the return value.

Inserting values in redirect destination URL path

To insert values in redirect destination URL path dynamically, add the value to be inserted in `RedirectAttributes`object of argument as shown in the example to set request parameters.

```
@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)
}
```

Sr.No.	Description
(1)	Specify attribute name and the value using `addAttribute`method of `RedirectAttributes`object.
(2)	Specify the path of the variable "{Attribute name}" to be inserted in the redirect URL. In the above example, it is redirected to `"/sample/hello/aaaa?complete"`.

Warning: In the above example, the result is same as of `"redirect:/sample/hello/" + id + "?complete";` (as shown in the commented out line in the above example). However, since URL encoding is also performed when using `addAttribute` method of `RedirectAttributes` object, the path values to be inserted dynamically **should be inserted using addAttribute method and path variable and should not be set to redirect URL specified as return value.**

Acquiring values from Cookie

Add `@CookieValue` annotation to the argument of processing method to acquire the values from a cookie.

```
@RequestMapping("readCookie")
public String readCookie(@CookieValue("JSESSIONID") String sessionId, Model model) { // (1)
    // do something
    return "sample/readCookie"; // returns view name
}
```

Sr.No.	Description
(1)	Specify name of the cookie in the `value` attribute of `@CookieValue` annotation. In the above example, "JSESSIONID" value is passed from cookie to sessionId argument.

Note:

As in the case of `@RequestParam` , it has `required` attribute and `defaultValue` attribute. Also, the data type of the a

Refer to *Retrieving request parameters individually* for details.

Writing values in Cookie

To write values in cookie, call `addCookie` method of `HttpServletResponse` object directly and add the value to cookie.

Since there is no way to write to cookie in Spring MVC (3.2.3 version), ** Only in this case, HttpServletResponse can fetched as an argument of processing method of controller.**

```
@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";
}
```

Sr.No.	Description
(1)	Specify `HttpServletResponse`object as argument to write to cookie.
(2)	Generate `Cookie`object and add to `HttpServletResponse`object. For example, `"hello world!"` value is assigned to Cookie name `"foo"`.

Tip: No difference compared to use of `HttpServletResponse`which fetched as an argument of processing method, however, `org.springframework.web.util.CookieGenerator`class is provided by Spring as a class to write values in cookie. It should be used if required.

Retrieving pagination information

Pagination related information is required for the requests performing list search.

Fetching `org.springframework.data.domain.Pageable`(henceforth called as `Pageable`) object as an argument of processing method enables to handle pagination related information (page count, fetch record count) easily.

Refer to *[coming soon] Pagination* for details.

Retrieving uploaded file

Uploaded file can be obtained in 2 ways.

- Provide `MultipartFile` property in form object.
- Use `org.springframework.web.multipart.MultipartFile` as an argument of processing method having `@RequestParam` annotation.

Refer to *[coming soon] File Upload* for details.

Displaying result message on the screen

`Model` object or `RedirectAttributes`object can be obtained as an argument of processing method and result message of business logic execution can be displayed by adding `ResultMessages` object to Model or RedirectAttributes.

Refer to *Message Management* for details.

Regarding return value of processing method

Various return types supported by the processing method are given in <http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-ann-return-types>'_ however, only the following basic values should be used.

- String (for logical name of view)

Return types depending on the purpose of usage are described below:

- *HTML response*
- *Responding to downloaded data*

HTML response

To get HTML response to display the output of processing method, it has to return view name of JSP.

ViewResolver, at the time of generating HTML using JSP, must be extended class of `UrlBasedViewResolver` (`InternalViewResolver` and `TilesViewResolver`).

An example using `InternalViewResolver`is mentioned below, however, it is recommended to use `TilesViewResolver` when the screen layout is in templated format.

Refer to *[coming soon] Screen Layout using Tiles* for the usage of `TilesViewResolver`.

- 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)
}
```

S.No.	Description
(1)	Specify base directory (prefix of file path) where JSP files are stored. By specifying prefix of file path, there is no need to specify physical storage location of JSP files, at the time of returning view name in the processing method of controller.
(2)	Specify extension (suffix of file path) of JSP file. By specifying suffix of file path, there is no need to specify extention of JSP files, at the time of returning view name in the processing method of controller.
(3)	Specify execution order when multiple ViewResolvers are specified. It can be specified in the range of Integers and executed in increasing order.
(4)	When View name `"sample/hello"` is the return value of processing method, `"/WEB-INF/views/sample/hello.jsp"` is displayed.

Note: HTML output is generated using JSP in the above example, however, even if HTML is generated using other template engine such as Velocity, FreeMarker, return value of processing method will be `"sample/hello`. `ViewResolver` takes care of task to determine which template engine is to be used.

Responding to downloaded data

In order to return the data stored in db as download data (`"application/octet-stream"`), it is recommended to create a view

for generating response data (download process).The processing method adds the data to be downloaded to `Model` and returns

name of the view which performs the actual download process.

The solution to create a separate ViewResolver to resolve a view using its view name, however, `BeanNameViewResolver`provided by Spring Framework is recommended.

Refer to *[coming soon] File Download* for the details of download processing.

- 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());
    }
}
```

S.No.	Description
(1)	`BeanNameViewResolver`is the class that resolves the view by searching for the bean which matches with the returned view name, from application context.
(2)	When `BeanNameViewResolver`is used along with `InternalViewResolver`or `TilesViewResolver`, it is recommended to give it a higher priority compared to these other ViewResolvers. For example, If `"0"` is specified as the priority for `BeanNameViewResolver`, `BeanNameViewResolver`is used prior to `InternalViewResolver`to resolve a view.
(3)	When `"sample/report"` is returned by the process menthod, the data generated by the view instance registered as bean in step (4) is returned as download data.
(4)	Specify view name as name of the component and register view object as a bean. For example, `x.y.z.app.views.XxxExcelView` instance is registered as a bean with bean name (view name) as `"sample/report"` .
(5)	Example of view implementation. View class that extends `org.springframework.web.servlet.view.document.AbstractExcelView` and generates Excel data.

Implementing the process

The point here is that **do not implement business logic in controller** .

Business logic must be implemented in Service. Controller must call the service methods in which the business logic is implemented.

Refer to *Domain Layer Implementation* for the details of implementation of business logic.

Note: Controller should be used only for routing purposes (mapping requests to corresponding business logic) and deciding the screen transition for each request as well as setting model data. Thereby, controller should be simple as much as possible. By consistently following this policy, the contents of controller become clear which ensures maintainability of controller even if the size of development is large.

Operations to be performed in controller are shown below:

- *Correlation check of input value*
- *Calling business logic*
- *Reflecting values to domain object*
- *Reflecting values to form object*

Correlation check of input value

Correlation check of input values should be done using `Validation` class which implements `org.springframework.validation.Validator`interface.

Bean Validation can also be used for correlation check of input values.

Refer to *Input Validation* for the details of implementation of correlation check.

The implementation of correlation check itself should not be written in the processing method of controller. However, it is necessary to add the `Validator`to `org.springframework.web.bind.WebDataBinder`.

```
    @Inject
    protected PasswordEqualsValidator passwordEqualsValidator; // (1)

    @InitBinder
    protected void initBinder(WebDataBinder binder){
        binder.addValidators(passwordEqualsValidator); // (2)
    }
```

Sr.No.	Description
(1)	Inject `Validator`that performs correlation check.
(2)	Add the injected `Validator` to `WebDataBinder`. Adding the above to `WebDataBinder` enables correlation check by executing `Validator`before the processing method gets called.

Calling business logic

Execute business logic by injecting the Service in which business logic is implemented and calling the injected Service method.

```
@Inject
protected SampleService sampleService; // (1)

@RequestMapping("hello")
public void hello(Model model){
    String message = sampleService.hello(); // (2)
    model.addAttribute("message", message);
    return "sample/hello";
}
```

Sr.No.	Description
(1)	Inject the `Service` in which business logic is implemented.
(2)	Call the injected `Service` method to execute business logic.

Reflecting values to domain object

In this guideline, it is recommended to bind the data sent by HTML form to form object instead of the domain object.

Therefore, the controller should perform the process of reflecting the values of form object to domain object which is then passed to the method of service class.

```
@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";
}
```

Sr.No.	Description
(1)	Create domain object and reflect the values bound to form object in the domain object.
(2)	Call the method of service class to execute business logic.
(3)	Add the data returned from business logic to `Model`.

The process of reflecting values to domain object should be implemented by the processing methodof controller. However considering the readability of processing

method in case of large amount of code, it is recommended to delegate the process to Helper class.

Example of delegating the process to Helper class is shown below:

- SampleController.java

```
@Inject
protected 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
    protected 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;
    }
}
```

S.No.	Description
(1)	Inject object of Helper class in controller.
(2)	Value is reflected to the domain object by calling the method of the injected Helper class. Delegating the process to Helper class enables to keep the implementation of controller simple.
(3)	Call the Service class method to execute the business logic after creating domain object.

Note: Bean conversion functionality can be used as an alternative way to delegate the process of reflecting form object values, to Helper class. Refer to *[coming soon] Bean Mapping (Dozer)* for the details of Bean conversion functionality.

Reflecting values to form object

In this guideline, it is recommended that form object (and not domain object) must be used to for that data which is to be binded to HTML form.

For this, it is necessary to reflect the values of domain object (returned by service layer) to form object. This conversion should be performed in controller class.

```
@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";
}
```

Sr.No.	Description
(1)	Call the method of service class in which business logic is implemented and fetch domain object.
(2)	Reflect values of acquired domain object to form object.
(3)	When there are fields only for display, add domain object to `Model`so that data can be referred.

Note: In JSP, it is recommended to refer the values from domain object instead of form object for the fields to be only displayed on the screen.

The process of reflecting value to form object should be implemented by the processing method of controller. However considering the readability of processing method in case of large amount of code, it is recommended to delegate the process to Helper class method.

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

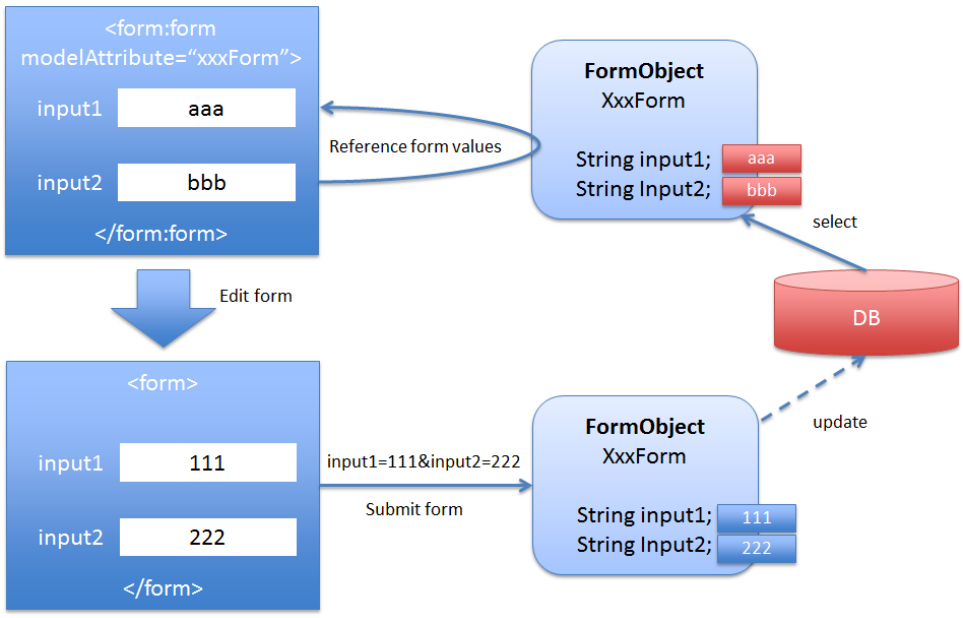
Sr.No.	Description
(1)	Call the method to reflect the values of domain object to form object.
(2)	Reflect the values of domain object to form object.

Note: Bean conversion functionality can be used as an alternative method to delegate the process to Helper class. Refer to *[coming soon] Bean Mapping (Dozer)* for the details of Bean conversion functionality.

4.3.2 Implementing form object

Form object is the object (JavaBean) which represents HTML form and plays the following role.

1. **Holds business data stored in the database so that it can be referred by HTML form (JSP).**
2. **Holds request parameters sent by HTML form so that they can be referred by processing method of controller.**



Implementation of form object can be described by focusing on the following points.

- *Creating form object*
- *Initializing form object*
- *Binding to HTML form*
- *Binding request parameters*

Creating form object

Create form object as a JavaBean. Spring Framework provides the functionality to convert and bind the request parameters (string) sent by HTML form to the format defined in form object. Hence, the fields to be defined in form object need not only be in `java.lang.String`format.

```
public class SampleForm implements Serializable {
    private String id;
    private String name;
    private Integer age;
    private String genderCode;
    private Date birthDate;
```

```
    // omitted getter/setter
}
```

Tip: **Regarding the mechanism provided by Spring Framework that performs format conversion **

Spring Framework executes format conversion using the following 3 mechanisms and supports conversion to basic format as standard. Refer to linked page for the details of each conversion function.

- Spring 3 Type Conversion
- Spring 3 Field Formatting
- java.beans.PropertyEditor implementations

Warning: In form object, it is recommended to maintain only the fields of HTML form and not the fields which are just displayed on the screen. If display only fields are also maintained in form object, more memory will get consumed at the time of storing form objectin HTTP session object causing memory exhaustion. In order to display the values of display only fields on the screen, it is recommended to add objects of domain layer (such as Entity) to request scope by using (`Model.addAttribute`).

Number format conversion of fields

Number format can be specified for each field using `@NumberFormat`annotation.

```
public class SampleForm implements Serializable {
    @NumberFormat(pattern = "#,#") // (1)
    private Integer price;
    // omitted getter/setter
}
```

Sr.No.	Description
(1)	Specify the number format of request parameter sent by HTML form. For example, binding of value formatted by ”,” is possible since “”#, #”” format is specified as pattern. When value of request parameter is “”1,050””, Integer object of “”1050”” will bind to the property `price` of form object.

Attributes of `@NumberFormat` annotation are given below.

Sr.No.	Attribute name	Description
1.	style	Specify number format style (NUMBER, CURRENCY, PERCENT). Refer to 'Javadoc <http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/format/annotation/NumberFormat.Style.html> of Spring Framework'_ for details.
2.	pattern	Specify number format of Java. Refer to 'Javadoc <http://docs.oracle.com/javase/7/docs/api/java/text/DecimalFormat.html> of JAVASE'_for details.

Date and time format conversion of fields

Date and time format for each field can be specified using `@DateTimeFormat`annotation.

```
public class SampleForm implements Serializable {
    @DateTimeFormat(pattern = "yyyyMMdd") // (1)
    private Date birthDate;
    // ommitted getter/setter
}
```

Sr.No.	Description
(1)	Specify the date and time format of request parameter sent by HTML form. For example, `"yyyyMMdd"` format is specified as pattern. When the value of request parameter is `"20131001"`, Date object of 1st October, 2013 will bind to property `birthDate` of form object.

Attributes of `@DateTimeFormat`annotation are given below.

Sr.No.	Attribute name	Description
1.	iso	Specify ISO date and time format. Refer to 'Javadoc <http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/format/annotation/DateTimeFormat.ISO.html> of Spring Framework'_for details.
2.	pattern	Specify Java date and time format. Refer to 'Javadoc <http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html> of JAVASE'_for details.
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

Todo

Description of style is incomplete.

DataType conversion in controller

`@InitBinder`annotation can be used to define datatype conversions at controller level.

```
    @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) {
        // ...
    }
```

Sr.No.	Description
(1)	If a method with `@InitBinder`annotation is provided, it is called before executing the binding process and thereby default operations can be customized.
(2)	For example, `"#.  #"` format is specified for a field of type Long. This enables binding of value formatted with ",".
(3)	Default operation for each form object can be customized by specifying it in the value attribute of `@InitBinder`annotation. In the above example, the method is called before binding form object `"sampleForm"`.

Specifying annotation for input validation

Since form object is validated using Bean Validation(JSR-303), it is necessary to specify annotation which indicates constraints of the field. Refer to *Input Validation* for the details of input validation.

Initializing form object

Form object can also be called as form-backing bean and binding can be performed using `@ModelAttribute`annotation. Initialize form-backing bean by the method having

`@ModelAttribute`annotation. In this guideline, such methods are called as ModelAttribute methods and defined with method names like `setUpXxxForm`.

```
@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;
}
```

Sr.No.	Description
(1)	Class name beginning with lower case letter will become the attribute name to add to `Model`. In the above example, `"sampleForm"` is the attribute name. The returned object is added to `Model`and an appropriate process `model.addAttribute(form)` is executed.
(2)	When attribute name is to be specified to add to `Model`, specify it in the value attribute of `@ModelAttribute`annotation. In the above example, `"xxx"` is the attribute name. For returned object, appropriate process “model.addAttribute(“xxx”, form)”is executed and it is returned to `Model`. When attribute name other than default value is specified, it is necessary to specify `@ModelAttribute("xxx")` at the time of specifying form object as an argument of processing method.
(3)	ModelAttribute method can pass the parameters required for initialization as with the case of processing method. In the above example, value of cookie is specified using `@CookieValue`annotation.

Note: When form object is to be initialized with default values, it should be done using ModelAttribute method. In point (3) in above example , value is fetched from cookie, However, fixed value defined in constant class can be set directly.

Note: Multiple ModelAttribute methods can be defined in the controller. Each method is executed before calling processing method of controller.

Warning: If ModelAttribute method is executed for each request, initialization needs to be repeated for each request and unnecessary objects will get created. So, for form objects which are required only for specific requests, should be created inside processing method of controller and not through the use of ModelAttribute method.

Binding to HTML form

It is possible to bind form object added to the `Model` to HTML form(JSP) using `<form:xxx>` tag.

Refer to Using Spring's form tag library for the details of `<form:xxx>` tag.

```
<%@ 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>
```

Sr.No.	Description
(1)	Define taglib to use `<form:form>` tag.
(2)	Specify form object stored in `Model` in the `modelAttribute` attribute of `<form:form>` tag.
(3)	Specify property name of form object in path attribute of `<form:input>` tag.

Binding request parameters

It is possible to bind the request parameters sent by HTML form to form object and pass it as an argument to the processing method of controller.

```
@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) {
    // ...
}
```

Sr.No.	Description
(1)	Form object is passed as an argument to the processing method of controller after reflecting request parameters to the form object.
(2)	When the attribute name is specified in ModelAttribute method, it is necessary to explicitly specify attribute name of form object as `@ModelAttribute("xxx")`.

Warning: When attribute name specified by ModelAttribute method and attribute name specified in the `@ModelAttribute("xxx")` in the argument of processing method are different, it should be noted that a new instance is created other than the instance created by ModelAttribute method. When attribute name is not specified with `@ModelAttribute` in the argument to processing method, the attribute name is deduced as the class name with first letter in lower case.

Determining binding result

Error (including input validation error) that occurs while binding request parameter sent by HTML form to form object, is stored in `org.springframework.validation.BindingResult`.

```
@RequestMapping("hello")
public String hello(
        @Validated SampleForm form,
        BindingResult result, // (1)
```

```
            Model model) {
        if (result.hasErrors()) { // (2)
            return "sample/input";
        }
        // ...
    }
```

Sr.No.	Description
(1)	When `BindingResult`is declared immediately after form object, it is possible to refer to the error inside the processing method of controller.
(2)	Calling `BindingResult.hasErrors()` can determine whether any error occurred in the input values of form object.

It is also possible to determine field errors, global errors (correlated check errors at class level) separately. These can be used separately if required.

Sr.No.	Method	Description
1.	`hasGlobalErrors()`	Method to determine the existence of global errors.
2.	`hasFieldErrors()`	Method to determine the existence of field errors.
3.	`hasFieldErrors(String field)`	Method to determine the existence of errors related to specified field.

4.3.3 Implementing View

View plays the following role.

1. **View generates response (HTML) as per the requirements of the client.**

   View retrieves the required data from model (form object or domain object) and generates response in the format which is required by the client for rendering.

Implementing JSP

Implement View using JSP to generate response(HTML) as per the requirement of the client.

Use the class provided by Spring Framework as the `ViewResolver` for calling JSP. Refer to *HTML response* for settings of `ViewResolver`.

Basic implementation method of JSP is described below.

- *Creating common JSP for include*
- *Displaying value stored in model*
- *Displaying numbers stored in model*
- *Displaying date and time stored in model*
- *Binding form object to HTML form*
- *Displaying input validation errors*
- *Displaying message of processing result*
- *Displaying codelist*
- *Displaying fixed text content*
- *Switching display according to conditions*
- *Repeated display of collection elements*
- *Displaying link for pagination*
- *Switching display according to authority*

In this chapter, usage of main JSP tag libraries are described. However, refer to respective documents for the detailed usage since all JSP tag libraries are not described here.

Sr.No.	JSP tag library name	Document
1.	Spring's form tag library	• http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/view.html#view-jsp-formtaglib • http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/spring-form.tld.html
2.	Spring's tag library	• http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/spring.tld.html
3.	JSTL	• http://download.oracle.com/otndocs/jcp/jstl-1.2-mrel2-eval-oth-JSpec/

Warning: If terasoluna-gfw-web 1.0.0.RELEASE is being used, `action` tag must be always be specified while using `<form:form>` tag of Spring's form tag library.
terasoluna-gfw-web 1.0.0.RELEASE has a dependency on Spring MVC(3.2.4.RELEASE). In this version of Spring MVC, if `action` attribute of `<form:form>` tag is not specified, it will expose a vulnerability of XSS(Cross-site scripting). For further details regarding the vulnerability, refer to CVE-2014-1904 of National Vulnerability Database (NVD).
Also, terasoluna-gfw-web 1.0.1.RELEASE have been upgraded to Spring MVC(3.2.10.RELEASE and above); hence this vulnerability is not present.

Creating common JSP for include

Create a JSP that contains directive declaration which are required by all the JSP files of the project. By specifying this JSP in `<jsp-config>/<jsp-property-group>/<include-prelude>` element of `web.xml`, eliminates the need to declare these directives and each and every JSP file of thep project. Further, this file is provided in blank project also.

- 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>
```

Sr.No.	Description
(1)	JSP tag libraries of JSTL are declared. In this example, `core` and `fmt` are used.
(2)	JSP tag libraries of Spring Framework are declared. In this example, `spring`, `form` and `sec` are used.
(3)	JSP tag libraries provided by common library are declared.
(4)	JSP which is specified at the top of `*.jsp` (extension is jsp) has been included (`/WEB-INF/views/common/include.jsp`).

Note: Refer to `JSP.1.10 Directives` of JavaServer Pages Specification(Version2.2)for details of directives.

Note: Refer to `JSP.3.3 JSP Property Groups` of JavaServer Pages Specification(Version2.2)for details of <jsp-property-group> element.

Displaying value stored in model

To display the value stored in `Model` (form object or domain object) in HTML, use EL expressions or JSP tag libraries provided by JSTL.

Display using EL expressions.

- 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) --%>
```

Sr.No.	Description
(1)	Add `HelloBean`object to `Model`object.
(2)	In View(JSP), data added to the `Model`object can be retrieved by describing `${Attribute name.Property name of JavaBean}`. In this example, HTML escaping is performed using `${f:h(Attribute name.Property name of JavaBean)}` function of EL expression.

Note: Since HTML escaping function (`f:h`) is provided in the common components, always use it if EL expressions are used to output values in HTML. For details of function of EL expression that perform HTML escaping, refer to *Cross Site Scripting*.

Display using `<c:out>` tag provided by JSP tag library of JSTL.

```
Message : <c:out value="${helloBean.message}" /> <%-- (1) --%>
```

Sr.No.	Description
(1)	Specify the values fetched using EL expressions in `value` attribute of `<c:out>` tag. HTML escaping is also performed.

Note: Refer to `CHAPTER 4 General-Purpose Actions` of JavaServer Pages Standard Tag Library(Version 1.2)for the details of `<c:out>`.

Displaying numbers stored in model

Use JSP tag library provided by JSTL to output format number.

Display using <fmt:formatNumber> tag provided by JSP tag library of JSTL.

```
    Number Item : <fmt:formatNumber value="${helloBean.numberItem}" pattern="0.00" /> <%-- (1) --
```

Sr.No.	Description
(1)	Specify the value acquired by EL expressions in the value attribute of <fmt:formatNumber> tag. Specify the format to be displayed in pattern attribute. For example, "0.00" is specified . When the value acquired in ${helloBean.numberItem} is "1.2" temporarily, "1.20" is displayed on the screen.

Note: Refer to CHAPTER 9 Formatting Actions of JavaServer Pages Standard Tag Library(Version 1.2)for the details of <fmt:formatNumber>.

Displaying date and time stored in model

Use JSP tag library provided by JSTL to output format date and time value.

Display using <fmt:formatDate> tag provided by JSP tag library of JSTL.

```
    Date Item : <fmt:formatDate value="${helloBean.dateItem}" pattern="yyyy-MM-dd" /> <%-- (1) --
```

Sr.No.	Description
(1)	Specify the value fetched using EL expression in `value` attribute of `<fmt:formatDate>` tag. Specify the format to be displayed in `pattern` attribute. In this example, "`yyyy-MM-dd`" is specified. When value recieved for `${helloBean.dateItem}` is 2013-3-2, "`2013-03-02`" is displayed on the screen.

Note: Refer to `CHAPTER 9 Formatting Actions` of JavaServer Pages Standard Tag Library(Version 1.2) for details of `<fmt:formatDate>`.

Note: JSP tag library provided by Joda Time should be used to use `org.joda.time.DateTime` as date and time object type. Refer to *Date Operations (Joda Time)* for the details of Joda Time.

Binding form object to HTML form

Use JSP tag library provided by Spring Framework to bind form object to HTML form and to display the values stored in form object.

Bind using `<form:form>` tag provided by Spring Framework.

```
<form:form action="${pageContext.request.contextPath}/sample/hello"
           modelAttribute="sampleForm"> <%-- (1) --%>
    Id : <form:input path="id" /> <%-- (2) --%>
</form:form>
```

Sr.No.	Description
(1)	Specify attribute name of form object stored in `Model`in `modelAttribute` atttribute of `<form:form>`tag.
(2)	Specify name of property to bind in the `path` attribute of `<form:xxx>` tag. `xxx` part changes along with each input element.

Note: Refer to Using Spring's form tag libraryfor the details of `<form:form>`, `<form:xxx>`.

Displaying input validation errors

To display the contents of input validation error, use JSP tag library provided by Spring Framework.

Display using `<form:errors>` tag provided by Spring Framework.

Refer to *Input Validation* for details.

```
<form:form action="${pageContext.request.contextPath}/sample/hello"
           modelAttribute="sampleForm">
    Id : <form:input path="id" /><form:errors path="id" /><%-- (1) --%>
</form:form>
```

Sr.No.	Description
(1)	Specify name of the property to display the error in `path` attribute of `<form:errors>`tag.

Displaying message of processing result

To display the message notifying the output of processing the request, use JSP tag library provided in common components.

Use `<t:messagesPanel>` tag provided in common components.

Refer to *Message Management* for details.

```
<div class="messages">
    <h2>Message pattern</h2>
    <t:messagesPanel /> <%-- (1) --%>
</div>
```

Sr.No.	Description
(1)	Messages stored with attribute name “`resultMessages`” are output.

Displaying codelist

To display the codelist (provided in common components), use JSP tag library provided by Spring Framework.

Codelist can be referred from JSP in the same way as `java.util.Map` interface.

Refer to *[coming soon] CodeList* for details.

Display codelist in select box.

```
<form:select path="orderStatus">
    <form:option value="" label="--Select--" />
    <form:options items="${CL_ORDERSTATUS}" /> <%-- (1) --%>
</form:select>
```

Sr.No.	Description
(1)	Codelist (`java.util.Map` interface) is stored with name (`"CL_ORDERSTATUS"`) as attribute name. Therefore, in JSP, codelist (`java.util.Map` interface) can be accessed using EL expression. Codelist can be displayed in select box by passing the object of `Map` interface to `items` attribute of `<form:options>`.

Label part is displayed on the screen for the value selected in select box.

```
Order Status : ${f:h(CL_ORDERSTATUS[orderForm.orderStatus])}
```

Sr.No.	Description
(1)	In the same way as in case of creating select box, Codelist (`java.util.Map` interface) is stored with name (`"CL_ORDERSTATUS"`) as attribute name. If value selected in select box is specified as key of the fetched `Map` interface, its possible to display the code name.

Displaying fixed text content

Strings for screen name, element name and guidance etc can be directly written in JSP when internationalization is not required.

However, when internationalization is required, display the values acquired from property file using JSP tag library provided by Spring Framework.

Display using <spring:message> tag provided by Spring Framework.

Refer to *[coming soon] Internationalization* for details.

- properties

```
# (1)
label.orderStatus=Order status
```

- jsp

```
<spring:message code="label.orderStatus" text="Order Status" /> : <%-- (2) --%>
    ${f:h(CL_ORDERSTATUS[orderForm.orderStatus])}
```

Sr.No.	Description
(1)	Define the string of label in properties file.
(2)	If key in the properties file is specified in code attribute of <spring:message>, string corresponding to the key is displayed.

Note: The value specified in text attribute is displayed when property value could not be acquired.

Switching display according to conditions

When display is to be switched according to some value in model, use JSP tag library provided by JSTL.

Switch display using <c:if> tag or <c:choose> provided by JSP tag library of JSTL.

Switching display by using <c:if>.

```
<c:if test="${orderForm.orderStatus != 'complete'}"> <%-- (1) --%>
        <%-- ... --%>
</c:if>
```

Sr.No.	Description
(1)	Put the condition for entering the branch in test attribute of <c:if>. In this example, when order status is not 'complete', the contents of the branch will be displayed.

Switching display using <c:choose>.

```
<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>
```

Sr.No.	Description
(1)	Put the condition for entering the branch in `test` attribute of `<c:when>`. In this example, when customer type is `'premium'`, the contents of the branch will be displayed. When condition specified in `test` attribute is `false`, next `<c:when>` tag is processed.
(2)	When result of `test` attribute of all `<c:when>` tags is `false`, `<c:otherwise>` tag is evaluated.

Note: Refer to `CHAPTER 5 Conditional Actions` of JavaServer Pages Standard Tag Library(Version 1.2)for details.

Repeated display of collection elements

To repeat display of collection stored in model, use JSP tag library provided by JSTL.

Repeated display can be done using `<c:forEach>` provided by JSP tag library of JSTL.

```
<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>
```

Sr.No.	Description
(1)	By specifying the collection object in `items` attribute of `<c:forEach>` tag, `<c:forEach>` tag is repeatedly executed to iterate over the collection. While iterating over the collection, if the current element is to be referred inside `<c:forEach>` tag, it can be done by specifying a variable name for the current element in `var` attribute.
(2)	By specifying a variable name in `varStatus` attribute, current position (count) of iteration in `<c:forEach>` tag can be fetched. Refer to JavaDocof `javax.servlet.jsp.jstl.core.LoopTagStatus` for attributes other than count.
(3)	This is the value acquired from the object stored in variable specified by `var` attribute of `<c:forEach>` tag.

Note: Refer to `CHAPTER 6 Iterator Actions` of JavaServer Pages Standard Tag Library(Version 1.2)for details.

Displaying link for pagination

To display links of pagination on the screen while displaying the list, use JSP tag library provided in common components.

Display the link for pagination using `<t:pagination>` provided in common components. Refer to *[coming soon] Pagination* for details.

Switching display according to authority

To switch display according to authority of the user who has logged in, use JSP tag library provided by Spring Security.

Switch display using `<sec:authorize>` provided by Spring Security. Refer to *[coming soon] Authorization* for details.

Implementing JavaScript

When it is necessary to control screen items (controls of hide/display, activate/deactivate, etc.) after screen rendering, control items using JavaScript.

Todo

TBD

Details will be included in the coming versions.

Implementing style sheet

It is recommended to specify attribute values related to screen design in style sheet (css file) and not in JSP(HTML) directly.

In JSP(HTML), specify `id` attribute to identify the items uniquely and `class` attribute which indicates classification of elements.

While, specify values related to actual location of elements or appearance should be specified in style sheet (css file).

By configuring in this way, it is possible to reduce the design related aspects from the implementation of JSP.

Simultaneously, if there is any change in design, only style sheet (css file) is modified and not JSP.

Note: When form is created using `<form:xxx>`, `id` attribute will be set automatically. Application developer should specify `class` attribute.

4.3.4 Implementing common logic

Implementing common logic to be executed before and after calling controller

Here, common logic indicates processes which are required to be executed before and after execution the controller.

Implementing Servlet Filter

Common processes independent of Spring MVC are implemented using Servlet Filter.

However, when common processes are to be executed only for the certain requests mapped with processing method of controller,

then it should be implemented using Handler Interceptor and not Servlet Filter.

Samples of Servlet Filter are given below.

In sample code, value is stored in `MDC` in order to output the log of IP address of client.

- 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>
```

Sr.No.	Description
(1)	In sample, it is guaranteed that it is executed only once for similar requests by creating the Servlet Filter as subclass of `org.springframework.web.filter.OncePerRequestFilter` provided by Spring Framework. Register the created Servlet Filter in `web.xml`.
(2)	Specify URL pattern to apply the registered Servlet Filter.
(3)	

Servlet Filter can also be defined as Bean of Spring Framework.

- 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) -->
```

Sr.No.	Description
(1)	In sample, process is delegated to Servlet Filter defined in step (2) by specifying `org.springframework.web.filter.DelegatingFilterProxy` provided by Spring Framework in Servlet Filter class.
(2)	Add the Servlet Filter class to Bean definition file (`applicationContext.xml`). At this time, `id` attribute of bean definition should be assigned with the filter name (value specified in `<filter-name>` tag) specified in `web.xml`.

Implementing HandlerInterceptor

Common processes dependent on Spring MVC are implemented using HandlerInterceptor.
HandlerInterceptor can execute common processes only for the requests which are allowed by the application since it is called after the determining the processing method
to which the request is to be mapped to.

HandlerInterceptor can execute the process keeping in mind the following 3 points.

- Before executing processing method of controller
  Implemented as `HandlerInterceptor#preHandle` method.
- After successfully executing processing method of controller
  Implemented as `HandlerInterceptor#postHandle` method.
- After completion of processing method of controller (executed irrespective of Normal / Abnormal)
  Implemented as `HandlerInterceptor#afterCompletion` method.

Sample of HandlerInterceptor is given below.
In sample code, log of info level is output after successfully executing controller process.

```
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>
```

Sr.No.	Description
(1)	In sample, HandlerInterceptor is created as the subclass of `org.springframework.web.servlet.handler.HandlerInterceptorAdapter` provided by Spring Framework. Since `HandlerInterceptorAdapter` provides blank implementation of `HandlerInterceptor` interface, it is not required to implement unnecessary methods in the subclass.
(2)	Specify the pattern of path, where the created HandlerInterceptor is to be applied.
(3)	Specify the pattern of path, where the created HandlerInterceptor need not be applied.
(4)	Add the created HandlerInterceptor to `<mvc:interceptors>` tag of `spring-mvc.xml`.

Implementing common processes of controller

Here, common process indicates the process that should be commonly implemented in all the controllers.

Implementing HandlerMethodArgumentResolver

When an object that is not supported by default in Spring Framework is to be passed as controller argument, HandlerMethodArgumentResolver is to be implemented in order to enable controller to be able to receive the argument.

Sample of HandlerMethodArgumentResolver is given below.

In sample code, common request parameters are converted to JavaBean which are then passed to processing method of controller.

- 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>
```

Sr.No.	Description
(1)	JavaBean that retains common parameters.
(2)	Implement `org.springframework.web.method.support.HandlerMethodArgumentResol` interface.
(3)	Determine parameter type. For example, when type of JavaBean that retains common parameters is specified as argument of controller, resolveArgument method of this class is called.
(4)	Fetch the values of request parameters, set the parameters and return the JavaBean that retains the value of common parameters.
(5)	Specify JavaBean that retains common parameters in the argument of processing method of controller. Object returned in step (4) is passed.
(6)	Add the created HandlerMethodArgumentResolver to `<mvc:argument-resolvers>` tag of `spring-mvc.xml`.

Note: When parameters are to be passed commonly to processing methods of all controllers, it is effective to convert the parameters to JavaBean using HandlerMethodArgumentResolver. Parameters referred here are not restricted to request parameters.

Implementing "@ControllerAdvice"

Implement the processes which are to be executed in all controllers in `@ControllerAdvice`, .

In `@ControllerAdvice`, the following processes are common.

- Common `@InitBinder` method

- Common `@ExceptionHandler` method
- Common `@ModelAttribute` method

Sample of `@InitBinder`method is given below.

In sample code, date that can be specified in request parameter is set to `"yyyy/MM/dd"` .

```
@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));
    }

}
```

Sr.No.	Description
(1)	It indicates that it is Bean of ControllerAdvice by assigning the `@ControllerAdvice`annotation.
(2)	It should be component-scan target by assigning the `@Component`annotation.
(3)	Specify priority for common processes by assigning the `@Order`annotation. It should be specified when multiple ControllerAdvice are to be created.
(4)	Implement `@InitBinder`method. `@InitBinder`method is applied to all Controllers.

Sample of `@ExceptionHandler`method is given below.

In sample code, View of lock error screen is returned by handling `org.springframework.dao.PessimisticLockingFailureException`.

```
// (1)
@ExceptionHandler(PessimisticLockingFailureException.class)
public String handlePessimisticLockingFailureException(
        PessimisticLockingFailureException e) {
    return "error/lockError";
}
```

Sr.No.	Description
(1)	Implement `@ExceptionHandler`method. `@ExceptionHandler`method is applied to all Controllers.

Sample of `@ModelAttribute`method is given below.

In sample code, common request parameters are converted to JavaBean and stored in `Model`.

- 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";
}
```

Sr.No.	Description
(1)	Implement `@ModelAttribute`method. `@ModelAttribute`method is applied to all Controllers.
(2)	Object created by `@ModelAttribute` method is passed.

4.3.5 Prevention of double submission

Measures should be taken to prevent double submission as the same process gets executed multiple times by clicking Send button multiple times or refreshing (refresh using F5 button) the Finish screen.

For the problems occurring when countermeasures are not taken and details of implementation method, refer to *[coming soon] Double Submit Protection* .

4.3.6 Usage of session

In the default operations of Spring MVC, model (form object, domain object etc.) is not stored in session.

When it is to be stored in session, it is necessary to assign `@SessionAttributes`annotation to the controller class.

When input forms are split on multiple screens, usage of `@SessionAttributes` annotation should be studied since model (form object, domain object etc.)

can be shared between multiple requests for executing a series of screen transitions.

However, whether to use `@SessionAttributes` annotation should be determined after confirming the warning signs of using the session.

For details of session usage policy and implementation at the time of session usage, refer to *[coming soon] Session Management* .

For the definition of layers, refer to *Application Layering* .

5

Architecture in Detail - TERASOLUNA Global Framework

The architecture adopted in this guideline is explained in detail here.

5.1 [coming soon] Database Access (Common)

Coming soon ...

5.2 [coming soon] Database Access (JPA)

Coming soon ...

5.3 [coming soon] Database Access (MyBatis2)

Coming soon ...

5.4 [coming soon] Exclusive Control

Coming soon ...

5.5 Input Validation

5.5.1 Overview

It is mandatory to check whether the value entered by the user is correct. Validation of input value is broadly classified into

1. Validation to determine whether the input value is valid by just looking at it irrespective of the context such as size and format.
2. Validation of whether the changes in input value are valid depending on the system status.

1. is an example of mandatory check and number of digits check and 2. is an example of check of whether EMail is registered and whether order count is within the count of the available stock.

In this section, 1. is explained and this check is called "Input validation". 2. is called "Business logic check". For business logic check, refer to *Domain Layer Implementation*.

In this guideline, validation check should be performed in application layer whereas business logic check should be performed in domain layer.

Input validation of Web application is performed at server side and client side (JavaScript). It is mandatory to check at the Server Side. However, if the same check is performed at the client side also, usability improves since validation results can be analyzed without communicating with the server.

> **Warning:** Input validation should be performed at the server side as the process at client side may be altered using JavaScript. If the validation is performed only at the client side without performing at the server side, the system may be exposed to danger.

Todo

Input validation at client side will be explained later. Only input validation at the server side is mentioned in the first version.

Classification of input validation

Input validation is classified into single item check and correlation item check.

Type	Description	Example	Implementation method
Single item check	Check completed in single field	Input mandatory check Digit check Type check	Bean Validation (Hibernate Validator is used as implementation library)
Correlation item check	Check comparing multiple fields	Password and confirm password check	Bean Validation or Validation class implementing org.springframework.validation.Validator interface

Spring supports JSR-303 Bean Validation. This Bean Validation is used for single item check. Bean Validation or `org.springframework.validation.Validator` interface provided by Spring is used for correlation item check.

5.5.2 How to use

Single item check

For the implementation of single item check,

- Bean Validation annotation should be assigned to the field of form class
- `@Validated` annotation should be assigned in Controller for validation
- Tag for displaying validation error message should be added to JSP

Note: `<mvc:annotation-driven>` settings are carried out in spring-mvc.xml, Bean Validation is enabled.

Basic single item check

Implementation method is explained using "New user registration" process as an example. Rules for checking "New user registration" form are provided below.

Field name	Type	Rules
name	java.lang.String	Mandatory input 1 or more characters 20 or less characters
email	java.lang.String	Mandatory input 1 or more characters 50 or less characters Email format
age	java.lang.Integer	Mandatory input 1 or more 200 or less

- Form class

  Assign Bean Validation annotation to each field of form class.

```
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
}
```

S.No.	Description
(1)	Assign `javax.validation.constraints.NotNull` indicating that the target field is not `null`. In Spring MVC, when form is sent with input fields left blank, **empty string instead of null binds**to form object by default. This `@NotNull` checks that `name` exists as request parameter.
(2)	Assign `javax.validation.constraints.Size` indicating that the string length (or collection size) of the target field is within the specified size range. Since empty string binds to the field where string is left blank by default in Spring MVC, '1 or more character' rule indicates Mandatory input.
(3)	Assign `org.hibernate.validator.constraints.Email` indicating that the target field is in RFC2822-compliant E-mail format. When E-mail format requirements are flexible than RFC2822-compliant constraints, regular expression should be specified using `javax.validation.constraints.Pattern` instead of `@Email`.
(4)	When form is sent without entering any number in input field, `null` binds to form object so `@NotNull` indicates mandatory input of `age`.
(5)	Assign `javax.validation.constraints.Min` indicating that the target field value must be greater than the specified value.
(6)	Assign `javax.validation.constraints.Max` indicating that the target field value must be less than the specified value.

Tip: Refer to *JSR-303 check rules* and *Hibernate Validator check rules* for standard annotations of Bean Validation and annotations provided by Hibernate.

Tip: Refer to *Binding null to blank string field* for the method of binding `null` when input field is left blank.

- Controller class

  Assign `@Validated` to form class for input validation.

```
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";
    }
}
```

S.No.	Description
(1)	Display "New user registration" form screen.
(2)	Assign `org.springframework.validation.annotation.Validated` to the form class argument to perform input validation.
(3)	Add `org.springframework.validation.BindingResult` that stores check result of input validation performed in step (2). This `BindingResult` must be specified immediately after the form argument. When not specified immediately, `org.springframework.validation.BindException` is thrown.
(4)	Use `BindingResult.hasErrors()` method to determine the check result of step (2). When the result of `hasErrors()` is `true`, return to form display screen as there is an error in input value.
(5)	**Input validation should be executed again**even at the time of submition on the confirmation screen. There is a possibility of data tempering and hence Input validation must be performed just before entering business logic.

Note: `@Validated` is not a standard Bean Validation annotation. It is an independent annotation provided by Spring. Bean Validation standard `javax.validation.Valid` annotation can also be used. However, `@Validated` is better as compared to `@Valid` annotation. Validation group can be specified in case of `@Validated` and hence `@Validated` is recommeded in this guideline.

- JSP

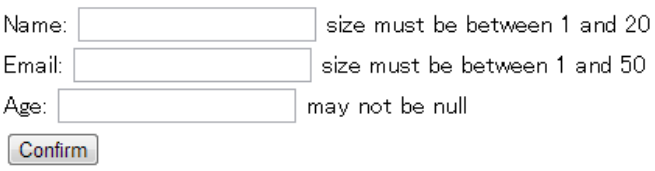
When there is input error, it can be displayed in `<form:errors>` tag.

```
<!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>
```

S.No.	Description
(1)	Specify target field name in `path` attribute of `<form:errors>` tag. Error message is displayed next to input field of each field.

Form is displayed as follows.

Error message is displayed as follows if this form is sent with all the input fields left blank.



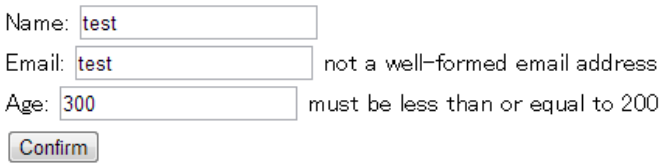
Error messages state that Name and Email are blank and Age is `null`.

Note: In Bean Validation, `null` is a valid input value except the following annotations.

- `javax.validation.constraints.NotNull`
- `org.hibernate.validator.constraints.NotEmpty`
- `org.hibernate.validator.constraints.NotBlank`

In the above example, error messages related to `@Min` and `@Max` annotations are not displayed. This is because `null` is a valid value for `@Min` and `@Max` annotations.

Next, send the form by entering any value in the field.



Error message is not displayed since input value of Name fulfills validation conditions.

Error message is displayed since input value of Email is not in Email format though it fulfills the conditions related to string length.

Error message is displayed since input value of Age exceeds maximum value.

Change the form as follows to change the style at the time of error.

```
<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>
```

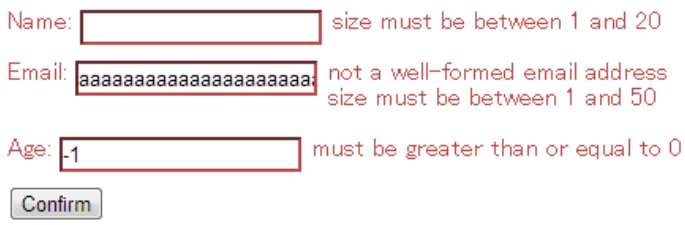
S.No.	Description
(1)	Specify class name for `<label>` tag in `cssErrorClass` attribute at the time of error.
(2)	Specify class name for `<input>` tag in `cssErrorClass` attribute at the time of error.
(3)	Specify class name for error messages in `cssClass` attribute.

For example, if the following CSS is applied to this JSP, error screen is displayed as follows.

```
.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;
}
```

CSS can be customized as per the requirements of screen.

Instead of displaying the error messages next to each input field, output them collectively.



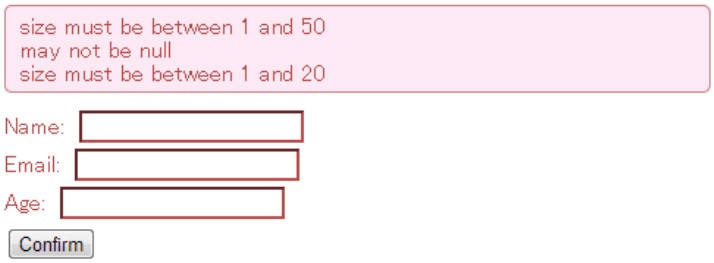

```
<form:form modelAttribute="userForm" method="post"
    action="${pageContext.request.contextPath}/user/create">
    <form:errors path="*" element="div" cssClass="error-message-list" /><%-- (1) --%>

    <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>
```

S.No.	Description
(1)	By specifying `*` in `path` attribute of `<form:errors>` in `<form:form>` tag, all error messages related to Model specified in `modelAttribute` attribute of `<form:form>` can be output. Tag name including these error messages can be specified in `element` attribute. By default, it is `span`. However, specify `div` to output error message list as block element. Specify CSS class in `cssClass` attribute.

An example of error message is shown when the following CSS class is applied.

```
.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;
}
```



By default, field name is not included in error message, so it is difficult to understand, which error message corresponds to which field.

Therefore, when error message is to be displayed in a list, it is necessary to define the message such that field name is included in the error message.

Note: Error messages are output in random order by default. Order can be controlled using @GroupSequence annotationHowever, cost is high.

If error messages are displayed in a list,

- Field name must be included in the error message.
- Order of the errorm message must be controlled.

The above points increase the cost. Hence, "Display error messages next to input field" is recommended if there are no restrictions to do so in screen requirements.

Note: Use `<spring:nestedPath>` tag to display messages collectively outside the `<form:form>` tag.

```
<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>
```

Single item check of nested Bean

The method to validate nested Bean using Bean Validation is explained below.

"Ordering" process of an EC site is considered as an example. Rules for checking "Order" form are provided below.

Field name	Type	Rules	Description
coupon	java.lang.String	5 or less characters Single byte alphanumeric characters	Coupon code
receiverAddress.name	java.lang.String	Mandatory input 1 or more characters 50 or less characters	Receiver name
receiverAddress.postcode	java.lang.String	Mandatory input 1 or more characters 10 or less characters	Receiver postal code
receiverAddress.address	java.lang.String	Mandatory input 1 or more characters 100 or less characters	Receiver address
senderAddress.name	java.lang.String	Mandatory input 1 or more characters 50 or less characters	Sender name
senderAddress.postcode	java.lang.String	Mandatory input 1 or more characters 10 or less characters	Sender postal code
senderAddress.address	java.lang.String	Mandatory input 1 or more characters 100 or less characters	Sender address

Use the same form class since `receiverAddress` and `senderAddress` are objects of the same class.

- Form class

```
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
}
```

S.No.	Description
(1)	This indicates that the child form is mandatory. When not set, it will be considered as valid even if `null` is set in `receiverAddress`.
(2)	Assign `javax.validation.Valid` annotation to enable Bean Validation of the nested Bean.

- Controller class

  It is not different from the Controller described earlier.

```
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>
```

S.No.	Description
(1)	When `receiverAddress.name`, `receiverAddress.postcode`, `receiverAddress.address` are not sent as request parameters due to invalid operation, `receiverAddress` is considered as `null` and error message is displayed.
(2)	Fields of nested bean are specified as `[parent field name].[child field name]`.

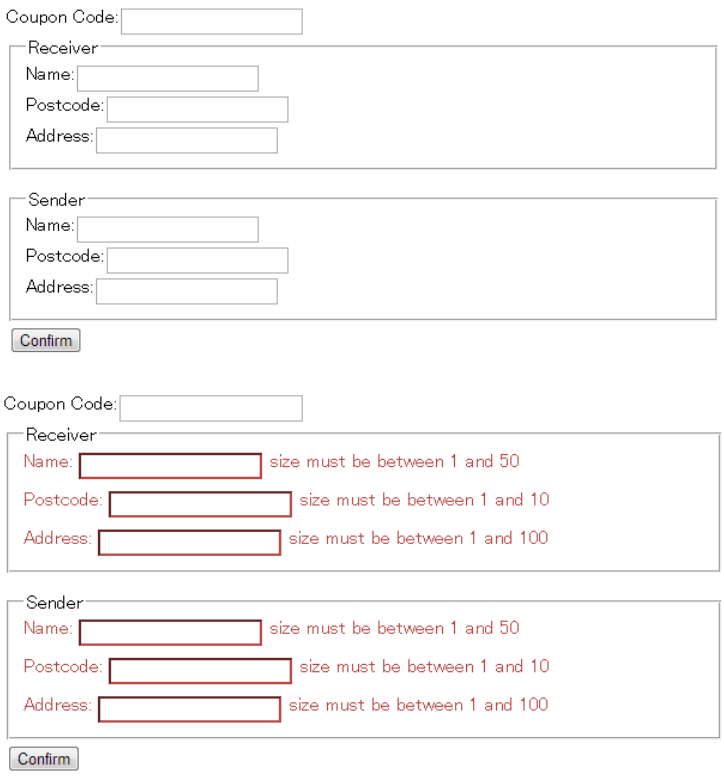
Form is displayed as follows.

Error message is displayed as follows if this form is sent with all the input fields left blank.

Validation of nested bean is enabled for collections also.

Add a field such that upto 3 addresses can be registered in “user registration” form explained at the beginning.

- Add list of `AddressForm` as a field in the form class.




```
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
    }
```

S.No.	Description
(1)	It is possible to use `@Size` annotation for checking size of collection as well.

- 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>
```

S.No.	Description
(1)	Display error message related to `address` field.
(2)	Process the collection of child forms in a loop using `<c:forEach>` tag.
(3)	Inside the loop, Specify the field of child form using `[parent field name][Index].[child field name]`.

- Controller class

```
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";
    }
}
```

S.No.	Description
(1)	Edit the form object to display a single address form at the time of initial display of “user registration” form.

- JavaScript

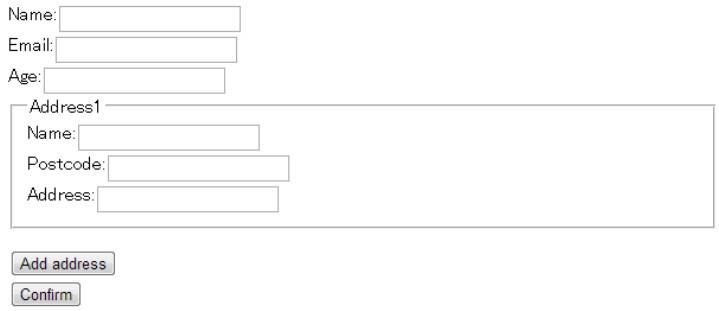
Below is the JavaScript to dynamically add address input field. However, the explanation of this code is

omitted as it is not required.

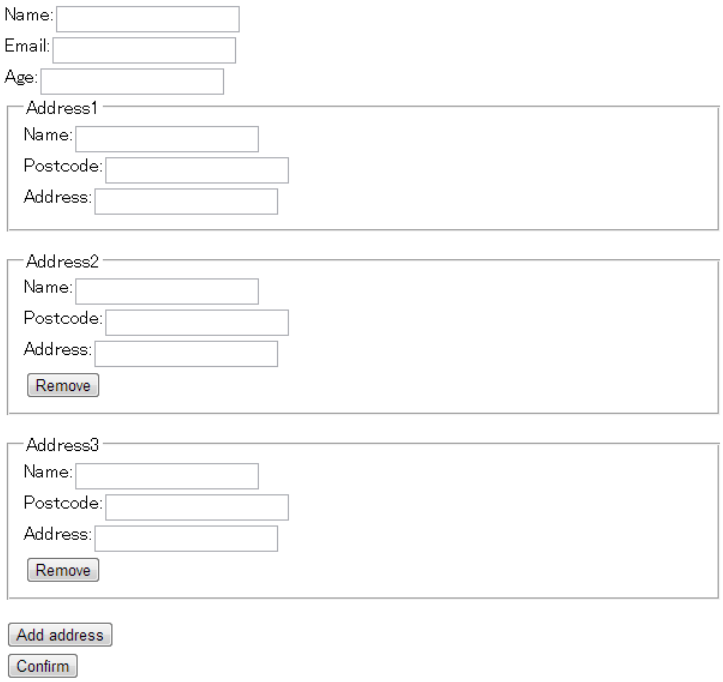
```
// webapp/resources/app/js/AddressesView.js

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);
    }
  });
```

```
    });
```

Form is displayed as follows.



Add 2 address forms by clicking “Add address” button twice.



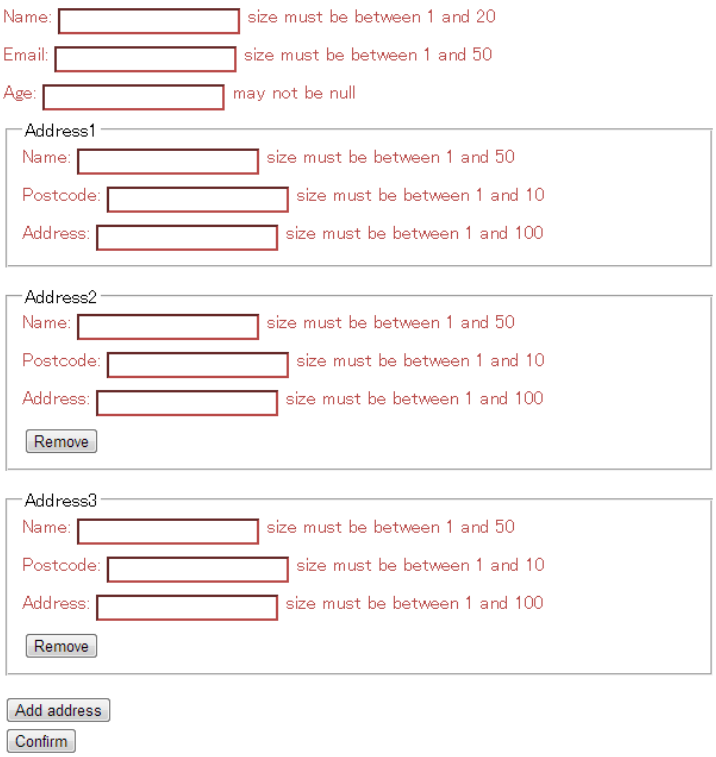
Error message is displayed as follows if this form is sent with all the input fields left blank.

Grouped validation

By creating validation group, input validation rules for a field can be specified for each group.

In the “new user registration” example, add “Must be an adult” rule for the `age` field. Add `country` field also as “Adult” rules differ with country.

To specify group in Bean Validation, set any `java.lang.Class` object representing a group in `group` attribute of the annotation.



Create the following 3 groups (interface) here.

Group	Adult condition
`Chinese`	18 years or more
`Japanese`	20 years or more
`Singaporean`	21 years or more

An example of executing validation using these groups is shown here.

- Form class

```
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
    }
```

S.No.	Description
(1)	Define each group as an interface.
(2)	`@Min.List` annotation is used to specify multiple `@Min` rules on a single field. It is same even while using other annotations.
(3)	Specify corresponding group class in the `group` attribute, in order to define rules for each group. When `group` attribute is not specified, `javax.validation.groups.Default` group is used.
(4)	Add a field which will be used to determine which group is to be applied.

- JSP

  There are no major changes in JSP.

```
<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 class

  By giving a group name to `@Validated` annotation, the rules defined for that group will be applied.

```
    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);
        }
    }
```

S.No.	Description
(1)	The parameter which is used as condition to dividing between the groups must be set to `param` attribute.
(2)	All the annotations, except `@Min` of `age` field, belongs to `Default` group; hence, specifying `Default` is mandatory.

In this example, the check result of the combination of each input value is as follows.

`age` value	`country` value	Input validation result	Error message
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	

Warning: Implementation of this Controller is inadequate; there is no handling when `country` value is neither "cn", "jp" nor "sg". 400 error is returned when unexpected `country` value is encountered.

Next, we can think of a condition where the number of countries increase and adult condition of 18 years or more is be set as a default rule.

Rules are as follows.

Group	Adult condition
`Japanese`	20 years or more
`Singaporean`	21 years or more
Country other than the above-mentioned(`Default`)	18 years or more

- Form class

  In order to specify a value to `Default` group (18 years or more), all groups should be specified explicitly in other annotations as well.

```
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
}
```

S.No.	Description
(1)	Set all groups to annotations other than `@Min` of `age` field as well.
(2)	Set the rule for `Default` group.

- JSP

  No change in JSP

- Controller class

```
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);
        }
    }
```

S.No.	Description
(1)	When the field `country` does not have a value, the request is mapped to a method in which `Default` group is specified in `@Validated` annotation.
(2)	When the field `country` has a value, the request is mapped to a method in which `Default` group is not included in `@Validated` annotation.

Till now, 2 patterns of using grouped validation have been explained.

In the previous pattern, `Default` group is used in Controller class and in the later one, `Default` group is used in form class.

Pattern	Advantages	Disadvantages	Decision points
Using `Default` group in Controller class	`group` attribute need not be set for the rules that need not be grouped.	Since all patterns of group should be defined, it is difficult to define when there are many group patterns.	Should be used when there are only a limited number of group patterns (New create group, Update group and Delete group)
Using `Default` group in form class	Since only the groups that do not belong to the default group need to be defined, it can be handled even if there are many patterns.	`group` attribute should be set for the rules that need not be grouped making the process complicated.	Should be used when there are many group patterns and majority of patterns have a common value.

** If none of the above decision points are applicable, then using Bean Validation itself might not be a good idea.** After reviewing the design, usage of Spring Validator or implementation of validation in business logic should be considered.

Note: In the examples explained so far, the switching of group validation is carried out using request parameter and parameter that can be specified in `@RequestMapping` annotation. It is not possible to switch between groups, if switching is to be performed based on permissions in authentication object or any information which cannot be handled by `@RequestMapping` annotation.

In such a case, `@Validated` annotation must not be used but `org.springframework.validation.SmartValidator` must be used. Group validation can be performed inside the handler method of controller.

```
@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) {
        Class<?> validationGroup = Default.class;
        // logic to determine validation group
        // if (xxx) {
        //     validationGroup = Xxx.class;
        // }
        smartValidator.validate(form, result, validationGroup); // (3)
        if (result.hasErrors()) {
            return "user/createForm";
```

```
            }
            return "user/createConfirm";
        }

    }
```

S.No.	Description
(1)	Inject `SmartValidator`. Since `SmartValidator` can be used if `<mvc:annotation-driven>` setting is carried out so there is no need to define separately.
(2)	Do not use `@Validated` annotation.
(3)	Execute grouped validation using `validate` method of `SmartValidator`. Multiple groups can be specified in `validate` method.

Since logic should not be written in Controller, if switching is possible using request parameters in `@RequestMapping`annotation, `SmartValidator`must not be used.

Correlation item check

For the validation of correlated items, Spring Validator(`Validator` implementing `org.springframework.validation.Validator` interface) or Bean Validation must be used.

Each one of above has been explained below. However, before that, their features and usage have been explained.

Format	Features	Usage
Spring Validator	It is easy to create input validation for a particular class. It is inconvenient to use in Controller.	Input validation implementation of unique business requirements depending on specific form
Bean Validation	Creation of input validation is not as easy as Spring Validator. It is easy to use in Controller.	Common input validation implementation of development project not depending on specific form

Correlation item check implementation using Spring Validator

Implementation method is explained with the help of "reset password" process as an example.

Implement the following rules. Following rules are provided in the "reset password" form.

Field name	Type	Rules	Description
password	`java.lang.String`	Mandatory input 8 or more characters **Must be same as confirmPassword**	Password
confirmPassword	`java.lang.String`	Nothing in particular	Confirm password

Check rule "Must be same as confirmPassword" is validation of correlated items as `password` field and `passwordConfirm` field should have the same value.

- Form class

  other than validation of correlated items, implement using Bean Validation annotation.

```
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
}
```

Note: Password is normally saved in database after hashing it, hence there is no need to check the maximum number of characters.

- Validator class

  Implement validation of correlated items using `org.springframework.validation.Validator` interface.

```
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) {
        PasswordResetForm form = (PasswordResetForm) target;
        String password = form.getPassword();
        String confirmPassword = form.getConfirmPassword();

        if (password == null || confirmPassword == null) { // (3)
            // must be checked by @NotNull
            return;
        }
        if (!password.equals(confirmPassword)) { // (4)
            errors.rejectValue(/* (5) */ "password",
            /* (6) */ "PasswordEqualsValidator.passwordResetForm.password",
            /* (7) */ "password and confirm password must be same.");
        }
    }
}
```

S.No.	Description
(1)	Assign `@Component` to make Validator the target of component scan.
(2)	Decide the argument is check target of this validator or not. Here `PasswordResetForm` class is the target to be checked.
(3)	Use `@NotNull` annotation to check whether the field is `null`. Thereby, consider the value to be valid if the field is `null` in this Validator.
(4)	Implement check logic.
(5)	Specify field name where there is error.
(6)	Specify code name of error message. Here, code is "[validator name].[form attribute name].[property name]" Refer to *Messages to be defined in application-messages.properties* for mesaage definition.
(7)	Set default message to be used when error message does not get resolved using code.

Note: Spring Validator implementation class should be placed in the same package as the Controller.

- Controller class

```
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";
    }
}
```

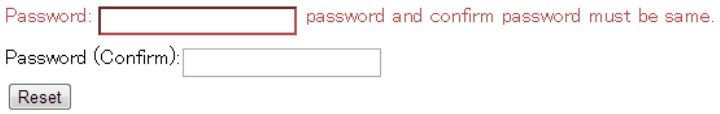
S.No.	Description
(1)	Inject Spring Validator to be used.
(2)	In the method having `@InitBinder` annotation, add Validators using `WebDataBinder.addValidators` method. By this, the added Validator is called when validation is exected with the use of `@Validated` annotation.
(3)	Implement input validation as per the process performed so far.

- JSP

  There are no points to mention for JSP.

```
<!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>
```

Error message as shown below is displayed when form is sent by entering different values in `password` field and `confirmPassword` fields.



Note: When `<form:password>`tag is used, data gets cleared at the time of redisplay.

Note: When multiple forms are used in a single controller, model name should be specified in `@InitBinder("xxx")` in order to limit the target of Validator.

```
@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
}
```

---

implementation of input check of correlated items using Bean Validation

Independent validation rules should be added to implement validation of correlated items using Bean Validation.

It is explained in *How to extend*.

Definition of error messages

Method to change error messages of input validation is explained.

Error messages of Bean Validation in Spring MVC are resolved in the following order.

1. If there is any message which matches with the rule, among the messages defined in `org.springframework.context.MessageSource`, then it will be used as error message (Spring rule).
2. If message cannot be found as mentioned in step 1, then error message is acquired from the `message` attribute of the annotation. (Bean Validation rule)
    1. When the value of `message` attribute is not in "{message key}" format, use that text as error message.
    2. When the value of `message` attribute is in "{message key}" format, search messages corresponding to message key from ValidationMessages.properties under classpath.
        1. When message corresponding to message key is defined, use that message
        2. When message corresponding to message key is not defined, use "{message key}" as error message

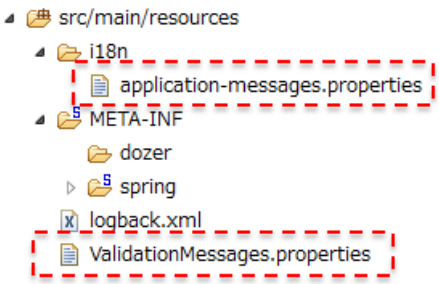
Basically, it is recommended to define error messages in properties file.

Messages should be defined at following places.

- properties file read by `org.springframework.context.MessageSource`
- ValidationMessages.properties under classpath

Considering that the following settings are done in applicationContext.xml, former is called as "application-messages.properties" and latter is called "ValidationMessages.properties".

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>i18n/application-messages</value>
        </list>
    </property>
</bean>
```



This guideline classifies the definition as follows.

Properties file name	Contents to be defined
ValidationMessages.properties	Default error messages of Bean Validation specified by the system
application-messages.properties	Error message of Bean Validation to be overwritten separately Error message of input validation implemented in Spring Validator

When ValidationMessages.properties is not provided, *Default messages provided by Hibernate Validator*is used.

Messages to be defined in ValidationMessages.properties

Define messages for message key specified in `message` attribute of Bean Validation annotation of ValidationMessages.properties under class path (normal src/main/resources).

It is explained below using the following form used at the beginning of *Basic single item check*.

- Form class (re-displayed)

```
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
}
```

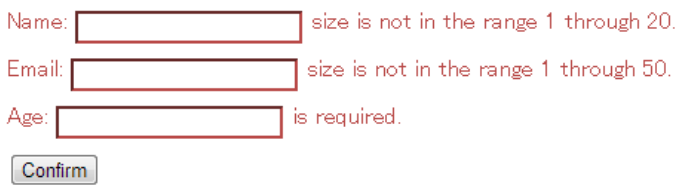
- ValidationMessages.properties

  Change error messages of `@NotNull`, `@Size`, `@Min`, `@Max`, `@Email`.

```
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.
```

S.No.	Description
(1)	It is possible to embed the value of attributes specified in the annotation using `{Attribute name}`.
(2)	It is possible to embed the invalid value using `{value}`.

When the form is sent with input fields left blank after adding the above settings, changed error messages are displayed as shown below.



Warning: Since `{FQCN of annotation.message}` is set in `message` attribute in Bean Validation standard annotations and independent Hibernate Validator annotations, messages can be defined in properties file in the above format. However, since all annotations may not be in this format, Javadoc or source code of the target annotation should be checked.

```
FQCN of annotation.message = Message
```

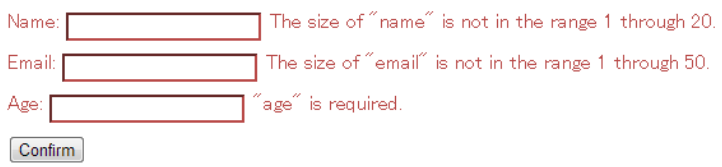
Add `{0}` to message as shown below when field name is to be included in error message.

- ValidationMessages.properties

  Change error message of `@NotNull`, `@Size`, `@Min`, `@Max` and `@Email`.

```
javax.validation.constraints.NotNull.message="{0}" is required.
javax.validation.constraints.Size.message=The size of "{0}" is not in the range {min} thr
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.
```

Error message is changed as follows.



In this way, property name of form class gets displayed on the screen and so it is not user friendly. To display an appropriate field name,it should be defined in **application-messages.properties**in the following format.

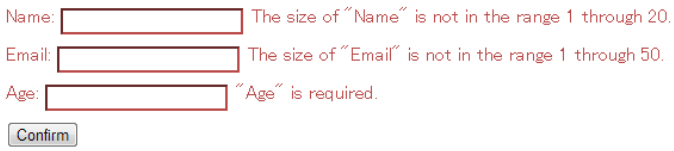
```
form property name=field name to be displayed
```

Adding the same to our example.

- application-messages.properties

```
name=Name
email=Email
age=Age
```

Error messages are changed as follows.



Note: Inserting field name in place of `{0}` is the functionality of Spring and not of Bean Validation. Therefore, the settings for changing field name should be defined in application-messages.properties(`ResourceBundleMessageSource`) which is directly under Spring management.

Messages to be defined in application-messages.properties

Default messages to be used in system are defined in ValidationMessages.properties however, depending on the screen, they may have to be changed from the default value.

In this case, define messages in the following format in application-messages.properties.

```
[annotation name].[form attribute name].[property name] = [target message]
```

Considering that the message for `age` field already exists *Messages to be defined in ValidationMessages.properties*. Override the message for `age` field using the below settings.

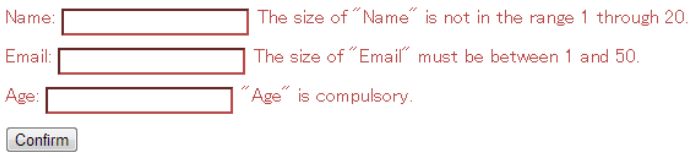
- application-messages.properties

```
# 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
```

Value of attributes of the annotation gets inserted after `{1}` onwards.

Error messages are changed as follows.



Note: There are other formatsas well for the message key application-messages.properties. However, if it is used with the purpose of overwriting some default messages, it should be in `[annotation name].[form attribute name].[property name]` format.

5.5.3 How to extend

Other than standard check rules, bean validation has a mechanism to develop annotations for independent rules .

The method of creating independent rules can be widely classified into the following two broader criteria.

- Combination of existing rules
- Creation of new rules

Basically, the below template can be used to create annotation for each rule.

```
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();
    }
}
```

Creation of Bean Validation annotation by combining existing rules

Consider the following restrictions at the system level and domain level respectively.

At the system level,

- String must be single byte alphanumeric characters
- Numbers must be positive

At the domain level,

- “User ID” must be between 4 and 20 single byte characters
- “Age” must be 1 year or more and 150 years or less

These can be implemented by combining `@Pattern`, `@Size`, `@Min`, `@Max` of the existing rules.

However, if the same rules are to be used at multiple places, settings get distributed and maintainability worsens.

One rule can be created by combining multiple rules. There is an advantage to be able to have not only common regular expression pattern and maximum/minimum values but also error message when an independent annotation is created. By this, reusability and maintainability increases. Even if multiple rules are not combined, it also proves beneficial if used only to give specific value to an attribute.

Implementation example is shown below.

- Implementation example of `@Alphanumeric` annotation which is restricted to single byte alphnumeric characters

```
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();
    }
}
```

S.No.	Description
(1)	This will consolidate error messages and return only the message of this annotation at the time of error.
(2)	Define rules used by this annotation.
(3)	Define default value of error message.

- Implementation example of `@NotNegative` annotation which is restricted to positive number

```
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();
    }
}
```

- Implementation example of `@UserId` annotation which regulates the format of "User ID".

```
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();
    }
}
```

- Implementation example of `@Age` annotation which regulates the constraints on "Age"

```
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();
    }
}
```

Note: If multiple rules are set in a single annotation, their AND condition forms the composite annotation. In Hibernate Validator, `@ConstraintComposition` annotation is provided to implement OR condition. Refer to Hibernate Validator documentfor details.

Creation of Bean Validation annotation by implementing new rules

Any rule can be created by implementing `javax.validation.ConstraintValidator` interface and creating annotation that uses this Validator.

The method of usage is as follows.

- Rules that cannot be implemented by combining the existing rules
- check rule for correlated items
- Business logic check

Rules that cannot be implemented by combining the existing rules

For the rules that cannot be implemented by combining `@Pattern`, `@Size`, `@Min`, `@Max`, implement `javax.validation.ConstraintValidator`.

For example, rules that check ISBN (International Standard Book Number)-13 format are given.

- Annotation

```
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();
    }
}
```

S.No.	Description
(1)	Specify the `ConstraintValidator` implementation class which will get executed when this annotation is used. Multiple constraints can also be specified.

- 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;
    }
}
```

S.No.	Description
(1)	Specify target annotation and field type in generics parameter.
(2)	Implement initialization process in `initialize` method.
(3)	Implement input validation in `isValid` method.
(4)	Consider input value as correct in case of `null`.
(5)	Check ISBN-13 format.

Tip: Example of *fileupload_validator* is classified in this category. Further, in common library as well, *@ExistInCodeList* is implemented in this way.

Check rules for correlated items

Check of correlated items, which span over multiple fields, can be done using Bean Validation as explained in *Correlation item check*.

It is recommended to target generalized rules, in case of deciding to use Bean Validation for check of correlated items.

An example of implementing the rule, "Contents of a field should match with its confirmation field" is given below.

Set constraint of assigning "confirm" as the prefix of confirmation field.

- Annotation

  Define such that annotation for validation of correlated items can be used at class level as well.

```
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();
    }
}
```

S.No.	Description
(1)	Narrow down the target of using this annotation, such that it can be added only to class or annotation.
(2)	Define parameter to pass to annotation.

- 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;
        }
    }

}
```

S.No.	Description
(1)	Use `org.springframework.beans.BeanWrapper` which is very convenient to access JavaBean properties.
(2)	Acquire property value from the form object through `BeanWrapper`.
(3)	Disable the creation of default `ConstraintViolation` object.
(4)	Create independent `ConstraintViolation` object. Define message to be output in `ConstraintValidatorContext.buildConstraintViolationWithTemplate`. Specify field name to output error message in `ConstraintViolationBuilder.addNode`. Refer to JavaDocfor details.

Check below for the changes, if the "Reset password" is re-implemented using `@Confirm` annotation.

- Form class

```
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
}
```

S.No.	Description
(1)	Assign `@Confirm` annotation at class level. By this, form object is passed as an argument to `ConstraintValidator.isValid`.

- Controller class

There is no need to inject Validator and add Validator using `@InitBinder`.

```
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";
    }
}
```

Business logic check

Business logic check should implement *Implementation in Service of domain layer* and store result message in `ResultMessages` object.

Accordingly, it is expected that, *they will be normally displayed at the top of the screen*.

However, there are cases, where business logic error message (such as, "whether the entered user name is already registered") of target input field is to be displayed next to the field. In such a case, service class is injected in Validator class and business logic check is executed in `ConstraintValidator.isValid`.

An example of implementing, "whether the entered user name is already registered" in Bean Validation is shown below.

- Service class

  Implementation class (UserServiceImpl) is omitted.

```
package com.example.sample.domain.service.user;

public interface UserService {

    boolean isUnusedUserId(String userId);

    // omitted other methods
}
```

- Annotation

```
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 class

```
    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)
        }

    }
```

S.No.	Description
(1)	Settings to enable component scan of the Validator class. Target package must be included in `<context:component-scan base-package="..." />` of Bean definition file.
(2)	Inject the service class to be called.
(3)	Return the result of business logic error check. Process should be delegated to service class. Logic must not be written directly in the `isValid` method.

5.5.4 Appendix

Input validation rules provided by Hibernate Validator

Hibernate Validator provides additional validation annotations, in addition to the annotation defined in JSR-303.

Refer to http://docs.jboss.org/hibernate/validator/4.3/reference/en-US/html_single/#section-builtin-constraintsfor the annotation list that can be used for validation.

JSR-303 check rules

JSR-303 annotations are shown below.

Refer to Chapter 6 JSR-303 Specification for details.

Annotation(javax.validation.*)	Target type	Application	Usage example
`@NotNull`	Arbitrary	Validates that the target field is not null.	@NotNull private String id;
`@Null`	Arbitrary	Validates that the target field is null. (Example: usage in group validation)	@Nulll(groups={Update.class}) private String id;
`@Pattern`	String	Whether the target field matches with the regular expression (In case of Hibernate Validator implementation, it is also possible to use it with arbitrary CharSequence inherited class)	@Pattern(regexp = “[0-9]+”) private String tel;
`@Min`	BigDecimal, BigInteger, byte, short, int, long and wrapper (In case of Hibernate Validator implementation, it is also possible to use it with arbitrary CharSequence, Number inherited class. However, only in cases where string can be converted to a number.)	Validate whether the value is greater than the minimum value.	Refer @Max
`@Max`	BigDecimal, BigInteger, byte, short, int, long and wrapper (In case of Hibernate Validator implementation, it is also possible to use it with arbitrary CharSequence, Number	Validate whether the value is less than the maximum value.	@Min(1) @Max(100) private int quantity;

Hibernate Validator check rules

All the major annotations of Hibernate Validator are shown below.

Refer to Hibernate Validator specificationsfor details.

Annotation(org.hibernate.validator.constraints.*)	Target type	Application	Usage example
`@CreditCardNumber`	It is applicable to the any CharSequence inherited class	Validate whether the credit card number is valid as per Luhn algorithm. It does not check whether the credit card number is available.	@CreditCardNumber private String cardNumber;
`@Email`	It is applicable to the any CharSequence inherited class	Validate whether the Email address is complaint with RFC2822.	@Email private String email;
`@URL`	It is applicable to any CharSequence inherited class	Validate whether it is compliant with RFC2396.	@URL private String url;
`@NotBlank`	It is applicable to any CharSequence inherited class	Validate that it is not Null, empty string (“”) and space only.	@NotBlank private String userId;
`@NotEmpty`	It is applicable to Collection, Map, arrays, and any CharSequence inherited class	Validate that it is not Null or empty. @NotEmpty should be used when check is to be done in @NotNull + @Min(1) combination.	@NotEmpty private String password;

Default messages provided by Hibernate Validator

In hibernate-validator-<version>.jar, there is a ValidationMessages.properties file at org/hibernate/validator location, which contains the default messages of all Hibernate provided annotations.

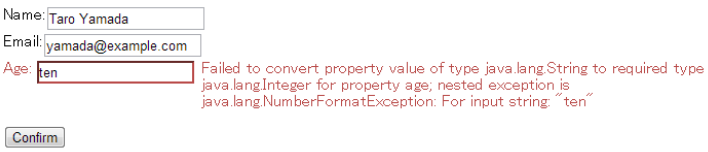
```
javax.validation.constraints.AssertFalse.message = must be false
javax.validation.constraints.AssertTrue.message  = must be true
javax.validation.constraints.DecimalMax.message  = must be less than or equal to {value}
```

```
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
```

Type mismatch

Pertaining the non-String fields of the form object, if values which cannot be converted to the corresponding data-type have been submitted, `org.springframework.beans.TypeMismatchException` is thrown.

In the "New user registration" example, The data-type of "Age" is `Integer`. However, if the value entered in this field cannot be converted to an integer, error message is displayed as follows.



Cause of exception is displayed as it is and is not appropriate as an error message. It is possible to define the error message for type mismatch in the properties file (application-messages.properties) which is read by `org.springframework.context.MessageSource`.

Error messages can be defined with the following rules.

Message key	Message contents	Application
`typeMismatch`	Default message for all type mismatch errors	Default value for entire system
`typeMismatch.[target FQCN]`	Default message for specific type mismatch error	Default value for entire system
`typeMismatch.[form attribute name].[property name]`	Type mismatch error message for specific form field	Customized message for each screen

When the following messages are added to application-messages.properties,

```
# 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.

# field names
name=Name
email=Email
age=Age
```

Error message gets changed as shown below.

Field name can be inserted in the message by specifying `{0}` in the message; This is as per the explaination in *Messages to be defined in application-messages.properties*.

Default messages should be defined.

Tip: Refer to Javadocfor the details of message key rules.

Binding null to blank string field

When the form is sent with input fields left blank in Spring MVC, empty string instead of `null` binds to form object by default.

In this case, the conditions like "blank is allowed but if specified, it must have at least 6 characters" is not satisfied by the use of existing annotations.

To bind `null` instead of empty string to the properties whichout any input, `org.springframework.beans.propertyeditors.StringTrimmerEditor` can be used as follows.

```
@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 ...
}
```

With the help of this configuration, it can be specified in the controller whether empty strings which are to be considered as `null` or not.

@ControllerAdvice can be set as a configuration common to the entire project when you want to set reply empty string to `null` in the entire project.

```
@ControllerAdvice
public class XxxControllerAdvice {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        // bind empty strings as null
        binder.registerCustomEditor(String.class, new StringTrimmerEditor(true));
    }

    // omitted ...
}
```

When this setting is made, all empty String values set to String fields of form object become `null`.

Therefore it must be noted that, it becomes necessary to specify `@NotNull` in case of mandatory check.

5.6 [coming soon] Logging

Coming soon ...

5.7 Exception Handling

This chapter describes exception handling mechanism for web applications created using this guideline.

5.7.1 Overview

This section illustrates handling of exceptions occurring within the boundary of Spring MVC. The scope of description is as follows:

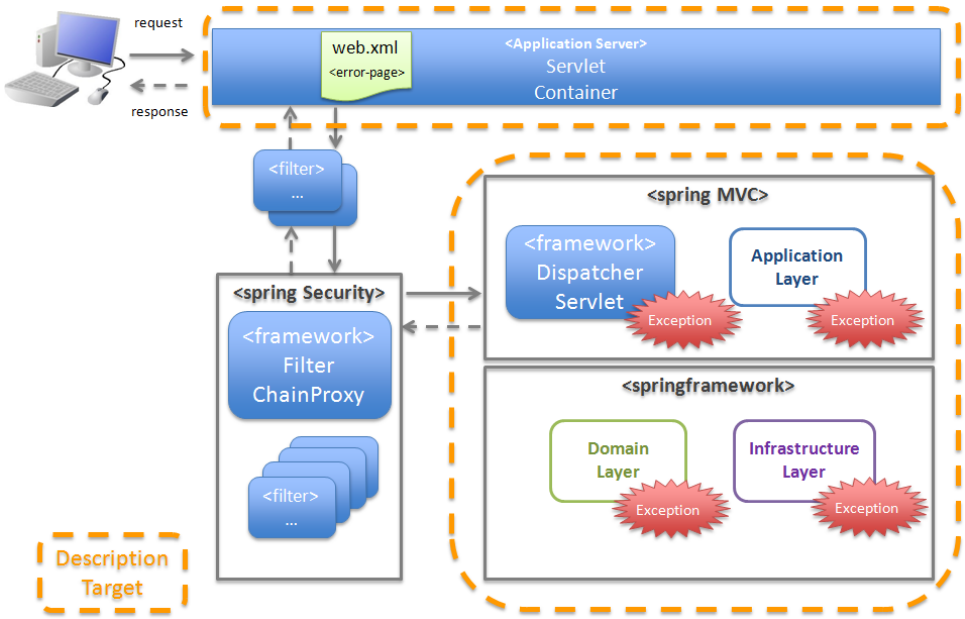


Figure.5.1 **Figure - Description Targets**

1. *Classification of exceptions*
2. *Exception handling methods*

Classification of exceptions

The exceptions that occur when an application is running, can be broadly classified into following 3 categories.

Table.5.1 Table - Types of exceptions that occur when an application is running

Sr. No.	Classification	Description	*Types of exceptions*
(1)	Exceptions wherein cause can be resolved if the user re-executes the operation (by changing input values etc.)	The exceptions wherein cause can be resolved if the user re-executes the operation, are handled in application code.	1. *Business exception* 2. *Library exceptions that occurs during normal operation*
(2)	Exceptions wherein cause cannot be resolved even if the user re-executes the operation	The exceptions wherein cause cannot be resolved even if the user re-executes the operation, are handled using the framework.	1. *System Exception* 2. *Unexpected System Exception* 3. *Fatal Errors*
(3)	Exceptions due to invalid requests from the client	The exceptions which occur due to invalid requests from the client, are handled using the framework.	1. *Framework exception in case of invalid requests*

Note: Who is a target audience for exceptions?

- Application Developer should be aware of exception (1).
- Application Architect should be aware of exceptions (2) and (3).

Exception handling methods

The exceptions which occur when the application is running, are handled using following 4 methods.

For details on flow of each handling method, refer to *Basic Flow of Exception Handling*.

Table.5.2 Table - Exception Handling Methods

Sr. No.	Handling Method	Description	*Exception Handling Patterns*
(1)	Use `try-catch` in the application code to carry out exception handling.	Use this in order to handle exceptions at request (Controller method) level. For details, refer to *Basic flow when the Controller class handles the exception at request level*.	1. *When notifying partial redo of a use case (from middle)*
(2)	Use `@ExceptionHandler` annotation to carry out exception handling in application code.	Use this when exceptions are to be handled at use case (Controller) level. For details, refer to *Basic flow when the Controller class handles the exception at use case level*.	1. *When notifying redo of a use case (from beginning)*
(3)	Use HandlerExceptionResolver mechanism provided by the framework to carry out exception handling.	Use this in order to handle exceptions at servlet level. When `<mvc:annotation-driven>` is specified, HandlerExceptionResolver uses *automatically registered class*, and `SystemExceptionResolver` provided by this framework. For details, refer to *Basic flow when the framework handles the exception at servlet level*.	1. *When notifying that the system or application is not in a normal state* 2. *When notifying that the request contents are invalid*
(4)	Use the error-page function of the servlet container to carry out exception handling.	Use this in order to handle fatal errors and exceptions which are out of the boundary of Spring MVC.	1. *When a fatal error has been detected* 2. *When notifying that an*

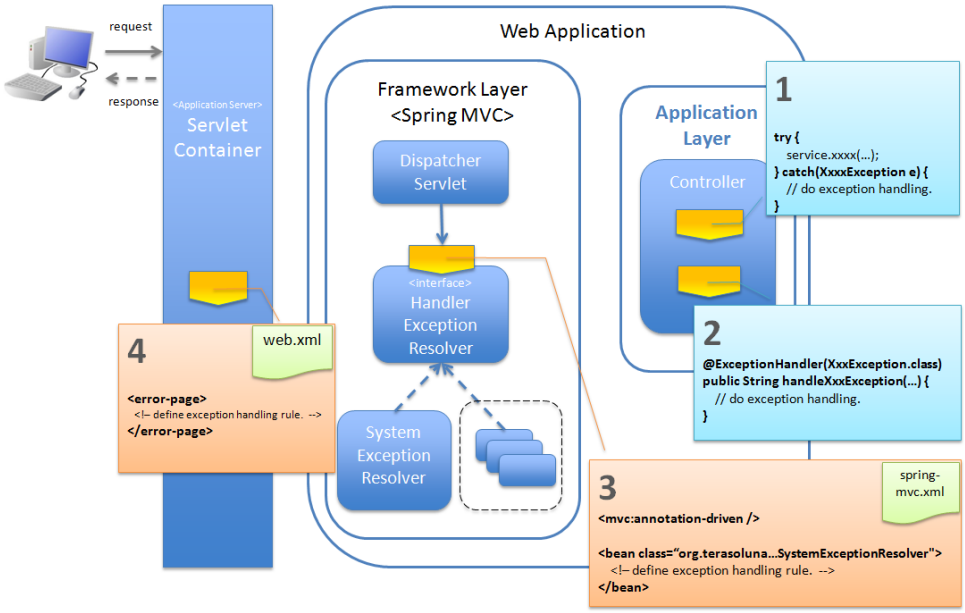


Figure.5.2 **Figure - Exception Handling Methods**

Note: Who will carry out exception handling?

- Application Developer should design and implement (1) and (2).
- Application Architect should design and configure (3) and (4).

Note: About automatically registered HandlerExceptionResolver

When <mvc:annotation-driven> is specified, the roles of HandlerExceptionResolver implementation class which is registered automatically are as follows:

The priority order will be as given below.

Sr. No.	Class (Priority order)	Role
(1)	ExceptionHandlerExceptionResolver (order=0)	This is a class for handling the exceptions by calling the methods of Controller class with `@ExceptionHandler` annotation. This class is necessary for implementing the handling method No. 2.
(2)	ResponseStatusExceptionResolver (order=1)	This is a class for handling the exceptions wherein `@ResponseStatus` is applied as class level annotation. `HttpServletResponse#sendError(int sc, String msg)` is called with the value specified in `@ResponseStatus`.
(3)	DefaultHandlerExceptionResolver (order=2)	This is a class for handling framework exceptions in Spring MVC. `HttpServletResponse#sendError(int sc)` is called using the value of HTTP response code corresponding to framework exception. For details on HTTP response code to be set, refer to *HTTP response code set by DefaultHandlerExceptionResolver*.

Note: What is a role of SystemExceptionResolver provided by common library?

This is a class for handling exceptions which are not handled by HandlerExceptionResolver which is registered automatically when <mvc:annotation-driven> is specified. Therefore, the order of priority of this class should be set after DefaultHandlerExceptionResolver.

Note: About @ControllerAdvice annotation added from Spring Framework 3.2

`@ControllerAdvice` made exception handling possible using `@ExceptionHandler` at servlet level. If methods with `@ExceptionHandler` annotation are defined in a class with `@ControllerAdvice` annotation, then exception handling carried out in the methods with `@ExceptionHandler` annotation can be applied to all the Controllers in Servlet. When implementing the above in earlier versions, it was necessary to define methods with `@ExceptionHandler` annotation in Controller base class and inherit each Controller from the base class.

Note: Where to use @ControllerAdvice annotation?

1. When logic other than determining the View name and HTTP response code is necessary for exception handling to be carried out at servlet level, (If only determining View name and HTTP response code is sufficient, it can be implemented using `SystemExceptionResolver`)
2. In case of exception handling carried out at servlet level, when response data is to be created by serializing the error model (JavaBeans) in JSON or XML formats without using template engines such as JSP. (Used for error handling at the time of creating Controller for AJAX or REST).

5.7.2 Detail

1. *Types of exceptions*
2. *Exception Handling Patterns*
3. *Basic Flow of Exception Handling*

Types of exceptions

There are 6 types of exceptions that occur in a running application.

Table.5.3 Table - Types of Exceptions

Sr. No.	Types of Exceptions	Description
(1)	*Business exception*	Exception to notify a violation of a business rule
(2)	*Library exceptions that occurs during normal operation*	Amongst the exceptions that occur in framework and libraries, the exception which is likely to occur when the system is operating normally.
(3)	*System Exception*	Exception to notify detection of a state which should not occur when the system is operating normally
(4)	*Unexpected System Exception*	Unchecked exception that does not occur when the system is operating normally
(5)	*Fatal Errors*	Error to notify an occurrence of a fatal error impacting the entire system (application)
(6)	*Framework exception in case of invalid requests*	Exception to notify that the framework has detected invalid request contents

Business exception

Exception to notify a violation of a business rule

This exception is generated in domain layer.

The situations in the system such as below are pre-defined and hence need not be dealt by the system administrator.

- If the reservation date is past the deadline when making travel reservations
- If a product is out of stock when it is ordered
- etc ...

Note: Corresponding Exception Class

- `org.terasoluna.gfw.common.exception.BusinessException` (Class provided by common library).
- When handling is to be carried out at detailed level, exception class that inherits BusinessException should be created.
- If the requirements cannot be met by the exception class provided by common library, a business exception class should be created for each project.

Library exceptions that occurs during normal operation

Amongst the exceptions that occur in the framework and libraries, **the exception which is likely to occur when the system is operating normally.**
The exceptions that occur in the framework and libraries cover the exception classes in Spring Framework or other libraries.
Situations such are below are pre-defined and hence need not be dealt by the system administrator.

- Optimistic locking exception and pessimistic locking exception that occur if multiple users attempt to update same data simultaneously.
- Unique constraint violation exception that occurs if multiple users attempt to register same data simultaneously.
- etc ...

Note: Examples of corresponding Exception Classes

- `org.springframework.dao.OptimisticLockingFailureException` (Exception that occurs if there is an error due to optimistic locking).
- `org.springframework.dao.PessimisticLockingFailureException` (Exception that occurs if there is an error due to pessimistic locking).

- `org.springframework.dao.DuplicateKeyException` (Exception that occurs if there is a unique constraint violation).
- etc ...

Todo

Currently it has been found that unexpected errors occur if JPA(Hibernate) is used.

- In case of unique constraint violation, `org.springframework.dao.DataIntegrityViolationException` occurs and not `DuplicateKeyException`.
- If pessimistic locking fails, the child class of `org.springframework.dao.UncategorizedDataAccessExcepti` occurs and not `PessimisticLockingFailureException`.

`UncategorizedDataAccessException` that occurs at the time of pessimistic locking error, is classified as system error, hence handling it in the application is not recommended. However, there might be cases wherein this exception may need to be handled. This exception can be handled since exception notifying the occurrence of pessimistic locking error is saved as the cause of exception.

=>Further analysis

The current behavior is as follows:

- 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 + Unique constraint
  - org.springframework.dao.DataIntegrityViolationException
  - Caused by: org.hibernate.exception.ConstraintViolationException

---

System Exception

Exception to notify that a state which should not occur when the system is operating normally has been detected.

This exception should be generated in application layer and domain layer.

The exception needs to be dealt by the system administrator.

- If the master data, directories, files, etc. those should have pre-existed, do not exist
- Amongst the checked exceptions that occur in the framework, libraries, if exceptions classified as system errors are caught (IOException during file operations etc.).
- etc ...

Note: Corresponding Exception Class

- `org.terasoluna.gfw.common.exception.SystemException` (Class provided by common library).
- When the error screen of View and HTTP response code need to be switched on the basis of the cause of each error, SystemException should be inherited for each error cause and the mapping between the inherited exception class and the error screen should be defined in the bean definition of SystemExceptionResolver.
- If the requirements are not met by the system exception class provided in the common library, a system exception class should be created in each respective project.

Unexpected System Exception

Unchecked exception that does not occur when the system is operating normally.

Action by system administrator or analysis by system developer is necessary.

Unexpected system exceptions should not be handled (try-catch) in the application code.

- When bugs are hidden in the application, framework or libraries
- When DB Server etc. is down
- etc ...

Note: Example of Corresponding Exception Class

- `java.lang.NullPointerException` (Exception caused due to bugs).
- `org.springframework.dao.DataAccessResourceFailureException` (Exception that occurs when DB server is down).
- etc ...

Fatal Errors

Error to notify that a fatal problem impacting the entire system (application), has occurred.

Action/recovery by system administrator or system developer is necessary.

Fatal Errors (error that inherits java.lang.Error) must not be handled (try-catch) in the application code.

- If the memory available in Java virtual machine is inadequate
- etc ...

Note: Example of corresponding Error Class

- `java.lang.OutOfMemoryError` (Error due to inadequate memory).
- etc ...

Framework exception in case of invalid requests

Exception to notify that the framework has detected invalid request contents.

This exception occurs in the framework (Spring MVC).

The cause lies at client side; hence it need not be dealt by the system administrator.

- Exception that occurs when a request path for which only POST method is permitted, is accessed using GET method.
- Exception that occurs when type-conversion fails for the values extracted from URI using @PathVariable annotation.
- etc ...

Note: Example of corresponding Exception Class

- `org.springframework.web.HttpRequestMethodNotSupportedException` (Exception that occurs when the access is made through a HTTP method which is not supported).
- `org.springframework.beans.TypeMismatchException` (Exception that occurs if the specified value cannot be converted to URI).
- etc ...

Class for which HTTP status code is "4XX" in the list given at *HTTP response code set by DefaultHandlerExceptionResolver*.

Exception Handling Patterns

There are 6 types of exception handling patterns based on the purpose of handling.

(1)-(2) should be handled at use case level and (3)-(6) should be handled at the entire system (application) level.

Table.5.4 Table - Exception Handling Patterns

Sr. No.	Purpose of handling	Types of exceptions	Handling method	Handling location
(1)	*When notifying partial redo of a use case (from middle)*	1. *Business exception*	Application code (try-catch)	Request
(2)	*When notifying redo of a use case (from beginning)*	1. *Business exception* 2. *Library exceptions that occurs during normal operation*	Application code (@ExceptionHandler)	Use case
(3)	*When notifying that the system or application is not in a normal state*	1. *System Exception* 2. *Unexpected System Exception*	Framework (Handling rules are specified in `spring-mvc.xml`)	Servlet
(4)	*When notifying that the request contents are invalid*	1. *Framework exception in case of invalid requests*	Framework	Servlet
(5)	*When a fatal error has been detected*	1. *Fatal Errors*	Servlet container (Handling rules are specified in `web.xml`)	Web application
(6)	*When notifying that an exception has occurred in the presentation layer (JSP etc.)*	1. All the exceptions and errors that occur in the presentation layer	Servlet container (Handling rules are specified in `web.xml`)	Web application

When notifying partial redo of a use case (from middle)

When partial redo (from middle) of a use case is to be notified, catch (try-catch) the exception in the application code of Controller class and carry out exception handling at request level.

Note: Example of notifying partial redo of a use case

- When an order is placed through a shopping site, if business exception notifying stock shortage occurs.

  In such a case, order can be placed if the no. of items to be ordered is reduced; hence display a screen on which no. of items can be changed and prompt a message asking to change the same.

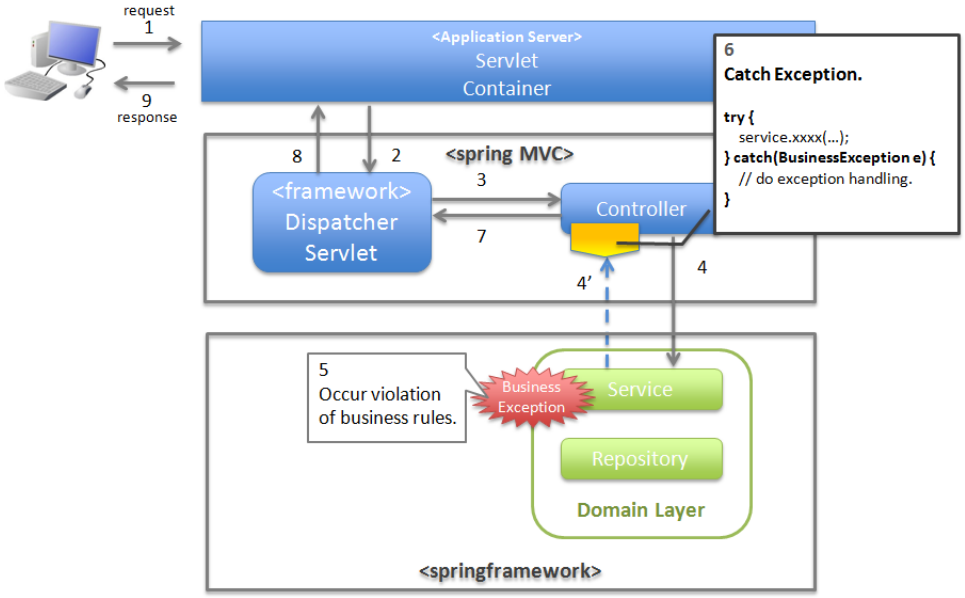


Figure.5.3 **Figure - Handling method when notifying partial redo of a use case (from middle)**

When notifying redo of a use case (from beginning)

When redo of a use case (from beginning) is to be notified, catch the exception using `@ExceptionHandler`, and carry out exception handling at use case level.

Note: Example when notifying redo of a use case (from beginning)

- At the time of changing the product master on shopping site (site for administrator), it has been changed by other operator (in case of optimistic locking exception).

  In such a case, the operation needs to be carried out after verifying the changes made by the other user; hence display the front screen of the use case (for example: search screen of product master) and prompt a message asking the user to perform the operation again.

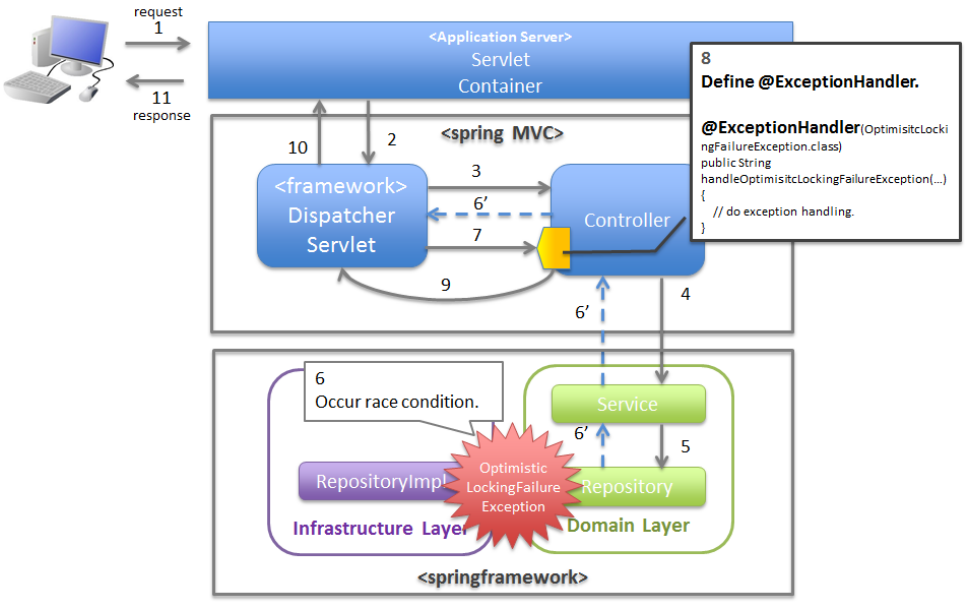


Figure.5.4 **Figure - Handling method when notifying redo of a use case (from beginning)**

When notifying that the system or application is not in a normal state

When an exception to notify that system or application is not in a normal state is detected, catch the exception using SystemExceptionResolver and carry out exception handling at servlet level.

Note: Examples for notifying that the system or application is not in a normal state

- In case of a use case for connecting to an external system, if an exception occurs notifying that the external system is blocked.

  In such a case, since use case cannot be executed until external system resumes service, display the error screen, and notify that the use case cannot be executed till the external system resumes service.

- When searching master information with the value specified in the application, if the corresponding master information does not exist.

  In such a case, there is a possibility of bug in master maintenance function or of error (release error) in data input by the system administrator; hence display the system error screen and notify that a system error has occurred.

- When IOException occurs from the API during file operations.

  This could be a case of disk failure etc.; hence display the system error screen and notify that system error has occurred.

- etc ...

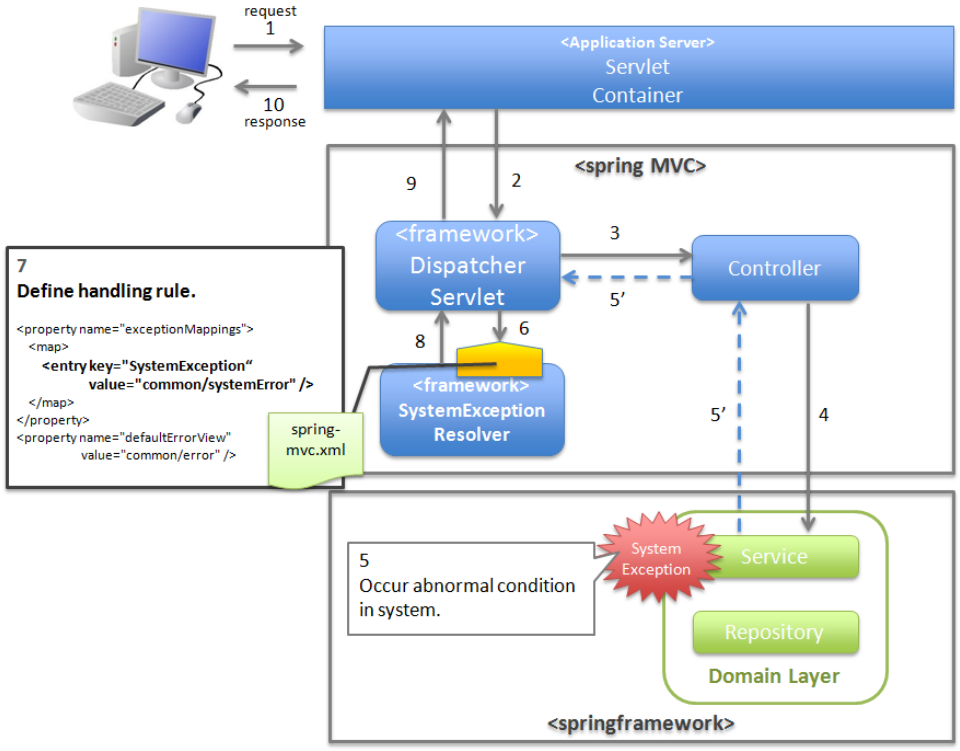


Figure.5.5 **Figure - Handling method when an exception to notify that system or application is not in a normal state is detected**

When notifying that the request contents are invalid

When notifying that an invalid request is detected by the framework, catch the exception using DefaultHandlerExceptionResolver, and carry out exception handling at servlet level.

Note: Example when notifying that the request contents are invalid

- When a URI is accessed using GET method, while only POST method is permitted.

  In such a case, it is likely that the access has been made directly using the Favorites feature etc. of the browser; hence display the error screen and notify that the request contents are invalid.

- When value cannot be extracted from URI using `@PathVariable` annotation

  In such a case, it is likely that the access has been made directly by replacing the value of the address bar on the browser; hence display the error screen and notify that the request contents are invalid.

- etc ...

When a fatal error has been detected

When a fatal error has been detected, catch the exception using servlet container and carry out exception handling at web application level.

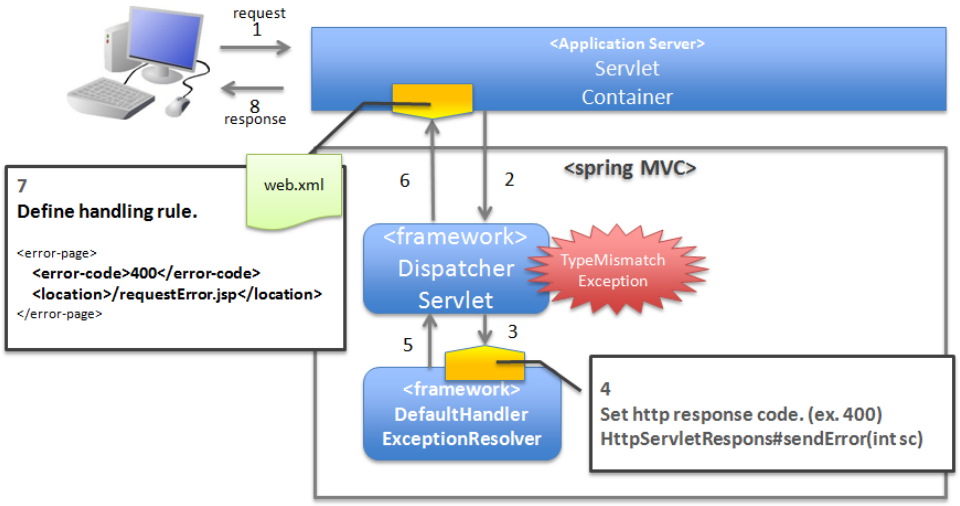


Figure.5.6 **Figure - Handling method when notifying that the request contents are invalid**

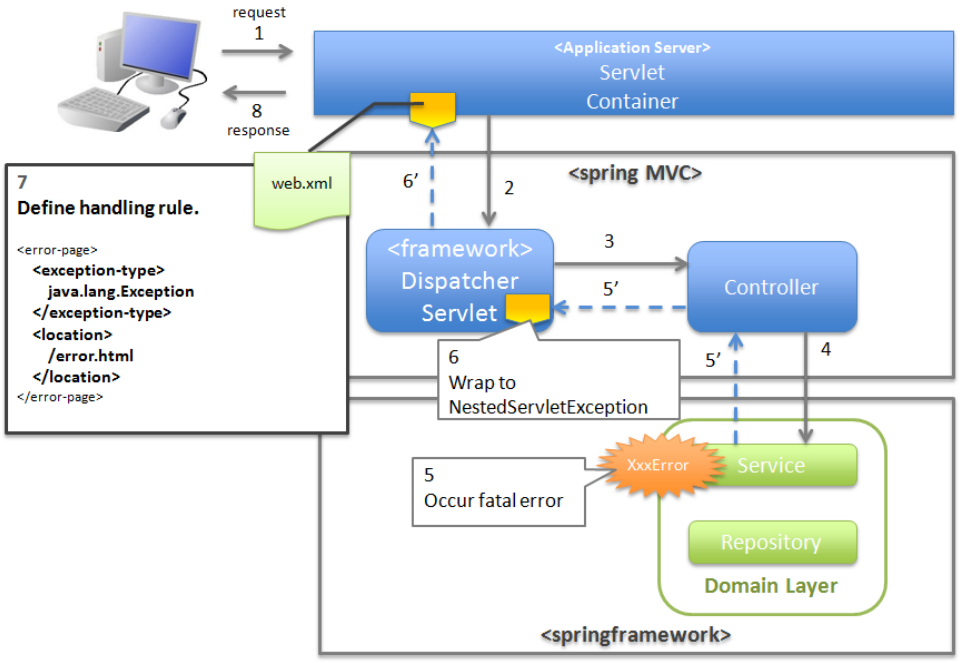


Figure.5.7 **Figure - Handling method when a fatal error has been detected**

When notifying that an exception has occurred in the presentation layer (JSP etc.)

When notifying that an exception has occurred in the presentation layer (JSP etc.), catch the exception using servlet container and carry out exception handling at web application level.

Basic Flow of Exception Handling

The basic flow of exception handling is shown below.

For an overview of classes provided by this framework, refer to *Exception handling classes provided by the*

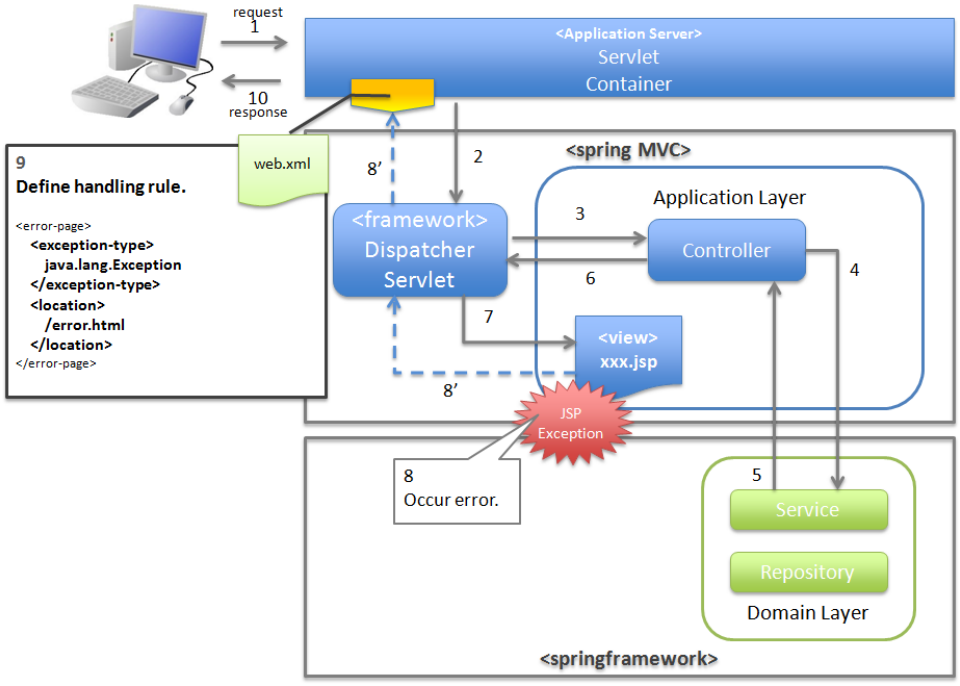


Figure.5.8 **Figure - Handling method when notifying an occurrence of exception in the presentation layer (JSP etc.)**

common library.

The processing to be implemented in the application code is indicated in Bold.

The log of stack trace and exception messages is output by the classes (Filter and Interceptor class) provided by the common library.

When any information other than exception messages and stack trace needs to be logged, it should be implemented separately in each of the logic.

This section describes the flow of exception handling; hence the description related to flow till the calling of Service class is omitted.

1. *Basic flow when the Controller class handles the exception at request level*
2. *Basic flow when the Controller class handles the exception at use case level*
3. *Basic flow when the framework handles the exception at servlet level*
4. *Basic flow when the servlet container handles the exception at web application level*

Basic flow when the Controller class handles the exception at request level

In order to handle the exception at request level, catch (try-catch) the exception in the application code of the Controller class.

Refer to the figure below:

It illustrates the basic flow at the time of handling a business exception

(`org.terasoluna.gfw.common.exception.BusinessException`) provided by this framework.

Log is output using interceptor (`org.terasoluna.gfw.common.exception.ResultMessagesLoggingInterceptor`) which records that an exception holding the result message has occurred.

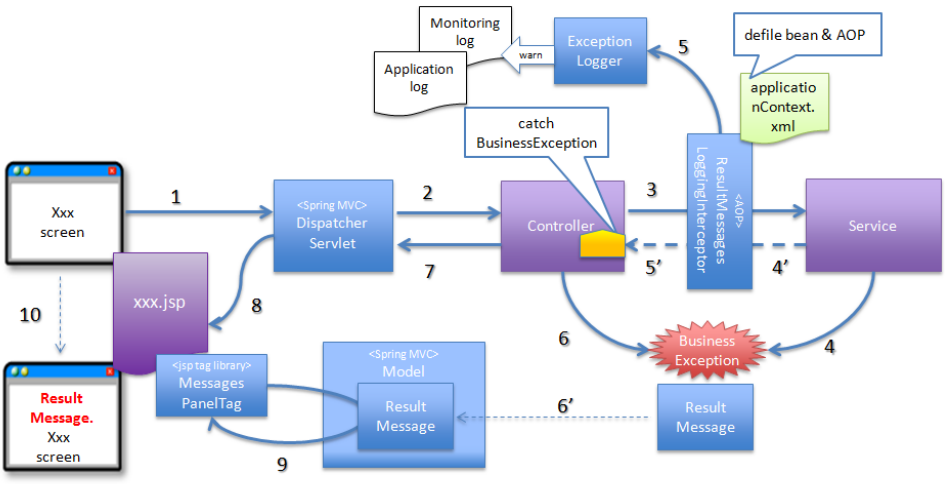


Figure.5.9 **Figure - Basic flow when the Controller class handles the exception at request level**

4. In Service class, BusinessException is generated and thrown.
5. ResultMessagesLoggingInterceptor calls ExceptionLogger, and outputs log of warn level (monitoring log and application log). ResultMessagesLoggingInterceptor class outputs logs only when sub exception (BusinessException/ResourceNotFoundException) of ResultMessagesNotificationException occurs.
6. **Controller class catches BusinessException, extracts the message information (ResultMessage) from BusinessException and sets it to Model for screen display(6').**
7. **Controller class returns the View name.**
8. DispatcherServlet calls JSP corresponding to the returned View name.
9. **JSP acquires message information (ResultMessage) using MessagesPanelTag and generates HTML code for message display.**
10. The response generated by JSP is displayed.

Basic flow when the Controller class handles the exception at use case level

In order to handle the exception at use case level, catch the exception using `@ExceptionHandler` of Controller class.

Refer to the figure below.

It illustrates the basic flow at the time of handling an arbitrary exception (`XxxException`).

Log is output using interceptor (`org.terasoluna.gfw.web.exception.HandlerExceptionResolverLoggingInterceptor`). This interceptor records that the exception is handled using HandlerExceptionResolver.

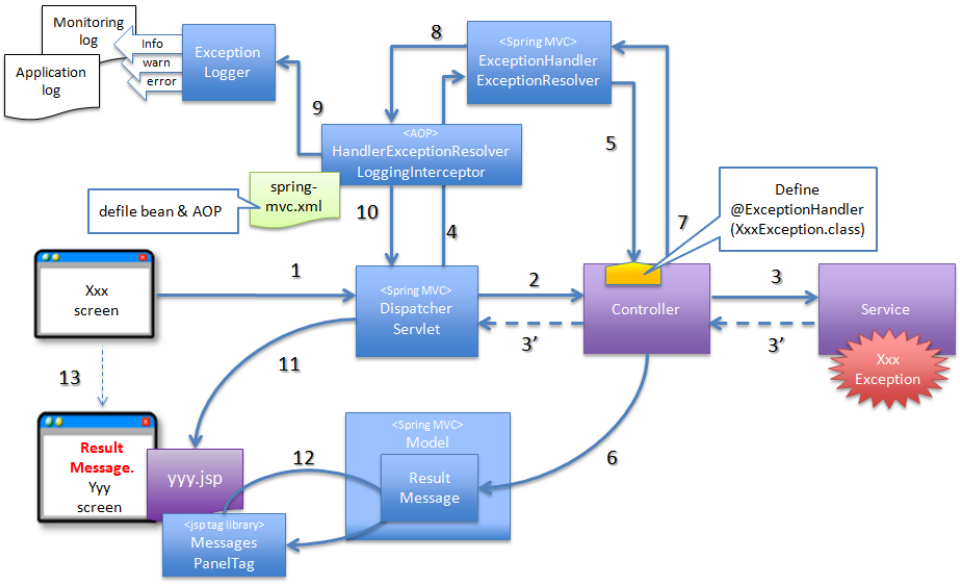


Figure.5.10 **Figure - Basic flow when the Controller class handles the exception at use case level**

3. Exception (XxxException) is generated in the Service class called from Controller class.
4. DispatcherServlet catches XxxException and calls ExceptionHandlerExceptionResolver.
5. ExceptionHandlerExceptionResolver calls exception handling method provided in Controller class.
6. **Controller class generates message information (ResultMessage) and sets it to the Model for screen display.**
7. **Controller class returns the View name.**
8. ExceptionHandlerExceptionResolver returns the View name returned by the Controller.
9. HandlerExceptionResolverLoggingInterceptor calls ExceptionLogger and outputs logs (monitoring log and application log) (at info, warn, error levels) corresponding to the exception code.
10. HandlerExceptionResolverLoggingInterceptor returns View name returned by ExceptionHandlerExceptionResolver.
11. DispatcherServlet calls JSP that corresponds to the returned View name.
12. **JSP acquires the message information (ResultMessage) using MessagesPanelTag and generates HTML code for message display.**
13. The response generated by JSP is displayed.

Basic flow when the framework handles the exception at servlet level

In order to handle the exception using the framework (at servlet level), catch the exception using SystemExceptionResolver.

Refer to the figure below.

It illustrates the basic flow at the time of handling the system exception (`org.terasoluna.gfw.common.exception.SystemException`) provided by this framework using `org.terasoluna.gfw.web.exception.SystemExceptionResolver`.

Log is output using interceptor (`org.terasoluna.gfw.web.exception.HandlerExceptionResolverLoggingInterceptor`) which records the exception specified in the argument of exception handling method.

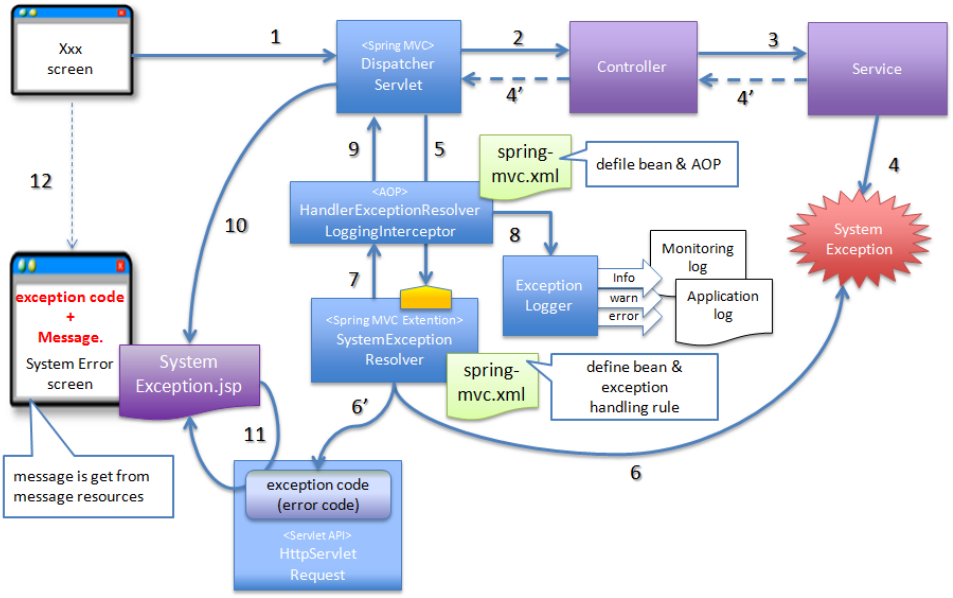


Figure.5.11 **Figure - Basic flow when the framework handles the exception at servlet level**

4. A state equivalent to the system exception is detected in Service class; hence throw a SystemException.
5. DispatcherServlet catches SystemException, and calls SystemExceptionResolver.
6. SystemExceptionResolver acquires exception code from SystemException and sets it to HttpServletRequest for screen display (6').
7. SystemExceptionResolver returns the View name corresponding to SystemException.
8. HandlerExceptionResolverLoggingInterceptor calls ExceptionLogger and outputs logs (monitoring log and application log) (at info, warn, error levels) corresponding to the exception code.
9. HandlerExceptionResolverLoggingInterceptor returns the View name returned by SystemExceptionResolver.

10. DispatcherServlet calls the JSP corresponding to the returned View name.
11. **JSP acquires the exception code from HttpServletRequest and inserts it in HTML code for message display.**
12. The response generated by JSP is displayed.

Basic flow when the servlet container handles the exception at web application level

In order to handle exceptions at web application level, catch the exception using servlet container.

Fatal errors, exceptions which are not handled using the framework (exceptions in JSP etc.) and exceptions occurred in Filter are to be handled using this flow.

Refer to the figure below.

It illustrates the basic flow at the time of handling java.lang.Exception by “error page”.

Log is output using servlet filter (`org.terasoluna.gfw.web.exception.ExceptionLoggingFilter`) which records that an unhandled exception has occurred.

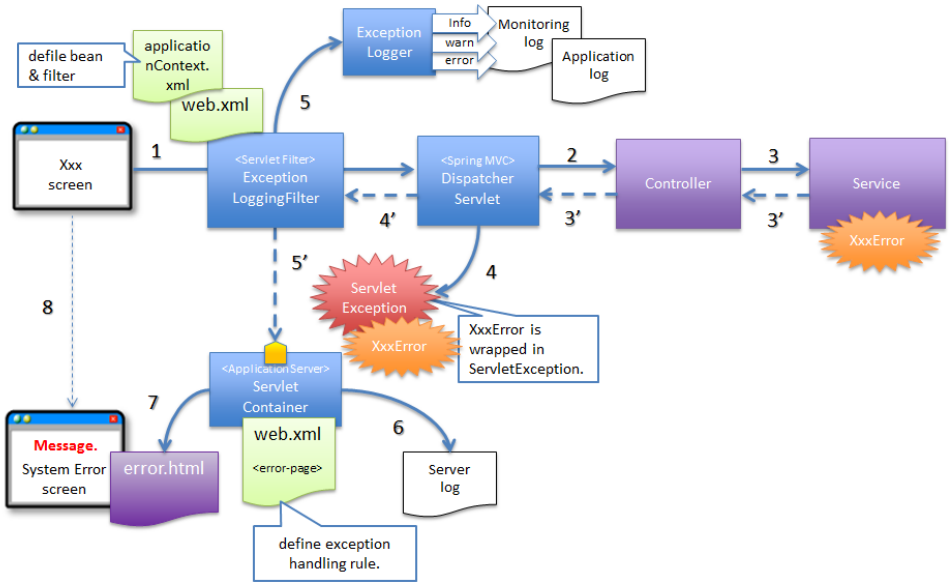


Figure.5.12 **Figure - Basic flow when servlet container handles the exception at web application level**

4. DispatcherServlet catches XxxError, wraps it in ServletException and then throws it.
5. ExceptionLoggingFilter catches ServletException and calls ExceptionLogger. ExceptionLogger outputs logs (monitoring log and application log) (at info, warn, error levels) corresponding to the exception code. ExceptionLoggingFilter re-throws ServletException.
6. ServletContainer catches ServletException and outputs logs to server log. Log level varies depending on the application server.

7. ServletContainer calls the View (HTML etc.) defined in `web.xml`.
8. The response generated by the View which is called by the ServletContainer, is displayed.

5.7.3 How to use

The usage of exception handling funcationality is described below.

For exception handling classes provided by common library, refer to *Exception handling classes provided by the common library*.

1. *Application Settings*
2. *Coding Points (Service)*
3. *Coding Points (Controller)*
4. *Coding points (JSP)*

Application Settings

The application settings required for using exception handling are shown below.

Further, these settings are already done in blank project. Hence, it will work only by doing changes given in **[Location to be customized for each project]**section.

1. *Common Settings*
2. *Domain Layer Settings*
3. *Application Layer Settings*
4. *Servlet Container Settings*

Common Settings

1. Add bean definition of logger class (`ExceptionLogger`) which will output exception log.

- **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>
```

Sr. No.	Description
(1)	Add `ExceptionCodeResolver` to bean definition.
(2)	Specify mapping between name of the exception to be handled and applicable "Exception Code (Message ID)". In the above example, if the "BusinessException" is included in the class name of exception class (or parent class), "w.xx.fw.8001" will be the "Exception code (Message ID)" and if "ResourceNotFoundException" is included in the class name of exception class (or parent class), "w.xx.fw.5001" will be the "Exception code (Message ID)". --- **Note: About the Exception Code (Message ID)** The exception code is defined here for taking into account the case wherein message ID is not specified in generated "BusinessException", however, it is recommended that you specify the "Exception Code (Message ID)" at the implementation side which generates the "BusinessException" (this point is explained later). Specification of "Exception Code (Message ID)" for "BusinessException" is an alternative measure in case it is not specified when "BusinessException" occurs. --- **[Location to be customized for each project]**
(3)	Specify default "Exception Code (Message ID)". In the above example, if "BusinessException" or "ResourceNotFoundException" is not included in the class names of exception class (or parent class), " e.xx.fw.9001" will be "Exception code (Message ID)". --- **[Location to be customized for each project]** --- **Note: Exception Code (Message ID)** Exception code is only to be output to log (It can

2. Add log definition.

- **logback.xml**

  Add log definition for monitoring log.

```
<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>
```

Sr. No.	Description
(1)	Specify appender definition used for outputting monitoring log. In the above example, an appender to be output to a file has been specified, however the appender used should be consider as per the system requirements. **[Location to be customized for each project]**
(2)	Specify logger definition for monitoring log. When creating ExceptionLogger, if any logger name is not specified, the above settings can be used as is. **Warning: About additivity setting value** Specify `false`. If `true` is specified, the same log will be output by upper level logger (for example, root).
(3)	Specify output level. In ExceptionLogger, 3 types of logs of info, warn, error are output; however, the level specified should be as per the system requirements. The guideline recommends Error level. **[Location to be customized for each project]**
(4)	Specify the appender which will act as output destination. **[Location to be customized for each project]**

Add log definition for application log.

```
<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>
```

Sr. No.	Description
(1)	Specify appender definition used for outputting application log. In the above example, an appender to be output to a file has been specified, however the appender used should be consider as per the system requirements. **[Location to be customized for each project]**
(2)	Specify logger definition for application log. When creating ExceptionLogger, if any logger name is not specified, the above settings can be used as is. **Note: Appender definition for outputting application log** Rather than defining a separate appender for logging exceptions, it is recommended to use the same appender which is used for logging in application code or framework. By using same output destination, it becomes easier to track the process until an exception occurs.
(3)	Specify the output level. In ExceptionLogger, 3 types of logs of info, warn, error are output; however, the level specified should be as per the system requirements. This guideline recommends info level. **[Location to be customized for each project]**
(4)	Log is transmitted to root since appender is not specified for the logger set in (2). Therefore, specify an appender which will act as output destination. Here, it will be output to "STDOUT" and "APPLICATION_LOG_FILE". **[Location to be customized for each project]**

Domain Layer Settings

When exceptions (BusinessException,ResourceNotFoundException) holding ResultMessages occur, add AOP settings and bean definition of interceptor class (`ResultMessagesLoggingInterceptor`) for log output.

- **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>
```

Sr. No.	Description
(1)	Add beand definition of `ResultMessagesLoggingInterceptor`.
(2)	Inject the logger which outputs exception log. Specify “exceptionLogger” defined in `applicationContext.xml`.
(3)	Apply ResultMessagesLoggingInterceptor for the method of the Service class (with `@Service` annotation).

Application Layer Settings

Add to bean definition, the class (`SystemExceptionResolver`) used for handling the exceptions which are not handled by HandlerExceptionResolver registered automatically when <mvc:annotation-driven> is specified. .

- **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>
```

Sr. No.	Description
(1)	Add `SystemExceptionResolver` to bean definition.
(2)	Inject the object that resolves exception code (Message ID). Specify “exceptionCodeResolver” defined in `applicationContext.xml`.
(3)	Specify the order of priority for handling. The value can be “3”. When `<mvc:annotation-driven>` is specified, automatically *registered class* is given higher priority. **ヒント: Method to disable exception handling carried out by DefaultHandlerExceptionResolver** When exception handling is carried out by `DefaultHandlerExceptionResolver`, HTTP response code is set; however since View is not resolved, it needs to be resolved using Error Page element of `web.xml`. When it is required to resolve the View using `HandlerExceptionResolver` and not in `web.xml`, then priority order of `SystemExceptionResolver` should be set to “1”. By doing this, the handling process can be executed before `DefaultHandlerExceptionResolver`. For mapping of HTTP response codes when handling is done by `DefaultHandlerExceptionResolver`, refer to *HTTP response code set by DefaultHandlerExceptionResolver*.
(4)	Specify the mapping between name of the exception to be handled and View name. In the above settings, if class name of the exception class (or parent class) includes ”.DataAccessException”,

AOP settings and interceptor class (`HandlerExceptionResolverLoggingInterceptor`) in order to output the log of exceptions handled by `HandlerExceptionResolver` should be added to bean definition.

- **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>
```

Sr. No.	Description
(1)	Add `ExceptionHandlerLoggingInterceptor` to bean definition.
(2)	Inject the logger object which outputs exception log. Specify the "exceptionLogger" defined in `applicationContext.xml`.
(3)	Apply `HandlerExceptionResolverLoggingInterceptor` to resolveException method of `HandlerExceptionResolver` interface. As per default settings, this class will not output the log for common library provided `org.terasoluna.gfw.common.exception.ResultMess` class and its subclasses. The reason the exceptions of the sub class of `ResultMessagesNotificationException` are excluded from the log output is because their log output is carried out by `org.terasoluna.gfw.common.exception.ResultMess` If default settings need to be changed, refer to *About HandlerExceptionResolverLoggingInterceptor settings*.

Filter class (`ExceptionLoggingFilter`) used to output log of fatal errors and exceptions that are out of the boundary of Spring MVC should be added to bean definition and `web.xml`.

- **applicationContext.xml**

```
<!-- Filter. -->
<bean id="exceptionLoggingFilter"
    class="org.terasoluna.gfw.web.exception.ExceptionLoggingFilter" > <!-- (1) -->
    <property name="exceptionLogger" ref="exceptionLogger" /> <!-- (2) -->
</bean>
```

Sr. No.	Description
(1)	Add `ExceptionLoggingFilter` to bean definition.
(2)	Inject the logger object which outputs exception log. Specify the "exceptionLogger" defined in `applicationContext.xml`.

- **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>
```

Sr. No.	Description
(1)	Specify the filter name. Match it with the bean name of `ExceptionLoggingFilter` defined in `applicationContext.xml`.
(2)	Specify the filter class. The value should be fixed to `org.springframework.web.filter.DelegatingFilte`
(3)	Specify the name of the filter to be mapped. The value specified in (1).
(4)	Specify the URL pattern to which the filter must be applied. It is recommended that you use `/*` for outputting log of fatal errors and exceptions that are out of the boundary of Spring MVC.

- Output Log

```
date:2013-09-25 19:51:52    thread:tomcat-http--3   X-Track:f94de92148f1489b9ceeac3b2f17c969
```

Servlet Container Settings

Add error-page definition for Servlet Container in order to handle error response (HttpServletResponse#sendError) received through default exception handling functionality of Spring MVC, fatal errors and the exceptions that are out of the boundary of Spring MVC.

- **web.xml**

  Add definitions in order to handle error response (HttpServletResponse#sendError) received through default exception handling functionality of Spring MVC.

```
<error-page>
    <!-- (1) -->
    <error-code>404</error-code>
    <!-- (2) -->
    <location>/WEB-INF/views/common/error/resourceNotFoundError.jsp</location>
</error-page>
```

Sr. No.	Description
(1)	Specify the **HTTP Response Code** to be handled. **[Location to be customized for each project]** For HTTP response code for which response is sent using the default exception handling function of Spring MVC, refer to *HTTP response code set by DefaultHandlerExceptionResolver*.
(2)	Specify the file name. It should be specified with the path from Web application root. In the above settings, "${WebAppRoot}/WEB-INF/views/common/error/resourceNotFoundError.jsp" will be the View file. **[Location to be customized for each project]**

Add definitions in order to handle fatal errors and exceptions that are out of the boundary of Spring MVC.

```
    <error-page>
        <!-- (3) -->
        <location>/WEB-INF/views/common/error/unhandledSystemError.html</location>
    </error-page>
```

Sr. No.	Description
(3)	Specify the file name. Specify with a path from web application root. In the above settings, "${WebAppRoot}/WEB-INF/views/common/error/unhandledSystemError.html" will be the View file. **[Location to be customized for each project]**

Note: About the path specified in location tag

If a fatal error occurs, there is high possibility of getting another error if the path of dynamic contents is specified.; hence in location tag, **it is recommended that you specify a path of static contents such as HTML** and not dynamic contents such as JSP.

Note: If untraceable error occurs during development

If an unexpected error response (HttpServletResponse#sendError) occurs after carrying out the above settings, there may be cases wherein it cannot be determined what kind of error response occured.

Error screen specified in location tag is displayed; however if the cause of error cannot be identified from logs, it is possible to verify the error response (HTTP response code) on screen by commenting out the above settings.

When it is necessary to individually handle the exceptions that are out of the boundary of Spring MVC, definition of each exception should be added.

```
    <error-page>
        <!-- (4) -->
        <exception-type>java.io.IOException</exception-type>
        <!-- (5) -->
        <location>/WEB-INF/views/common/error/systemError.jsp</location>
    </error-page>
```

Sr. No.	Description
(4)	Specify the **Exception Class Name (FQCN)** to be handled.
(5)	Specify the file name. Specify it using a path from web application root. In above case, "${WebAppRoot}/WEB-INF/views/common/error/systemError.jsp" will be the View file. **[Location to be customized for each project]**

Coding Points (Service)

The coding points in Service when handling the exceptions are given below.

1. *Generating Business Exception*
2. *Generating System Exception*
3. *Catch the exception to continue the execution*

Generating Business Exception

The method of generating Business Exception is given below.

Note: Notes related to the method of generating business exception

- It is recommended that you generate the business exception by detecting violation of business rules in logic.
- When it is required by the API specification of underlying base framework or existing layer of application, that violation of business rule be notified through an exception, then it is OK to catch the exception and throw it as business exception.

  Use of an exception to control the processing flow lowers the readability of overall logic and thus may reduce maintainability.

Generate business exceptions by detecting violation of business rules in logic.

Warning:

- It is assumed that business exception is generated in Service by default. In *AOP settings*, log of business exception occurred in class with `@Service` annotation, is being output. Business exceptions should not be logged in Controller etc. This rule can be changed if needed in the project.

- 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)
        }
        ...
```

Sr. No.	Description
(1)	Check whether there is any violation of a business rule.
(2)	If there is a violation, generate ResultMessages. In the above example, ResultMessages of error level are being generated. For the details on method of generating ResultMessages, refer to *Message Management*.
(3)	Add ResultMessage to ResultMessages. Specify message ID as 1st argument (mandatory) and value to be inserted in message as 2nd argument (optional). The value to be inserted in message is a variable-length parameter; hence multiple values can be specified.
(4)	Specify ResultMessages and generate BusinessException.

Tip: For the purpose of explanation, `xxxService.java` logic is written in steps (2)-(4) as shown above, however it can also be written in a single step.

```
    throw new BusinessException(ResultMessages.error().add(
            "e.ad.od.5001", stockQuantity));
```

- xxx.properties

  Add the below settings to the properties file.

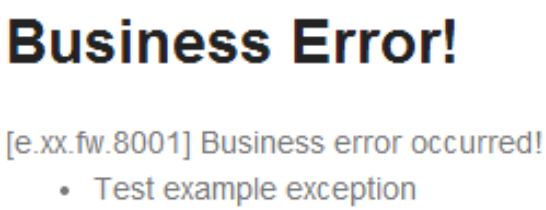
```
    e.ad.od.5001 = Order number is higher than the stock quantity={0}. Change the order number.
```

An application log as shown below is output.

```
date:2013-09-17 22:25:55      thread:tomcat-http--8   X-Track:6cfb0b378c124b918e40ac0c32a1fac7
org.terasoluna.gfw.common.exception.BusinessException: ResultMessages [type=error, list=[ResultMes
```

```
// stackTrace omitted
...

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
```

Displayed screen



Warning: It is recommended that you handle business exception in Controller and display a message on each business screen. The above example illustrates a screen which is displayed when the exception is not handled in Controller.

Catch an exception to generate a business exception

```
    try {
        order(orderQuantity, itemId );
    } catch (StockNotEnoughException e) {                    // (1)
        throw new BusinessException(ResultMessages.error().add(
                "e.ad.od.5001", e.getStockQuantity()), e); // (2)
    }
```

Sr. No.	Description
(1)	Catch the exception that occurs when a business rule is violated.
(2)	Specify ResultMessages and **Cause Exception (e)** to generate BusinessException.

Generating System Exception

The method of generating SystemException is given below.

Generate a system exception by detecting a system error in logic.

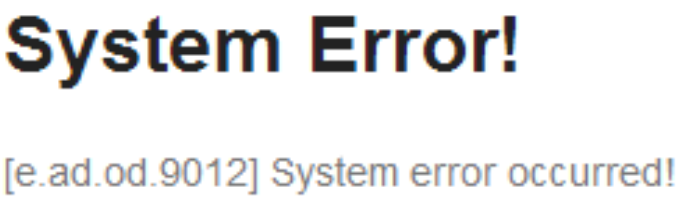
```
    if (itemEntity == null) {                                          // (1)
        throw new SystemException("e.ad.od.9012",
            "not found item entity. item code [" + itemId + "]."); // (2)
    }
```

Sr. No.	Description
(1)	Check whether the system is in normal state. In this example, it is checked whether the requested product code (itemId) exists in the product master (Item Master). If it does not exist in the product master, it would be considered that a resource which should have been available in the system, does not exist and hence it is treated as system error.
(2)	If the system is in an abnormal state, specify the exception code (message ID) as 1st argument. Specify exception message as 2nd argument to generate SystemException. In the above example, the value of variable “itemId” is inserted in message text.

Application log as shown below is output.

```
    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
    ...
    // stackTrace omitted
```

Displayed screen



Note: It is desirable to have a common system error screen rather than creating multiple screens for system errors.

The screen mentioned in this guideline displays a (business-wise) message ID for system errors and has a fixed message. This is because there is no need to inform the details of error to the operator and it is sufficient to only convey that the system error has occured. Therefore, in order to enhance the response for inquiry against system errors, the Message ID which acts as a key for the message text is displayed on the screen, in order to make the analysis easier for the development side. Displayed error screens should be designed in accordance with the UI standards of each project.

Catch an exception to generate system exception.

```
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)
}
```

Sr. No.	Description
(1)	Catch the checked exception classified as system error.
(2)	Specify the exception code (message ID), message, **Cause Exception (e)** to generate SystemException.

Catch the exception to continue the execution

When it is necessary to catch the exception to continue the execution, the exception should be logged before continuing the execution.

When fetching customer interaction history from external system fails, the process of fetching information other than customer history can still be continued. This is illustrated in the following example.

In this example, although fetching of customer history fails, business process does not have to be stopped and hence the execution continues.

```
    @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);

    // ...
```

Sr. No.	Description
(1)	Inject the object of `org.terasoluna.gfw.common.exception.ExceptionL` provided by the common library for log output.
(2)	Catch the exception to be handled.
(3)	Output the handled exception to log. In the example, log method is being called; however if the output level is known in advance and if there is no possibility of any change in output level, it is ok to call info, warn, error methods directly.
(4)	Continue the execution just by outputting the log in (3).

An application log as shown below is output.

```
    date:2013-09-19 21:31:47      thread:tomcat-http--3   X-Track:df5271ece2304b12a2c59ff4948063
    org.springframework.web.client.RestClientException: Test example exception
    ...
    // stackTrace omitted
```

> **Warning:** When log() is used in `exceptionLogger`, since it will be output at error level; by default, it will be output in monitoring log also.

```
date:2013-09-19 21:31:47  X-Track:df5271ece2304b12a2c59ff494806397        level:ERROR     me
```

As shown in this example, if there is no problem in continuing the execution, and if monitoring log is being monitored through application monitoring, it should be set to a level such that it will not get monitored at log output level or defined such that it does not get monitored from the log content (log message).

```
} catch (RestClientException e) {
    exceptionLogger.info(e);
}
```

As per default settings, monitoring log other than error level will not be output. It is output in application log as follows:

```
date:2013-09-19 22:17:53    thread:tomcat-http--3   X-Track:999725b111b4445b8d10b4ea44639c61
org.springframework.web.client.RestClientException: Test example exception
```

Coding Points (Controller)

The coding points in Controller while handling the exceptions are given below.

1. *Method to handle exceptions at request level*
2. *Method to handle exception at use case level*

Method to handle exceptions at request level

Handle the exception at request level and set the message related information to Model.
Then, by calling the method for displaying the View, generate the model required by the View and determine the View name.

```
@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

    }
```

Sr. No.	Description
(1)	As of the parameters of the method, define Model in argument as an object which will be used to link the error information with View.
(2)	Exceptions which need to be handled should be caught in application code.
(3)	Add ResultMessages object to Model.
(4)	Call the method for displaying the View at the time of error. Fetch the model and View name required for View display and then return the View name to be displayed.

Method to handle exception at use case level

Handle the exception at use case level and generate ModelMap (ExtendedModelMap) wherein message related information etc. is stored.

Then, by calling the method for displaying the View, generate the model required by the View and determine the View name.

```
    @ExceptionHandler(BusinessException.class) // (1)
    @ResponseStatus(HttpStatus.CONFLICT) // (2)
    public ModelAndView handleBusinessException(BusinessException e) {
        ExtendedModelMap modelMap = new ExtendedModelMap();                    // (3)
        modelMap.addAttribute(e.getResultMessages());                          // (4)
        String viewName = top(modelMap);                                       // (5)
        return new ModelAndView(viewName, modelMap);                           // (6)
    }
```

Sr. No.	Description
(1)	The exception class which need to be handled should be specified in the value attribute of `@ExceptionHandler` annotation. You can also specify multiple exceptions which are in the scope of handling.
(2)	Specify the HTTP status code to be returned to value attribute of `@ResponseStatus` annotation. In the example, "409: Conflict" is specified.
(3)	Generate ExtendedModelMap as an object to link the error information and model information with View.
(4)	Add ResultMessages object to ExtendedModelMap.
(5)	Call the method to display the View at the time of error and fetch model and View name necessary for View display.
(6)	Generate ModelAndView wherein View name and Model acquired in steps (3)-(5) are stored and then return the same.

Coding points (JSP)

The coding points in JSP while handling the exceptions are given below.

1. *Method to display messages on screen using MessagesPanelTag*

2. *Method to display system exception code on screen*

Method to display messages on screen using MessagesPanelTag

The example below illustrates implementation at the time of outputting ResultMessages to an arbitrary location.

```
<t:messagesPanel /> <!-- (1) -->
```

Sr. No.	Description
(1)	<t:messagesPanel> tag should be specified at a location where the message is to be output. For details on usage of <t:messagesPanel> tag, refer to *Message Management*.

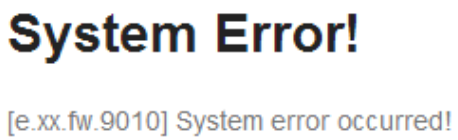
Method to display system exception code on screen

The example below illustrates implementation at the time of displaying exception code (message ID) and fixed message at an arbitrary location.

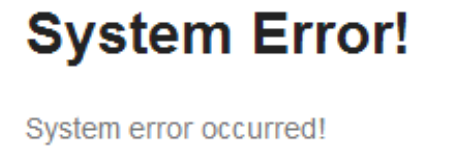
```
<p>
    <c:if test="${!empty exceptionCode}">  <!-- (1) -->
        [${f:h(exceptionCode)}]            <!-- (2) -->
    </c:if>
    <spring:message code="e.cm.fw.9999" /> <!-- (3) -->
</p>
```

Sr. No.	Description
(1)	Check whether exception code (message ID) exists. Perform existence check when the exception code (message ID) is enclosed with symbols as shown in the above example.
(2)	Output exception code (message ID).
(3)	Output fixed message acquired from message definition.

- Output Screen (With exceptionCode)



- Output Screen (Without exceptionCode)



Note: Messages to be output at the time of system exception

- When a system exception occurs, it is recommended that you display a message which would only convey that a system exception has occurred, without outputting a detailed message from which cause of error can be identified or guessed.
- On displaying a detailed message from which cause of error can be identified or guessed, the vulnerabilities of system are likely to get exposed.

Note: Exception code (message ID)

- When a system exception occurs, it is desirable to output an exception code (message ID) instead of a detailed message.
- By outputting the exception code (message ID), it is possible for development team to respond quickly to the inquiries from system user.
- Only a system administrator can identify the cause of error from exception code (message ID); hence the risk of exposing the vulnerabilities of system is lowered.

5.7.4 How to use (Ajax)

For the exception handling of Ajax, refer to *[coming soon] Ajax*.

5.7.5 Appendix

1. *Exception handling classes provided by the common library*
2. *About SystemExceptionResolver settings*
3. *HTTP response code set by DefaultHandlerExceptionResolver*

Exception handling classes provided by the common library

In addition to the classes provided by Spring MVC, the classes for carrying out exception handling are being provided by the common library.

The roles of classes are as follows:

Table.5.5 Table - Classes under org.terasoluna.gfw.common.exception package

Sr. No.	Class	Role
(1)	ExceptionCode Resolver	An interface for resolving exception code (message ID) of exception class. An exception code is used for identifying the exception type and is expected to be output to system error screen and log. It is referenced from `ExceptionLogger`, `SystemExceptionResolver`.
(2)	SimpleMapping ExceptionCode Resolver	Implementation class of `ExceptionCodeResolver` contains the mapping of exception class name and exception code, through which the exception code is resolved. The name of exception class need not be FQCN, it can be a part of FQCN or parent class name. **Warning:** • Note that when a part of FQCN is specified, it may be matched with classes which were not assumed. • Note that when the name of parent class is specified, all the child classes will also be matched.
(3)	enums. ExceptionLevel	enum indicating exception level corresponding to exception class. INFO, WARN, ERROR are defined.

Table.5.6 Table - Classes under org.terasoluna.gfw.web.exception package

Sr. No.	Class	Role
(13)	SystemException Resolver	This is a class for handling the exceptions that will not be handled by `HandlerExceptionResolver`, which is registered automatically when `<mvc:annotation-driven>` is specified. It inherits `SimpleMappingExceptionResolver` provided by Spring MVC and adds the functionality of referencing ResultMessages of exception code from View.
(14)	HandlerException ResolverLogging Interceptor	This is an Interceptor class to output the log of exceptions handled by `HandlerExceptionResolver`. In this Interceptor class, the output level of log is switched based on the classification of HTTP response code resolved by `HandlerExceptionResolver`. 1. When it is “100-399”, log is output at INFO level. 2. When it is “400-499”, log is output at WARN level. 3. When it is “500-”, log is output at ERROR level. 4. When it is “-99”, log is not output. By using this Interceptor, it is possible to output the log of all exceptions which are within the boundary of Spring MVC.
		Log is output using `ExceptionLogger`.

About SystemExceptionResolver settings

This section describes the settings which are not explained above. The settings should be performed depending on the requirements.

Table.5.7 List of settings not explained above

Sr. No.	Field Name	Property Name	Description	Default Value
(1)	Attribute name of result message	resultMessagesAttribute	Specify the attribute name (String) used for setting message of businessException to Model. This attribute name is used for accessing result message in View(JSP).	resultMessages
(2)	Attribute name of exception code (message ID)	exceptionCode Attribute	Specify the attribute name (String) used for setting the exception code (message ID) to HttpServletRequest. This attribute name is used for accessing the exception code (message ID) in View(JSP).	exceptionCode
(3)	Header name of exception code (message ID)	exceptionCode Header	Specify the header name (String) used for setting the exception code (message ID) to response header of HttpServletResponse.	X-Exception-Code
(4)	Attribute name of exception object	exceptionAttribute	Specify the attribute name (String) used for setting the handled exception object to Model. This attribute name is used for accessing the exception object	exception

(1)-(3) are the settings of `org.terasoluna.gfw.web.exception.SystemExceptionResolver`.
(4) is the setting of `org.springframework.web.servlet.handler.SimpleMappingExceptionResolver`.
(5)-(7) are the settings of `org.springframework.web.servlet.handler.AbstractHandlerExceptionResolver`.

Attribute name of result message

If the message set by handling the exception using SystemExceptionResolver and the message set by handling the exception in application code, both are to be output in separate messagesPanel tags in View(JSP), then specify an attribute name exclusive to SystemExceptionResolver.
The example below illustrates settings and implementation when changing the default value to "resultMessagesForExceptionResolver".

- **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) -->
```

Sr. No.	Description
(1)	Specify "resultMessagesForExceptionResolver" in result message attribute name (resultMessagesAttribute).
(2)	Specify attribute name that was set in SystemExceptionResolver, in message attribute name (messagesAttributeName).

Attribute name of exception code (message ID)

When default attribute name is being used in the application code, a different value should be set in order to avoid duplication. When there is no duplication, there is no need to change the default value.

The example below illustrates settings and implementation when changing the default value to "exceptionCodeForExceptionResolver".

- **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>
```

Sr. No.	Description
(1)	Specify “exceptionCodeForExceptionResolver” in attribute name (exceptionCodeAttribute) of exception code (message ID).
(2)	Specify the value (exceptionCodeForExceptionResolver) set in SystemExceptionResolver as a variable name (for Empty check) to be tested.
(3)	Specify the value (exceptionCodeForExceptionResolver) set in SystemExceptionResolver as a variable name to be output.

Header name of exception code (message ID)

When default header name is being used, set a different value in order to avoid duplication. When there is no duplication, there is no need to change the default value.

The example below illustrates settings and implementation when changing the default value to “X-Exception-Code-ForExceptionResolver”.

- **spring-mvc.xml**

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">

    <!-- omitted -->

    <property name="exceptionCodeHeader" value="X-Exception-Code-ForExceptionResolver" /> <!

    <!-- omitted -->
</bean>
```

Sr. No.	Description
(1)	Specify “X-Exception-Code-ForExceptionResolver” in the header name (exceptionCodeHeader) of the exception code (message ID).

Attribute name of exception object

When default attribute name is being used in the application code, set a different value in order to avoid duplication. When there is no duplication, there is no need to change the default value.

The example below illustrates settings and implementation when changing the default value to “exceptionForExceptionResolver”.

- **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) -->
```

Sr. No.	Description
(1)	Specify “exceptionForExceptionResolver” in attribute name (exceptionAttribute) of exception object.
(2)	Specify the value (exceptionForExceptionResolver) set in SystemExceptionResolver as a variable name for fetching message from exception object.

Cache control flag of HTTP response

When header for cache control is to be added to HTTP response, specify true: Yes.

- **spring-mvc.xml**

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">

    <!-- omitted -->

    <property name="preventResponseCaching" value="true" /> <!-- (1) -->

    <!-- omitted -->
</bean>
```

Sr. No.	Description
(1)	Set the cache control flag (preventResponseCaching) of HTTP response to true: Yes.

Note: HTTP response header when Yes is specified

If cache control flag of HTTP response is set to Yes, the following HTTP response header is output.

Cache-Control:no-store

Cache-Control:no-cache

Expires:Thu, 01 Jan 1970 00:00:00 GMT

Pragma:no-cache

About HandlerExceptionResolverLoggingInterceptor settings

This section describes the settings which are not explained above. The settings should be performed depending on the requirements.

Table.5.8 List of settings not described above

Sr. No.	Field Name	Property Name	Description	Default Value
(1)	List of exception classes to be excluded from scope of logging	ignoreExceptions	Amongst the exceptions handled by `HandlerExceptionResolver`, the exception classes which are not to be logged should be specified in list format. When an exception of the specified exception class and sub class occurs, the same is not logged in this class. Only the exceptions which are logged at a different location (using different mechanism) should be specified here.	`ResultMessagesNotific` Exceptions of `ResultMessagesNotific` and its sub classes are logged by `ResultMessagesLogging` hence, as default settings, they are being excluded.

List of exception classes to be excluded from scope of logging

The settings when exception classes provided in the project are to be excluded from the scope of logging, are as follows:

- **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>
```

Sr. No.	Description
(1)	`ResultMessagesNotificationException` specified in the default settings of common library, should be specified under exception to be excluded.
(2)	Specify the exception classes created in the project.

The settings when all the exception classes are to be logged are as follows:

- **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>
```

Sr. No.	Description
(3)	Specify `null` in ignoreExceptions properties. If `null` is specified, all the exception classes will become target for logging.

HTTP response code set by DefaultHandlerExceptionResolver

The mapping between framework exceptions handled using DefaultHandlerExceptionResolver and HTTP status code is shown below.

Sr. No.	Handled framework exceptions	HTTP Status Code
(1)	org.springframework.web.servlet.mvc.multiaction.NoSuchRequestHandlingMethodExcept	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 [coming soon] Session Management

Coming soon ...

5.9 Message Management

5.9.1 Overview

A message consists of fixed text displayed on screens or reports, or dynamic text displayed depending on screen operations performed by user.

It is recommended to define a message in as much details as possible.

> **Warning:** In following cases, there is a risk of inability to identify error cause during the production phase or during the testing just before entering into production phase (however, such risks may not surface during the development phase).
>
> - When only one error message is defined
> - When only two types of error messages ("Important" and "Warning") are defined
>
> Thus, if messages are changed when the number of developers in the team is less, the cost to modify these messages would increase as the development progresses. It is, therefore, recommended to define the messages in advance at a detailed level.

Types of messages

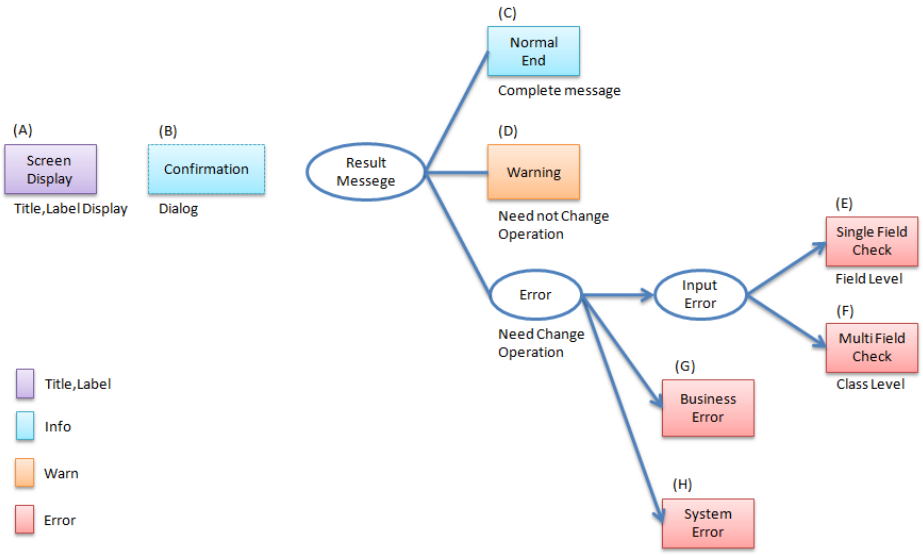
Messages should be classified into following 3 types depending on the message contents.

When defining any message, it important to note its type.

Message type	Category	Overview
info	Information message	This message is displayed when a process is executed normally by the user.
warn	Warning message	This message is displayed to indicate that there are some problems in the execution; however the process can be continued. (Example: Notification indicating that the password is about to expire)
error	Input error message	This message is displayed on input screen when value entered by the user is invalid.
	Business error message	This message is displayed to indicate error in the business logic
	System error message	This message is displayed when system errors (database connection failure etc.) occur and recovery is not possible by user operations.

Types of messages depending on patterns

Message output patterns are shown below.



Message patterns, message display contents and the message type are shown below.

Symbol	Pattern	Display contents	Message type	Example
(A)	Title	Screen title	-	• Employee Registration screen
	Label	Screen, report field name Table field name Comment	-	• User name • Password
(B)	Dialog	Confirmation message	info	• Are you sure you want to register? • Are you sure you want to delete?
(C)	Result message	Successful completion	info	• Registered. • Deleted.
(D)		Warning	warn	• Password is about to expire. Please change the password. • Server is busy. Please try again later.
(E)		Single field validation error	error	• “User name” is mandatory. • Please enter “Name” within 20 characters. • Please enter the “Amount” in number.
(F)		Correlation check error	error	• “Password” and “Confirm Password” do not match.
(G)		Business error	error	• Failed to cancel the reservation as cancellation period has elapsed. • Failed to register as number of allowed registrations exceeded.
(H)		System error	error	• XXXSystem is blocked, please try again later. • Timeout has occurred. • System error.

Message ID

For effective message management, adding an ID to the message is recommended.

The advantages of adding an ID are as follows:

- To change the message without modifying the source code.
- To be able to identify the message output location easily
- To support internationalization

From maintenance perspective, it is strongly recommended that you define the message IDs by creating and standardizing the rules.

See the example below for Message ID rules for each message pattern.

Refer to these rules when message ID rules are not defined in a development project.

Title

The method of defining message ID to be used in screen title is described below.

- Format

Prefix	Delimiter	Business process name	Delimiter	Screen name
title	.	nnn*	.	nnn*

- Description

Field	Position	Contents	Remarks
Prefix	1st - 5th digit (5 digits)	“title” (fixed)	
Business process name	Variable length：Optional	Directory under prefix of viewResolver defined in spring-mvc.xml (parent directory of JSP)	
Screen name	Variable length：Optional	JSP name	“aaa” when file name is “aaa.jsp”

- Example

```
In case of # "/WEB-INF/views/admin/top.jsp"
title.admin.top=Admin Top
In case of # "/WEB-INF/views/staff/createForm.jsp"
title.staff.createForm=Staff Register Input
```

Tip: This example is valid when using Tiles. For details, refer to *[coming soon] Screen Layout using Tiles*. When not using Tiles, follow the *Labels* rules explained later.

Labels

The method of defining message ID to be used in screen label and fixed text of reports is described below.

- Format

Prefix	Delimiter	Project code	Delimiter	Business process name	Delimiter	Field name
label	.	xx	.	nnn*	.	nnn*

- Description

Field	Position	Contents	Remarks
Prefix	1st - 5th digit (5 digits)	“label” (Fixed)	
Project code	7th - 8th digit (2 digits)	Enter 2 alphabets of project name	
Business process name	Variable length：Optional		
Field name	Variable length：Optional	Label name, Caption	

- Example

```
# Form field name on Staff Registration screen
# Project code=em (Event Management System)
label.em.staff.staffName=Staff name
# In case of a caption to be displayed on Tour Search screen
# Project code=tr (Tour Reservation System)
label.tr.tourSearch.tourSearchMessage=You can search tours with the specified conditions.
```

In case of multiple projects, define a project code to avoid duplication of message ID. Even if there is a single project, it is recommended to define a project code for future enhancements.

Result messages

Messages commonly used in business processes To avoid duplication of messages, the messages which are common in multiple business processes are explained below.

- Format

Message type	Delimiter	Project code	Delimiter	Common message code	Delimiter	Error level	Sr. No
x	.	xx	.	fw	.	9	999

- Description

Field	Position	Contents	Remarks
Message type	1st digit (1 digit)	info : i warn : w error : e	
Project code	3rd - 4th digit (2 digits)	Enter 2 alphabets of project name	
Common message code	6th - 7th digit (2 digits)	“fw” (fixed)	
Error level	9th digit (1 digit)	0-1 : Normal message 2-4 : Business error (semi-normal message) 5-7 : Input validation error 8 : Business error (error) 9 : System error	
Sr. No.	10th -12th digit (3 digits)	Use as per serial number (000-999)	Even if the message is deleted, serial number field should be blank and it should not be deleted.

- Example

```
# When registration is successful (Normal message)
i.ex.fw.0001=Registered successfully.
# Insufficient server resources
w.ex.fw.9002=Server busy. Please, try again.
# When system error occurs (System error)
e.ex.fw.9001=A system error has occurred.
```

Messages used individually in each business process The messages used individually in each business process are explained below.

- Format

Message type	Delimiter	Project code	Delimiter	Business process message code	Delimiter	Error level	Sr. No
x	.	xx	.	xx	.	9	999

- Description

Field	Position	Contents	Remarks
Message type	1st digit (1 digit)	info : i warn : w error : e	
Project code	3rd -4th digit (2 digits)	Enter 2 alphabets of project name	
Business process message code	6th -7th digit (2 digits)	2 characters defined for each business process such as Business ID	
Error level	9th digit (1 digit)	0-1 : Normal message 2-4 : Business error (semi-normal message) 5-7 : Input validation error 8 : Business error (error) 9 : System error	
Sr. No.	10th -12th digit (3 digits)	Use as per serial number (000-999)	Even if the message is deleted, serial number field should be blank and it should not be deleted.

- Example

```
# When file upload is successful.
i.ex.an.0001={0} upload completed.
# When the recommended password change interval has passed.
w.ex.an.2001=The recommended change interval has passed password. Please change your pass
# When file size exceeds the limit.
e.ex.an.8001=Cannot upload, Because the file size must be less than {0}MB.
# When there is inconsistency in data.
e.ex.an.9001=There are inconsistencies in the data.
```

Input validation error message

For the messages to be displayed in case of input validation error, refer to *Definition of error messages*.

Note: Basic policies related to output location of input validation error are as follows:

- Single field input validation error messages should be displayed next to the target field so that it can be identified easily.
- Correlation input validation error messages should be displayed collectively on the top of the page .
- When it is difficult to display the single field validation message next to the target field, it should be displayed on the top of the page.

  In that case, field name should be included in the message.

5.9.2 How to use

Display of messages set in properties file

Settings at the time of using properties

Define implementation class of `org.springframework.context.MessageSource` which is used for performing message management.

- 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>
```

Sr. No.	Description
(1)	Definition of `MessageSource`. Here, use `ResourceBundleMessageSource`.
(2)	Define the base name of message property to be used. Specify it with relative class path. In this example, read "src/main/resources/i18n/application-messages.properties".

Display of messages set in properties

- application-messages.properties

  See the example below for defining the messages in `application-messages.properties`.

```
label.aa.bb.year=Year
label.aa.bb.month=Month
label.aa.bb.day=Day
```

Note: Earlier, it was necessary to convert the characters (such as Japanese characters etc.) that cannot be expressed into "ISO-8859-1" with the help of `native2ascii` command. However, from JDK version 6 onwards, it has become possible to specify the character encoding; hence character conversion is no longer needed. By setting the character encoding to UTF-8, Japanese characters etc. can be used directly in properties file.

  - application-messages.properties

```
label.aa.bb.year= Year
label.aa.bb.month= Month
label.aa.bb.day= Day
```

  In such a case, it is necessary to specify the character encoding that can also be read in `ResourceBundleMessageSource`.

  - 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>
```

ISO-8859-1 is used by default; hence when describing the Japanese characters directly in properties file, make sure that the character encoding is set as value of `defaultEncoding` property.

- JSP

  Messages set above can be displayed using `<spring:message>` tag in JSP. For using it, settings mentioned in *Creating common JSP for include* must be done.

```
<spring:message code="label.aa.bb.year" />
<spring:message code="label.aa.bb.month" />
<spring:message code="label.aa.bb.day" />
```

When used with form label, it can be used as follows:

```
<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>
```

It is displayed in browser as follows:

Tip: When supporting internationalization,

```
src/main/resources/i18n
                    ├ application-messages.properties (English message)
                    ├ application-messages_fr.properties (French message)
                    ├ ...
                    └ application-messages_ja.properties (Japanese message)
```

properties file should be created for each language as shown above. For details, refer to *[coming soon] Internationalization* .

Display of result messages

`org.terasoluna.gfw.common.message.ResultMessages` and `org.terasoluna.gfw.common.message.Res` are provided in common library, as classes storing the result messages which indicate success or failure of process at server side.

Class name	Description
`ResultMessages`	Class having list of result messages and message type. List of Result messages is expressed in terms of `List<ResultMessage>` and message type is expressed in terms of `org.terasoluna.gfw.common.message.ResultMessageType` interface.
`ResultMessage`	Class having result message ID or message text.

`<t:messagesPanel>` tag is also provided as JSP tag library for displaying this result message in JSP.

Using basic result messages

The way of creating `ResultMessages` in Controller, passing them to screen and displaying the result messages using `<t:messagesPanel>` tag in JSP, is displayed below.

- Controller class

  The methods of creating `ResultMessages` object and passing the messages to screen are given below. An example of *Messages used individually in each business process* should be defined in application-messages.properties.

```
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";
  }
}
```

Sr. No.	Description
(1)	Create `ResultMessages` wherein message type is “error” and set result messages wherein message ID is “e.ex.an.9001”. This process is same as follows: `ResultMessages.error().add(ResultMessage.fromCode("e.ex.an.9001"));` Since it is possible to skip the creation of `ResultMessage` object if message ID is specified, it is recommended to skip the same.
(2)	Add `ResultMessages` to Model. It is ok even if the attribute is not specified. (Attribute name is “resultMessages”)

- JSP

  Write WEB-INF/views/message/index.jsp as follows:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Result Message Example</title>
</head>
<body>
    <h1>Result Message</h1>
    <t:messagesPanel /><!-- (1) -->
</body>
```

```
</html>
```

Sr. No.	Description
(1)	<t:messagesPanel> tag is used with default settings. By default, "resultMessages" object is displayed. Therefore, attribute name need not be specified when ResultMessages is set in Model from Controller with default settings.

It is displayed in browser as follows:

HTML output by <t:messagesPanel> is shown below. (The format makes the explanation easier).

```
<div class="alert alert-error"><!-- (1) -->
  <ul><!-- (2) -->
    <li>There are inconsistencies in the data.</li><!-- (3) -->
  </ul>
</div>
```

Sr. No.	Description
(1)	"alert-error"class is assigned in accordance with the message type. "error error-[Message type]" is assigned to tag class by default.
(2)	Result message list is output using tag.
(3)	The message corresponding to message ID is resolved from MessageSource.

<t:messagesPanel> outputs only HTML with class; hence it is necessary to customize the look and feel using CSS as per the output class (explained later).

Note: Message text can be hard-coded such as ResultMessages.error().add(ResultMessage.fromText are inconsistencies in the data."));; however, to enhance maintainability, it is rec-

ommended to create `ResultMessage` object using message key, and to fetch the message text from properties file.

For inserting a value in message placeholder, set second or subsequent arguments of `add` method as follows:

```
ResultMessages messages = ResultMessages.error().add("e.ex.an.8001", 1024);
model.addAttribute(messages);
```

In such a case, the HTML shown below is output using `<t:messagesPanel />` tag.

```
<div class="alert alert-error">
  <ul>
    <li>Cannot upload, Because the file size must be less than 1,024MB.</li>
  </ul>
</div>
```

> **Warning: Points to be noted when inserting values in placeholder using terasoluna-gfw-web 1.0.0.RELEASE**
> When using terasoluna-gfw-web 1.0.0.RELEASE, **if the user entered value is inserted in the placeholder, there is a risk of XSS vulnerability.**If the user entered value is likely to include XSS vulnerable characters, then the value should not be inserted in the placeholder.
> When using terasoluna-gfw-web 1.0.1.RELEASE or higher version, XSS vulnerability does not occur even after inserting the user entered value in the placeholder.

Note: `ResourceBundleMessageSource` uses `java.text.MessageFormat` at the time of creating a message; hence `1024` is displayed as `1,024` with comma. When comma is not required, perform settings in properties file as shown below.

```
e.ex.an.8001=Cannot upload, Because the file size must be less than {0,number,#}MB.
```

For details, refer to Javadoc.

It is also possible to set multiple result messages as shown below.

```
ResultMessages messages = ResultMessages.error()
    .add("e.ex.an.9001")
    .add("e.ex.an.8001", 1024);
model.addAttribute(messages);
```

In such a case, HTML is output as follows (no need to change JSP).

```
<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>
```

In order to display info message, it is desirable to create `ResultMessages` object using `ResultMessages.info()` method as shown below.

```
ResultMessages messages = ResultMessages.info().add("i.ex.an.0001", "XXXX");
model.addAttribute(messages);
```

HTML shown below is output.

```
<div class="alert alert-info"><!-- (1) -->
  <ul>
    <li>XXXX upload completed.</li>
  </ul>
</div>
```

Sr. No.	Description
(1)	The output class name has changed to “alert alert-**info**” in accordance with the message type.

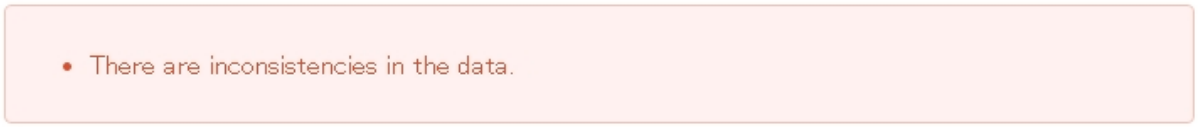
Fundamentally the following message types are created.

Message type	Creation of `ResultMessages` object	Default class name
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

CSS should be defined according to the message type. Example of applying CSS is given below.

```
.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);
}
```

- Example wherein `ResultMessages.error().add("e.ex.an.9001")` is output using `<t:messagesPanel />`



- Example wherein `ResultMessages.warn().add("w.ex.an.2001")` is output using `<t:messagesPanel />`

- Example wherein `ResultMessages.info().add("i.ex.an.0001", "XXXX")` is output using `<t:messagesPanel />`

Note: "success" and "danger" are provided to have diversity in style. In this guideline, success is synonymous with info and error is synonymous with danger.

Tip: Alerts component of Bootstrap 3.0.0 which is a CSS framework can be used with default settings of `<t:messagePanel />`.

Warning: In this example, message keys are hardcoded. However, in order to improve maintainability, it is recommended to define message keys in constant class.
Refer to *Auto-generation tool of message key constant class*.

Specifying attribute name of result messages

Attribute name can be omitted when adding `ResultMessages` to Model.

However, `ResultMessages` cannot represent more than one message type.

In order to **simultaneously** display the `ResultMessages` of different message types on 1 screen, it is necessary to specify the attribute name explicitly and set it in Model.

- Controller (Add to MessageController)

```
@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";
}
```

Sr. No.	Description
(1)	Add `ResultMessages` of "warn" message type to Model with attribute name "messages1".
(2)	Add `ResultMessages` of "info" message type to Model with attribute name "messages2".

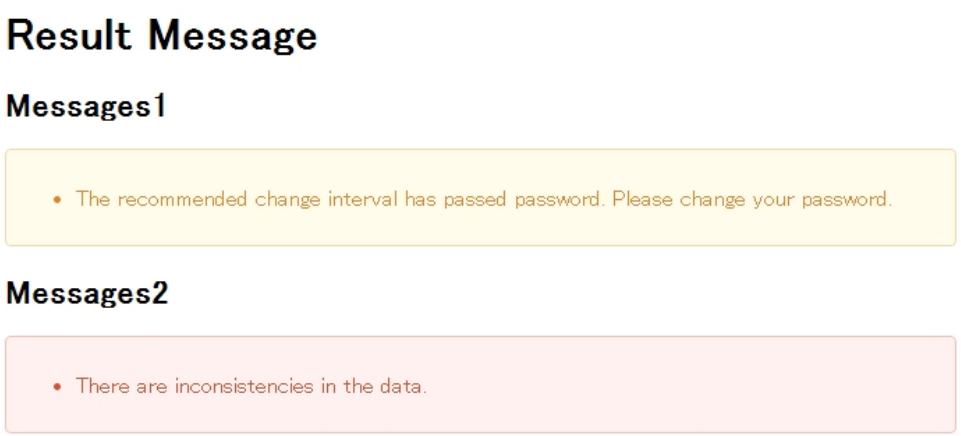
- JSP (WEB-INF/views/message/showMessages.jsp)

```
<!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>
```

Sr. No.	Description
(1)	Display `ResultMessages` having attribute name “messages1”.
(2)	Display `ResultMessages` having attribute name “messages2”.

It is displayed in browser as follows:



Displaying business exception messages

`org.terasoluna.gfw.common.exception.BusinessException` and `org.terasoluna.gfw.common.exception.ResourceNotFoundException` stores `ResultMessages` internally.

When displaying the business exception message, `BusinessException` wherein `ResultMessages` is set should be thrown in Service class.

Catch `BusinessException` in Controller class and add the result message fetched from the caught exception to Model.

- Service class

```
@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);
        }
    }

}
```

Sr. No.	Description
(1)	Create error message using `ResultMessages` and set in `BusinessException`.

- Controller class

```
@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
}
```

Sr. No.	Description
(1)	Fetch `ResultMessages` held by `BusinessException` and add to Model.

Normally, this method should be used to display error message instead of creating `ResultMessages` object in Controller.

5.9.3 How to extend

Creating independent message types

The method of creating independent message type is given below.

Normally, the available message types are sufficient. However, new message type may need to be added depending upon the CSS library. See the example below for adding the message type “notice”.

First, create independent message type class wherein `org.terasoluna.gfw.common.message.ResultMessageType` interface is implemented as follows:

```
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;
    }
}
```

Sr. No.	Description
(1)	Define Enum wherein `ResultMessageType` interface is implemented. A new message type can be created using constant class; however it is recommended to create it using Enum.
(2)	Return value of `getType` corresponds to class name of CSS which is output.

Create `ResultMessages` using this message type as mentioned below.

```
ResultMessages messages = new ResultMessages(ResultMessageTypes.NOTICE) // (1)
        .add("w.ex.an.2001");
model.addAttribute(messages);
```

Sr. No.	Description
(1)	Specify `ResultMessageType` in constructor of `ResultMessages`.

In such a case, HTML shown below is output in `<t:messagesPanel />`.

```
<div class="alert alert-notice">
  <ul>
    <li>The recommended change interval has passed password. Please change your password.</li>
  </ul>
</div>
```

Tip: For extension method, refer to `org.terasoluna.gfw.common.message.StandardResultMessageType`

5.9.4 Appendix

Changing attribute of <t:messagesPanel> tag

`<t:messagesPanel>` tag contains various attributes for changing the display format.

Table.5.9 <t:messagesPanel> Tag attribute list

Option	Contents	Default setting value
panelElement	Result message display panel elements	div
panelClassName	CSS class name of result message display panel.	alert
panelTypeClassPrefix	Prefix of CSS class name	alert-
messagesType	Message type. When this attribute is set, the set message type is given preference over the message type of `ResultMessages` object.	
outerElement	Outer tag of HTML configuring result messages list	ul
innerElement	Inner tag of HTML configuring result messages list	li
disableHtmlEscape	Flag for disabling HTML escaping. By setting the flag to `true`, HTML escaping is no longer performed for the message to be output. This attribute is used to create different message styles by inserting HTML into the message to be output. **When the flag is set to true, XSS vulnerable characters should not be included in the message.** This attribute can be used with terasoluna-gfw-web 1.0.1.RELEASE or higher version.	`false`

For example, following CSS is provided in CSS framework "BlueTrip".

```
.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;
}
```

To use this CSS, the message `<div class="error">...</div>` should be output.

In this case, `<t:messagesPanel>` tag can be used as follows (no need to modify the Controller):

```
<t:messagesPanel panelClassName="" panelTypeClassPrefix="" />
```

HTML shown below is output.

```
<div class="error">
  <ul>
    <li>There are inconsistencies in the data.</li>
  </ul>
</div>
```

It is displayed in browser as follows:

- There are inconsistencies in the data.

When you do not want to use `<ul>` tag to display the message list, it can be customized using `outerElement` attribute and `innerElement` attribute.

When the attributes are set as follows:

```
<t:messagesPanel outerElement="" innerElement="span" />
```

HTML shown below is output.

```
<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>
```

Set the CSS as follows:

```
.alert > span {
    display: block; /* (1) */
}
```

Sr. No.	Description
(1)	Set tag which is a child element of “alert” class to Block-level element.

It is displayed in browser as follows:

There are inconsistencies in the data.
Cannot upload, Because the file size must be less than 1,024MB.

When disableHtmlEscape attribute is set to `true`, the output will be as follows:

In the example below, font of a part of the message has been set to 16px Red.

- 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
```

- Output image

- Please confirm order content. If this orders submitted, cannot cancel.

When disableHtmlEscape attribute is `false`(default), the output will be as follows after HTML escaping.

- Please confirm order content. If this orders submitted, cannot cancel.

Display of result message wherein ResultMessages is not used

Apart from `ResultMessages` object, `<t:messagesPanel>` tag can also output the following objects.

- `java.lang.String`
- `java.lang.Exception`

- java.util.List

Normally <t:messagesPanel> tag is used to output the ResultMessages object; however it can also be used to display strings (error messages) set in the request scope by the framework.

For example, at the time of authentication error, Spring Security sets the exception class with attribute name "SPRING_SECURITY_LAST_EXCEPTION" in the request scope.

Perform the following settings if you want to output this exception message in <t:messagesPanel> tag similar to the result messages.

```
<!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>
```

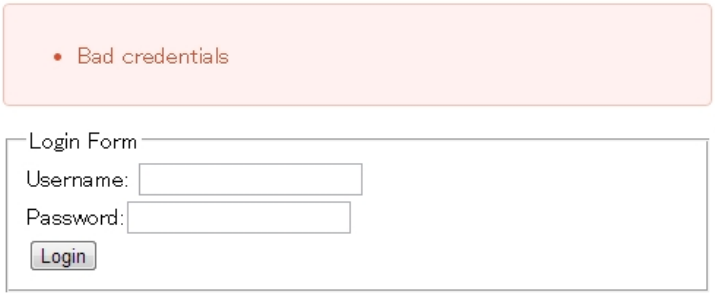
```
            <legend>Login Form</legend>
            <div>
                <label for="username">Username: </label><input
                    type="text" id="username" name="j_username">
            </div>
            <div>
                <label for="username">Password:</label><input
                    type="password" id="password" name="j_password">
            </div>
            <div>
                <input type="submit" value="Login" />
            </div>
        </fieldset>
    </form:form>
</body>
</html>
```

Sr. No.	Description
(1)	Re-define the CSS. It is strongly recommended to mention it in CSS file.
(2)	In `messagesAttributeName` attribute, specify the attribute name wherein `Exception` object is stored. Unlike the `ResultMessages` object, it does not contain the information of message type; hence it is necessary to explicitly specify the message type in `messagesType` attribute.

The HTML output in case of an authentication error will be,

```
<div class="alert alert-error"><ul><li>Bad credentials</li></ul></div>
```

and it will be displayed in the browser as follows:



Tip: For details on JSP for login, refer to *[coming soon] Authentication*.

Auto-generation tool of message key constant class

In all earlier examples, message keys were hard-coded strings; however it is recommended that you define the message keys in constant class.

This section introduces the program that auto-generates message key constant class from properties file and the corresponding usage method. You can customize and use them based on the requirements.

1. Creation of message key constant class

   First, create an empty message key constant class. Here, it is `com.example.common.message.MessageKeys`.

```
package com.example.common.message;

public class MessageKeys {

}
```

2. Creation of auto-generation class

   Next, create `MessageKeysGen` class in the same package as `MessageKeys` class and write the logic as follows:

```
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/applicatic
        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. Provision of message properties file

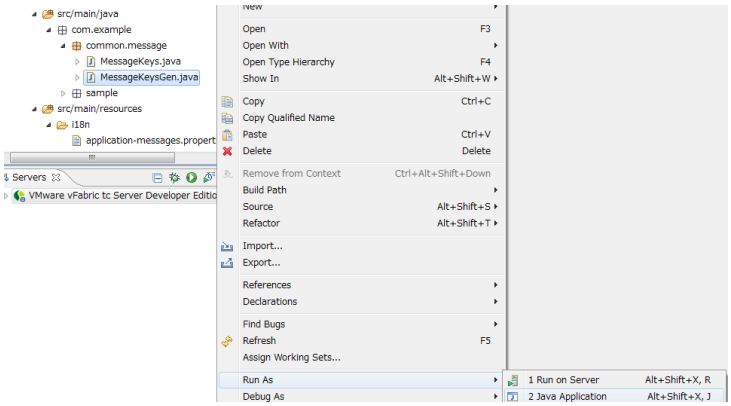
   Define the messages in src/main/resource/i18m/application-messages.properties. The settings are carried out as follows:

```
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. Execution of auto-generation class

   `MessageKeys` class is overwritten as follows:

```
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 [coming soon] Property Management

Coming soon ...

5.11 [coming soon] Pagination

Coming soon ...

5.12 [coming soon] Double Submit Protection

Coming soon ...

5.13 [coming soon] Internationalization

Coming soon ...

5.14 [coming soon] CodeList

Coming soon ...

5.15 [coming soon] Ajax

Coming soon ...

5.16 [coming soon] RESTful Web Service

Coming soon ...

5.17 [coming soon] File Upload

Coming soon ...

5.18 [coming soon] File Download

Coming soon ...

5.19 [coming soon] Screen Layout using Tiles

Coming soon ...

5.20 [coming soon] System Date

Coming soon ...

5.21 Utilities

5.21.1 [coming soon] Bean Mapping (Dozer)

Coming soon ...

5.21.2 Date Operations (Joda Time)

Overview

The API of `java.util.Date` , `java.util.Calender` class is poorly built to perform complex date calculations.

This guideline recommends the usage of Joda Time which provides quality replacement for the Java Date and Time classes.

In Joda Time, date is expressed using `org.joda.time.DateTime` object instead of `java.util.Date`. `org.joda.time.DateTime` object is immutable (the result of date related calculations, etc. is returned in a new object).

How to use

The usage of Joda Time, Joda Time JSP tags is explained below.

Fetching date

Fetching current time

As per the requirements, any of the following classes can be used.
`org.joda.time.DateTime` , `org.joda.time.LocalDate` , `org.joda.time.LocalTime` and `org.joda.time.DateMidnight`. The usage method is shown below.

1. Use `org.joda.time.DateTime` to fetch time up to milliseconds.

```
DateTime dateTime = new DateTime();
```

2. Use `org.joda.time.LocalDate` when only date, which does not include time and TimeZone, is required.

```
LocalDate localDate = new LocalDate();
```

3. Use `org.joda.time.LocalTime` when only time, which does not include date and TimeZone, is required.

```
LocalTime localTime = new LocalTime();
```

4. Use `org.joda.time.DateMidnight` to fetch the current date and time set at midnight.

```
DateMidnight dateMidnight = new DateMidnight();
```

Note: LocalDate and LocalTime do not have the TimeZone information, however, DateMidnight has TimeZone information.

Note: It is recommended that you use `org.terasoluna.gfw.common.date.DateFactory`, for fetching instance of DateTime, LocalDate and LocalTime at the time of fetching current time.

```
    DateTime dateTime = dataFactory.newDateTime();
```

Refer to *[coming soon] System Date* for using DateFactory.

LocalDate, LocalTime and DateMidnight can be generated in the following way.

```
    LocalDate localDate = dataFactory.newDateTime().toLocalDate();
    LocalTime localTime = dataFactory.newDateTime().toLocalTime();
    DateMidnight dateMidnight = dataFactory.newDateTime().toDateMidnight();
```

Fetching current time by specifying the time zone

`org.joda.time.DateTimeZone` class indicates time zone.

This class is used to fetch the current time for the specified time zone. The usage method is shown below.

```
DateTime dateTime = new DateTime(DateTimeZone.forID("Asia/Tokyo"));
```

`org.terasoluna.gfw.common.date.DateFactory`is used as follows:

```
// Fetching current system date using default TimeZone
DateTime dateTime = dataFactory.newDateTime();

// Changing to TimeZone of Tokyo
DateTime dateTimeTokyo = dateTime.withZone(DateTimeZone.forID("Asia/Tokyo"));
```

For the list of other available Time zone ID strings, refer to Available Time Zones.

Fetching the date and time by specifying Year Month Day Hour Minute and Second　Specific time can be specified in the constructor. An example is given below.

- Fetching DateTime by specifying time up to milliseconds

```
DateTime dateTime = new DateTime(year, month, day, hour, minute, second, millisecond);
```

- Fetching LocalDate by specifying Year Month and Day

```
LocalDate localDate = new LocalDate(year, month, day);
```

- Fetching LocalDate by specifying Hour Minute and Second

```
LocalTime localTime = new LocalTime(hour, minutes, seconds, milliseconds);
```

Fetching Year Month Day individually

DateTime provides a method to fetch Year and Month. The example is shown below.

```
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)
```

Sr. No.	Description
(1)	Get Year. In this example, `2013` is returned.
(2)	Get Month. In this example, `1` is returned.
(3)	Get Day. In this example, `10` is returned.
(4)	Get Day of Week. In this example, `4` is returned. The mapping of returned values and days of week is as follows: [1:Monday, 2:Tuesday:, 3:Wednesday, 4:Thursday, 5:Friday, 6:Saturday, 7:Sunday]
(5)	Get Hour. In this example, `2` is returned.
(6)	Get Minute. In this example, `30` is returned.
(7)	Get Second. In this example, `22` is returned.
(8)	Get Millisecond. In this example, `123` is returned.

Note: getDayOfMonth() starts with 1, differing from the specifications of `java.util.Calendar`.

Type conversion

Interoperability with java.util.Date

In DateTime, type conversion with `java.util.Date` can be easily performed.

```
Date date = new Date();

DateTime dateTime = new DateTime(date);  // (1)

Date convertDate = dateTime.toDate();  // (2)
```

Sr. No.	Description
(1)	Convert `java.util.Date` to DateTime by passing `java.util.Date` as an argument to DateTime constructor.
(2)	Convert DateTime to `java.util.Date` using DateTime#toDate method.

String format

```
DateTime dateTime = new DateTime();

dateTime.toString("yyyy-MM-dd HH:mm:ss");  // (1)
```

Sr. No.	Description
(1)	String of “yyyy-MM-dd HH:mm:ss” format is fetched. For values that can be specified as arguments of toString, refer to Input and Output .

Parsing from string

```
DateTime dateTime = DateTimeFormat.forPattern("yyyy-MM-dd").parseDateTime("2012-08-09");  // (1)
```

Sr. No.	Description
(1)	Convert “yyyy-MM-dd” string format to DateTime type. For values that can be specified as arguments of DateTimeFormat#forPattern, refer to Formatters.

Date operations

Date calculations

DateTime provides methods to perform date calculations. Examples are shown below.

```
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)
```

Sr. No.	Description
(1)	The value specified in argument of DateTime#minusDays is subtracted from the date. In this example, it becomes `2013-01-09T13:30:22.123Z`.
(2)	The value specified in argument of DateTime#plusDays is added to the date. In this example, it becomes `2013-01-11T13:30:22.123Z`.
(3)	The value specified in argument of DateTime#plusMonths is added to the number of months. In this example, it becomes `2013-04-10T13:30:22.123Z`.
(4)	The value specified in argument of DateTime#plusYears is added to the number of years. In this example, it becomes `2014-01-10T13:30:22.123Z`.

For methods other than those mentioned above, refer to DateTime JavaDoc .

Fetching first and last day of the month

The method of fetching the first and the last day of the month by considering the current date and time as base, is shown below.

Value of Hour/Minute/Second/Millisecond fetched in new DateTime() is kept as it is.

```
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)
```

Sr. No.	Description
(1)	Get Property object that holds the attribute values related to day of current month.
(2)	Get first day of the month by fetching the minimum value from Property object. In this example, it becomes `2013-01-01T13:30:22.123Z`.
(3)	Get last day of the month by fetching the maximum value from Property object. In this example, it becomes `2013-01-31T13:30:22.123Z`.

Fetching the first and the last day of the week

The method of fetching the first and the last day of the week by considering the current date and time as base, is shown below.

Value of Hour/Minute/Second/Millisecond fetched in new DateTime() is kept as it is.

```
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)
```

Sr. No.	Description
(1)	Get Property object that holds attribute values related to the day of current week.
(2)	Get first day of the week (Monday) by fetching the minimum value from Property object. In this example, it becomes `2013-01-07T13:30:22.123Z`.
(3)	Get last day of the week (Sunday) by fetching the maximum value from Property object. In this example, it becomes `2013-01-13T13:30:22.123Z`.

Comparison of date time　By comparing the date and time, it can be determined whether it is a past or future date.

```
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
```

Sr. No.	Description
(1)	`isAfter` method returns `true` when the target date and time is later than the date and time of argument.
(2)	`isBefore` method returns `true` when the target date and time is prior to the date and time of argument.
(3)	`isEqual` method returns `true` when the target date and time is same as the date and time of argument.

Fetching the duration

Joda-Time provides several classes related to duration. The following 2 classes are explained here.

- `org.joda.time.Interval`
- `org.joda.time.Period`

Interval　Class indicating the interval between two instances (DateTime）.

The following 4 are checked using the Interval class.

- Checking whether the interval includes the specified date or interval.
- Checking whether the 2 intervals are consecutive.
- Fetching the difference between 2 intervals in an interval
- Fetching the overlapping interval between 2 intervals

For implementation, refer to the following example.

```
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)
```

Sr. No.	Description
(1)	Check whether the interval includes the specified date and interval, using Interval#contains method. If the interval includes the specified date and interval, return “true”. If not, return “false”.
(2)	Check whether the 2 intervals are consecutive, using Interval#abuts method. If the 2 intervals are consecutive, return “true”. If not, return “false”.
(3)	Fetch the difference between 2 intervals in an interval, using Interval#gap method. In this example, the interval between “2013-08-16~2013-08-18” is fetched. When there is no difference between intervals, return null.
(4)	Fetch the overlapping interval between 2 intervals, using Interval#overlap method. In this example, the interval between “2013-08-15~2013-08-16” is fetched. When there is no overlapping interval, return null.

Intervals can be compared by converting into Period.

- Convert the intervals into Period when comparing them from abstract perspectives such as Month or Day.

```
// Convert to Period
interval1.toPeriod();
```

Period　Period is a class indicates duration in terms of Year, Month and Week.

For example, when Period of “1 month” is added to Instant（DateTime）indicating “March 1”, DateTime will be “April 1”.
The result of adding the Period of “1 month” to “March 1” and “April 1” is as shown below.

- Number of days is “31” when a Period of “1 month” is added to “March 1”.
- Number of days is “30” when a Period of “1 month” is added to “April 1”.

The result of adding a Period of “1 month” differs depending on the target DateTime.

Two different types of implementations have been provided for Period.

- Single field Period (Example：Type having values of single unit such as “1 Day” or “1 month”)
- Any field Period (Example：Type indicating the period and having values of multiple units such as “1 month 2 days 4 hours”)

For details, refer to Period.

JSP Tag Library

fmt:formatDate tag of JSTL handles objects of java.util.Date and java.util.TimeZone.
Use Joda tag library to handle DateTime, LocalDateTime, LocalDate, LocalTime and DateTimeZone objects of Joda-time.
The functionalities of JSP Tag Library are almost same as those of JSTL, hence it can be used easily if the person has knowledge of JSTL.

How to set　The following taglib definition is required to use the tag library.

```
<%@ taglib uri="http://www.joda.org/joda/time/tags" prefix="joda"%>
```

joda:format tag　joda:format tag is used to format the objects of DateTime, LocalDateTime, LocalDate and LocalTime.

```
<% 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" />
```

Output result

Using pattern="yyyyMMdd" to format the current system date
20131025
Using style="SM" to format the current system date
10/25/13 1:02:32 PM

The list of attributes of joda:format tag is as follows:

Table.5.10 Attribute information

No.	Attributes	Description
1.	value	Set the instance of ReadableInstant or ReadablePartial.
2.	var	Variable name
3.	scope	Scope of variables
4.	locale	Locale information
5.	style	Style information for doing formatting (2 digits. Set the style for date and time. Values that can be entered are S=Short, M=Medium, L=Long, F=Full, -=None)
6.	pattern	Pattern for doing formatting (yyyyMMdd etc.). For patterns that can be entered, refer to Input and Output.
7.	dateTimeZone	Time zone

For other Joda-Time tags, refer to Joda Time JSP tags User guide.

Note: When the date and time is displayed by specifying style attributes, the displayed contents differ depending on browser locale. Locale of the format displayed in the above style attribute is “en”.

Example (display of calendar)

Using Spring MVC, sample for displaying a month wise calender, is shown below.

Process name	URL	Processing method
Display of current month's calendar	/calendar	today
Display of specified month's calendar	/calendar/month?year=yyyy&month=m	month

The controller is implemented as follows:

```
@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";
    }
}
```

The `CalendarOutput` class mentioned below is JavaBean having the consolidated information to be output on the screen.

```
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
}
```

> **Warning:** For the sake of simplicity, this sample code includes all the logic in the processing method of Controller, but in real scenario, this logic should be delegated to Helper classes to improve maintainability.

In JSP(calendar.jsp), it is output as follows:

```
<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>
```

Access {contextPath}/calendar to display the calendar below (showing result of November 2012).

Access {contextPath}/calendar/month?year=2012&month=12 to display the calendar below.

← 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						

6

Security for TERASOLUNA Global Framework

6.1 [coming soon] Spring Security Overview

Coming soon ...

6.2 [coming soon] Spring Security Tutorial

Coming soon ...

6.3 [coming soon] Authentication

Coming soon ...

6.4 [coming soon] Password Hashing

Coming soon ...

6.5 [coming soon] Authorization

Coming soon ...

6.6 XSS Countermeasures

6.6.1 Overview

Cross Site Scripting (below abbreviated as XSS) is injection of malicious scripts across trusted web sites used deliberately making use of the security defects in the web application.
For example, when data entered in Web Application (form input etc.) is output in HTML without appropriate escaping,
the characters of tag existing in input value are interpreted as HTML as is.

If a script with malicious value is run, attacks such as session hijack occur
due to cookie tampering and fetching of cookie values.

Stored & Reflected XSS Attacks

XSS attacks are broadly classified into two categories.

Stored XSS Attacks

In Stored XSS Attacks, the malicious code is permanently stored on target servers (such as database).
Upon requesting the stored information, the user retrieves the malicious script from the server and ends up running the same.

Reflected XSS Attacks

In Reflected attacks, the malicious code sent as a part of the request to the server is
reflected back along with error messages, search results, or other different types of responses.

When a user clicks the malicious link or submits a specially crafted form,
the injected code returns a result reflecting an occurrence of attack on user's browser.
The browser ends up executing the code because the value came from a trusted server.

Both Stored XSS Attacks and Reflected XSS Attacks can be prevented by escaping output value.

6.6.2 How to use

When the input from user is output as is, the system gets exposed to XSS vulnerability.

Therefore, as a countermeasure against XSS vulnerability, it is necessary to escape the characters which have specific meaning in the HTML markup language.

Escaping should be divided into 3 types if needed.

Escaping types:

- Output Escaping
- JavaScript Escaping
- Event handler Escaping

Output Escaping

Escaping HTML special characters is a fundamental countermeasure against XSS vulnerability.

Example of HTML special characters that require escaping and example after escaping these characters are as follows:

Before escaping	After escaping
&	`&`
<	`<`
>	`>`
"	`&quot;`
'	`&#39;`

To prevent XSS, `f:h()` should be used in all display items that are to be output as strings.

An example of application where input value is to be re-output on different screen is given below.

Example of vulnerability when output values are not escaped

This example below is given only for reference; it should never be implemented.

Implementation of output screen

```
<!-- omitted -->
<tr>
    <td>Job</td>
    <td>${customerForm.job}</td>  <!-- (1) -->
</tr>
<!-- omitted -->
```

Sr. No.	Description
(1)	Job, which is a customerForm field, is output without escaping.

Enter <script> tag in Job field on input screen.

Figure.6.1 **Picture - Input HTML Tag**

It is recognized as <script> tag and dialog box is displayed.

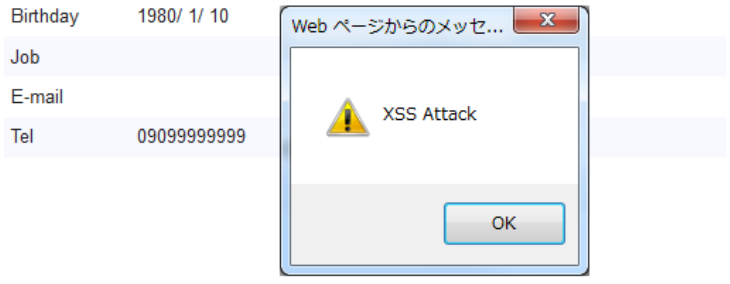


Figure.6.2 **Picture - No Escape Result**

Example of escaping output value using f:h() function

Implementation of output screen

```
<!-- omitted -->
<tr>
    <td>Job</td>
    <td>${f:h(customerForm.job)}</td>  <!-- (1) -->
</tr>
.<!-- omitted -->
```

Sr. No.	Description
(1)	EL function `f:h()` is used for escaping.

Enter <script> tag in Job field on input screen.

Figure.6.3 **Picture - Input HTML Tag**

By escaping special characters, input value is output as is without being recognized as <script> tag.

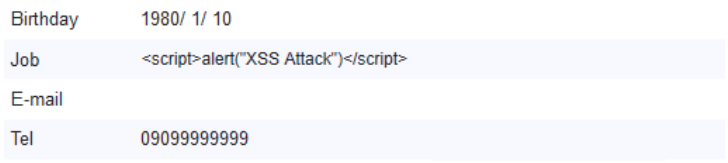


Figure.6.4 **Picture - Escape Result**

Output result

```
<!-- omitted -->
<tr>
    <td>Job</td>
    <td>&lt;script&gt;alert(&quot;XSS Attack&quot;)&lt;/script&gt;</td>
</tr>
<!-- omitted -->
```

Tip: java.util.Date subclass format

It is recommended that you use `<fmt:formatDate>` of JSTL to format and display java.util.Date subclasses. See the example below.

```
    <fmt:formatDate value="${form.date}" pattern="yyyyMMdd" />
```

If `f:h()` is used for setting the value of "value" attribute, it gets converted into String and `javax.el.ELException` is thrown; hence `${form.date}` is used as is. However, it is safe from XSS attack since the value is in yyyyMMdd format.

Tip: String that can be parsed into java.lang.Number or subclass of java.lang.Number

It is recommended that you use `<fmt:formatNumber>` to format and display the string that can be parsed to java.lang.Number subclasses or java.lang.Number. See the example below.

```
    <fmt:formatNumber value="${f:h(form.price)}" pattern="###,###" />
```

There is no problem even if the above is a String; hence when `<fmt:formatNumber>` tag is no longer used, `f:h()` is being used explicitly so that no one forgets to use `f:h()`.

JavaScript Escaping

Escaping JavaScript special characters is a fundamental countermeasure against XSS vulnerability.

Escaping is must if it is required to dynamically generate JavaScript based on the outside input.

Example of JavaScript special characters that require escaping and example after escaping these characters are as follows:

Before escaping	After escaping
'	\'
"	\"
\	\\
/	\/
<	\x3c
>	\x3e
0x0D(Return)	\r
0x0A(Linefeed)	\n

Example of vulnerability when output values are not escaped

Example of occurrence of XSS problem is given below.

```
<html>
  <script  type="text/javascript">
      var aaa = '<script>${warnCode}<\/script>';
      document.write(aaa);
  </script>
<html>
```

Attribute name	Value
warnCode	<script></script><script>alert('XSS Attack!');</script><\/script>

As shown in the above example, in order to dynamically generate JavaScript elements

such as generating the code based on the user input, string literal gets terminated unintentionally leading to XSS vulnerability.

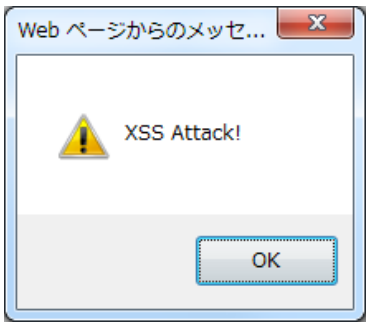


Figure.6.5 **Picture - No Escape Result**

Output result

```
<script type="text/javascript">
    var aaa = '<script><\/script><script>alert('XSS Attack!');<\/script><\/script>';
    document.write(aaa);
</script>
```

> **Tip:** Dynamically generated javascript code depending on user input carries a risk of any script being inserted; hence an alternate way should be considered or it should be avoided as much as possible unless there is a specific business requirement.

Example of escaping output value using f:js() function

To prevent XSS, it is recommended that you use EL function `f:js()` for the value entered by user.

Usage example is shown below.

```
<script type="text/javascript">
  var message = '<script>${f:js(message)}<\/script>';  // (1)
  <!-- omitted -->
</script>
```

Sr. No.	Description
(1)	By using `f:js()` of EL function, the value is set as variable after escaping the value entered by user.

Output result

```
<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

To escape the value of event handler of javascript, `f:hjs()` should be used instead of `f:h()` or `f:js()`. It is equivalent to `${f:h(f:js())}`.

This is because, when `"');alert("XSS Attack");// "` is specified as event handler value such as `<input type="submit" onclick="callback('xxxx');">`, different script gets inserted, After escaping the value in character reference format, escaping in HTML needs to be done.

Example of vulnerability when output values are not escaped

Example of occurrence of XSS problem is given below.

```
<input type="text" onmouseover="alert('output is ${warnCode}') . ">
```

Attribute name	Value
warnCode	 '); alert('XSS Attack!'); // When the above values are set, string literal is terminated unintentionally leading to XSS attack.

XSS dialog box is displayed on mouse over.

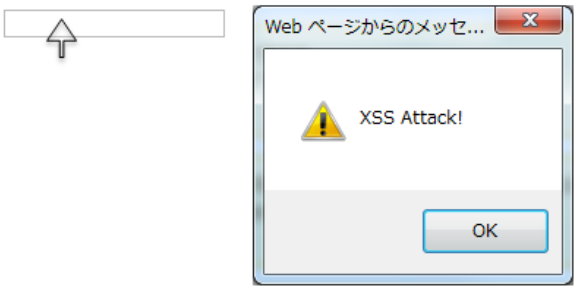


Figure.6.6 **Picture - No Escape Result**

Output result

```
<!-- omitted -->
<input type="text" onmouseover="alert('output is'); alert('XSS Attack!'); // .') ">
<!-- omitted -->
```

Example of escaping output value using f:hjs() function

Example is shown below:

```
<input type="text" onmouseover="alert('output is ${f:hjs(warnCode)}') . ">  // (1)
```

Sr. No.	Description
(1)	Value after escaping by EL function `f:hjs()` is set as an argument of javascript event handler.

XSS dialog is not output on mouse over.

Figure.6.7 **Picture - Escape Result**

Output result

```
<!-- omitted -->
<input type="text" onmouseover="alert(’output is \&#39;); alert(\&#39;XSS Attack!\&#39;);\&quot;
<!-- omitted -->
```

6.7 [coming soon] CSRF(Cross Site Request Forgeries) Countermeasures

Coming soon ...

7

Appendix

7.1 [coming soon] Create Project from Blank Project

Coming soon ...

7.2 [coming soon] Maven Repository Management using Nexus

Coming soon ...

7.3 [coming soon] Exclude Environmental Dependencies

Coming soon ...

7.4 [coming soon] Project Structure Standard

Coming soon ...

7.5 Reference Books

Listed books were referred on writing this guideline. It should be referred to as needed.

Book titles	Publisher	Remarks
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	Japanese
Beginning Java EE 6 GlassFish 3 で始めるエンタープライズ Java	翔泳社	Japanese
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	Japanese

7.6 [coming soon] Spring Comprehension Check

Coming soon ...