
TERASOLUNA Global Framework Development Guideline Documentation

Release 1.0.2-SNAPSHOT

NTT DATA

January 22, 2015

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	14
	MVC Framework	14
	O/R Mapper	14
	View	15
	Security	15
	Validation	15
	Logging	15
	Common Library	15
2.1.3	OSS Versions	16
2.1.4	Building blocks of Common Library	18
	terasoluna-gfw-common	18
	terasoluna-gfw-web	19
	terasoluna-gfw-security-web	21
2.2	Overview of Spring MVC Architecture	22
2.2.1	Overview of Spring MVC Processing Sequence	22
2.2.2	Implementation of each component	23
	Implementation of HandlerMapping	23
	Implementation of HandlerAdapter	24
	Implementation of ViewResolver	24

	Implementation of View	25
2.3	First application based on Spring MVC	27
2.3.1	Prerequisites	27
2.3.2	Create a New Project	27
2.3.3	Run on Server	31
2.3.4	Create an Echo Application	32
	Creating a form object	33
	Create a Controller	33
	Create JSP Files	35
	Implement Input Validation	36
	Summary	40
2.4	Application Layering	41
2.4.1	Layer definition	41
	Application layer	42
	Controller	42
	View	42
	Form	43
	Helper	43
	Domain layer	44
	DomainObject	44
	Repository	45
	Service	45
	Infrastructure layer	45
	RepositoryImpl	45
	O/R Mapper	46
	Integration System Connector	46
2.4.2	Dependency between layers	47
	Processing and data flow with Repository	48
	Processing and data flow without Repository	50
2.4.3	Project structure	51
	[projectName]-domain	52
	[projectName]-web	53
	[projectName]-env	55
3	Tutorial (Todo Application)	59
3.1	Introduction	59
3.1.1	Points to study in this tutorial	59
3.1.2	Target readers	59
3.1.3	Verification environment	59
3.2	Description of application to be created	60
3.2.1	Overview of application	60
3.2.2	Business requirements of application	60

3.2.3	Screen transition of application	61
	Show all TODO	61
	Create TODO	61
	Finish TODO	61
	Delete TODO	62
3.2.4	Error message list	62
3.3	Environment creation	62
3.3.1	Project creation	62
3.3.2	Maven settings	64
3.3.3	Project configuration	68
3.3.4	Creation of configuration file	68
	web.xml settings	68
	Settings of common JSP	72
	Settings of Bean definition file	72
	applicationContext.xml	73
	todo-domain.xml	75
	todo-infra.xml	76
	spring-mvc.xml	77
	logback.xml settings	79
3.3.5	Operation verification	82
3.4	Creation of Todo application	85
3.4.1	Creation of Domain layer	86
	Creation of Domain Object	86
	Repository creation	88
	Creation of RepositoryImpl (Infrastructure layer)	89
	Service creation	91
	Creation of JUnit for Service	96
3.4.2	Creation of application layer	96
	Creation of Controller	96
	Show all TODO	97
	Create TODO	101
	Finish TODO	108
	Delete TODO	114
3.5	Change of infrastructure layer	121
3.5.1	Common settings	121
	Modifications in pom.xml	121
	Definition of data source	122
	Modifications in todo-infra.xml	122
	Creation of todo-env.xml	122
	todo-infra.properties	123
	Modifications in todo-domain.xml	124
	Modifications in TodoServiceImpl	124

3.5.2	Use Spring Data JPA	126
	Modifications in configuration file for using Spring Data JPA	126
	Modifications in pom.xml	126
	Modifications in todo-infra.xml	126
	Modifications in todo-env.xml	128
	Modifications in spring-mvc.xml	129
	Modifications in logback.xml	130
	Settings of Entity	131
	Modifications in TodoRepository	133
	Modifications in TodoRepositoryImpl	134
3.5.3	Use TERASOLUNA DAO	134
	Setting for using the TERASOLUNA DAO	135
	Modifications in pom.xml	135
	Modifications in todo-infra.xml	135
	Modifications in todo-env.xml	136
	Modifications in logback.xml	137
	Creation of sqlMapConfig	138
	Modifications in RepositoryImpl	138
	Create SQLMap file	141
3.6	In the end...	143
4	Application Development using TERASOLUNA Global Framework	145
4.1	Domain Layer Implementation	145
4.1.1	Roles of domain layer	145
4.1.2	Flow of development of domain layer	146
4.1.3	Implementation of Entity	148
	Policy of creating Entity class	148
	Example of creating Entity class	149
	Table structure	149
	Entity structure	151
4.1.4	Implementation of Repository	155
	Roles of Repository	155
	Structure of Repository	155
	Creation of Repository	158
	Example of creating Repository	158
	Structure of Repository	159
	Definition of Repository interface	159
	Creation of Repository interface	159
	Method definition of Repository interface	161
	Creation of RepositoryImpl	164
4.1.5	Implementation of Service	164
	Roles of Service	164

Structure of Service class	166
Reason for separating Service and SharedService	168
Reason for prohibiting the calling of other Service classes from Service class	168
Regarding interface and base classes to limit signature of method	169
Patterns of creating service class	169
Image of Application development - Creating Service for each Entity -	171
Image of Application development - Creating Service for each use case	172
Image of Application development - Creating Service for each use event	175
Creation of Service class	178
Methods of creating Service class	178
Creation of methods of Service class	180
Regarding arguments and return values of methods of Service class	181
Implementation of SharedService class	183
Creation of SharedService class	183
Creation of SharedService class method	183
Regarding arguments and return values of SharedService class method	183
Implementation of logic	183
Operate on business data	184
Returning messages	184
Returning warning message	185
Notifying business error	186
Notifying system error	187
4.1.6 Regarding transaction management	189
Method of transaction management	189
Declarative transaction management	189
Information required for “Declarative transaction management”	189
Propagation of transaction	191
Way of calling the method which is under transaction control	194
Settings for using transaction management	194
PlatformTransactionManager settings	195
Settings for enabling @Transactional	196
Regarding attributes of <tx:annotation-driven> element	197
4.1.7 Appendix	197
Regarding drawbacks of transaction management	197
Programmatic transaction management	198
Sample of implementation of interface and base classes to limit signature	198
4.1.8 Tips	201
Method of dealing with violation of business rules as field error	201
4.2 Implementation of Infrastructure Layer	202
4.2.1 Implementation of RepositoryImpl	202
Implementing Repository using JPA	202
Implementing Repository using MyBatis2	203

Implementing Repository to link with external system using RestTemplate	209
4.3 Implementation of Application Layer	210
4.3.1 Implementing Controller	210
Creating Controller class	212
Mapping request and processing method	212
Mapping with request path	214
Mapping by HTTP method	215
Mapping by request parameter	216
Mapping using request header	216
Mapping using Content-Type header	217
Mapping using Accept header	217
Mapping request and processing method	217
Overview of sample application	218
Request URL	218
Mapping request and processing method	220
Implementing form display	223
Implementing the display of user input confirmation screen	225
Implementing ‘redisplay of form’	228
Implementing ‘create new user’ business logic	231
Implementing notification of create new user process completion	232
Placing multiple buttons on HTML form	234
Source code of controller of sample application	235
Regarding arguments of processing method	236
Passing data to screen (View)	238
Retrieving values from URL path	240
Retrieving request parameters individually	241
Retrieving request parameters collectively	243
Performing input validation	245
Passing data while redirecting request	246
Passing request parameters to redirect destination	249
Inserting values in redirect destination URL path	250
Acquiring values from Cookie	251
Writing values in Cookie	251
Retrieving pagination information	252
Retrieving uploaded file	252
Displaying result message on the screen	253
Regarding return value of processing method	253
HTML response	253
Responding to downloaded data	255
Implementing the process	257
Correlation check of input value	258
Calling business logic	259

Reflecting values to domain object	260
Reflecting values to form object	262
4.3.2 Implementing form object	263
Creating form object	264
Number format conversion of fields	265
Date and time format conversion of fields	266
DataType conversion in controller	267
Specifying annotation for input validation	268
Initializing form object	268
Binding to HTML form	271
Binding request parameters	271
Determining binding result	272
4.3.3 Implementing View	273
Implementing JSP	274
Creating common JSP for include	275
Displaying value stored in model	277
Displaying numbers stored in model	278
Displaying date and time stored in model	278
Binding form object to HTML form	279
Displaying input validation errors	280
Displaying message of processing result	280
Displaying codelist	281
Displaying fixed text content	281
Switching display according to conditions	282
Repeated display of collection elements	283
Displaying link for pagination	284
Switching display according to authority	284
Implementing JavaScript	285
Implementing style sheet	285
4.3.4 Implementing common logic	285
Implementing common logic to be executed before and after calling controller	286
Implementing Servlet Filter	286
Implementing HandlerInterceptor	288
Implementing common processes of controller	289
Implementing HandlerMethodArgumentResolver	289
Implementing “@ControllerAdvice”	291
4.3.5 Prevention of double submission	294
4.3.6 Usage of session	294
5 Architecture in Detail - TERASOLUNA Global Framework	297
5.1 Database Access (Common)	297
5.1.1 Overview	297

About JDBC DataSource	297
JDBC datasource provided by Application Server	298
JDBC datasource provided by OSS/Third-Party library	299
JDBC datasource provided by Spring Framework	299
About transaction management	300
About declaration of transaction boundary/attribute	300
About exclusion control of data	300
About exception handling	300
About multiple datasources	302
About common library classes	303
5.1.2 How to use	304
Datasource settings	304
Settings when using DataSource defined in Application Server	304
Settings when using DataSource for which Bean is defined	305
Settings to enable transaction management	307
JDBC debug log settings	308
Settings related to datasource provided by log4jdbc	308
log4jdbc logger settings	308
Settings of log4jdbc option	311
5.1.3 How to extend	311
Settings for using multiple datasources	311
5.1.4 How to resolve the problem	311
How to resolve N+1	312
Resolving N+1 using JOINs (Join Fetch)	313
Resolving N+1 by fetching related records in batch	314
5.1.5 Appendix	316
Escaping during LIKE search	316
Specifications of escaping of common library	316
About escaping methods provided by common library	317
How to use common library	318
About Sequencer	319
About classes provided by common library	319
How to use common library	320
Classes provided by Spring Framework for converting to data access exception	322
JDBC datasource classes provided by Spring Framework	323
5.2 Database Access (JPA)	325
5.2.1 Overview	325
About JPA	326
O/R Mapping of JPA	326
Basic JPA terminology	326
Managing life cycle of entity	328
About Spring Data JPA	330

5.2.2 How to use	333
pom.xml settings	333
Application Settings	333
Datasource settings	333
EntityManager settings	333
PlatformTransactionManager settings	337
persistence.xml settings	338
Settings for validating Spring Data JPA	338
Settings for using JPA annotations	341
Settings for converting JPA exception to DataAccessException	341
OpenEntityManagerInViewInterceptor settings	341
Creating Repository interface	345
Inheriting the interface of Spring Data	346
Inheriting a common project specific interface in which only the required methods are defined	351
Not inheriting the interface	352
Adding query method	353
Defining query method	354
Specifying query to be executed	354
Fetching entity lock	356
Operating the entities of Persistence Layer directly	357
Setting QueryHints	359
Specifying a query while calling a query method	359
Specifying the query using @Query annotation	359
Specifying with the method name based on naming conventions	364
Specifying as Named query in Properties file	367
Implementing the process to search entities	368
Searching all entities matching the conditions	368
Searching page of entities matching the conditions	370
Implementing search process as per the dynamic conditions of entities	371
Searching all entities matching the dynamic conditions	372
Page search for the entities matching the dynamic conditions	379
Implementing the process to fetch entities	383
Fetching 1 record of entity by specifying ID	383
Fetching 1 record of entity by specifying conditions other than ID	385
Adding entities	387
How to add entities	387
Adding parent-entity and related-entity	391
Adding the related-entity	394
Adding the related-entity directly	395
Updating entities	396
How to update entities	396

Updating the related-entity	397
Updating the related-entity directly	398
Updating by using query method	399
Deleting entities	399
Deleting parent-entity and related-entity	399
Deleting the related-entity	400
Deleting the related-entity directly	402
Deleting using query method	403
Escaping at the time of LIKE search	403
Usage method when type of matching is to be specified in query	404
Usage method when specifying type of matching in logic	405
JOIN FETCH	406
5.2.3 How to extend	408
How to add custom method	408
Adding individual custom method to entity specific Repository interface	408
Adding the custom methods to all Repository interfaces in batch	410
Storing query fetch results in objects other than entity	416
Setting Audit properties	418
Adding common conditions to JPQL to fetch entities from persistence layer	423
Adding common conditions in JPQL to fetch entities	424
Adding common conditions to JPQL to fetch the related-entities	425
How to use multiple PersistenceUnits	427
How to use Native query	427
5.3 Database access (Mybatis2)	428
5.3.1 Overview	428
About Mybatis	428
About TERASOLUNA DAO	429
5.3.2 How to use	431
pom.xml settings	431
Application settings	431
Datasource settings	431
PlatformTransactionManager settings	431
TERASOLUNA DAO settings	432
Settings for LOB handling	434
Mybatis settings	436
Implementing SQL mapping (Basic version)	438
select element implementation	440
“insert” element implementation	443
“update” element implementation	444
“delete” element implementation	445
“procedure” element implementation	445
“sql” element implementation	447

LOB update implementation	448
Implementation for fetching LOB type	452
SQL mapping implementation (dynamic SQL version)	454
Determining whether parameter object is specified	454
Determining whether parameter object (JavaBean) properties exist	455
Determining whether property value of parameter object (JavaBean) is set	456
Determining the property value of parameter object (JavaBean)	459
Common attributes of decision elements	460
Collection iteration	461
Dynamic SQL blocking	464
How to use QueryDAO	465
Single record search	465
Multiple records search	466
Pagination search (TERASOLUNA DAO standard function system)	467
Pagination search (SQL refinement)	470
How to use UpdateDAO	473
Inserting a single record	473
Inserting multiple records (batch execution)	473
Updating a single record	475
Updating multiple records (batch execution)	476
Updating multiple records (Specifying WHERE clause)	476
Deleting a single record	477
Deleting multiple records (batch execution)	477
Deleting multiple records (Specifying WHERE clause)	477
Example showing use of StoredProcedureDAO	477
Example showing use of QueryRowHandleDAO	478
Escaping during LIKE search	479
How to use escaping when matching method is specified in Query	480
How to use escaping when the matching method is specified in Logic	481
About the measures to be taken against SQL Injection	482
Embedding values using bind variable	483
How to embed a value using substitution variable	484
5.3.3 Appendix	485
Implementation wherein related objects are fetched collectively by a single SQL	485
5.4 Exclusive Control	508
5.4.1 Overview	508
Necessity of exclusive control	509
Problem 1	509
Problem 2	510
Problem 3	511
Exclusive control according to the isolation level of transaction	512
Exclusive control using database locking	514

Exclusive control using row lock function of the database	516
Exclusive control using optimistic locking	520
Exclusive control using pessimistic locking	522
Prevention of deadlock	527
Deadlock in table	527
Deadlock between tables	528
5.4.2 How to use	530
How to implement while using JPA (Spring Data JPA)	530
Row lock function of RDBMS	530
Optimistic locking	531
Pessimistic locking	535
Implementation method when using Mybatis	538
Row lock function of RDBMS	538
Optimistic locking	540
Pessimistic locking	545
How to handle an exclusive error	545
Error handling in case of optimistic locking failure	545
Error handling in case of pessimistic locking failure	547
5.5 Input Validation	550
5.5.1 Overview	550
Classification of input validation	550
5.5.2 How to use	551
Single item check	551
Basic single item check	551
Single item check of nested Bean	562
Grouped validation	573
Correlation item check	584
Correlation item check implementation using Spring Validator	585
implementation of input check of correlated items using Bean Validation	590
Definition of error messages	590
Messages to be defined in ValidationMessages.properties	592
Messages to be defined in application-messages.properties	594
5.5.3 How to extend	595
Creation of Bean Validation annotation by combining existing rules	596
Creation of Bean Validation annotation by implementing new rules	600
Rules that cannot be implemented by combining the existing rules	601
Check rules for correlated items	603
Business logic check	608
5.5.4 Appendix	610
Input validation rules provided by Hibernate Validator	610
Bean Validation check rules	610
Hibernate Validator check rules	612

Default messages provided by Hibernate Validator	612
Type mismatch	613
Binding null to blank string field	615
5.6 Logging	617
5.6.1 Overview	617
Types of Logs	617
Output contents of log	619
Log output points	622
5.6.2 How to use	623
Settings of Logback	623
Basic log output by calling API of SLF4J	628
Points to be noted for the description of log output	631
5.6.3 Appendix	632
Using MDC	632
Basic usage method	632
Setting value in MDC using Filter	633
Log output related functionalities provided by common library	636
HttpSessionEventLoggingListener	636
TraceLoggingInterceptor	637
ExceptionLogger	638
5.7 Exception Handling	639
5.7.1 Overview	639
Classification of exceptions	639
Exception handling methods	640
5.7.2 Detail	644
Types of exceptions	644
Business exception	645
Library exceptions that occurs during normal operation	646
System Exception	647
Unexpected System Exception	648
Fatal Errors	649
Framework exception in case of invalid requests	649
Exception Handling Patterns	650
When notifying partial redo of a use case (from middle)	652
When notifying redo of a use case (from beginning)	652
When notifying that the system or application is not in a normal state	653
When notifying that the request contents are invalid	654
When a fatal error has been detected	654
When notifying that an exception has occurred in the presentation layer (JSP etc.) .	655
Basic Flow of Exception Handling	655
Basic flow when the Controller class handles the exception at request level	656
Basic flow when the Controller class handles the exception at use case level	657

Basic flow when the framework handles the exception at servlet level	659
Basic flow when the servlet container handles the exception at web application level	660
5.7.3 How to use	661
Application Settings	661
Common Settings	661
Domain Layer Settings	668
Application Layer Settings	668
Servlet Container Settings	674
Coding Points (Service)	676
Generating Business Exception	676
Generating System Exception	679
Catch the exception to continue the execution	681
Coding Points (Controller)	683
Method to handle exceptions at request level	683
Method to handle exception at use case level	684
Coding points (JSP)	686
Method to display messages on screen using MessagesPanelTag	686
Method to display system exception code on screen	686
5.7.4 How to use (Ajax)	688
5.7.5 Appendix	688
Exception handling classes provided by the common library	688
About SystemExceptionResolver settings	691
Attribute name of result message	693
Attribute name of exception code (message ID)	694
Header name of exception code (message ID)	695
Attribute name of exception object	696
Cache control flag of HTTP response	697
About HandlerExceptionResolverLoggingInterceptor settings	697
List of exception classes to be excluded from scope of logging	698
HTTP response code set by DefaultHandlerExceptionResolver	699
5.8 Session Management	701
5.8.1 Overview	701
Session lifecycle	703
Generating a session	703
Storing attributes in a session	705
Deleting attributes from a session	707
Discarding a session	707
Detecting a request after session timeout	710
About using a session	712
Advantages and disadvantages of using session	713
Advantages and disadvantages of not using session	714
About the data to be stored in session	715

Serializable objects	715
Amount of data to be stored in session	715
Points to be considered in case of application server clustering	716
About storage location of session	717
5.8.2 How to use	717
Using @SessionAttributes annotation	717
Specifying the object to be stored in session	718
Adding object to session	720
Fetching the object stored in session	722
Deleting the object stored in session	725
Process implementation using @SessionAttributes	728
Using session-scoped bean of Spring Framework	728
Bean definition of session scope	729
Using session-scoped bean	731
Deleting objects stored in session	732
Process implementation using session-scoped bean	733
Debug log output of session operations	734
Using the JSP implicit object sessionScope	734
5.8.3 How to extend	734
Synchronizing requests in same session	734
5.8.4 Appendix	736
Example of screen transition implementation using @SessionAttributes in wizard format	736
Example of screen transition across multiple Controllers using session-scoped bean.	751
5.9 Message Management	762
5.9.1 Overview	762
Types of messages	762
Types of messages depending on patterns	763
Message ID	765
Title	765
Labels	766
Result messages	768
Input validation error message	772
5.9.2 How to use	772
Display of messages set in properties file	772
Settings at the time of using properties	772
Display of messages set in properties	773
Display of result messages	775
Using basic result messages	775
Specifying attribute name of result messages	781
Displaying business exception messages	783
5.9.3 How to extend	785

Creating independent message types	785
5.9.4 Appendix	786
Changing attribute of <t:messagesPanel> tag	786
Display of result message wherein ResultMessages is not used	789
Auto-generation tool of message key constant class	792
5.10 Properties Management	795
5.10.1 Overview	795
5.10.2 How to use	796
About properties file definition	796
Using properties in bean definition file	798
Using properties in Java class	800
5.10.3 How to extend	801
Decrypting encrypted values and using them	802
5.11 Pagination	808
5.11.1 Overview	808
Display of list screen at the time of dividing into pages	808
Page search	810
Page search functionality of Spring Data	810
Display of pagination link	813
Structure of pagination link	813
HTML of pagination link	817
Parameters of JSP tag library	820
Process flow when pagination is used	826
5.11.2 How to use	828
Application settings	828
Settings for enabling pagination functionality of Spring Data	828
Page search	829
Implementation of application layer	829
Implementation of domain layer (JPA)	835
Implementation of Service (Mybatis2)	836
Implementation of JSP (Base version)	836
Display of fetched data	836
Display of Pagination link	839
Display of pagination information	843
Carrying forward search conditions using page link	846
Carrying forward the sort condition using page link	848
Implementation of JSP (layout change)	848
Removal of link to navigate to the first page and the last page	848
Removal of link to navigate to previous page and next page	849
Removal of disabled link	850
Change in maximum number of display links to navigate to the specified page	851
Removal of link to navigate to the specified page	852

Implementation of JSP (Operation)	853
Specifying sort condition	853
5.11.3 Appendix	854
About property values of <code>PageableHandlerMethodArgumentResolver</code>	854
Property value of <code>SortHandlerMethodArgumentResolver</code>	859
5.12 Double Submit Protection	860
5.12.1 Overview	860
Problems	860
Double clicking of ‘Update’ button	860
Reloading of screen after completion of update process	862
Invalid screen transition using ‘Back’ button of the browser	863
Solutions	866
Preventing double clicking of a button using JavaScript	868
About PRG (Post-Redirect-Get) pattern	868
Transaction Token Check	870
About NameSpace of transaction token	877
Problems that occur when there is no NameSpace	878
Behavior when NameSpace is specified	879
5.12.2 How to use	880
Preventing double clicking of button using JavaScript	880
Using PRG (Post-Redirect-Get) pattern	881
Using transaction token check	885
Transaction token check provided by common library	885
Attributes of <code>@TransactionTokenCheck</code> annotation	885
Format of transaction token	886
Lifecycle of transaction token	889
Settings for using a transaction token check	892
Settings for handling transaction token errors	893
How to use transaction token check in Controller	894
How to use transaction token check in View (JSP)	895
When multiple usecases are to be implemented in one Controller	898
Typical example of using transaction token check	901
Exclusion control of parallel processing while using a session	903
5.12.3 How to extend	903
How to manage the lifecycle of transaction tokens using a program	903
How to change the maximum limit of transaction tokens	904
5.12.4 Appendix	905
Global Tokens	905
Changing the maximum limit of transaction tokens that can be stored for each NameSpace	905
Implementation of Controller	906
Quick Reference	909

5.13	Internationalization	912
5.13.1	Overview	912
5.13.2	How to use	913
	Configuration to define messages	913
	Changing locale as per browser settings	915
	Settings of AcceptHeaderLocaleResolver	915
	Definition of messages	915
	Implementation of JSP	916
	Changing locale depending on screen operations dynamically	917
	How to define LocaleChangeInterceptor	918
	How to define SessionLocaleResolver	919
	How to define CookieLocaleResolver	920
	Messages settings	921
	Implementation of JSP	922
5.14	Codelist	923
5.14.1	Overview	923
5.14.2	How to use	924
	Using SimpleMapCodeList	925
	Example of codelist settings	925
	Using codelist in JSP	927
	Using codelist in Java class	928
	Using NumberRangeCodeList	929
	Example of codelist settings	930
	Using codelist in JSP	931
	Using codelist in Java class	932
	Using JdbcCodeList	932
	Example of codelist settings	933
	Using codelist in JSP	934
	Using codelist in Java class	935
	How to use SimpleI18nCodeList	936
	Example of setting a codelist	936
	Using codelist in JSP	941
	Using codelist in Java class	943
	Input validation of code value using codelist	943
	Example of @ExistInCodeList settings	944
5.14.3	How to extend	944
	When large number of records need to be read from JdbcCodeList	944
	When reloading the codelist	946
	Using Task Scheduler	947
	Calling refresh method in Controller (Service) class	948
	Customizing the codelist independently	950
5.14.4	Appendix	953

Setting SimpleI18nCodeList	953
Set <code>java.util.Map</code> (<code>key = code value, value = label</code>) for each locale by rows .	953
Set <code>java.util.Map</code> (<code>key = locale, value = label</code>) for each code value by columns	954
Variations of NumberRangeCodeList	955
Create the Descending NumberRangeCodeList	955
Change interval of NumberRangeCodeList	957
5.15 Ajax	959
5.15.1 Overview	959
5.15.2 How to use	959
Application settings	959
Settings to enable the Ajax functionality in Spring MVC	959
Implementing Controller	962
Fetching data	962
Posting form data	968
Posting form data in JSON format	975
Input error handling	978
Handling <code>BindException</code>	978
Handling <code>MethodArgumentNotValidException</code>	982
Handling <code>HttpMessageNotReadableException</code>	982
Handling by using <code>BindingResult</code>	983
Business error handling	985
Handling business exception by exception handling method	986
Handling business exception in processing method	986
5.16 RESTful Web Service	988
5.16.1 Overview	988
What is RESTful Web Service	988
RESTful Web Service development	990
Configuration for RESTful Web Service module	993
REST API implementation sample	995
5.16.2 Architecture	1000
Publishing as a resource on Web	1002
Identifying the resource using URI	1002
Resource operations using HTTP methods	1003
Using an appropriate format	1007
Using the appropriate HTTP status code	1008
Stateless communication between client and server	1009
Link to related resource	1010
5.16.3 How to design	1013
Resource extraction	1013
Assigning URI	1014
Assigning a URI that indicates the API as REST API	1014
Assigning a URI for identifying the API version	1014

Assigning a path for identifying resource	1015
Assigning HTTP methods	1016
Assigning HTTP methods for resource collection URI	1017
Assigning HTTP methods for URI of specific resources	1018
Resource format	1018
JSON Field name	1018
NULL and blank characters	1019
Date format	1019
Hypermedia link format	1020
Format at the time of error response	1021
HTTP Status Code	1021
HTTP status codes when the request is successful	1022
HTTP status code when the cause of request failure lies at client side	1023
HTTP status code when the cause of request failure lies at server side	1025
Authentication and Authorization	1026
Conditional update control of resource	1026
Conditional acquisition control of resource	1027
Cache control of resource	1027
Versioning	1027
5.16.4 How to use	1028
Web application configuration	1028
Application settings	1030
Settings for activating the Spring MVC components necessary for RESTful Web Service	1030
Servlet settings for RESTful Web Service	1034
REST API implementation	1036
Creating REST API packages	1040
Creating Resource class	1041
Creating Controller class	1046
Implementing REST API that fetches collection of resources	1046
Implementing REST API that adds a resource to collection	1054
Implementing REST API that fetches specified resource	1056
Implementing REST API that updates specified resource	1058
Implementing REST API that deletes specified resource	1060
Implementing exception handling	1062
Implementation to output error information in response Body	1064
Implementing input error exception handling	1070
Implementing exception handling for “Resource not found” error	1076
Implementing exception handling for business errors	1079
Implementing exception handling for exclusive errors	1081
Implementing exception handling for system errors	1082
Resolving error codes and messages using ExceptionCodeResolver	1084

Implementing the error handling notified to Servlet Container	1086
Implementing the Controller that sends error response	1089
Creating a static JSON file to be sent as response when a fatal error occurs	1090
Settings for handling an error that is notified to Servlet Container	1091
Security measures	1093
Authentication and Authorization	1093
CSRF measures	1093
Conditional operations for resource	1094
Cache control for resource	1094
5.16.5 Appendix	1094
Settings when RESTful Web Service and client application are operated as the same Web application	1094
How to set DispatcherServlet for RESTful Web Service	1094
Implementing hypermedia link	1097
Implementing common parts	1097
Implementation for each resource	1100
Creating RESTful Web Service conforming to HTTP specifications	1103
Settings for location header at the time of POST	1103
Implementation of common components	1103
Implementation for each resource	1104
Setting to dispatch OPTIONS method request to the Controller	1106
Implementing OPTIONS method	1107
Implementing HEAD method	1109
Disabling CSRF measures	1111
Enabling XXE Injection measures	1112
How to copy Joda-Time classes using Dozer	1115
Source code for application layer	1117
MemberRestController.java	1118
ApiErrorCreator.java	1120
ApiGlobalExceptionHandler.java	1122
Source code of the domain layer class created at the time of REST API implementation	1124
Member.java	1126
MemberCredential.java	1129
Gender.java	1131
MemberRepository.java	1132
MemberService.java	1133
MemberServiceImpl.java	1133
DomainMessageCodes.java	1136
member-mapping.xml	1137
5.17 File Upload	1138
5.17.1 Overview	1138
Basic flow of upload process	1138

About classes provided by Spring Web	1140
5.17.2 How to use	1142
Application settings	1142
Settings to enable Servlet3.0 upload functionality	1142
Settings to enable fetching of request parameters in Servlet Filter processing	1146
Settings to link Spring MVC with upload functionality of Servlet3.0	1147
Settings for exception handling	1147
Uploading a single file	1151
Implementing form	1151
Implementing JSP	1152
Implementing Controller	1153
Bean Validation of file upload	1156
Implementing validation to verify that the file is selected	1156
Implementing validation to verify that the file is not empty	1157
Implementing validation to verify that file size is within allowable range	1158
Implementing form	1159
Implementing Controller	1160
Uploading multiple files	1160
Implementing form	1161
Implementing JSP	1162
Implementing Controller	1163
Uploading multiple files using the “multiple” attribute of HTML5	1164
Implementing form	1164
Implementing Validator	1165
Implementing JSP	1167
Implementing Controller	1167
Temporary upload	1168
Implementing Controller	1170
5.17.3 How to extend	1172
Housekeeping of unnecessary files at the time of temporary upload	1172
Implementing component class to delete unnecessary files	1172
Scheduling settings of the process for deleting unnecessary files	1174
5.17.4 Appendix	1176
Dos attack with respect to upload functionality	1176
Attack by executing uploaded files on Web Server	1177
5.18 File Download	1178
5.18.1 Overview	1178
5.18.2 How to use	1179
Downloading PDF files	1179
Implementation of Custom View	1179
Definition of ViewResolver	1181
Specifying View in controller	1182

Downloading Excel files	1182
Implementation of Custom View	1183
Definition of ViewResolver	1184
Specifying View in controller	1184
Downloading arbitrary files	1185
Implementation of Custom View	1186
Definition of ViewResolver	1187
Specifying View in controller	1187
5.19 Screen Layout using Tiles	1189
5.19.1 Overview	1189
5.19.2 How to use	1191
pom.xml setting	1191
Integration of Spring MVC and Tiles	1192
5.19.3 How to extend	1201
Setting multiple layouts	1201
5.20 System Date	1209
5.20.1 Overview	1209
5.20.2 How to use	1210
Returning system time of server	1211
Returning the fixed time fetched from DB	1212
Returning time obtained by adding the difference registered in DB to the server system time	1214
Caching and reloading the difference	1217
5.20.3 Testing	1218
Unit Test	1218
Example wherein process changes with date	1221
Integration Test	1224
System Test	1225
Production	1225
5.21 Utilities	1227
5.21.1 Bean Mapping (Dozer)	1227
Overview	1227
How to use	1229
Bean definition for using Dozer	1229
Mapping when the field name and the type between Beans is same	1229
Mapping when the field name is the same and type between Beans is different . . .	1232
Mapping when the field name between Beans is different	1232
One-way/Two-way mapping	1235
Mapping of nested fields	1236
Collection mapping	1238
How to extend	1248
Creating custom convertor	1248
Appendix	1251

Field exclusion settings (field-exclude)	1251
Specifying mapping (map-id)	1253
Settings to exclude null/empty field of copy source (map-null, map-empty)	1255
Mapping from string to date/time object	1256
Mapping error	1258
5.21.2 Date Operations (Joda Time)	1259
Overview	1259
How to use	1259
Fetching date	1259
Type conversion	1263
Date operations	1264
Fetching the duration	1267
JSP Tag Library	1269
Example (display of calendar)	1272
6 Security for TERASOLUNA Global Framework	1277
6.1 Spring Security Overview	1277
6.1.1 Overview	1277
Authentication	1277
Password hashing	1278
Authorization	1278
6.1.2 How to use	1278
pom.xml settings	1279
Web.xml settings	1279
spring-security.xml settings	1280
6.2 [coming soon] Spring Security Tutorial	1282
6.3 [coming soon] Authentication	1283
6.4 Password Hashing	1284
6.4.1 Overview	1284
6.4.2 How to use	1285
BCryptPasswordEncoder	1285
Configuration example of BCryptPasswordEncoder	1286
StandardPasswordEncoder	1287
Configuration example of StandardPasswordEncoder	1289
NoOpPasswordEncoder	1290
6.4.3 How to extend	1290
Example where ShaPasswordEncoder is used	1291
6.4.4 Appendix	1293
6.5 [coming soon] Authorization	1294
6.6 XSS Countermeasures	1295
6.6.1 Overview	1295
Stored & Reflected XSS Attacks	1295

6.6.2	How to use	1296
	Output Escaping	1296
	Example of vulnerability when output values are not escaped	1296
	Example of escaping output value using f:h() function	1298
	JavaScript Escaping	1299
	Example of vulnerability when output values are not escaped	1300
	Example of escaping output value using f:js() function	1301
	Event handler Escaping	1301
	Example of vulnerability when output values are not escaped	1302
	Example of escaping output value using f:hjs() function	1302
6.7	[coming soon] CSRF(Cross Site Request Forgeries) Countermeasures	1304
7	Appendix	1305
7.1	Create New Project From Blank	1305
	7.1.1 Creating a new project	1305
	7.1.2 Creating simple Echo project	1310
7.2	Maven Repository Management using NEXUS	1312
	7.2.1 Why NEXUS?	1312
	7.2.2 Install and Start-up	1312
	7.2.3 Add TERASOLUNA Global Framework repository	1313
	7.2.4 settings.xml	1314
	7.2.5 mvn deploy how to	1315
	7.2.6 pom.xml	1316
	7.2.7 Upload 3rd party artifact (ex. ojdbc6.jar)	1316
	use artifact	1318
7.3	Removing Environment Dependency	1319
	7.3.1 Objective	1319
	7.3.2 Rules	1320
	7.3.3 Deployment	1321
	Deployment in Tomcat	1321
	Deployment to other application server	1321
	Continuous deployment	1322
	Operations of SNAPSHOT version	1322
	Operations of RELEASE version	1323
	Release on Application Server	1324
7.4	Project Structure Standard	1327
	7.4.1 Simple pattern	1327
	7.4.2 Complex pattern	1328
7.5	Reference Books	1330
7.6	Spring Framework Comprehension Check	1331

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.

- TERASOLUNA is a registered trademark of NTT DATA Corporation.
- All other company names and product names are the registered trademarks or trademarks of their respective companies.

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 building Web Application using Maven

This guideline is not for beginners.

In order to check whether the readers have basic knowledge of Spring Framework, refer to [Spring Framework 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	<ul style="list-style-type: none"> • <i>Database Access (JPA)</i> • <i>Database access (Mybatis2)</i> <p>(Details about using bind variable at the time of placeholders for query parameters)</p>
A1	Injection XXE(XML External Entity) Injection	<ul style="list-style-type: none"> • <i>Ajax</i>
A2	Broken Authentication and Session Management	<ul style="list-style-type: none"> • <i>[coming soon] Authentication</i>
A3	Cross-Site Scripting (XSS)	<ul style="list-style-type: none"> • <i>XSS Countermeasures</i>
A4	Insecure Direct Object References	No mention in particular
A5	Security Misconfiguration	<ul style="list-style-type: none"> • <i>Logging</i>(Mention about message contents of log) • <i>Method to display system exception code on screen</i>(Mention about message output at the time of system exception)
A6	Sensitive Data Exposure	<ul style="list-style-type: none"> • <i>Properties Management</i> • <i>Password Hashing</i> (Mention about password hash only)
A7	Missing Function Level Access Control	<ul style="list-style-type: none"> • <i>[coming soon] Authorization</i>
A8	Cross-Site Request Forgery (CSRF)	<ul style="list-style-type: none"> • <i>[coming soon] CSRF(Cross Site Request Forgeries) Countermeasures</i>
A9	Using Components with Known Vulnerabilities	No mention in particular
A10	Unvalidated Redirects and Forwards	<ul style="list-style-type: none"> • <i>[coming soon] Authentication</i>(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 . Fixed guideline errors (corrected typos, mistakes in description etc.) For details, refer to Issue list of 1.0.1 . Added Japanese version of the following. <ul style="list-style-type: none">• <i>Criterion-based mapping of guideline</i>• <i>RESTful Web Service</i>• Tutorial (Todo Application for REST)
	Overall modifications	
	Japanese version	
	English version	Added English version of the following. <ul style="list-style-type: none">• <i>In the Beginning</i>• <i>Summary - Architecture of TERASOLUNA Global Framework</i>• <i>Tutorial (Todo Application)</i>• <i>Application Development using TERASOLUNA Global Framework</i>• <i>Input Validation</i>• <i>Exception Handling</i>• <i>Message Management</i>• <i>.../ArchitectureInDetail/utilities/JodaTime</i>• <i>XSS Countermeasures</i>• <i>Reference Books</i>
	<i>Stack of TERASOLUNA Global Framework</i>	Updated the OSS version in accordance with bug fixes. <ul style="list-style-type: none">• GroupId (<code>org.springframework</code>) updated to 3.2.10.RELEASE from 3.2.4.RELEASE• GroupId (<code>org.springframework.data</code>)/ArtifactId(<code>spring-data</code>) updated to 1.6.4.RELEASE from 1.6.1.RELEASE• GroupId (<code>org.springframework.data</code>)/ArtifactId(<code>spring-data</code>) updated to 1.4.3.RELEASE from 1.4.1.RELEASE• GroupId (<code>org.aspectj</code>) updated to 1.7.4 from 1.7.3• Deleted GroupId (<code>javax.transaction</code>)/ArtifactId(<code>jta</code>)
	<i>Implementation of Application Layer</i>	Added a warning about CVE-2014-1904 (XSS Vulnerability of action attribute in <code><form:form></code> tag)
	Japanese version	Added description about bug fix
	<i>Message Management</i>	<ul style="list-style-type: none">• Fixed bugs of <code><t:messagesPanel></code> tag of common library (terasoluna-gfw#10)
1.7. Change Log	Japanese version	Updated description about bug fix
	<i>Pagination</i>	<ul style="list-style-type: none">• Fixed bugs of <code><t:pagination></code> tag of common library (terasoluna-gfw#12)• Fixed bugs of Spring Data Commons (terasoluna-gfw#22)

2

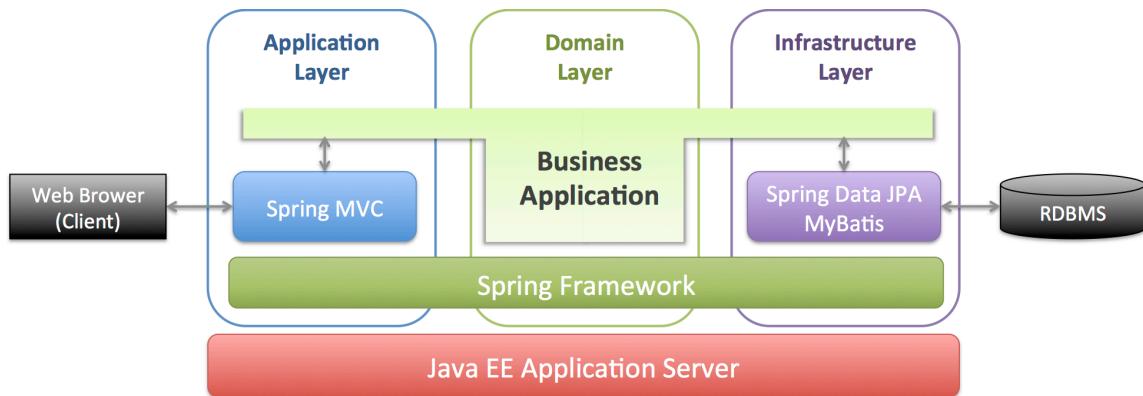
Summary - Architecture of TERASOLUNA Global Framework

The architecture adopted in this guideline is explained here.

2.1 Stack of TERASOLUNA Global Framework

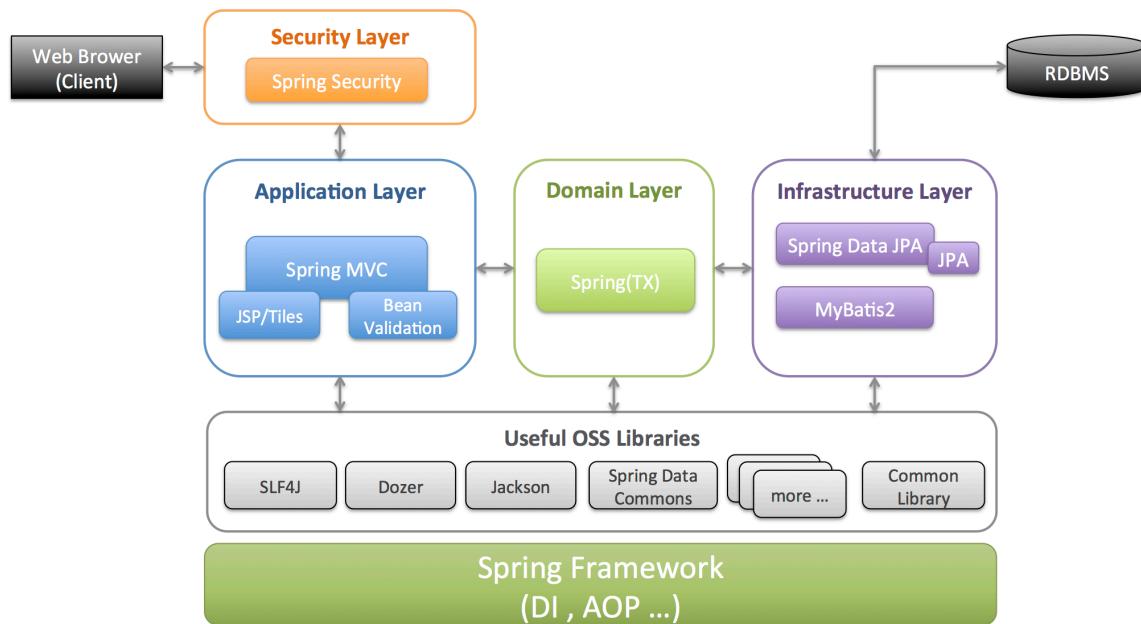
2.1.1 Summary of Software Framework of TERASOLUNA Global Framework

Software Framework being used in TERASOLUNA Global Framework is not a proprietary 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 Framework 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.

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.

If standardize layout of view, 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 4.3 is 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 situation.

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 1.0.2.RELEASE.

Type	GroupId	ArtifactId	Version	Remarks
Spring	org.springframework	spring-aop	3.2.13.RELEASE	
Spring	org.springframework	spring-aspects	3.2.13.RELEASE	
Spring	org.springframework	spring-beans	3.2.13.RELEASE	
Spring	org.springframework	spring-context	3.2.13.RELEASE	
Spring	org.springframework	spring-context-support	3.2.13.RELEASE	
Spring	org.springframework	spring-core	3.2.13.RELEASE	
Spring	org.springframework	spring-expression	3.2.13.RELEASE	
Spring	org.springframework	spring-jdbc	3.2.13.RELEASE	
Spring	org.springframework	spring-orm	3.2.13.RELEASE	
Spring	org.springframework	spring-tx	3.2.13.RELEASE	
Spring	org.springframework	spring-web	3.2.13.RELEASE	
Spring	org.springframework	spring-webmvc	3.2.13.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	

Continued on Next page

Table. 2.1 – continued from previous page

Type	GroupId	ArtifactId	Version	Remarks
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	
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	
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

terasoluna-gfw-common provide following components.

Classification	Component Name	Description
<i>Exception Handling</i>	Exception Class	Provide general exception classes.
	Exception Logger	Provide logger class for logging the exception that occurred in application.
	Exception Code	Provide mechanism (classes) for resolving the exception code (message ID) that corresponds to the exception class.
	Exception Logging Interceptor	Provide interceptor class of AOP for logging the exception that occurred in domain layer.
<i>System Date</i>	System Date Time Factory	Provide classes for acquiring the system date time.
<i>Codelist</i>	CodeList	Provide classes for generating CodeList.
<i>Database Access (Common)</i>	Query Escape	Provide class for escape processing of value to bind into the SQL and JPQL.
	Sequencer	Provide classes for acquiring the sequence value.

terasoluna-gfw-web

terasoluna-gfw-web provide following components.

Classification	Component Name	Description
<i>Double Submit Protection</i>	Transaction Token Check	Provide mechanism (classes) for protecting Web Application from double submitting of request.
<i>Exception Handling</i>	Exception Handler	Provide exception handler class(sub class of class that provided by Spring MVC) for integrating with exception handling components that provided from common library.
	Exception Logging Interceptor	Provide interceptor class of AOP for logging the exception that handled by Spring MVC.
<i>Codelist</i>	Populate CodeList interceptor	Provide interceptor class of Spring MVC for storing CodeList information into request scope, for the purpose of acquiring CodeList from View.
<i>File Download</i>	General Download View	Provide abstract class for acquiring data from input stream and writing to stream for download.
<i>Logging</i>	ServletFilter for storing Tracking ID	Provide Servlet Filter class for setting Tracking ID into MDC(Mapped Diagnostic Context) and request scope and response header, for the purpose of improving traceability. (If does not exist a Tracking ID in request header, generate a Tracking ID by this component)
	General ServletFilter for storing to MDC	Provide abstract class for storing any value into Logger's MDC
	ServletFilter for clearing MDC	Provide ServletFilter class for clearing information that stored in Logger's MDC.
<i>Pagination</i>	JSP Tag for displaying Pagination Links	Provide JSP Tag Library for displaying Pagination Links using classes that provided by Spring Data Commons.
<i>Message Management</i>	JSP Tag for displaying Result Messages	Provide JSP Tag Library for displaying Result Messages.
<i>TagLibAndELFunctionsOverview</i>	EL Functions for XSS countermeasures	Provide EL Functions for XSS countermeasures.

terasoluna-gfw-security-web

terasoluna-gfw-security-web provide following components.

Classification	Component Name	Description
<i>Logging</i>	ServletFilter for storing name of authenticated user	Provide ServletFilter class for setting name of authenticated user into MDC, for the purpose of improving traceability.
<i>[coming soon] Authentication</i>	Authentication Success Handler that can be specified redirect path	Provide Authentication Success Handler class that redirect to specified path in the Web Application when authentication is successful.
<i>[coming soon] CSRF(Cross Site Request Forgeries) Countermeasures</i>	CSRF(Cross site request forgeries) Check	Provide mechanism (classes) for protecting Web Application from attack using CSRF. (Interim measure until the introduction of Spring Security 3.2)

2.2 Overview of Spring MVC Architecture

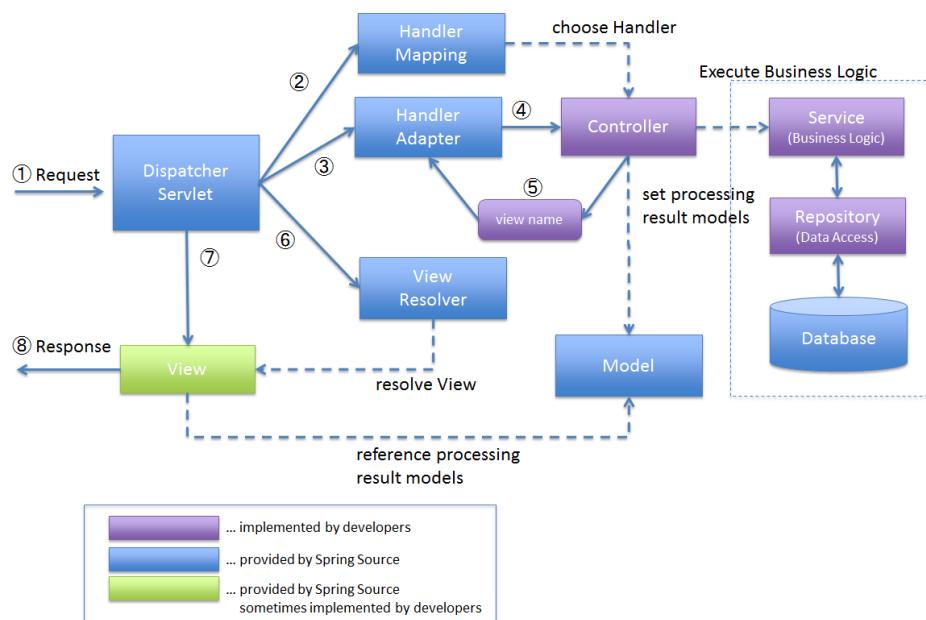
Official website of Spring MVC says the following

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.

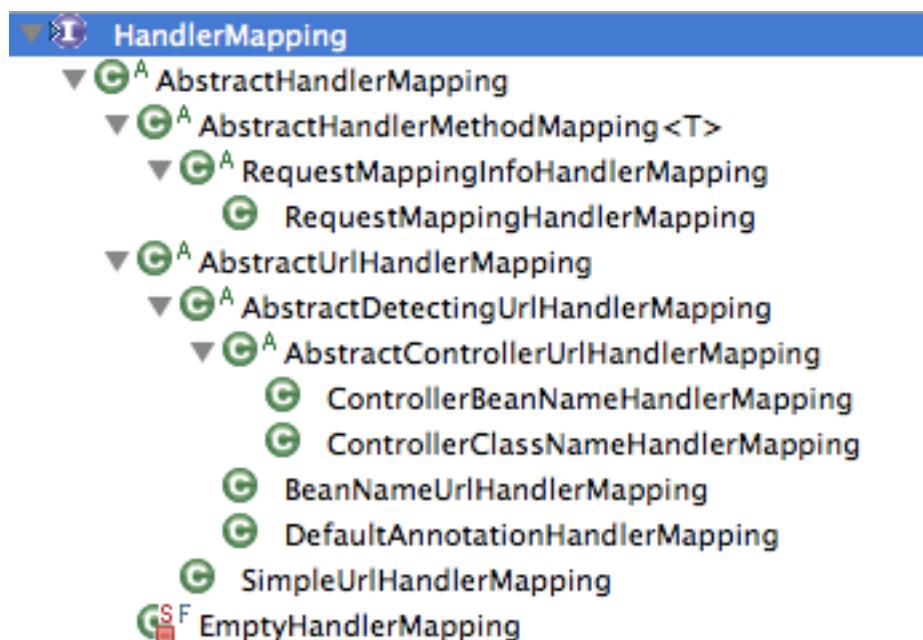
5. Controller executes the business logic, sets the processing result in Model and returns the logical name of view to HandlerAdapter.
6. DispatcherServlet dispatches the task of resolving the View corresponding to the View name to ViewResolver. ViewResolver returns the View mapped to View name.
7. DispatcherServlet 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.

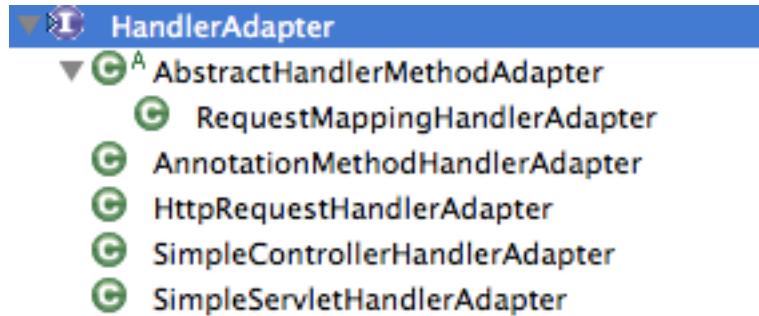


Usually `org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping` is used. This class reads `@RequestMapping` annotation from the Controller and uses the method of Controller that matches with URL as Handler class.

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 through `<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.

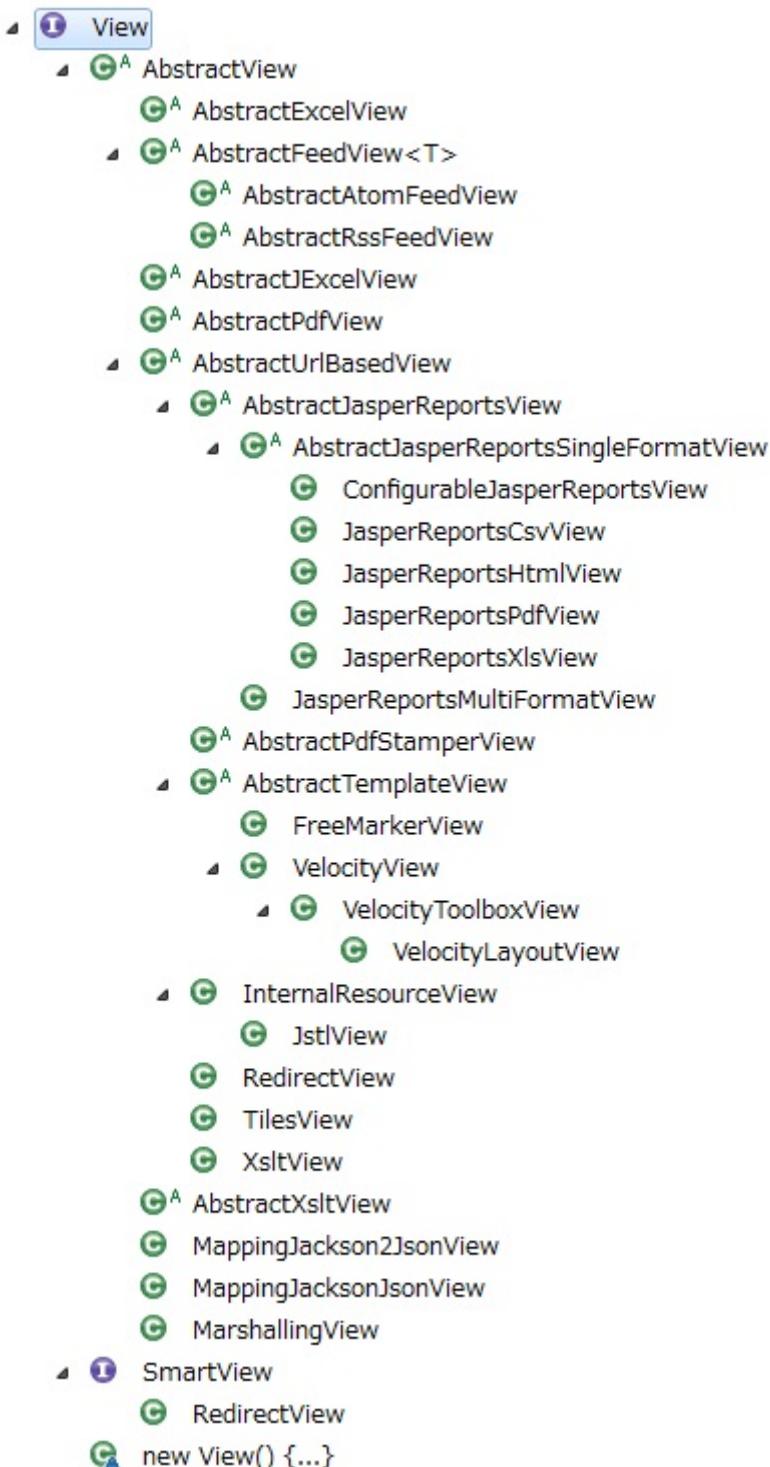
For screen (JSP), View is resolved using InternalResourceViewResolver and for File download View is resolved using BeanNameViewResolver.

For details, refer [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 [File Download](#).



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 [Create New Project From Blank](#) in case using Spring Tool Suite 3.4.

We plan to update the contents of this chapter in future.

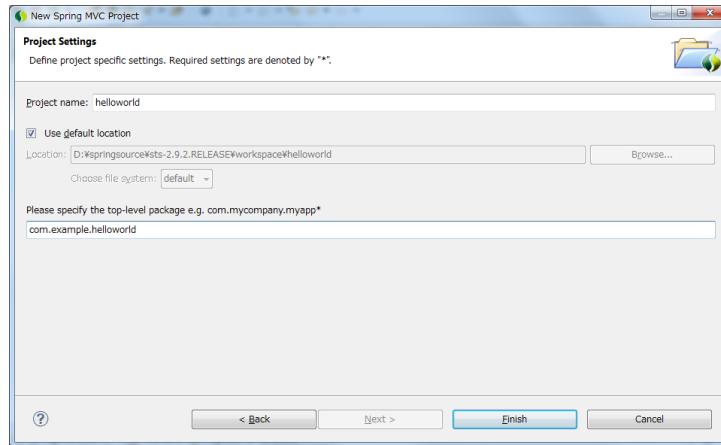
2.3.2 Create a New Project

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

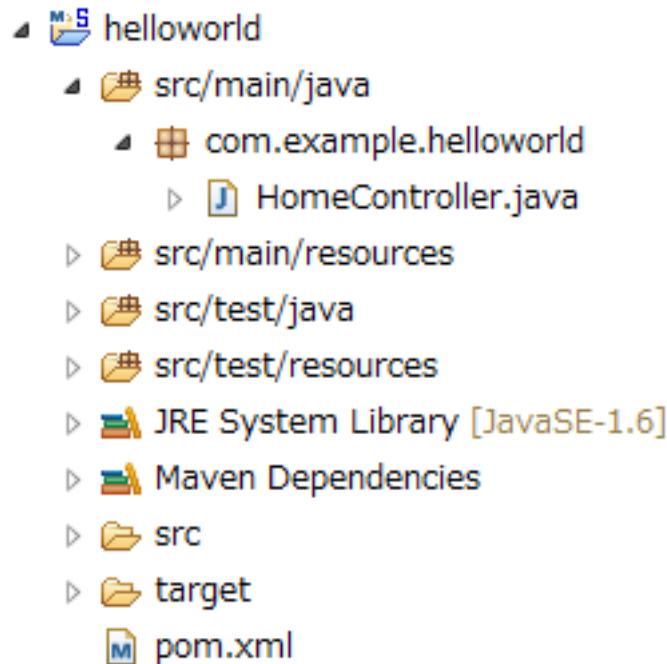
Note: If proxy server is being used, there is a possibility of not being able to select Spring Template Project. In order to resolve this, do the following.

- 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 Templates.
-

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.org/schema/beans"
    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/spring-
        beans.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-
        context.xsd"/>
```

```

<!-- 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 <code>@Controller</code> 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 <code>WEB-INF/views/home.jsp</code> as per the configuration in “Explanation of Spring MVC configuration files (2)”.

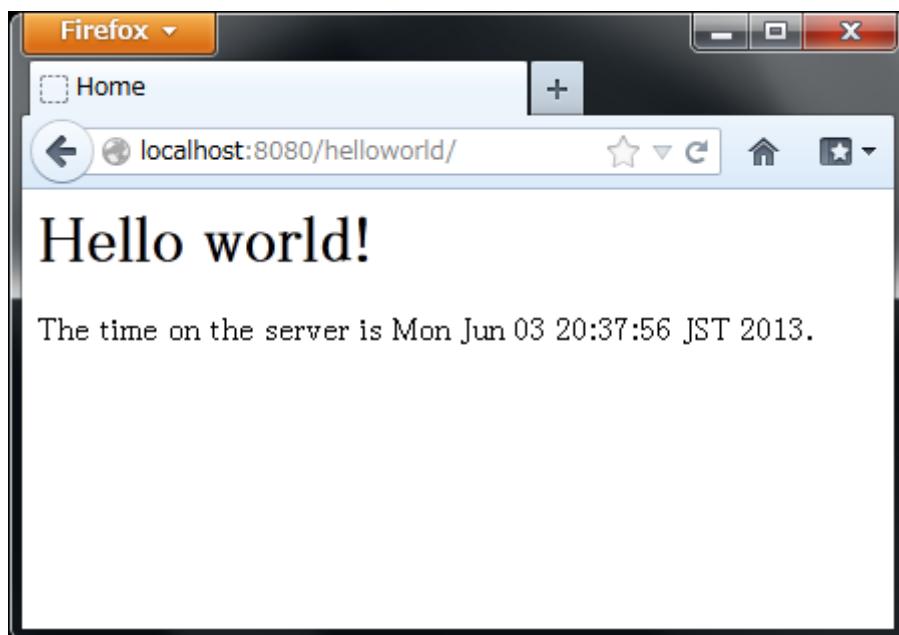
The object set in Model is set in `HttpServletRequest`. The value passed from Controller can be output by mentioning `${serverTime}` in `home.jsp` as shown below. (**However, as described below, it is necessary to perform HTML escaping since there is possibility of `${XXX}` becoming a target of XSS**)

```
<%@ 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>
```

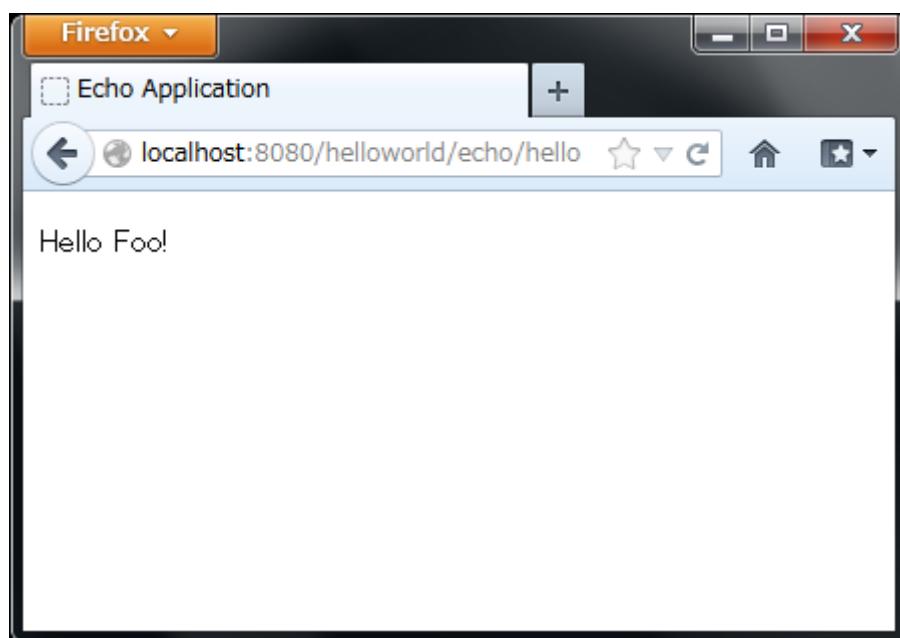
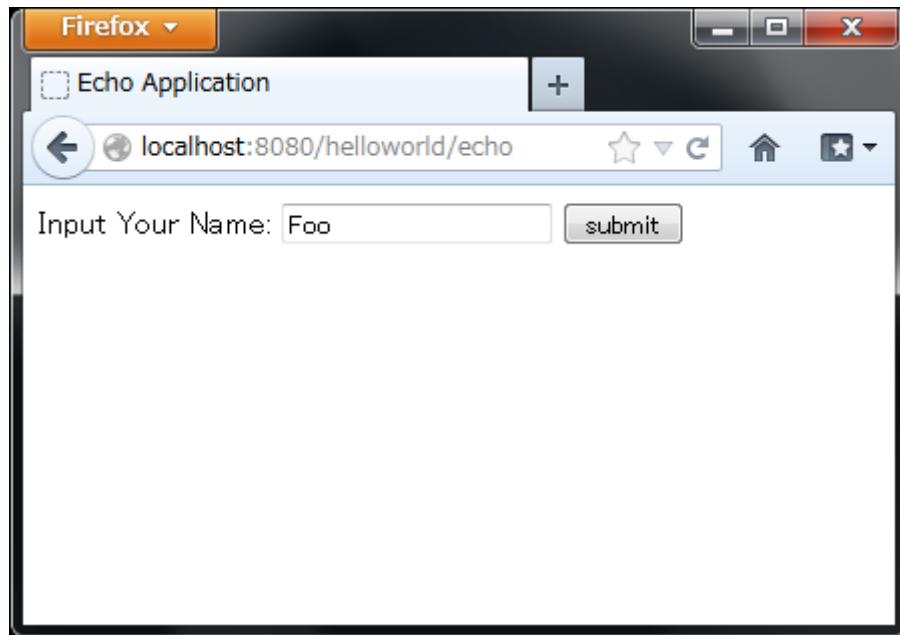
2.3.3 Run on Server

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)	<p>Add @ModelAttribute annotation to the method. Return value of such a method is automatically added to the Model.</p> <p>Attribute name can be specified in @ModelAttribute, but the class name with the first letter in lower case is the default attribute name.</p> <p>In this case it will be echoForm. This attribute name must match with the value of modelAttribute of form:form tag.</p>
(2)	<p>When nothing is specified in value attribute of @RequestMapping annotation at the method level, it is mapped to @RequestMapping added at class level.</p> <p>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.</p>
(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)	<p>HTML form is constructed by using tag library. Specify the name of form object created by Controller in modelAttribute.</p> <p>Refer herefor tag library.</p>

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 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}">
```

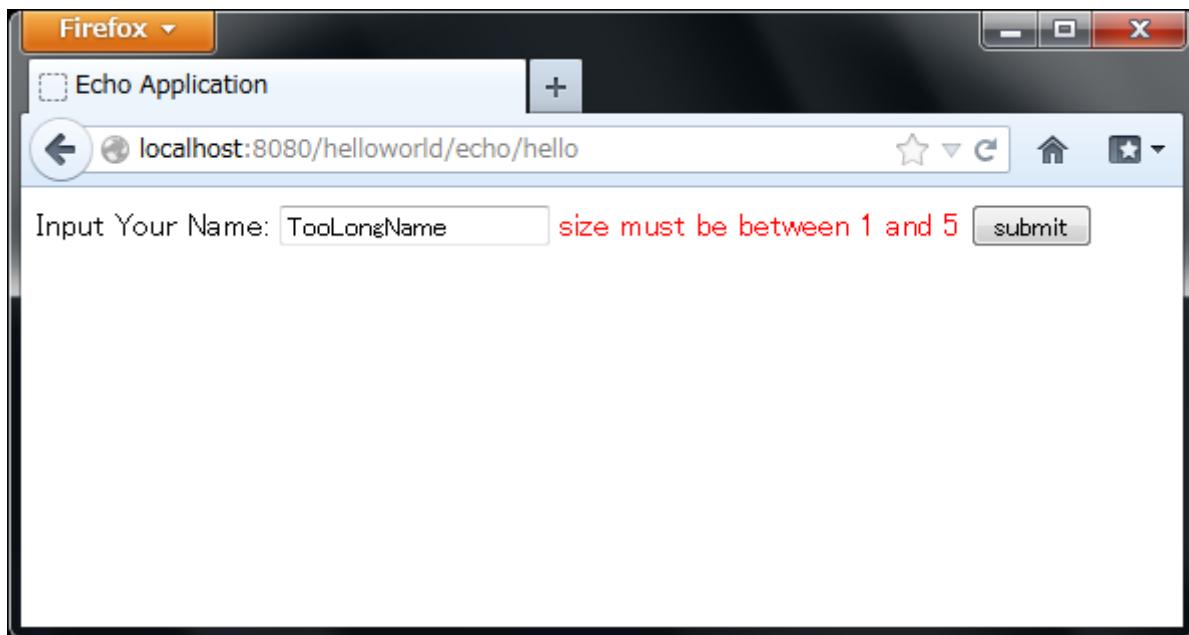
```
<form:label path="name">Input Your Name:</form:label>
<form:input path="name" />
<form:errors path="name" cssStyle="color:red" /><!-- (1) -->
<input type="submit" />
</form:form>
</body>
</html>
```

Item number	Description
(1)	Add <code>form:errors</code> 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:

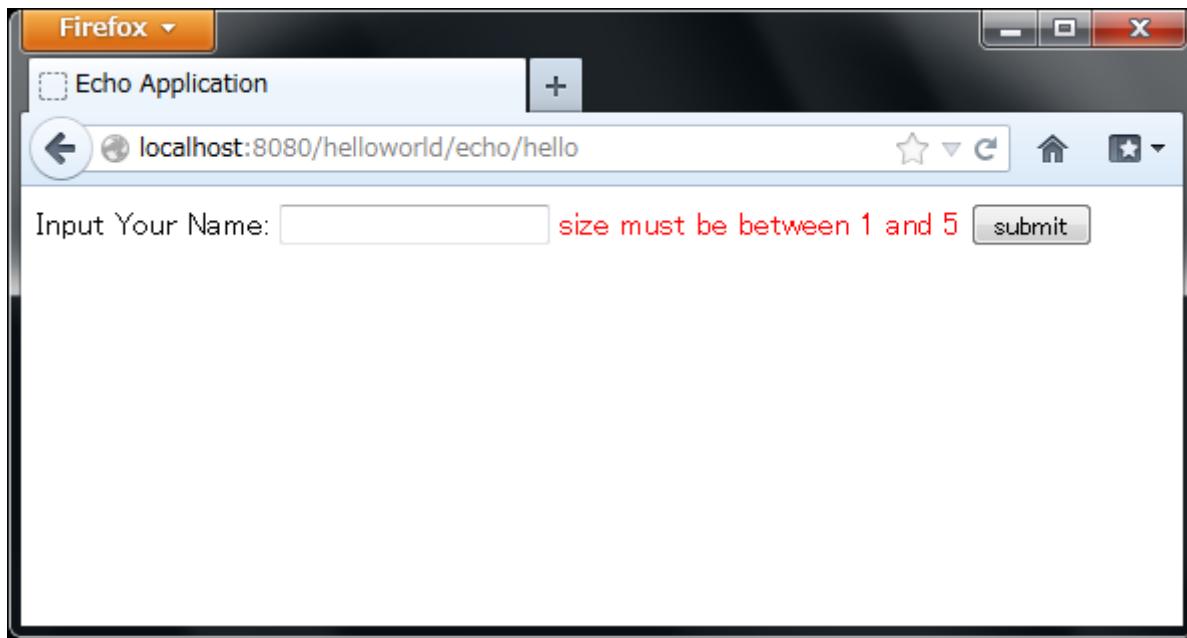
- When an empty name is sent
- Size is more than 5 characters.



The generated HTML is as follows

```
<body>

<form id="echoForm" action="/helloworld/echo/echo" 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

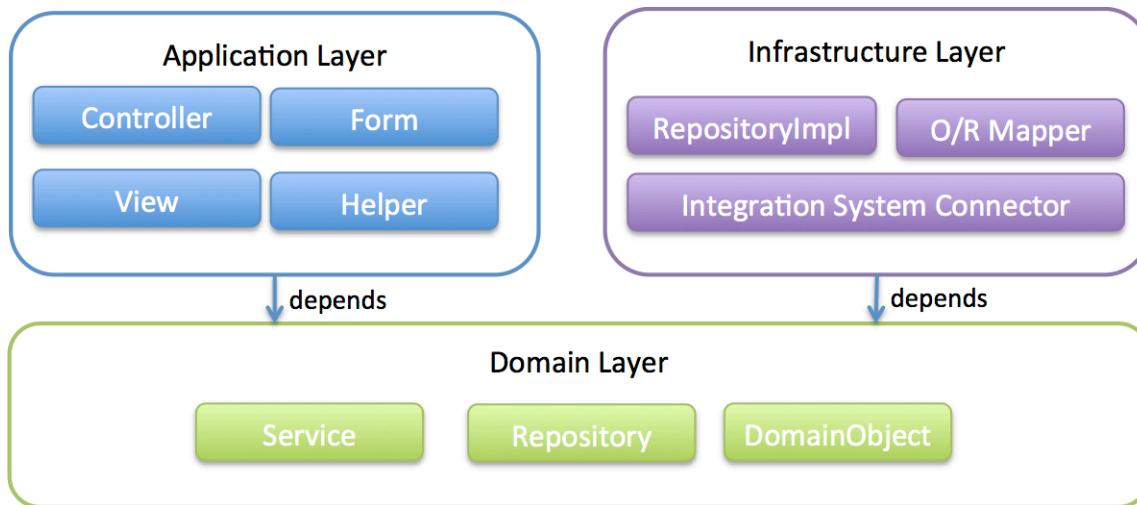
If above points are still not understood, it is recommended to read this chapter again and start again from building the environment. This will imporve the understanding of above concepts.

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.



Both application layer and infrastructure layer depend 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.

In this layer, it provides the following implementations:

- Provides UI(User Interface) to input and output information.
- Handles the request from clients.
- Validates the input data from a clients
- Calls components in the domain layer corresponding to the request from clients.

This layer should be as thin as possible and should not contain any business rules.

Controller

Controller is mainly responsible for the below:

- Controls of screen flow (mapping the request to the corresponding process and returning the result)
- Calls services in the domain layer (executing main logic corresponding to the request)

In Spring MVC, POJO class having `@Controller` annotation becomes the Controller class.

Note: When storing the data in the session, controllers also have a role to control the lifecycle of objects in the session.

View

View's responsibilities is generating output to the client. Returns output results in various formats such as HTML/PDF/Excel/JSON.

In Spring MVC, all this is done by View class.

Tip: If use the JSON or XML as output results at REST API or Ajax request, `HttpMessageConverter` class plays this responsibility.

Details refer to “[RESTful Web Service](#)”.

Form

Form's main responsibilities are the following:

- Represents the form of HTML. (sending form information to the Controller, or outputting the processing result data to form of HTML)
- Defines validation rules for input data. (by adding annotations of Bean Validation)

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

Note: To retain the domain layer which is independent from the application layer, need to perform following processing at the application layer.

- Conversion from Form to DomainObject(such as Entity)
- conversion from DomainObject to Form

If conversion code is too many, recommend to perform of either or both of the following measures and keep Controller's source code with simple state.

- Create helper class and delegate conversion logic to helper classes.
- Use the *Dozer*.

Tip: If use the JSON or XML as input data at REST API or Ajax request, Resource class plays this responsibility. Also, `HttpMessageConverter` class plays responsibility to convert from input data of JSON or XML format to Resource class.

Details refer to “*RESTful Web Service*”.

Helper

It plays the role of assisting the Controller.

Helper is an option. Please create as POJO class if required.

Note: The main duty of Controller is routing (URL mapping and specifying 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.

Helper class exists to improve the visibility of Controller; hence, it is OK to think of it as a part of Controller.
(Helper class is similar to private methods of Controller)

Domain layer

Domain layer is the core of the application, and execute business rules.

In this layer, it provides the following implementations.

- DomainObject
- Checking business rules corresponding to the DomainObject (like whether there is sufficient balance when crediting amount into the account)
- Executing business rules (reflects values corresponding to business rules)
- Executing CRUD operations for the DomainObject

Domain layer is independent from 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 etc.(generally indicated by a verb)
- Summary model such as YearlySales, MonthlySales etc.

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

Note: Mainly [Models holding state only](#) are 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.

In this guideline, it is recommended to draw the transaction boundary at the method of Service.

Note: 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.

This layer also can be the transaction boundary depending on the requirements.

Tip: When using Spring Data JPA or MyBatis3, most RepositoryImpls are created automatically.

O/R Mapper

It is responsible for mapping database with Entity.

This function is provided by JPA, MyBatis and Spring JDBC.

Specifically, the following classes are the O/R Mapper.

- `EntityManager` in case of using JPA,
- `Mapper interface` or `SqlSession` in case of using MyBatis3
- `QueryDAO` or `UpdateDAO` in case of using MyBatis2(TERASOLUNA DAO)
- `JdbcTemplate` in case of using plain Spring JDBC

O/R Mapper used for implementation of Repository.

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 integrates a data store other than the database; such as messaging system, Key-Value-Store, Web service, existing legacy system, external system etc.

It is also used for implementation of Repository.

2.4.2 Dependency between layers

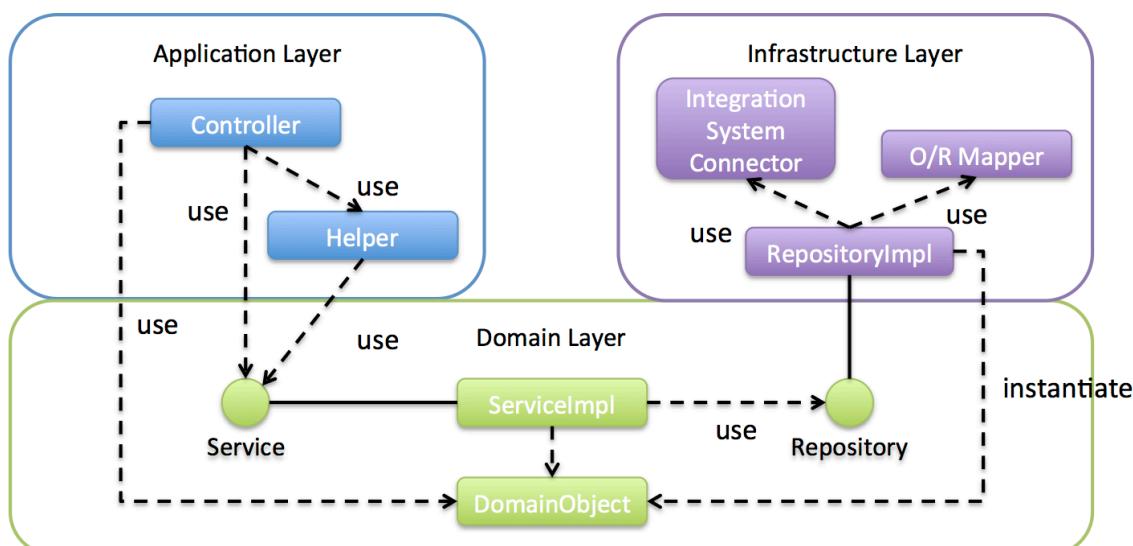
As explained earlier, domain layer is the core of the application, and application layer and infrastructure layer are dependent on it.

In this guideline, it is assumed that,

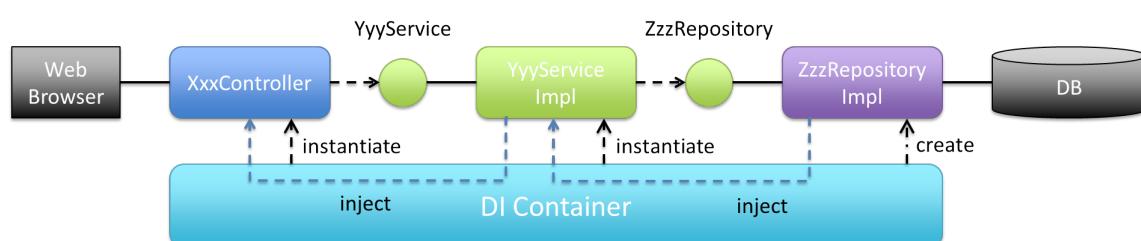
- Spring MVC is used in application layer
- Spring Data JPA and MyBatis are used in infrastructure layer

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.

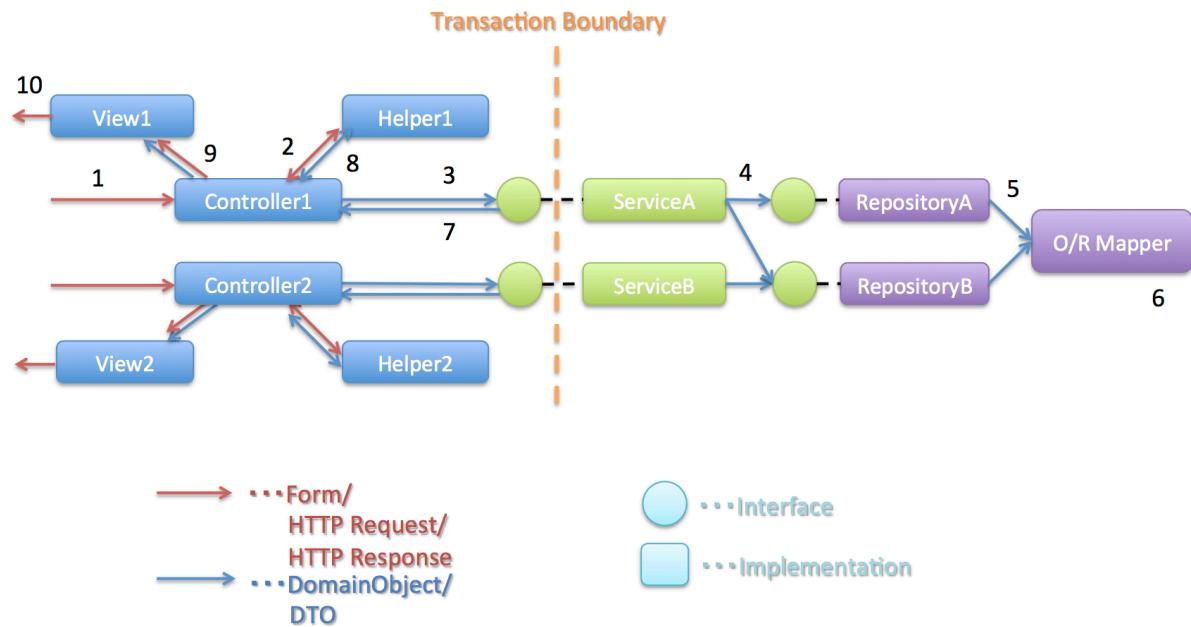


Object dependency in each layer can be resolved by DI container.



Processing and data flow with Repository

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



Sequence is explained using the example of update operation.

No.	Description
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.**

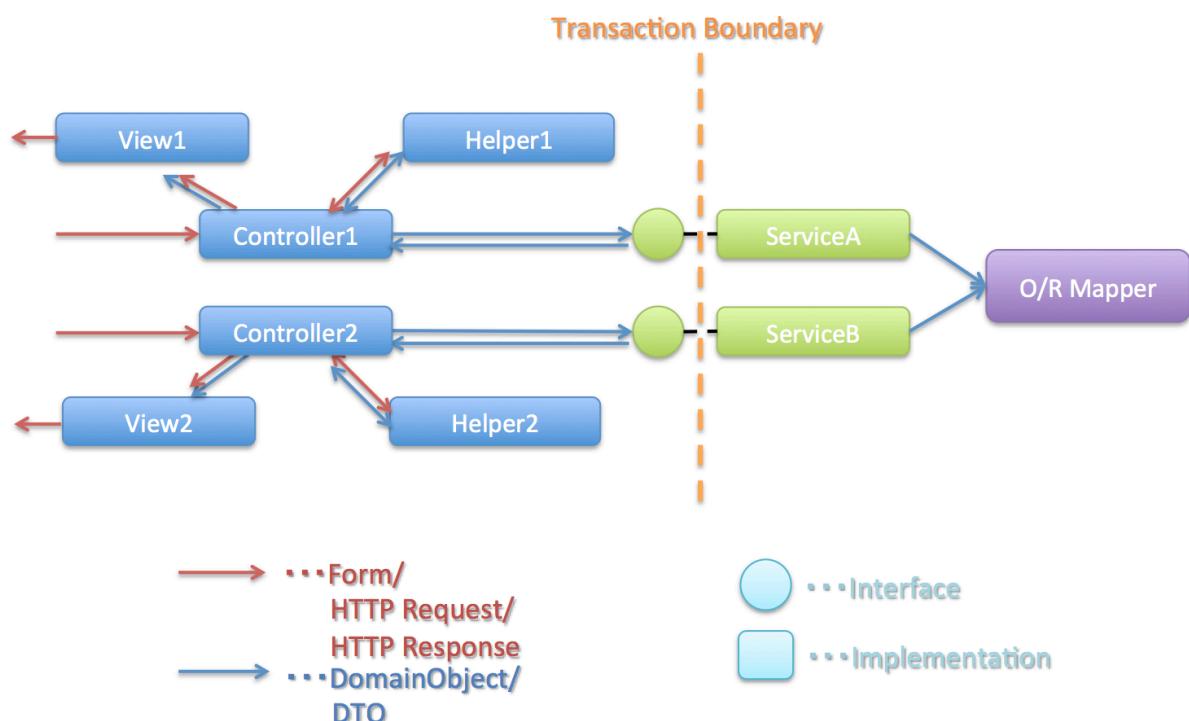
If services that can be used even from other services are required, SharedService should be created in order to clarify such a possibility. Refer to [Domain Layer Implementation](#) for the details.

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 becomes 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.

Processing and data flow without Repository

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.

Project Name	Description
[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 ... (2)
        |           |   Xxx.java
        |           |   Yyy.java
        |           |   Zzz.java
        |           repository ... (3)
        |           |   xxx
        |           |     |   XxxRepository.java
        |           |     YYY
        |           |     |   YyyRepository.java
        |           |     zzz
        |           |     |   ZzzRepository.java
        |           |     ZzzRepositoryImpl.java
        |           service ... (4)
        |             aaa
        |               |   AaaService.java
        |               |   AaaServiceImpl.java
        |             bbb
        |               BbbService.java
        |               BbbServiceImpl.java
    resources
      META-INF
        spring
          [projectName]-codelist.xml ... (5)
          [projectName]-domain.xml ... (6)
          [projectName]-infra.xml ... (7)
```

No.	Details
(1)	Package to store configuration elements of domain layer.
(2)	Package to store DomainObjects classes.
(3)	Package to 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(holds such as search criteria) 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.
(4)	Package to store Service classes. 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.
(5)	Bean definition for CodeList.
(6)	Bean definition pertaining to domain layer.
(7)	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
  resources ... (7)
  WEB-INF
    views ... (8)
      |   abc
      |   |   list.jsp
      |   |   createForm.jsp
      |   def
      |     list.jsp
      |     createForm.jsp
    web.xml ... (9)
```

No,	Details
(1)	Package to store configuration elements of application layer.
(2)	Bean defined 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 static resources(css, js, image, etc)
(8)	Store View(jsp) files.
(9)	Servlet definitions

[projectName]-env

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

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

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

No.	Details
(1)	Directory to define configurations depends on the environment for all environments.
(2)	Directory to define configurations depeands on each environment. The directory name is used as the name to identify the environment.
(3)	Directory to define configurations depeands on each environment. The sub directory structure and files are same as (4).
(4)	Directory to define configurations depeands on the local development environment.
(5)	Bean definitions that depend on the local development environment (like dataSource etc).
(6)	Property definitions which depend on the local development environment.
(7)	Log output definitions which depend on the local development 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 [*Removing Environment Dependency*](#).

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

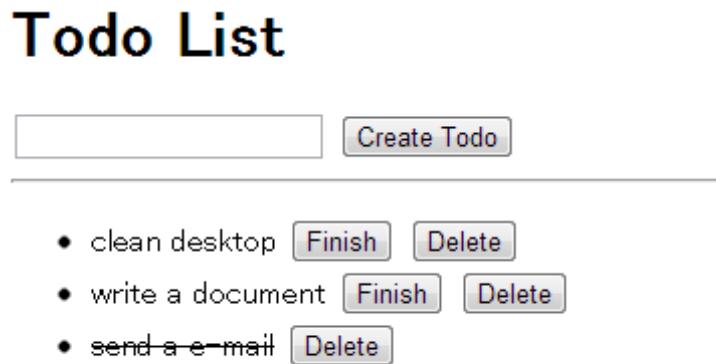
In this tutorial, operations are verified in the following environment.

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

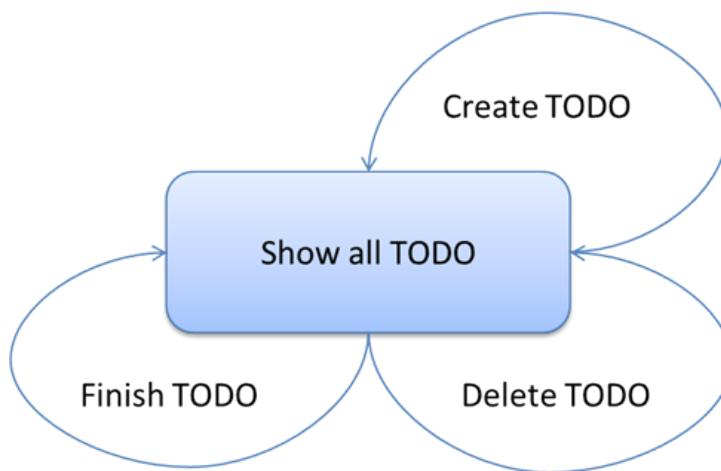
Application to manage TODO list is to be developed. TODO list display, TODO registration, TODO completion and TODO deletion can be performed.



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

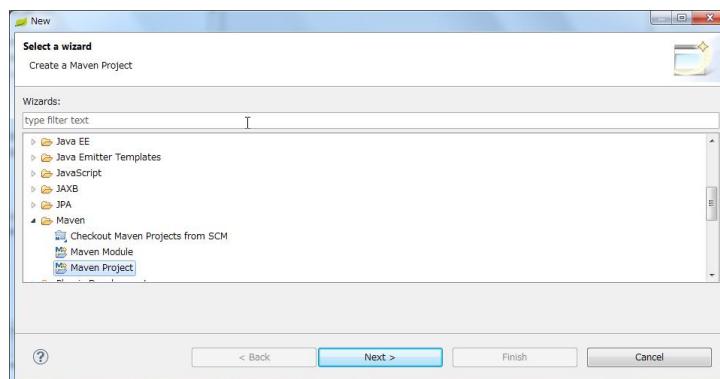
3.2.4 Error message list

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

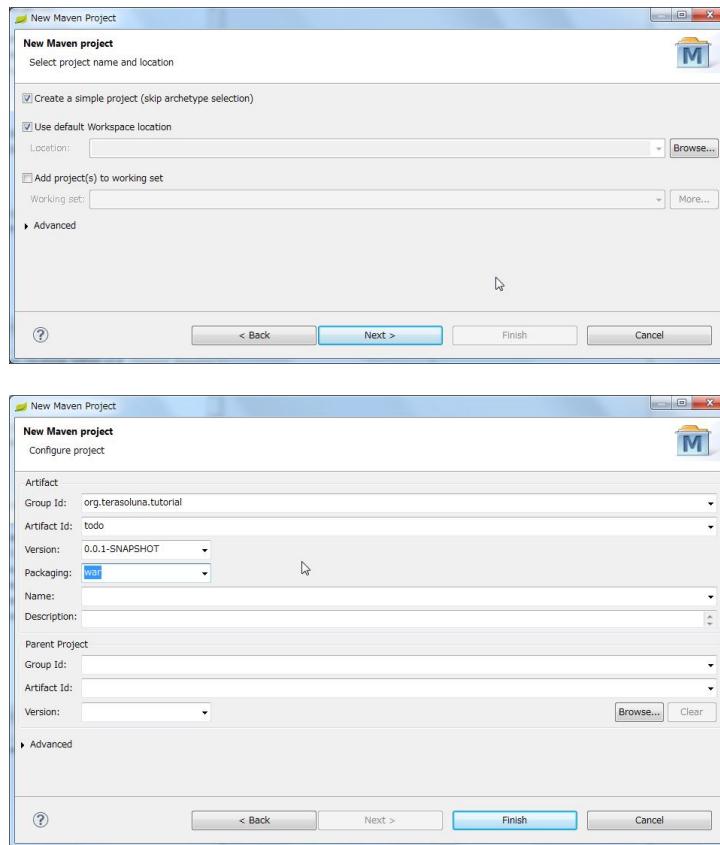
Select File -> Other -> Maven -> Maven Project and proceed to Next



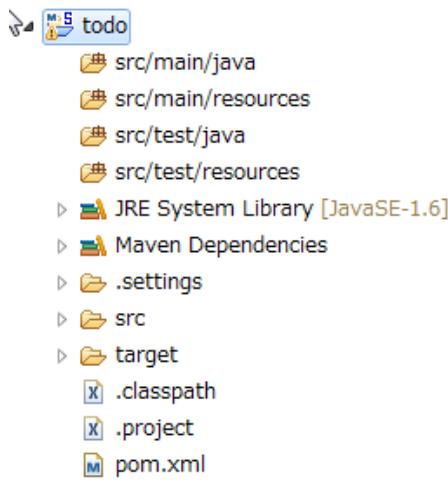
Insert check-mark to Create a simple project and proceed to Next

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

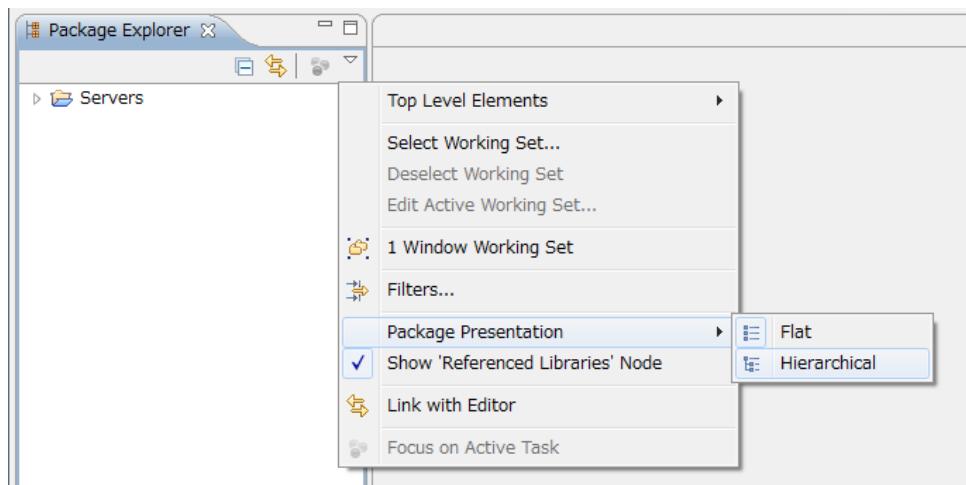
Finish



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-in
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/<url>
    </repository>
    <repository>
        <releases>
            <enabled>false</enabled>
```

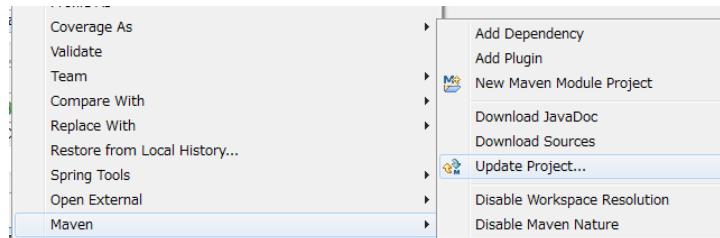
```
</releases>
<snapshots>
  <enabled>true</enabled>
</snapshots>
<id>terasoluna-gfw-snapshots</id>
<url>http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-snapshots/</url>
</repository>
<repository>
  <releases>
    <enabled>true</enabled>
  </releases>
  <snapshots>
    <enabled>false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

▷ JRE System Library [JavaSE-1.6]
▷ Maven Dependencies

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)	<p>Specify parent pom file of TERASOLUNA Global Framework.</p> <p>In this way, even without specifying the version, the library defined in terasoluna-parent can be added to dependency.</p>
(2)	Specify URL of Maven repository for using TERASOLUNA Global Framework.
(3)	Add common library of TERASOLUNA Global Framework to dependency.
(4)	<p>Add the library group recommended by TERASOLUNA Global Framework.</p> <p>Since terasoluna-gfw-recommended-dependencies is just pom file, <type>pom</type> should be mentioned.</p>
(5)	<p>Add the libraries which are recommended by TERASOLUNA Global Framework</p> <p>terasoluna-gfw-recommended-dependencies is just a pom file; hence <type>pom</type> must be specified.</p>
(6)	<p>Add Servlet/JSP API to dependency. Compatiblity with Servlet3 is necessary.</p> <p>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.</p> <p>(Further, though dependency name is tomcat-xxx, but the package of embedded class is javax.servlet, there is no dependency on tomcat)</p>
<hr/>	
<p>Note:</p> <p>When proxy server is used to access the internet, perform the following settings in <HOME>/.m2/settings.xml.</p> <p>(In case of Windows7 C:\Users\<YourName>\.m2\settings.xml)</p> <pre> <settings> <proxies> <proxy> <active>true</active> <protocol>[Proxy Server Protocol (http)]</protocol> <port>[Proxy Server Port]</port> <host>[Proxy Server Host]</host> <username>[Username]</username> <password>[Password]</password> </proxy> </proxies> </settings></pre>	
3.3. Environment creation	67

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.

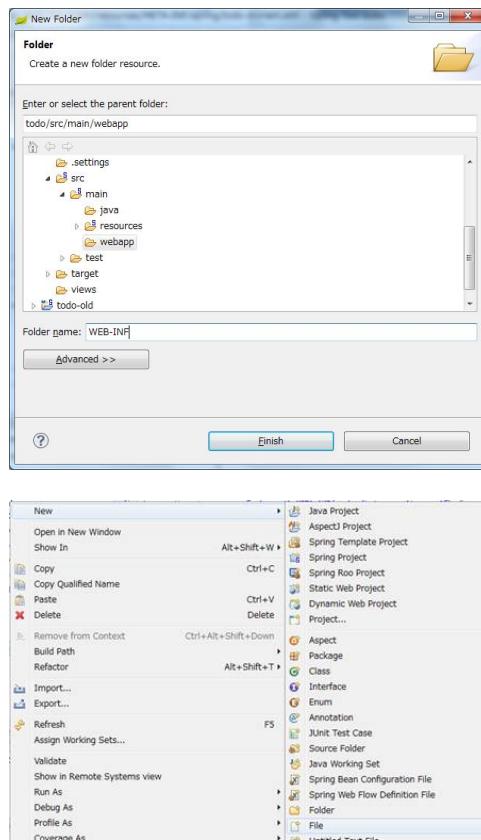
Note: It had been recommended to use a multi-project structure in “*Project Structure*” section of previous chapter . In this tutorial, a single project configuration is used because it focuses on ease of learning. However, when in a real project, multi project configuration is strongly recommended.

3.3.4 Creation of configuration file

web.xml settings

Create `src/main/wepapp/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-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app.xsd"
           version="3.0">
<!-- (2) -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <!-- Root ApplicationContext -->
    <param-value>
        classpath*:META-INF/spring/applicationContext.xml
    </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>
</filter>
```

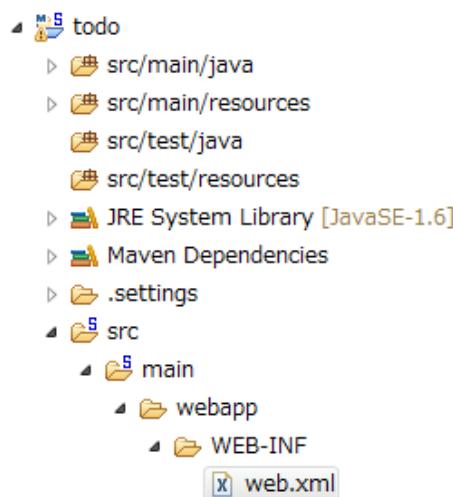
```
</init-param>
<init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
</init-param>
</filter>
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- (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.

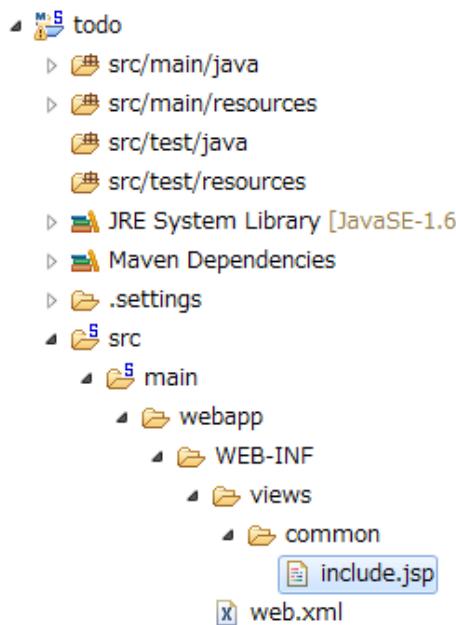
```
<%@ page session="false"%>
<!-- (1) -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt "%>
<!-- (2) -->
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@ taglib uri="http://www.springframework.org/tags/form"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

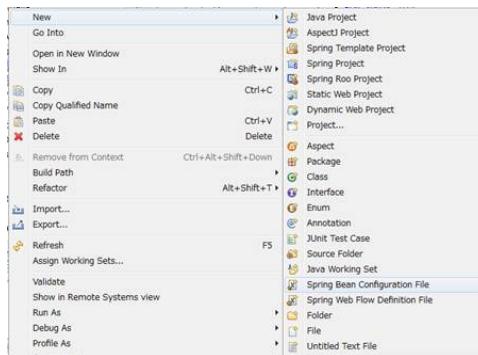


- spring-mvc.xml

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.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context.xsd">

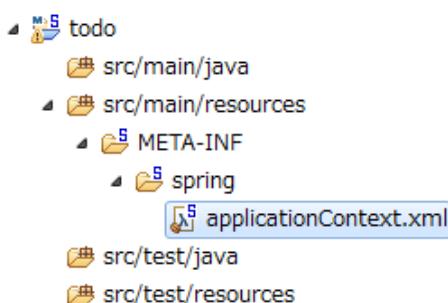
    <!-- (1) -->
    <import resource="classpath:/META-INF/spring/todo-domain.xml" />
```

```
<!-- (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>

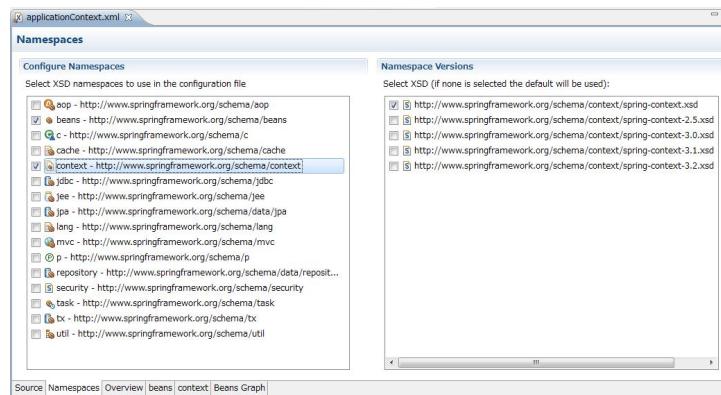
</beans>
```

Sr.No.	Description
(1)	Import Bean definition file related to domain layer.
(2)	Read the settings of property file. Read any property file under <code>src/main/resources/META-INF/spring</code> . Using this setting, it is possible to insert property file value in <code> \${propertyName}</code> format in Bean definition file and to inject using <code>@Value("\${propertyName}")</code> 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 <code>src/main/resources/META-INF/dozer/xxx-mapping.xml</code> format. For the mapping file, refer to Dozer manual .)

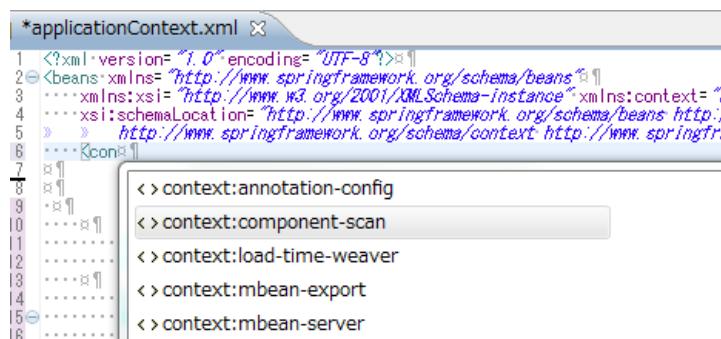


Note: While entering the above contents manually without copying them, open namespace tab and insert

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

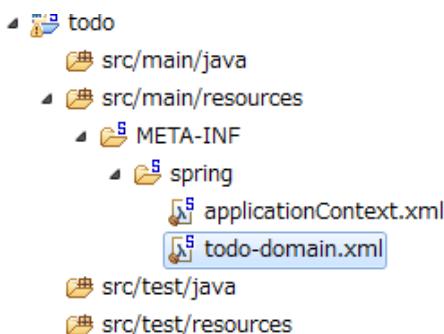
Carry out settings related to domain layer in `src/main/resources/META-INF/spring/todo-domain.xml`.

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.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
                           http://www.springframework.org/schema/context http://www.springframework.org/schema/context">
    <!-- (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.

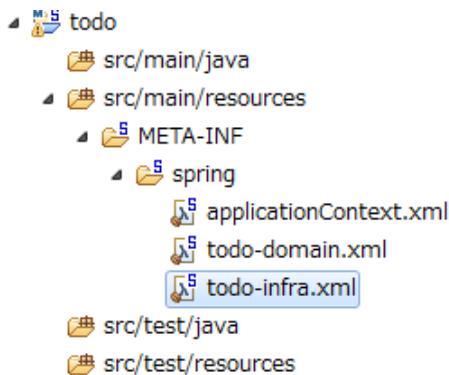


todo-infra.xml

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/
```



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.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/mvc.xsd
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/beans.xsd
        http://www.springframework.org/schema/util http://www.springframework.org/schema/util/util.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/context.xsd">

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

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

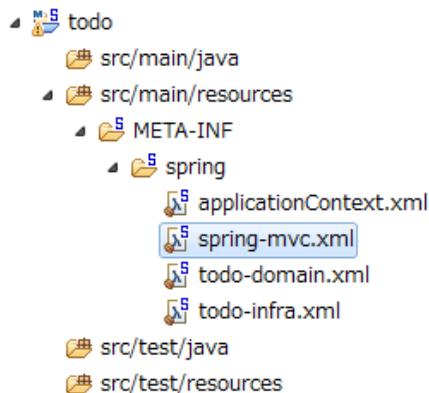
    <!-- (3) -->
    <mvc:resources mapping="/resources/**"
        location="/resources/, classpath:META-INF/resources/"
        cache-period="#{60 * 60}" />

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

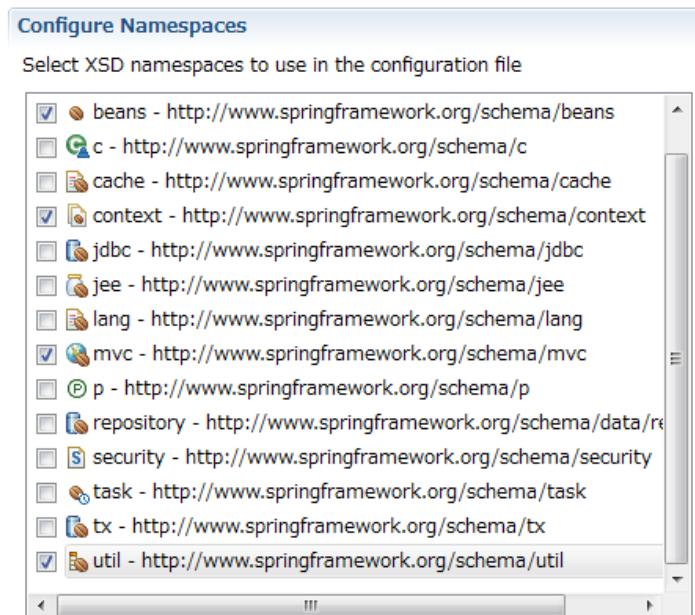
```
        class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>

<!-- (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 <code><contextPath>/resources/css/styles.css</code> , 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 demonstrate 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]</pattern>
        </encoder>
    </appender>

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

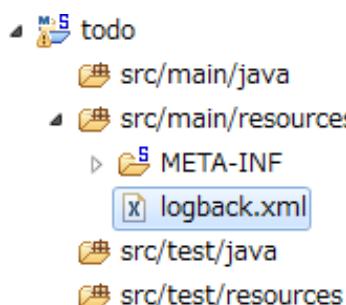
    <!-- TERASOLUNA -->
    <!-- (3) -->
    <logger name="org.terasoluna.gfw">
        <level value="info" />
    </logger>
    <!-- (4) -->
    <logger name="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor">
        <level value="trace" />
    </logger>

    <!-- 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 <code>TraceLoggingInterceptor</code> which is defined in <code>spring-mvc.xml</code> .
(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 <code>org.springframework.web.servlet</code> so that logs valueable to development activity gets output.
(7)	Set so that log level of ‘warn’ and above is output by default.

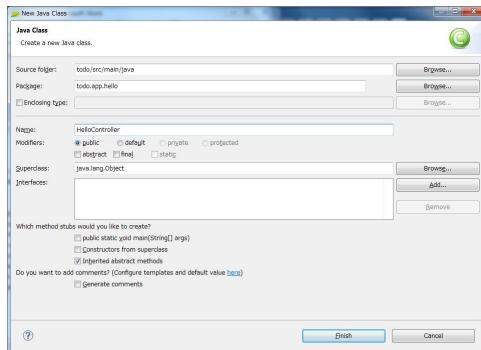


3.3.5 Operation verification

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 <code>@Controller</code> annotation to class level.
(2)	Generate logger. Since logback implements logger and API is SLF4J, <code>org.slf4j.Logger</code> should be used.
(3)	Set mapping of methods for accessing / (root) using <code>@RequestMapping</code> .
(4)	Output debug log. {} is the placeholder.
(5)	For passing date to the screen, add Date object with name <code>now</code> to Model.
(6)	Return hello as view name. Using ViewResolver settings, WEB-INF/views/hello.jsp is output.

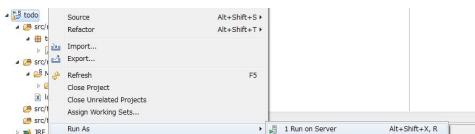
Next, create view(jsp). Create src/main/webapp/WEB-INF/views/hello.jsp as follows.

```

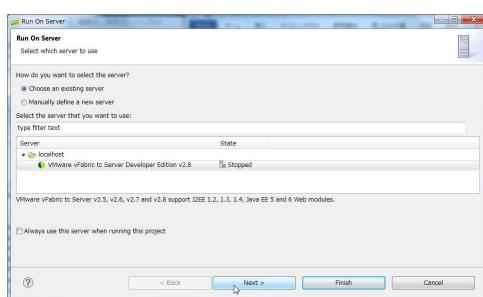
<!DOCTYPE html>
<html>
<head>
<title>Hello World!</title>
</head>
<body>
    <h1>Hello World!</h1>
    <p>
        Today is
        <!-- (1) -->
        <fmt:formatDate value="${now}" pattern="yyyy-MM-dd HH:mm:ss" />
    </p>
</body>
</html>
```

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

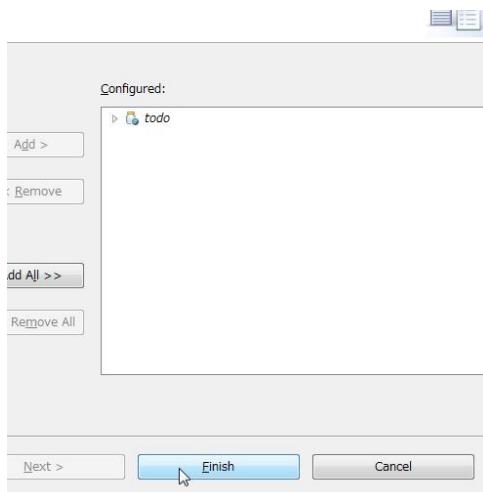
Right click package project name todo and click Run As -> Run on Server



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.DispatcherSer...
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.DispatcherSer...
```

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.

Hello World!

Today is 2013-06-14 15:40:59

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.

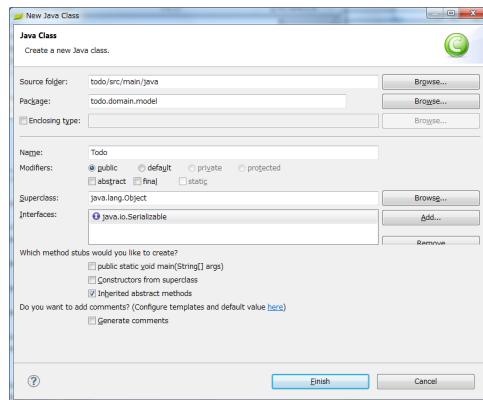
Package:	todo.domain.model
Name:	Todo
Interfaces:	java.io.Serializable

```
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;
}

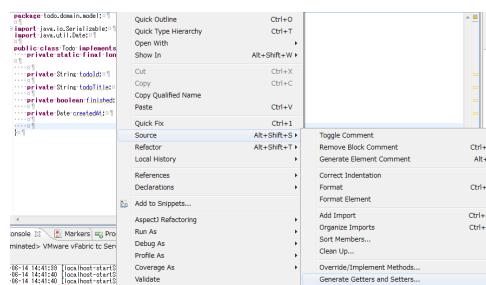
public Date getCreatedAt() {
    return createdAt;
}

public void setCreatedAt(Date createdAt) {
    this.createdAt = createdAt;
}
```

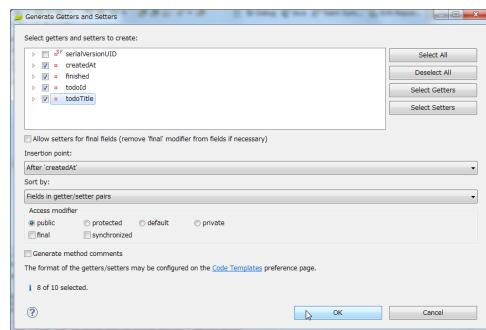
}



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;

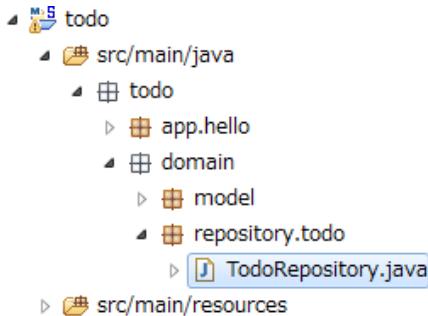
public interface TodoRepository {
    Todo findOne(String todoId);

    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.

@Repositoryannotation 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();
    }

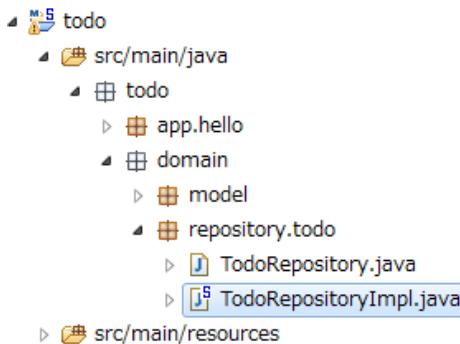
    @Override
    public Todo save(Todo todo) {
        return TODO_MAP.put(todo.getTodoId(), todo);
    }

    @Override
    public void delete(Todo todo) {
        TODO_MAP.remove(todo.getTodoId());
    }

    @Override
    public long countByFinished(boolean finished) {
        long count = 0;
        for (Todo todo : TODO_MAP.values()) {
            if (finished == todo.isFinished()) {
                count++;
            }
        }
        return count;
    }
}
```

Sr.No.	Description
(1)	To consider Repository as component scan target, add @Repositoryannotation 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.servic.todo.TodoService`.

```

package todo.domain.service.todo;

import java.util.Collection;
    
```

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

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

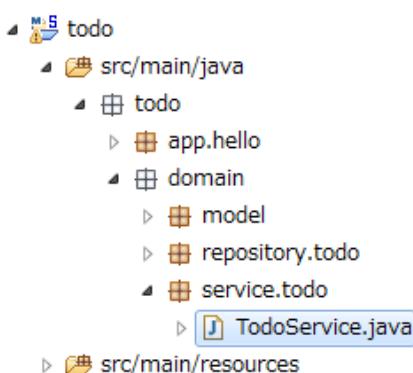
    Todo create(Todo todo);

    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;
    }

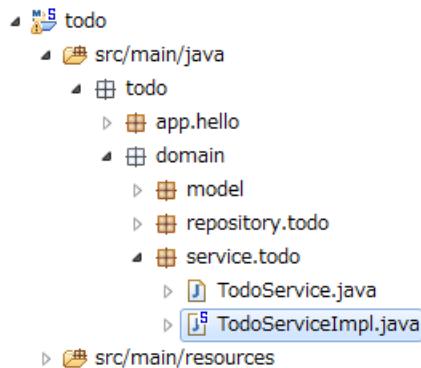
    @Override
    public Todo finish(String todoId) {
        Todo todo = findOne(todoId);
        if (todo.isFinished()) {
            ResultMessages messages = ResultMessages.error();
            messages.add(ResultMessage
                .fromText("[E002] The requested Todo is already finished. (id="
                + 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 <code>@Service</code> 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, <code>@Transactional</code> should be added at class level. It is described in <i>Change of infrastructure layer</i> .
(3)	Inject TodoRepository implementation using <code>@Inject</code> .
(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 <code>org.terasoluna.gfw.common.message.ResultMessage</code> provided in common library, as a class that stores result messages. Currently, for throwing error message, <code>ResultMessage</code> is added by specifying message type using <code>ResultMessages.error()</code> .
(6)	When target data is not found, <code>org.terasoluna.gfw.common.exception.ResourceNotFoundException</code> provided in common library is thrown.
(7)	When business error occurs, <code>org.terasoluna.gfw.common.exception.BusinessException</code> 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 [Properties 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.

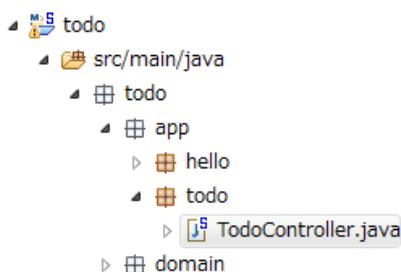
```
package todo.app.todo;

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 <code>modelAttribute</code> attribute. contextPath to be specified in <code>action</code> attribute can be fetched in <code> \${pageContext.request.contextPath}</code>
(2)	Bind form property using <form:input> tag. Property name of form which is specified in <code>modelAttribute</code> should match with the value of <code>path</code> attribute.
(3)	Display entire list of Todo using <c:forEach> tag.
(4)	Determine whether to decorate text using strikethrough(<code>text-decoration: line-through;</code>) 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 <code>f:h()</code> 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.

Todo List



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
    TodoService todoService;

    // (8)
    @Inject
    Mapper beanMapper;

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

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

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

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

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

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

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

}
```

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 java.io.Serializable;

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

public class TodoForm implements Serializable {
    private static final long serialVersionUID = 1L;

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

    public String getTodoTitle() {
        return todoTitle;
    }

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

```

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 displaying 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.

Todo List

[Read a book](#) [Create Todo](#)

Todo List

- Created successfully!

-
- Read a book

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

Todo List

- [E001] The count of un-finished Todo must not be over 5.

-
- aaaaa
 - bbbb
 - dd
 - e
 - ccc

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

Todo List

 size must be between 1 and 30

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.

Todo List

A screenshot of a web application titled "Todo List". At the top, there is a green banner with the text "Created successfully!". Below the banner is a text input field and a "Create Todo" button. A horizontal line separates this from the main list area. In the list area, there is a single item: "• Read a book".

Todo List

A screenshot of a web application titled "Todo List". At the top, there is a red banner with the text "[E001] The count of un-finished Todo must not be over 5.". Below the banner is a text input field with the value "eeee" and a "Create Todo" button. A horizontal line separates this from the main list area. In the list area, there are five items: "• cccc", "• dddd", "• Read a book", "• bbbb", and "• aaaaaa".

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.

Todo List

A screenshot of a web application titled "Todo List". At the top, there is a red banner with the text "size must be between 1 and 30". Below the banner is a text input field with the value "eeee" and a "Create Todo" button. A horizontal line separates this from the main list area. There are no items in the list area.

Finish TODO

Add Finish button to List display screen. If the form is submitted, 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 java.io.Serializable;

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

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

```

public static interface TodoFinish {
};

private static final long serialVersionUID = 1L;

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

// (5)
@NotNull(groups = { TodoCreate.class })
@Size(min = 1, max = 30, groups = { TodoCreate.class })
private String todoTitle;

public String getTodoId() {
    return todoId;
}

public void setTodoId(String todoId) {
    this.todoId = todoId;
}

public String getTodoTitle() {
    return todoTitle;
}

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

}

```

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

import todo.app.todo.TodoForm.TodoCreate;
import todo.app.todo.TodoForm.TodoFinish;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

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

    @Inject
    Mapper beanMapper;

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

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

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

```
public String create(
    @Validated({ Default.class, TodoCreate.class }) TodoForm todoForm, // (16)
    BindingResult bindingResult, Model model,
    RedirectAttributes attributes) {

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

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

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

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

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

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

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

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.

Todo List

- Read a book

Todo List

- Finished successfully!

- Read a book

```
<!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 java.io.Serializable;

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

public class TodoForm implements Serializable {
    public static interface TodoCreate {
    }

    public static interface TodoFinish {
    }

    // (6)
    public static interface TodoDelete {
    }

    private static final long serialVersionUID = 1L;

    // (7)
    @NotNull(groups = { TodoFinish.class, TodoDelete.class })
    private String todoId;

    @NotNull(groups = { TodoCreate.class })
    @Size(min = 1, max = 30, groups = { TodoCreate.class })
    private String todoTitle;

    public String getTodoId() {
        return todoId;
    }

    public void setTodoId(String todoId) {
        this.todoId = todoId;
```

```
}

public String getTodoTitle() {
    return todoTitle;
}

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

}
```

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.TodoDelete;
import todo.app.todo.TodoForm.TodoCreate;
import todo.app.todo.TodoForm.TodoFinish;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;
```

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

    @Inject
    Mapper beanMapper;

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

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

    @RequestMapping(value = "create", method = RequestMethod.POST)
    public String create(
        @Validated({ Default.class, TodoCreate.class }) TodoForm todoForm,
        BindingResult bindingResult, Model model,
        RedirectAttributes attributes) {

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

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

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

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

    @RequestMapping(value = "finish", method = RequestMethod.POST)
    public String finish(
        @Validated({ Default.class, TodoFinish.class }) TodoForm form,
        BindingResult bindingResult, Model model,
        RedirectAttributes attributes) {
```

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

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

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

@RequestMapping(value = "delete", method = RequestMethod.POST)
public String delete(
    @Validated({ Default.class, TodoDelete.class }) TodoForm form,
    BindingResult bindingResult, Model model,
    RedirectAttributes attributes) {

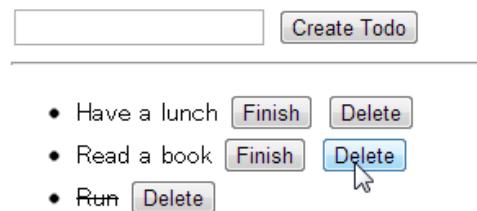
    if (bindingResult.hasErrors()) {
        return list(model);
    }

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

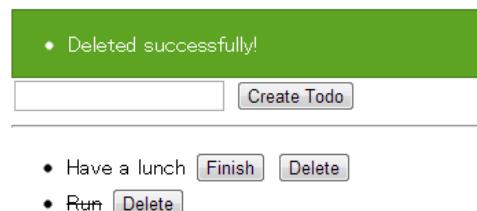
    attributes.addFlashAttribute(ResultMessages.success().add(
        ResultMessage.fromText("Deleted successfully!")));
    return "redirect:/todo/list";
}
```

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

Todo List



Todo List



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.springframework.org/schema/beans.xsd">

    <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.org/schema/beans.xsd">

    <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}" />
    </bean>
</beans>
```

```

<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.properties`

```

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.org/
        http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-
        http://www.springframework.org/schema/context http://www.springframework.org/schema/cont
<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
    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.org/schema/beans.xsd
                        http://www.springframework.org/schema/util http://www.springframework.org/schema/util.xsd
                        http://www.springframework.org/schema/data/jpa http://www.springframework.org/schema/data/jpa.xsd
                        http://www.springframework.org/schema/annotation http://www.springframework.org/schema/annotation.xsd">

    <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.org/schema/beans.xsd">

    <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)	<p>Set transaction manager. <code>id</code> should set to <code>transactionManager</code>.</p> <p>When another name is to be specified, transaction manager name should be specified in <code><tx:annotation-driven></code> tag in <code>todo-domain.xml</code> and <code><jpa:repository></code> tag in <code>todo-infra.xml</code>.</p>

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] [%-48llogger{48}] - %m]]</pattern>
        </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 <code>@Entity</code> showing that it is JPA entity and set the corresponding table name using <code>@Table</code> .
(2)	Add <code>@Id</code> to the field corresponding to primary key column.
(3)	Set the corresponding column name by <code>@Column</code> .
(4)	It has to be clearly specified that Date type corresponds to which of <code>java.sql.Date</code> , <code>java.sql.Time</code> , <code>java.sql.Timestamp</code> . Here <code>Timestamp</code> 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)	<p>Set interface that extends JpaRepository. Specify Entity class (Todo) and primary key type (String) sequentially in Generics parameter.</p> <p>Since basic CRUD operations (findOne, findAll, save, delete etc.) are defined in upper level interface, only <code>countByFinished</code> can be defined in <code>TodoRepository</code>.</p>
(2)	Specify JPQL executed at the time of calling <code>countByFinished</code> by <code>@Query</code> .
(3)	<p>Set bind variable of JPQL specified in (2) by <code>@Param</code>.</p> <p>Here, add <code>@Param("finished")</code> to method argument for inserting '<code>:finished</code>' in JPQL.</p>

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.org/schema/beans/spring-beans.xsd">

    <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.org/schema/beans.xsd">

    <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)	<p>Set transaction manager. id should be set to transactionManager.</p> <p>If a separate name is to be specified, transaction manager name should be specified even in <tx:annotation-driven> tag.</p>

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]]</pattern>
        </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>

```

```
</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
    QueryDAO queryDAO;

    @Inject
    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)	<p>Manage all the transactions of public methods by adding <code>@Transactional</code> at class level.</p> <p>Since settings are made even at Service side that calls the Repository, the transactions can be managed even without adding <code>@Transactional</code>.</p> <p>However, since the <code>propagation</code> attribute is <code>REQUIRED</code> by default, internal (Repository side) transactions take part in external (Service side) transactions when the transactions are nested.</p>
(2)	Inject QueryDAO and UpdateDAO using <code>@Inject</code> .
(3)	Implementing repository methods involves passing of SQLID and parameters to TERASOLUNA DAO.
	Use <code>QueryDAO</code> for reference and <code>UpdateDAO</code> for update. Set SQL corresponding to SQLID as follows.
(4)	<p>Create new and update are executed using <code>save</code> method. In order to determine which process is to be executed, create ‘exists’ method.</p> <p>In this method, the record count of <code>todoId</code> is acquired and whether the record count is greater than 0 is checked.</p>

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 Spring 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 → *Properties Management*
- To externalize the messages → *Message Management*
- To add paging → *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 will 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.	<i>Implementation of Entity</i>	Creation of classes (Entity class) to hold business data.
2.	<i>Implementation of Repository</i>	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.	<i>Implementation of Service</i>	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.

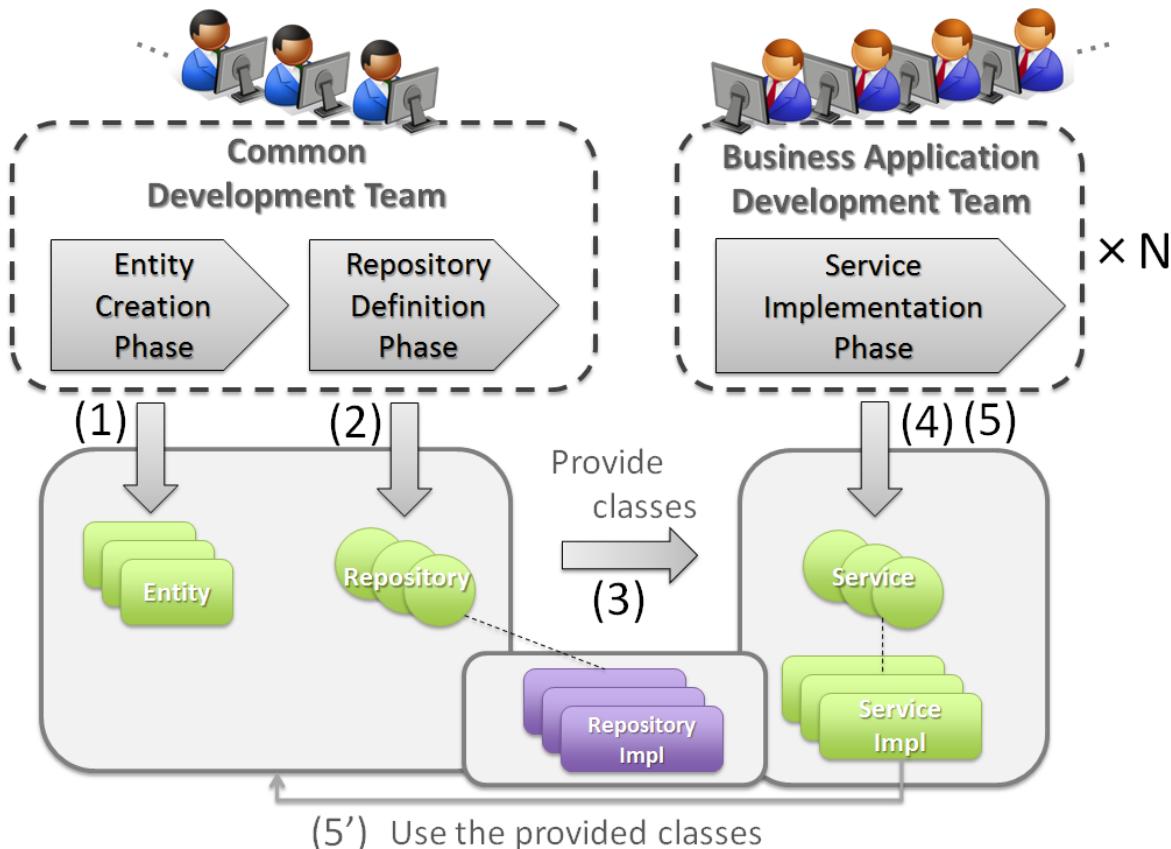
Note: Though this guideline recommends a structure to create Entity classes and Repository, it is not mandatory to perform development in this structure.

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	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 based on the Entity classes and Repository provided by (3).

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 relationship between the tables. Further, when the tables are not normalized, Entity class for each table rule may not be applicable. Refer to the <i>Warning as well as Note outside this table</i> 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 <code>java.util.List<E></code> or <code>java.util.Set<E></code> . The Entity corresponding to the FK destination table is called as the related Entity in this guideline.
3.	Treat the code related tables as <code>java.lang.String</code> 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.

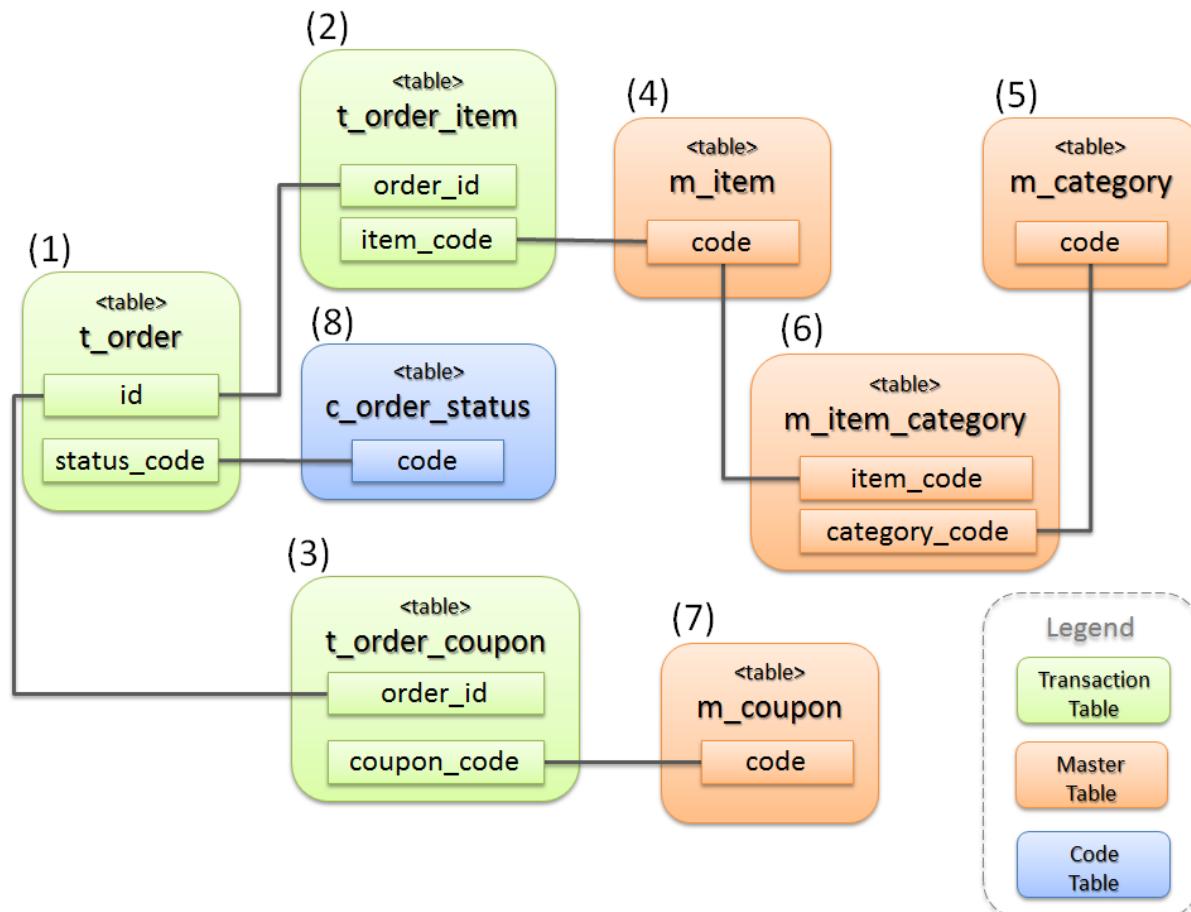
Example of creating Entity class

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

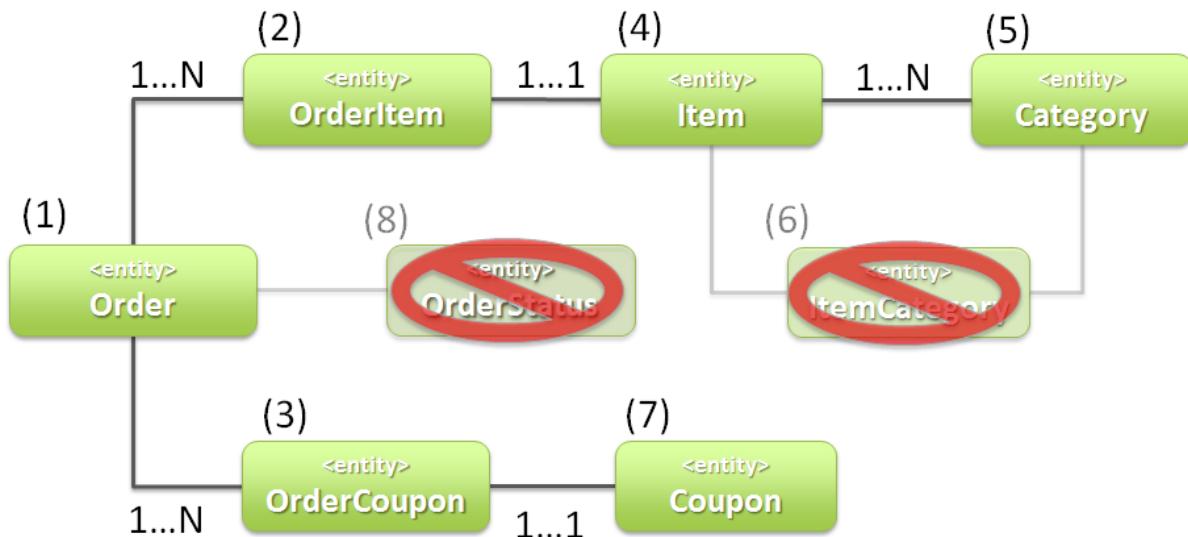
The table structure is as given below:



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_item_category	Master table to define the category of the product.

Entity structure

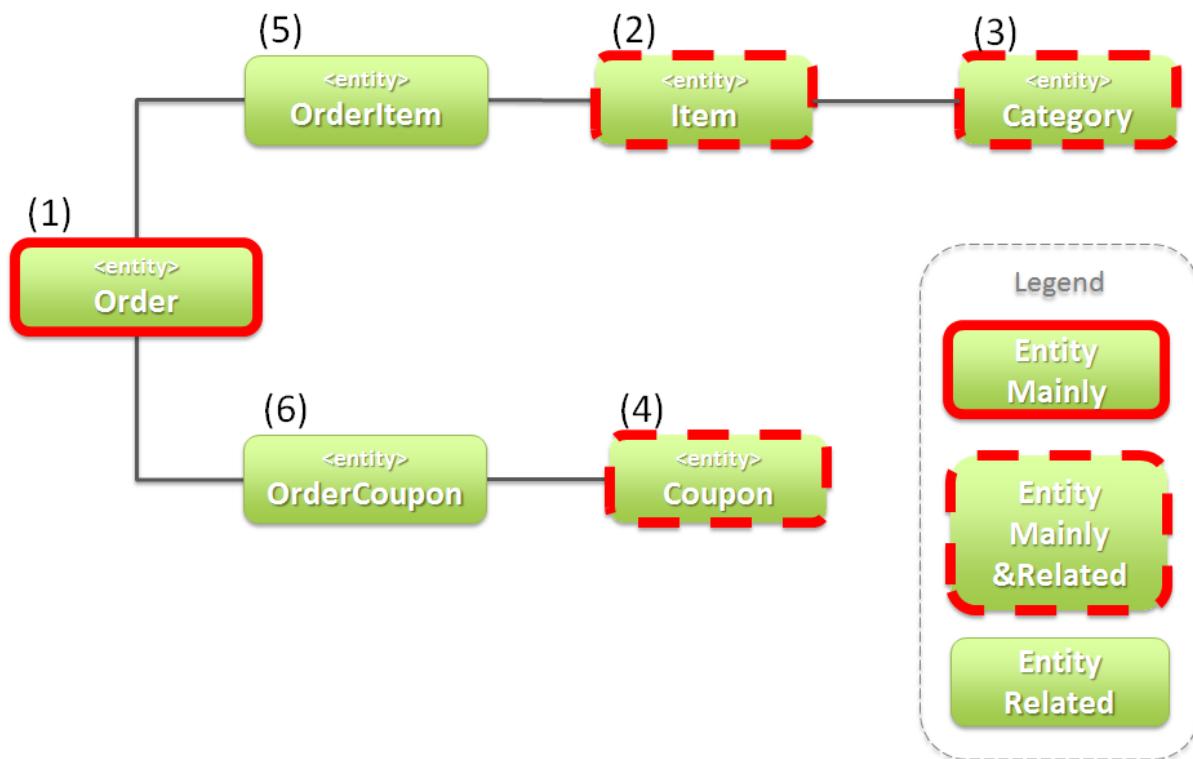
If Entity classes are created with the help of policy defined by the above table, it results into the following structure.



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.

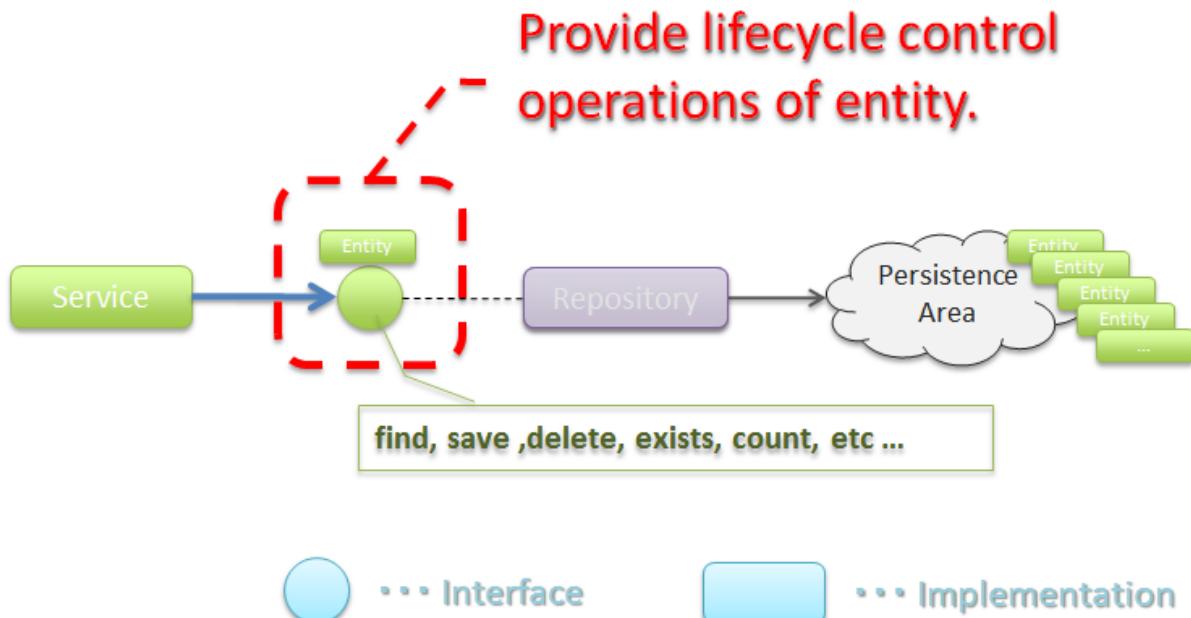
4.1.4 Implementation of Repository

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.

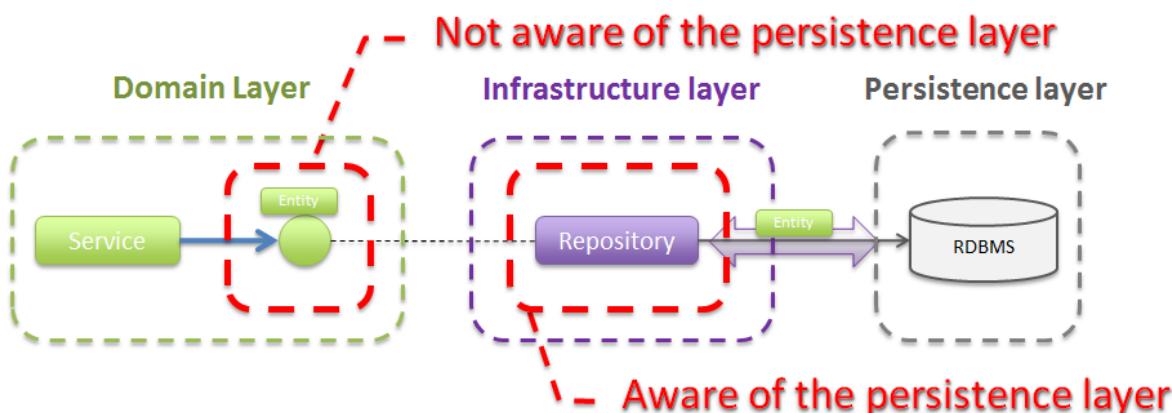
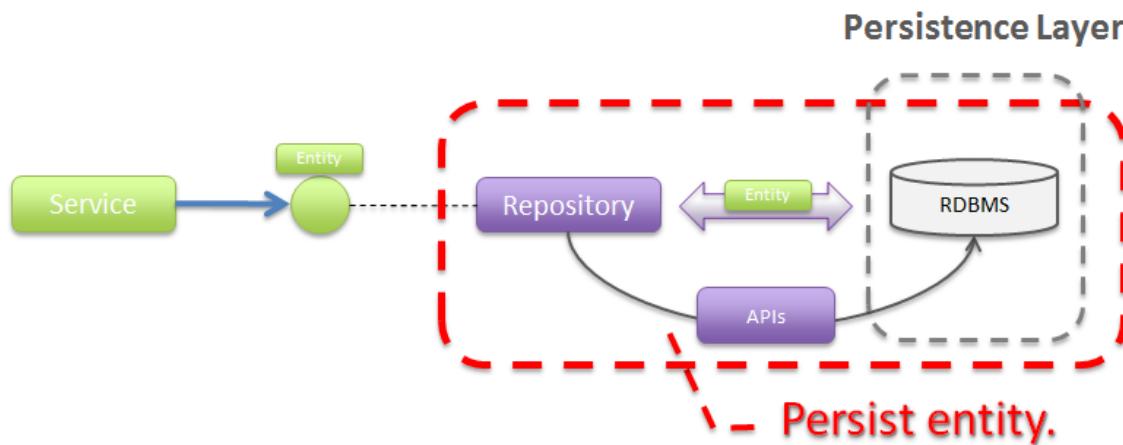
Mostly relational database is the permanent destination of Entity. However, NoSQL database, cache server, external system and file (shared disk) can also be the permanent destination.

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

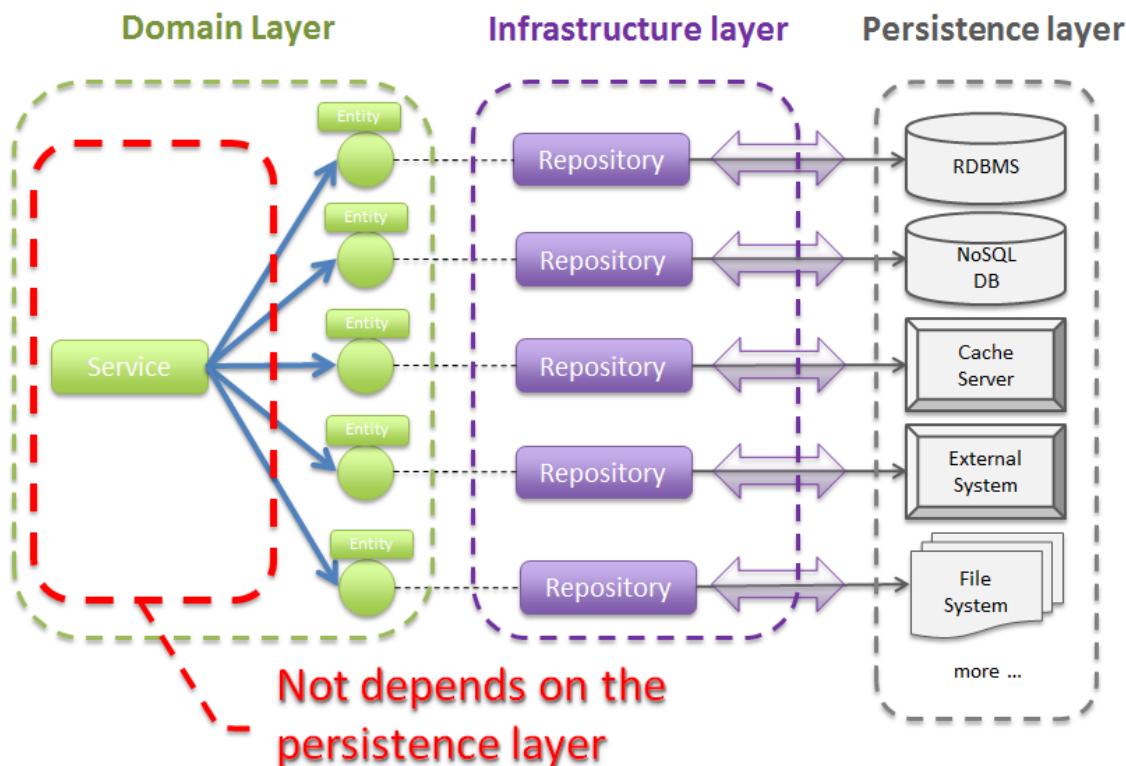
Repository consists of Repository interface and RepositoryImpl and performs the following roles.



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.
156	4 Application Development using TERASOLUNA Global Framework		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 cases 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 the 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.

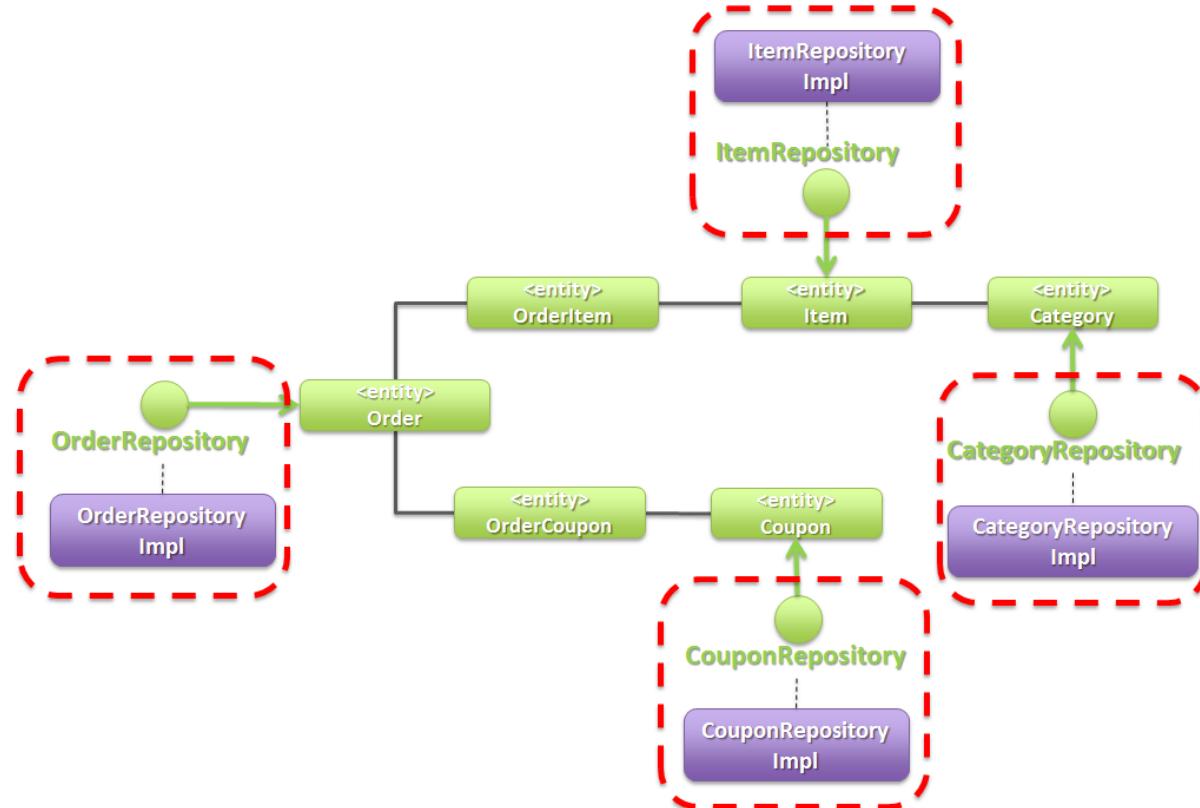
Example of creating Repository

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* 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 <code>java.util.Iterable</code> . Here as a sample, it is set to <code>java.util.List</code> .
(4)	Method to fetch collection of Entity objects corresponding to the specified pagination information (start position, record count, sort information). <code>Pageable</code> and <code>Page</code> 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 SimpleCrudRepositoryinterface 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	<ol style="list-style-type: none"> Method name beginning with findOneBy to indicate that this method fetches a single record that matches with the condition. 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”. There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided. Return value must be Entity class.
2.	Method for searching multiple records	<ol style="list-style-type: none"> Method name beginning with findAllBy to indicate that this method fetches all the records that matches with the condition. 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”. There must be an argument for each search condition. However, when there are many conditions, DTO containing all search conditions can be provided. Return value must be collection of Entity class.
3.	Method for searching multiple records with pagination	<ol style="list-style-type: none"> Method name beginning with findPageBy to indicate that this method fetches pages that matches with the condition. 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”. 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). Return value should be Page interface provided by Spring Data.
4.	Count method related	<ol style="list-style-type: none"> Method name beginning with countBy to indicate that this method fetches count of Entities which matches with the condition. Return value must be long type. 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”.
162	4 Application Development using TERASOLUNA Global Framework	<p>name must be such that it becomes possible to estimate “the kind of entity that can be fetched using this method”.</p> <ol style="list-style-type: none"> 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 findByPage.
(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.**

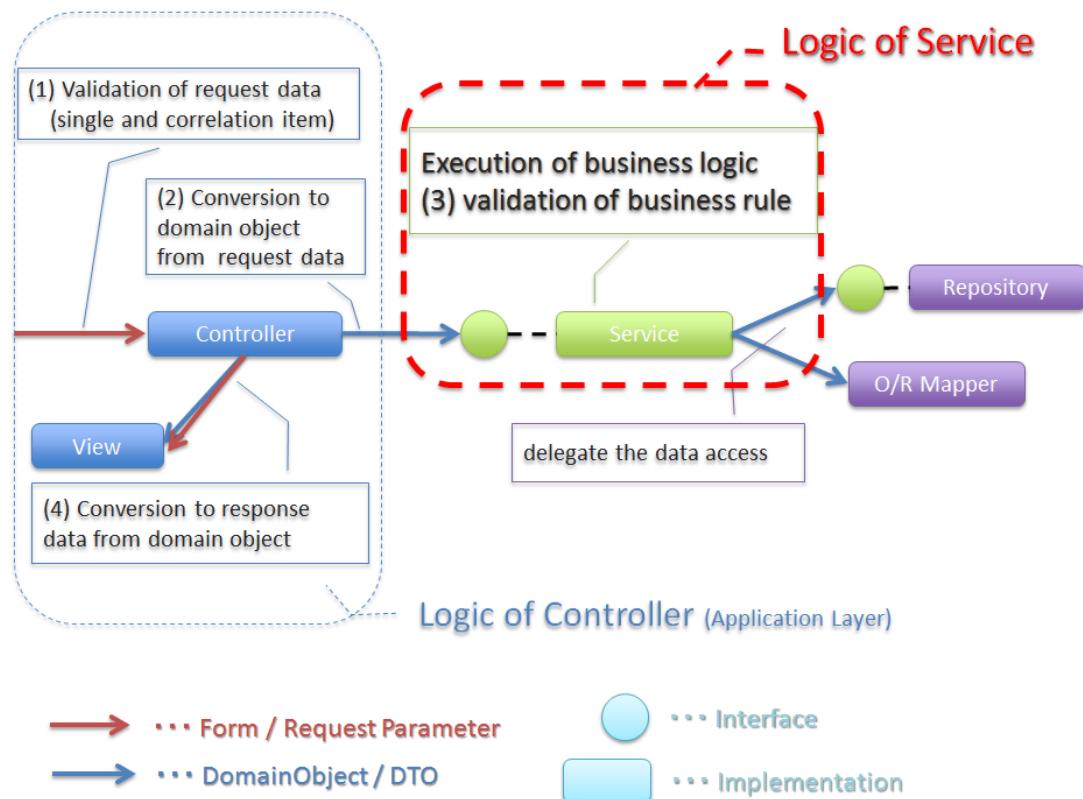
Business logic consists of create, update, consistency check etc of business data as well as all the processes related to business logic.

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).
-

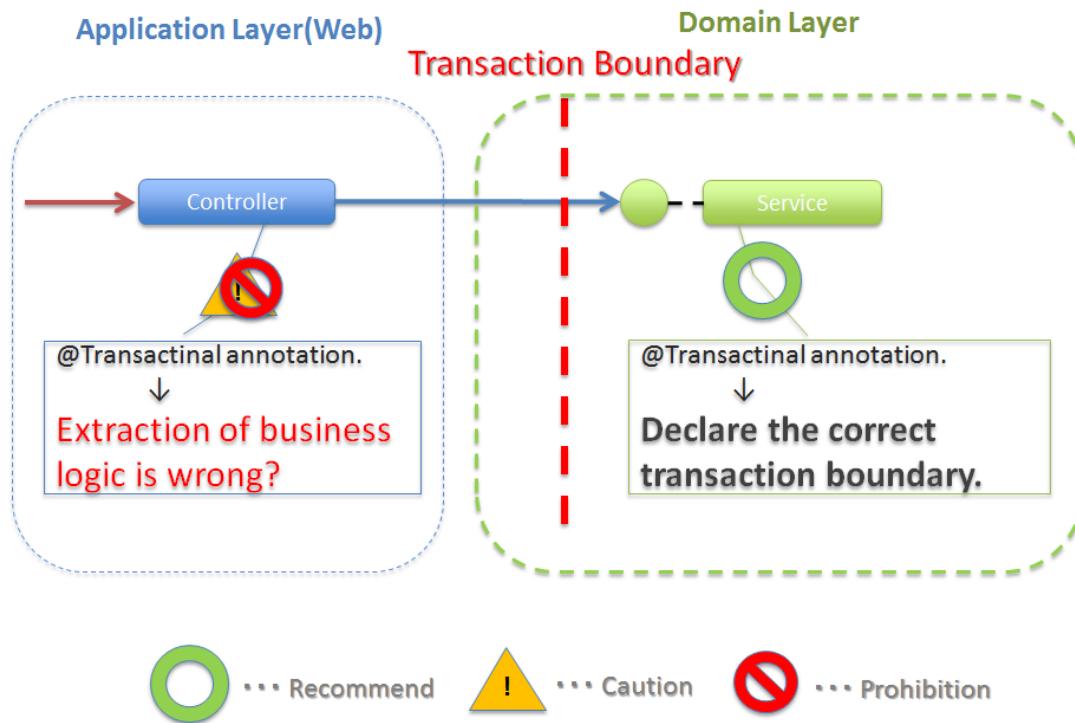


2. Declare transaction boundary.

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.

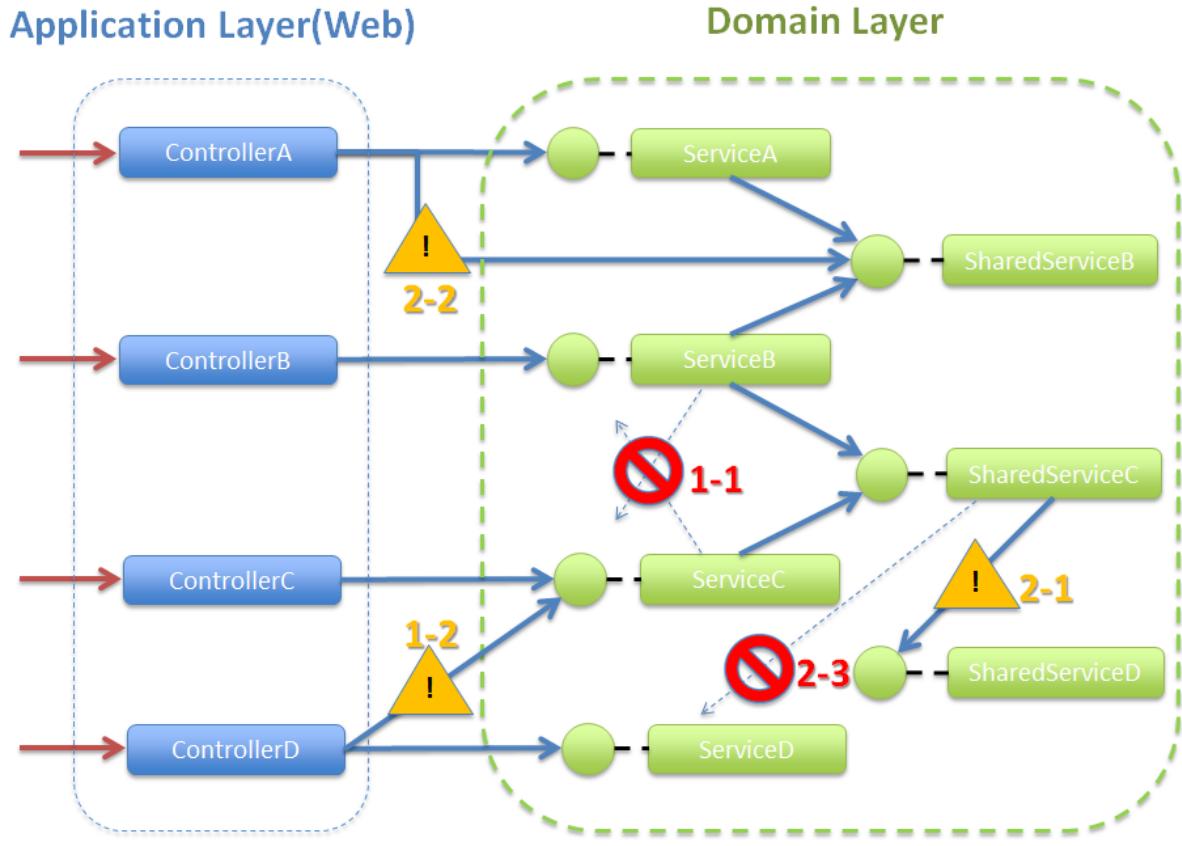
In this guideline, POJO (Plain Old Java Object) having `@Service`annotation is defined as Service or SharedService class.

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	<p>Provides business logic to the specific Controller.</p> <p>Service class methods must not implement logic that need to be reused.</p>	<ol style="list-style-type: none"> It is prohibited to call a method of Service class from another Service class method (Figure 1-1). For shared logic, create SharedService class. 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	<p>Provides shared (reusable) logic for multiple Controllers and Service classes.</p>	<ol style="list-style-type: none"> 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. 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. 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 SharedService

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.	<p>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.</p> <p>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.</p>
2.	<p>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.</p>

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 differences 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 development 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 recommend 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.	<p>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.</p> <p>If service is created in this way, business logic will also be created for each entity and it will become to extract shared logic.</p> <p>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.</p> <p>It can be said that the pattern is suitable when for the developing small or medium sized application.</p>
2.	For each use-case	Create Service paired with the use-case.	<p>If the application is to be designed and implemented with focus on events on the screen, Service should be created in this way.</p> <p>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.</p> <p>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.</p> <p>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.</p>
170	3	4 Application Development using TERASOLUNA Global Framework	<p>If the application is to be designed and implemented with focus on events on the screen,</p>

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.

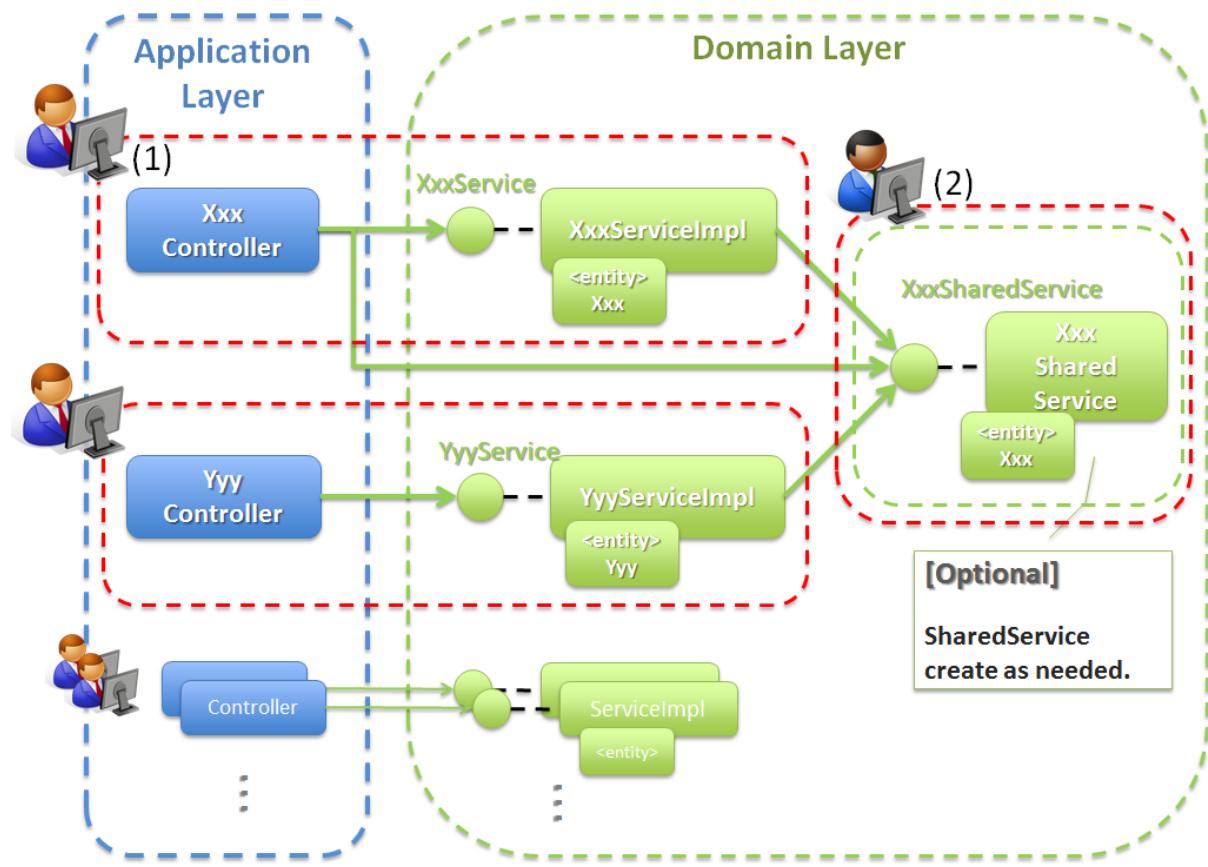
Tip: BLogic is generated directly from design documents when using “TERASOLUNA ViSC”.

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.



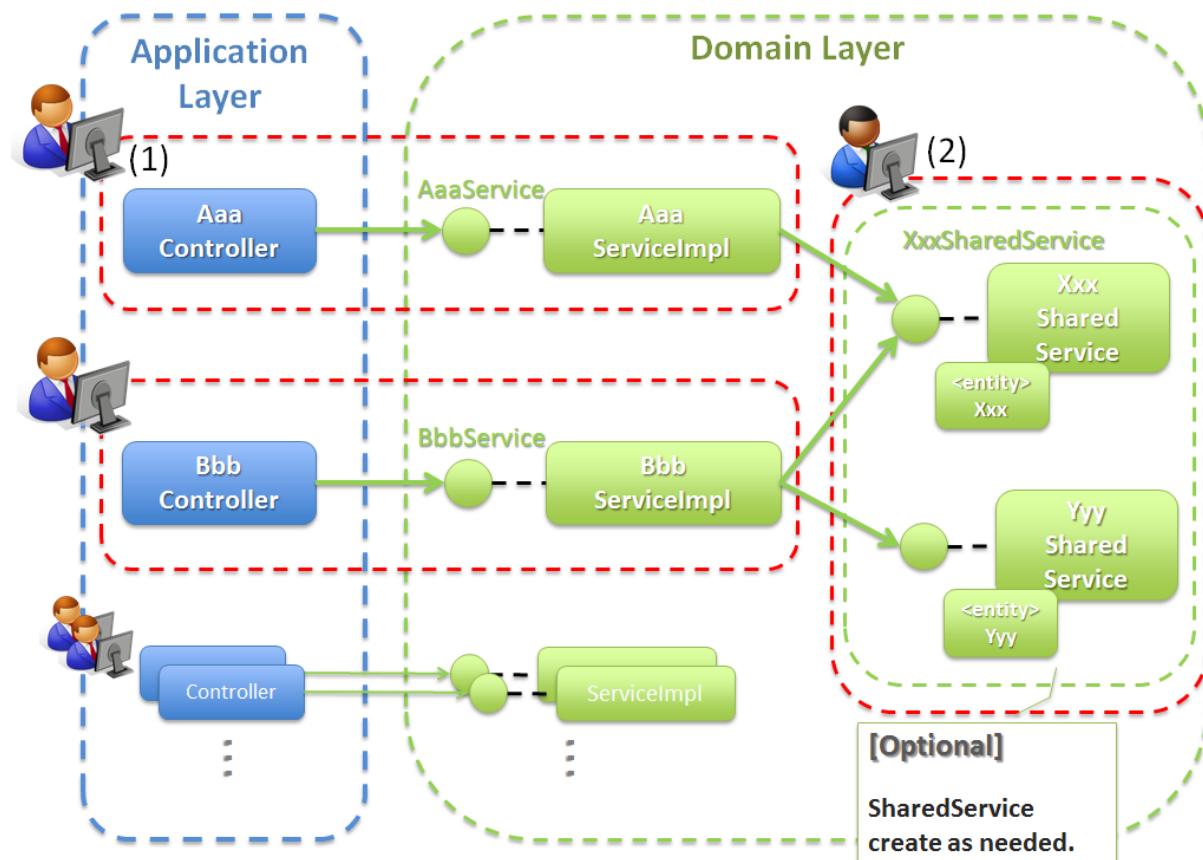
S.No.	Description
(1)	<p>Implement Service by assigning a person for each Entity.</p> <p>If there is no specific reason, it is desirable that Controller must also be created for each Entity and must be developed by the same developer who created the Service class.</p>
(2)	<p>Implement SharedService if there is shared logic between multiple business logics.</p> <p>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.</p>

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.

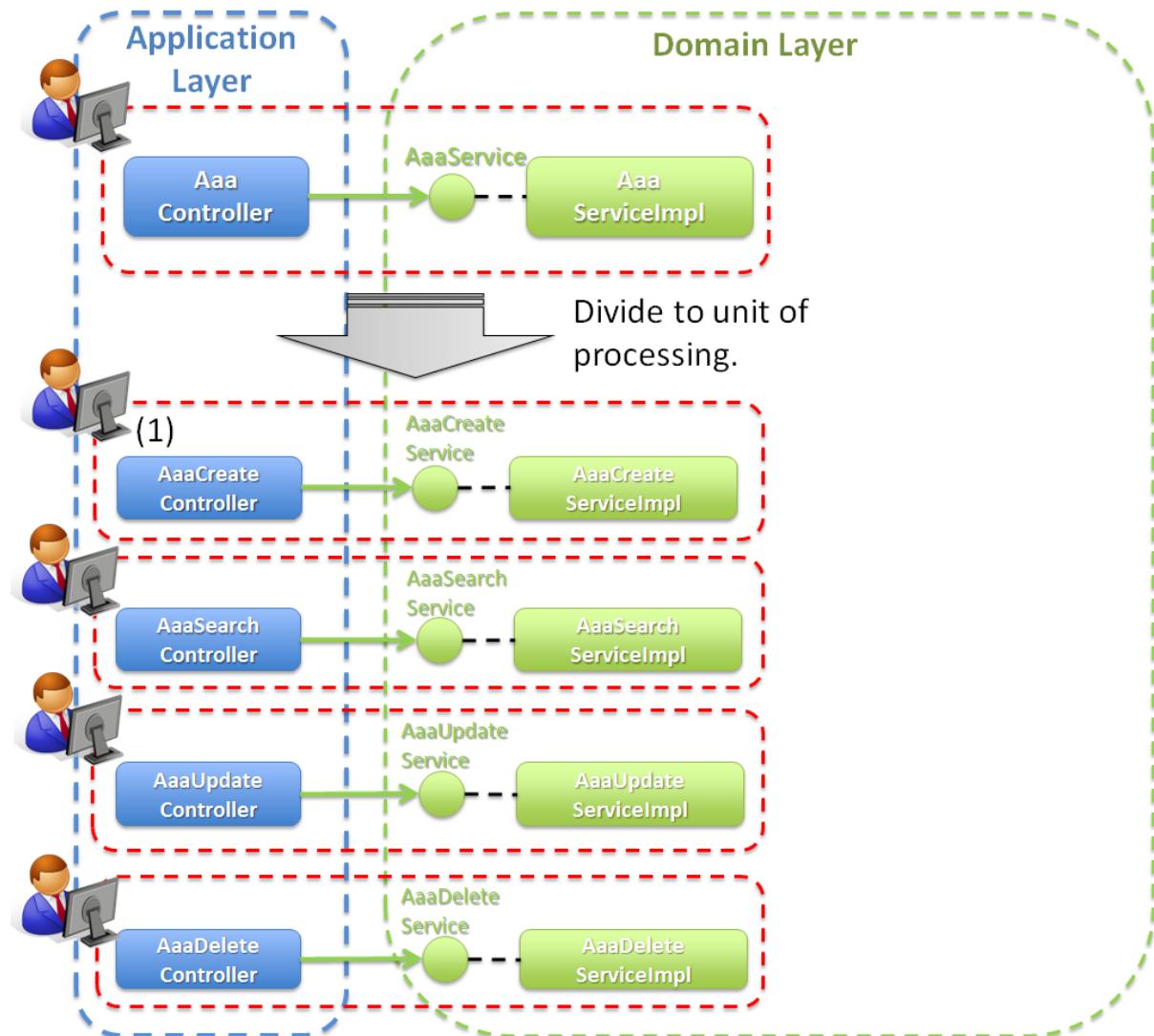


S.No.	Description
(1)	<p>Implement Service by assigning a person for each use-case.</p> <p>If there is no specific reason, it is desirable that Controller must also be created for each use-case and must be developed by the same developer who created the Service class.</p>
(2)	<p>Implement SharedService if there is shared logic between multiple business logics.</p> <p>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.</p>

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.

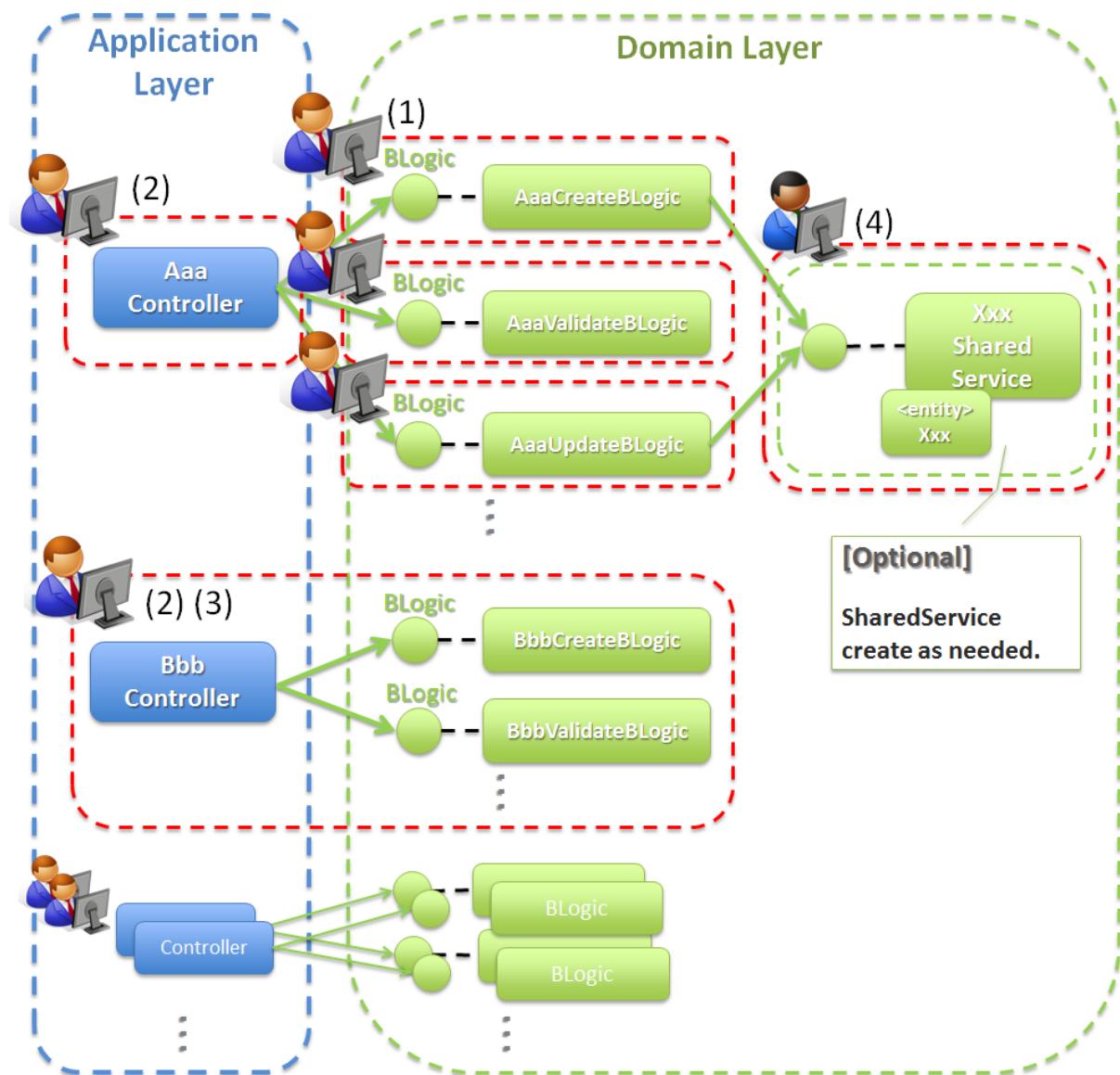


S.No.	Description
(1)	<p>Divide the use-case into finer processes which make-up the complete use-case. Assign each fine process to a developer. Each developer creates the Service for assigned process.</p> <p>Note that the processes here are operations like search, create, update, delete etc. and these processes do not have a direct mapping to the processing required to be done for each event generated on screen.</p> <p>For example, if 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.</p> <p>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.</p>

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 image of application development when creating a Service(BLogic) for each event.



S.No.	Description
(1)	<p>Implement Service(BLogic) by assigning a person for each event.</p> <p>Above example is an extreme case where separate developer is assigned for each service(BLogic).</p> <p>In reality, a single person must be assigned for a use-case.</p>
(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.

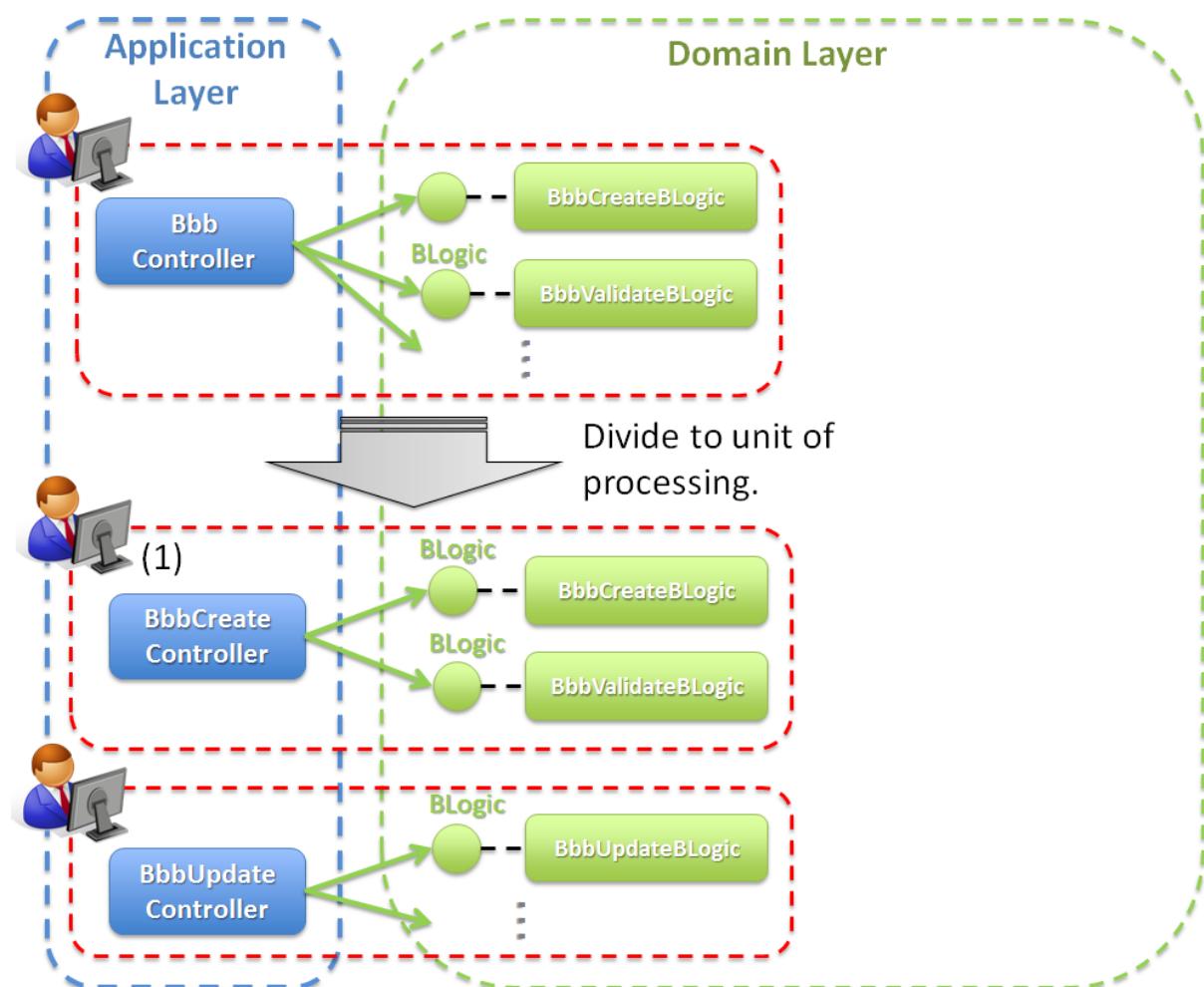
Note:

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)	<p>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.</p> <p>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.</p> <p>For example, if 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.</p> <p>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.</p>

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)	<p>It is recommended to create Service interface.</p> <p>By providing an interface, it is possible to execute the method published as Service explicitly.</p>

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 Document](#) for 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 <i>Information required for “Declarative transaction management”</i> 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 @Transactionalannotation 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)	<p>Add @Transactional annotation for changes to default transaction definition (class level annotation).</p> <p>Attributes should be specified as per the requirement.</p> <p>Refer to <i>Information required for “Declarative transaction management”</i> for details.</p>

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.

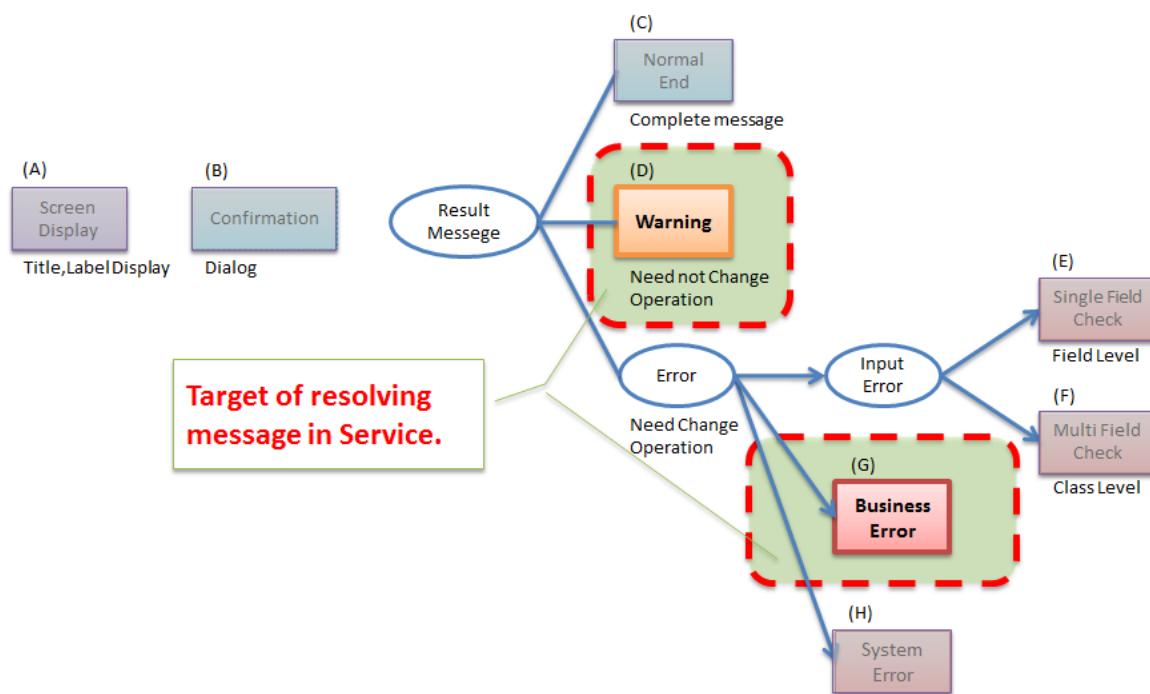
- When using JPA, *Database Access (JPA)*
- When using Mybatis2, *Database access (Mybatis2)*

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 abnormal 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 `@Transactional` 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 [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 Document](#) for 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	<p>Specify transaction propagation method.</p> <p>[REQUIRE] Starts transaction if not started. (default when omitted)</p> <p>[REQUIRES_NEW] Always starts a new transaction.</p> <p>[SUPPORTS] Uses transaction if started. Does not use if not started.</p> <p>[NOT_SUPPORTED] Does not use transaction.</p> <p>[MANDATORY] Transaction should start. An exception occurs if not started.</p> <p>[NEVER] Does not use transaction (never start). An exception occurs if started.</p> <p>[NESTED] save points are set. They are valid only in JDBC.</p>
2	isolation	<p>Specify isolation level of transaction.</p> <p>Since this setting depends on DB specifications, settings should be decided by checking DB specifications.</p> <p>[DEFAULT] Isolation level provided by DB by default.(default when omitted)</p> <p>[READ_UNCOMMITTED] Reads (uncommitted) data modified in other transactions.</p> <p>[READ_COMMITTED] Does not read (uncommitted) data modified in other transactions.</p> <p>[REPEATABLE_READ] Data read by other transactions cannot be updated.</p> <p>[SERIALIZABLE] Isolates transactions completely.</p> <p>Isolation level of transaction is considered as the parameter related to exclusion control.</p>
190	4	<p>Application Development using TERASOLUNA Global Framework Refer to Exclusive Control for exclusion control.</p>
	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

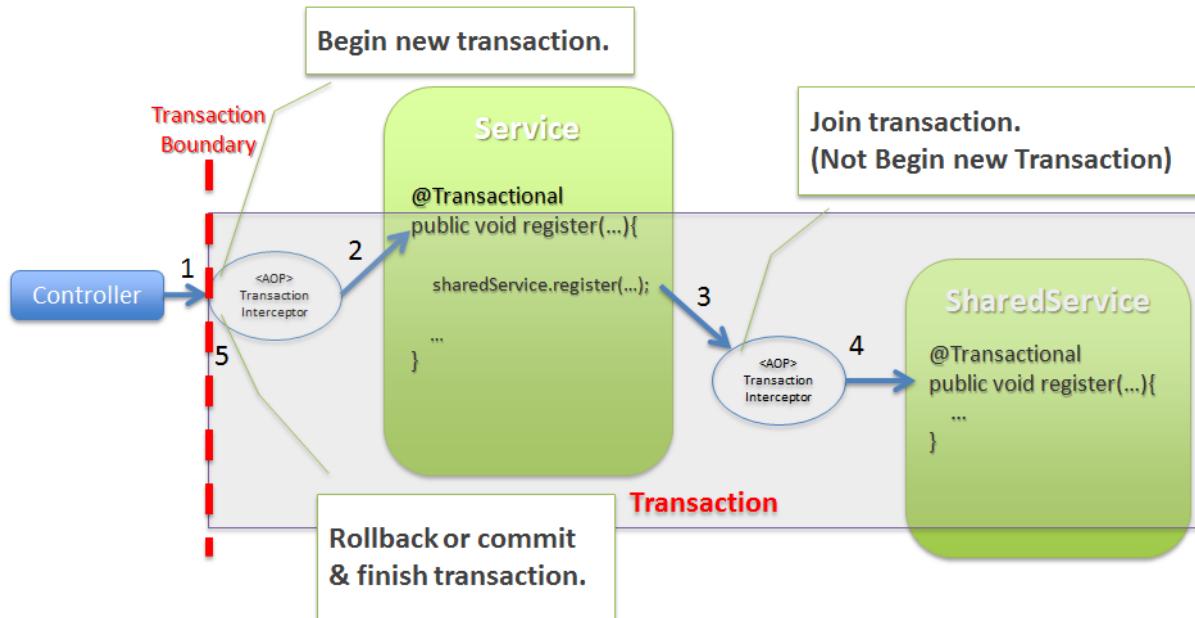
In most of the cases, Propagation method of transaction is “REQUIRED”.

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.

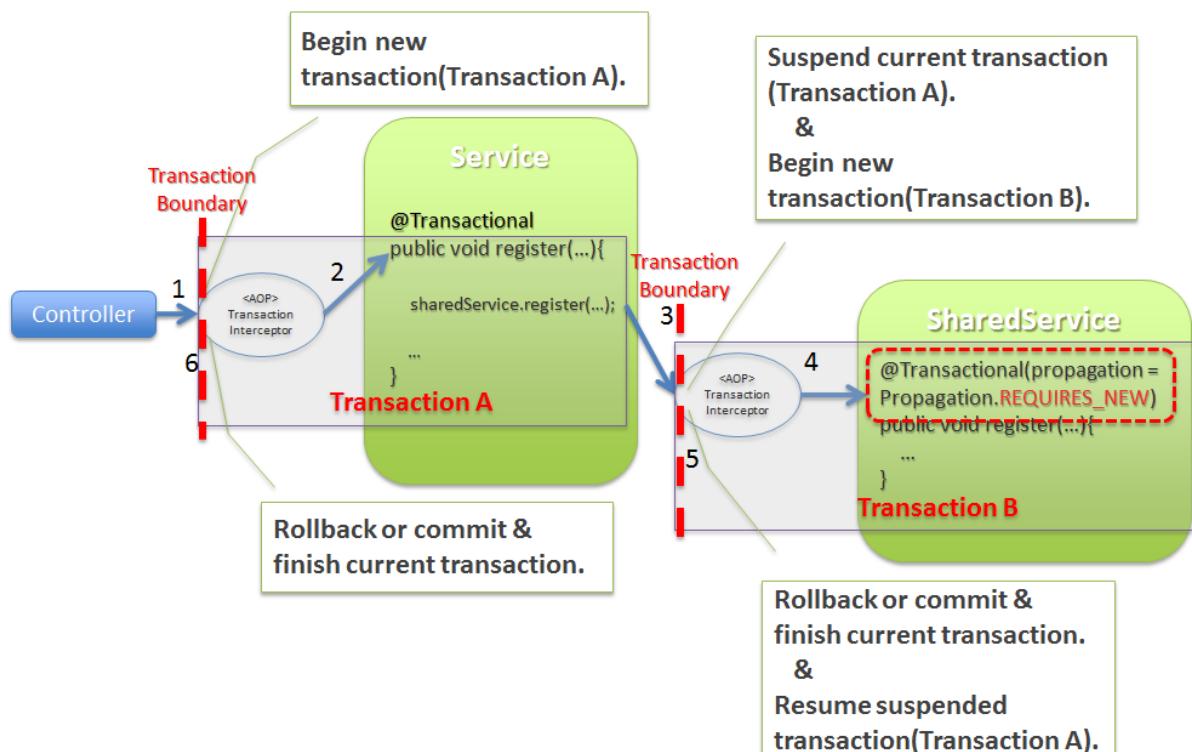
Note: Reason for occurrence of org.springframework.transaction.UnexpectedRollbackException

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.

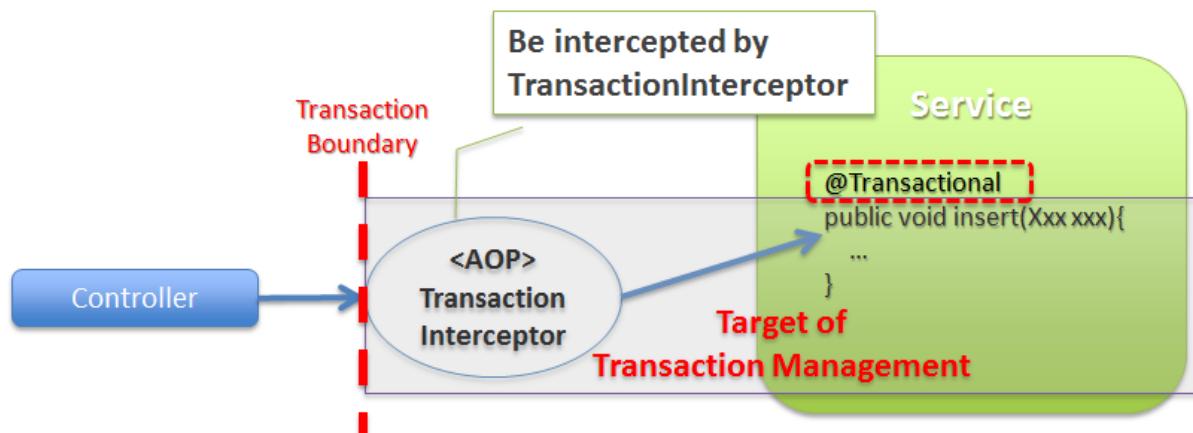
Way of calling the method which is under transaction control

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



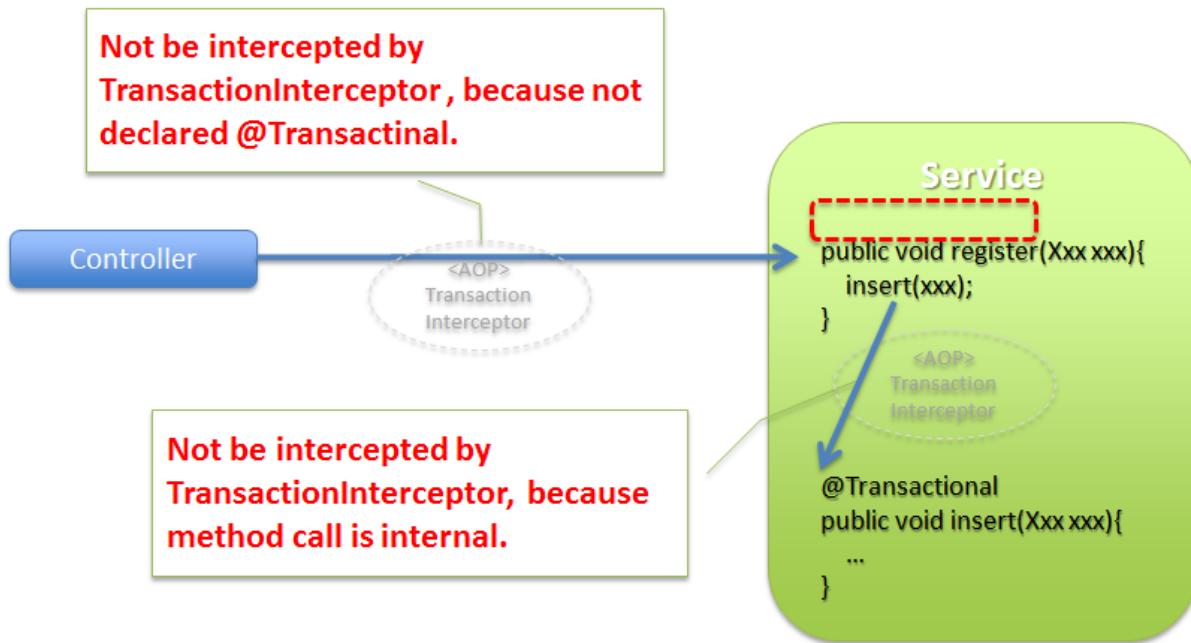
- Way of calling the method which is not under transaction control

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.	<code>org.springframework.jdbc.datasource.DataSourceTransactionManager</code>	Implementation class for managing the transaction by calling API of JDBC(<code>java.sql.Connection</code>). Use this class when Mybatis or <code>JdbcTemplate</code> is to be used.
2.	<code>org.springframework.orm.jpa.JpaTransactionManager</code>	Implementation class for managing the transaction by calling API of JPA(<code>javax.persistence.EntityTransaction</code>). Use this class when JPA is to be used.
3.	<code>org.springframework.transaction.jta.JtaTransactionManager</code>	Implementation class for managing the transaction by calling API of JTA(<code>javax.transaction.UserTransaction</code>). 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". <ul style="list-style-type: none"> 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 DeveloperWorks](#)for details.

Programmatic transaction management

In this guideline, “Declarative transaction management” is recommended. However, programmatic transaction management is also possible. Refer to [Spring Reference Document](#)for 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)	<p>Interface to limit signature of implementation method of business logic.</p> <p>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.</p> <p>In this guideline, the above interface is called BLogic interface.</p>

- 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 [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 [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 <code>JpaRepository</code> , basic CRUD operations for <code>Todo</code> 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 [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 unnecessary 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
    QueryDAO queryDAO; // (3)

    @Inject
    UpdateDAO updateDAO; // (4)

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

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

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

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

    @Override
```

```
@Transactional(readOnly = true) // (5)
public Iterable<Todo> findAll(Sort sort) { // (7)
    return queryDAO.executeForObjectList("todo.findAllSort", sort);
}

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

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

@Override
public <S extends Todo> S save(S todo) { // (6)
    if(exists(todo.getId())){
        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.class);
    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.

- [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 and correlation check by Spring Validator or Bean Validation.

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.

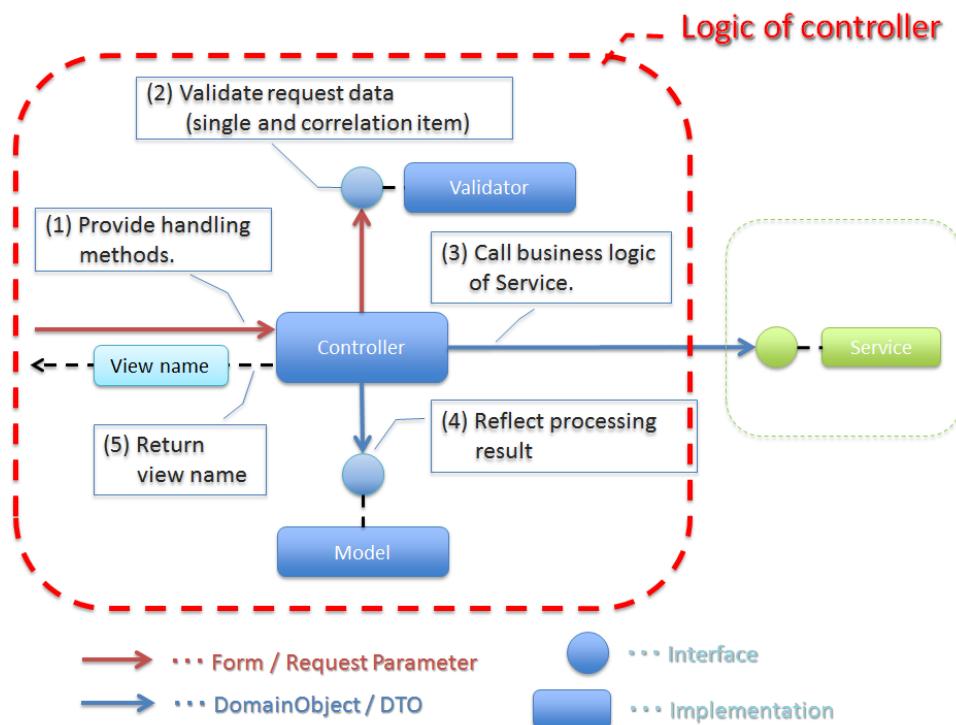


Figure.4.1 Picture - Logic of controller

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 (<code>RequestMethodType</code>) 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 documentation 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 paramName exists.
2.	!paramName	Mapping is performed when request parameter of the specified paramName does not exist.
3.	paramName=paramValue	Mapping is performed when value of the specified paramName is paramValue.
4.	paramName!=paramValue	Mapping is performed when value of the specified paramName 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.

- *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.

- *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.

- *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.

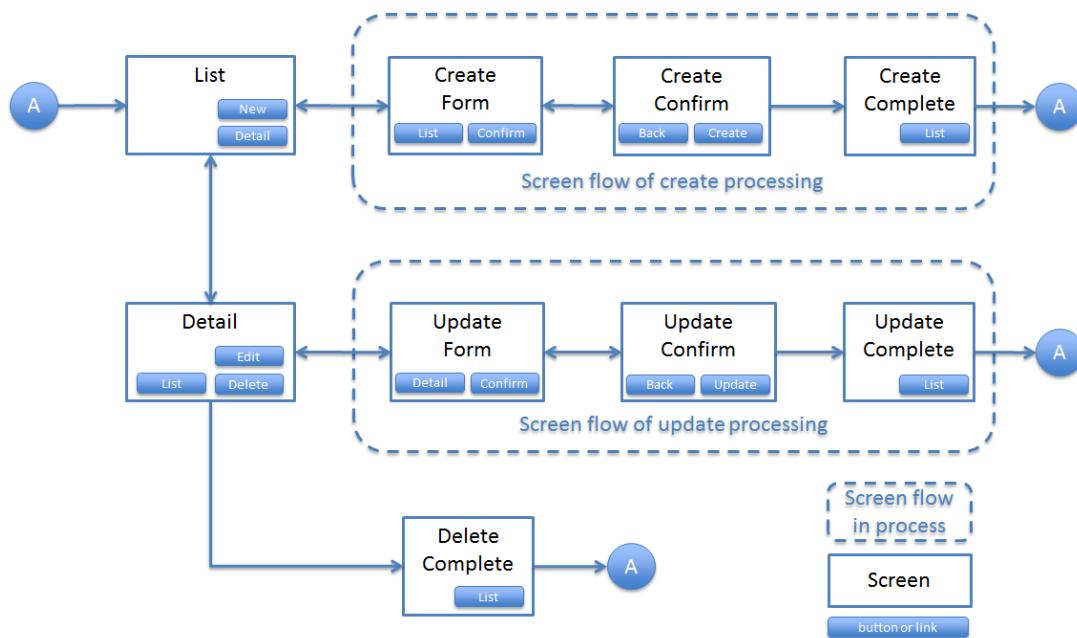


Figure.4.2 Picture - Screen flow of entity management function

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 Pattern** and 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:

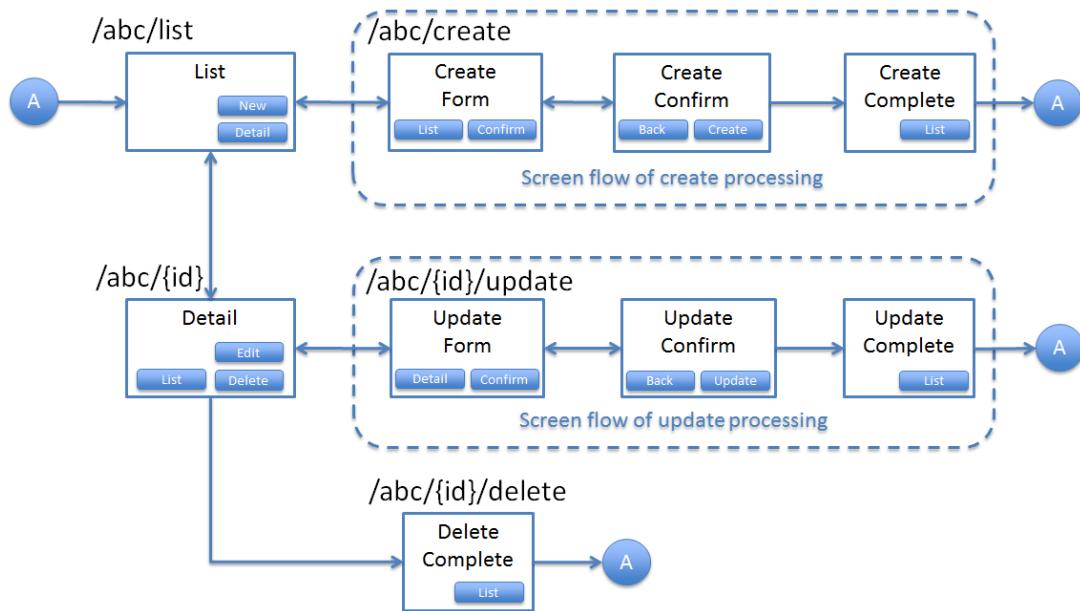


Figure.4.3 Picture - Screen flow of entity management function and corresponding assigned URL

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	createComplete
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	updateConfirm
10.			Form re-display	POST	redo	updateRedo
11.			Update	POST	-	update
12.			Displaying completion of update process	GET	complete	updateComplete
13.	Entity delete	/abc/{id}/delete	Delete	POST	-	delete
14.			Displaying completion of delete process	GET	complete	deleteComplete

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.

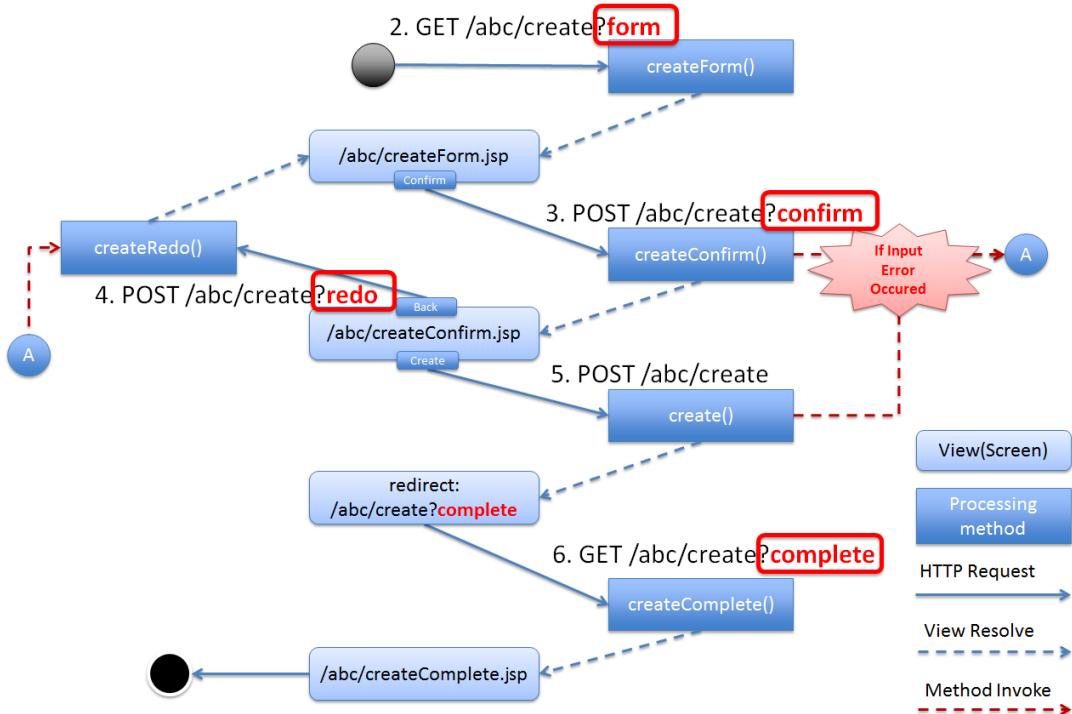


Figure.4.4 Picture - Request flow of entity create processing

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 <code>params</code> 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>
```

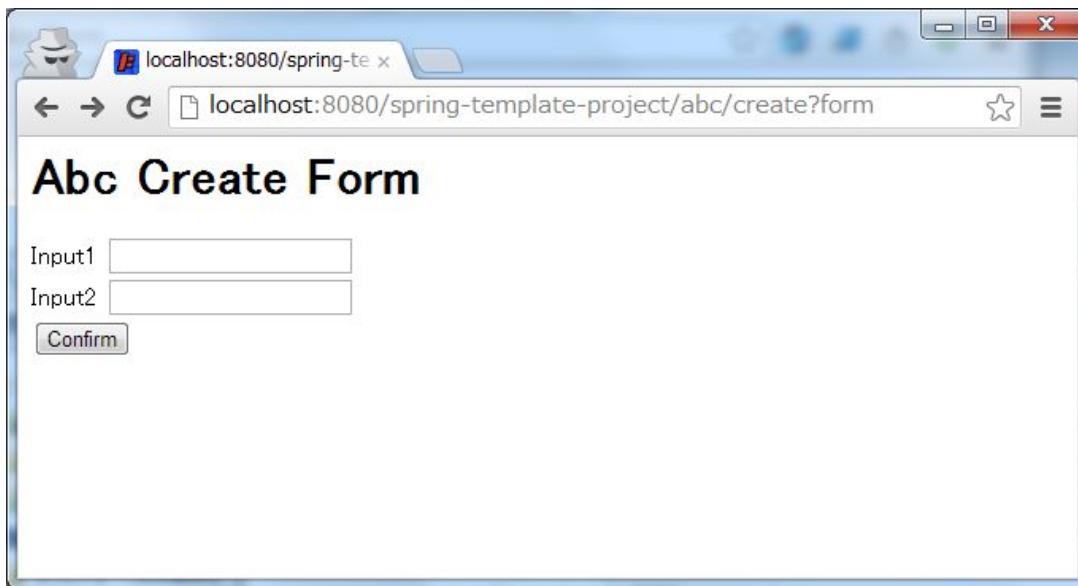
Sr.No.	Description
(1)	Specify name="confirm" parameter for submit button to transit to confirmation screen.

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)
    }
    // omitted
    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. Specify “name=”redo”“parameter for submit button to return to form screen.
(2)	
(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.

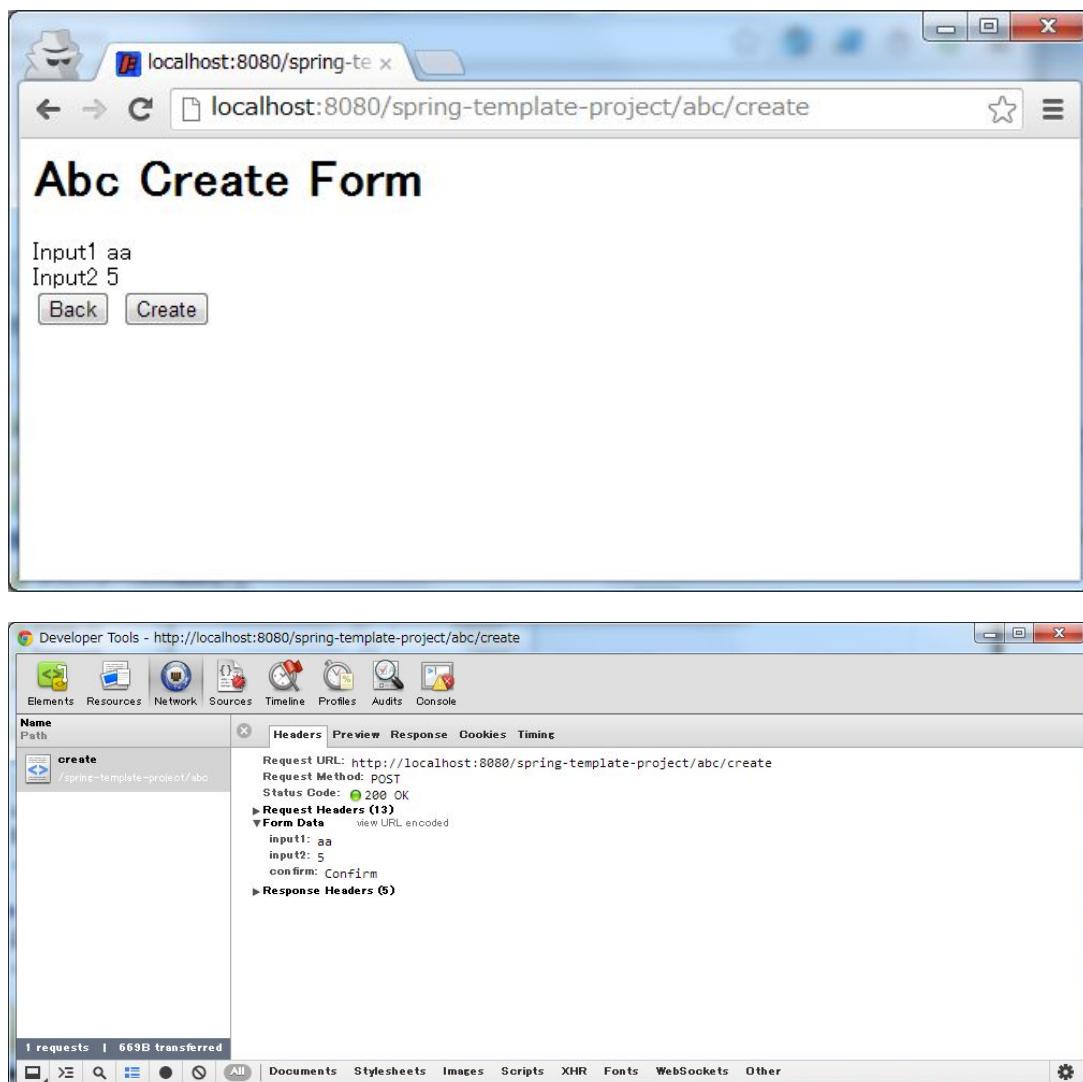
Send the request for displaying user input confirmation.

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. Return view name of JSP to render the form screen.
(2)	

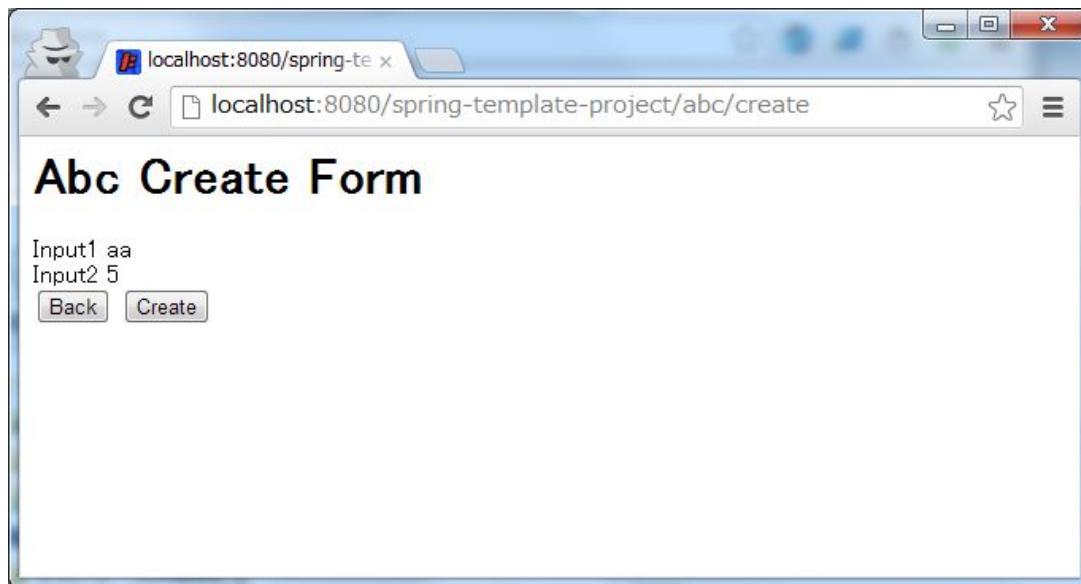
Operation is described below.

Send the request to redisplay the form screen.

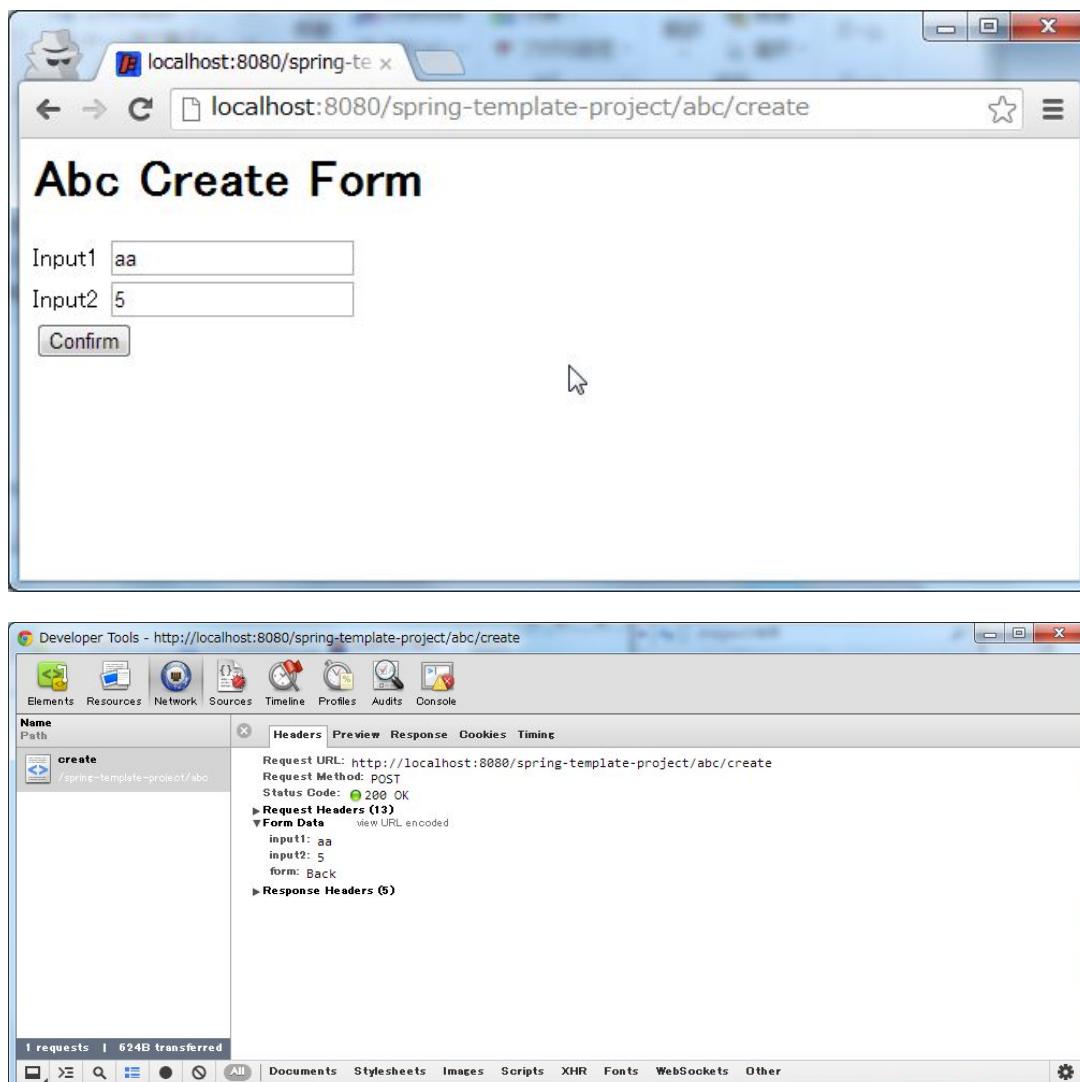
Click Back button on user input confirmation screen.

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 *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 *Double Submit Protection* .

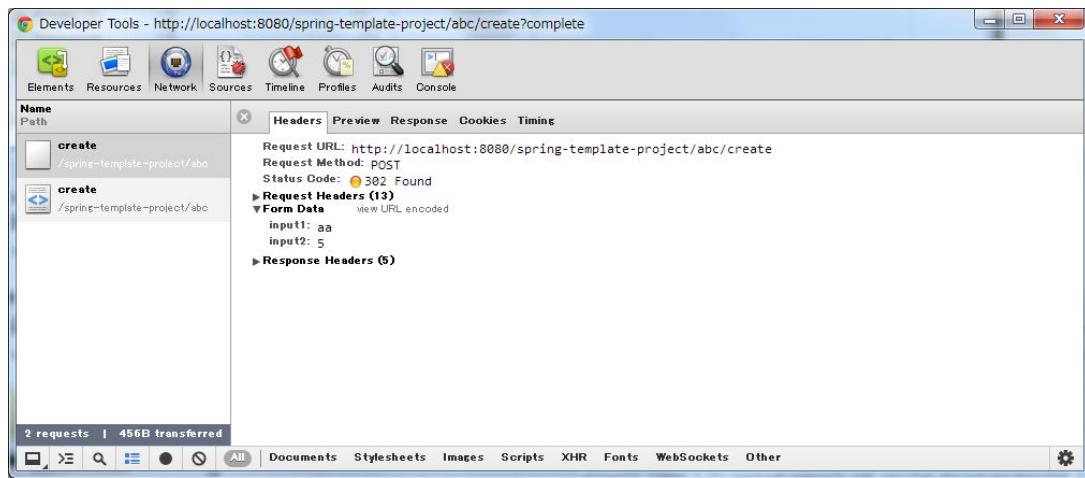
Operation is described below.

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)
}
```

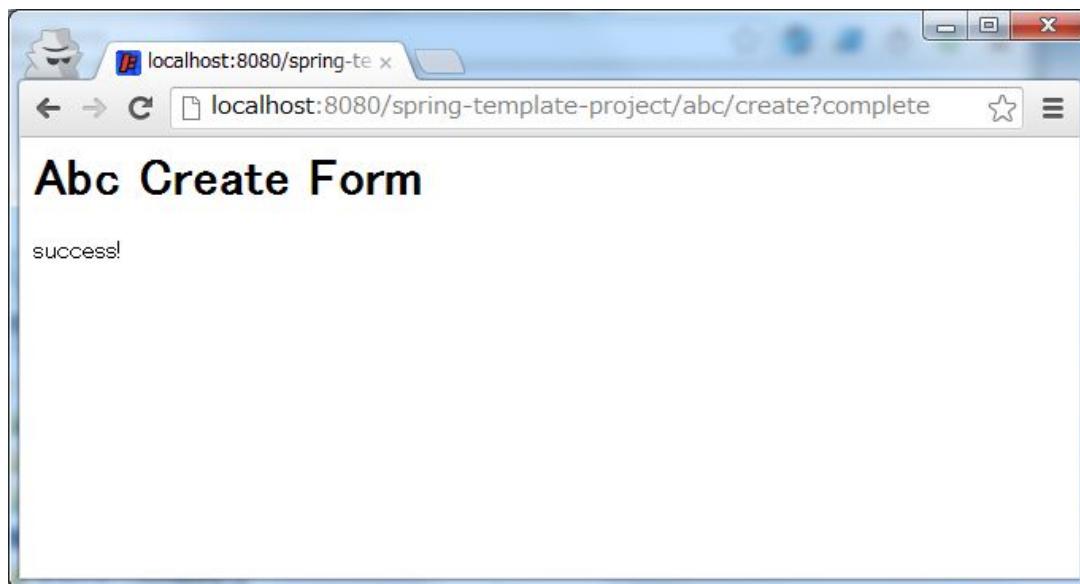
Sr.No.	Description
(1)	Specify "complete" in params attribute. Return View name of JSP to render the create completion screen.
(2)	

Note: In this processing method, `method` attribute is not specified since it is not required for HTTP GET method.

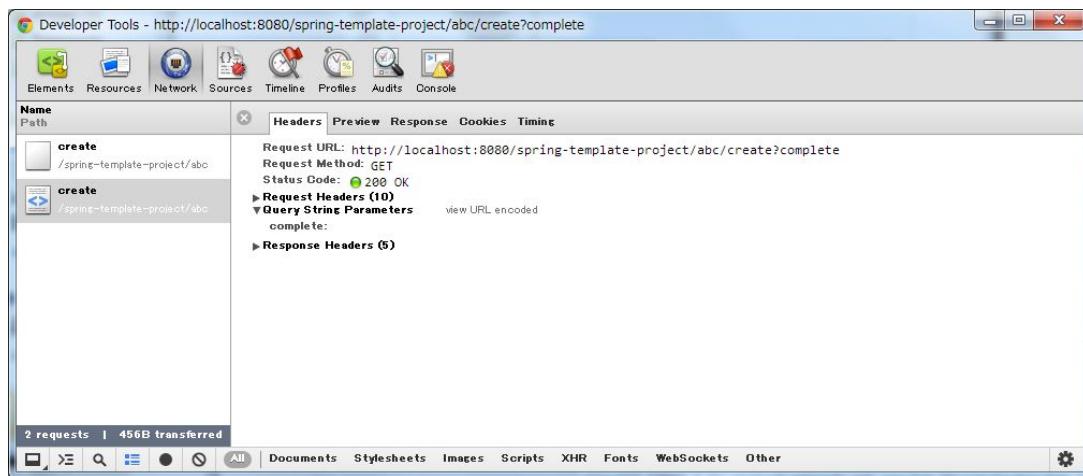
Operation is described below.

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 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'*.

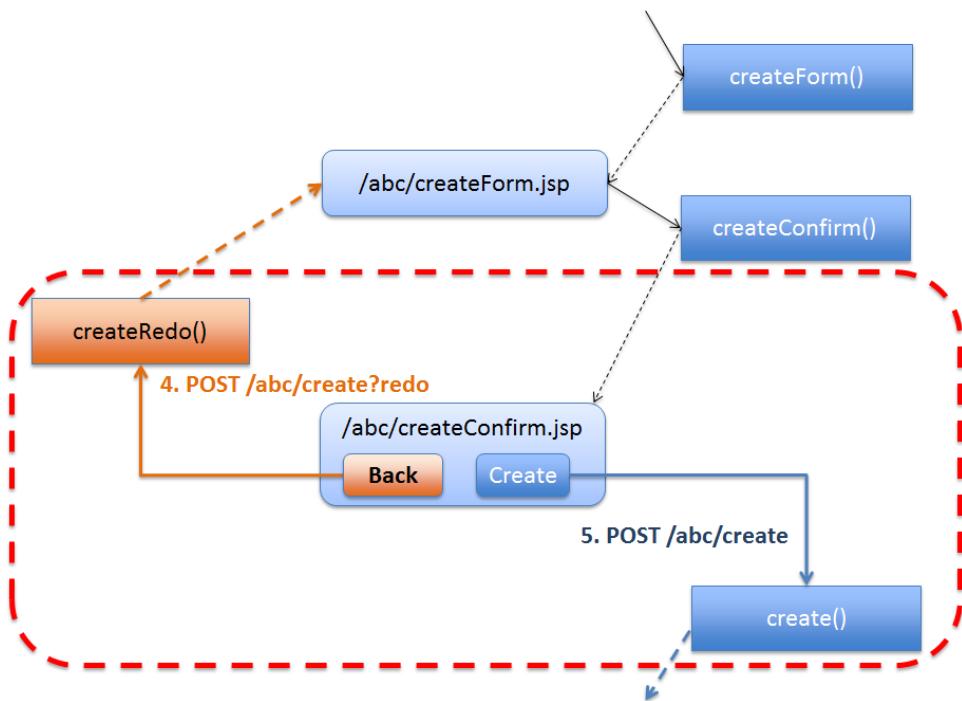


Figure.4.5 Picture - Multiple button in the HTML form

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 `Modelobject`.

- 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 Modelobject as argument.
(2)	Call addAttributemethod of Modelobject received as argument, and add the data to Modelobject. For example, "HelloWorld!" string is added to the attribute name "hello".
(3)	If first argument of addAttributemethod 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 Modelobject 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 <i>Cross Site Scripting</i> .
(5)	The values of JavaBean stored in Modelcan be acquired by specifying "\${Attribute name.property name}".
(6)	JSP is output in HTML format.

Note: Even though the Modelis 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 <code>@RequestMapping</code> annotation. Specify path variable in “{variable name}” format. For example, 2 path variables such as “ <code>id</code> ” and “ <code>version</code> ” are specified.
(2)	Specify variable name of path variable in <code>@PathVariable</code> annotation. For example, when the URL “ <code>sample/hello/aaaa/1</code> ” is accessed, the string “ <code>aaaa</code> ” is passed to argument “ <code>id</code> ”.
(3)	Value attribute of <code>@PathVariable</code> annotation can be omitted. When it is omitted, the argument name is considered as the request parameter name. For example, when the URL “ <code>sample/hello/aaaa/1</code> ” is accessed, value “ <code>1</code> ” is passed to argument “ <code>version</code> ”. 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") String
    Model model) {
    // do something
    return "sample/hello"; // returns view name
}
```

Sr.No.	Description
(1)	<p>Specify request parameter name in the value attribute of @RequestParamannotation.</p> <p>For example, when the URL "sample/hello?id=aaaa" is accessed, the string "aaaa" is passed to argument "id".</p>
(2)	<p>value attribute of @RequestParamannotation can be omitted. When it is omitted, the argument name becomes the request parameter name.</p> <p>For example, when the URL "sample/hello?name=bbbb&...." is accessed, string "bbbb" is passed to argument "name".</p> <p>However, if value attribute is omitted, the compilation should be done in debug mode.</p>
(3)	<p>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.</p> <p>For example, when it is accessed where request parameter age does not exist, null is passed to argument "age".</p>
(4)	<p>When default value is to be used if the specified request parameter does not exist, specify the default value in defaultValue attribute.</p> <p>For example, when it is accessed where request parameter genderCode does not exist, "unknown" is passed to argument "genderCode".</p>

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") String 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 form object, 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 @Validatedannotation to SampleFormargument, 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 <code>RedirectAttributes</code> object as argument of the processing method of controller.
(2)	Call <code>addFlashAttribute</code> method of <code>RedirectAttributes</code> and add the data to <code>RedirectAttributes</code> object. For example, the string data "HelloWorld!" is added to attribute name "hello".
(3)	If first argument of <code>addFlashAttribute</code> method is omitted, the class name beginning with lower case letter becomes the attribute name. For example, the result of <code>model.addFlashAttribute("helloBean", new HelloBean())</code> ; is same as <code>model.addFlashAttribute(new HelloBean())</code> .
(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 <code>RedirectAttributes</code> 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 <i>Cross Site Scripting</i> .
(7)	The value stored in <code>RedirectAttributes</code> 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 `addAttributemethod` of `Model`. However survival time of data differs. In `addFlashAttributeof 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.

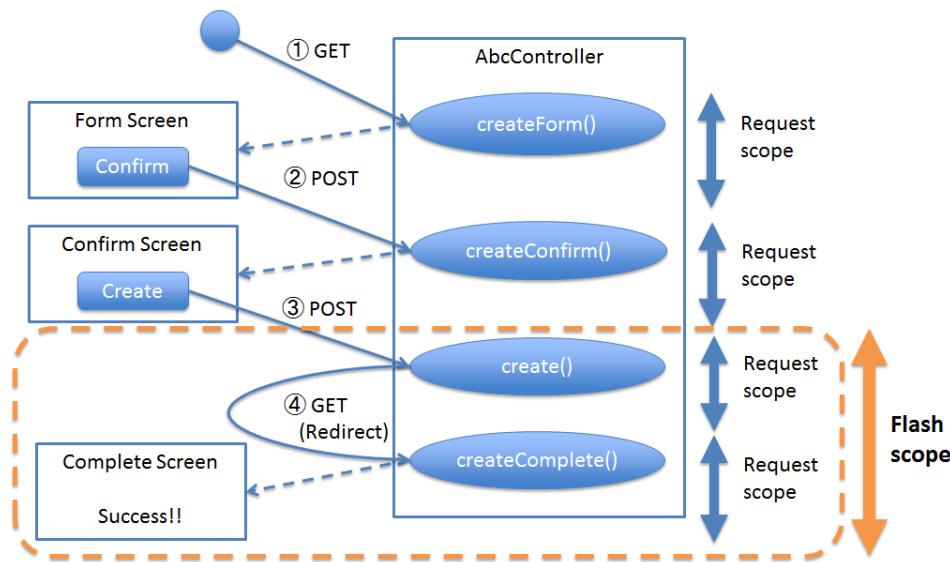


Figure.4.6 Picture - Survival time of flush scope

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)	<p>Specify request parameter name in argument name and request parameter value in argument ``value and call addAttributemethod of RedirectAttributesobject.</p> <p>In the above example, it is redirected to "/sample/hello?complete&id=aaaa".</p>

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 RedirectAttributesobject 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 RedirectAttributesobject 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)	<p>Specify attribute name and the value using addAttributemethod of RedirectAttributesobject.</p>
(2)	<p>Specify the path of the variable "{Attribute name}" to be inserted in the redirect URL.</p> <p>In the above example, it is redirected to "/sample/hello/aaaa?complete".</p>

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 @CookieValueannotation 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 @CookieValueannotation. 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 attribute is String . 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 <code>HttpServletResponse</code> object as argument to write to cookie.
(2)	Generate <code>Cookie</code> object and add to <code>HttpServletResponse</code> 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 [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 [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>](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 [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 [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 method, 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.Validatorinterface.

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
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
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 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. Example of delegating the process to Helper class is shown below:

- SampleController.java

```
@Inject
SampleHelper sampleHelper; // (1)

@RequestMapping("hello")
public void hello(@Validated SampleForm form, BindingResult result) {
    // ommited
    String message = sampleHelper.hello(form); // (2)
    model.addAttribute("message", message);
    return "sample/hello";
}
```

- SampleHelper.java

```
public class SampleHelper {

    @Inject
    SampleService sampleService;

    public void hello(SampleForm form) { // (3)
        Sample sample = new Sample();
        sample.setField1(form.getField1());
        sample.setField2(form.getField2());
        sample.setField3(form.getField3());
        // ...
        // and more ...
        // ...
        String message = sampleService.hello(sample);
        return message;
    }
}
```

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 [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 ...
    ...
}

```

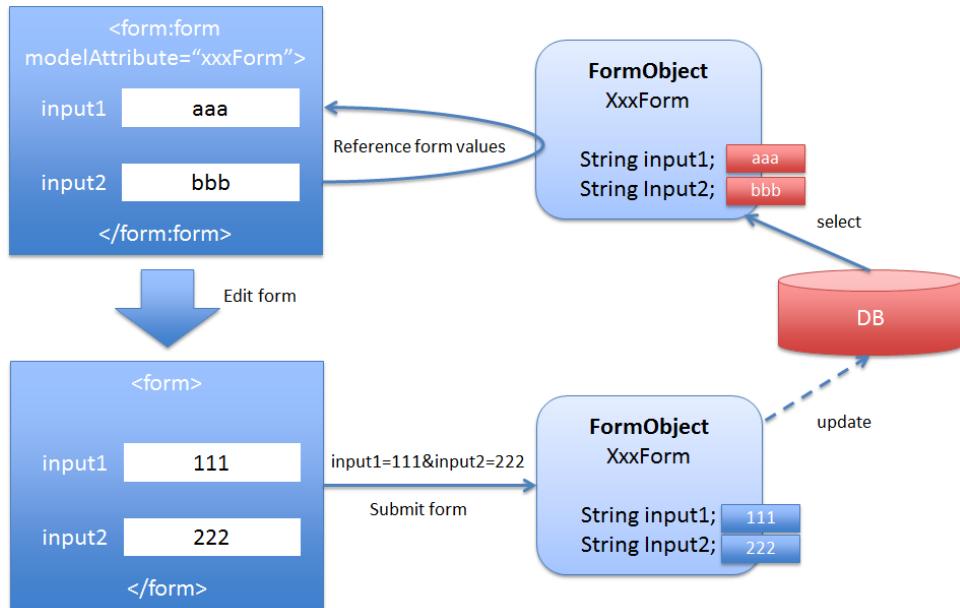
Sr.No.	Description
(1)	Call the method to reflect the values of domain object to form object.
(2)	Reflect the values of domain object to form object.

Note: Bean conversion functionality can be used as an alternative method to delegate the process to Helper class. Refer to [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`.

```
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 object in 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.
	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;  
    // omitted 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 <code>birthDate</code> 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	<p>日付と時刻のスタイルを 2 桁の文字列として指定する。</p> <p>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。</p> <p>スタイルとして指定できる値は以下の値となる。</p> <p>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。</p> <p>(指定例及び変換例)</p> <p>MM : Dec 9, 2013 3:37:47 AM M- : Dec 9, 2013 -M : 3:41:45 AM</p>

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 @InitBinderannotation 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 @InitBinderannotation. 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, 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 @ModelAttributeannotation. Initialize form-backing bean by the method having

@ModelAttributeannotation. 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)	<p>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.</p> <p>The returned object is added to Model and an appropriate process <code>model.addAttribute(form)</code> is executed.</p>
(2)	<p>When attribute name is to be specified to add to Model, specify it in the value attribute of <code>@ModelAttribute</code> annotation. In the above example, "xxx" is the attribute name.</p> <p>For returned object, appropriate process "model.addAttribute("xxx", form)" is executed and it is returned to Model.</p> <p>When attribute name other than default value is specified, it is necessary to specify <code>@ModelAttribute ("xxx")</code> at the time of specifying form object as an argument of processing method.</p>
(3)	<p><code>ModelAttribute</code> 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 <code>@CookieValue</code> annotation.</p>

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 <code>@ModelAttribute ("xxx")</code> .

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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/spring.tld.html
3.	JSTL	<ul style="list-style-type: none"> • 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 the 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 jpg) 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
I(1)	Add HelloBeanobject to Modelobject.
(2)	In View(JSP), data added to the Modelobject 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 <i>value</i> attribute of <i><c:out></i> 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)	<p>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 .</p> <p>When the value acquired in \${helloBean.numberItem} is “1.2” temporarily, “1.20” is displayed on the screen.</p>

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)	<p>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.</p> <p>When value received for \${helloBean.dateItem} is 2013-3-2, “2013-03-02” is displayed on the screen.</p>

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 attribute of <form:form> tag.

Sr.No.	Description
(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 library](#)for 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 [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 (<code>java.util.Map</code> interface) is stored with name ("CL_ORDERSTATUS") as attribute name. Therefore, in JSP, codelist (<code>java.util.Map</code> interface) can be accessed using EL expression. Codelist can be displayed in select box by passing the object of Map interface to <code>items</code> attribute of <code><form:options></code> .

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 (<code>java.util.Map</code> 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 [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>code</code> attribute of <code><spring:message></code> , 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 <code>test</code> attribute of <code><c:if></code> . In this example, when order status is not ' <code>complete</code> ', 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 <code><c:when></code> . 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 <code><c:when></code> tag is processed.
(2)	When result of test attribute of all <code><c:when></code> tags is false, <code><c:otherwise></code> 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 <code>items</code> attribute of <code><c:forEach></code> tag, <code><c:forEach></code> tag is repeatedly executed to iterate over the collection. While iterating over the collection, if the current element is to be referred inside <code><c:forEach></code> tag, it can be done by specifying a variable name for the current element in <code>var</code> attribute.
(2)	By specifying a variable name in <code>varStatus</code> attribute, current position (count) of iteration in <code><c:forEach></code> tag can be fetched. Refer to JavaDoc of 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 <code>var</code> attribute of <code><c:forEach></code> 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 [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> <!-- (1) -->
</filter>
<filter-mapping>
  <filter-name>clientInfoPutFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- applicationContext.xml

```
<bean id="clientInfoPutFilter" class="x.y.z.ClientInfoPutFilter" /> <!-- (2) -->
```

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. Specify the pattern of path, where the created HandlerInterceptor is to be applied.
(2)	Specify the pattern of path, where the created HandlerInterceptor need not be applied.
(3)	Add the created HandlerInterceptor to <mvc:interceptors> tag of spring-mvc.xml.
(4)	

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 <code>org.springframework.web.method.support.HandlerMethodArgumentResolver</code> interface.
(3)	Determine parameter type. For example, when type of JavaBean that retains common parameters is specified as argument of controller, <code>resolveArgument</code> 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 <code>HandlerMethodArgumentResolver</code> to <code><mvc:argument-resolvers></code> tag of <code>spring-mvc.xml</code> .

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 `@InitBindermethod` 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 <code>@ControllerAdvice</code> annotation.
(2)	It should be component-scan target by assigning the <code>@Component</code> annotation.
(3)	Specify priority for common processes by assigning the <code>@Order</code> annotation. It should be specified when multiple ControllerAdvice are to be created.
(4)	Implement <code>@InitBindermethod</code> . <code>@InitBindermethod</code> 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 @ExceptionHandlermethod. @ExceptionHandlermethod 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 <code>@ModelAttribute</code> method. <code>@ModelAttribute</code> method is applied to all Controllers. Object created by <code>@ModelAttribute</code> method is passed.
(2)	

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 [*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 [*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 Database Access (Common)

Todo

TBD

The following topics in this chapter are currently under study.

- About multiple datasources

For details, refer to [Overview - About multiple datasources](#) and [How to extends - About multiple datasources](#).

- About handling of unique constraint errors and pessimistic exclusion errors when using JPA

For details, refer to [Overview - About exception handling](#).

5.1.1 Overview

This chapter explains the method of accessing the data stored in RDBMS.

For O/R Mapper dependent part, refer to [Database Access \(JPA\)](#) and [Database access \(Mybatis2\)](#).

About JDBC DataSource

RDBMS can be accessed from the application by referring to JDBC datasource.

JDBC datasource can be used to exclude settings such as JDBC driver load, connection information (URL, user, password etc.) from application.

Therefore, RDBMS to be used and environment in which the application is to be deployed need not be taken into account.

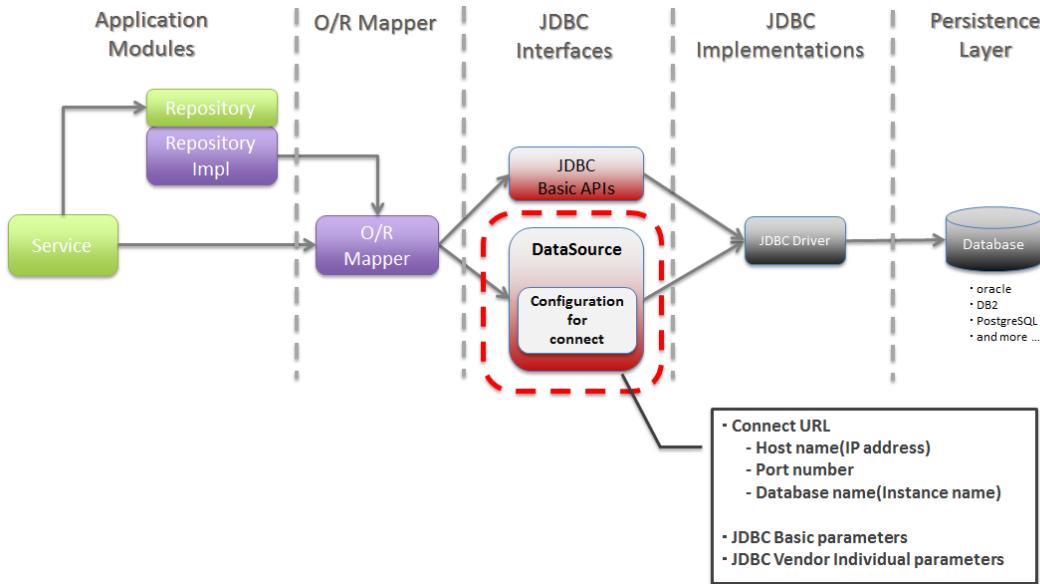


Figure.5.1 Picture - About JDBC DataSource

JDBC datasource is implemented from Application Server, OSS library, Third-Party library, Spring Framework etc.; hence it is necessary to select the datasource based on project requirements and deployment environment. The typical datasources are introduced below.

- *JDBC datasource provided by Application Server*
- *JDBC datasource provided by OSS/Third-Party library*
- *JDBC datasource provided by Spring Framework*

JDBC datasource provided by Application Server

When datasource is to be used in Web application, normally JDBC datasource provided by Application Server is used.

JDBC datasource of Application Server provides functionalities required in web application such as Connection Pooling as standard functionalities.

Table.5.1 Datasources provided by Application Server

Sr. No.	Application Server	Reference page
1.	Apache Tomcat 7	Refer to Apache Tomcat 7 User Guide (The Tomcat JDBC Connection Pool) . Refer to Apache Tomcat 7 User Guide (JNDI Datasource HOW-TO) (Apache Commons DBCP) .
2.	Oracle WebLogic Server 12c	Refer to Oracle WebLogic Server Product Documentation .
3.	IBM WebSphere Application Server Version 8.5	Refer to WebSphere Application Server Online information center .
4.	Resin 4.0	Refer to Resin Documentation .

JDBC datasource provided by OSS/Third-Party library

When JDBC datasource of Application Server is not used, JDBC datasource of OSS/Third-Party library should be used.

This guideline introduces only Apache Commons DBCP; however other libraries can also be used.

Table.5.2 JDBC datasource provided by OSS/Third-Party library

Sr. No.	Library name	Description
1.	Apache Commons DBCP	Refer to Apache Commons DBCP .

JDBC datasource provided by Spring Framework

Implementation class of JDBC datasource of Spring Framework cannot be used as datasource of Web application since it does not provide connection pooling.

In Spring Framework, implementation class and adapter class of JDBC datasource are provided; however they are introduced as *JDBC datasource classes provided by Spring Framework* of Appendix, since usage is restricted.

About transaction management

When transactions are to be stored using Spring Framework functionality, PlatformTransactionManager needs to be selected based on project requirements and deployment environment.

For details, refer to [*Settings for using transaction management*](#) of [*Domain Layer Implementation*](#).

About declaration of transaction boundary/attribute

Transaction boundary and transaction attributes should be declared by specifying @Transactional annotation in Service.

For details, refer to [*Regarding transaction management*](#) of [*Domain Layer Implementation*](#).

About exclusion control of data

When updating data, it is necessary to execute exclusion control to ensure data consistency and integrity.

For details on exclusion control of data, refer to [*Exclusive Control*](#).

About exception handling

In Spring Framework, a function is provided to convert JDBC exception (`java.sql.SQLException`) and O/R Mapper specific exception to data access exception (subclass of `(org.springframework.dao.DataAccessException)` provided by Spring Framework).

For the class which is converted to data access exception of Spring Framework, refer to [*Classes provided by Spring Framework for converting to data access exception*](#) of Appendix.

The converted data access exception need not be handled in application code; however, some errors (such as unique constraint violation, exclusion error etc.) need to be handled as per the requirements.

When handling data access exception, exception of subclass notifying error details should be caught instead of `DataAccessException`.

Typical subclasses which are likely to be handled in application code are as follows:

Table.5.3 Subclasses of DB access exception, which are likely to be handled

Sr. No.	Class name	Description
1.	org.springframework.dao. DuplicateKeyException	Exception that occurs in case of unique constraint violation.
2.	org.springframework.dao. OptimisticLockingFailureException	Exception that occurs in case of optimistic locking failure. It occurs when same data is updated with different logic. This exception occurs when JPA is used as O/R Mapper. Mybatis does not have optimistic locking function; hence this exception does not occur from O/R Mapper.
3.	org.springframework.dao. PessimisticLockingFailureException	Exception that occurs in case of pessimistic locking failure. It occurs when same data is locked with different logic and the lock is not released even after “waiting for unlocking” timeout period has elapsed.

Note: When optimistic locking is to be implemented using Mybatis in O/R Mapper, it should be implemented as Service or Repository process.

As a method of notifying the optimistic locking failure to Controller, this guideline recommends generation of OptimisticLockingFailureException and exception of its child class.

This is to make implementation of application layer (implementation of Controller) independent of O/R Mapper to be used.

Todo

It has been recently found that using JPA (Hibernate) results in occurrence of unexpected errors.

- 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.UncategorizedDataAccessException` occurs and not `PessimisticLockingFailureException`.

`UncategorizedDataAccessException` that occurs in case 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

See the example below for handling unique constraint violation as business exception.

```
try {
    accountRepository.saveAndFlash(account);
} catch(DuplicateKeyException e) { // (1)
    throw new BusinessException(ResultMessages.error().add("e.xx.xx.0002"), e); // (2)
}
```

Sr. No.	Description
(1)	Exception (DuplicateKeyException) that occurs in case of unique constraint violation is caught.
(2)	Business exception indicating that there is duplicate data is thrown. When exception is caught, make sure to specify the cause of exception (e) in business exception.

About multiple datasources

Multiple datasources may be required depending on the application.

Typical cases wherein multiple datasources are required, are shown below.

Table.5.4 Typical case where multiple datasources are required

Sr. No.	Case	Example	Feature
1.	When database and schema are divided according to data (tables).	When group of tables maintaining customer information and group of tables maintaining invoice information are stored in separate database and schema.	The data to be handled in the process is fixed; hence the datasource to be used can be defined statically.
2.	When database and schema to be used are divided according to users (login users).	When database and schema are divided according to users (Multitenant etc.).	The datasource to be used differs depending on users; hence the datasource to be used dynamically can be defined.

Todo**TBD**

The following details will be added in future.

- Conceptual diagram
- Details of above two cases Especially, in case (1), transaction management method is likely to change depending on processing pattern (such as DB update using multiple datasources; DB update using only a single datasource, DB read only, no concurrent access etc.); hence plan to break down considering all these aspects.

About common library classes

Common library provides classes that carry out following processes.

For more details about common library, refer to links given below.

- *Escaping during LIKE search*
- *About Sequencer*

5.1.2 How to use

Datasource settings

Settings when using DataSource defined in Application Server

When using datasource defined in Application Server, it is necessary to perform settings in Bean definition file to register the object fetched through JNDI as a bean.

Settings when PostgreSQL is used as database and Tomcat7 is used as Application Server are given below.

- xxx-context.xml (Tomcat config file)

```
<!-- (1) -->
<Resource
    type="javax.sql.DataSource"
    name="jdbc/SampleDataSource"
    driverClassName="org.postgresql.Driver"
    url="jdbc:postgresql://localhost:5432/terasoluna"
    username="postgres"
    password="postgres"
    defaultAutoCommit="false"
/> <!-- (2) -->
```

- xxx-env.xml

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/SampleDataSource" /> <!-- (3) -->
```

Sr. No.	Attribute name	Description
(1)	-	Define datasource.
	type	Specify resource type. Specify javax.sql.DataSource.
	name	Specify resource name. The name specified here is JNDI name.
	driverClassName	Specify JDBC driver class. In the example, JDBC driver class provided by PostgreSQL is specified.
	url	Specify URL. [Needs to be changed as per environment]
	username	Specify user name. [Needs to be changed as per environment]
	password	Specify password of user. [Needs to be changed as per environment]
	defaultAutoCommit	Specify default value of auto commit flag. Specify ‘false’. It is forcibly set to ‘false’ when it is under Transaction Management.
(2)	-	In case of Tomcat7, tomcat-jdbc-pool is used if factory attribute is omitted. For more details about settings, refer to Attributes of The Tomcat JDBC Connection Pool .
(3)	-	Specify JNDI name of datasource. In case of Tomcat, specify the value specified in resource name “(1)-name” at the time of defining datasource.

Settings when using DataSource for which Bean is defined

When using datasource of OSS/Third-Party library or JDBC datasource of Spring Framework without using the datasource provided by Application Server,

bean for DataSource class needs to be defined in Bean definition file.

Settings when PostgreSQL is used as database and Apache Commons DBCP is used as datasource are given below.

- `xxx-env.xml`

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <!-- (1) (8) -->
    <property name="driverClassName" value="org.postgresql.Driver" /> <!-- (2) -->
    <property name="url" value="jdbc:postgresql://localhost:5432/terasoluna" /> <!-- (3) -->
    <property name="username" value="postgres" /> <!-- (4) -->
    <property name="password" value="postgres" /> <!-- (5) -->
    <property name="defaultAutoCommit" value="false"/> <!-- (6) -->
<!-- (7) -->
</bean>
```

Sr. No.	Description
(1)	Specify implementation class of datasource. In the example, datasource class (<code>org.apache.commons.dbcp.BasicDataSource</code>) provided by Apache Commons DBCP is specified.
(2)	Specify JDBC driver class. In the example, JDBC driver class provided by PostgreSQL is specified.
(3)	Specify URL. [Needs to be changed as per environment]
(4)	Specify user name. [Needs to be changed as per environment]
(5)	Specify password of user. [Needs to be changed as per environment]
(6)	Specify default value of auto commit flag. Specify ‘false’. It is forcibly set to ‘false’ when it is under Transaction Management.
(7)	In BasicDataSource, configuration values common in JDBC, JDBC driver specific properties values, connection pooling configuration values can be specified other than the values mentioned above. For more details about settings, refer to DBCP Configuration .
(8)	In the example, values are specified directly; however, for fields where configuration values change with the environment, actual configuration values should be specified in properties file using Placeholder(\${...}). For Placeholder, refer to PropertyPlaceholderConfigurer of Spring Reference Document .

Settings to enable transaction management

For basic settings to enable transaction management, refer to [Settings for using transaction management of Domain Layer Implementation](#).

For PlatformTransactionManager, the class to be used changes depending on the O/R Mapper used; hence for

detailed settings, refer to *Database Access (JPA)*, *Database access (Mybatis2)*.

JDBC debug log settings

When more detailed information than the log output using O/R Mapper(Hibernate, MyBatis) is required, the information output using log4jdbc(log4jdbc-remix) can be used.

For details on log4jdbc, refer to [log4jdbc project page](#).

For details on log4jdbc-remix, refer to [log4jdbc-remix project page](#).

Warning: This is a debug setting; hence it should not be performed in case of applications that are to be released in performance test environment or production environment.

Settings related to datasource provided by log4jdbc

- `xxx-env.xml`

```
<jee:jndi-lookup id="dataSourceSpied" jndi-name="jdbc/SampleDataSource" /> <!-- (1) -->
<bean id="dataSource" class="net.sf.log4jdbc.Log4jdbcProxyDataSource"> <!-- (2) -->
  <constructor-arg ref="dataSourceSpied" /> <!-- (3) -->
</bean>
```

Sr. No.	Description
(1)	Define actual datasource. In the example, the datasource fetched through JNDI from Application Server is being used.
(2)	Specify <code>net.sf.log4jdbc.Log4jdbcProxyDataSource</code> provided by log4jdbc.
(3)	In constructor, specify bean which is an actual datasource.

Warning: When the application is to be released in performance test environment or production environment, Log4jdbcProxyDataSource should not be used as datasource.

Specifically, exclude settings of (2) and (3) and change bean name of "dataSourceSpied" to "dataSource".

log4jdbc logger settings

- `logback.xml`

```
<!-- (1) -->
<logger name="jdbc.sqltiming">
  <level value="debug" />
</logger>

<!-- (2) -->
<logger name="jdbc.sqlonly">
  <level value="warn" />
</logger>

<!-- (3) -->
<logger name="jdbc.audit">
  <level value="warn" />
</logger>

<!-- (4) -->
<logger name="jdbc.connection">
  <level value="warn" />
</logger>

<!-- (5) -->
<logger name="jdbc.resultset">
  <level value="warn" />
</logger>

<!-- (6) -->
<logger name="jdbc.resultsettable">
  <level value="debug" />
</logger>
```

Sr. No.	Description
(1)	Logger to output SQL execution time and SQL statement wherein the value is set in bind variable. Since this SQL contains values for bind variables, it can be executed using DB access tool.
(2)	Logger to output SQL statement wherein the value is set in bind variable. The difference with (1) is that SQL execution time is not output.
(3)	Logger to exclude ResultSet interface, call methods of JDBC interface and to output arguments and return values. This log is useful for analyzing the JDBC related issues; however volume of the output log is large.
(4)	Logger to output connected/disconnected events and number of connections in use. This log is useful for analyzing connection leak, but it need not be output unless there is connection leak issue.
(5)	Logger to call methods of ResultSet interface and output arguments and return values. This log is useful during analysis when actual result differs from expected result; however volume of the output log is large.
(6)	Logger to output the contents of ResultSet by converting them into a format so that they can be easily verified. This log is useful during analysis when actual result differs from expected result; however volume of the output log is large.

Warning: Large amount of log is output depending on the type of logger; hence only the required logger should be defined or output.

In the above sample, log level for loggers which output very useful logs during development, is set to "debug". As for other loggers, the log level needs to be set to "debug" whenever required.

When the application is to be released in performance test environment or production environment, log using log4jdbc logger should not be output at the time of normal end of process.

Typically log level should be set to "warn".

Settings of log4jdbc option

Default operations of log4jdbc can be customized by placing properties file `log4jdbc.properties` under class path.

- `log4jdbc.properties`

```
# (1)
log4jdbc.dump.sql.maxlinelength=0
# (2)
```

Sr. No.	Description
(1)	Specify word-wrap setting for SQL statement. If '0' is specified, SQL statement is not wrapped. For option details, refer to log4jdbc project page .
(2)	

5.1.3 How to extend

Settings for using multiple datasources

Todo

TBD

Following details will be added in future.

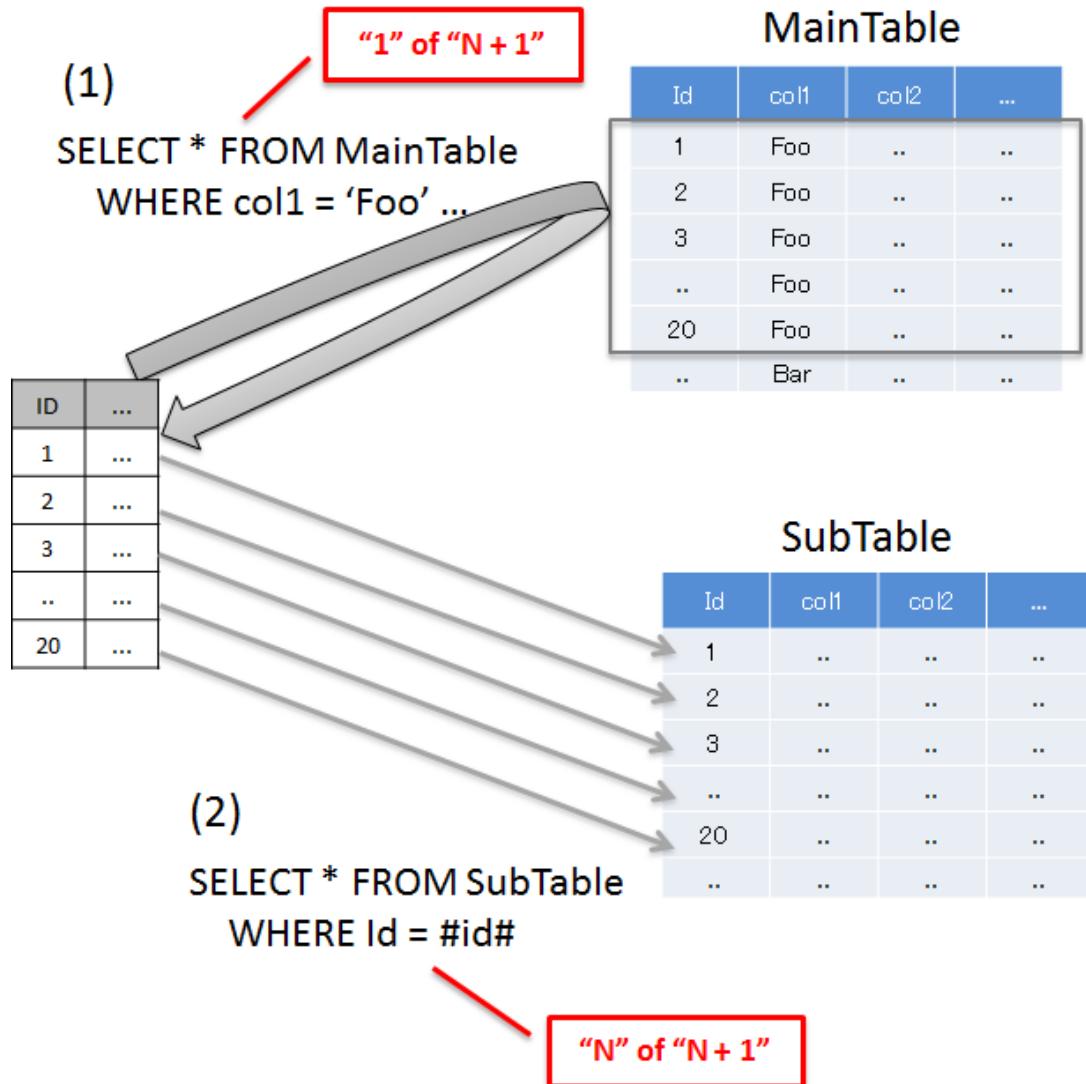
- Example of settings based on notes to be considered while using multiple datasources
 - Example of implementation if it is getting impacted
-

5.1.4 How to resolve the problem

How to resolve N+1

N+1 occurs when more number of SQL statements need to be executed in accordance with the number of records to be fetched from the database. This problem causes high load on the database and deteriorates response time.

Details are given below.



Sr. No.	Description
(1)	<p>Search the records matching the search conditions from MainTable.</p> <p>In the above example, col1 of MainTable fetches 'FOO' records and the total records fetched are 20.</p>
(2)	<p>For each record searched in (1), related records are fetched from SubTable.</p> <p>In the above example, the id column of SubTable fetches the same records as the id column of records fetched in (1).</p> <p>This SQL is executed for number of records fetched in (1).</p>

In the above example, **SQL is executed totally 21 times**.

Supposing there are 3 SubTables, **SQL is executed totally 61 times; hence countermeasures are required.**

Typical example to resolve N+1 is given below.

Resolving N+1 using JOINs (Join Fetch)

By performing JOIN on SubTable and MainTable, records of MainTable and SubTable are fetched by executing SQL once.

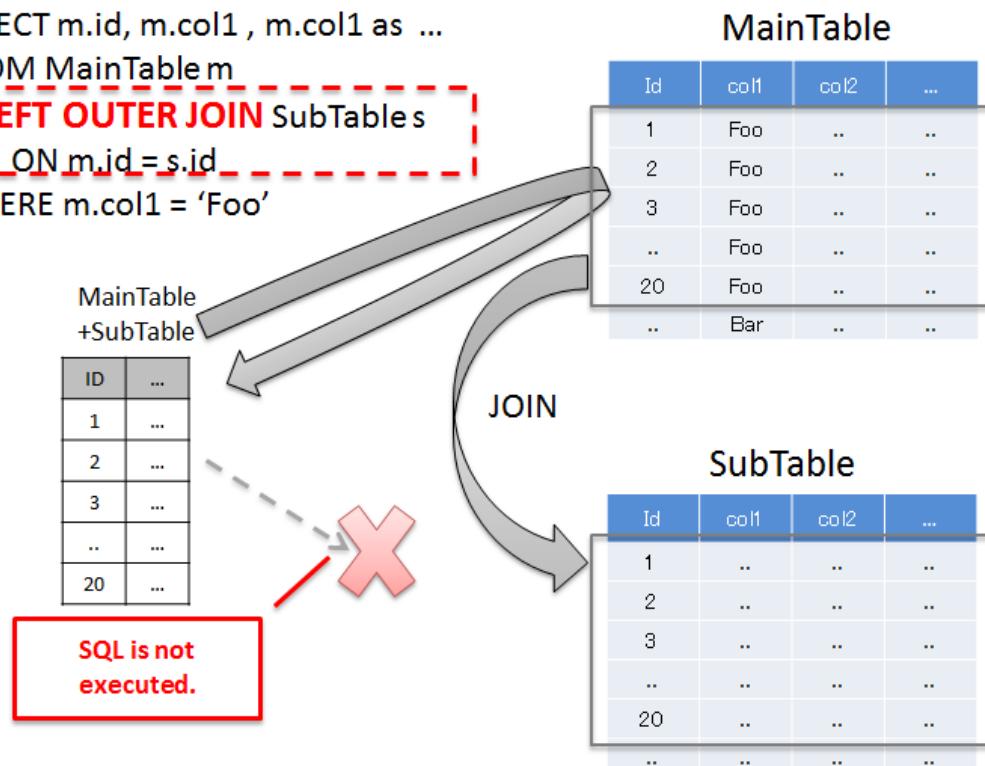
When relation of MainTable and SubTable is 1:1, check whether N+1 can be resolved using this method.

Sr. No.	Description
(1)	<p>When searching records matching the search conditions, the records are fetched in batch from MainTable and SubTable, by performing JOIN on SubTable.</p> <p>In the above example, col1 of MainTable collectively fetches 'FOO' records and records of SubTable that match the id of the records matching with search conditions.</p> <p>When there are duplicate column names, it is necessary to assign alias name in order to identify the table to which that column belongs.</p>

If JOIN (Join Fetch) is used, **all the required data can be fetched by executing SQL once.**

(1)

```
SELECT m.id, m.col1 , m.col1 as ...
FROM MainTable m
LEFT OUTER JOIN SubTables
ON m.id=s.id
WHERE m.col1 = 'Foo'
```



Note: When performing JOIN by JPQL

For example of performing JOIN using JPQL, refer to [JOIN FETCH](#).

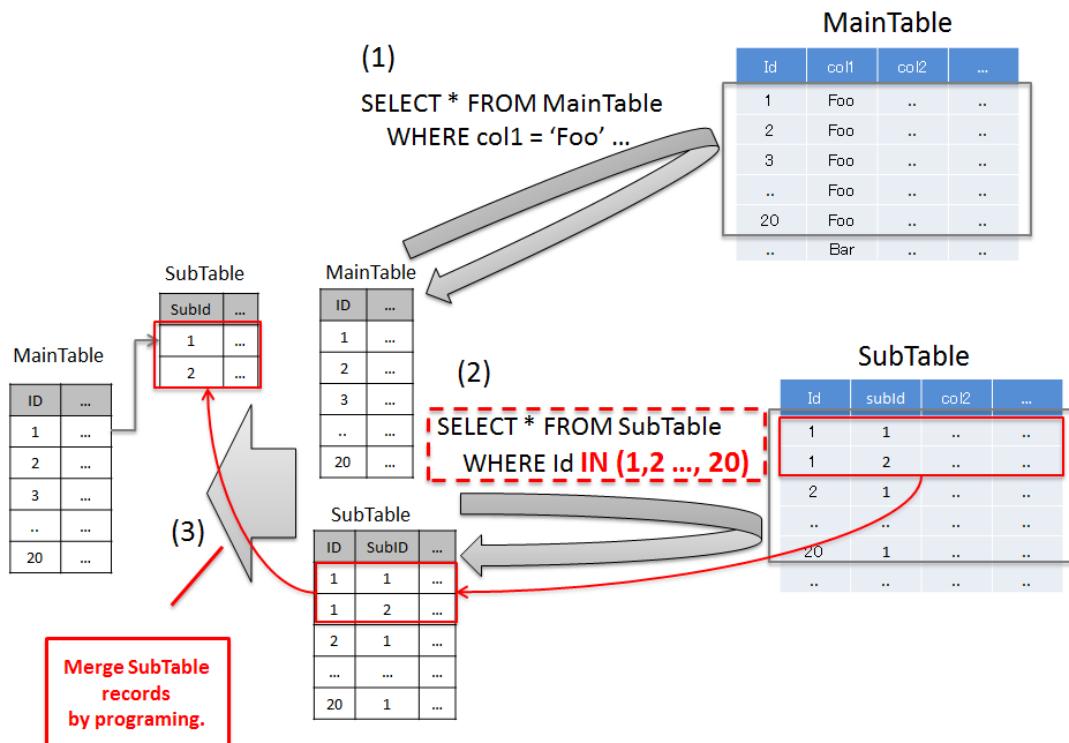
Warning: When relation with SubTable is 1:N, the problem can be resolved using JOIN (Join Fetch); however the following points should be noted.

- When JOIN is performed on records having 1:N relation, unnecessary data is fetched depending on the number of records in SubTable. For details, refer to [Notes during collective fetch](#).
- When using JPA (Hibernate), if N portions in 1:N are multiple, then it is necessary to use `java.util.Set` instead of `java.util.List` as a collection type storage N portion.

Resolving N+1 by fetching related records in batch

There are cases where, it has proved better when the related records are fetched in batch for patterns with multiple 1:N relations etc.; and then sorted by programming.

When relation with SubTable is 1:N, analyze whether the problem can be resolved using this method.



Sr. No.	Description
(1)	<p>Search the records matching the search conditions from MainTable.</p> <p>In the above example, <code>col1</code> of MainTable fetches '<code>Foo</code>' records and the total records fetched are 20.</p>
(2)	<p>For each record searched in (1), related records are fetched from SubTable.</p> <p>Related records are not fetched one by one; but the records matching the foreign key of each record fetched in (1), are fetched in batch.</p> <p>In the above example, <code>id</code> column of SubTable collectively fetches same records as <code>id</code> column of records fetched in (1) using <code>IN</code> clause.</p>
(3)	SubTable records fetched in (2) sorted and merged with records fetched in (1).

In the above example, **all the required data can be fetched by executing SQL twice**.

Even supposing there are 3 SubTables, **SQL needs to be executed totally 4 times**.

Note: This method has a special feature. It can fetch only the required data by optimizing SQL execution. It is necessary to sort SubTable records by programming; however when there are many SubTables or when number of N records in 1:N is more, there are cases wherein it is better to resolve the problem using this method.

5.1.5 Appendix

Escaping during LIKE search

While performing LIKE search, the values to be used as search conditions need to be escaped.

Escaping for LIKE search can be done using methods of

`org.terasoluna.gfw.common.query.QueryEscapeUtils` class provided by common library.

Specifications of escaping of common library

Specifications of escaping provided by common library are as follows:

- Escape character is "`~`" .
- Characters to be escaped are 4, namely "`%`" , "`_`" , "`%`" , "`__`" .

See the example of escaping below.

Sr. No.	Target string	String after escaping	Escape flag	Explanation
1.	"a"	"a"	OFF	Escaping not done as the string does not contain character to be escaped.
2.	"a~"	"a~~"	ON	Escaping done as the string contains escape character.
3.	"a%"	"a~%"	ON	Escaping done as the string contains character to be escaped.
4.	"a_"	"a~_"	ON	Similar to No.3.
5.	"_a%"	"~_a~%"	ON	Escaping done as the string contains characters to be escaped. When there are multiple characters to be escaped, escaping is done for all characters.
6.	"a %"	"a~%"	ON	Similar to No.3.
7.	"a __"	"a~__"	ON	Similar to No.3.
8.	" "	" "	OFF	Similar to No.1.
9.	""	""	OFF	Similar to No.1.
10.	null	null	OFF	Similar to No.1.

About escaping methods provided by common library

List of escaping methods of `QueryEscapeUtils` class provided by common library is given below.

Sr. No.	Method name	Description
1.	toLikeCondition(String)	String passed as an argument is escaped for LIKE search. When specifying type of matching (Forward match, Backward match and Partial match) at SQL or JPQL side, perform only escaping using this method.
2.	toStartingWithCondition(String)	After escaping a string passed as an argument for LIKE search, assign "% " at the end of the string after escaping. This method is used in order to convert into a value for Forward match search.
3.	toEndingWithCondition(String)	After escaping a string passed as an argument for LIKE search, assign "% " at the beginning of the string after escaping. This method is used in order to convert into a value for Backward match search.
4.	toContainingCondition(String)	After escaping a string passed as an argument for LIKE search, assign "% " at the beginning and end of the string after escaping. This method is used in order to convert into a value for Partial match search.

Note: Methods of No.2, 3, 4 are used when specifying the type of matching (Forward match, Backward match and Partial match) at program side and not at SQL or JPQL side.

How to use common library

For example of escaping at the time of LIKE search, refer to Document for O/R Mapper to be used.

- When using JPA (Spring Data JPA), refer to *Escaping at the time of LIKE search of Database Access (JPA)*.
- When using Mybatis2 (TERASOLUNA DAO), refer to *Escaping during LIKE search of Database access (Mybatis2)*.

About Sequencer

Sequencer is a common library for fetching sequence value.

Use the sequence value fetched from Sequencer as a configuration value of primary key column of the database.

Note: Reason for creating Sequencer as a common library

The reason for creating Sequencer is that there is no mechanism to format the sequence value as string in ID generator functionality of JPA. In actual application development, sometimes the formatted string is also set as primary key; hence Sequencer is provided as common library.

When value set as primary key is number, it is recommended to use ID generator functionality of JPA. For ID generator functionality of JPA, refer to [How to add entities of Database Access \(JPA\)](#).

The primary objective of creating Sequencer is to supplement functions which are not supported by JPA; but it can also be used when sequence value is required in the processes not relating to JPA.

About classes provided by common library

List of classes of Sequencer functionality of common library is as follows:

For usage example, refer to [How to use common library](#) of How to use.

Sr. No.	Class name	Description
1.	org.terasoluna.gfw.common.sequencer.Sequencer	Interface that defines the method to fetch subsequent sequence value (getNext) and method to return current sequence value (getCurrent).
2.	org.terasoluna.gfw.common.sequencer.JdbcSequencer	Implementation class of Sequencer interface for JDBC. This class is used to fetch sequence value by executing SQL in the database. For this class, it is assumed that values are fetched from sequence object of the database; however it is also possible to fetch the values from other than sequence object by calling function stored in the database.

How to use common library

Define a bean for Sequencer.

- xxx-infra.xml

```
<!-- (1) -->
<bean id="articleIdSequencer" class="org.terasoluna.gfw.common.sequencer.JdbcSequencer">
    <!-- (2) -->
    <property name="dataSource" ref="dataSource" />
    <!-- (3) -->
    <property name="sequenceClass" value="java.lang.String" />
    <!-- (4) -->
    <property name="nextValueQuery"
        value="SELECT TO_CHAR(NEXTVAL('seq_article'), 'AFM0000000000') " />
    <!-- (5) -->
    <property name="currentValueQuery"
        value="SELECT TO_CHAR(CURRVAL('seq_article'), 'AFM0000000000') " />
</bean>
```

Sr. No.	Description
(1)	Define a bean for class that implements <code>org.terasoluna.gfw.common.sequencer.Sequencer</code> . In the above example, (<code>JdbcSequencer</code>) class for fetching sequence value by executing SQL is specified.
(2)	Specify the datasource for executing the SQL to fetch sequence value.
(3)	Specify the type of sequence value to be fetched. In the above example, since conversion to string is done using SQL; <code>java.lang.String</code> type is specified.
(4)	Specify SQL for fetching subsequent sequence value. In the above example, sequence value fetched from sequence object of (PostgreSQL) database is formatted as string. When sequence value fetched from the database is 1, "A0000000001" is returned as return value of <code>Sequencer#getNext()</code> method.
(5)	Specify SQL for fetching current sequence value. When sequence value fetched from the database is 2, "A0000000002" is returned as return value of <code>Sequencer#getCurrent()</code> method.

Fetch sequence value from Sequencer for which bean is defined.

- Service

```
// omitted

// (1)
@Inject
@Named("articleIdSequencer") // (2)
Sequencer<String> articleIdSequencer;

// omitted

@Transactional
```

```
public Article createArticle(Article inputtedArticle) {  
  
    String articleId = articleIdSequencer.getNext(); // (3)  
    inputtedArticle.setArticleId(articleId);  
  
    Article savedArticle = articleRepository.save(inputtedArticle);  
  
    return savedArticle;  
}
```

Sr. No.	Description
(1)	<p>Inject Sequencer object for which bean is defined.</p> <p>In the above example, since sequence value is fetched as formatted string, <code>java.lang.String</code> type is specified as generics type of Sequencer.</p>
(2)	<p>Specify bean name of the bean to be injected in value attribute of <code>@javax.inject.Named</code> annotation.</p> <p>In the above example, bean name ("articleIdSequencer") defined in <code>xxx-infra.xml</code> is specified.</p>
(3)	<p>Call Sequencer#getNext () method and fetch the subsequent sequence value.</p> <p>In the above example, fetched sequence value is used as Entity ID.</p> <p>When fetching current sequence value, call Sequencer#getCurrent () method.</p>

Tip: When Sequencer for which bean is defined is 1, `@Named` annotation can be omitted. When specifying multiple sequencers, bean name needs to be specified using `@Named` annotation.

Classes provided by Spring Framework for converting to data access exception

Classes of Spring Framework which play a role in converting an exception to data access exception, are as follows:

Table.5.5 Classes of Spring Framework for converting to data access exception

Sr. No.	Class name	Description
1.	org.springframework.orm.hibernate3.SessionFactoryUtils	When JPA (Hibernate implementation) is used, O/R Mapper exception is converted to data access exception of Spring Framework using this class.
2.	Sub classes of org.hibernate.dialect.Dialect	When JPA (Hibernate implementation) is used, exceptions are converted to JDBC exception and O/R Mapper exception using this class.
3.	org.springframework.jdbc.support. SQLExceptionSQLExceptionTranslator	When Mybatis or JdbcTemplate is used, JDBC exception is converted to data access exception of Spring Framework using this class. Conversion rules are mentioned in XML file. XML file used by default is org/springframework/jdbc/support/sql-error-codes.xml in spring-jdbc.jar. It is also possible to change the default behavior by placing XML file (sql-error-codes.xml) just below class path.

JDBC datasource classes provided by Spring Framework

Spring Framework provides implementation of JDBC datasource. However since they are very simple classes, they are rarely used in production environment.

These classes are mainly used during Unit Testing.

Table.5.6 JDBC datasource classes provided by Spring Framework

Sr. No.	Class name	Description
1.	org.springframework.jdbc.datasource.DriverManagerDataSource	Datasource class for creating new connection by calling <code>java.sql.DriverManager#getConnection</code> when connection fetch request is received from the application. When connection pooling is required, Application Server datasource or datasource of OSS/Third-Party library should be used.
2.	org.springframework.jdbc.datasource.SingleConnectionDataSource	Child class of <code>DriverManagerDataSource</code> . This class provides implementation of single shared connection. This is a datasource class for unit test which works with single thread. Even in case of Unit Testing, if this class is used when datasource is to be accessed with multithread, care needs to be taken as it may not show the expected behavior.
3.	org.springframework.jdbc.datasource.SimpleDriverDataSource	Datasource class for creating new connection by calling <code>java.sql.DriverManager#getConnection</code> when connection fetch request is received from the application. When connection pooling is required, Application Server datasource or datasource of OSS/Third-Party library should be used.

Spring Framework provides adapter classes with extended JDBC datasource operations.

Specific adapter classes are introduced below.

Table.5.7 JDBC datasource adapter classes provided by Spring Framework

Sr. No.	Class name	Description
1.	org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy	Adapter class for converting a datasource which does not store transactions, into a datasource storing Spring Framework transactions.
2.	org.springframework.jdbc.datasource.lookup.IsolationLevelDataSourceRoute	Adapter class for switching the datasource to be used based on independence level of an active transaction.

5.2 Database Access (JPA)

Todo

TBD

The following topics in this chapter are currently under study.

- persistence.xml settings
- Implementing dynamic query using QueryDSL
- Examples of cases wherein it is desirable to change the default setting values for specifying fetch method of related-entities.
- Using multiple PersistenceUnits
- Using Native query

5.2.1 Overview

This section explains the method to access the database using JPA.

In this guideline, it is assumed that Hibernate is used as JPA provider and Spring Data JPA as JPA wrapper.

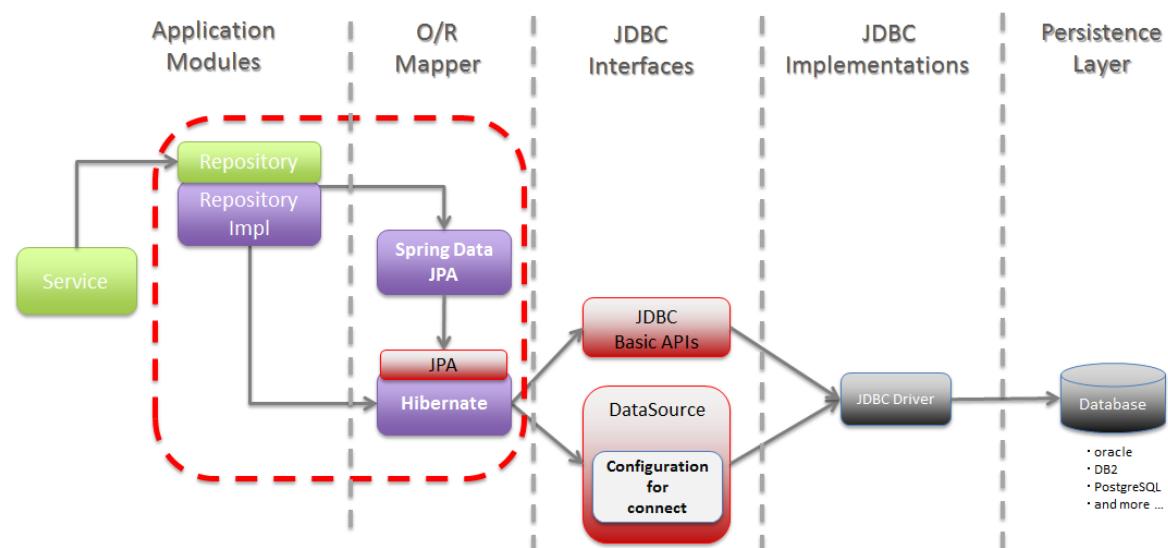


Figure.5.2 Picture - Target of description

Warning: The contents described in this chapter may not be applicable for JPA providers other than Hibernate such as EclipseLink etc.

About JPA

JPA (Java Persistence API) defines the following as an API of Java:

1. a way of mapping the records in a relational database, with the java objects
2. a mechanism for reflecting the operations done on the java object, to the records in a relational database.

JPA defines only specifications, it does not provide implementation.

JPA implementation is provided as a reference implementation by the vendors developing O/R Mapper such as Hibernate.

The reference implementation provided by the vendors developing O/R Mapper is called JPA Provider.

O/R Mapping of JPA

Mapping of Java objects to the relational database records at the time of using JPA is as follows:

In JPA, if the value stored in the “managed” entity is changed (by calling setter method), there is a mechanism to reflect the changes in the relational database.

This mechanism is quite similar to the client software such as Table viewer with Edit functionality.

In client software such as Table viewer, if the value of viewer is changed, it is reflected in the database. While in JPA, if the value of Java object (JavaBean) called “Entity” is changed, it is reflected in the database.

Basic JPA terminology

The basic terminology of JPA is described below.

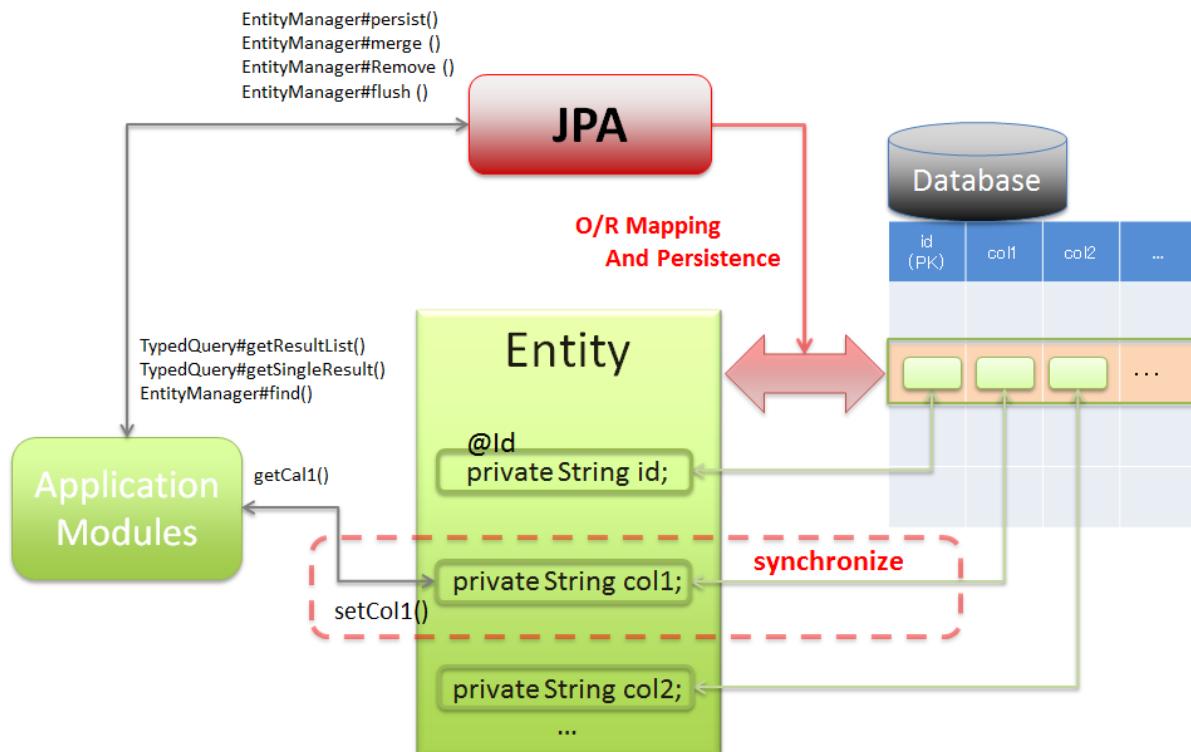


Figure.5.3 Picture - Image of O/R Mapping

Sr. No.	Term	Description
1.	Entity class	A Java class representing the records in the relational database The class with <code>@javax.persistence.Entity</code> annotation is an Entity class.
2.	EntityManager	An interface which provides API necessary for managing the life cycle of entity Using methods of <code>javax.persistence.EntityManager</code> , the application handles the relational database records as Java objects. When using Spring Data JPA, this interface is usually not used directly; however, if it is necessary to generate a query that cannot be expressed using the Spring Data JPA mechanism, then this interface can be used to fetch the entity.
3.	TypedQuery	An interface which provides API for searching an entity Using methods of <code>javax.persistence.TypedQuery</code> , the application searches for the entity matching the specified conditions other than ID.
5.2. Database Access (JPA)		327
When using Spring Data JPA, this interface is usually not used directly; however, if it is necessary to generate a query that cannot be expressed using Spring Data JPA mechanism, then this interface can be used to search the entity.		

Managing life cycle of entity

The life cycle of entity is managed as follows:

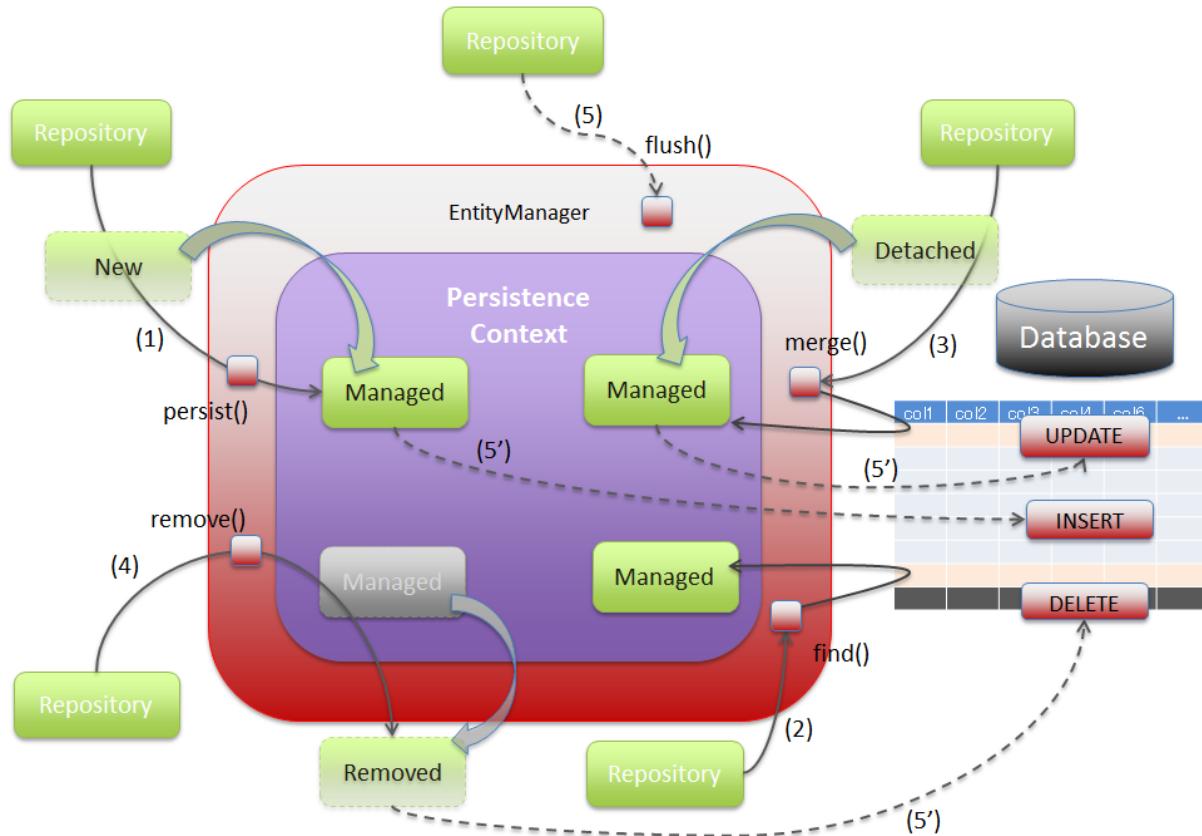


Figure.5.4 Picture - Life cycle of entity

Sr. No.	Description
(1)	When persist method of EntityManager is called, the entity (“New” entity) passed as an argument is stored in PersistenceContext as “managed” entity.
(2)	When find method of EntityManager is called, a “managed” entity with ID passed as an argument, is returned. If it does not exist in PersistenceContext, the records to be mapped are retrieved from the relational database by executing a query and stored as “managed” entity.
(3)	When merge method of EntityManager is called, the state of the entity (“detached”) passed as an argument, is merged with “managed” entity. If it does not exist in PersistenceContext, the records to be mapped are retrieved from the relational database by executing a query. The state of the entity passed as an argument is merged after the “managed” entity is stored. Note that when this method is called, the entity passed as an argument does not necessarily get stored as “managed” entity unlike persist method.
(4)	When remove method of EntityManager is called, the “managed” entity passed as an argument becomes “removed” entity. If this method is called, it is not possible to retrieve the “removed” entity.
(5)	When flush method of EntityManager is called, the operations of the entity accumulated using persist, merge and remove methods are reflected in the relational database. By calling this method, the changes done for an entity are synchronized with the records of relational database. However, the changes made only for the records of relational database are not synchronized with the entity. If the entity is searched by executing a query without using find method of EntityManager, then prior to the search process, a process similar to flush method is executed in the internal logic of EntityManager and the operations of the accumulated entity are reflected in the relational database. For timing to reflect the persistence operations at the time of using Spring Data JPA, refer to
5.2. Database Access (JPA)	<p><i>Reflection timing of persistence processing (1)</i></p> <p><i>Reflection timing of persistence processing (2)</i></p>
	329

Note: About other life cycle management methods

The detach method, refresh method and clear method are available in EntityManager to manage the entity life cycle. However, when using Spring Data JPA, there is no mechanism to call these methods using the default function, hence only their roles are described below.

- detach method is used to set a “managed” entity to “detached” entity.
- refresh method is used to update the “managed” entity as per the state of relational database.
- clear method is used to delete the entity managed in PersistenceContext and the accumulated operations from the memory.

clear method can be called by setting the clearAutomatically attribute of @Modifying annotation of Spring Data JPA to true. For details, refer to *Operating the entities of Persistence Layer directly*.

Note: About operations of “new” and “detached” entities

The operations performed on “new” and “detached” entities are not reflected in the relational database unless persist method or merge method is called.

About Spring Data JPA

Spring Data JPA provides the library to create Repository using JPA.

If Spring Data JPA is used, it is possible to retrieve an entity that matches the specified conditions only by defining the

query method in the Repository interface; hence the amount of implementation for performing the entity operations can be reduced.

However, only static query which can be expressed using annotation, can be defined in query method; hence it is necessary to implement custom Repository class for the query such as dynamic query which cannot be expressed using annotation.

The basic flow at the time of accessing the database using Spring Data JPA is shown below.

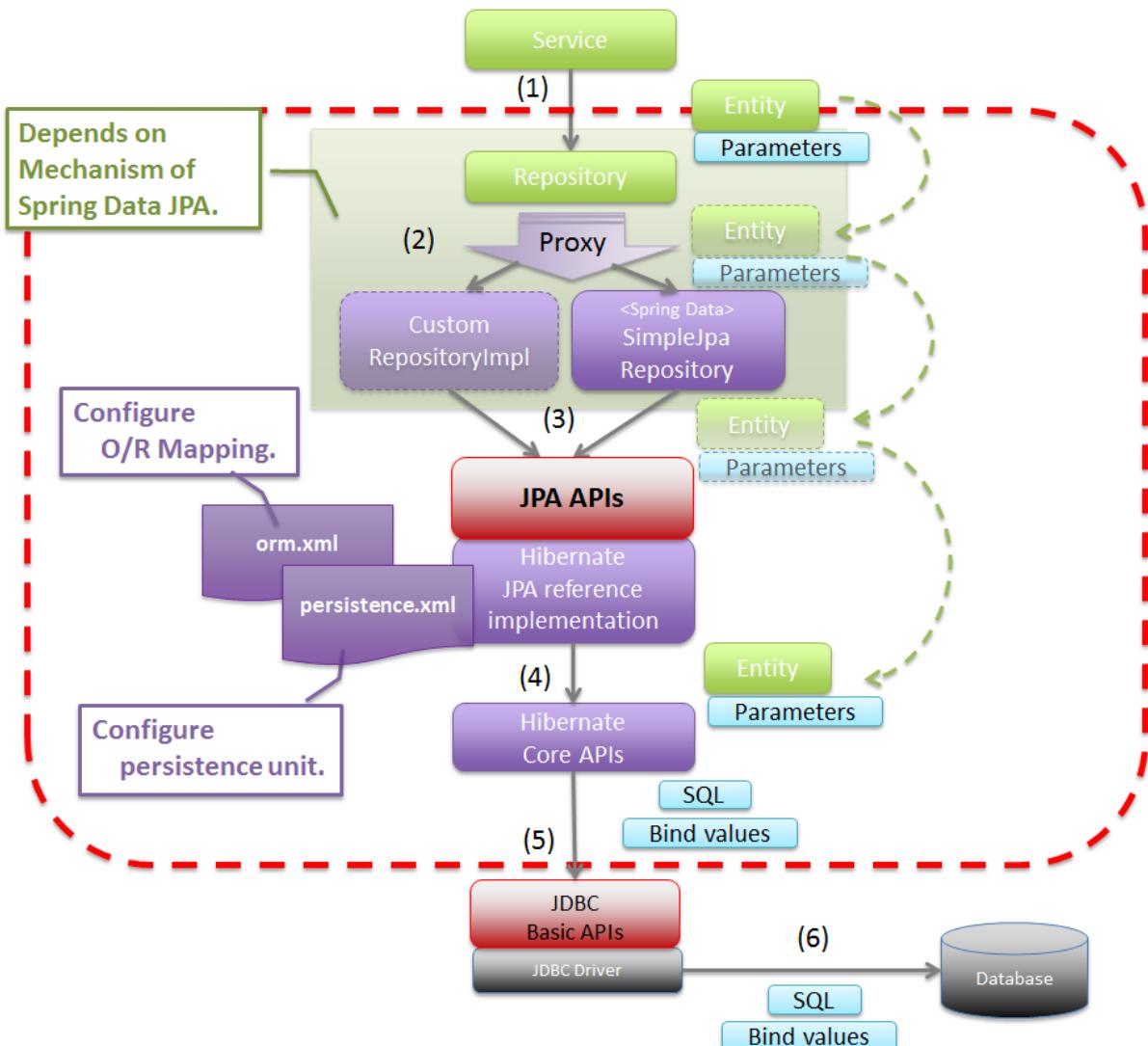


Figure.5.5 Picture - Basic flow of Spring Data JPA

Sr. No.	Description
(1)	Call the method of Repository interface from Service. Entity object, Entity ID etc. are passed as method calling parameters. In the above example, entity is passed, however a primitive value can also be passed.
(2)	Proxy class which dynamically implements Repository interface, delegates the process to <code>org.springframework.data.jpa.repository.support.SimpleJpaRepository</code> or custom Repository class. Parameters specified by Service are passed.
5.2. Database Access (JPA)	Repository implementation class calls JPA APIs. The parameters specified by Service and the parameters generated by implementation class of Repository are passed.

When creating the repository using Spring Data JPA, APIs of JPA need not be called directly; however, it is better to know which JPA method is being called by methods of Repository interface of Spring Data JPA.

The JPA methods called by main methods of Repository interface of Spring Data JPA are shown below.

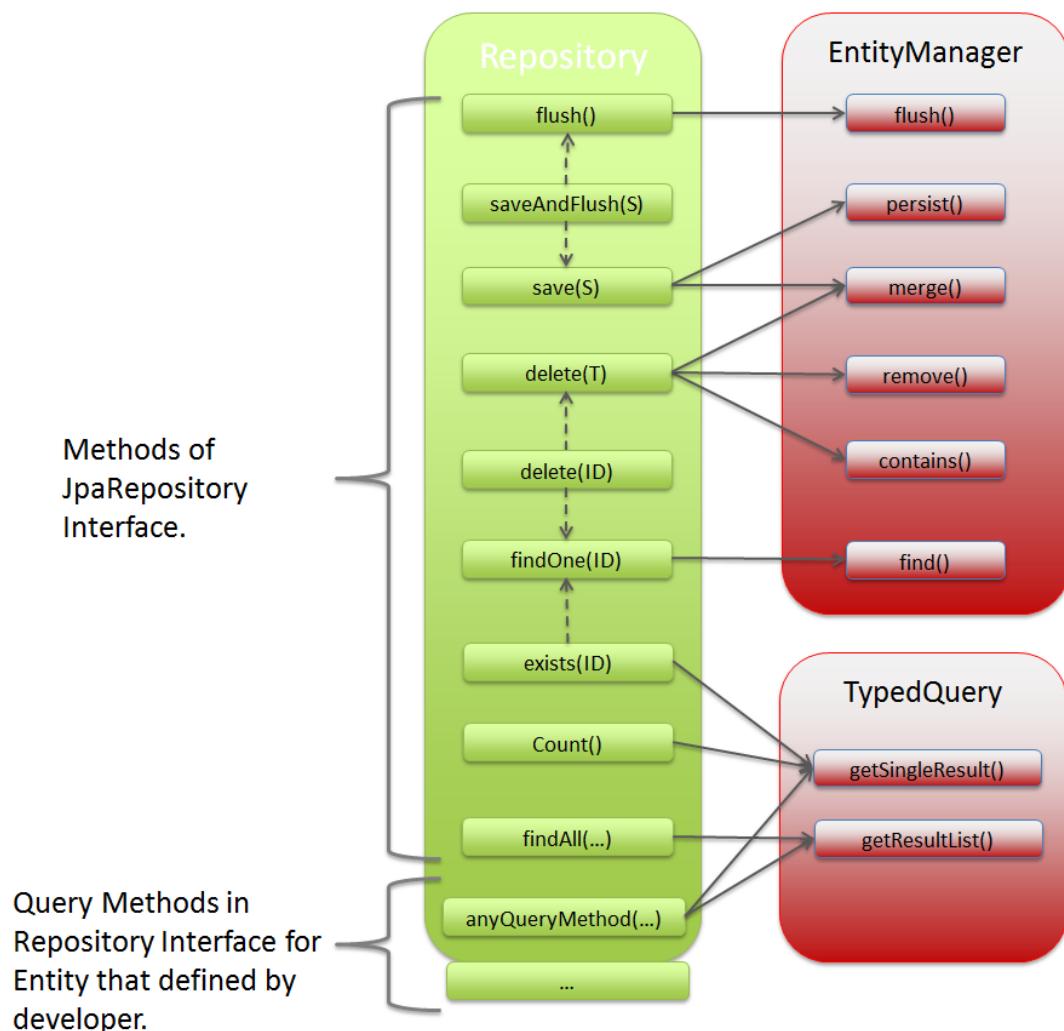


Figure.5.6 Picture - API Mapping of Spring Data JPA and JPA

5.2.2 How to use

pom.xml settings

When using JPA (Spring Data JPA) in infrastructure layer, add the following dependency to pom.xml

```
<!-- (1) -->
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-jpa</artifactId>
</dependency>
```

Sr. No.	Description
(1)	terasoluna-gfw-jpa where the libraries associated with JPA are defined should be added to dependency.

Application Settings

Datasource settings

Set connection information of the database to datasource.

For datasource settings, refer to [Datasource settings](#).

EntityManager settings

Perform settings to use EntityManager.

- xxx-infra.xml

```
<!-- (1) -->
<bean id="jpaVendorAdapter"
      class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <!-- (2) -->
    <property name="showSql" value="false" />
    <!-- (3) -->
    <property name="database" value="POSTGRESQL" />
</bean>

<!-- (4) -->
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <!-- (5) -->
    <property name="packagesToScan" value="xxxxxxxx.yyyyyyy.zzzzzz.domain.model" />
    <!-- (6) -->
```

```
<property name="dataSource" ref="dataSource" />
<!-- (7) -->
<property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
<!-- (8) -->
<property name="jpaPropertyMap">
    <util:map>
        <entry key="hibernate.hbm2ddl.auto" value="none" />
        <entry key="hibernate.ejb.naming_strategy"
               value="org.hibernate.cfg.ImprovedNamingStrategy" />
        <entry key="hibernate.connection.charSet" value="UTF-8" />
        <entry key="hibernate.show_sql" value="false" />
        <entry key="hibernate.format_sql" value="false" />
        <entry key="hibernate.use_sql_comments" value="true" />
        <entry key="hibernate.jdbc.batch_size" value="30" />
        <entry key="hibernate.jdbc.fetch_size" value="100" />
    </util:map>
</property>
</bean>
```

Sr. No.	Description
(1)	<p>Specify the adapter class associated with JPA provider. Hibernate will be used as JPA provider; hence specify <code>org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter</code>.</p> <p>Set the SQL output flag. In the example, “false: Do not output” has been specified.</p>
(2)	
(3)	<p>Set the value corresponding to RDBMS to be used. It is possible to specify the value defined in <code>org.springframework.orm.jpa.vendor.Database</code> enumerator type. In the example, “PostgreSQL” has been specified.</p> <p>[The value should be changed according to the database used in the project] If the database to be used changes with the environment, the value should be defined in properties file.</p>
(4)	<p>Specify FactoryBean class to create <code>javax.persistence.EntityManagerFactory</code> instance. Specify <code>org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean</code> .</p>
(5)	<p>Specify the package where entity classes are kept. The entity classes of the specified package can be managed using <code>javax.persistence.EntityManager</code>.</p> <p>[The value should be changed to the relevant package name according to the project]</p>
(6)	Specify the datasource to be used for accessing persistence layer (DB).
(7)	<p>Specify <code>JpaVendorAdapter</code> bean. Specify the bean set in (1).</p>
5.2. Database Access (JPA)	<p>(8) Specify the settings to configure <code>EntityManager</code> of Hibernate. For details, refer to “Hibernate Reference Documentation” .</p>

Tip: When using the Oracle database, ANSI standard SQL JOIN for combining tables, can be used by specifying the following settings in jpaPropertyMap mentioned in (8).

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <!-- omitted -->
    <property name="jpaPropertyMap">
      <util:map>
        <!-- omitted -->
        <entry key="hibernate.dialect"
              value="org.hibernate.dialect.Oracle10gDialect" /> <!-- (9) -->
      </util:map>
    </property>
  </bean>
```

Sr. No.	Description
(9)	Specify org.hibernate.dialect.Oracle10gDialect in "hibernate.dialect". By specifying Oracle10gDialect, ANSI standard SQL JOIN clause for combining the tables can be used.

Perform the following settings when transaction manager (JTA) of the application server is to be used.

The difference with the case wherein JTA is not used, is explained below.

For other locations, same settings as the case wherein JTA is not used can be performed.

- xxx-infra.xml

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <!-- omitted -->
    <!-- (10) -->
    <property name="jtaDataSource" ref="dataSource" />
    <!-- omitted -->
    <property name="jpaPropertyMap">
      <util:map>
        <!-- omitted -->
    </util:map>
  </property>
</bean>
```

```

<!-- (11) -->
<entry key="hibernate.transaction.jta.platform"
       value="org.hibernate.service.jta.platform.internal.WeblogicJtaPlatform" />

      </util:map>
    </property>
</bean>

```

Sr. No.	Description
(10)	<p>Specify the datasource to be used for accessing persistence layer (DB).</p> <p>When using JTA, specify the DataSource defined in application server in "jtaDataSource" property and not in "dataSource" property.</p> <p>Refer to <i>Datasource settings</i> of common edition for the method to fetch DataSource defined in application server.</p>
(11)	<p>Add JTA platform specification in "jpaPropertyMap" property.</p> <p>The above example illustrates usage of Weblogic JTA.</p> <p>The configurable value (platform) is FQCN of <code>org.hibernate.service.jta.platform.spi.JtaPlatform</code> implementation class.</p> <p>The implementation class for main application servers is provided by Hibernate.</p>

Note: When it is necessary to switch the transaction manager to be used as per the environment, then it is recommended that you define "entityManagerFactory" bean in `xxx-env.xml` instead of `xxx-infra.xml`.

An example wherein it is necessary to change the transaction manager as per environment can be: use of application server without JTA function such as Tomcat in case of local environment, and use of application server with JTA function such as Weblogic in case of production environment as well as various test environments.

PlatformTransactionManager settings

Perform the following settings when using local transaction.

- `xxx-env.xml`

```

<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager"> <!-- (1) -->
      <property name="entityManagerFactory" ref="entityManagerFactory" /> <!-- (2) -->

```

</bean>

Sr. No.	Description
(1)	Specify <code>org.springframework.orm.jpa.JpaTransactionManager</code> . This class controls transaction by calling APIs of JPA.
(2)	Specify Factory of <code>EntityManager</code> to be used in the transaction.

Perform the following settings when transaction manager (JTA) of the application server is to be used.

- `xxx-env.xml`

<`tx:jta-transaction-manager` /> <!-- (1) -->

Sr. No.	Description
(1)	The most appropriate <code>org.springframework.transaction.jta.JtaTransactionManager</code> is defined as bean with id as “transactionManager”, in the application server on which the application has been deployed. This class controls the transaction by calling JTA APIs.

persistence.xml settings

When using `LocalContainerEntityManagerFactoryBean`, there are no mandatory settings to be performed in `persistence.xml`.

Todo

TBD

Currently, there are no mandatory settings to be performed in `persistence.xml`; however such need may arise in future.

When using `EntityManagerFactory` in application server of Java EE, it may be necessary to perform few settings in `persistence.xml`; hence we are planning to provide maintenance for such settings in future.

Settings for validating Spring Data JPA

- `xxx-infra.xml`

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=".....
       http://www.springframework.org/schema/data/jpa
       http://www.springframework.org/schema/data/jpa/spring-jpa.xsd"> <!-- (1) -->

<!-- ... -->

</beans>

<jpa:repositories base-package="xxxxxx.yyyyyy.zzzzz.domain.repository" /> <!-- (2) -->

```

Sr. No.	Description
(1)	Import schema definition for Spring Data JPA configuration and assign ("jpa") as namespace.
(2)	Specify base package wherein Repository interface and custom Repository class are stored. Interface inheriting org.springframework.data.repository.Repository and interface with org.springframework.data.repository.RepositoryDefinition annotation are automatically defined as a bean of Repository class of Spring Data JPA.

•Attributes of <jpa:repositories> element

entity-manager-factory-ref, transaction-manager-ref, named-queries-location, query-lookup-strategy, factory-class and repository-impl-postfix are present as attributes.

Sr. No.	Element	Description
1.	entity-manager-factory-ref	<p>Specify Factory for generating EntityManager to be used in Repository. If multiple Factories of EntityManager are to be created, then it is necessary to specify the bean to be used.</p>
2.	transaction-manager-ref	<p>Specify PlatformTransactionManager to be used when the methods of Repository are called. The bean registered with "transactionManager" bean name is used by default.</p> <p>It needs to be specified when the bean name of PlatformTransactionManager to be used is not "transactionManager".</p>
3.	named-queries-location	<p>Specify the location of Spring Data JPA properties file wherein Named Query is specified. "classpath: META-INF/jpa-named-queries.properties" is used by default.</p>
4.	query-lookup-strategy	<p>Specify the method to Lookup the query to be executed when query method is called. By default, it is "CREATE_IF_NOT_FOUND". For details, refer to Spring Data JPA - Reference Documentation "Query lookup strategies". Use the default settings if there is no specific reason.</p>
5.	factory-class	<p>Specify Factory for generating class to implement the process when the method of Repository interface is called.</p> <p><code>org.springframework.data.jpa.repository.support.JpaRepository</code> is used by default. Specify the Factory created for changing default implementation of Spring Data JPA or for adding a new method.</p> <p>For how to add a new method, refer to Adding the custom methods to all Repository interfaces in batch.</p>
6.	repository-impl-postfix	<p>Specify suffix indicating that it is an implementation class of custom Repository.</p> <p>By default, it is "Impl". For example: when Repository interface name is OrderRepository, OrderRepositoryImpl will be the implementation class of custom Repository. Use the default settings if there is</p>

Settings for using JPA annotations

To inject `javax.persistence.EntityManagerFactory` and `javax.persistence.EntityManager` using the annotations (`javax.persistence.PersistenceContext` and `javax.persistence.PersistenceUnit`) provided by JPA, `org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor` should be defined as bean.

When `<jpa:repositories>` element is specified, bean is defined by default; hence no need to define it separately.

Settings for converting JPA exception to DataAccessException

To convert JPA exception to `DataAccessException` of Spring Framework, `org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor` should be defined as bean.

When `<jpa:repositories>` element is specified, bean is defined by default; hence no need to define it separately.

OpenEntityManagerInViewInterceptor settings

To perform Lazy Fetching of Entity in application layer such as Controller and JSP etc., the lifetime of `EntityManager` should be extended till application layer using `org.springframework.orm.jpa.support.OpenEntityManagerInViewInterceptor`.

When `OpenEntityManagerInViewInterceptor` is not to be used, the lifetime of `EntityManager` becomes same as that of transaction; hence it is necessary to either fetch the data required in application layer as a process of Service class or to use Eager Fetch instead of Lazy Fetch.

Considering the following perspectives, it is recommended that you use Lazy Fetch as fetch method and `OpenEntityManagerInViewInterceptor`.

- Fetching as a process of Service class leads to insignificant implementation such as calling only getter method or accessing the collection fetched by calling getter method.
- When Eager Fetch is used, it is likely that the data which is not used in application layer is also fetched impacting the performance.

See the example of `OpenEntityManagerInViewInterceptor` settings below.

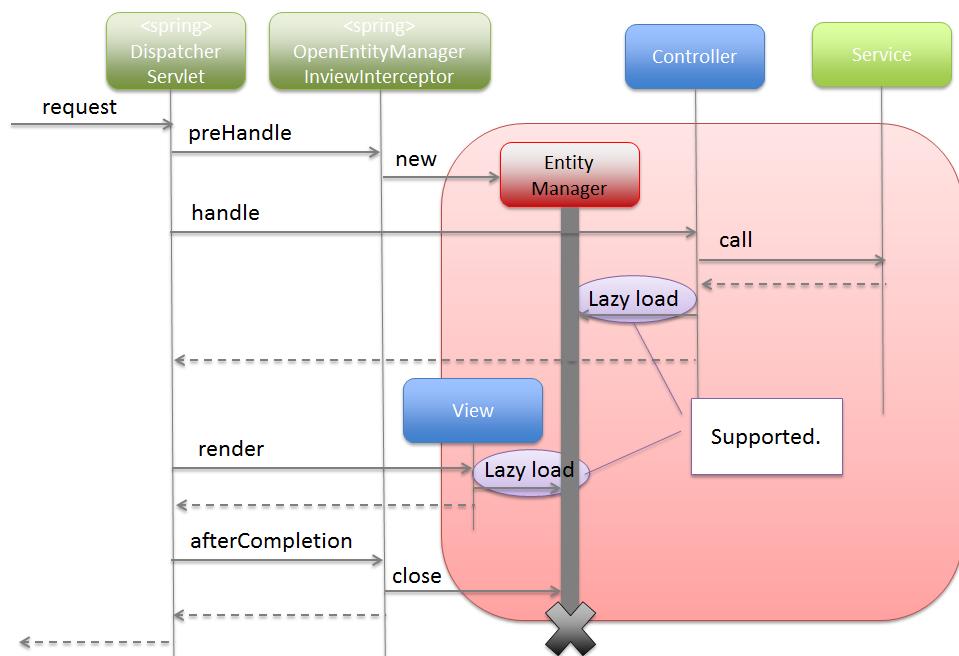


Figure.5.7 Picture - Lifetime of EntityManager on OpenEntityManagerInViewInterceptor

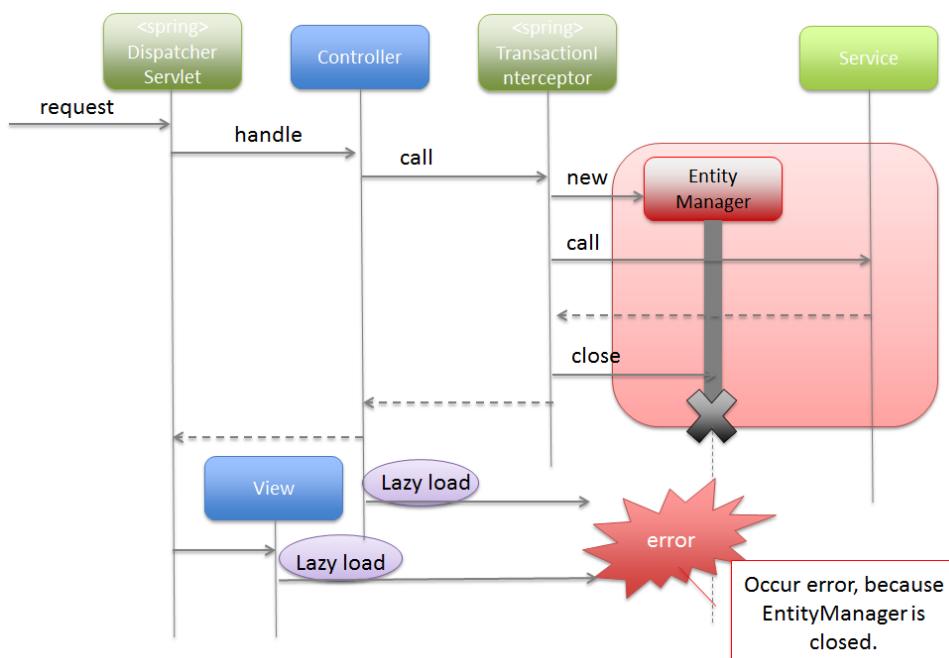


Figure.5.8 Picture - Default Life time of EntityManager

- spring-vmc.xml

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**" /> <!-- (1) -->
    <mvc:exclude-mapping path="/resources/**" /> <!-- (1) -->
    <mvc:exclude-mapping path="/**/*.html" /> <!-- (1) -->
    <!-- (2) -->
    <bean
      class="org.springframework.orm.jpa.support.OpenEntityManagerInViewInterceptor" />
  </mvc:interceptor>
</mvc:interceptors>
```

Sr. No.	Description
(1)	<p>Specify the path for which interceptor is to be applied and path for which interceptor is not to be applied.</p> <p>In this example, interceptor is being applied for paths other than paths of resource files (js, css, image etc.) and static web page (HTML).</p>
(2)	<p>Specify</p> <p><code>org.springframework.orm.jpa.support.OpenEntityManagerInViewInterceptor</code>.</p>

Note: Interceptor not to be applied to the path of static resources

It is recommended that interceptor not be applied to the path of static resources (js, css, image, html etc.), as there is no data access in such cases. Application of interceptor to the path of static resources leads to execution of unnecessary processes (such as instance generation and close process).

When Lazy Fetch is required in Servlet Filter, it is necessary to extend the lifetime of EntityManager till the Servlet Filter layer using

`org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter`.

For example, this case is applicable when

`org.springframework.security.core.userdetails.UserDetailsService` of SpringSecurity is inherited and if Entity object is accessed in the inherited logic.

However, if Lazy Fetch is not required, there is no need to extend the lifetime of EntityManager till the

Servlet Filter layer.

Note: About Lazy Fetch in Servlet Filter layer

It is recommended that you design and implement such that Lazy Fetch does not occur in Servlet Filter layer. If OpenEntityManagerInViewInterceptor is used, it is possible to specify the applicable and non-applicable URL patterns; thus the path for which lifetime of “EntityManager” is to be extended till application layer can also be easily specified. For the data access required in Servlet Filter, the data should either be fetched in advance in Service class or should be loaded in advance using Eager Fetch; thereby, avoiding the occurrence of Lazy Fetch.

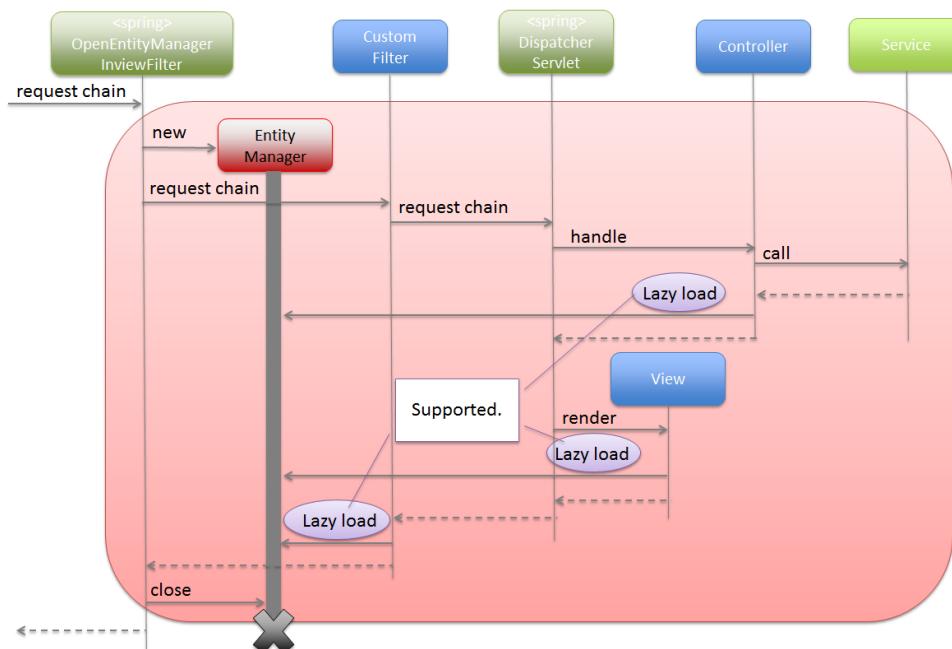


Figure.5.9 Picture - Lifetime of EntityManager on OpenEntityManagerInViewFilter

See the example of OpenEntityManagerInViewFilter settings below.

- web.xml

```
<!-- (1) -->
<filter>
  <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
  <filter-class>org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter</filter-class>
</filter>
<!-- (2) -->
```

```
<filter-mapping>
  <filter-name>Spring_OpenEntityManagerInViewFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Sr. No.	Description
(1)	Specify org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter. This Servlet Filter needs to be defined before the Servlet Filter in which Lazy Fetch occurs.
(2)	Specify the pattern of the URL for which filter is to be applied. It is recommended that you apply the filter only to the required path; however, if the settings are complicated, you can also specify “/*” (All Requests).

Note: If “/*” (All Requests) are specified in the pattern of the URL for which OpenEntityManagerInViewFilter is to be applied, OpenEntityManagerInViewInterceptor settings are not required.

Creating Repository interface

Spring Data provides the following 3 methods to create entity specific Repository interface.

Sr. No.	How to create	Description
1.	<i>Inheriting the interface of Spring Data</i>	Create entity specific Repository interface by inheriting from the interface of Spring Data. If there is no specific reason, then it is recommended that you create the entity specific Repository interface using this method.
2.	<i>Inheriting a common project specific interface in which only the required methods are defined</i>	Out of all the methods of Repository interface of Spring Data, create a common project specific interface wherein only the required methods are specified. Inherit the common interface to create entity specific Repository interface.
3.	<i>Not inheriting the interface</i>	Create entity specific Repository interface without inheriting the interface of Spring Data or common project specific common interface.

Inheriting the interface of Spring Data

The method to create entity specific Repository interface by inheriting from the interface of Spring Data is explained below.

Interfaces that can be inherited are as follows:

Sr. No.	Interface	Description
1.	org.springframework.data.repository.CrudRepository	Repository interface for generic CRUD operations.
2.	org.springframework.data.repository.PagingAndSortingRepository	Repository interface wherein Pagination function and Sort function are added to findAll method of CrudRepository.
3.	org.springframework.data.jpa.repository.JpaRepository	Repository interface that provides JPA specifications dependent methods. PagingAndSortingRepository is inherited; hence methods of PagingAndSortingRepository and CrudRepository can also be used. If there is no specific reason, it is recommended that you create entity specific Repository interface by inheriting this interface.

Note: About default implementation of Repository interface of Spring Data

The methods defined in the above interface are implemented using `org.springframework.data.jpa.repository.support.SimpleJpaRepository` of Spring Data JPA.

The example is given below.

```
public interface OrderRepository extends JpaRepository<Order, Integer> { // (1)  
}
```

Sr. No.	Description
(1)	Inherit JpaRepository and specify entity type in generic type <T> and entity ID type in generic type <ID extends Serializable>. In the above example, Order type is specified in entity and Integer type in entity ID.

If entity specific Repository interface is created by inheriting JpaRepository, then the following methods can be implemented.

Sr. No.	Method	Description
1.	<S extends T> S save(S entity)	<p>Method to accumulate persistence operations (INSERT/UPDATE) for the specified entity in <code>javax.persistence.EntityManager</code>.</p> <p>If the value is not set in ID property (property with <code>@javax.persistence.Id</code> annotation or <code>@javax.persistence.EmbeddedId</code> annotation), <code>persist</code> method of <code>EntityManager</code> is called and when the value is set, <code>merge</code> method is called.</p> <p>When merge method is called, please note that the returned Entity object is different from the Entity which is passed as an argument.</p>
2.	<S extends T> List<S> save(Iterable<S> entities)	<p>Method to accumulate persistence operations for multiple specified entities in <code>EntityManager</code>.</p> <p>The method is implemented by calling <code><S extends T> S save(S entity)</code> method repeatedly.</p>
3.	T saveAndFlush(T entity)	<p>Once the persistence operations for the specified entity are accumulated in <code>EntityManager</code>, this method reflects the accumulated persistence operations (INSERT/UPDATE/DELETE) in persistence layer (DB).</p>
4.	void flush()	<p>Method to execute persistence operations (INSERT/UPDATE/DELETE) for the entity accumulated in <code>EntityManager</code> in persistence layer (DB).</p>
5.	void delete(ID id)	<p>Method to accumulate delete operation for the entity of specified ID, in <code>EntityManager</code>.</p> <p>This method calls <code>T findOne(ID)</code> method and converts the entity object to “managed” state under <code>EntityManager</code> and then deletes that object.</p> <p>If entity is not present when <code>T findOne(ID)</code> method is called, <code>org.springframework.dao.EmptyResultDataAccessException</code> occurs.</p>
348	6. void delete(T entity)	<p>5 Architecture in Detail - TERASOLUNA Global Framework</p> <p>Method to accumulate delete operation for the specified entity, in <code>EntityManager</code>.</p>
	7. void delete(Iterable<? extends T>	

Warning: Behavior when using optimistic locking (@javax.persistence.Version) of JPA

When the entity is updated or deleted at the time of using optimistic locking (@Version) of JPA, org.springframework.dao.OptimisticLockingFailureException occurs. OptimisticLockingFailureException may occur in the following methods.

- <S extends T> S save(S entity)
- <S extends T> List<S> save(Iterable<S> entities)
- T saveAndFlush(T entity)
- void delete(ID id)
- void delete(T entity)
- void delete(Iterable<? extends T> entities)
- void deleteAll()
- void flush()

For details on optimistic locking of JPA, refer to [Exclusive Control](#).

Note: Timing to reflect persistence operations (1)

For the entity managed under EntityManager, accumulated persistence operations are executed just before committing a transaction and reflected in persistence layer (DB). Therefore, in order to handle errors such as unique constraint violation in transaction management (Service processing), it is necessary to call “saveAndFlush” method or “flush” method and execute persistence operations for the Entity accumulated in “EntityManager” forcibly. If only an error is to be notified to the client, it is OK to perform exception handling in Controller and set an appropriate message.

saveAndFlush and flush are JPA dependent methods, hence do not use these methods if there is no specific purpose.

- Normal flow
- flush flow

Note: Timing to reflect persistence operations (2)

When the following method is called, in order to avoid inconsistency between the data managed in EntityManager and persistence layer (DB), the persistence operations of the entity accumulated in EntityManager are reflected in the persistence layer (DB) before the main process is carried out.

- List<T> findAll method
- boolean exists(ID id)
- long count ()

In case of above methods, query is executed directly in the persistence layer (DB); hence inconsistency may

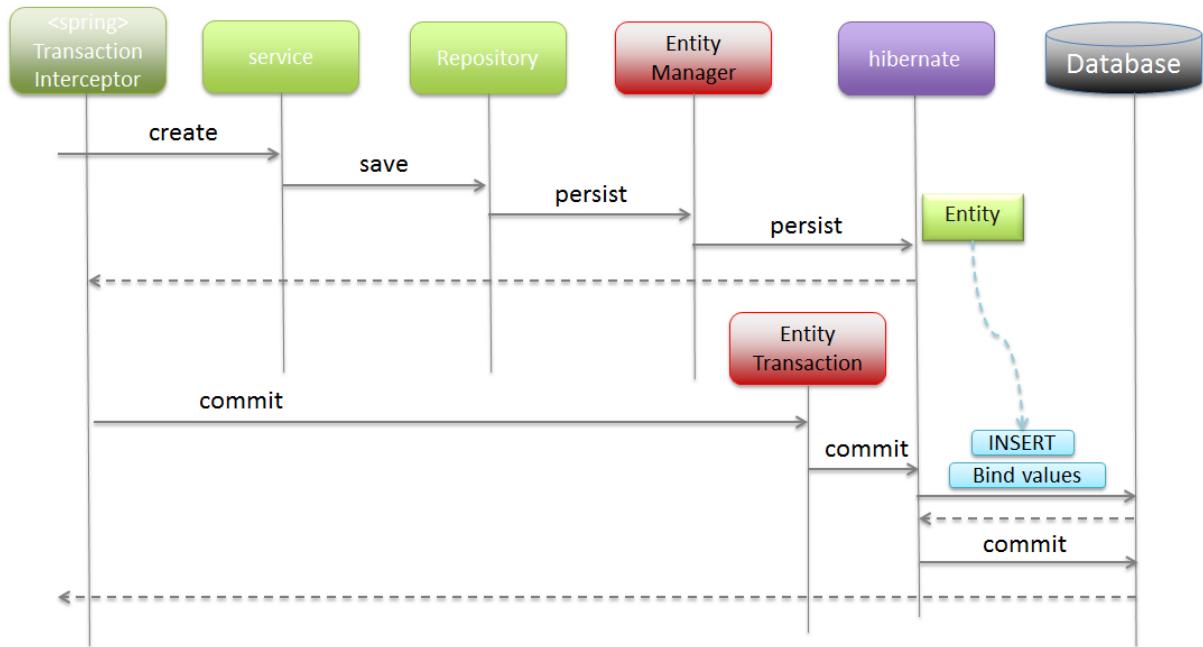


Figure.5.10 Picture - Normal sequence of persistence processing

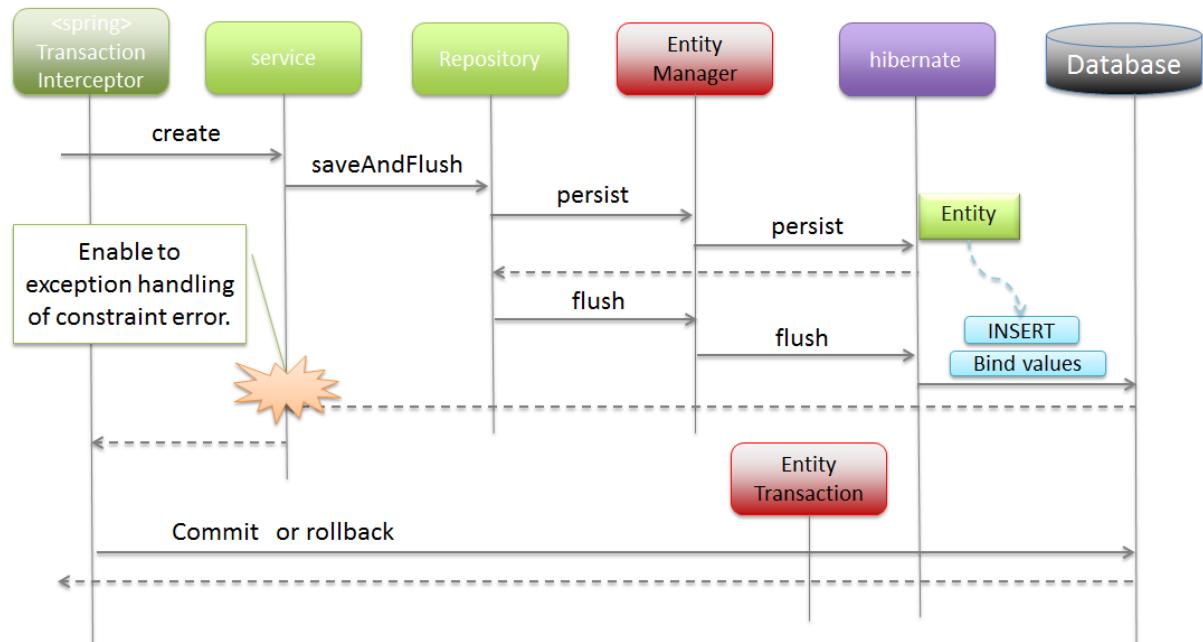


Figure.5.11 Picture - Sequence of persistence processing when flush method is used

occur unless the operations are reflected in the persistence layer (DB) before the main process is carried out. Calling of query methods described later also triggers the reflection of persistence operations for the entity accumulated in EntityManager, in the persistence layer (DB).

- Flow at the time of issuing queries

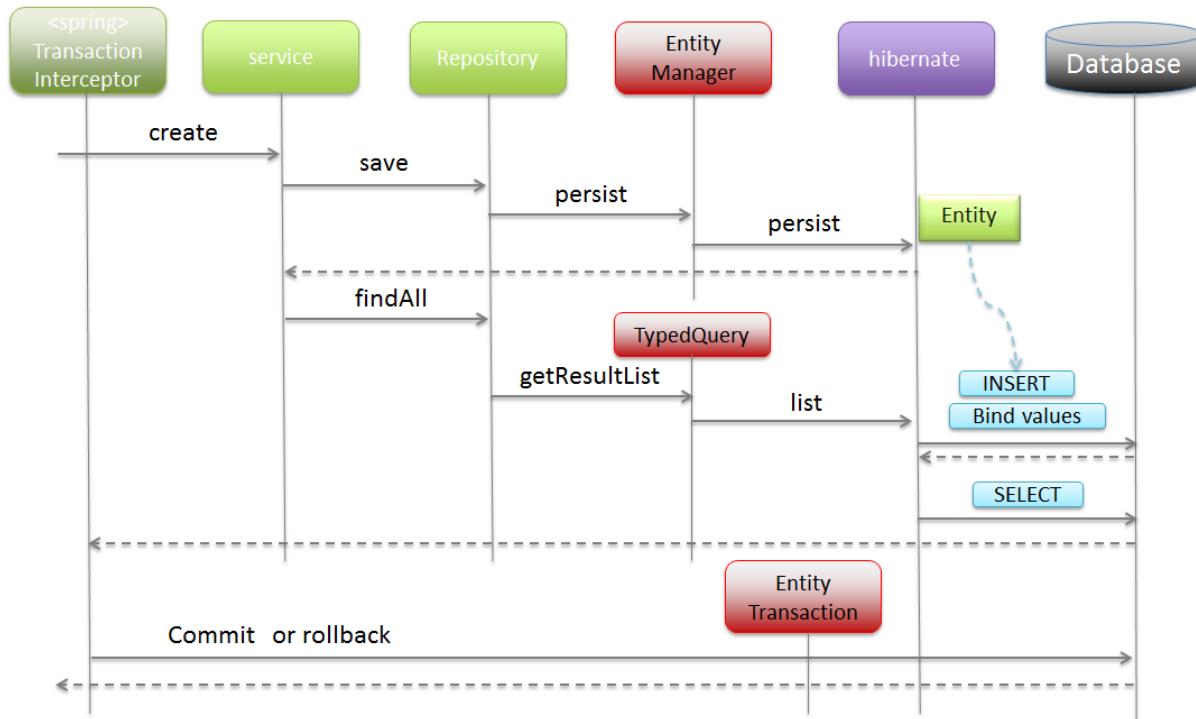


Figure 5.12 Picture - Sequence of persistence processing when query method is used

Inheriting a common project specific interface in which only the required methods are defined

Amongst the methods defined in interface of Spring Data, this section defines the method to create entity specific Repository interface by creating and inheriting a common project specific interface in which only the required methods are defined.

The signature of methods should match with the methods of Repository interface of Spring Data;

Note: Assumed cases

Amongst the methods of Repository interface of Spring Data, there are methods which are not used or which are not desirable to be used in the actual application. In order to remove such methods from Repository interface, refer below. The methods defined in interface are implemented using org.springframework.data.jpa.repository.support.SimpleJpaRepository of Spring Data JPA.

See the example below.

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

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

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

    // ...

}

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

}
```

Sr. No.	Description
(1)	Define a common interface for the project by inheriting <code>org.springframework.data.repository.Repository</code> . Use generic type since it is a not entity specific interface.
(2)	Select and define the required methods from the methods of Repository interface of Spring Data.
(3)	Inherit this common interface and specify the type of entity in generic type <code><T></code> and type of entity ID in generic type <code><ID extends Serializable></code> . In this example, <code>Order</code> type is specified in entity and <code>Integer</code> type in entity ID.

Not inheriting the interface

This section explains how to create entity specific Repository interface without inheriting any interface of Spring Data or common interface.

Specify `@org.springframework.data.repository.RepositoryDefinition` annotation as class annotation and specify entity type in `domainClass` attribute and entity ID type in `idClass` attribute.

The methods which have the same signature as methods defined in Repository interface of Spring Data need not be implemented.

Note: Assumed cases

Repository can be created in this way when common entity operations are not required. The methods having same signature as methods defined in Repository interface of Spring Data are implemented using `org.springframework.data.jpa.repository.support.SimpleJpaRepository` provided by Spring Data JPA.

See the example below.

```
@RepositoryDefinition(domainClass = Order.class, idClass = Integer.class) // (1)
public interface OrderRepository { // (2)

    Order findOne(Integer id); // (3)

    Order save(Order entity); // (3)

    // ...
}
```

Sr. No.	Description
(1)	Specify <code>@RepositoryDefinition</code> annotation. In the example, <code>Order</code> type is specified in <code>domainClass</code> attribute (entity type) and <code>Integer</code> type in <code>idClass</code> attribute (entity ID type).
(2)	There is no need to inherit the interface (<code>org.springframework.data.repository.Repository</code>) of Spring Data.
(3)	Define the methods required for each entity.

Adding query method

It is difficult to develop the actual application using only the Spring Data interface which is used for performing generic CRUD operations.

Therefore Spring Data provides a mechanism to add “query methods” for performing any persistence operations (SELECT/UPDATE/DELETE) for the entity specific Repository interface.

In the added query method, entity operations are performed using query language (JPQL or Native SQL).

Note: What is JPQL

JPQL is an abbreviation of “Java Persistence Query Language” and is the query language to perform entity operations (SELECT/UPDATE/DELETE) corresponding to the records of persistence layer (DB). The syntax is similar to SQL; however, JPQL operates the entities mapped to the records of persistence layer instead of operating these records directly. The entity operations are reflected to persistence layer (DB) using JPA provider (Hibernate).

For details on JPQL, refer to [JSR 317: Java Persistence API, Version 2.0 Specification \(PDF\)](#) “Chapter 4 Query Language” (P.131-178).

Defining query method

Query method is defined as a method of entity specific Repository interface.

```
public interface OrderRepository extends JpaRepository<Order, Integer> {  
    List<Order> findByStatusCode(String statusCode);  
}
```

Specifying query to be executed

Query to be executed should be specified at the time of calling query method.

The method of specifying the query is as below. For details, refer to [*Specifying a query while calling a query method.*](#)

Sr. No.	Method to specify a query	Description
1.	<p><i>@Query annotation</i> (Spring Data functionality)</p>	<p>In the method to be added to entity specific Repository interface, specify <code>@org.springframework.data.jpa.repository.Query</code> annotation and the query to be executed.</p> <p>When there is no specific reason, it is recommended that you specify the query using this method.</p>
2.	<p><i>Method name based on naming conventions</i> (Spring Data functionality)</p>	<p>Specify the query to be executed by assigning a method name as per Spring Data naming conventions.</p> <p>Query (JPQL) is generated from the method name using Spring Data JPA functionality. Only a SELECT clause of JPQL can be generated.</p> <p>For a simple query having few conditions, this method can be used instead of using @Query annotation. However, for a complex query with many conditions, a simple method name indicating behavior should be used and Query should be specified using <code>@Query</code> annotation.</p>
3.	<p><i>Named query of properties file</i> (Spring Data functionality)</p>	<p>Specify the query in a properties file.</p> <p>The location of method definition (entity specific Repository interface) and location wherein the query is specified (properties file) are separated; hence this way of specifying the query is not recommended.</p> <p>However, when using Native SQL as query, check whether it is necessary to define the database dependent SQL in the properties file.</p> <p>In case of applications for which any database can be selected or when the database changes (or is likely to be changed) depending on execution environment, then it is necessary to specify the Query using this method and manage the Properties file as environment dependent material.</p>

Note: Using multiple query specification methods

Particularly, there is no restriction on using multiple query specification methods. Query specification methods and restriction on their concurrent usage should be determined in accordance with the project.

Note: Query Lookup methods

The operations would be as follows since the Spring Data default setting is CREATE_IF_NOT_FOUND.

1. Look for the query specified in @Query annotation.
2. Look for the corresponding query from Named query.
3. Create a query (JPQL) from method name and use it.
4. An error occurs when query (JPQL) cannot be created from method name.

For details on Query Lookup methods, refer to [Spring Data JPA - Reference Documentation “1.2.2 Defining query methods” - “Query lookup strategies”](#).

Fetching entity lock

To fetch entity lock, add @org.springframework.data.jpa.repository.Lock annotation to query method and specify the lock mode.

For details, refer to [Exclusive Control](#).

```
@Query(value = "SELECT o FROM Order o WHERE o.status.code = :statusCode ORDER BY o.id DESC")
@Lock(LockModeType.PESSIMISTIC_WRITE) // (1)
List<Order> findByStatusCode(@Param("statusCode") String statusCode);
```

```
-- (2) statusCode='accepted'
SELECT
    order0_.id AS id1_5_
    ,order0_.status_code AS status2_5_
FROM
    t_order order0_
WHERE
    order0_.status_code = 'accepted'
ORDER BY
    order0_.id DESC
FOR UPDATE
```

Sr. No.	Description
(1)	Specify the lock mode in value attribute of @Lock annotation. For the details on lock mode that can be specified, refer to Java Platform, Enterprise Edition API Specification .
(2)	Native SQL converted from JPQL.(DB to be used is PostgreSQL) In the example, LockModeType.PESSIMISTIC_WRITE has been specified; hence “FOR UPDATE” clause is added to SQL.

Operating the entities of Persistence Layer directly

It is recommended to perform update and delete operations on entity objects managed in EntityManager.

However, when entities need to be updated or deleted in a batch, check whether the entities of persistence layer (DB) are operated using query method.

Note: Reducing the causes of performance degradation

Operating the entities of persistence layer directly reduces the frequency of SQLs that would be required to be executed for operating these entities. Therefore, in case of applications that demand high performance, the causes of performance degradation can be reduced by operating the entities in batch using this method. Such SQLs are as follows:

- SQL for loading all entity objects in EntityManager. Need not be executed.
 - SQL for updating and deleting entity. This SQL was earlier required to be executed n times, but now it is sufficient to execute it only once.
-

Note: Standards for deciding whether to operate entities of persistence layer directly

When operating the entities of persistence layer directly, since there are certain points to be careful about from functionality point of view, **in case of applications which do not demand high performance, it is recommended that the batch operations must also be performed through the entity objects managed in EntityManager.** For the points to be careful, refer to the example below.

The example of directly operating the entities of persistence layer using query method is shown below.

```
@Modifying // (1)
@Query("UPDATE OrderItem oi SET oi.logicalDelete = true WHERE oi.id.orderId = :orderId") //
int updateToLogicalDelete(@Param("orderId") Integer orderId); // (3)
```

Sr. No.	Description
(1)	Specify <code>@org.springframework.data.jpa.repository.Modifying</code> annotation indicating that the method is UPDATE query method. If not specified, error will occur at the time of execution.
(2)	Specify UPDATE or DELETE query.
(3)	If update count or delete count is required, specify <code>int</code> or <code>java.lang.Integer</code> as return value and if count is not required, specify <code>void</code> .

Warning: Consistency with entities managed in EntityManager

When entities of persistence layer are operated directly using query method, there is no change in the entities managed in EntityManager as per the default behavior of Spring Data JPA. Therefore, it should be noted that the entity object fetched immediately after calling `JpaRepository#findOne(ID)` method would be in a state prior to the state of operating the entities.

This behavior can be avoided by setting the `clearAutomatically` attribute of `@Modifying` annotation to `true`. When `clearAutomatically` attribute is set to `true`, `clear()` method of EntityManager is called after operating the entities of persistence layer directly, and the entity objects managed in EntityManager and the accumulated persistence operations are deleted from EntityManager. Therefore, if `JpaRepository#findOne(ID)` method is called immediately, the mechanism is such that the latest entity would be fetched from the persistence layer and EntityManager status would be synchronized with the persistence layer.

Warning: Points to be noted while using `@Modifying(clearAutomatically = true)`

By using `@Modifying(clearAutomatically = true)`, it should be noted that the accumulated persistence operations (INSERT/UPDATE/DELETE) are also deleted from EntityManager. Bugs may occur as the required persistence operations may not be reflected in the persistence layer.

In order to avoid this problem, `JpaRepository#saveAndFlush(T entity)` or `JpaRepository#flush()` method should be called and the accumulated persistence operations should be reflected in the persistence layer before directly operating the entities of persistence layer.

Setting QueryHints

When it is necessary to set a hint in query, add `@org.springframework.data.jpa.repository.QueryHints` annotation to query method and specify `QueryHint (@javax.persistence.QueryHint)` in value attribute.

```
@Query(value = "SELECT o FROM Order o WHERE o.status.code = :statusCode ORDER BY o.id DESC")
@Lock(LockModeType.PESSIMISTIC_WRITE)
@QueryHints(value = { @QueryHint(name = "javax.persistence.lock.timeout", value = "0") })
List<Order> findByStatusCode(@Param("statusCode") String statusCode);
```

Sr. No.	Description
(1)	<p>Specify hint name in name attribute of <code>@QueryHint</code> annotation and hint value in value attribute.</p> <p>In addition to the hint stipulated in JPA specifications, provider specific hint can be specified.</p> <p>In the above example, lock timeout is set to 0 (DB used is PostgreSQL). “FOR UPDATE NOWAIT” clause is added to SQL.</p>

Note: QueryHints that can be specified in Hibernate

QueryHints stipulated in JPA specifications are as follows: For details, refer to [JSR 317: Java Persistence API, Version 2.0 Specification \(PDF\)](#).

- `javax.persistence.query.timeout`
- `javax.persistence.lock.timeout`
- `javax.persistence.cache.retrieveMode`
- `javax.persistence.cache.storeMode`

For Hibernate specific QueryHints, refer to “3.4.1.8. Query hints” of [Hibernate EntityManager User guide](#).

Specifying a query while calling a query method

The method of specifying a query to be executed while calling query method is given below.

- *Specifying the query using `@Query` annotation*
- *Specifying with the method name based on naming conventions*
- *Specifying as Named query in Properties file*

Specifying the query using `@Query` annotation

Specify the query(JPQL) to be executed in value attribute of `@Query` annotation.

```
@Query(value = "SELECT o FROM Order o WHERE o.status.code = :statusCode ORDER BY o.id DESC")
List<Order> findByStatusCode(@Param("statusCode") String statusCode);
```

```
-- (2) statusCode='accepted'
SELECT
    order0_.id AS id1_5_
    ,order0_.status_code AS status2_5_
FROM
    t_order order0_
WHERE
    order0_.status_code = 'accepted'
ORDER BY
    order0_.id DESC
```

Sr. No.	Description
(1)	Specify the query(JPQL) to be executed in value attribute of @Query annotation. In the above example, query for fetching the Order object in the descending order of id property has been specified. Here, the code property (String type) value of status property (OrderStatus type) stored in Order object is matched with the specified parameter value (statusCode).
(2)	Native SQL converted from JPQL. Query(JPQL) specified in value attribute of @Query annotation is converted to Native SQL of database to be used.

Note: How to specify Native SQL directly instead of JPQL

Native SQL can be specified as query instead of JPQL by setting nativeQuery attribute to true. **Fundamentally it is recommended that you use JPQL; however, when there is a need to generate the query that cannot be expressed in JPQL, Native SQL can be specified directly.** To specify the database dependent SQL, analyze whether it can be defined in the properties file.

For method of defining SQL in properties file, refer to “[Specifying as Named query in Properties file](#)”.

Note: Named Parameters

Named parameter can be used by assigning a name to bind parameter of the query and using this assigned name to specify the value. To use Named Parameter, add @org.springframework.data.repository.query.Param annotation to the argument from which the value has to be used to bind to the named parameter in the query. Specify the assigned parameter name in value attribute of param annotation. On the query side, at the position where parameter is to be bound in the query, specify it in the “:parametername” format.

When there is no specific reason, It is recommended to use Named Parameters considering maintain-

ability and readability of query.

In case of LIKE search, if the type of matching (Forward match, Backward match and Partial match) is fixed, "% " can be specified in JPQL.

However, this is in extended Spring Data JPA format and not in standard JPQL, so it can be specified only in JPQL specified with @Query annotation.

An error occurs if "%" is specified in JPQL specified as Named query.

Sr. No.	Type of matching	Format	Specific example
1.	Forward match	:parameterName% or ?n%	SELECT a FROM Account WHERE a.firstName LIKE :firstName% SELECT a FROM Account WHERE a.firstName LIKE ?1%
2.	Backward match	%:parameterName or %?n	SELECT a FROM Account WHERE a.firstName LIKE %:firstName SELECT a FROM Account WHERE a.firstName LIKE %?1
3.	Partial match	%:parameterName% or %?n%	SELECT a FROM Account WHERE a.firstName LIKE %:firstName% SELECT a FROM Account WHERE a.firstName LIKE %?1%

Note: Escaping at the time of LIKE search

Search condition values should be escaped during LIKE search.

The method for escaping these values is provided in `org.terasoluna.gfw.common.query.QueryEscapeUtils` class; this class can be used if it meets the requirements. For details on `QueryEscapeUtils` class, refer to “*Escaping during LIKE search*” of “*Database Access (Common)*”.

Note: When the type of matching needs to be changed dynamically

When it is necessary to change the type of matching (Forward match, Backward match and Partial match) dynamically, "%" should be added before and after the parameter value (same as conventional method), instead of specifying % in JPQL.

The method for converting into search condition value corresponding to the type of matching is provided in `org.terasoluna.gfw.common.query.QueryEscapeUtils` class; this class can be used if it meets the requirements. For details on `QueryEscapeUtils` class, refer to “*Escaping during LIKE search*” of “*Database Access (Common)*”.

Sort conditions can be directly specified in query.

See the example below.

```
// (1)
@Query(value = "SELECT o FROM Order o WHERE o.status.code = :statusCode ORDER BY o.id DESC")
Page<Order> findByStatusCode(@Param("statusCode") String statusCode, Pageable pageable);
```

Sr. No.	Description
(1)	Specify "ORDER BY" in query. Specify DESC for sorting in descending order and ASC for ascending order. By default it is "ASC", when nothing is specified.

In addition to directly specifying the sort conditions in query, they can be specified in `org.springframework.data.domain.Sort` object stored in `Pageable` object.

When specifying the sort conditions using this method, there is no need to specify `countQuery` attribute.

Example of sorting using `Sort` object stored in `Pageable` object is shown below.

- Controller

```
@RequestMapping("list")
public String list(@PageableDefault(
    size=5,
    sort = "id", // (1)
    direction = Direction.DESC // (1)
) Pageable pageable,
Model model) {
```

```
Page<Order> orderPage = orderService.getOrders(pageable); // (2)
model.addAttribute("orderPage", orderPage);
return "order/list";
}
```

Sr. No.	Description
(1)	Specify the sort conditions. Sort conditions are set in <code>Sort</code> object that can be fetched by <code>Pageable#getSort()</code> method. In the above example, DESC is specified as a sort condition for id field.
(2)	Specify <code>Pageable</code> object and call Service method.

- Service (Caller)

```
public String getOrders(Pageable pageable) {
    return orderRepository.findByStatusCode("accepted", pageable); // (3)
}
```

Sr. No.	Description
(3)	Call Repository method by specifying <code>Pageable</code> object passed by Controller.

- Repository interface

```
@Query(value = "SELECT o FROM Order o WHERE o.status.code = :statusCode") // (4)
Page<Order> findByStatusCode(@Param("statusCode") String statusCode, Pageable pageable);

-- (5) statusCode='accepted'
SELECT
    COUNT(order0_.id) AS col_0_0_
FROM
    t_order order0_
WHERE
    order0_.status_code = 'accepted'
```

```
-- (6) statusCode='accepted'
SELECT
    order0_.id AS id1_5_
   ,order0_.status_code AS status2_5_
  FROM
    t_order order0_
 WHERE
    order0_.status_code = 'accepted'
 ORDER BY
    order0_.id DESC
 LIMIT 5
```

Sr. No.	Description
(4)	Do not specify “ORDER BY” clause in query. No need to specify countQuery attribute also.
(5)	Native SQL for count converted from JPQL.
(6)	Native SQL for fetching the entity of the specified page location converted from JPQL. Not specified in query; however “ORDER BY” clause is added to the condition specified in Sort object stored in Pageable object. In this example, it is SQL for PostgreSQL.

Specifying with the method name based on naming conventions

Specify the query (JPQL) to be executed through method name as per the naming conventions of Spring Data JPA.

JPQL is created from the method name by the functionality of Spring Data JPA.

However, this is only possible for SELECT queries and not for UPDATE and DELETE.

For naming conventions for creating JPQL, refer to the following pages.

Sr. No.	Reference page	Description
1.	Spring Data JPA - “Query creation” of Reference Documentation “1.2.2 Defining query methods”	This section describes method to specify Distinct, ORDER BY and Case insensitive.
2.	Spring Data JPA - “Property expressions” of Reference Documentation “1.2.2 Defining query methods”	This section describes method to specify the nested entity property in condition.
3.	Spring Data JPA - “Special parameter handling” of Reference Documentation “1.2.2 Defining query methods”	This section describes special method arguments (Pageable, Sort).
4.	Spring Data JPA - Reference Documentation “2.2.2. Query creation”	This section describes naming conventions (keywords) for creating JPQL.
5.	Spring Data JPA - Reference Documentation “Appendix B. Repository query keywords”	This section describes naming conventions (keywords) for creating JPQL.

See the example below.

- OrderRepository.java

```
Page<Order> findByStatusCode(String statusCode, Pageable pageable); // (1)
```

Sr. No.	Description
(1)	<p>When the method name matches with <code>^ (find read get) .*By (.+)</code> pattern, JPQL is created from method name.</p> <p>In the <code>(.+)</code> portion, specify the property of entity which forms the query condition or keywords indicating the operation.</p> <p>In the example, <code>Order</code> object where <code>code</code> property (String type) value of <code>status</code> property (OrderStatus type) stored in <code>Order</code> object is matched with the specified parameter value (<code>statusCode</code>), is being fetched in page format.</p>

- Count Query

```
-- (2) JPQL
SELECT
    COUNT(*)
FROM
    ORDER AS generatedAlias0
        LEFT JOIN generatedAlias0.status AS generatedAlias1
WHERE
    generatedAlias1.code = ?1

-- (3) SQL statusCode='accepted'
SELECT
```

```
COUNT(*) AS col_0_0_
FROM
    t_order order0_
        LEFT OUTER JOIN c_order_status orderstatu1_
            ON order0_.status_code = orderstatu1_.code
WHERE
    orderstatu1_.code = 'accepted'
```

Sr. No.	Description
(2)	JPQL query for count created from method name.
(3)	Native SQL for count converted from JPQL of step (2).

- Query for fetching entities

```
-- (4) JPQL
SELECT
    generatedAlias0
FROM
    ORDER AS generatedAlias0
        LEFT JOIN generatedAlias0.status AS generatedAlias1
WHERE
    generatedAlias1.code = ?1
ORDER BY
    generatedAlias0.id DESC;

-- (5) statusCode='accepted'
SELECT
    order0_.id AS id1_5_
    ,order0_.status_code AS status2_5_
FROM
    t_order order0_
        LEFT OUTER JOIN c_order_status orderstatu1_
            ON order0_.status_code = orderstatu1_.code
WHERE
    orderstatu1_.code = 'accepted'
ORDER BY
    order0_.id DESC
LIMIT 5
```

Sr. No.	Description
(4)	JPQL query for fetching entities created from method name.
(5)	Native SQL for fetching the entities converted from JPQL of step (4).

Specifying as Named query in Properties file

Specify the query in the properties file (classpath: META-INF/jpa-named-queries.properties) of Spring Data JPA.

Consider using this method when it is required to write a database platform specific SQL at the time of using NativeQuery.

Even if it is database platform specific SQL, it is recommended that you use a method to directly specify in @Query annotation when there is no dependency on the execution environment.

- OrderRepository.java

```
@Query(nativeQuery = true)
List<Order> findAllByStatusCode(@Param("statusCode") String statusCode); // (1)
```

Sr. No.	Description
(1)	Regarding the lookup name of named query, class name of entity and method name linked by ". " (dot) is used. In the above example, "Order.findAllByStatusCode" is used as Lookup name.

Tip: Specifying Lookup name of Named query

As per default behavior, Lookup name is constructed by connecting class name of the entity linked with method name using ". " (dot). However, any name can be specified.

- For fetching entities, specify it in name attribute of @Query annotation.
- For count at the time of page search, specify it in countName attribute of @Query annotation.

```
@Query(name = "OrderRepository.findAllByStatusCode", nativeQuery = true) // (2)
List<Order> findAllByStatusCode(@Param("statusCode") String statusCode);
```

Sr. No.	Description
(2)	In the above example, "OrderRepository.findAllByStatusCode" is specified as the Lookup name for the query.

- jpa-named-queries.properties

```
# (3)
Order.findAllByStatusCode=SELECT * FROM order WHERE status_code = :statusCode
```

Sr. No.	Description
(3)	Specify the SQL to be executed by using lookup name for the query as the key. In the above example, the SQL to be executed has been specified by using "Order.findAllByStatusCode" as key.

Tip: The method of specifying Named Query in any properties file instead of the properties file of Spring Data JPA is explained below.

- xxx-infra.xml

```
<!-- (4) -->
<jpa:repositories base-package="xxxxxxxx.yyyyyy.zzzzz.domain.repository"
    named-queries-location="classpath: META-INF/jpa/jpa-named-queries.properties" />
```

Sr. No.	Description
(4)	Specify any properties file in named-queries-location attribute of <jpa:repositories> element. In the above example, META-INF/jpa/jpa-named-queries.properties on class path is used.

Implementing the process to search entities

The method to search entities is explained below.

Searching all entities matching the conditions

Call a query method to fetch all entities that match the conditions.

- Repository interface

```
public interface AccountRepository extends JpaRepository<Account, String> {

    // (1)
    @Query("SELECT a FROM Account a WHERE :createdDateFrom <= a.createdDate AND a.createdDate <= :createdDateTo")
    List<Account> findByCreatedDate(
        @Param("createdDateFrom") Date createdDateFrom,
        @Param("createdDateTo") Date createdDateTo);

}
```

Sr. No.	Description
(1)	Define a query method to return java.util.List interface.

- Service

```
public List<Account> getAccounts(Date targetDate) {
    DateMidnight targetDateMidnight = new DateMidnight(targetDate);
    Date fromDate = targetDateMidnight.toDate();
    Date toDate = targetDateMidnight.dayOfYear().addToCopy(1).toDate();

    // (2)
    List<Account> accounts = accountRepository.findByCreatedDate(fromDate,
        toDate);
    if (accounts.isEmpty()) { // (3)
        // ...
    }
    return accounts;
}
```

Sr. No.	Description
(2)	Call the query method defined in Repository interface.
(3)	If the search result is 0 records, a blank list is returned. As null is not returned, null check is not required. If needed, implement the process when the search result is 0 records.

Searching page of entities matching the conditions

Amongst the entities matching the conditions, call a query method to fetch the entities of the specified page.

- Repository interface

```
public interface AccountRepository extends JpaRepository<Account, String> {

    // (1)
    @Query("SELECT a FROM Account a WHERE :createdDateFrom <= a.createdDate AND a.createdDate <= :createdDateTo")
    Page<Account> findByCreatedDate(
        @Param("createdDateFrom") Date createdDateFrom,
        @Param("createdDateTo") Date createdDateTo, Pageable pageable);

}
```

Sr. No.	Description
(1)	Receive <code>org.springframework.data.domain.Pageable</code> interface as an argument and define query method for returning <code>org.springframework.data.domain.Page</code> interface.

- Controller

```
@RequestMapping("list")
public String list(@RequestParam("targetDate") Date targetDate,
    @PageableDefaults(
        pageNumber = 0,
        value = 5,
        sort = { "createdDate" },
        sortDir = Direction.DESC)
    Pageable pageable, // (2)
    Model model) {
    Page<Order> accountPage = accountService.getAccounts(targetDate, pageable);
    model.addAttribute("accountPage", accountPage);
    return "account/list";
}
```

Sr. No.	Description
(2)	Create object (<code>org.springframework.data.domain.Pageable</code>) for paging search provided by Spring Data. For details, refer to “ Pagination ”.

- Service

```

public Page<Account> getAccounts(Date targetDate ,Pageable pageable) {

    DateMidnight targetDateMidnight = new DateMidnight(targetDate);
    Date fromDate = targetDateMidnight.toDate();
    Date toDate = targetDateMidnight.dayOfYear().addToCopy(1).toDate();

    // (3)
    Page<Account> page = accountRepository.findByCreatedDate(fromDate,
        toDate, pageable);
    if (!page.hasContent()) { // (4)
        // ...
    }
    return page;
}

```

Sr. No.	Description
(3)	Call the query method defined in Repository interface.
(4)	If the search result is 0 records, a blank list will be set in Page object and false will be returned for Page#hasContent() method. If needed, implement the process when the search result is 0 records.

Implementing search process as per the dynamic conditions of entities

To add query method to Repository for searching the entities as per dynamic conditions, search process should be implemented by creating custom Repository interface and custom Repository class for the entity specific Repository interface. For method of creating custom Repository interface and custom Repository class, refer to “[Adding individual custom method to entity specific Repository interface](#)”.

See the description below to search entities by applying dynamic conditions.

Todo

TBD

Following contents will be added in future.

- Example illustrating implementation of dynamic query using QueryDSL.
-

Searching all entities matching the dynamic conditions

Implement and call the query method for fetching all entities matching the dynamic conditions.

See the example below.

Here, the conditions below are specified as dynamic conditions.

- Order ID
- Product name
- Order status (multiple statuses can be specified)

Further, the search is narrowed down using AND operator for the orders matching the specified conditions. If no condition is specified, a blank list will be returned.

- Criteria (JavaBean)

```
public class OrderCriteria implements Serializable { // (1)

    private Integer id;

    private String itemName;

    private List<String> statusCodes;

    // ...

}
```

Sr. No.	Description
(1)	Create a Criteria object (JavaBean) storing the search conditions.

- Custom Repository interface

```
public interface OrderRepositoryCustom {

    Page<Order> findAllByCriteria(OrderCriteria criteria); // (2)

}
```

Sr. No.	Description
(2)	Define a method in custom Repository interface which receives Criteria object as an argument and returns List of Order objects.

- Custom Repository class

```

public class OrderRepositoryImpl implements OrderRepositoryCustom { // (3)

    @PersistenceContext
    EntityManager entityManager; // (4)

    public List<Order> findAllByCriteria(OrderCriteria criteria) { // (5)

        // Collect dynamic conditions.
        // (6)
        final List<String> andConditions = new ArrayList<String>();
        final List<String> joinConditions = new ArrayList<String>();
        final Map<String, Object> bindParameters = new HashMap<String, Object>();

        // (7)
        if (criteria.getId() != null) {
            andConditions.add("o.id = :id");
            bindParameters.put("id", criteria.getId());
        }
        if (!CollectionUtils.isEmpty(criteria.getStatusCodes())) {
            andConditions.add("o.status.code IN :statusCodes");
            bindParameters.put("statusCodes", criteria.getStatusCodes());
        }
        if (StringUtils.hasLength(criteria.getItemName())) {
            joinConditions.add("o.orderItems oi");
            joinConditions.add("oi.item i");
            andConditions.add("i.name LIKE :itemName ESCAPE '~'");
            bindParameters.put("itemName", SqlUtils
                .toLikeCondition(criteria.getItemName()));
        }

        // (8)
        if (andConditions.isEmpty()) {
            return Collections.emptyList();
        }

        // (9)
        // Create dynamic query.
        final StringBuilder queryString = new StringBuilder();

        // (10)
        queryString.append("SELECT o FROM Order o");
    }
}

```

```
// (11)
// add join conditions.
for (String joinCondition : joinConditions) {
    queryString.append(" LEFT JOIN ").append(joinCondition);
}
// add conditions.
Iterator<String> andConditionsIt = andConditions.iterator();
if (andConditionsIt.hasNext()) {
    queryString.append(" WHERE ").append(andConditionsIt.next());
}
while (andConditionsIt.hasNext()) {
    queryString.append(" AND ").append(andConditionsIt.next());
}

// (12)
// add order by condition.
queryString.append(" ORDER BY o.id");

// (13)
// Create typed query.
final TypedQuery<Order> findQuery = entityManager.createQuery(
    queryString.toString(), Order.class);
// Bind parameters.
for (Map.Entry<String, Object> bindParameter : bindParameters
    .entrySet()) {
    findQuery.setParameter(bindParameter.getKey(), bindParameter
        .getValue());
}

// (14)
// Execute query.
return findQuery.getResultList();

}
```

Sr. No.	Description
(3)	Create implementation class of custom Repository interface.
(4)	Inject EntityManager. Inject using <code>@javax.persistence.PersistenceContext</code> annotation.
(5)	Implement the query method to fetch all the entities matching the dynamic conditions. In the above example, the method is not split for explanation purpose; however it can be split if required.
(6)	Define the variables (list for AND condition, list for join condition, bind parameter map) to build dynamic queries. Set the required information in the variables for the items for which conditions are specified in OrderCriteria object.
(7)	Determine whether conditions are specified in OrderCriteria object and set the required information for building dynamic queries. In the above example, the information is set to fetch the items wherein <code>id</code> is completely matched with the specified value, <code>statusCodes</code> is included in the specified list, and <code>itemName</code> satisfying forward match with the specified value. For <code>itemName</code> , the related-entities having the values to be compared have a complex nested relation; hence these entities should be joined.
(8)	In the above example, when conditions are not specified, return a blank list as there is no need to perform search.
(9)	When conditions are specified, build the query for searching entities. In the above example, query to be executed is built using <code>java.lang.StringBuilder</code> class.
(10)	Build static query elements. 5.2. Database Access (JPA) In the above example, SELECT clause and FROM clause are built as static query elements.
(11)	Build dynamic query elements.

- Entity specific Repository interface

```
public interface OrderRepository extends JpaRepository<Order, Integer>,
    OrderRepositoryCustom { // (15)
    // ...
}
```

Sr. No.	Description
(15)	Inherit the custom Repository interface in entity specific Repository interface.

- Service (Caller)

```
// condition values for sample.
Integer conditionValueOfId = 4;
List<String> conditionValueOfStatusCodes = Arrays.asList("accepted");
String conditionValueOfItemName = "Wat";

// implementation of sample.
// (16)
OrderCriteria criteria = new OrderCriteria();
criteria.setId(conditionValueOfId);
criteria.setStatusCodes(conditionValueOfStatusCodes);
criteria.setItemName(conditionValueOfItemName);
List<Order> orders = orderRepository.findAllByCriteria(criteria); // (17)
if (orders.isEmpty()) { // (18)
    // ...
}
```

Sr. No.	Description
(16)	Specify the search conditions in OrderCriteria object.
(17)	Use OrderCriteria as an argument and call the query method to get all the entities matching the dynamic conditions.
(18)	Analyze the search results and perform the processing required in case of 0 records.

- Executed JPQL(SQL)

```
-- (19)
-- conditionValueOfId=4
-- conditionValueOfStatusCodes = ["accepted"]
-- conditionValueOfItemName = "Wat"
```

```
-- JPQL
SELECT
  o
FROM
  ORDER o
    JOIN o.orderItems oi
      JOIN oi.item i
WHERE
  o.id = :id
  AND o.status.code IN :statusCodes
  AND i.name LIKE :itemName ESCAPE '~'
ORDER BY
  o.id

-- SQL
SELECT
  order0_.id AS id1_6_
,order0_.created_by AS created2_6_
,order0_.created_date AS created3_6_
,order0_.last_modified_by AS last4_6_
,order0_.last_modified_date AS last5_6_
,order0_.status_code AS status6_6_
FROM
  t_order order0_ INNER JOIN t_order_item orderitems1_
    ON order0_.id = orderitems1_.order_id INNER JOIN m_item item2_
      ON orderitems1_.item_code = item2_.code
WHERE
  order0_.id = 4
  AND (
    order0_.status_code IN ('accepted')
  )
  AND (
    item2_.name LIKE 'Wat%' ESCAPE '~'
  )
ORDER BY
  order0_.id
```

Sr. No.	Description
(19)	Example of generated JPQL and SQL when all the conditions are specified.

```
-- (20)
-- conditionValueOfId=4
-- conditionValueOfStatusCodes = ["accepted"]
-- conditionValueOfItemName = ""
-- JPQL
SELECT
  o
FROM
```

```
    ORDER o
  WHERE
    o.id = :id
    AND o.status.code IN :statusCodes
  ORDER BY
    o.id

-- SQL
SELECT
  order0_.id AS id1_6_
 ,order0_.created_by AS created2_6_
 ,order0_.created_date AS created3_6_
 ,order0_.last_modified_by AS last4_6_
 ,order0_.last_modified_date AS last5_6_
 ,order0_.status_code AS status6_6_
FROM
  t_order order0_
WHERE
  order0_.id = 4
  AND (
    order0_.status_code IN ('accepted')
  )
ORDER BY
  order0_.id;
```

Sr. No.	Description
(20)	Example of generated JPQL and SQL when conditions other than itemName are specified.

```
-- (21)
-- conditionValueOfId=4
-- conditionValueOfStatusCodes = []
-- conditionValueOfItemName = ""
-- JPQL
SELECT
  o
FROM
  ORDER o
WHERE
  o.id = :id
ORDER BY
  o.id

-- SQL
SELECT
  order0_.id AS id1_6_
 ,order0_.created_by AS created2_6_
 ,order0_.created_date AS created3_6_
 ,order0_.last_modified_by AS last4_6_
```

```

,order0_.last_modified_date AS last5_6_
,order0_.status_code AS status6_6_
FROM
t_order order0_
WHERE
order0_.id = 4
ORDER BY
order0_.id;

```

Sr. No.	Description
(21)	Example of generated JPQL and SQL when only <code>id</code> is specified.

Page search for the entities matching the dynamic conditions

Implement and call the query method to fetch the entities corresponding to the specified page, amongst the entities matching the dynamic conditions.

As shown in the example below, the specification is same as that for normal search except for fetching the corresponding page. Further, the description for fetching all records is omitted.

- Custom Repository interface

```

public interface OrderRepositoryCustom {
    Page<Order> findPageByCriteria(OrderCriteria criteria, Pageable pageable); // (1)
}

```

Sr. No.	Description
(1)	Define the query method to fetch the entities corresponding to the specified page, amongst the entities matching the dynamic conditions.

- Custom Repository class

```

public class OrderRepositoryCustomImpl implements OrderRepositoryCustom {
    @PersistenceContext
    EntityManager entityManager;

    public Page<Order> findPageByCriteria(OrderCriteria criteria,

```

```
    Pageable pageable) { // (2)

    // collect dynamic conditions.
    final List<String> andConditions = new ArrayList<String>();
    final List<String> joinConditions = new ArrayList<String>();
    final Map<String, Object> bindParameters = new HashMap<String, Object>();

    if (criteria.getId() != null) {
        andConditions.add("o.id = :id");
        bindParameters.put("id", criteria.getId());
    }
    if (!CollectionUtils.isEmpty(criteria.getStatusCodes())) {
        andConditions.add("o.status.code IN :statusCodes");
        bindParameters.put("statusCodes", criteria.getStatusCodes());
    }
    if (StringUtils.hasLength(criteria.getItemName())) {
        joinConditions.add("o.orderItems oi");
        joinConditions.add("oi.item i");
        andConditions.add("i.name LIKE :itemName ESCAPE '~'");
        bindParameters.put("itemName", SqlUtils.toLikeCondition(criteria
            .getItemName()));
    }

    if (andConditions.isEmpty()) {
        List<Order> orders = Collections.emptyList();
        return new PageImpl<Order>(orders, pageable, 0); // (3)
    }

    // create dynamic query.
    final StringBuilder queryString = new StringBuilder();
    final StringBuilder countQueryString = new StringBuilder(); // (4)
    final StringBuilder conditionsString = new StringBuilder(); // (4)

    queryString.append("SELECT o FROM Order o");
    countQueryString.append("SELECT COUNT(o) FROM Order o"); // (5)

    // add join conditions.
    for (String joinCondition : joinConditions) {
        conditionsString.append(" JOIN ").append(joinCondition);
    }

    // add conditions.
    Iterator<String> andConditionsIt = andConditions.iterator();
    if (andConditionsIt.hasNext()) {
        conditionsString.append(" WHERE ").append(andConditionsIt.next());
    }
    while (andConditionsIt.hasNext()) {
        conditionsString.append(" AND ").append(andConditionsIt.next());
    }
    queryString.append(conditionsString); // (6)
    countQueryString.append(conditionsString); // (6)
```

```
// add order by condition.  
// (7)  
String orderByString = QueryUtils.applySorting("", pageable.getSort(), "o");  
queryString.append(orderByString);  
  
// create typed query.  
final TypedQuery<Long> countQuery = entityManager.createQuery(  
    countQueryString.toString(), Long.class); // (8)  
  
final TypedQuery<Order> findQuery = entityManager.createQuery(  
    queryString.toString(), Order.class);  
  
// bind parameters.  
for (Map.Entry<String, Object> bindParameter : bindParameters  
    .entrySet()) {  
    countQuery.setParameter(bindParameter.getKey(), bindParameter  
        .getValue()); // (8)  
    findQuery.setParameter(bindParameter.getKey(), bindParameter  
        .getValue());  
}  
  
long total = countQuery.getSingleResult().longValue(); // (9)  
List<Order> orders = null;  
if (total != 0) { // (10)  
    findQuery.setFirstResult(pageable.getOffset());  
    findQuery.setMaxResults(pageable.getPageSize());  
    // execute query.  
    orders = findQuery.getResultList();  
} else { // (11)  
    orders = Collections.emptyList();  
}  
  
return new PageImpl<Order>(orders, pageable, total); // (12)  
}  
}
```

Sr. No.	Description
(2)	<p>Implement the query method to fetch the entities corresponding to the specified page, amongst the entities matching the dynamic conditions.</p> <p>In the above example, the method is not split for explanation purpose; however it can be split if required.</p>
(3)	<p>In the above example, when conditions are not specified, return a blank page information as there is no need to perform search.</p>
(4)	<p>Create variables to build the queries to fetch records and to build the conditions (join condition and AND condition).</p> <p>Same conditions need to be used in query for fetching entities and in query for fetching records; hence variables are created to build the conditions (join condition and AND condition).</p>
(5)	<p>Build the static query elements of query to fetch records.</p> <p>In the above example, SELECT clause and FROM clause are built as static query components.</p>
(6)	<p>Build the dynamic query elements for query to fetch entities and query to fetch records.</p>
(7)	<p>Sort conditions (ORDER BY clause) are built as dynamic query elements for the query to fetch entities.</p> <p>Utility component <code>(org.springframework.data.jpa.repository.query.QueryUtils)</code> provided by Spring Data JPA is used for building the ORDER BY clause.</p>
(8)	<p>Convert the query string for fetching the dynamically built records into <code>javax.persistence.TypedQuery</code> and set the bind parameter for required for executing the query.</p>
(9)	<p>Execute the query for fetching the records and get the total number of records matching the conditions.</p>
382	<p style="text-align: center;">5 Architecture in Detail - TERASOLUNA Global Framework</p> <p>If the entities matching the conditions exist, execute the query for fetching the entities and fetch the information of the corresponding page.</p>

- Service (Caller)

```
// condition values for sample.
Integer conditionValueOfId = 4;
List<String> conditionValueOfStatusCodes = Arrays.asList("accepted");
String conditionValueOfItemName = "Wat";

// implementation of sample.
OrderCriteria criteria = new OrderCriteria();
criteria.setId(conditionValueOfId);
criteria.setStatusCodes(conditionValueOfStatusCodes);
criteria.setItemName(conditionValueOfItemName);
Page<Order> orderPage = orderRepository.findPageByCriteria(criteria,
    pageable); // (13)
if (!orderPage.hasContent()) {
    // ...
}
```

Sr. No.	Description
(13)	Call the query method to fetch the entities corresponding to the specified page, amongst the entities matching the dynamic conditions.

Implementing the process to fetch entities

The method of fetching entities is explained below.

Fetching 1 record of entity by specifying ID

If the ID (Primary Key) is known, fetch the entity object by calling the findOne method of Repository interface.

```
public Account getAccount(String accountUuid) {
    Account account = accountRepository.findOne(accountUuid); // (1)
    if (account == null) { // (2)
        // ...
    }
    return account;
}
```

Sr. No.	Description
(1)	Specify the ID (Primary Key) of entity and call the findOne(ID) method of Repository interface.
(2)	When the specified entity ID does not exist, the return value would be null, hence null check is necessary. If needed, implement the process which is carried out when the specified entity ID does not exist.

Note: Returned entity objects

When the entity object of specified ID is already managed by EntityManager, entity object managed by EntityManager is returned without accessing the persistence layer (DB). Therefore, if findOne method is used, unnecessary access to persistence layer can be controlled.

Note: Load timing of the related-entity

Load of the related-entity during query execution is determined based on the value specified in fetch attribute of annotations (@javax.persistence.OneToOne , @javax.persistence.OneToMany , @javax.persistence.ManyToOne , @javax.persistence.ManyToMany).

- In case of javax.persistence.FetchType#LAZY , related-entity is not covered under JOIN FETCH; hence it is loaded at the time of initial access.
- In case of javax.persistence.FetchType#EAGER , related-entity is covered under JOIN FETCH; hence it is loaded at the time of loading the parent-entity.

Default values of fetch attribute differ depending on annotations. See the default values below:

- @OneToOne annotation: EAGER
 - @ManyToOne annotation: EAGER
 - @OneToMany annotation: LAZY
 - @ManyToMany annotation: LAZY
-
-

Note: Sort order of the related-entities having 1:N(N:M) relationship

Specify @javax.persistence.OrderBy annotation on the property of related-entities to control the sort order.

See the example below.

```
@OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)
@OrderBy // (1)
private Set<OrderItem> orderItems;
```

Sr. No.	Description
(1)	When value attribute is not specified, the entities are sorted in ascending order of ID. For details, refer to JSR 317: Java Persistence API, Specification (PDF) of Version 2.0 “OrderBy Annotation” (P.404-406).

Todo

TBD

Following contents will be added in future.

- Example of cases wherein it is better to change fetch attribute from default value.

Fetching 1 record of entity by specifying conditions other than ID

When the ID is not known, call the query method to search entities by conditions other than ID.

- Repository interface

```
public interface AccountRepository extends JpaRepository<Account, String> {

    // (1)
    @Query("SELECT a FROM Account a WHERE a.accountId = :accountId")
    Account findByAccountId(@Param("accountId") String accountId);

}
```

Sr. No.	Description
(1)	Define the query method to search 1 record of entity by specifying items other than ID as conditions.

- Service

```
public Account getAccount(String accountId) {
    Account account = accountRepository.findById(accountId); // (2)
    if (account == null) { // (3)
        ...
    }
}
```

Sr. No.	Description
(2)	Call the query method defined in Repository interface.
(3)	Similar to findOne method of Repository interface, when the entities matching the conditions do not exist, null will be returned. Hence null check is necessary. If needed, implement the process for the scenario when entities matching the specified conditions do not exist. When multiple entities matching the conditions exist, <code>org.springframework.dao.IncorrectResultSizeDataAccessException</code> occurs.

Note: Returned entity objects

When a query method is called, the query is always executed on persistence layer (DB). However, when the entities which are fetched by executing the query are already being managed in EntityManager, the entity objects fetched from DB are discarded and the entity objects managed in EntityManager are returned.

Note: Query method using ID + α as condition

It is recommended not to create a query method wherein ID + α is used as condition. It can be implemented by creating a logic that compares property value of entity objects fetched by calling findOne method.

The reason for not recommending the creation of this query method is that there is a possibility of the entity objects fetched by executing the query getting discarded and unnecessary query getting executed. If the ID is known, it is desirable to use findOne method which prevents execution of unnecessary queries. This should be consciously implemented especially in case of applications with high performance requirements.

However, if the following conditions are applicable, use of query method may reduce the frequency of query execution; hence query method can be used in such cases.

- Properties of related objects (column of related table) are included in a part of + α condition.
 - LAZY is included in FetchType of the related-entities as a condition.
-
-

Note: Load timing of the related-entities

Related-entities specified in JOIN FETCH are loaded immediately after executing the query.

The related-entities not specified in JOIN FETCH performs the following operations as per the values specified in fetch attribute of associated annotations (@OneToOne , @OneToMany , @ManyToOne , @ManyToMany).

- javax.persistence.FetchType#LAZY is covered under Lazy Load; hence the related-entities are loaded at the time of initial access.
 - In case of javax.persistence.FetchType#EAGER, query is executed to load the related-entities and objects of the related-entities are loaded.
-
-

Note: Sort order of the related-entities having 1:N(N:M) relationship

- The sort order of the related-entities specified in JOIN FETCH is controlled by specifying “ORDER BY” clause in JPQL.
 - The sort order of the related-entities loaded after executing the query is controlled by specifying @javax.persistence.OrderBy annotation to the property of the related-entities.
-

Adding entities

Example of adding entities is shown below.

How to add entities

In order to add an entity, create an entity object and call the save method of Repository interface.

- Service

```
Order order = new Order("accepted"); // (1)
order = orderRepository.save(order); // (2)
```

Sr. No.	Description
(1)	<p>Create an instance of entity object and set the values for required properties.</p> <p>In the above example, ID generator of JPA is used for setting the ID. When ID generator of JPA is to be used, ID should not be set in the application code.</p> <p>If you set the ID in the application code, merge method of EntityManager gets called leading to execution of unnecessary processing.</p>
(2)	<p>Call the save method of Repository interface and manage the entity objects created in (1) in EntityManager.</p> <p>Take note that entity object passed as an argument of save method will not be the one managed by EntityManager, but the entity object returned by save method will be the one managed by EntityManager.</p> <p>ID is set by the ID generator of JPA at the time of this process.</p>

Note: Demerits of the merge method getting called

merge method of EntityManager has a mechanism to fetch the entities having same ID from persistence layer (DB), when the entities are to be managed in EntityManager. Process to fetch the entities becomes unnecessary while adding the entities. In case of application with high performance requirements, ID generation timing should also be taken into account.

Note: Constraint error handling

When save method is called, query (INSERT) is not executed in the persistence layer (DB). Therefore, when constraint error such as unique constraint violation needs to be handled, saveAndFlush method or flush method should be called instead of save method of Repository interface.

- Entity

```
@Entity // (3)
@Table(name = "t_order") // (4)
public class Order implements Serializable {

    // (5)
    @SequenceGenerator(name = "GEN_ORDER_ID", sequenceName = "s_order_id",
                       allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                   generator = "GEN_ORDER_ID")
    @Id // (6)
    private int id;
```

```
// (7)
@ManyToOne
@JoinColumn(name = "status_code")
private OrderStatus status;

// ...

public Order(String statusCode) {
    this.status = new OrderStatus(statusCode);
}

// ...

}
```

Sr. No.	Description
(3)	Assign <code>@javax.persistence.Entity</code> annotation to the entity class.
(4)	Specify the table name to be mapped with the entity class in name attribute of <code>@javax.persistence.Table</code> annotation. The table name need not be specified if it can be resolved from entity name; however, it has to be specified if it cannot be resolved from entity name.
(5)	The annotations required for using ID generator of JPA are being specified. When using ID generator of JPA, specify <code>@javax.persistence.GeneratedValue</code> annotation. When using sequence object, <code>@javax.persistence.SequenceGenerator</code> annotation should be specified. When using table generator, <code>@javax.persistence.TableGenerator</code> annotation should be specified. In the above example, ID is generated using the sequence object called "s_order_id" name.
(6)	Assign <code>@javax.persistence.Id</code> annotation to the property that holds the primary key. In case of composite key, assign <code>@javax.persistence.EmbeddedId</code> annotation.
(7)	Assign the associated annotations (<code>@OneToOne</code> , <code>@OneToMany</code> , <code>@ManyToOne</code> , <code>@ManyToMany</code>) to the property that has a relationship with other entities.

Note: Annotations for generating IDs

For details on each annotation, refer to [JSR 317: Java Persistence API, Version 2.0 Specification \(PDF\)](#).

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

Note: About method of generating IDs

For generating ID, specify the value of `javax.persistence.GenerationType` in strategy

attribute of @GeneratedValue annotation. Values that can be specified are as follows:

- TABLE: Generate ID using persistence layer (DB) table.
- SEQUENCE: Generate ID using sequence object of persistence layer (DB).
- IDENTITY: Generate ID using identity column of persistence layer (DB).
- AUTO: Generate ID by selecting the most appropriate method in persistence layer (DB).

Generally it is recommended to explicitly specify the type to be used instead of using AUTO.

Adding parent-entity and related-entity

In order to add parent-entity and related-entity together, call the save method of Repository interface and manage the entity objects under EntityManager. Then create the related-entity objects and map them with parent-entity objects.

In order to use this method, persist needs to be included in the cascade operation of the related-entity.

Note: Behavior of entities in case of cascade operations

When the related-entity is specified under cascade operations, JPA operations for parent-entity (persist, merge, remove, refresh, detach) are linked with related-entity and then carried out.

Mapping with the operations of Repository interface of Spring Data JPA is as follows:

- save method : persist or merge
- delete method : remove

refresh, detach are not executed in default implementation of Spring Data JPA.

- Service

```
String itemCode = "ITM0000001";
int itemQuantity = 10;
String wayToPay = "card";

Order order = new Order("accepted");
order = orderRepository.save(order); // (1)

OrderItem orderItem = new OrderItem(order.getId(), itemCode,
    itemQuantity);
```

```
order.setOrderItems(Collections.singleton(orderItem)); // (2)
order.setOrderPay(new OrderPay(order.getId(), wayToPay)); // (2)
```

Sr. No.	Description
(1)	<p>First, call the save method of Repository interface and bring the entity object under EntityManager.</p> <p>In the above example, save method is being called before setting the related-entity objects. This is because it is necessary to generate the ID (orderId) of parent-entity; this ID is used as a part of ID of the related entity.</p>
(2)	<p>Set the related-entity objects for the parent-entity object (that are to be managed under EntityManager).</p> <p>When committing the transaction, persist operation (INSERT) on parent-entity object is linked with the related-entity objects.</p>

- Entity

```
@Entity
@Table(name = "t_order")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @SequenceGenerator(name = "GEN_ORDER_ID", sequenceName = "s_order_id",
                       allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                   generator = "GEN_ORDER_ID")
    private int id;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL, // (3)
               orphanRemoval = true)
    @OrderBy
    private Set<OrderItem> orderItems;

    @OneToOne(mappedBy = "order", cascade = CascadeType.ALL,
               orphanRemoval = true)
    private OrderPay orderPay;

    @ManyToOne
    @JoinColumn(name = "status_code")
    private OrderStatus status;

    // ...
}
```

```

public Order(String statusCode) {
    this.status = new OrderStatus(statusCode);
}

// ...
}

```

Sr. No.	Description
(3)	<p>Specify the cascade operation type (<code>javax.persistence.CascadeType</code>) in <code>cascade</code> attribute in corresponding annotation.</p> <p>In the above example, all operations are target of cascade to the related-entity.</p> <p>If there is no specific reason, it is recommended to consider all the operations for cascade.</p>

Warning: Related-entity for which cascade attribute should not be specified

If the transaction entity is associated with code and master entities, cascade attribute should not be specified. In the above example, `OrderStatus` is the code entity; hence cascade attribute should not be specified in `@ManyToOne` annotation of `status` property of `Order`.

- Related-entity

```

@Entity
@Table(name = "t_order_item")
public class OrderItem implements Serializable {

    @EmbeddedId
    private OrderItemPK id;

    private int quantity;

    @ManyToOne
    @JoinColumn(name = "order_id", insertable = false, updatable = false)
    private Order order;

    // ...

    public OrderItem(Integer orderId, String itemCode, int quantity) {
        this.id = new OrderItemPK(orderId, itemCode);
        this.quantity = quantity;
    }

    // ...
}

```

```
@Entity
@Table(name = "t_order_pay")
public class OrderPay implements Serializable {

    @Id
    @Column(name = "order_id")
    private Integer orderId;

    @Column(name = "way_to_pay")
    private String wayToPay;

    @OneToOne
    @JoinColumn(name = "order_id")
    private Order order;

    // ...

    public OrderPay(int orderId, String wayToPay) {
        this.orderId = orderId;
        this.wayToPay = wayToPay;
    }

    // ...
}
```

Adding the related-entity

In order to add a related-entity, link the newly created related-entity object with the parent-entity object fetched through Repository interface.

For using this method, `persist` and `merge` should be included in the cascade operation of the related-entity.

```
String itemCode = "ITM000003";
int quantity = 30;

Order order = orderRepository.findOne(orderId); // (1)

OrderItem orderItem = new OrderItem(order.getId(), itemCode, quantity);
order.getOrderItems().add(orderItem); // (2)

OrderPay orderPay = order.getOrderPay();
if (orderPay == null) {
    order.setOrderPay(new OrderPay(order.getId(), "cash")); // (3)
} else {
```

```

        orderPay.setWayToPay("cash");
    }
}

```

Sr. No.	Description
(1)	Fetch the entity object using the Repository interface.
(2)	In case of entity with 1: N relationship, add the related-entity object to the collection fetched from the parent-entity object.
(3)	In case of entity with 1:1 relationship, set the related-entity object in the parent-entity object.

Adding the related-entity directly

When a related-entity object is to be added directly without linking it with the parent-entity object, save it using the Repository interface of related-entity.

```

String itemCode = "ITM000003";
int quantity = 40;

OrderItem orderItem = new OrderItem(orderId, itemCode, quantity); // (1)

orderItemRepository.save(orderItem); // (2)

```

Sr. No.	Description
(1)	Create an object of related-entity.
(2)	Call the save method of Repository interface of the related-entity.

Note: Merits of saving the related-entity object directly

The number of objects created is less. When fetching the parent-entity object, related-entity objects which are not necessary for the processing may also get created.

Note: Demerits of saving the related-entity directly

When ID of the parent-entity is being used as a part of the related-entity ID, the ID should be set before calling the save method. If ID is set, merge method of EntityManager is called as per the default implementation of Spring Data JPA. Therefore, the entity with same ID is always fetched from persistence layer (DB).

Warning: Points to be noted at the time of using parent-entity object after the related-entity is added

When the related-entity is added using Repository save method of related-entity, it cannot be fetched through the parent-entity object since it is not linked with the parent-entity object.

To avoid this problem,

1. Related-entity object should not be added directly. It should first be linked with the parent-entity object, and then added.
2. Synchronization with persistence layer (DB) should be done before fetching the parent-entity, using saveAndFlush method.

In such a case, if there is no specific reason, objects of the related-entity should be added by the former method. In the latter method, the problem cannot be avoided if the parent-entity object is already the “managed” entity.

Updating entities

The example of updating entities is explained below.

How to update entities

In order to update the entity, set the changed value to the entity object fetched using the Repository interface method.

```
Order order = orderRepository.findOne(orderId); // (1)
order.setStatus(new OrderStatus("checking")); // (2)
```

Sr. No.	Description
(1)	Fetch the entity object using Repository interface method.
(2)	Update the state of entity object by calling setter method.

Note: Calling the save method of Repository

The entity object fetched using Repository interface method are managed under EntityManager. For the entity objects managed under EntityManager, just by changing the state of objects using setter method, the changes are reflected in persistence layer (DB) at the time of committing the transaction. Therefore, there is no need to explicitly call the save method of Repository interface.

However, save method needs to be called when the entity objects are not managed under EntityManager. For example, when entity object is created based on the request parameters sent from the screen.

Updating the related-entity

In order to update the related-entity, first fetch the related-entity from the parent entity which in turn can be fetched using Repository interface and then set the values to be updated in related-entity object.

For using this method, merge should be included in the cascade operation of related-entity.

```
Order order = orderRepository.findOne(orderId); // (1)

for (OrderItem orderItem : order.getOrderItems()) {
    int newQuantity = quantityMap.get(orderItem.getId().getItemCode());
    orderItem.setQuantity(newQuantity); // (2)
}

order.getOrderPay().setWayToPay("cash"); // (3)
```

Sr. No.	Description
(1)	Use the Repository interface method to fetch entity objects.
(2)	In case of entity with 1:N relationship, the state of related-entity object stored in the collection fetched from parent-entity object is to be updated by calling the setter method.
(3)	In case of entity with 1:1 relationship, the state of related-entity object fetched from parent-entity object is to be updated by calling the setter method.

Updating the related-entity directly

In order to update the related-entity directly without using the parent-entity, fetch the related-entity directly using its corresponding Repository interface and set the value to be changed.

```
int quantity = 43;

OrderItem orderItem = orderItemRepository.findOne(new OrderItemPK(
    orderId, itemCode)); // (1)

orderItem.setQuantity(quantity); // (2)
```

Sr. No.	Description
(1)	Call findOne method of Repository interface of the related-entity and fetch the related-entity object.
(2)	Update the state of related-entity object using setter method.

Note: Behavior at the time of using parent-entity after the related-entity is updated

When the related-entity is updated using save method of Repository for related-entity, the related-entity stored in the parent-entity object is also updated unlike the case wherein the related-entity is added. This is because the parent-entity stores the reference of same instance which is managed under EntityManager.

Updating by using query method

Use query method to update the entity of persistence layer (DB) directly.

For details, refer to “*Operating the entities of Persistence Layer directly*”.

Deleting entities

Deleting parent-entity and related-entity

In order to delete parent-entity and related-entity together, call delete method of Repository interface.

For using this method, `remove` should be included in the cascade operation of the related-entity or the setting for deleting the related-entity should be enabled (orphanRemoval attribute should be set to `true`).

- Service

```
orderRepository.delete(orderId); // (1)
```

Sr. No.	Description
(1)	Specify ID or entity object and call delete method of Repository interface.

- Entity

```
@Entity
@Table(name = "t_order")
public class Order implements Serializable {

    // ...

    @OneToMany(mappedBy = "order",
               cascade = CascadeType.ALL, orphanRemoval = true) // (2)
    @OrderBy
    private Set<OrderItem> orderItems;

    // ...
}
```

}

Sr. No.	Description
(2)	Include <code>remove</code> in cascade operations and enable the setting to delete the related-entity (set <code>orphanRemoval</code> attribute of the associated annotation to <code>true</code>).

Note: Deleting related-entity

If you do not want to delete the related-entity object, perform settings such that `remove` is not included in cascade operation. Also, set `orphanRemoval` attribute to `false`.

Deleting the related-entity

In order to delete the related-entity, delete the related-entity object from the entity objects fetched through Repository interface.

For using this method, setting to delete the related-entity should be enabled (`orphanRemoval` attribute of the associated annotation should be set to `true`).

Note: Behavior when the setting to delete the related-entity is enabled

Behavior when the setting to delete the related-entity is enabled is as follows:

- In case of entity with 1:N relationship, if the related-entity object is deleted from the collection, it is also deleted from the persistence layer (DB) at the time of committing the transaction.
- In case of entity with 1:1 relationship, if the related-entity is set to `null`, it is also deleted from the persistence layer (DB) at the time of committing the transaction.

When the value of `orphanRemoval` attribute is set to `false` (which is also the default value), the related-entity is cleared from the memory; however persistence operation (UPDATE/DELETE) is not carried out for deleted related-entity object.

- Service

```
Order order = orderRepository.findOne(orderId); // (1)  
// (2)  
Set<OrderItem> orderItemsOfRemoveTarget = new LinkedHashSet<OrderItem>(); // (3)
```

```

for (OrderItem orderItem : order.getOrderItems()) {
    String itemCode = orderItem.getId().getItemCode();
    if (quantityMap.containsKey(itemCode)) {
        int newQuantity = quantityMap.get(itemCode);
        orderItem.setQuantity(newQuantity);
    } else {
        orderItemsOfRemoveTarget.add(orderItem); // (4)
    }
}
order.getOrderItems().removeAll(orderItemsOfRemoveTarget); // (5)

order.setOrderPay(null); // (6)

```

Sr. No.	Description
(1)	Fetch the entity object using Repository interface.
(2)	Describe the example to delete the entity with 1:N relationship.
(3)	Create a collection for storing the related-entity object to be deleted.
(4)	Add the related-entity object to be deleted to the collection of (3).
(5)	Delete it from the collection fetched from the related-entity object to be deleted.
(6)	In case of entity with 1:1 relationship, set the property to that stores the related-entity object to be deleted to null.

- Entity

```

@Entity
@Table(name = "t_order")
public class Order implements Serializable {

    @Id
    @SequenceGenerator(name = "GEN_ORDER_ID", sequenceName = "s_order_id", allocationSize =
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "GEN_ORDER_ID")
    private int id;

```

```
@OneToMany (mappedBy = "order", cascade = CascadeType.ALL,  
            orphanRemoval = true) // (7)  
@OrderBy  
private Set<OrderItem> orderItems;  
  
@OneToOne (mappedBy = "order", cascade = CascadeType.ALL,  
            orphanRemoval = true) // (7)  
private OrderPay orderPay;  
  
@ManyToOne  
@JoinColumn (name = "status_code")  
private OrderStatus status;  
  
// ...  
}
```

Sr. No.	Description
(7)	Enable the setting to delete the related-entity (set orphanRemoval attribute to true).

Note: Associated annotations that can be specified for orphanRemoval attribute

There are 2 associated annotations that can be specified for orphanRemoval attribute; @OneToOne and @OneToMany.

Deleting the related-entity directly

In order to delete the related-entity without using the parent-entity, call the delete method of Repository interface of related-entity.

```
int quantity = 43;  
  
orderItemRepository.delete(new OrderItemPK(orderId, itemCode)); // (1)
```

Sr. No.	Description
(1)	Specify ID or entity object and call delete method of Repository interface of the related-entity.

Warning: Points to be noted while deleting the related-entity directly

When delete method of Repository for the related-entity is called, it should be noted that related-entity may not get deleted from persistence layer (DB) in some cases.

Such cases are as follows:

- findOne method is called in delete method; hence when the relation with parent-entity is @OneToOne, the parent-entity ends up getting managed under EntityManager. If the parent-entity ends up getting managed under EntityManager, the related-entity that was to be deleted using delete method may be loaded in the parent-entity object. Once it is loaded under the parent-entity object, it will not be deleted from persistence layer (DB).
- If the parent-entity object is fetched after calling the delete method, the related-entity which is deleted using delete method may be loaded in the parent-entity object.

Once it is loaded under parent-entity object, it cannot be deleted from persistence layer (DB).

How to avoid this problem,

1. Instead of deleting the related-entity object directly, its association with the parent-entity should be deleted.

It may be possible to avoid this problem by reconsidering the associated annotation; however the best way to avoid this problem is not to delete the related-entity directly.

Deleting using query method

Use query method in order to directly delete the entity from persistence layer (DB).

For details, refer to “[Operating the entities of Persistence Layer directly](#)”.

Warning: Handling of related-objects

When the entity is deleted directly from the persistence layer (DB) using query method, irrespective of whether or not the associated annotation is specified, the related-entity object is not deleted from the persistence layer.

`void deleteInBatch(Iterable<T> entities)` and `void deleteAllInBatch()` of Repository interface operate in the same way.

Escaping at the time of LIKE search

Escaping the values to be used as search criteria should be done for LIKE search.

Escaping for LIKE search can be done using

`org.terasoluna.gfw.common.query.QueryEscapeUtils` class method of the common library.

For specifications of escaping in common library, refer to “*Escaping during LIKE search*” of “*Database Access (Common)*”.

For the escaping method provided by common library, see the description below.

Usage method when type of matching is to be specified in query

When type of matching (Forward match, Backward Match, Partial Match) is to be specified as JPQL, use the method that performs only escaping.

- Repository

```
// (1) (2)
@Query("SELECT a FROM Article a WHERE"
       + " (a.title LIKE %:word% ESCAPE '~' OR a.overview LIKE %:word% ESCAPE '~')")
Page<Article> findPageBy(@Param("word") String word, Pageable pageable);
```

Sr. No.	Description
(1)	Specify wildcard characters ("%" or "_") for LIKE search in JPQL to be specified in <code>@Query</code> annotation. In the above example, the type of matching is set to partial match by specifying wildcard ("%") before and after the argument <code>word</code> .
(2)	In case of escaping provided by the common library, "~" is being used as escape characters; hence specify "ESCAPE ' ~'" after LIKE clause.

- Service

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

```

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

        Page<Article> page = articleRepository.findPageBy(escapedWord, pageable); // (4)

        return page;
    }
}

```

Sr. No.	Description
(3)	When type of matching of LIKE search is to be specified in query, call <code>QueryEscapeUtils#toLikeCondition(String)</code> method and perform only escaping for LIKE search.
(4)	Pass the value escaped for LIKE search as the argument of query method.

Usage method when specifying type of matching in logic

When the type of matching (Forward match, Backward match, Partial match) is to be determined in logic, use the method that assigns wildcard to the escaped values.

- Repository

```

// (1)
@Query("SELECT a FROM Article a WHERE"
       + " (a.title LIKE :word ESCAPE '~' OR a.overview LIKE :word ESCAPE '~')")
Page<Article> findPageBy(@Param("word") String word, Pageable pageable);

```

Sr. No.	Description
(1)	Do not specify wildcard for LIKE search in JPQL to be specified in <code>@Query</code> annotation.

- Service

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

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

    Page<Article> page = articleRepository.findPageBy(word, pageable); // (3)

    return page;
}
```

Sr. No.	Description
(2)	When the type of matching is to be specified in logic, call any of the following methods and perform escaping for LIKE search and assign wildcard for LIKE search. QueryEscapeUtils#toStartingWithCondition(String) QueryEscapeUtils#toEndingWithCondition(String) QueryEscapeUtils#toContainingCondition(String)
(3)	Pass the value escaped for LIKE search + assigned with wildcard as an argument of query method.

JOIN FETCH

JOIN FETCH is a mechanism to reduce the number of queries generated for fetching entities, by joining and fetching the related-entities in batch.

When fetching the entities by executing any query, make sure to perform JOIN FETCH for properties where fetch attribute of associated annotation is EAGER.

If JOIN FETCH is not performed, it may impact performance since queries to fetch the related-entities will be generated separately.

For properties where fetch attribute of related-annotation is LAZY, consider the frequency of use and determine whether or not to perform JOIN FETCH.

JOIN FETCH should be performed in case of related-entities that are always referenced. However, if the cases wherein related-entities will be referenced are only few, then it is better to fetch the entities by executing the queries at the time of initial access instead of performing JOIN FETCH.

- Repository

```
@Query("SELECT a FROM Article a"
    + " INNER JOIN FETCH a.articleClass" // (1)
    + " WHERE a.publishedDate = :publishedDate"
    + " ORDER BY a.articleId DESC")
List<Article> findAllByPublishedDate(
    @Param("publishedDate") Date publishedDate);
```

Sr. No.	Description
(1)	<p>Specify in " [LEFT [OUTER] INNER] JOIN FETCH Properties to be joined".</p> <p>See the pattern below.</p> <p>1. LEFT JOIN FETCH Entity is obtained even if the related-entity does not exist. SQL will be "left outer join RelatedTable r on m.fkColumn = r.fkColumn".</p> <p>2. LEFT OUTER JOIN FETCH Same as 1.</p> <p>3. INNER JOIN FETCH Entity is not obtained if related-entity does not exist. SQL will be "inner join RelatedTable r on m.fkColumn = r.fkColumn".</p> <p>4. JOIN FETCH Same as 3.</p> <p>In the above example, articleClass property of Article class is specified as the JOIN FETCH target.</p>

Note: JOIN FETCH is used as one of the solutions for N+1 problem. For N+1 problem, refer to "[How to resolve N+1](#)".

5.2.3 How to extend

How to add custom method

Spring Data provides mechanism to add any custom methods for the Repository interface.

Sr. No.	How to Add	Use case
1.	<i>Adding individual custom method to entity specific Repository interface</i>	<p>Use when it is necessary to execute a query that cannot be expressed using the mechanism of query method of Spring Data.</p> <p>Add methods using this method when executing dynamic queries.</p>
2.	<i>Adding the custom methods to all Repository interfaces in batch</i>	<p>Use when it is necessary to execute a generic query that can be used in all Repository interfaces.</p> <p>Add methods using this method when executing project specific generic queries.</p> <p>If it is necessary to change the behavior of default implementation (<code>SimpleJpaRepository</code>) of Spring Data JPA, override the methods using this mechanism.</p>

Adding individual custom method to entity specific Repository interface

See the description below for adding individual custom method to entity specific Repository interface.

- Entity specific custom Repository interface

```
// (1)
public interface OrderRepositoryCustom {
    // (2)
    Page<Order> findByCriteria(OrderCriteria criteria, Pageable pageable);
}
```

Sr. No.	Description
(1)	Create a custom interface for defining custom methods. There are no specific restrictions for naming the interface; however it is recommended that you set it to entity specific Repository interface name + "Custom".
(2)	Define custom methods.

- Entity specific custom Repository class

```
// (3)
public class OrderRepositoryImpl implements OrderRepositoryCustom {

    @PersistenceContext
    EntityManager entityManager; // (4)

    // (5)
    public Page<Order> findByCriteria(OrderCriteria criteria, Pageable pageable) {
        // ...
        return new PageImpl<Order>(orders, pageable, totalCount);
    }

}
```

Sr. No.	Description
(3)	Create implementation class of custom interface. The class name should be, entity specific Repository interface name + "Impl".
(4)	Using @javax.persistence.PersistenceContext annotation, inject javax.persistence.EntityManager which is necessary for executing the query.
(5)	Implement the method defined in custom interface.

- Entity specific Repository interface

```
public interface OrderRepository extends JpaRepository<Order, Integer>,
    OrderRepositoryCustom { // (6)
    // ...
}
```

Sr. No.	Description
(6)	Inherit custom interface in entity specific Repository interface. Methods of implementation class of custom interface are called at runtime by inheriting Repository interface.

- Service(Caller)

```
public Page<Order> search(OrderCriteria criteria, Pageable pageable) {  
    return orderRepository.findByCriteria(criteria, pageable); // (7)  
}
```

Sr. No.	Description
(7)	Methods of entity specific Repository interface can be called similar to other methods. In the above example, if OrderRepository#findByCriteria(OrderCriteria, Pageable) is called, OrderRepositoryImpl#findByCriteria(OrderCriteria, Pageable) is executed.

Adding the custom methods to all Repository interfaces in batch

See the description below for adding the custom methods to all Repository interfaces in batch.

The method added in the example given below checks the version of the fetched entity and throws an optimistic exclusion error in case of a mismatch.

- Common Repository interface

```
// (1)  
@NoRepositoryBean // (2)  
public interface MyProjectRepository<T, ID extends Serializable> extends  
    JpaRepository<T, ID> {  
  
    // (3)  
    T findOneWithValidVersion(ID id, Integer version);  
}
```

Sr. No.	Description
(1)	Create a common Repository interface for defining the methods to be added. In the above example, common Repository interface is being created by inheriting JpaRepository of Spring Data.
(2)	Annotation to prevent common Repository interface implementation class (in this example, MyProjectRepositoryImpl) from being automatically registered as Repository Bean. When common Repository interface implementation class is to be stored under the package specified in base-package attribute of <jpa:repositories> element, there will be an error at start-up unless this annotation is specified.
(3)	Define the method to be added. In the example, method to check the version of fetched entity is being added.

- Common Repository interface implementation class

```
// (6)
public class MyProjectRepositoryImpl<T, ID extends Serializable>
    extends SimpleJpaRepository<T, ID>
    implements MyProjectRepository<T, ID> {

    private JpaEntityInformation<T, ID> entityInformation;
    private EntityManager entityManager;

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

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

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

    // (9)
    public T findOneWithValidVersion(ID id, Integer version) {
```

```
    if (versionMethod == null) {
        throw new UnsupportedOperationException(
            String.format(
                "Does not found version field in entity class. class is '%s'.",
                entityInformation.getJavaType().getName())));
    }

    T entity = findOne(id);

    if (entity != null && version != null) {
        Integer currentVersion;
        try {
            currentVersion = (Integer) versionMethod.invoke(entity);
        } catch (IllegalAccessException | IllegalArgumentException
            | InvocationTargetException e) {
            throw new IllegalStateException(e);
        }
        if (!version.equals(currentVersion)) {
            throw new ObjectOptimisticLockingFailureException(
                entityInformation.getJavaType().getName(), id);
        }
    }

    return entity;
}

}
```

Sr. No.	Description
(6)	Create an implementation class of common Repository interface (in this example, <code>MyProjectRepository</code>). In the above example, implementation class is being created by inheriting “ <code>SimpleJpaRepository</code> “ of Spring Data.
(7)	Create a constructor that takes <code>org.springframework.data.jpa.repository.support.JpaEntityInformation</code> and <code>javax.persistence.EntityManager</code> as arguments.
(8)	Store <code>JpaEntityInformation</code> and <code>EntityManager</code> as fields.
(9)	Implement the method defined in common Repository interface.

Note: Changing default implementation

If the behavior of default implementation (`SimpleJpaRepository`) is to be changed, the method for which behavior is to be changed can be overridden in this class.

- Factory class for creating instances of common Repository interface implementation class

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

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

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

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

        MyProjectRepositoryImpl<T, ID> repositoryImpl = new MyProjectRepositoryImpl<T, ID>()
    }
}
```

```
        entityInformation, entityManager);
repositoryImpl
    .setLockMetadataProvider(LockModeRepositoryPostProcessor.INSTANCE
        .getLockMetadataProvider()); // (13)

    return repositoryImpl;
}

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

Sr. No.	Description
(10)	Create a Factory class for creating instances of common Repository interface implementation class. In the example, <code>org.springframework.data.jpa.repository.support.JpaRepositoryFactory</code> is being inherited in order to restrict the implementation of Factory class.
(11)	Define a constructor that takes EntityManager as argument and call constructor of parent class.
(12)	Override the method creating instances of implementation class. Create and return instances of the created implementation class (in this example, <code>MyProjectRepositoryImpl</code>).
(13)	This code is required to specify the lock mode using <code>@Lock</code> annotation for <code>findOne</code> method and <code>findAll</code> methods of <code>SimpleJpaRepository</code> class of Spring Data JPA. Make sure to implement this piece of logic, since it is implemented in the method of <code>JpaRepositoryFactory</code> which is being overridden. This process is required when the lock mode is to be specified using <code>findOne</code> and <code>findAll</code> methods amongst the methods to be extended.
(14)	Override the method that returns base interface of entity specific Repository interface, and return the created interface (in this example, <code>MyProjectRepository</code>) type.

- FactoryBean for creating Factory class instances

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

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

Sr. No.	Description
(15)	Create FactoryBean class for creating Factory class instances. In the example, <code>org.springframework.data.jpa.repository.support.JpaRepositoryFactory</code> is being inherited in order to restrict the implementation of FactoryBean class.
(16)	Override the method used for creating the Factory class instances. Then, create and return instances of the created Factory class.

- Entity specific Repository interface

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

Sr. No.	Description
(17)	Inherit all entity specific Repository interfaces from the common interface (In this example: <code>MyProjectRepository</code>).

- `xxx-infra.xml`

```
<jpa:repositories base-package="x.y.z.domain.repository"
    factory-class="x.y.z.domain.repository.MyProjectRepositoryFactoryBean" /> <!-- (18) -->
```

Sr. No.	Description
(18)	Specify class name of the created FactoryBean class (in the example, MyProjectRepositoryFactoryBean) in factory-class attribute of <jpa:repositories> element.

- Service(Caller)

```
public Order updateOrder(Order chngedOrder, Integer version) {  
  
    Order order = orderRepository.findOneWithValidVersion(chngedOrder.getId(), version); //  
  
    // ....  
  
    return order;  
}
```

Sr. No.	Description
(19)	The methods of entity specific Repository interface can be called similar to other methods. In the above example, when OrderRepository#findOneWithValidVersion(Integer, Integer) is called, MyProjectRepositoryImpl#findOneWithValidVersion(Integer, Integer) is executed.

Storing query fetch results in objects other than entity

Query fetch results can be mapped to other objects than entities. Use this method when records stored in persistence layer (DB) are to be handled as objects (JavaBean) other than entity.

Note: Assumed cases

1. When the aggregated information is to be fetched using Aggregate function in query, it is not possible to map the aggregation result to entity; hence it should be mapped with different objects.
2. In order to refer to only a part of information in a huge entity, or of a related-entity with complex nesting, there may be cases wherein it is desirable to map the results with JavaBean containing only the necessary properties. This is because processing performance may get hampered due to the fact that mapping is carried out for items which are not needed in application processing or fetching of

unnecessary information during the processing leading to memory exhaustion. It may be obtained as entity if there is no significant impact on processing performance.

See the example below.

- JavaBean

```
// (1)
public class OrderSummary implements Serializable {

    private Integer id;
    private Long totalPrice;

    // ...

    public OrderSummary(Integer id, Long totalPrice) { // (2)
        super();
        this.id = id;
        this.totalPrice = totalPrice;
    }

    // ...
}
```

Sr. No.	Description
(1)	Create JavaBean for storing query results.
(2)	Create a constructor for generating objects using query execution results.

- Repository interface

```
// (3)
@Query("SELECT NEW x.y.z.domain.model.OrderSummary(o.id, SUM(i.price*oi.quantity))"
       + " FROM Order o LEFT JOIN o.orderItems oi LEFT JOIN oi.item i"
       + " GROUP BY o.id ORDER BY o.id DESC")
List<OrderSummary> findOrderSummaries();
```

Sr. No.	Description
(3)	Specify the constructor created in (2). Specify the constructor by FQCN after “NEW” keyword.

Setting Audit properties

This section introduces a mechanism to set values to Audit properties (Created By, Created Date-Time, Last Modified By, Last Modified Date-Time) of persistence layer and method to apply the same.

Spring Data JPA provides a mechanism to set values to Audit properties for newly created entities as well as modified entities. This mechanism facilitates separation of logic of setting values to Audit properties from application code such as Service etc.

Note: Reasons to separate the logic to set values to Audit properties from application code

1. Setting the values to Audit properties is generally not an application requirement, but is essential as per the audit requirements of data. This logic does not essentially belong to application code such as Service etc.; hence it is better to separate it from application code.
 2. As per JPA specifications, only when the values of the entities fetched from persistence layer change, the changes will be reflected in the persistence layer (by executing the SQL), thereby avoiding the unnecessary access to persistence layer. If the values are set unconditionally in Audit properties in the application code such as Service, even if only Audit properties change, the changes are reflected in persistence layer making the effective JPA functionality ineffective. This problem can be avoided if the values are set in Audit properties only when if they are changed; however it cannot be recommended as it makes the application code complex.
-

Note: When Audit column of the persistence layer needs to be updated irrespective of the change in values

If updating the Audit columns (Last Modified By, Last Modified Date-Time) even if there is no change in values is a part of application specifications, then it becomes necessary to set Audit properties in application code such as Service etc.

However, in such a case, the data modeling or application specifications are likely to be incorrect; hence it is better to revise these specifications.

See the example below.

- Entity class

```
public class XxxEntity implements Serializable {
    private static final long serialVersionUID = 1L;

    // ...

    @Column(name = "created_by")
    @CreatedBy // (1)
    private String createdBy;

    @Column(name = "created_date")
    @CreatedDate // (2)
    @Type(type = "org.jadira.usertype.dateandtime.joda.PersistentDateTime") // (3)
    private DateTime createdDate; // (4)

    @Column(name = "last_modified_by")
    @LastModifiedBy // (5)
    private String lastModifiedBy;

    @Column(name = "last_modified_date")
    @LastModifiedDate // (6)
    @Type(type = "org.jadira.usertype.dateandtime.joda.PersistentDateTime") // (3)
    private DateTime lastModifiedDate; // (4)

    // ...

}
```

Sr. No.	Description
(1)	Assign <code>@org.springframework.data.annotation.CreatedBy</code> annotation to the field that stores “Created by”.
(2)	Assign <code>@org.springframework.data.annotation.CreatedDate</code> annotation to the field that stores “Created date-time”.
(3)	When <code>org.joda.time.DateTime</code> type is to be used, assign <code>@org.hibernate.annotations.Type</code> annotation to the field in order to handle in Hibernate. type attribute is fixed to <code>"org.jadira.usertype.dateandtime.joda.PersistentDateTime"</code> . This is applicable for “Last modified date-time” field also.
(4)	Field type that stores the “Created date-time” supports <code>org.joda.time.DateTime</code> , <code>java.util.Date</code> , <code>java.lang.Long</code> and <code>long</code> types. This is applicable for “Last modified date” field also.
(5)	Assign <code>@org.springframework.data.annotation.LastModifiedBy</code> annotation to the field type that stores “Last modified by”.
(6)	Assign <code>@org.springframework.data.annotation.LastModifiedDate</code> annotation to the field that stores the “Last modified date-time”.

Warning: `@Type` annotation is a Hibernate specific annotation and not a standard JPA annotation.

- AuditorAware interface implementation class

```
// (7)
@Component // (8)
public class SpringSecurityAuditorAware implements AuditorAware<String> {

    // (9)
    public String getCurrentAuditor() {
```

```

        Authentication authentication = SecurityContextHolder.getContext()
            .getAuthentication();
        if (authentication == null || !authentication.isAuthenticated()) {
            return null;
        }
        return ((UserDetails) authentication.getPrincipal()).getUsername();
    }
}

```

Sr. No.	Description
(7)	<p>Create <code>org.springframework.data.domain.AuditorAware</code> interface implementation class.</p> <p><code>AuditorAware</code> interface is used for resolving the entity operator (Created by or Last Modified by).</p> <p>This class should be created for each project.</p>
(8)	<p>By assigning <code>@Component</code> annotation, this class comes under the target of component-scan.</p> <p>Without assigning <code>@Component</code> annotation, it is also ok to define a bean of this class in bean definition file.</p>
(9)	<p>Return the object to be set in the properties of entity operator (Created by or Last Modified by).</p> <p>In the above example, login user name (string) authenticated by SpringSecurity is returned as entity operator.</p>

- Object/relational mapping file(`orm.xml`)

```

<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
    http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
    version="2.0">

    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <entity-listeners>
                <entity-listener
                    class="org.springframework.data.jpa.domain.support.AuditingEntityListene
                </entity-listeners>

```

```
</persistence-unit-defaults>
</persistence-unit-metadata>

</entity-mappings>
```

Sr. No.	Description
(10)	Specify <code>org.springframework.data.jpa.domain.support.AuditingEntityListener</code> class as Entity Listener. Values are set in Audit properties of the methods implemented in this class.

- `infra.xml`

```
<jpa:auditing auditor-aware-ref="springSecurityAuditorAware" /> <!-- (11) -->
```

Sr. No.	Description
(11)	Specify bean of class for resolving entity operator created in (7) in auditor-aware-ref attribute of <code><jpa:auditing></code> element. In the above example, component-scan is performed for <code>SpringSecurityAuditorAware</code> implementation class; hence bean name " <code>springSecurityAuditorAware</code> " is specified.

As default implementation, the values of `org.joda.time.DateTime` instance returned by `getDateTime()` method of `org.springframework.data.auditing.CurrentDateTimeProvider` are used for the values set in the fields with `@CreatedDate` and `@LastModifiedDate` annotations.

Extended example of changing the method to create the values to be used is shown below.

```
// (1)
@Component // (2)
public class AuditDateTimeProvider implements DateTimeProvider {

    @Inject
    DateFactory dateFactory;

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

}

Sr. No.	Description
(1)	Create <code>org.springframework.data.auditing.DateTimeProvider</code> interface implementation class.
(2)	Assign <code>@Component</code> annotation to perform component-scan. Bean can be defined in Bean definition file without assigning <code>@Component</code> annotation.
(3)	Return the instances to be set in the properties of entity operation date-time (Created Date-Time or Last Modified Date-Time). In the above example, the instances fetched from <code>org.terasoluna.gfw.common.date.DateFactory</code> of common library are returned as Operation Date-Time. For details on <code>DateFactory</code> , refer to “ System Date ”.

- `infra.xml`

```
<jpa:auditing
    auditor-aware-ref="springSecurityAuditorAware"
    date-time-provider-ref="auditDateTimeProvider" /> <!-- (4) -->
```

Sr. No.	Description
(4)	In date-time-provider-ref attribute of <code><jpa:auditing></code> element, specify the bean of class to return the value set in entity operation date-time created in (1). In the above example, component-scan is performed for <code>AuditDateTimeProvider</code> implementation class; hence bean name "auditDateTimeProvider" is specified.

Adding common conditions to JPQL to fetch entities from persistence layer

This section introduces a mechanism to add common conditions to JPQL for fetching the entities from persistence layer (DB). This is an extended Hibernate function and not included in standard JPA specifications.

Note: Assumed cases

As per data monitoring and data storage period requirements, when an entity is to be deleted, the application should be designed such that it will delete the record logically (UPDATE of Logical Delete flag) and will not delete (DELETE) the record physically. In case of such applications, the records deleted logically need to be uniformly excluded from the search target; hence rather than writing the condition to exclude logically deleted records in each and every query, it is better to specify it as a common condition.

Warning: Applicable scope

It should be noted that the specified conditions will be added for all queries executed to operate the entities for which this mechanism is used. This mechanism cannot be used even if there is a single Query for which conditions are not to be added.

Adding common conditions in JPQL to fetch entities

The method to add common conditions for JPQL which is executed at the time of calling Repository interface methods is as follows:

- Entity

```
@Entity
@Table(name = "t_order")
@Where(clause = "is_logical_delete = false") // (1)
public class Order implements Serializable {
    ...
    @Id
    private Integer id;
    ...
}
```

- SQL executed at the time of calling the findOne method of Repository interface

```
SELECT
    ...
FROM
    t_order order0_
WHERE
    order0_.id = 1
    AND (
        order0_.is_logical_delete = false -- (2)
    );
```

Sr. No.	Description
(1)	<p>Assign <code>@org.hibernate.annotations.Where</code> annotation as entity class annotation and specify the common conditions in clause attribute.</p> <p>The WHERE clause should be specified in SQL instead of JPQL i.e. it is necessary to specify the column name instead of the property name of Java object.</p>
(2)	The condition specified with <code>@Where</code> annotation is added.

Note: Class that can be specified

`@Where` annotation is valid only in the class with `@Entity`. i.e. it is not assigned to SQL even if it is assigned to the base entity class with `@javax.persistence.MappedSuperclass`.

Warning: `@Where` annotation is a Hibernate specific annotation and not the standard JPA annotation.

Adding common conditions to JPQL to fetch the related-entities

The method for adding common conditions for JPQL is shown below. JPQL is used for fetching the related-entities of the parent-entity which is fetched by calling Repository interface methods.

- Entity

```

@Entity
@Table(name = "t_order")
@Where(clause = "is_logical_delete = false")
public class Order implements Serializable {
    // ...
    @Id
    private Integer id;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)
    @OrderBy
    @Where(clause="is_logical_delete = false") // (1)
    private Set<OrderItem> orderItems;
    // ...

}

```

- SQL (Lazy Load) generated when accessing the related-entity

```
SELECT
    -- ...
FROM
    t_order_item orderitems0_
WHERE
    (orderitems0_.is_logical_delete = false) -- (2)
    AND orderitems0_.order_id = 1
ORDER BY
    orderitems0_.item_code ASC
    ,orderitems0_.order_id ASC;
```

- SQL(Eager Load/Join Fetch) generated at the time of fetching parent-entity and its related-entity simultaneously

```
SELECT
    -- ...
FROM
    t_order order0_
    LEFT OUTER JOIN t_order_item orderitems1_
        ON order0_.id = orderitems1_.order_id
        AND (orderitems1_.is_logical_delete = false) -- (2)
WHERE
    order0_.id = 1
    AND (
        order0_.is_logical_delete = false
    )
ORDER BY
    orderitems1_.item_code ASC
    ,orderitems1_.order_id ASC;
```

Sr. No.	Description
(1)	Assign <code>@org.hibernate.annotations.Where</code> annotation as a related-entity field allocation and specify common conditions in clause attribute. The WHERE clause should be specified in SQL format instead of JPQL i.e. it is necessary to specify the column name instead of the property name of Java object.
(2)	The condition specified with <code>@Where</code> annotation is added.

Note: Types of related-entities that can be specified

Adding common conditions to SQL generated at the time fetching the related-entities is possible only for the entities having `@OneToMany` and `@ManyToMany` relationship.

Warning: `@Where` annotation is a Hibernate specific annotation and not the standard JPA annotation.

How to use multiple PersistenceUnits

Todo

TBD

Following topic will be added later.

- Examples for using multiple PersistenceUnits
-

How to use Native query

Todo

TBD

Following topic will be added later.

- Examples of query that uses Native query
-

5.3 Database access (Mybatis2)

5.3.1 Overview

This chapter describes how to access database using MyBatis2.x.

This guideline presumes the use of TERASOLUNA DAO (Mybatis DAO wrapper).

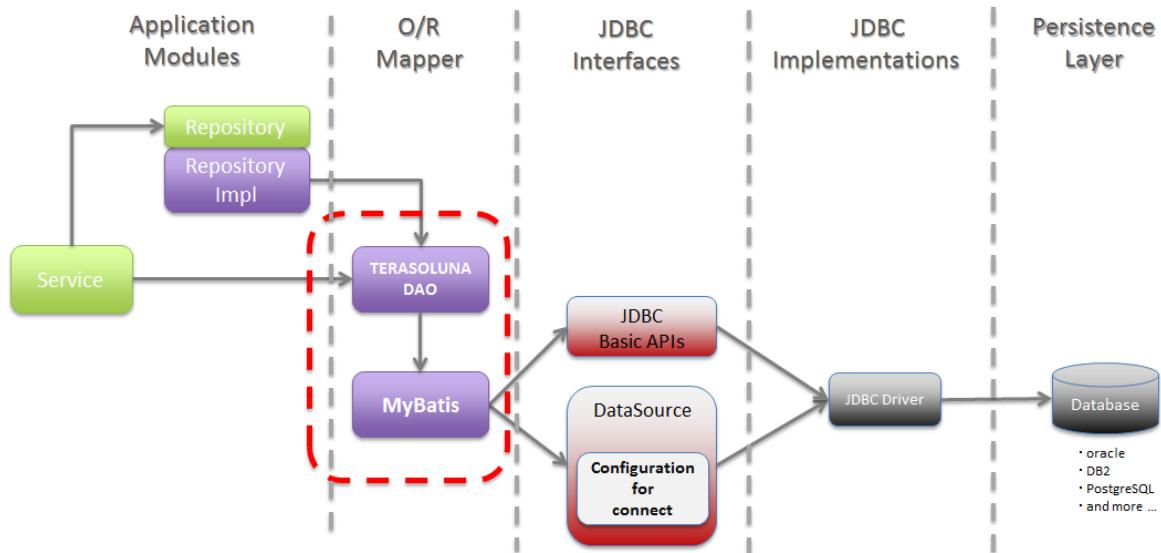


Figure.5.13 Picture - Target of description

About Mybatis

Mybatis is a type of O/R Mapper. The idea behind developing MyBatis is to use it for mapping the SQL statements to objects,

rather than for mapping the database records to the objects.

Thus, this O/R Mapper can be effectively used to access de-normalized databases or to obtain full control of SQL execution.

For details of Mybatis2.x, refer to [Mybatis Developer Guide \(PDF\)](#).

About TERASOLUNA DAO

TERASOLUNA DAO provides a DAO interface for hiding processes dependent on O/R Mapper and a DAO implementation class which uses Mybatis2.x.

DAO interfaces provided by TERASOLUNA DAO are as follows:

Table.5.8 DAO interfaces provided by TERASOLUNA DAO

Sr. No.	Class name	Description
1.	jp.terasoluna.fw.dao. QueryDAO	DAO interface for executing QuerySQL
2.	jp.terasoluna.fw.dao. UpdateDAO	DAO interface for executing UpdateSQL
3.	jp.terasoluna.fw.dao. StoredProcedureDAO	DAO interface for executing StoredProcedure
4.	jp.terasoluna.fw.dao. QueryRowHandleDAO	DAO interface to process each record fetched by executing QuerySQL.

Basic flow to access database using TERASOLUNA DAO (Mybatis implementation) is shown below.

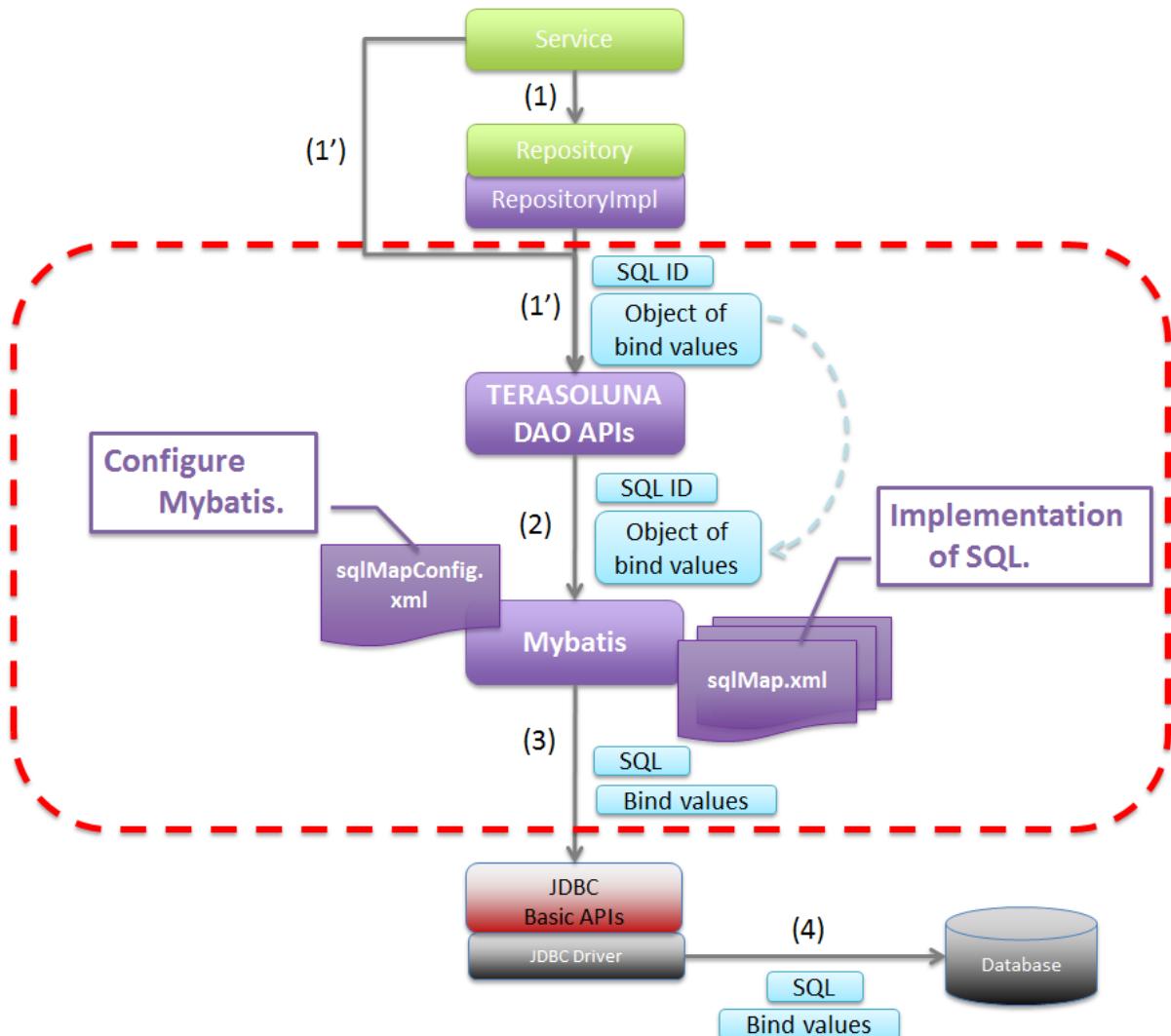


Figure.5.14 Picture - Basic flow of TERASOLUNA DAO

Sr. No.	Description
(1)	Call the DAO interface method of TERASOLUNA DAO, from Service or Repository. Pass the object which holds the values to be embedded in SQLID and SQL, as the calling parameter of the method.
(2)	TERASOLUNA DAO delegates the process to Mybatis API. The object which holds the values to be embedded in SQL and the SQLID specified from Service or Repository, are also passed to Mybatis.
(3)	Mybatis fetches the SQL corresponding to the specified SQLID from configuration file (sqlMap.xml), and passes the SQL and bind value to JDBC driver. 5 Architecture in Detail - TERASOLUNA Global Framework (java.sql.PreparedStatement API is used for actual bind value).
430	JDBC driver executes the SQL by sending the bind value and passed SQL to database.

5.3.2 How to use

pom.xml settings

To use MyBatis2 (TERASOLUNA DAO) in infrastructure layers, add the following dependency to pom.xml.

```
<!-- (1) -->
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-mybatis2</artifactId>
</dependency>
```

Sr. No.	Description
(1)	Add <code>terasoluna-gfw-mybatis2</code> that defines the library group relevant to MyBatis2, to dependency.

Application settings

Datasource settings

For datasource settings, refer to the common version *Datasource settings*.

PlatformTransactionManager settings

To use local transactions, perform the following settings.

When a local transaction is used, call the API of JDBC and use `org.springframework.jdbc.datasource.DataSourceTransactionManager` that performs transaction control.

- `xxx-env.xml`

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

Sr. No.	Description
(1)	Specify org.springframework.jdbc.datasource.DataSourceTransactionManager.
(2)	Specify the bean of configured datasource.

To use the transaction manager provided by application server, perform the following settings.

When transaction manager provided by application server is used, call the API of JTA and use `org.springframework.transaction.jta.JtaTransactionManager` that performs transaction control.

- xxx-env.xml

```
<tx:jta-transaction-manager /> <!-- (1) -->
```

Sr. No.	Description
(1)	Using "transactionManager" id, bean is defined for <code>JtaTransactionManager</code> , which is the most appropriate transaction manager for the application server on which the application is deployed,

TERASOLUNA DAO settings

Define the beans for factory class of `SqlMapClient` provided by Spring Framework and TERASOLUNA DAO.

- xxx-infra.xml

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

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

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

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

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

Sr. No.	Description
(1)	Specify <code>org.springframework.orm.ibatis.SqlMapClientFactoryBean</code> as the factory class of <code>SqlMapClient</code> class.
(2)	Specify the location of Mybatis configuration file. In this example, it is the file that ends with " <code>sqlMapConfig.xml</code> " stored in " <code>/META-INF/mybatis/config/</code> " directory in class path. For configuration file, refer to Mybatis settings .
(3)	Specify the location of Mybatis SQL mapping file. In this example, it is the file that ends with " <code>-sqlmap.xml</code> " stored under " <code>/META-INF/mybatis/sql/</code> " directory (including sub-directory) in class path. For SQL mapping file, refer to Implementing SQL mapping (Basic version) .
(4)	Specify the bean for the configured datasource.
(5)	Define the bean by specifying Mybatis implementation class of TERASOLUNA DAO.
(6)	Specify the bean for the factory class of <code>SqlMapClient</code> class defined in (1).

Settings for LOB handling

When Large Object like BLOB or CLOB is to be handled, specify `LobHandler` in the factory class of `SqlMapClient` class.

- `xxx-infra.xml`

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

<!-- (2) -->
<bean id="lobHandler" class="org.springframework.jdbc.support.lob.OracleLobHandler">
    <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor" /> <!-- (3) -->
</bean>
```

```

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

```

Sr. No.	Description
(1)	<p>Define bean for the implementation class of <code>org.springframework.jdbc.support.nativejdbc.NativeJdbcExtractor</code> interface.</p> <p>In this example, <code>org.springframework.jdbc.support.nativejdbc.SimpleNativeJdbcExtractor</code> is specified.</p> <p>However, at times, native datasource cannot be fetched in AP servers other than Tomcat. Therefore, it is either necessary to specify another <code>NativeJdbcExtractor</code> provided by Spring or create a new <code>NativeJdbcExtractor</code> for each AP server.</p>
(2)	<p>Define bean for the implementation class of <code>org.springframework.jdbc.support.lob.LobHandler</code> interface.</p> <p>In this example, the <code>org.springframework.jdbc.support.lob.OracleLobHandler</code> interface, which is specified when using Oracle is specified.</p> <p>However, specify <code>org.springframework.jdbc.support.lob.DefaultLobHandler</code> when Oracle is not used.</p>
(3)	Specify the bean for <code>NativeJdbcExtractor</code> defined in (1).
(4)	Specify the bean for <code>LobHandler</code> defined in (3).

Mybatis settings

Customize the default operations of SqlMapClient. They should be customized as per the requirement.

- sqlMapConfig.xml

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

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

Sr. No.	Description
(1)	Specify the DTD file. This will enable schema check and IDE code completion.
(2)	By setting useStatementNamespaces="true", namespace specified in SQL mapping file is used as SQLID.

- About sqlMapConfig child elements

properties, settings, resultObjectFactory, typeAlias, transactionManager and sqlMap are the child elements.

They should be set if required.

For details, refer to “The SQL Map XML Configuration File” (P.8-16) of Mybatis Developer Guide (PDF).

Table.5.9 child elements of sqlMapConfig

Sr. No.	Elements	Description
1.	properties	<p>Elements for reading a property file. Properties defined in the read property file can be referred in a "\${Property name}" format from the Mybatis configuration file and SQL mapping file.</p> <p>It is used while defining environment dependent values or common configuration values.</p> <p>For details, refer to “The SQL Map XML Configuration File” (P.9) of Mybatis Developer Guide (PDF).</p>
2.	settings	<p>Element for customizing default operations of SqlMapClient.</p> <p>For details of setting field, refer to “The SQL Map XML Configuration File” (P.9-11) of Mybatis Developer Guide (PDF).</p>
3.	resultObjectFactory	<p>Element for specifying the factory class which generates the instance for class specified in the class attribute of resultMap element or the resultClass attribute of select element, statement element and procedure element of SQL mapping file.</p> <p>When not specified, the instance generated by the default implementation method <code>java.lang.Class#newInstance()</code> is used.</p> <p>For details, refer to “The SQL Map XML Configuration File” (P.11-12) of Mybatis Developer Guide (PDF).</p>
4.	typeAlias	<p>Element for assigning an alias (short name) to class name (FQCN).</p> <p>The alias name defined here can be used at locations where Mybatis configuration file and SQL mapping file class are specified. Normally, a simple class name without the package name is specified.</p> <p>For details, refer to “The SQL Map XML Configuration File” (P.12) of Mybatis Developer Guide (PDF).</p>
5.	transactionManager	<p>It is not necessary to define transaction manager since Spring Framework functions are used for transaction management.</p>
6.	sqlMap	<p>It is not necessary to define sqlMap since it is already set in TERASOLUNA DAO settings.</p>

Implementing SQL mapping (Basic version)

Basic SQL mapping implementation is shown below.

Implement the SQL to be used in application.

- xxx-sqlmap.xml

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

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

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

    <!-- ... -->

</sqlMap>
```

Sr. No.	Description
(1)	Specify the DTD file. This will enable schema check and STS code completion.
(2)	Specify namespace.
(3)	It is set such that namespace is used as SQLID in sqlMapConfig.xml. Hence, the SQLID specified for executing this SQL is “xxx.findOne”.

- About sqlMap child elements

cacheModel, typeAlias, parameterMap, resultMap, select, insert, update, delete, statement, sql and procedure are the child elements.

Table.5.10 Child elements of sqlMap

Sr. No.	Elements	Description
1.	typeAlias	Same as typeAlias of sqlMapConfig.xml.
2.	cacheModel	Element defining an object cache
3.	parameterMap	Element defining the mapping of SQL bind parameters (object)
4.	resultMap	Element defining the mapping of records and objects returned as SQL execution result
5.	select	Element describing SELECT statement
6.	insert	Element describing INSERT statement
7.	update	Element describing UPDATE statement
8.	delete	Element describing DELETE statement
9.	statement	Generic elements including elements like select, insert, update, delete and procedure. It is recommended to use individual elements (select, insert, update, delete, procedure).
10.	sql	Element describing the SQL statement (part of SQL statement) to be included from select, insert, update, delete, statement. Effective use of these elements enables standardization of duplicate parts used in multiple SQLs.
11.	procedure	Element describing PROCEDURE call

Note: For details, refer to the following chapters of Mybatis Developer Guide (PDF).

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

Simple definition example of SQL mapping file is described.

- Mapped Statements (P.18-26)

Basic use of elements for building SQL is described.

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

Mapping of SQL bind parameters (objects) is described in detail.

- Substitution Strings (P.32)

SQL bind variables are described.

- Result Maps (P.32-41)

Mapping of records and objects returned as SQL execution result is described in detail.

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

The types and extension methods supported by ParameterMap and ResultMap are described.

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

Cache is described in detail.

- Dynamic Mapped Statements (P.48-53)

Dynamic SQL is described in detail.

- Simple Dynamic SQL Elements (P.53)

Simple implementation of dynamic SQL is described.

Warning: Appropriate value should be set in fetchSize attribute to describe a query such as, returning a large volume of data using statement, select and procedure elements. fetchSize attribute is the parameter that sets the number of data records which are retrieved in a single communication between JDBC driver and database. Default value of each JDBC driver is used when fetchSize attribute is omitted. However, care needs to be taken in case of JDBC driver wherein default value is used to fetch all the records, as it may result in memory exhaustion.

select element implementation

Define mapping for the searched record column and JavaBean property before implementing select elements.

- xxx-sqlmap.xml

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

Sr. No.	Attribute	Description
(1)	-	Map the searched records and JavaBean. For details, refer to Developer Guide.
	id	Specify the ID to identify mapping. It is referred from select attribute.
	class	Specify the FQCN of JavaBean to be mapped.
(2)	-	Map JavaBean property with searched record columns.
	property	Specify JavaBean property name.
	column	Specify the column name of the record to be mapped in property specified in “properties” attribute.

Implement select element.

- xxx-sqlmap.xml

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

Sr. No.	Attribute	Description
(3)	-	<p>Execute the Search SQL.</p> <p>Specify the ID for identifying Search SQL.</p> <p>id</p> <p>Specify the type of bind object.</p> <p>parameterClass</p> <p>In the example, <code>java.lang.String</code> is specified. However, JavaBean can also be specified when multiple parameters (search conditions) are to be passed.</p>
(4)	-	<p>Specify the resultMap defined in (1).</p> <p>It is possible to map automatically to the JavaBean properties specified in class attribute, without using resultMap. However, column name of fetched record should match with the JavaBean property name.</p> <p>To match the column name of fetched record with JavaBean property name, an alias is assigned to the column using AS clause. For example, when SQL is set as <code>"SELECT todo_title AS todoTitle, ..."</code>, a value is set to the <code>todoTitle</code> property of JavaBean.</p> <p>Specify bind value in SQL.</p> <p>In the example, since a single object (<code>java.lang.String</code>) is used instead of JavaBean, any name can be specified for bind variable.</p> <p>When JavaBean is used for bind object, bind variable name must match with the JavaBean property name.</p>

Note: About automatic mapping

It is possible to map automatically to the JavaBean property specified in resultClass attribute, without using resultMap. However, the column name of fetched record must match with the JavaBean property name. To match the column name of fetched record with JavaBean property name, an alias is assigned to the column using AS clause. Implementation with auto mapping is shown below.

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

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

Auto mapping is the simplest method for mapping fetched records and JavaBean. However, following restrictions and precautions should be considered while using auto mapping,

- Type declaration and conversion definition of the value fetched in SQL cannot be performed.
 - Complex mapping (for example, mapping to nested JavaBean) cannot be performed.
 - There is a slight degradation in performance as `java.sql.ResultSetMetaData` is accessed during mapping.
-

“insert” element implementation

Implement insert element.

- `xxx-sqlmap.xml`

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

Sr. No.	Attribute	Description
(1)	-	Implement Insert SQL. Specify the ID to identify Insert SQL.
(2)	id parameterClass -	Specify the type of bind object. JavaBean can also be specified. Specify the bind value in SQL. When JavaBean is used for bind object, bind variable name must match with the JavaBean property name.

Note: Type declaration or conversion definition of SQL bind value can be performed by using the parameterMap attribute or “Inline Parameter Maps” feature. For example, when bind value is null, default value can be set. For details, refer to “Parameter Maps and Inline Parameters” (P.27-31) of Mybatis Developer Guide (PDF).

“update” element implementation

Implement update element.

- xxx-sqlmap.xml

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

Sr. No.	Description
(1)	Implement Update SQL.

“delete” element implementation

Implement delete element.

- xxx-sqlmap.xml

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

Sr. No.	Description
(1)	Implement Delete SQL.

“procedure” element implementation

Calling the function created in PostgreSQL using procedure element is shown below.

SQL to create functions (PL/pgSQL implementation) and tables is as follows:

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

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

Implement parameterMap element.

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

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

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

Sr. No.	Description
(1)	Define mapping of IN parameter and OUT parameter to be passed to the function.
(2)	Mapping of IN parameter is defined. SalesItem#id is mapped with IN parameter.
(3)	Mapping of OUT parameters is defined. 1st OUT parameter is mapped with SalesItem#quantity and the 2nd OUT parameter is mapped with SalesItem#total.
(4)	JavaBean to be mapped.

Note: Mapping can also be carried out with “Inline Parameter Maps” feature, without using parameterMap attribute. For specific examples, refer to “Parameter Maps and Inline Parameters” (P.31) of Mybatis Developer Guide(PDF).

Implement procedure element.

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

Sr. No.	Description
(1)	<p>Specify the Procedure or Function to be called in “{call Procedure/Function name (IN parameter ...,OUT parameter...)}” format.</p> <p>In the example, one IN parameter and two OUT parameters are specified for <code>sales_item</code> function.</p> <p>Bound values serve as a definition sequence of mapping definitions specified in <code>parameterMap</code> element.</p>

“sql” element implementation

Implement sql element.

- xxx-sqlmap.xml

```

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

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

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

Sr. No.	Description
(1)	WHERE clause shared by the SQLs of (2) and (4) is defined. SQL to be included should be defined before the SQL that implements ‘include’.
(2)	SQL for fetching the data that matches the condition.
(3)	Include the SQL implemented by WHERE clause defined in (1).
(4)	SQL for fetching data records that match the conditions
(5)	Include the SQL implemented by WHERE clause defined in (1).

LOB update implementation

Implementation for updating Large Objects like BLOB and CLOB in database is shown below.

In the following example, records are inserted in the table that handles BLOB.

- DDL

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

Sr. No.	Description
(1)	Define BLOB column. In the above example, the DDL is assumed to use Oracle as a database.

- DTO (JavaBean)

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

```

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

// omitted setter/getter
}

```

Sr. No.	Description
(2)	Define the property that stores BLOB value in <code>java.io.InputStream</code> type. In the above example, input stream of uploaded file is set in <code>InputStream</code> .

Warning: It is recommended to always define the type of property that handles BLOB, in `InputStream` type. BLOB can also be handled as a byte array. However, it may cause memory exhaustion in case of increased data capacity.

It is recommended to always define the type of property that handles CLOB, in `java.io.Reader` type. CLOB can also be handled as a string. However, it may cause memory exhaustion in case of increased data capacity.

- `xxx-sqlmap.xml`

```

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

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

```

Sr. No.	Description
(3)	Specify the definition that is necessary to register the parameter that stores the registration values of BLOB column. Specify "BLOB" in jdbcType attribute and "jp.terasoluna.fw.orm.ibatis.support.BlobInputStreamTypeHandler" in typeHandler attribute.
(4)	SQL to register records in the table with a BLOB column.

Note: While handling CLOB, specify "CLOB" in jdbcType attribute and "jp.terasoluna.fw.orm.ibatis.support.ClobReaderTypeHandler" in typeHandler attribute.

Tip: Class name specified in FQCN can be described in a simple manner by assigning an alias to it using typeAlias element.

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

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

<!-- omitted -->

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

</parameterMap>
```

Sr. No.	Description
(5)	Assign an alias to the class name (FQCN) using typeAlias element. In the above example, an alias is assigned to BinaryFile class and BlobInputStreamTypeHandler class. typeAlias element can be defined in both sqlMapConfig.xml and xxxx-sqlmap.xml.
(6)	Specify the alias name for the class name (FQCN) assigned in (5).

- Service

```
// omitted

@.Inject
UpdateDAO updateDAO;

// omitted

public BinaryFile uploadBinaryFile(String fileName,
    InputStream contentInputStream) {

    // (7)
    BinaryFile inputFile = new BinaryFile();
    inputFile.set fileId(UUID.randomUUID().toString());
    inputFile.set fileName(fileName);
    inputFile.set Content(contentInputStream);

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

    return inputFile;
}

// omitted
```

Sr. No.	Description
(7)	<p>Set the information required for registering records in DTO.</p> <p>In the above example, file ID is numbered as UUID and the file name received by argument and the <code>InputStream</code> object that stores the file contents are set in DTO.</p>
(8)	<p>With DTO that stores information required for registration as the argument, call <code>UpdateDAO</code>.</p> <p>Method for calling DAO is same as the method when BLOB is not handled.</p>

- Controller

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

Sr. No.	Description
(9)	Call Service method with uploaded file name and <code>InputStream</code> that stores the file contents in argument.

Implementation for fetching LOB type

Implementation for fetching Large Objects like BLOB and CLOB from database is shown below.

In the following example, records are fetched from the table that handles BLOB.

For details on DDL and DTO (JavaBean) that are used to create necessary tables, refer to [LOB update implementation](#).

- xxx-sqlmap.xml

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

```

</resultMap>

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

```

Sr. No.	Description
(1)	Specify the definition required to fetch the value for the property that stores the value fetched from BLOB column. Specify "BLOB" in jdbcType attribute and "jp.terasoluna.fw.orm.ibatis.support.BlobInputStreamTypeHandler" in typeHandler attribute.
(2)	SQL for fetching records from the table with BLOB column.

Note: While handling CLOB, specify "CLOB" in jdbcType attribute and "jp.terasoluna.fw.orm.ibatis.support.ClobReaderTypeHandler" in typeHandler attribute.

- Service / Repository

```

// omitted

@Inject
QueryDAO queryDAO;

// omitted

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

// omitted

```

Sr. No.	Description
(3)	With fetch conditions specified by the controller as argument, call <code>QueryDAO</code> . In the above example, upload file information matching with file ID is fetched. Method for calling DAO is same as the method when BLOB is not handled.

SQL mapping implementation (dynamic SQL version)

There is default mechanism provided in Mybatis to build SQL dynamically.

Methods to build SQL dynamically are given below.

For details, refer to “Dynamic Mapped Statements” (P.48-53) of Developer Guide (PDF).

Determining whether parameter object is specified

SQL can be built by determining whether the parameter object passed to SQL is specified.

Decision elements are as follows:

Sr. No.	Element	Description
1.	<code>isParameterPresent</code>	Element for building SQL when parameter object is specified (Not NULL).
2.	<code>isNotParameterPresent</code>	Element for building SQL when parameter object is not specified (NULL).

Implementation is as follows:

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

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

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

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

```
</select>
```

Sr. No.	Description
(1)	In the example, id column is set in WHERE clause when the parameter object is specified.
(2)	In the example, the condition “1=2” is set so that there are no matching records, when the parameter object is not specified.

Following 2 patterns of SQL are generated by the dynamic SQL mentioned above.

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

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

Determining whether parameter object (JavaBean) properties exist

SQL can be built by determining whether the properties specified in parameter object (JavaBean) passed to SQL, exist.

Decision elements are as follows:

Sr. No.	Element	Description
1.	isPropertyAvailable	Element for building SQL when the specified property exists.
2.	isNotPropertyAvailable	Element for building SQL when the specified property does not exist.

Implementation is as follows:

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

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

        <isNotPropertyAvailable property="statusCode"> <!-- (2) -->
```

```
<![CDATA[  
    status_code <> 'completed'  
]]>  
</isNotPropertyAvailable>  
  
<!-- ... -->  
  
</select>
```

Sr. No.	Description
(1)	In the example, when <code>statusCode</code> property exists, WHERE clause is set such that <code>status_code</code> column fetches records matching with <code>statusCode</code> .
(2)	In the example, when <code>statusCode</code> property does not exist, WHERE clause is set such that <code>status_code</code> column fetches records other than ' <code>completed</code> '.

Following 2 patterns of SQL are generated by dynamic SQL mentioned above.

```
-- (1) statusCode='checking'  
SELECT * FROM t_order WHERE status_code = 'checking'  
  
-- (2)  
SELECT * FROM t_order WHERE status_code <> 'completed'
```

Determining whether property value of parameter object (JavaBean) is set

SQL can be built by determining whether value is specified in the parameter object (JavaBean) property passed to SQL.

Determination of elements is as follows:

Sr. No.	Element	Description
1.	<code>isNull</code>	Element for building SQL when property value is <code>null</code> .
2.	<code>isNotNull</code>	Element for building SQL when property value is not <code>null</code> .
3.	<code>isEmpty</code>	Element for building SQL when property value is <code>null</code> or empty. It can be specified for <code>Collection</code> and <code>String</code> .
4.	<code>isNotEmpty</code>	Element for building SQL when property value is <code>null</code> and not empty. It can be specified for <code>Collection</code> and <code>String</code> .

Implementation is as follows:

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

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

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

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

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

</select>
```

Sr. No.	Description
(1)	In the example, when <code>orderedDate</code> property (Date type) value is null, WHERE clause is set such that the <code>ordered_date</code> column can fetch records starting from one month prior to the current date.
(2)	In the example, when <code>orderedDate</code> property (Date type) value is not null, WHERE clause is set such that the <code>ordered_date</code> column can fetch records that match with <code>orderedDate</code> .
(3)	In the example, when <code>statusCodes</code> property (List<String> type) value is empty, WHERE clause is set such that the <code>status_code</code> column can fetch records other than 'completed'.
(4)	In the example, when <code>statusCodes</code> property (List<String> type) value is not empty, WHERE clause is set such that the <code>status_code</code> column can fetch records that match with any of the values stored in <code>statusCodes</code> . iterate element is described later.

Following 4 SQL patterns are generated by the dynamic SQL mentioned above.

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

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

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

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

Determining the property value of parameter object (JavaBean)

SQL can be built by determining the value specified in the property of parameter object (JavaBean) passed to SQL.

Decision elements are as follows:

Sr. No.	Element	Description
1.	isEqual	Element for building SQL when property value matches with the specified value.
2.	isNotEqual	Element for building SQL when property value does not match with the specified value.
3.	isGreaterThan	Element for building SQL when property value is greater than the specified value.
4.	isGreaterEqual	Element for building SQL when property value is greater than or equal to the specified value.
5.	isLessThan	Element for building SQL when property value is less than the specified value.
6.	isLessEqual	Element for building SQL when property value is less than or equal to the specified value.

Implementation is as follows:

```

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

```

Sr. No.	Description
(1)	In the example, when the <code>containCompletedOrder</code> property (Boolean type) value is <code>true</code> , WHERE clause is set such that the <code>status_code</code> column fetches ' <code>completed</code> ' records as well.

Note: Other property values in JavaBean can be compared by using `compareProperty` attribute.

Following 2 SQL patterns are generated by the dynamic SQL mentioned above.

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

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

Common attributes of decision elements

Elements that build dynamic SQL have the following common attributes.

Sr. No.	Attribute	Description
1.	prepend	When the SQL statement is determined as <code>true</code> by the decision element used for building dynamic SQL, specify the character string set at the beginning of the SQL.
2.	open	Specify the character string to be added before the built SQL in the decision element used for building dynamic SQL.
3.	close	Specify the character string to be added at the end of the built SQL in the decision element used for building dynamic SQL.

Implementation is as follows:

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

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

```

</isNotEmpty>

</select>

```

Sr. No.	Attribute	Description
(1)	-	In the example, when a value is specified in <code>statusCode</code> property, on setting the <code>WHERE</code> clause in <code>status_code</code> column, if the <code>containCompletedOrder</code> property (Boolean type) value is <code>true</code> , <code>WHERE</code> clause is set such that <code>status_code</code> column fetches ' <code>completed</code> ' records as well.
-	prepend	" <code>WHERE</code> " clause is set in SQL when value is specified in <code>statusCode</code> property.
-	open	When the <code>containCompletedOrder</code> property (Boolean type) value is <code>true</code> , opening character " <code>(</code> ", used for grouping the conditions for <code>status_code</code> column is specified, so as to add the OR condition.
-	close	Closing character " <code>)</code> " used for grouping the conditions for <code>status_code</code> column is specified.

Following 3 SQL patterns are generated by dynamic SQL mentioned above.

```

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

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

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

```

Collection iteration

When a collection or array of bind values is passed to SQL, the SQL can be built by iterating the process for element part of the collection and array.

Element is as follows:

Sr. No.	Element	Description
1.	iterate	Element that builds SQL by carrying out iteration process for collection and array.

Implementation is as follows:

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

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

</select>
```

Sr. No.	Attribute	Description
(1)	-	In the example, value stored in <code>statusCodes</code> property (<code>List<String></code>) is set as a value for IN clause.
	prepend	When collection or array elements exist, specify the character string to be set first. In the example, the column name to be added to the condition as well as IN clause are specified.
	open	Specify the character string to be set before processing the first element of collection or array. In the example, the starting enclosure character " (" of the value specified in IN clause is specified.
	conjunction	If next element exists, specify the character string to be set before processing the next element. In the example, delimiter ", " of the value specified in IN clause is specified.
	close	Specify the character string to be set after processing the last element of collection or array. In the example, ending enclosure character ") " of the value specified in IN clause is specified.

Note: The above example illustrates the implementation wherein the JavaBean properties are of collection type. However, the parameter objects themselves can be set to collection type. In that case, it can be accessed in # [] # format without specifying the property attribute.

JavaBean can also be stored in collection and JavaBean nested collection can also be accessed. For details, refer to “Dynamic Mapped Statements” (P.52) of the Developer Guide (PDF).

Following 3 SQL patterns are generated by the dynamic SQL mentioned above.

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

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

-- (3) statusCodes=['accepted','checking']
```

```
SELECT * FROM t_order WHERE status_code IN ('accepted' , 'checking')
```

Dynamic SQL blocking

By blocking individual dynamic SQL, prepend, open and close attributes can be controlled as an entire block.

The elements are as follows:

Sr. No.	Element	Description
1.	dynamic	Element that blocks the elements building the dynamic SQL.

Implementation is as follows:

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

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

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

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

    </dynamic>
</select>
```

Sr. No.	Attribute	Description
(1)	-	Dynamic SQLs of (2) and (3) are blocked.
-	prepend	Specify the character string to be prepended to the SQL built within the block. Value specified here is used as the prepend attribute value of the dynamic SQL matched in the block at first. In the above example, when value is specified in the <code>id</code> property, the value of <code>prepend</code> attribute in (2) becomes "WHERE" rather than "AND".
-	open	Specify the character string to be added before the SQL built in block.
-	close	Specify the character string to be added after the SQL built in block.

Following 4 SQL patterns are generated by the dynamic SQL mentioned above.

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

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

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

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

How to use QueryDAO

Single record search

Implementation is as follows when a query with search results as 0 to 1 record is issued.

- Xxx.java

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

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

Sr. No.	Description
(1)	Call the method (<code>QueryDAO#executeForObject</code>) that fetches search result as the object of type specified in (4).
(2)	Specify SQLID of the SQL with search result 0 to 1 record. Mybatis throws <code>java.sql.SQLException</code> when the search results are more than one.
(3)	Specify SQL bind parameter. In the example, bind parameter is <code>java.lang.String</code> . However, when multiple parameters (search conditions) are to be passed, JavaBean can also be specified.
(4)	Specify the type of object that maps the fetch results of SQL.
(5)	When the search result shows 0 records, the value becomes null. Hence, null check is necessary.
(6)	Implement the process when search result is 0 records.

Multiple records search

Implementation is as follows when query with search result 0 to N records is issued and all the data matching with the conditions is fetched.

- Xxx.java

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

Sr. No.	Description
(1)	Call the method to fetch the objects list.
(2)	Specify SQLID of the SQL with search result 0 to N records.
(3)	Specify the SQL bind parameter. In the example, bind parameter is set as boolean. However, when multiple parameters (search conditions) are to be passed, JavaBean can also be specified.
(4)	When search result is 0 records, an empty list is returned. null check is not required as null value is not returned.
(5)	Implement the process when search result is 0 records.

Pagination search (TERASOLUNA DAO standard function system)

When issuing a query with search result 0 to N records and fetching a part of data (specified part of page) matching with the condition, the implementation is as follows:

Following implementation is done by using the API provided by TERASOLUNA DAO.

Warning: Points to be noted when an extremely large number of data records match with search conditions

In the TERASOLUNA DAO standard function pagination search, the part up to starting position of the records that are fetched using `java.sql.ResultSet#next` is skipped. As a result, when the number of data records matching with search conditions is huge, it may impact the process performance. When there is a possibility of a huge number of data records matching the search conditions, SQL refinement should be adopted instead of pagination search of TERASOLUNA DAO standard function.

- Xxx.java

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

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

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

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

- xxx-sqlmap.xml

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

Sr. No.	Description
(1)	Generate paging search object (<code>org.springframework.data.domain.PageRequest</code>) provided by Spring Data. Pageable object can also be received as Controller argument by specifying in request parameters. For details, refer to Pagination .
(2)	Execute by specifying the SQLID of the SQL that fetches total number of data records matching with the condition.
(3)	Specify Long.class since this involves fetching records.
(4)	Execute by specifying the SQLID of the SQL with search result 0 to N records.
(5)	Specify the start position from where records are to be fetched. Starting from 0. When 10 records are to be fetched, on specifying 10, records from 11 to 20th are fetched.
(6)	Specify the number of records to be fetched. When the start position for fetching records is 0, on specifying 10, records from 1 to 10th are fetched.
(7)	Generate the page object (<code>org.springframework.data.domain.PageImpl</code>) provided by Spring Data.
(8)	Carry out pagination search and specify the fetched list.
(9)	Specify the paging search object (Pageable) used in pagination search.
(10)	Specify the total number of data records that match with the condition.
(11)	SQL implementation. It is not necessary to consider the fetch position in case of SQL.

Pagination search (SQL refinement)

When a query with search result 0 to N records is issued and a part of data (specified part of page) matching with the condition is fetched, the implementation is as follows:

Following implementation is done by using SQL instead of the API provided by TERASOLUNA DAO.

- PageableBindParams.java (Sample class)

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

- Xxx.java

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

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

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

```
totalCount); // (8)
```

- xxx-sqlmap.xml

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

Sr. No.	Description
(1)	JavaBean storing the parameters (bind parameter) that form search conditions and paging search object provided by Spring Data (<code>org.springframework.data.domain.Pageable</code>). An aggregate object like this class is necessary since only a single object can be passed to DAO. This class should be provided as required in each project since it is a sample implementation. Fetch the total number of records similar to using TERASOLUNA DAO standard function.
(2)	
(3)	Generate the bind object to be passed to DAO. In the example, class provided in (1) is used.
(4)	Specify the search condition to narrow down target data. In the example, “false” is specified as the value of “finished”.
(5)	Specify the search condition for narrowing down data on the corresponding page. In the example, paging search object provided by Spring Data (<code>org.springframework.data.domain.PageRequest</code>) is specified. The Pageable object can also be received as Controller argument by specifying it in request parameters. For details, refer to Pagination .
(6)	Specify SQLID of the SQL executed by the SQL that extracts data on the corresponding page.
(7)	Specify the bind object generated in (3).
(8)	Same as using TERASOLUNA DAO standard function, generate the page object provided by Spring Data (<code>org.springframework.data.domain.PageImpl</code>).
(9)	SQL implementation. In the example, PostgreSQL function (OFFSET, LIMIT) are used. As an SQL, notice the fetch position.
(10)	Specify the start position from where records are to be fetched. Starting from 0. When 10 records are to be fetched, on specifying 10, records from 11 to 20th
472	are fetched. (Use PostgreSQL function). 5 Architecture in Detail - TERASOLUNA Global Framework
(11)	Specify the number of records to be fetched. When the start position for fetching records is 0, on specifying 10, records from 1 to 10th are

How to use UpdateDAO

Inserting a single record

The implementation is as follows when a single data record is to be inserted.

- Xxx.java

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

Sr. No.	Description
(1)	Generate the data to be inserted (JavaBean).
(2)	Execute DAO by specifying SQLID of Insert SQL and the data to be inserted (JavaBean).
(3)	If required, check the number of data records actually inserted. In the example, it is checked whether a single record is inserted.
(4)	If required, carry out the process when the number of actually inserted records differs from number of expected records.

Inserting multiple records (batch execution)

Implementation in case of inserting multiple data records through batch execution of multiple SQLs is as follows:

Use `jp.terasoluna.fw.dao.SqlHolder` provided by TERASOLUNA DAO.

- Xxx.java

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

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

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

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

Sr. No.	Description
(1)	Generate the data to be inserted (JavaBean). 1st data record.
(2)	Generate the data to be inserted (JavaBean). 2nd data record.
(3)	Generate a list of <code>jp.terasoluna.fw.dao.SqlHolder</code> provided by TERASOLUNA DAO, for batch execution.
(4)	Add the data generated in (1), (2) to the SqlHolder list as bind objects. In the example, 2 records are added to the list.
(5)	Execute batch by specifying the SqlHolder list generated in (1)~(4).
(6)	If required, check the number of actually inserted data records. In the example, it is checked whether 2 records are inserted.
(7)	If required, carry out the process when the number of actually inserted records differs from the number of expected records.

Warning: About number of records inserted in batch execution

There are cases where, on batch execution, exact number of rows cannot be fetched using JDBC driver. When a driver that is unable to fetch the exact number of rows is used, batch execution should not be carried out for the cases wherein number of inserted records need to be checked. (This holds true for updated records and deleted records as well).

Updating a single record

Implementation is as follows while updating a single data record.

It is the same as in case of inserting a single record. SQL to be used is Update SQL.

- Xxx.java

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

Sr. No.	Description
(1)	Search the data to be updated (JavaBean).
(2)	Update data. In the example, “finished” is updated from false to true.
(3)	Execute DAO by specifying SQLID of the Update SQL and the data to be updated (JavaBean).
(4)	If required, check the number of actually updated data records. In the example, it is checked whether a single record is updated.
(5)	If required, carry out the process when the number of actually updated records differs from the number of expected records.

Updating multiple records (batch execution)

Implementation for updating multiple data records through batch execution is same as in case of inserting multiple records (batch execution).

Updating multiple records by batch execution is effective when the update value differs for each record.

Updating multiple records (Specifying WHERE clause)

Implementation is as follows when the data that matches with the condition specified by SQL is collectively updated.

Updating multiple records by specifying WHERE clause is effective when all records are collectively updated to the same value.

- Xxx.java

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

- xxx-sqlmap.xml

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

Sr. No.	Description
(1)	Execute DAO by specifying SQLID of the SQL used for batch update and condition that extracts the data to be updated.
(2)	Batch update SQL implementation. In the example, string “[Finished Deadline]” is added to the beginning of incomplete TODO titles for which 7 days have passed since their creation.

Deleting a single record

Implementation is as follows while deleting a single data record.

- Xxx.java

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

Sr. No.	Description
(1)	Execute DAO by specifying PK and SQLID of the Delete SQL. In the example, <code>java.lang.String</code> is specified. However, JavaBean can also be specified in case of composite keys.
(2)	If required, carry out the process when the number of actually deleted records differs than the expected records.

Deleting multiple records (batch execution)

Implementation for deleting multiple data records through batch execution using multiple SQL is same as that in case of updating multiple records (batch execution).

When it is necessary to share the process for deleting a single record, use multiple records deletion by batch execution. However, when a large amount of data is to be deleted, the batch deletion by specifying the WHERE clause should be considered.

Deleting multiple records (Specifying WHERE clause)

Implementation for collectively deleting the data matching with SQL specified conditions is same as that in case of updating multiple records (batch execution).

However, when a large number of records are to be deleted, batch deletion by specifying WHERE clause is effective.

Example showing use of StoredProcedureDAO

The implementation is as follows when a procedure or a function is called.

- Xxx.java

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

Sr. No.	Description
(1)	Generate the bind object that stores IN and OUT parameters of a procedure or function.
(2)	Set the ID for IN parameter. In the example, 1 is set as the ID.
(3)	Call StoredProcedureDAO method with bind object and SQLID of the SQL used for calling stored procedure as the argument.
(4)	When the method of StoredProcedureDAO is called successfully, OUT parameter of the procedure or function is set in bind object. In the example, the OUT parameter value set in bind object is output to log.

Example showing use of QueryRowHandleDAO

- Xxx.java

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

Sr. No	Description
(1)	Specify SQLID of the SQL with search result 0 to N records.
(2)	Specify SQL bind parameter. In the example, it is specified as boolean. However, when multiple parameters (search conditions) are passed, JavaBean can also be specified.
(3)	Specify the implementation object of <code>jp.terasoluna.fw.dao.event.DataRowHandler</code> . In the example, an unnamed class is used. However, creating an implementation class should be considered in actual project.
(4)	handleRow method is called for each record of search result. Object that is passed to an argument becomes object of the class specified in class attribute of resultMap element or the class specified in resultClass attribute of select element.
(5)	In the example, only log output is carried out. However, while using it in actual project, processes for file output, aggregate of each record value and value processing etc. are carried out.

Escaping during LIKE search

While performing LIKE search, the values to be used as search conditions need to be escaped.

Escaping for LIKE search can be done using methods of

`org.terasoluna.gfw.common.query.QueryEscapeUtils` class provided by common library.

For specifications of escaping provided by common library, refer to [Escaping during LIKE search of Database Access \(Common\)](#).

How to use escaping provided by common library, is explained below.

How to use escaping when matching method is specified in Query

When matching method (forward match, backward match and partial match) are specified as JPQL, use a method that performs only escaping.

- xxx-sqlmap.xml

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

Sr. No.	Description
(1)	Specify the wild card ("%" or "_") for LIKE search, in SQL. In the above example, a matching method is a “partial match” by specifying wild card (“%”) before and after word argument.
(2)	Specify "ESCAPE ' ~'" at the end of LIKE clause since " ~" is used as escape character in the escaping provided by common library.

- Service or Repository

```
@Inject
QueryDAO queryDAO;

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

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

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

```

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

```

Sr. No.	Description
(3)	When the matching method of LIKE search is specified in Query, call <code>QueryEscapeUtils#toLikeCondition(String)</code> method and perform escaping for LIKE search only.
(4)	Pass the value escaped for LIKE search, to the bind parameter of <code>QueryDAO</code> .

How to use escaping when the matching method is specified in Logic

When matching method (forward match, backward match and partial match) is determined at Logic side, use the method wherein wild card is assigned to the escaped value.

- `xxx-sqlmap.xml`

```

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

```

Sr. No.	Description
(1)	Do not specify the wild card for LIKE search in SQL.

- Service or Repository

```

@.Inject
QueryDAO queryDAO;

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

```

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

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

Sr. No.	Description
(2)	Call any one of the following methods while specifying the matching method in logic and assign the Escape and wild card values for LIKE search. QueryEscapeUtils#toStartingWithCondition(String) QueryEscapeUtils#toEndingWithCondition(String) QueryEscapeUtils#toContainingCondition(String)
(3)	Pass the escape + wild card value assigned for LIKE search to the bind parameter of QueryDAO.

About the measures to be taken against SQL Injection

When building SQL, it is necessary to ensure prevention of SQL Injection.

Mybatis2 provides 2 methods for embedding values in SQL.

- Method to embed values using bind variable

By this method, values can be safely embedded post SQL building, using
java.sql.PreparedStatement.

When embedding an input value from a user in SQL, bind variable should be used.

- Method to embed values using substitution variable.

This method does not assure safe embedding of values since they get substituted as strings during SQL building.

Warning: It should be noted that, when an input value from the user is embedded using substitution variable, a risk of SQL Injection attack is higher. When it is necessary to use substitution variable to embed input value from the user, input check that ensures prevention of SQL Injection must always be implemented.

Basically, **it is strongly recommended not to use the values entered by a user as is.**

Embedding values using bind variable

Use ParameterMap or Inline Parameters in case of bind variable.

An example of usage is shown below.

Following example shows the use of ParameterMap.

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

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

Sr. No.	Description
(1)	Define the property that embeds the value as bind variable. Defined order should correspond with the position of ? specified in (2).
(2)	Specify the bind variable in SQL. Values are bound to ? part as per the order defined in (1).

Following example shows the use of Inline Parameters.

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

Sr. No.	Description
(3)	Specify name of the property that stores bind value as bind variable by enclosing it in #.

How to embed a value using substitution variable

Following example shows the use of bind variable.

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

Sr. No.	Description
(4)	Enclose name of the property that stores the value to be substituted, with \$ and specify it as substitution variable. In the above example, \${direction\$} is replaced either by "DESC" or "ASC".

Warning: It is recommended to embed values using substitution variable on ensuring that the value is safe for the application and limiting its usage to table names, column names and sort conditions etc.

As shown in the example below, it is desirable to store the code value and the safe value to be actually used, in Map as a pair

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

and to convert the value during SQL execution so that actual input is changed to code value.

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

Codelist can also be used.

5.3.3 Appendix

Implementation wherein related objects are fetched collectively by a single SQL

How to fetch related objects collectively by a single SQL while accessing database, by providing a JavaBean such as Entity for each table, is explained below.

This method is also used to prevent N+1.

Warning: Following points should be noted while using this implementation.

- In this example, all related objects are fetched collectively in a single SQL, in order to explain the method. However, in actual project, it must be ensured to fetch only the related objects that are required in the process. This is because the simultaneous fetching of unused related objects can result in performance degradation in some cases.
- Related objects with 1:N ratio that are rarely used, are not fetched collectively. Fetching such related objects individually, when required, has proved to be better. They may be fetched collectively if the performance requirements are satisfied.
- When there are many related objects with 1:N ratio, fetching them collectively has degraded the performance in some cases as unnecessary data not used in mapping is fetched. They may be fetched collectively if the performance requirements are satisfied. However, it is better to look out for other methods.

Tip: For details on how to avoid N+1, refer to “Result Maps/Avoiding N+1 Selects (1:1)” (P.37-38) and “Result Maps/Avoiding N+1 Selects (1:M and M:N)” (P.39-40) of Mybatis Developer Guide (PDF).

Implementation using order table is explained below.

Table used is as follows:

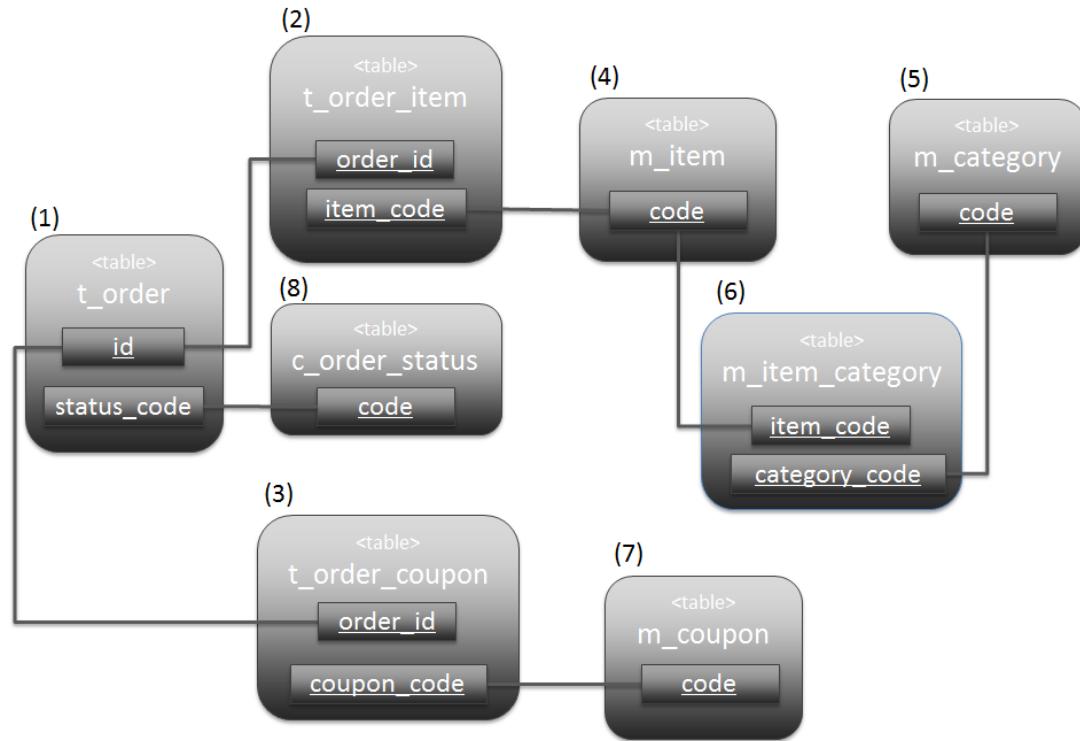


Figure.5.15 Picture - ER diagram

Sr. No.	Category	Table name	Description
(1)	Transaction system	t_order	Table that stores the order. 1 record is stored for 1 order.
(2)		t_order_item	Table that stores products purchased in a single order. When multiple products are purchased in a single order, records of multiple products are stored.
(3)		t_order_coupon	Table that stores coupons used in one order. When multiple coupons are used in one order, multiple coupon records are stored. No record is stored if a coupon is not used.
(4)	Master system	m_item	Master table for defining the product.
(5)		m_category	Master table for defining the category.
(6)		m_item_category	Master table for defining category to which the product belongs. It stores the mapping of product and category. It serves as a model wherein a single product can belong to multiple categories.
(7)		m_coupon	Master table for defining the coupon.
(8)	Code system	c_order_status	Code table for defining the order status.

Transaction table layout and records stored are as follows:

t_order

id(PK)	status_code
1	accepted
2	checking

t_order_item

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

t_order_coupon

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

Master table layout and records stored are as follows:

m_item

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

m_category

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

m_item_category

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

m_coupon

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

Code table layout and records stored are as follows:

c_order_status

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

In the following implementation, data stored in above tables is fetched by mapping with the JavaBean mentioned below.

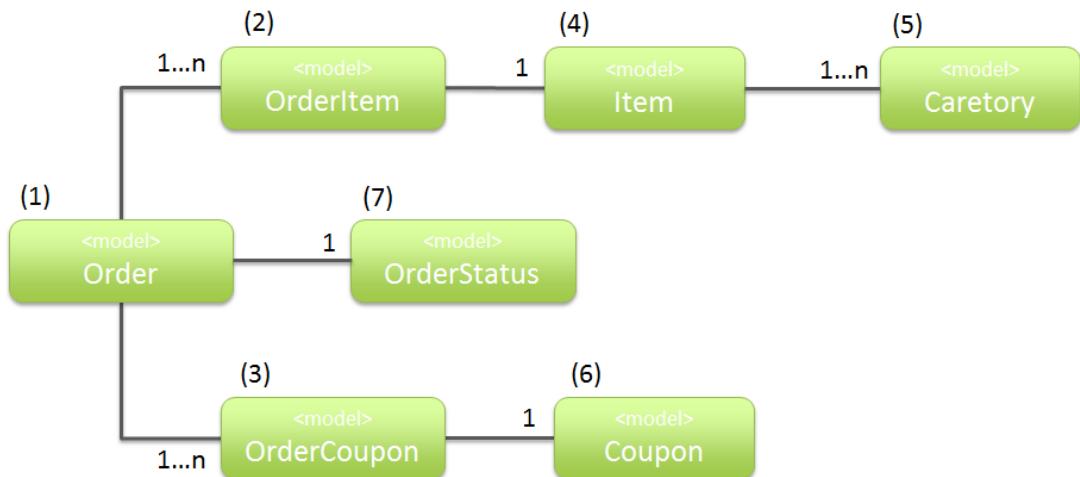


Figure.5.16 Picture - Class (JavaBean) diagram

Sr. No.	Class name	Description
(1)	Order	JavaBean that represents one record of t_order table. OrderStatus, OrderItem and OrderCoupon are stored as multiple related objects.
(2)	OrderItem	JavaBean that represents one record of t_order_item table. Item is stored as related object.
(3)	OrderCoupon	JavaBean that represents one record of t_order_coupon table. Coupon is stored as related object.
(4)	Item	JavaBean that represents one record of m_item table. Multiple Category to which the records belong are stored as related objects. Item and Category are associated by m_item_category table.
(5)	Category	JavaBean that represents one record of m_category table.
(6)	Coupon	JavaBean that represents one record of m_coupon table.
(7)	OrderStatus	JavaBean that represents one record of c_order_status table.

JavaBean property definition is as follows:

- Order.java

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

- OrderItem.java

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

```
}
```

Sr. No.	Description
(1)	Value to be stored overlaps with code property of the immediately subsequent item variable. This is defined since it is required while grouping the records using groupBy attribute of resultMap element, described later.

- OrderCoupon.java

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

Sr. No.	Description
(1)	Value to be stored overlaps with code property of the immediately subsequent Coupon variable. This is defined since it is required while grouping the records using groupBy attribute of resultMap element, described later.

- Item.java

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

- Category.java

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

- Coupon.java

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

- OrderStatus.java

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

Implement SQL mapping.

When related objects are collectively fetched in a single SQL, JOIN the tables to be fetched and acquire all the records necessary for mapping.

Perform mapping definition for the fetched records in resultMap element and map with JavaBean.

Following implementations show the SQL that fetches a single Order (findOne) and one that fetches all Orders (findAll).

Further, in following implementation, "order" is specified in name space and the class name without package name, defined using typeAlias, is assumed for the JavaBean to be mapped.

- sqlMapConfig.xml

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

- order-sqlmap.xml

```
<sqlMap namespace="order">  
    <!-- ... -->  
</sqlMap>
```

At first, describe the SQL

Actual definition should be subsequent to the resultMap element.

Implementing SQL

- SQL definitions (sql elements) of the common parts of findOne/findAll

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

Sr. No.	Description
(1)	sql elements used to share SELECT clause, FROM clause and JOIN clause in findOne and findAll. They have been standardized as there are many common parts in the SQLs for findOne and findAll.
(2)	Fetch all the data required to generate related objects. Ensure that the column name is not repeated. In the above example, since there are duplicate code, name and price, an alias is specified for them using AS clause.
(3)	Join the tables storing the data required to generate related objects.
(4)	Perform Outer JOIN for the tables in which data may not be stored. When coupon is not used, records are not stored in t_group_coupon; hence Outer JOIN should be used. This is also applicable for t_coupon that joins with t_group_coupon.

- SQL definition for findOne

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

Sr. No.	Description
(1)	SQL for fetching related object and Order object of the specified order ID.
(2)	Includes the SQL that implements SELECT clause, FROM clause and JOIN clause which are shared with findAll.
(3)	Specify the order ID passed by bind value, in WHERE clause.
(4)	When related objects with 1:N ratio exist, specify the ORDER BY clause to control the order within the list. Need not be specified if there is no significance to the order.
(5)	To be specified in order to set the Order#orderItems list in ascending order of code column of t_item table.
(6)	To be specified in order to set the Item#categories list in ascending order of code column of t_category table.
(7)	To be specified in order to set the Order#orderCoupons list in ascending order of t_coupon codes.

- SQL definition of findAll

```

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

```

Sr. No.	Description
(1)	SQL to fetch all Orders and related objects.
(2)	sql element that shares the SELECT clause, FROM clause and JOIN clause in findOne and findAll.
(3)	To be specified so as to set the order of fetched list in descending order of t_order ids.

Following records are fetched on implementing the above SQL (findAll).

2 records are fetched as order records. However a total of 9 records are fetched as it joins with related table that stores multiple records.

1st~3rd rows are records that generate Order objects with order ID 2 whereas rows 4th~9th are records that generate Order objects with order ID 1.

id	status_code	status_name	quantity	item_code	item_name	item_price	category_code	category_name	coupon_code	coupon_name	coupon_price
2	checking	Stock checking	3	ITM0000001	Orange juice	100	CTG0000001	Drink	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000002	PC	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000003	Hot selling	<NULL>	<NULL>	<NULL>
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000001	Join coupon	3000
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000002	PC coupon	30000

Figure.5.17 Picture - Result Set of findAll

How to map the above records with Order object and related objects is explained below.

Implementing resultMap element

Each element is explained later.

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

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

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

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

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

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

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

The role of each resultMap element and its dependency is as shown below.

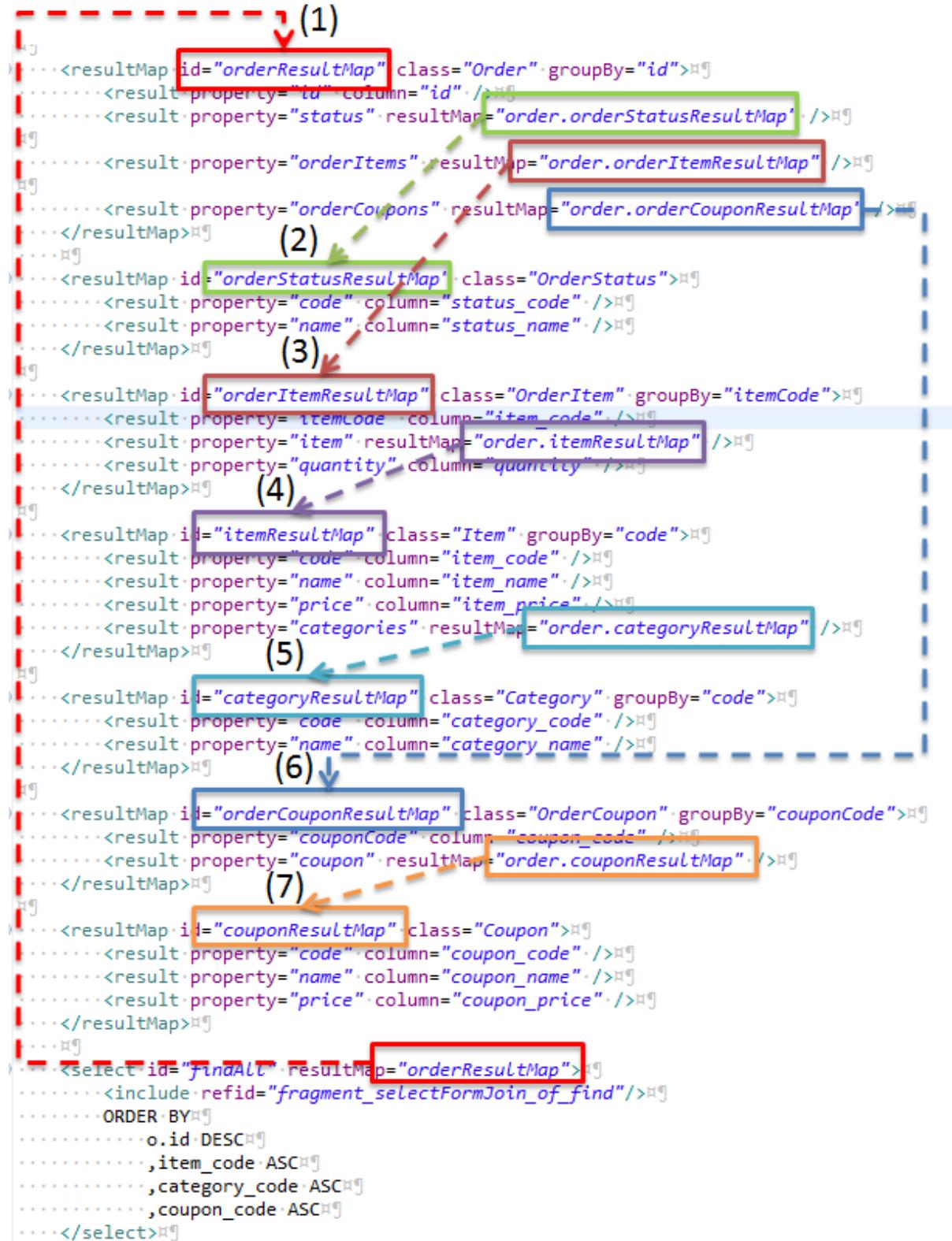


Figure.5.18 Picture - Implementation of ResultMap

Sr. No.	Description
(1)	Definition to map the fetched records to Order objects. Mapping of related objects (OrderStatus, OrderItem and OrderCoupon) is delegated to another resultMap.
(2)	Definition to map the fetched records to OrderStatus objects.
(3)	Definition to map the fetched records to OrderItem objects. Mapping of related object (Item) is delegated to another resultMap.
(4)	Definition to map the fetched records to Item objects. Mapping of related object (Category) is delegated to another resultMap.
(5)	Definition to map the fetched records to Category objects.
(6)	Definition to map the fetched records to OrderCoupon objects. Mapping of related object (Coupon) is delegated to another resultMap.
(7)	Definition to map the fetched records to Coupon objects.

Mapping the records to Order object.

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

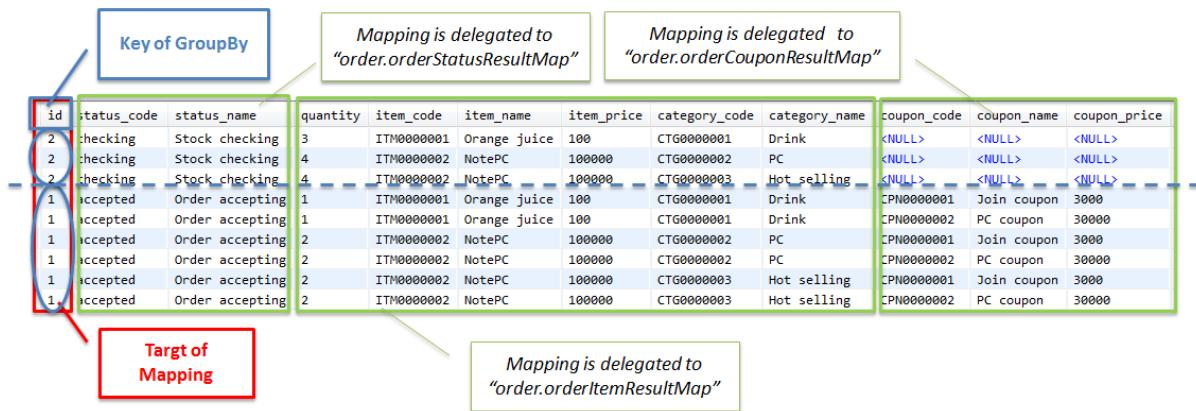


Figure.5.19 Picture - ResultMap for Order

Sr. No.	Description
(1)	<p>It is necessary to group the fetched records for each order. For this, specify the <code>id</code> property storing the value which uniquely identifies the order, in <code>groupBy</code> attribute.</p> <p>In this example, as the records are grouped by <code>id</code> property, 2 <code>Order</code> objects namely, <code>id=1</code> and <code>id=2</code> are generated.</p>
(2)	Set the <code>id</code> column value of fetched records in <code>Order#id</code> .
(3)	Delegate generation of <code>OrderStatus</code> objects to the <code>id="order.orderStatusResultMap"</code> <code>resultMap</code> and set the generated objects in <code>Order#status</code> .
(4)	Delegate generation of <code>OrderItem</code> objects to the <code>id="order.orderItemResultMap"</code> <code>resultMap</code> and add the generated objects in <code>Order#orderItems</code> list.
(5)	Delegate generation of <code>OrderCoupon</code> objects to the <code>id="order.orderCouponResultMap"</code> <code>resultMap</code> and add the generated objects in <code>Order#orderCoupons</code> list.

Following is described mainly focusing on `id=1` `Order` object mapping.

Mapping the records to `OrderStatus` objects.

```
<resultMap id="orderStatusResultMap" class="OrderStatus"> <!-- (1) -->
  <result property="code" column="status_code" /> <!-- (2) -->
  <result property="name" column="status_name" /> <!-- (3) -->
</resultMap>
```

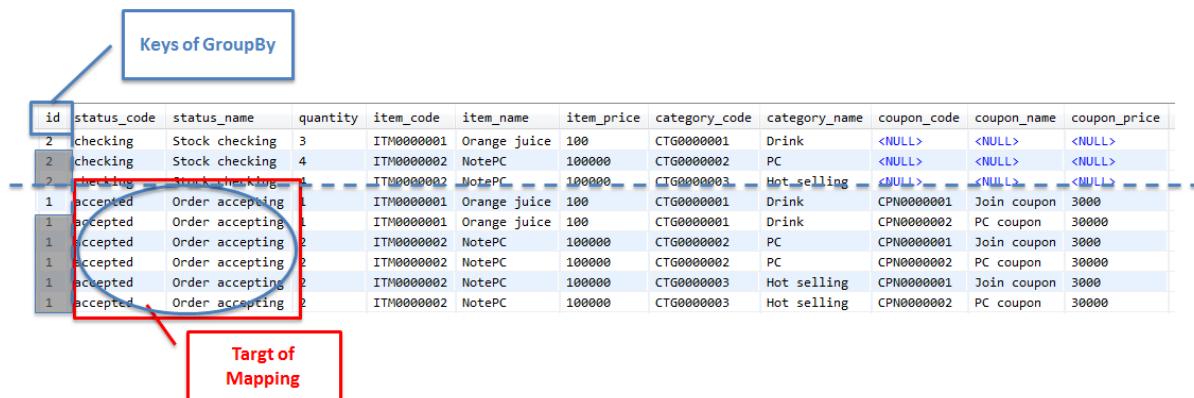


Figure.5.20 Picture - ResultMap for OrderStatus

Sr. No.	Description
(1)	<p>It is not necessary to specify groupBy attribute since there is a 1:1 relationship between Order and OrderStatus.</p> <p>In this example, OrderStatus object of code=accepted is generated.</p>
(2)	Set the status_code column values of fetched records to OrderStatus#code.
(3)	Set the status_name column values of fetched records to OrderStatus#name.

Mapping to OrderItem objects.

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

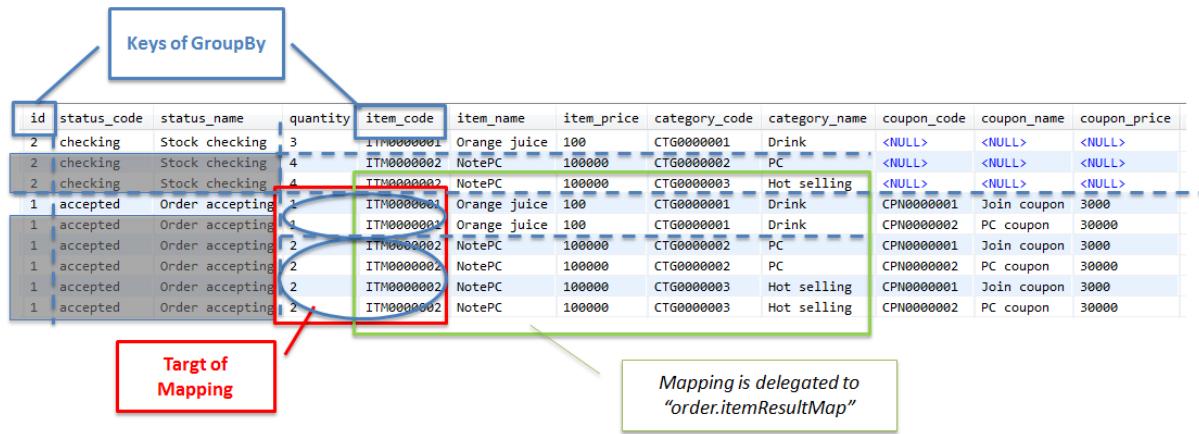


Figure.5.21 Picture - ResultMap for OrderItem

Sr. No.	Description
(1)	<p>It is necessary to specify groupBy attribute since there is a 1:N relationship between Order and OrderItem.</p> <p>Order products need to be grouped with the primary keys of t_order_item (order_id, item_code). Since order_id column is specified in the parent resultMap, specify only itemCode property that stores the values of item_code column.</p> <p>In this example, as the order products are grouped in itemCode property, 2 OrderItem objects namely itemCode=ITM000001 and itemCode=ITM000002 are generated.</p>
(2)	<p>item_code column values of fetched records are set in OrderItem#itemCode.</p> <p>itemCode property overlaps with Item#code generated in (3). However, it is necessary for grouping OrderItem.</p>
(3)	<p>Generation of Item object is delegated to id="order.itemResultMap" resultMap and the generated object is set in OrderItem#item.</p>
(4)	<p>quantity column values of fetched records are set in OrderItem#quantity.</p>

Mapping the records to Item objects.

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

```

<result property="price" column="item_price" /> <!-- (4) -->
<result property="categories" resultMap="order.categoryResultMap" /> <!-- (5) -->
/>resultMap>

```

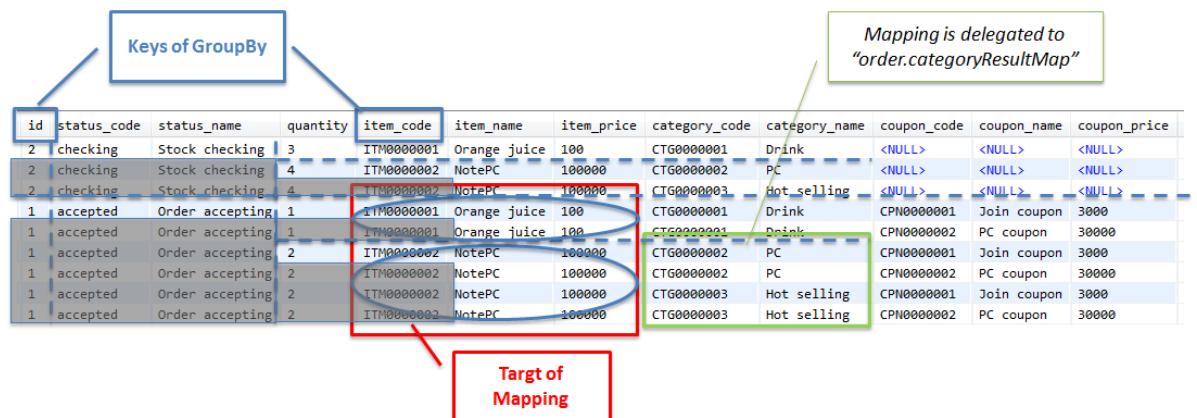


Figure.5.22 Picture - ResultMap for Item

Sr. No.	Description
(1)	<p>There is a 1:1 relationship between <code>OrderItem</code> and <code>Item</code>; however since <code>Item</code> and <code>Category</code> have 1:N relationship, <code>groupBy</code> attribute needs to be specified.</p> <p>It is necessary to group the categories for each product. Therefore, specify <code>code</code> property storing the value which uniquely identifies the order, in <code>groupBy</code> attribute.</p> <p>In this example, <code>Item</code> object of <code>code=ITM0000001</code> is generated for <code>OrderItem#itemCode=ITM0000001</code> and <code>Item</code> object of <code>code=ITM0000002</code> is generated for <code>OrderItem#itemCode=ITM0000002</code>. (Total 2 objects are generated.)</p>
(2)	<code>item_code</code> column values of fetched records are set in <code>Item#code</code> .
(3)	<code>item_name</code> column values of fetched records are set in <code>Item#name</code> .
(4)	<code>item_price</code> column values of fetched records are set in <code>Item#price</code> .
(5)	Delegate generation of <code>Category</code> objects to <code>id="order.categoryResultMap"</code> <code>resultMap</code> and add the generated objects to <code>Item#categories</code> list.

Mapping the records to Category objects.

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

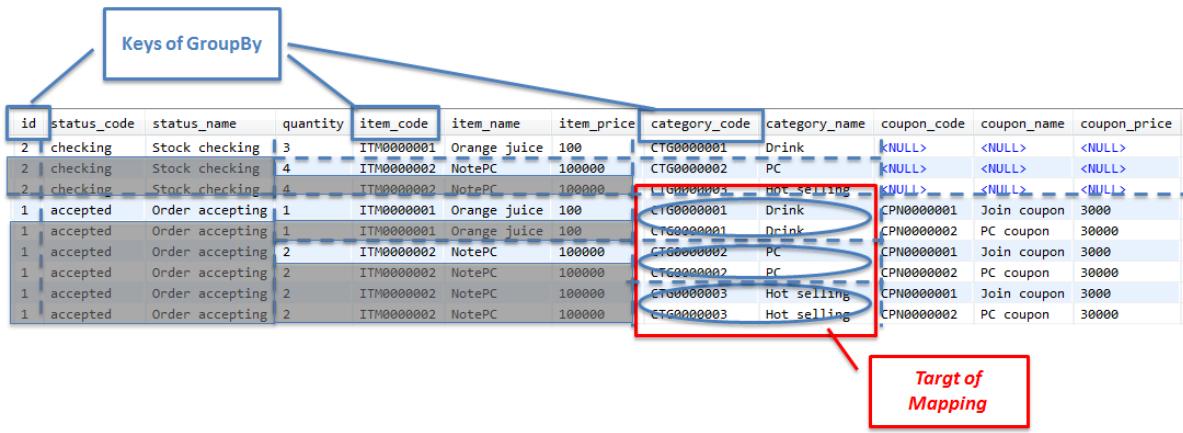


Figure.5.23 Picture - ResultMap for Item

Sr. No	Description
(1)	<p>In this example, multiple tables with 1:N relationship (t_order and t_order_line, t_order and t_order_coupon) are combined. As a result, if multiple records are stored in t_order_coupon, there is a duplication of list of Category objects stored in Item objects.</p> <p>In order to avoid this duplication, specify code property storing the value which uniquely identifies a category, in groupBy attribute. Category objects with same code property value are merged in one and thus preventing duplication.</p> <p>In this example, Category object code=CTG0000001 is generated for Item#code=ITM0000001 and 2 Category objects namely, code=CTG0000002 and code=CTG0000003 are generated for Item#code=ITM0000002. (Total 3 objects are generated.)</p>
(2)	item_code column values of fetched records are set in Item#code.
(3)	item_name column values of fetched records are set in Item#name.

Mapping the records to OrderCoupon objects.

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

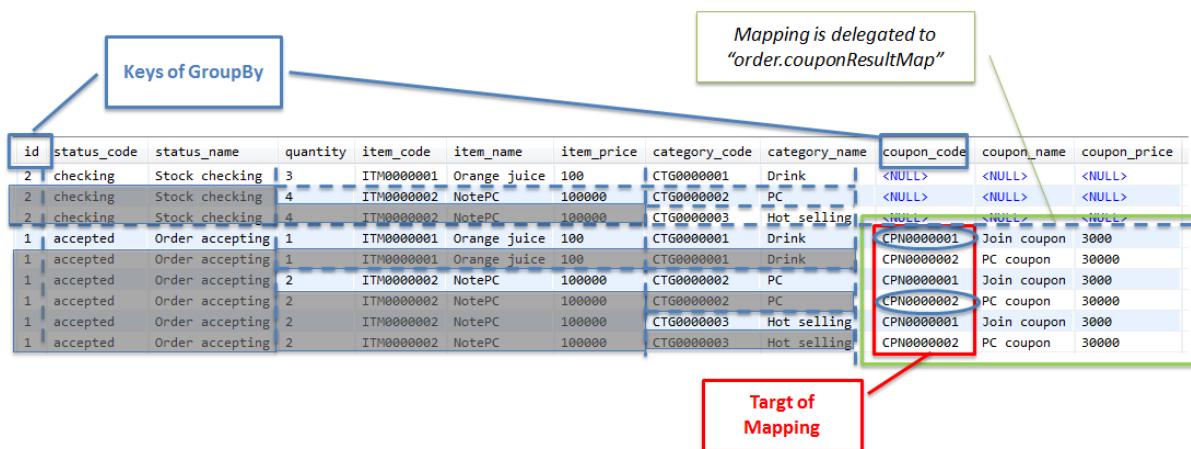


Figure.5.24 Picture - ResultMap for OrderCoupon

Sr. No.	Description
(1)	<p>It is necessary to specify groupBy attribute since there is a 1:N relationship between Order and OrderCoupon.</p> <p>Order coupons need to be grouped with the primary keys of t_order_coupon (order_id, coupon_code). However, order_id column is specified in the parent resultMap. Hence, specify only the couponCode property that stores coupon_code column values here.</p> <p>In this example, as the order coupons are grouped in couponCode property, 2 OrderCoupon objects, namely, couponCode=CPN0000001 and couponCode=CPN0000002 are generated.</p>
(2)	<p>coupon_code column values of fetched records are set in OrderCoupon#couponCode. couponCode property overlaps with Coupon#code generated in (3). However, the property is required for grouping OrderCoupon.</p>
(3)	<p>Delegate generation of Coupon object to id="order.couponResultMap" resultMap and set the generated objects in OrderCoupon#coupon.</p>

Mapping the records to Coupon objects

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

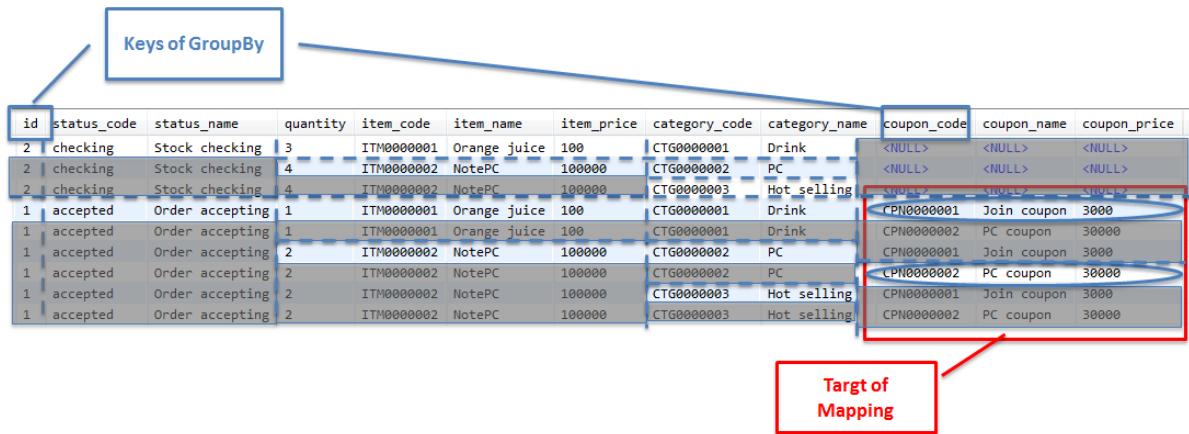


Figure 5.25 Picture - resultMap for Coupon

Sr. No.	Description
(1)	<p>It is not necessary to specify groupBy attribute since there is a 1:1 relationship between OrderCoupon and Coupon.</p> <p>In this example, Coupon object of code=CPN0000001 is generated for OrderCoupon#couponCode=CPN0000001 and Coupon object of code=CPN0000001 is generated for OrderCoupon#couponCode=CPN0000001. (Total 2 objects are generated.)</p>
(2)	coupon_code column values of fetched records are set in Coupon#code.
(3)	coupon_name column values of fetched records are set in Coupon#name.
(4)	coupon_price column values of fetched records are set in Coupon#price.

Records and columns mapped to JavaBean are as follows:

By specifying groupBy attribute, grayed out part is merged with the non-grayed part.

<u>id</u>	<u>status_code</u>	<u>status_name</u>	<u>quantity</u>	<u>item_code</u>	<u>item_name</u>	<u>item_price</u>	<u>category_code</u>	<u>category_name</u>	<u>coupon_code</u>	<u>coupon_name</u>	<u>coupon_price</u>
2	checking	Stock checking	3	ITM0000001	Orange juice	100	CTG0000001	Drink	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000002	PC	<NULL>	<NULL>	<NULL>
2	checking	Stock checking	4	ITM0000002	NotePC	100000	CTG0000003	Hot selling	<NULL>	<NULL>	<NULL>
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000001	Join coupon	3000
1	accepted	Order accepting	1	ITM0000001	Orange juice	100	CTG0000001	Drink	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000002	PC	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000002	PC coupon	30000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000001	Join coupon	3000
1	accepted	Order accepting	2	ITM0000002	NotePC	100000	CTG0000003	Hot selling	CPN0000002	PC coupon	30000

Figure.5.26 Picture - Valid Result Set for result mapping

Warning: It should be noted that when the records with 1:N relationship are mapped using JOINs, fetching the data of grayed out part becomes unnecessary.

If the same SQL is used in a process that does not use data from the N part, it results in fetching of unnecessary data. Therefore, two separate SQLs, one that fetches the N part and another that does not fetch the N part should be created.

The status of actually mapped Order objects and related objects is as follows:

Tip: Another method to fetch related objects includes executing a separate SQL internally, using values of fetched records. Implementing a separate SQL internally simplifies the definition of resultMap element and individual SQL to a great extent. However, it is necessary to be aware of the fact that fetching related objects by this method can cause N+1.

For details on how to execute a separate SQL internally, refer “Result Maps/Complex Properties” (P.36-37) and “Result Maps/Composite Keys or Multiple Complex Parameters Properties” (P.40-41) of Mybatis Developer Guide (PDF).

Tip: When a separate SQL is internally executed, related objects are “Eager Loaded”. As a result, SQL gets executed even when the related objects are not used. To avoid this, Mybatis provides an option of “Lazy Load” for related objects.

Settings to enable “Lazy Load” are as follows:

- Set the enhancementEnabled attribute of setting element of the Mybatis configuration file, to true.
- Add CGLIB 2.x to class path.

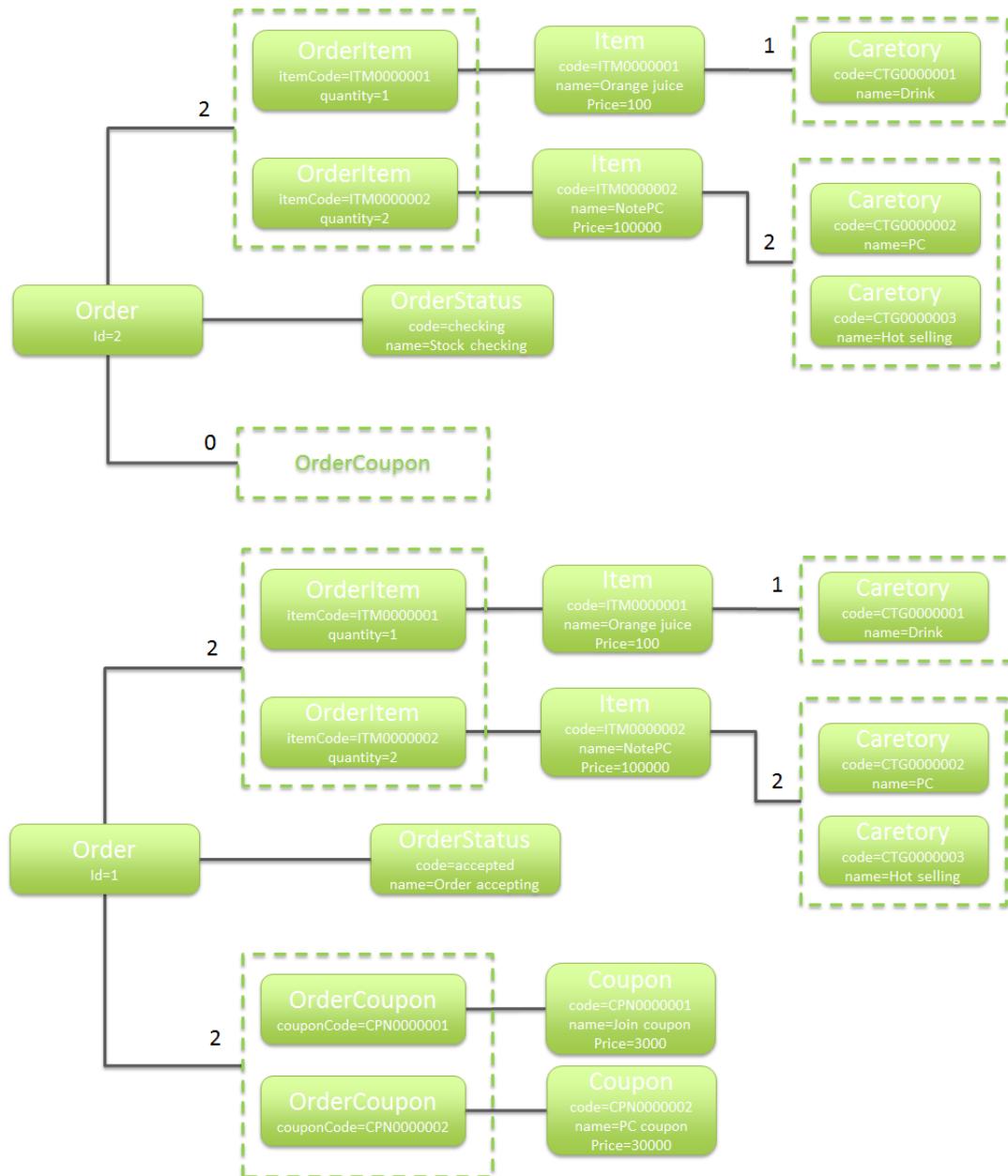


Figure 5.27 Picture - Mapped object diagram

5.4 Exclusive Control

5.4.1 Overview

Exclusive control is a process to maintain data consistency when same data is to be updated simultaneously by multiple transactions.

Exclusive control should be performed when same data is likely to get updated simultaneously by multiple transactions. These transactions are not necessarily restricted to database transactions only; they also include long transactions.

Note: Long transactions

Long transactions are the transactions where data fetch and data update are performed as separate database transactions.

To illustrate a specific example, the transactions are observed in an application where the fetched data is displayed on the Edit screen and the value edited on the screen is updated in database.

This chapter explains about exclusive control for the data stored in the database.

However, it should also be performed in a similar way for the data stored in data-store apart from database (such as memory, files etc.).

Necessity of exclusive control

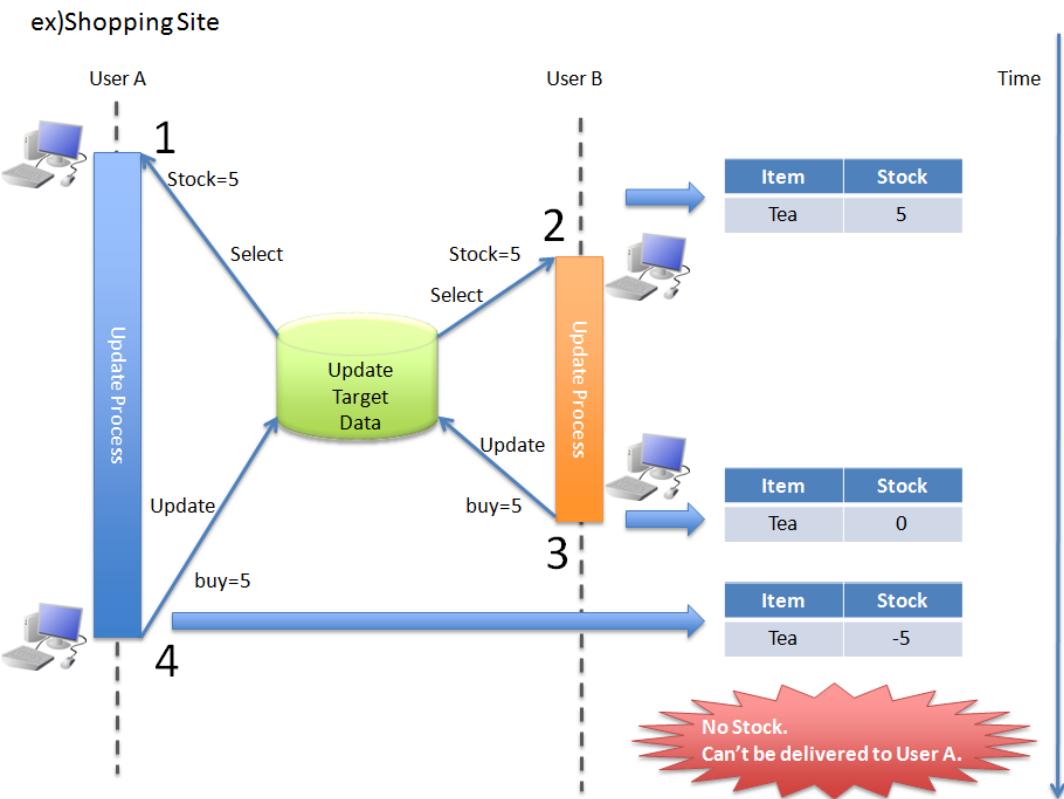
To understand why exclusive control is necessary, let us first see the issues that occur in the absence of exclusive control using the examples given below.

Problem 1

See the example below of receiving an order for Tea from users on a shopping site.

Sr. No.	UserA	UserB	Description
1.	<input type="radio"/>	-	User A confirms that the quantity of Tea on Product screen is 5nos.
2.	-	<input type="radio"/>	User B confirms that the quantity of Tea on Product screen is 5nos.
3.	-	<input type="radio"/>	User B orders 5nos. of Tea. Stock of Tea in the DB reduces by 5 and becomes 0.
4.	<input type="radio"/>	-	User A orders 5nos. of Tea. Stock of Tea in the DB reduces by 5 and becomes -5.

Order from User A is accepted however an apology is conveyed due to non-availability of actual stock.



The stock quantity of Tea stored in the table also shows a value (minus value) different from actual stock quantity of Tea.

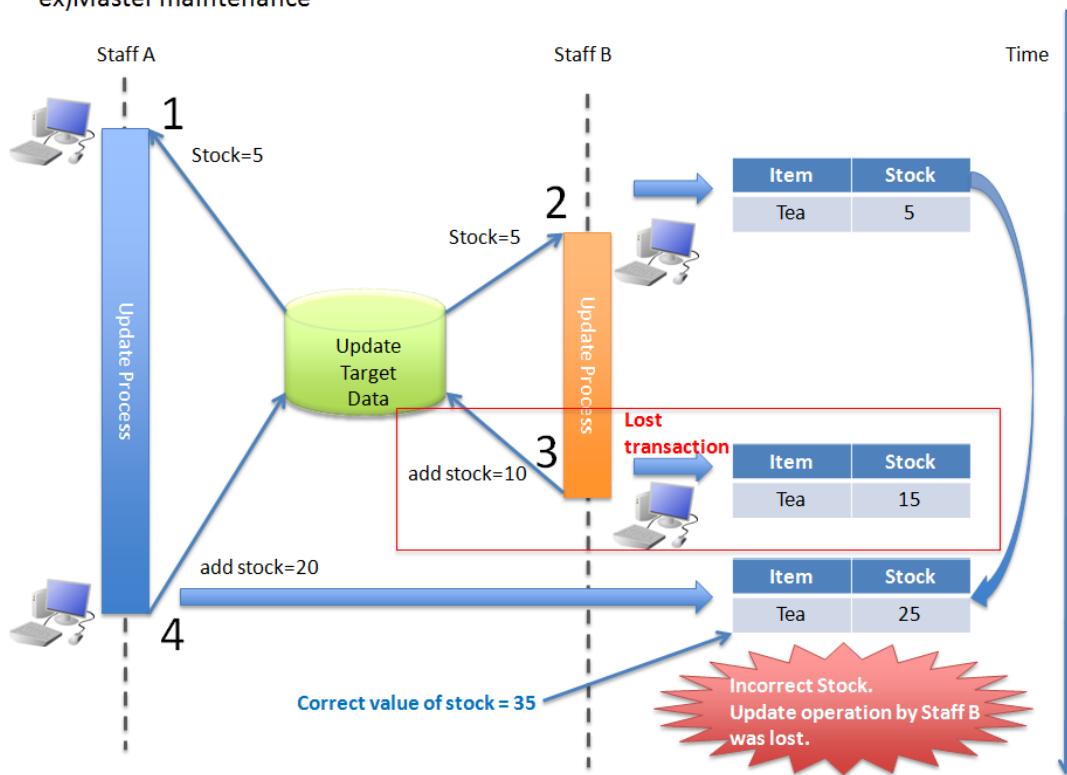
Problem 2

See the example below wherein the staff that manages the stock quantity of Tea on shopping site, displays the stock quantity of Tea, calculates the added stock quantity at Client side and updates the stock quantity of Tea.

Sr. No.	UserA	UserB	Description
1.	<input type="radio"/>	-	Staff A confirms that the quantity of Tea is 5nos.
2.	-	<input type="radio"/>	Staff B confirms that the quantity of Tea is 5nos.
3.	-	<input type="radio"/>	Staff B adds stock of 10nos. of Tea and updates the stock quantity on Client side as $5+10=15$.
4.	<input type="radio"/>	-	Staff A adds stock of 20nos. of Tea and updates the stock quantity on Client side as $5+20=25$.

The stock of tea 10nos. added in process 3 is lost and there is a mismatch in actual stock quantity (35nos.).

ex)Master maintenance

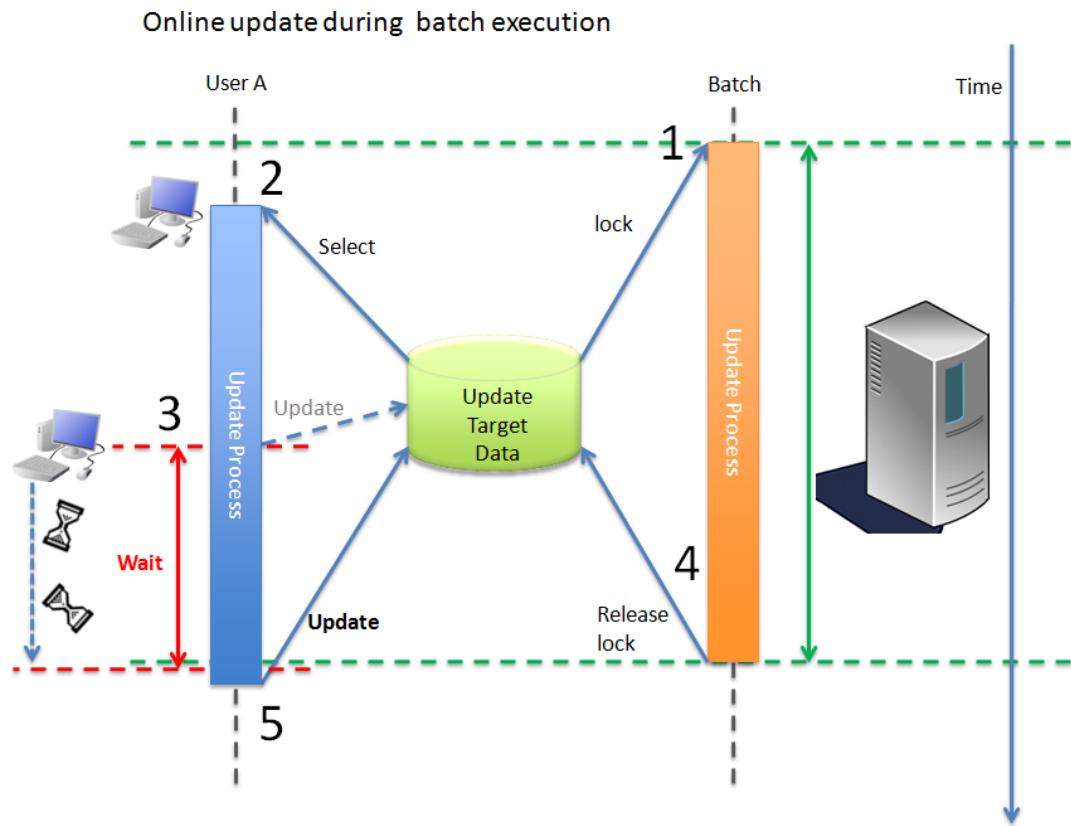


Problem 3

See the example below wherein the data locked by batch processing is updated by online processing.

Sr. No.	UserA	Batch	Description
1.	-	○	Batch locks the table row (temporarily all rows) to be updated and does not allow the update by other processes.
2.	○	-	User A searches the updated information. Since batch is not committed at this point, the information prior to batch update can be fetched.
3.	○	-	User A requests for update and waits as it is locked in batch.
4.	-	○	Batch terminates the process to release the lock.
5.	○	-	The update process for which user A was waiting can now be executed.

User A executes update process after waiting for the batch to release the lock. However, data fetched by User A prior to waiting is the data before batch update and batch processing may



overwrite the data with the updated data.

Batch takes a longer time as compared to online processing and user has to wait longer.

Exclusive control according to the isolation level of transaction

The easiest way to resolve all the 3 problems given in *Necessity of exclusive control* is to sequentially execute the database processes.

When the processes are sequentially executed, there is no effect on transactions.

However, when the processes are sequentially executed, the number of transactions that can be executed in a unit of time shows reduction resulting in deterioration in the performance.

In ANSI/ISO SQL standards, the guidelines indicating the isolation level (extent of impact by each transaction) are defined. The 4 isolation levels of transaction are given below. Events that occur at each isolation level are explained below.

Sr. No.	Isolation levels	DIRTY READ	NON-REPEATABLE READ	PHANTOM READ
1.	READ UNCOMMITTED	Yes No	Yes Yes	Yes
2.	READ COMMITTED	No	No	Yes
3.	REPEATABLE READ	No	No	Yes
4.	SERIALIZABLE	No	No	No

Tip: DIRTY READ

Dirty Read occurs when the data written by uncommitted transaction is read by other transaction.

Tip: NON-REPEATABLE READ

When the same record is likely to be read twice in the same transaction, if other transaction is committed during the first read and second read, the details read at first time and those at second time may differ. The multiple data readings may vary based on commit timing of other transaction.

Tip: PHANTOM READ

In Phantom Read, when a same record is being read twice in the same transaction, if other transaction adds or deletes the record, it causes difference in the number of records (details) fetched during the first read and second read.

The isolation level defined in above table gets higher as we go down.

If the isolation level is high, the data can be protected; however it increases the locking overhead resulting in

deterioration of the performance.

It is not desirable to select SERIALIZABLE unless the access frequency is fairly low.

This is because all the data is accessed sequentially one by one including SELECT.

The relationship between level of isolation and degree of concurrency between transactions is a Trade-off relationship.

In other words, high isolation level leads to reduction in concurrency and vice versa.

Thus, it is necessary to balance the level of isolation and degree of concurrency of transactions in accordance with application requirements.

The supported isolation level differs depending on the database to be used, it is necessary to understand the characteristics of the database to be used.

Isolation levels supported by each database and their default values are shown below.

Sr. No.	Database	READ UN-COMMITTED	READ COMMITTED	REPEATABLE READ	SERIALIZABLE
1.	Oracle	✗	○ (default)	✗	○
2.	PostgreSQL	✗	○ (default)	✗	○
3.	DB2	○	○ (default)	○	○
4.	MySQL InnoDB	○	○	○ (default)	○

When a balance is to be maintained between isolation and concurrency while maintaining data consistency, it is necessary to perform exclusive control using Database Locking which is described below.

Exclusive control using database locking

It is necessary to lock the data to be updated using an appropriate method due to the following reasons:

- To maintain consistency of data stored in the database
- To prevent conflicts in update process

The three types of methods to lock the data stored in the database are as follows:

The Architect should adequately understand the characteristics of such locking and use the appropriate locking method in accordance with the characteristics of application.

Table.5.11 Types of locking

Sr. No.	Types of locking	Applicable cases	Characteristics
1.	Automatic locking by RDBMS	<ul style="list-style-type: none"> • When the conditions necessary to ensure data consistency can be specified as update conditions for data. • When concurrency for the same data is less and update process also takes less time. 	<ul style="list-style-type: none"> • Effective since check and update process are executed using single SQL. • Conditions for ensuring data consistency need to be analyzed separately as compared to optimistic locking.
2.	Optimistic locking	<ul style="list-style-type: none"> • When the already fetched data is being updated by other transaction and if the updated contents need to be verified. • When concurrency for the same data is less and update process also takes less time. 	<ul style="list-style-type: none"> • Ensures that the fetched data is not updated by other transaction. • Column to manage versions needs to be defined in the table.
3.	Pessimistic locking	<ul style="list-style-type: none"> • When the data that is likely to remain in locked state for a longer period is updated. • When data consistency check needs to be carried out since optimistic locking cannot be used (column cannot be defined to manage versions). • When concurrency for the same data is more and update process takes longer time. 	<ul style="list-style-type: none"> • Possibility of a process failure due to process results of other transaction is eliminated. • Costly since it is necessary to execute SELECT statement for obtaining pessimistic lock.

Note: Standards for adopting types of locking

The Architect should decide the type of locking to be used based on the functional and performance requirements.

- Optimistic locking is necessary to ensure that the database transactions such as returning and changing the data on screen are cut off and the data remains unchanged in subsequent transaction.
 - When locking is needed in a single transaction, both pessimistic and optimistic locking can be implemented; however when pessimistic locking is used, lock is implemented at database level thus resulting in the possible increase in database processing cost. It is always preferable to use optimistic locking unless there are any specific issues.
 - If optimistic locking is used in a process with higher update frequency wherein multiple tables are to be updated in a single transaction, the waiting time for obtaining a lock can be minimized. However the possibility of error occurrences increases since an exclusive error may occur in between the process. If pessimistic locking is used, the waiting time for obtaining a lock is likely to increase; however since exclusive error does not occur once lock is obtained, it reduces the possibility of error occurrences.
-

Tip: Business transaction

In actual application development, there could also be cases where exclusive control is necessary for the transactions at business flow level. A typical example of business flow level transactions could be an application used at a travel agency for making reservations while talking to the customer.

While making travel reservations, means of transport such as railway, accommodation facilities and additional scheme, etc. are discussed. At this point, the reserved accommodation and additional scheme should remain unavailable for other users. In such a case, the status of table should be updated from ‘Temporarily Reserved’ to ‘Reserved’. Even if the reservation is in process, other users should not be able to make the corresponding reservation.

Description of exclusive control for business transaction is skipped in this chapter since it should be analyzed and designed under business process design or functional design.

Exclusive control using row lock function of the database

In most databases, when a record is updated (UPDATE, DELETE), a row lock is obtained to prevent updates by other transactions till the primary transaction is committed or rolled back.

Thus, if the number of updated records is as anticipated, the data consistency can be ensured.

Exclusive control can be performed by using this characteristic and specifying the conditions to ensure data consistency for WHERE clause at the time of update.

The support status of row lock at the time of update for each database is shown below.

Sr. No.	Database	Version	Default setting lock	Remarks
1.	Oracle	11	Row lock	Increased memory usage due to locking.
2.	PostgreSQL	9	Row lock	There is no maximum limit on the number of rows that can be locked simultaneously since information for the modified rows is not stored in the memory. However, it is necessary to perform VACUUM periodically to write to the table.
3.	DB2	9	Row lock	Increased memory usage due to locking.
4.	MySQL InnoDB	5	Row lock	Increased memory usage due to locking.

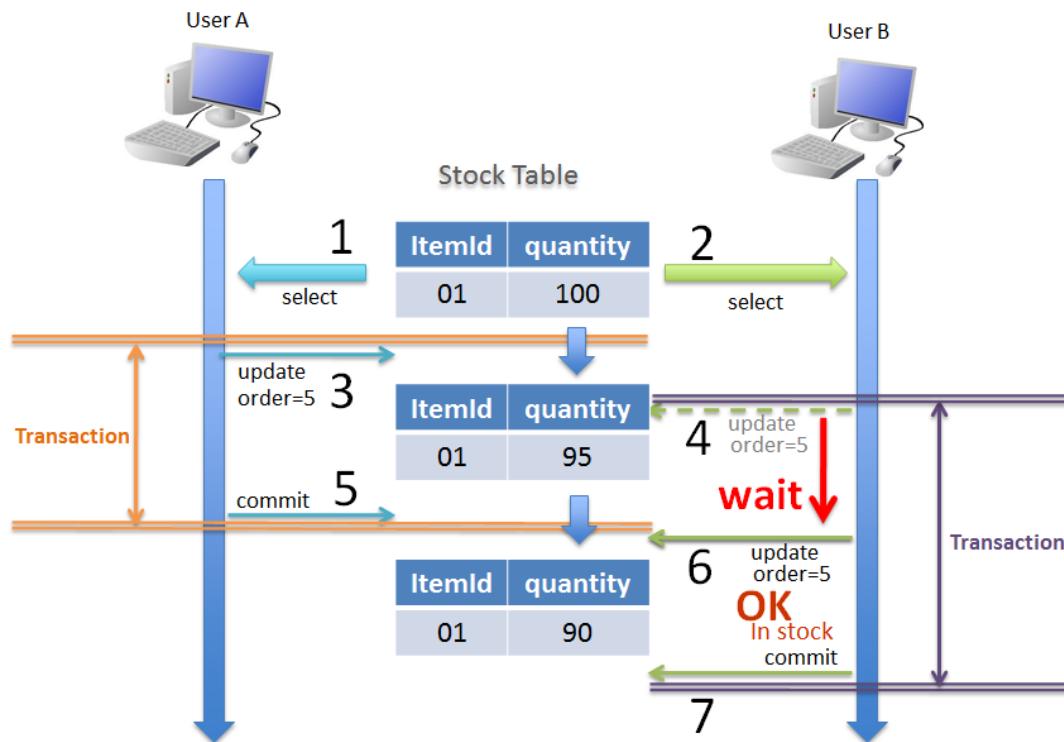
Exclusive control using row lock function of database can be used when it is not necessary to verify the contents updated by other transaction.

For example, it can be used in the purchase process of a shopping site wherein the purchased product quantity is deducted from the records that manage product stock quantity.

Exclusive control is not recommended in the processes used for status management since the earlier status is important in such processes.

See the example below illustrating a specific scenario.

- On a shopping site, both User A and User B are displayed a purchase screen of the same product at the same time. A stock quantity fetched from Stock Table is also displayed at the same time.
- 5 products were purchased at the same time; but since User A clicked “Purchase” button a bit earlier, User A buys the product first and then User B.



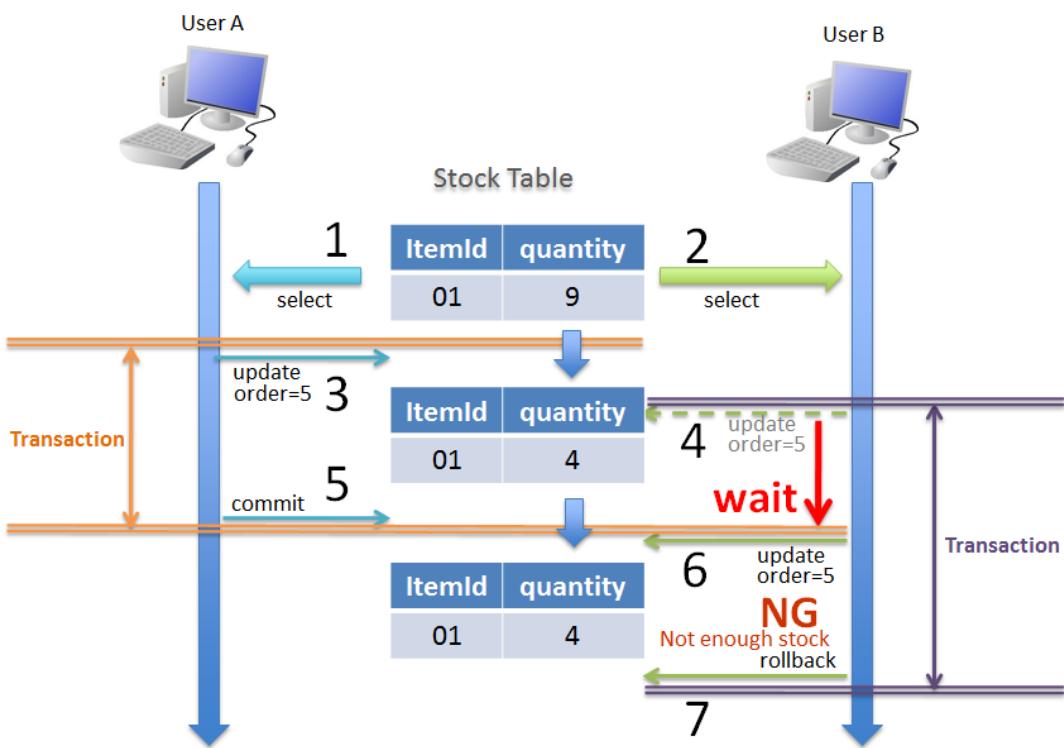
Sr. No.	UserA	UserB	Description
1.	<input checked="" type="radio"/>	-	User A displays a purchase screen of the product. A stock quantity of 100nos. is displayed on the screen. <code>select quantity from Stock where ItemId = '01'</code>
2.	-	<input checked="" type="radio"/>	User B displays a purchase screen of the product. A stock quantity of 100nos. is displayed on the screen. <code>select quantity from Stock where ItemId = '01'</code>
3.	<input checked="" type="radio"/>	-	User A purchases 5 products of ItemId=01. 5nos. are deducted from Stock Table. <code>Update from Stock set quantity = quantity - 5 where ItemId='01' and quantity >= 5</code>
4.	-	<input checked="" type="radio"/>	User B purchases 5 products of ItemId=01. The system tries to deduct 5 products from Stock Table; however since transaction of User A is not yet completed, the purchase process of User B is kept on hold.
5.	<input checked="" type="radio"/>	-	Transaction of user A is committed.
6.	-	<input checked="" type="radio"/>	The transaction of user A is now committed, hence purchase for User B which was kept on hold in step 4 is now resumed. If a stock screen is viewed at this point, the stock quantity is now 95 instead of 100.
518			However, since the balance stock quantity is more than the purchase quantity (in the 5 Architecture in Detail - TERASOLUNA Global Framework above example, 5), 5 is deducted from Stock Table. <code>Update from Stock set quantity = quantity - 5 where ItemId='01' and quantity >= 5</code>

Note: Important

It is important to specify the deduction ("quantity - 5") and update condition ("and quantity >= 5") in SQL.

In the similar scenario as above, if the stock quantity when a product purchase screen is displayed is 9, the stock quantity when the User B resumes update process becomes 4. As it does not satisfy `quantity >= 5` condition, update count becomes 0.

If the update count in the application is 0, purchase is rolled back and User B is prompted for re-execution.

**Note: Important**

It is important to verify the update count in the application and an error should occur if it is different from the expected count and transaction should be rolled back.

When this method is used for locking, the process can be continued depending on conditions even if there is a change in the referred information and data consistency can be ensured by using database function.

Exclusive control using optimistic locking

Optimistic locking is a method for ensuring data consistency wherein the data is not locked for transaction and is updated only after checking whether it is same as fetched data.

When optimistic locking is to be used, a Version column for managing versions is created to determine if the data to be updated is same as fetched data.

Data consistency can be ensured by keeping both the versions at the time of data fetch and data update same as a condition for update.

Note: Version column

This column is used in optimistic locking for managing the update count of a record. It is set to 0 when a record is inserted and then it is incremented with each successful update. The Version column can be also be substituted with the latest update timestamp in place of a number. However, when timestamp is used, uniqueness when processes are executed simultaneously cannot be ensured. Hence, in order to ensure uniqueness, it is necessary to use a number in Version column.

Exclusive control with optimistic locking is used when it is necessary to verify the contents updated by other transaction.

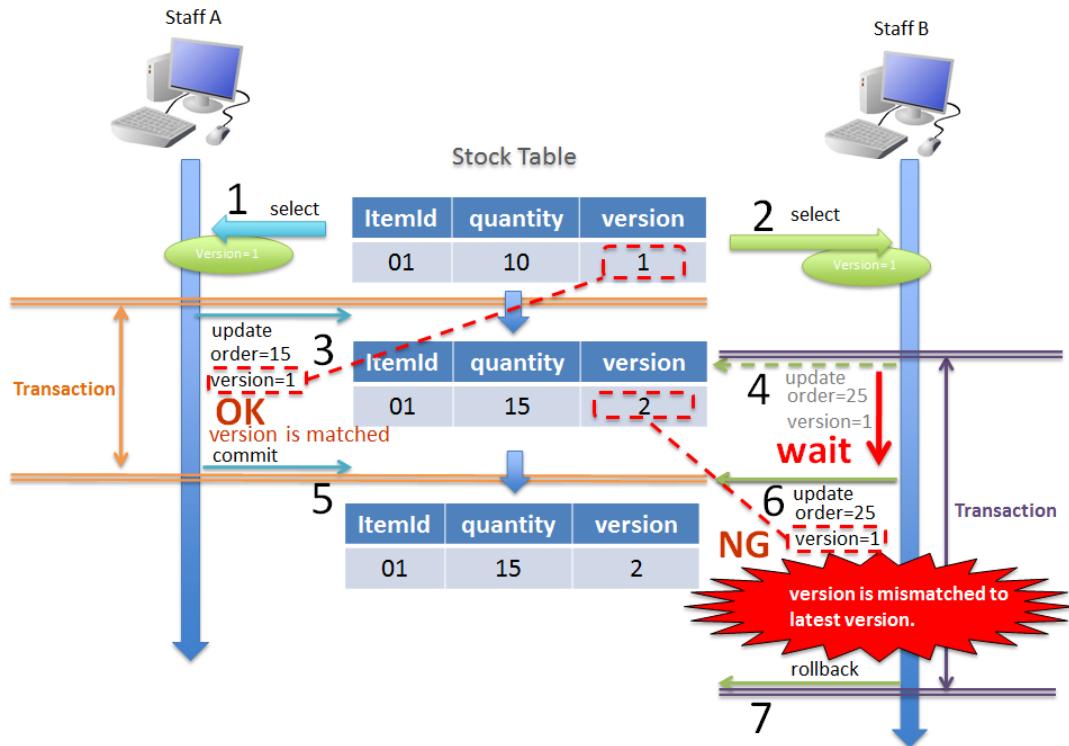
For example, consider a case of workflow application wherein an applicant and an approver perform concurrent operations (withdrawal and approval).

In this case, using exclusive control with optimistic locking, it is possible to notify the applicant and the approver that the operation is yet to be completed since the status before and after the operation are different.

Warning: When an optimistic locking is to be performed, it is not appropriate to update/delete the record by adding conditions other than ID and Version. This is because when the data cannot be updated, it is difficult to determine whether the reason for update failure is version mismatch or condition mismatch. When the conditions for update are different, it is necessary to check whether previous process meets all the conditions.

See the example below illustrating a specific scenario.

- Staff (Staff A, Staff B) who manage the stock quantity of shopping site add the product stock. Let us assume that Staff A adds 5nos. and Staff B adds 15nos.
- A stock management screen is displayed in order to reflect the added stock in the stock management system. The stock quantity being stored in Stock Management System is then displayed.
- Against the displayed stock quantity, the value calculated by summing up the quantity added by the staff is entered in the Update form and the total stock is updated.



Sr. No.	StaffA	StaffB	Description
1.	<input type="radio"/>	-	Staff A displays stock management screen of the product. The stock quantity of 10nos. is displayed on the screen. Version of the referred data is 1.
2.	-	<input type="radio"/>	Staff B displays stock management screen of the product. The stock quantity of 10nos. is displayed on the screen. Version of the referred data is 1.
3.	<input type="radio"/>	-	Staff A adds the stock of 5 nos. against the stock quantity of 10 nos. displayed on the screen and updates stock quantity as 15nos. Version of the referred data is included as an update condition. <code>UPDATE Stock SET quantity = 15, version = version + 1 WHERE itemId = '01' and version = 1</code>
4.	-	<input type="radio"/>	Although Staff B adds 15nos. to the stock quantity of 10 nos. displayed on the screen and attempts to update the stock quantity to 25 nos.; however the transaction is kept on hold since the transaction of Staff A is not completed yet. Version of the referred data is included as an update condition.
5.	<input type="radio"/>	-	Transaction of Staff A is committed. Version becomes 2 at this point.
6.	<input type="radio"/>	-	Update process of Staff B which was kept on hold in step 4 is now resumed since the transaction of Staff A is committed. At this time, since the version of Stock Table data is 2, update result is 0 records. When the update result is 0 records, an exclusive error occurs. <code>UPDATE Stock SET quantity = 25, version = version + 1 WHERE itemId = '01' and version = 1</code>
7.	<input type="radio"/>	-	The transaction of Staff B is rolled back.

Note: Points

Version ("version + 1") should be incremented and update condition ("and version = 1") should be specified in SQL.

Exclusive control using pessimistic locking

Pessimistic locking is a method to lock the data to be updated at the time of fetching so that it is not updated by other transaction.

When pessimistic locking is to be used, lock is obtained for the record to be updated immediately after starting a transaction.

Since the locked record is not updated by other transaction till the transaction is committed or rolled back, data consistency can be ensured.

Table.5.12 RDBMS-wise pessimistic lock acquisition method

Sr. No.	Database	Pessimistic locking method
1.	Oracle	FOR UPDATE
2.	PostgreSQL	FOR UPDATE
3.	DB2	FOR UPDATE WITH
4.	MySQL	FOR UPDATE

Note: About pessimistic locking timeout

At the time of obtaining a pessimistic lock, if a lock is obtained by some other transaction, then the expected behavior is specified as an option in some cases. In case of Oracle,

- Default value is `select for update [wait]`, and it waits till lock is released.
- If set to `select for update nowait`, it immediately gives a resource busy error, if locked by other transaction.
- If set to `select for update wait 5`, it waits for 5 seconds, and gives a resource busy error, if the lock is not released within 5 seconds.

Although there are variations in the functions as per DB, it is necessary to analyze the method to be used when using pessimistic locking.

Note: When using JPA (Hibernate)

Although the method of fetching a pessimistic lock varies for each database, such differences are absorbed by JPA (Hibernate). Refer to [Hibernate Developer Guide](#) for RDBMS that supports Hibernate.

Exclusive control with pessimistic locking is used when it is applicable to any of the 3 cases given below.

1. Data to be updated is managed by dividing it in multiple tables.

When the data to be updated is divided into multiple tables, it is necessary to ensure that there are no updates by other transaction, till the update of each table is completed.

2. Status of the fetched data needs to be checked before performing update.

After completing the checks, it is necessary to ensure that there are no updates by other transaction.

3. Online processing may be executed during batch execution.

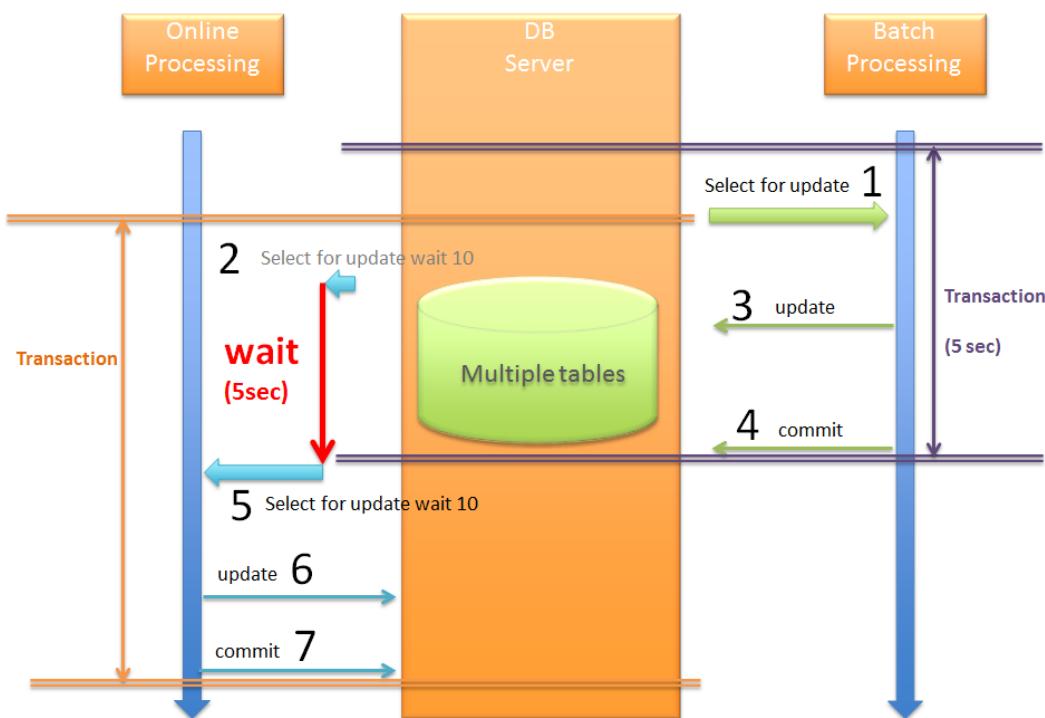
In batch processing, locks are collectively obtained for the data to be updated so as to ensure that exclusive error does not occur in between the execution.

In case of locks which are obtained collectively, the locking period for online processing may be longer.

In such a case, it is advisable to use pessimistic locking by specifying a timeout period.

See the example below illustrating specific scenario.

- Batch processing has already started execution, and data to be updated online is locked by pessimistic locking.
- Timeout period of 10 seconds is specified for the online processing and lock is obtained for the data to be updated.
- Batch processing is terminated after 5 seconds (before timeout).



Sr. No.	Online	Batch	Description
1.	-	<input type="radio"/>	Batch processing obtains the pessimistic lock for the data to be updated in online processing.
2.	<input type="radio"/>	-	Online processing tries to perform pessimistic locking for the data to be updated; however it is kept on hold since the pessimistic locking is performed by the transaction of batch processing. <code>SELECT * FROM Stock WHERE quantity < 5 FOR UPDATE WAIT 10</code>
3.	-	<input type="radio"/>	Batch processing updates data.
4.	-	<input type="radio"/>	Batch processing transaction is committed.
5.	<input type="radio"/>	-	Since the batch processing transaction is committed, online processing is resumed. As the fetched data reflects the update results of batch processing, data inconsistency does not occur.
6.	<input type="radio"/>	-	Online processing updates data.
7.	<input type="radio"/>	-	Online processing transaction is committed.

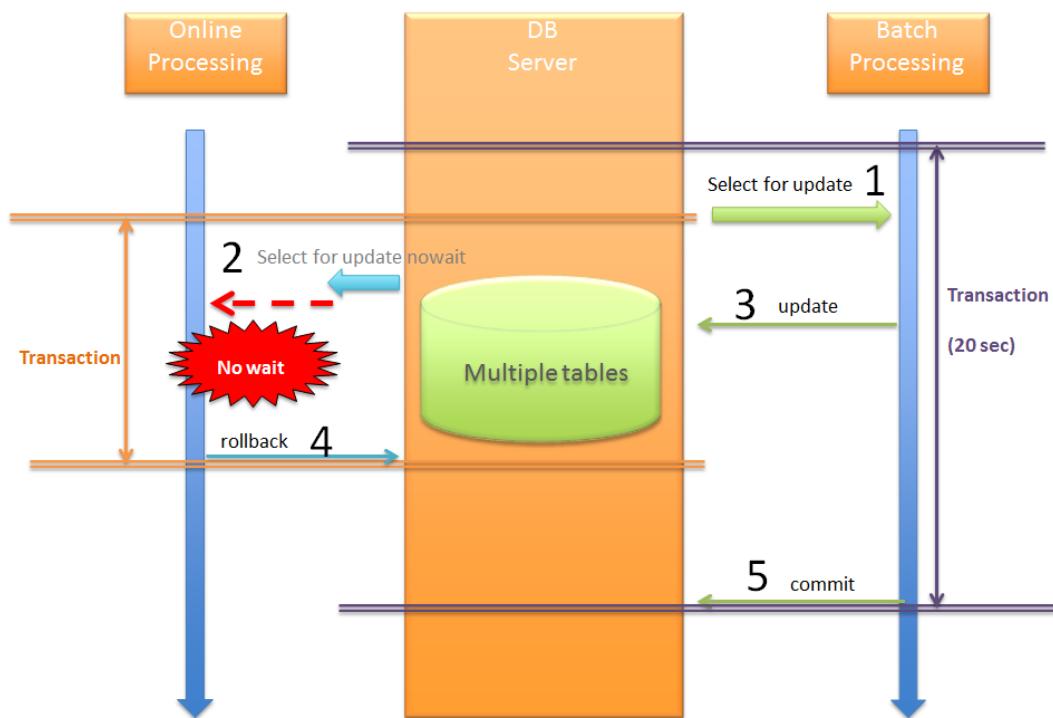
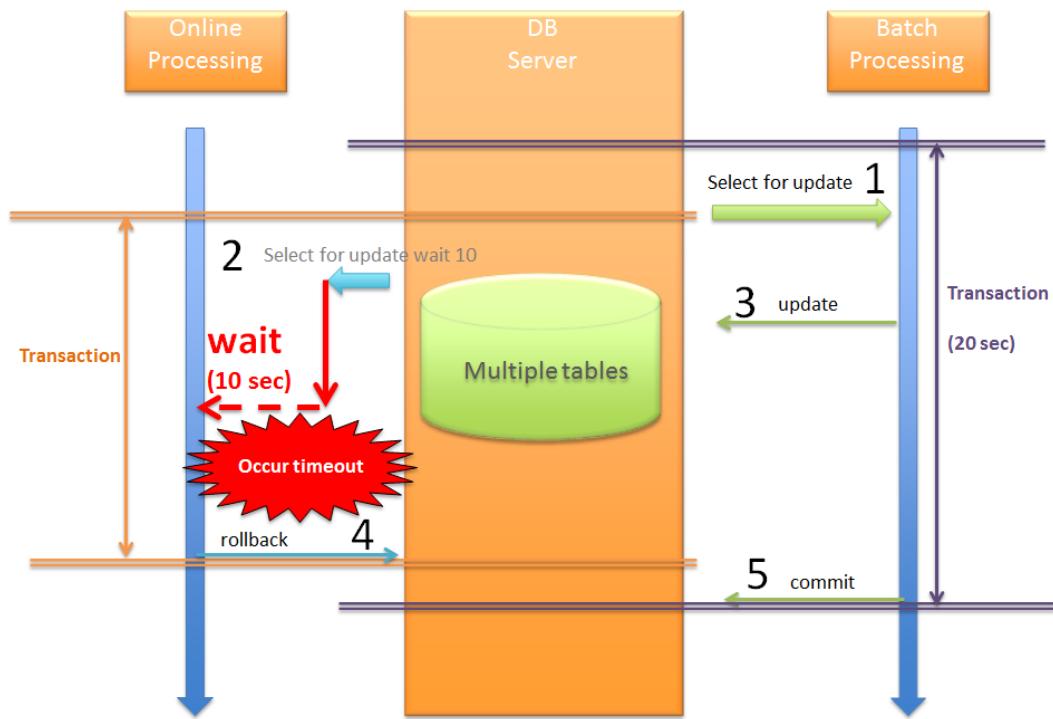
The flow at the time of timeout is given below.

Exclusive error occurs without waiting for the batch processing to end.

The flow given below illustrates a case wherein pessimistic lock is being obtained by other transaction in case of “pessimistic lock no wait” setting.

An exclusive error occurs immediately without waiting for the release of pessimistic lock.

When there is a possibility of conflict between batch processing and online processing and if batch processing is going to take longer time, it is recommended to specify timeout period of pessimistic exclusive locking. The timeout period should be determined based on the online processing requirements.



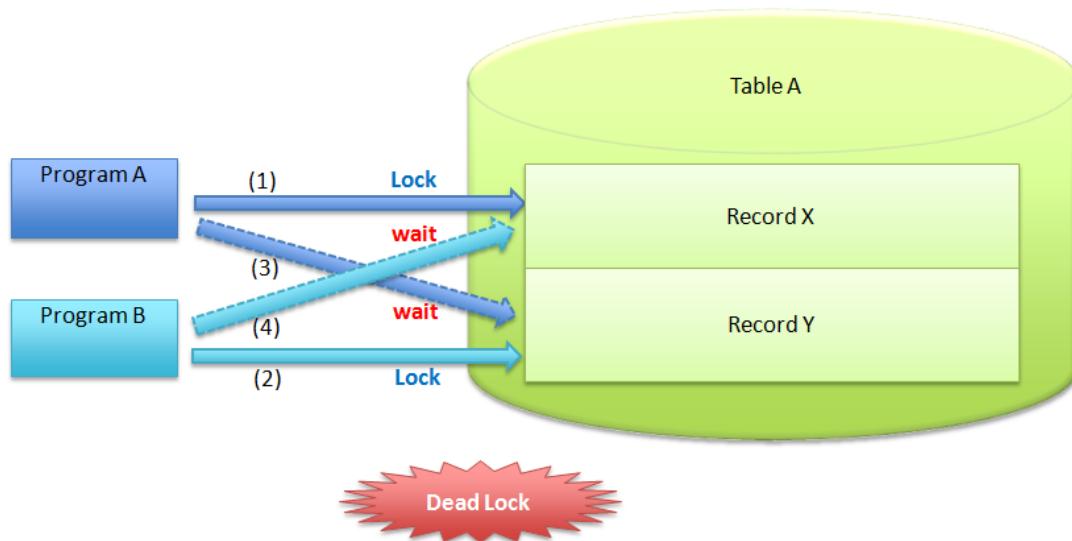
Prevention of deadlock

When using database locks, it should be noted that if multiple records are updated in same transaction, 2 deadlocks shown below are likely to occur.

- *Deadlock in table*
- *Deadlock between tables*

Deadlock in table

This deadlock occurs when the records of the same table are locked by multiple transactions as shown in the flow of (1)-(5) below.



Sr. No.	Program A	Program B	Description
(1)	<input type="radio"/>	-	Program A obtains the lock for Record X.
(2)	<input type="radio"/>	-	Program B obtains the lock for Record Y.
(3)	<input type="radio"/>	-	Program A tries to obtain the lock for Record Y that is locked by the transaction of Program B, however since lock of (2) is not released, the status is changed to “Waiting for release”.
(4)	-	<input type="radio"/>	Program B tries to obtain the lock for Record X that is locked by the transaction of Program A, however since lock of (1) is not released, the status is changed to “Waiting for release”.
(5)	-	-	Since Program A and Program B both have the “Waiting for release” status for each other, it results into a deadlock. When a deadlock occurs, it is detected by the database and error is thrown.

Note: How to resolve a deadlock

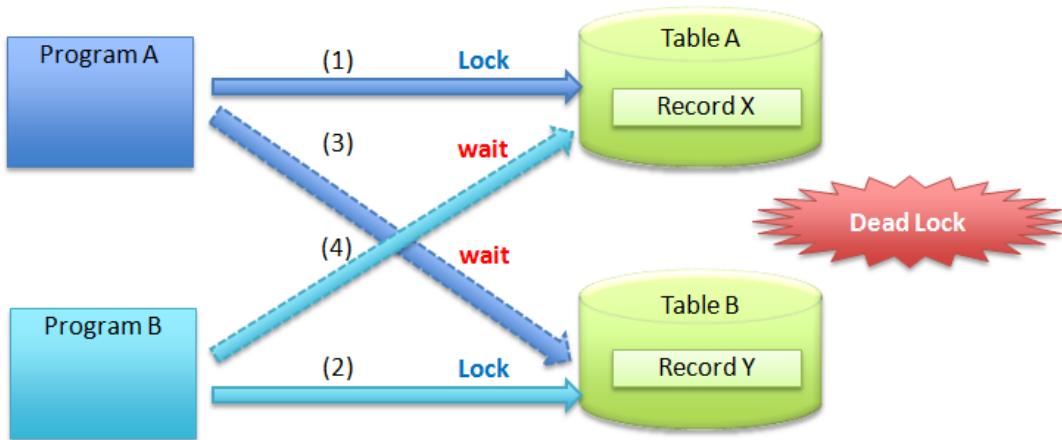
The deadlock can be resolved by timeout or retry; however, it is important to determine the rules for the update sequence of records in the same table. When rows are to be updated one by one, the rules such as updating in ascending order of PK (PRIMARY KEY) should be set.

Let us say if both Program A and Program B follow the rule of starting the update from Record X, the deadlock shown in figure [Deadlock in table](#) above no longer occurs.

Deadlock between tables

This deadlock occurs when records of different tables are locked by multiple transactions as shown in the flow of (1)-(5) below.

The basic concept is same as [Deadlock in table](#).



Sr. No.	Program A	Program B	Description
(1)	<input checked="" type="radio"/>	-	Program A obtains the lock for Record X of Table A.
(2)	<input checked="" type="radio"/>	-	Program B obtains the lock for Record Y of Table B.
(3)	<input checked="" type="radio"/>	-	Program A tries to obtain the lock for Record Y of Table B that is locked by the transaction of Program B, however since lock of (2) is not released, the status is changed to "Waiting for release".
(4)	-	<input checked="" type="radio"/>	Program B tries to obtain the lock of Record X of Table A that is locked by the transaction of Program A, however since lock of (1) is not released, the status is changed to "Waiting for release".
(5)	-	-	Since Program A and Program B both have the "Waiting for release" status for each other, it results into a deadlock. When a deadlock occurs, it is detected by the database and error is thrown.

Note: How to resolve a deadlock

Although deadlocks can be resolved by a timeout or retry; it is important to define rules for update sequence across the tables.

Let us say if both Program A and Program B follow the rule of starting the update from Table A, the deadlock shown in figure [Deadlock between tables](#) above no longer occurs.

Warning: A deadlock might still occur due to sequence of locking records even after adopting either of the methods as a precaution. The rules should be defined for the lock sequence of tables and records.

5.4.2 How to use

How to implement while using JPA (Spring Data JPA)

Row lock function of RDBMS

When exclusive control is to be performed using a row lock function of RDBMS, Query method is added to Repository interface for implementation.

For Query method, refer to *Adding query method* and *Operating the entities of Persistence Layer directly*.

- Repository interface

```
public interface StockRepository extends JpaRepository<Stock, String> {

    @Modifying
    @Query("UPDATE Stock s
        + " SET s.quantity = s.quantity - :quantity"
        + " WHERE s.itemCode = :itemCode"
        + " AND :quantity <= s.quantity") // (1)
    public int decrementQuantity(@Param("itemCode") String itemCode,
                                @Param("quantity") int quantity);

}
```

Sr. No.	Description
(1)	When the stock quantity is equal to or more than the order quantity, JPQL is specified in Query method to reduce stock quantity. Since it is necessary to check the update count, int is specified as the return value of Query method.

- Service

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

int updateCount = stockRepository.decrementQuantity(itemCodeOfOrder, quantityOfOrder); // (2)
if (updateCount == 0) { // (3)
    ResultMessages message = ResultMessages.error();
    message.add(ResultMessage
        .fromText("Not enough stock. Please, change quantity."));
    throw new BusinessException(message); // (4)
}
```

```
update m_stock set quantity=quantity-31
where item_code='ITM0000001' and 31<=quantity -- (5)
```

Sr. No.	Description
(2)	Query method is called.
(3)	Call results of Query method are determined. Since update conditions are not satisfied in case of 0, the stock quantity becomes inadequate.
(4)	The message notifying “No stock” or “Not enough stock” is stored and a business error is generated. The generated error must be handled appropriately in Controller as per the requirements. In the above example, only business rules are checked along with exclusive control; hence when update conditions are not satisfied, it is treated as business error and not exclusive error. For error handling methods, refer to <i>exception-handling-how-to-use-codingpoint-contoller-label</i> .
(5)	SQL that is executed while calling the Query method.

Optimistic locking

In JPA, optimistic locking can be performed by specifying `@javax.persistence.Version` annotation in property for version control.

- Entity

```
@Entity
@Table(name = "m_stock")
public class Stock implements Serializable {

    @Id
    @Column(name = "item_code")
    private String itemCode;

    private int quantity;

    @Version // (1)
    private long version;
```

```
// ...
```

```
}
```

Sr. No.	Description
(1)	@Version annotation is specified in property for version control.

- Service

```
String itemCode = "ITM0000001";
int newQuantity = 30;

Stock stock = stockRepository.findOne(itemCode); // (2)
if (stock == null) {
    ResultMessages messages = ResultMessages.error().add(ResultMessage
        .fromText("Stock not found. itemCode : " + itemCode));
    throw new ResourceNotFoundException(messages);
}

stock.setQuantity(newQuantity); // (3)

stockRepository.flush(); // (4)

update m_stock set quantity=30, version=7
    where item_code='ITM0000001' and version=6 -- ( 5)
```

Sr. No.	Description
(2)	findOne method of Repository interface is called and entity is fetched.
(3)	Value to be updated is specified for the entity fetched in step (2).
(4)	The modification details of (3) are reflected in the persistence layer (DB). This process is usually not required since it is performed for the description purpose. Normally, it is reflected automatically when the transaction is committed. In the above example, when the version held by the entity fetched in step (2) and the version stored in persistence layer (DB) do not match, optimistic locking error (<code>org.springframework.dao.OptimisticLockingFailureException</code>) occurs.
(5)	SQL that is executed while reflecting to persistence layer (DB) of step (4).

It is important to note the following points while performing optimistic locking for long transactions.

Warning: It is not sufficient to simply assign `@Version` annotation for optimistic locking which is to be performed for long transactions. When optimistic locking is to be performed for long transactions, version check should also be carried out while fetching the data to be updated, in addition to the check at the time of update performed using JPA function.

An implementation example is given below.

- Service

```

long version = 12;
String itemCode = "ITM0000001";
int newQuantity = 30;

Stock stock = stockRepository.findOne(itemCode); // (1)
if (stock == null || stock.getVersion() != version) { // (2)
    throw new ObjectOptimisticLockingFailureException(Stock.class, itemCode); // (3)
}

stock.setQuantity(newQuantity);

stockRepository.flush();

```

Sr. No.	Description
(1)	An entity is fetched from persistence layer (DB).
(2)	Version of the entity fetched by a different database transaction in advance is compared with the latest version of persistence layer (DB) fetched in step (1). If versions match, optimistic locking which uses @Version annotation becomes valid in subsequent processes.
(3)	If the versions are different, optimistic locking error (<code>org.springframework.dao.ObjectOptimisticLockingFailureException</code>) is generated.

Warning: Setting a value in property for Version control

Entity fetched using Repository interface is called “Managed entity”.

For “Managed entity”, it should be noted that a value cannot be set in a process for the property for Version control.

Even if a process as shown below is carried out, the value of version that is set to “Managed entity” is not reflected. Hence it is not used to fetch an optimistic lock. The version when a value is fetched using findOne method is used for optimistic locking.

```
long version = 12;
String itemCode = "ITM0000001";
int newQuantity = 30;

Stock stock = stockRepository.findOne(itemCode);
if (stock == null) {
    ResultMessages messages = ResultMessages.error().add(ResultMessage
        .fromText("Stock not found. itemCode : " + itemCode));
    throw new ResourceNotFoundException(messages);
}
stock.setVersion(version); // Invalid Processing
stock.setQuantity(newQuantity);

stockRepository.flush();
```

For example, even if the value of the version sent from the screen is overwritten, it is not reflected in entity. Hence the exclusive control can no longer be appropriately performed.

Note: Standardization of optimistic locking for long transactions

When optimistic locking for long transactions becomes necessary in multiple processes, it is desirable to standardize the processes of (1) ~ (3) described above. For standardization method, refer to [How to add custom method](#)

It is important to note the following points when both row lock function of RDBMS and optimistic locking function are to be used.

Warning: **Version must always be updated** in Query method that uses row lock function of RDBMS in case of applications wherein a process performing exclusive control using a row lock function of RDBMS and a process performing exclusive control using optimistic locking function co-exist, for the same data. If version is not updated in Query method that performs exclusive control using row lock function of RDBMS, the contents updated by Query method may be overwritten by the process of a different transaction. Hence, the exclusive control is not performed properly.

An implementation example is given below.

- Repository interface

```
public interface StockRepository extends JpaRepository<Stock, String> {

    @Modifying
    @Query("UPDATE Stock s SET s.quantity = s.quantity - :quantity"
        + ", s.version = s.version + 1" // (1)
        + " WHERE s.itemCode = :itemCode"
        + " AND :quantity <= s.quantity")
    public int decrementQuantity(@Param("itemCode") String itemCode,
        @Param("quantity") int quantity);

}
```

Sr. No.	Description
(1)	Version needs to be updated (<code>s.version = s.version + 1</code>).

Pessimistic locking

In Spring Data JPA, the pessimistic lock can be performed by specifying `@org.springframework.data.jpa.repository.Lock` annotation.

- Repository interface

```
public interface StockRepository extends JpaRepository<Stock, String> {

    @Lock(LockModeType.PESSIMISTIC_WRITE) // (1)
    @Query("SELECT s FROM Stock s WHERE s.itemCode = :itemCode")
    Stock findOneForUpdate(@Param("itemCode") String itemCode);

}
```

```
-- (2)
SELECT
    stock0_.item_code AS item1_5_
 ,stock0_.quantity AS quantity2_5_
 ,stock0_.version AS version3_5_
FROM
    m_stock stock0_
WHERE
    stock0_.item_code = 'ITM0000001'
FOR UPDATE;
```

Sr. No.	Description
(1)	@Lock annotation is specified in Query method.
(2)	Executed SQL. In the above example, the SQL executed while using PostgreSQL is given.

The types of pessimistic locking that can be specified using @Lock annotation are as given below.

Sr. No.	LockModeType	Description	Issued SQL
1.	PESSIMISTIC_READ	<p>Pessimistic lock for read is obtained. It acts as a shared lock rather than an exclusive lock depending on the database.</p> <p>Lock is released at the time of commit or rollback</p>	select ... for update / select ... for share
2.	PESSIMISTIC_WRITE	<p>Pessimistic lock for update is obtained and an exclusive lock is applied.</p> <p>In case of exclusive lock, if a lock has already been applied, the entity is fetched after waiting for the lock to be released.</p> <p>Lock is released at the time of commit or rollback</p>	select ... for update
3.	PESSIMISTIC_FORCE_INCREMENT	<p>Exclusive lock is applied to the target data when entity is fetched. Version is forcibly updated immediately after fetching the entity.</p> <p>Lock is released at the time of commit or rollback</p>	select ... for update + update

Note: Lock timeout period

Timeout period can be specified by specifying "javax.persistence.lock.timeout" as JPA (EntityManager) settings or Query hint.

There are 2 methods for specifying the locking timeout period: 1. Method wherein it is specified for the entire process 2. Method wherein it is specified for each Query.

Method wherein it is specified for the entire process is as follows:

- xxx-infra.xml

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="packagesToScan" value="xxxxxxxx.yyyyyy.zzzzz.domain.model" />
    <property name="dataSource" ref="dataSource" />
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
    <property name="jpaPropertyMap">
```

```
<util:map>
    <!-- ... -->
    <entry key="javax.persistence.lock.timeout" value="1000" /> <!-- (1) -->
</util:map>
</property>
</bean>
```

Sr. No.	Description
(1)	The timeout is specified in milliseconds. If 1000 is specified, it equals 1 second.

Note: nowait support

When 0 is specified for Oracle and PostgreSQL, nowait is added, and when locked by another transaction, an exclusive error occurs without waiting for release of lock.

Warning: Restrictions of PostgreSQL

Although nowait can be specified in PostgreSQL, it is not possible to specify waiting time. Therefore, the measures such as separately providing timeout for Query etc. should be implemented.

Method wherein it is specified for each Query is as follows:

- Repository interface

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
@QueryHints(@QueryHint(name = "javax.persistence.lock.timeout", value = "2000")) // (1)
@Query("SELECT s FROM Stock s WHERE s.itemCode = :itemCode")
Stock findOneForUpdate(@Param("itemCode") String itemCode);
```

Sr. No.	Description
(1)	The timeout is specified in milliseconds. If 2000 is specified, it equals 2 seconds. All the specified values are overwritten.

Implementation method when using Mybatis

Row lock function of RDBMS

When exclusive control is performed using the row lock function of RDBMS, it is implemented by adding SQL definition to sqlmap file.

- xxx-sqlmap.xml

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

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

</update>
```

Sr. No.	Description
(1)	When the stock quantity is equal to or more than the order quantity, an SQL to reduce the stock quantity is specified in sqlmap file.

- Repository(RepositoryImpl)

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

```
public class StockRepositoryImpl implements StockRepository {

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

}
```

Sr. No.	Description
(2)	Method is added to Repository.
(3)	Parameters necessary for SQL execution are generated and SQL is called.

Note: When Repository is not created

When Repository is not created, the process described above is implemented in Service.

- Service

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

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

Sr. No.	Description
(4)	Query method is called.
(5)	Call results of Query method are determined. Since update conditions are not satisfied in case of 0, the stock quantity becomes inadequate.
(6)	The message notifying “No stock” or “Not enough stock” is stored and a business error is generated. The generated error must be handled appropriately in Controller as per the requirements. In the above example, only business rules are checked along with exclusive control; hence when update conditions are not satisfied, it is treated as business error and not exclusive error. For error handling methods, refer to <i>exception-handling-how-to-use-codingpoint-contoller-label</i> .

Optimistic locking

Mybatis does not provide a library with a mechanism to perform optimistic locking.

Therefore, ‘version’ needs to be specified in SQL to perform optimistic locking.

- Entity

```
public class Stock implements Serializable {

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

    // ...
}
```

Sr. No.	Description
(1)	A property for version control is created.

- xxx-sqlmap.xml

```

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

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

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

```

Sr. No.	Description
(2)	Value set in column for version control is fetched using SQL that retrieves the data to be updated.
(3)	Version is incremented at the time of update.
(4)	“Version should match” is added as an update condition.

- Repository(RepositoryImpl)

```

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

```

```

public class StockRepositoryImpl implements StockRepository {

    public Stock findOne(String itemCode) {

```

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

    public Stock save(Stock stock) {
        if(exists(stock.getItemCode())) {
            int updateCount = updateDAO.execute("stock.update", stock);
            if(updateCount == 0) {
                throw new ObjectOptimisticLockingFailureException(Stock.class, itemCode); // (1)
            }
        } else {
            updateDAO.execute("stock.insert", stock);
        }
        return stock;
    }

}
```

Sr. No.	Description
(5)	When update result is 0, since it will be treated as update by other transaction; optimistic locking error (<code>org.springframework.orm.ObjectOptimisticLockingFailureException</code>) is generated.

Note: When Repository is not created

When Repository is not created, the process described above should be implemented in Service.

- Service

```
String itemCode = "ITM0000001";
int newQuantity = 30;

Stock stock = stockRepository.findOne(itemCode); // (2)
if (stock == null) {
    ResultMessages messages = ResultMessages.error().add(ResultMessage
        .fromText("Stock not found. itemCode : " + itemCode));
    throw new ResourceNotFoundException(messages);
}

stock.setQuantity(newQuantity); // (3)

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

Sr. No.	Description
(2)	findOne method of Repository interface is called and entity is fetched.
(3)	Value to be updated is specified for the entity fetched in step (2).
(4)	The modification details of (3) are reflected in persistence layer (DB).

It is important to note the following points while performing optimistic locking for long transactions.

Warning: When performing optimistic locking for long transactions, version check in addition to the check at the time of update, should also be carried out while fetching the data.

An implementation example is given below.

- Service

```
long version = 12;
String itemCode = "ITM0000001";
int newQuantity = 30;

Stock stock = stockRepository.findOne(itemCode); // (1)
if (stock == null || stock.getVersion() != version) { // (2)
    throw new ObjectOptimisticLockingFailureException(Stock.class, itemCode); // (3)
}

stock.setQuantity(newQuantity);

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

Sr. No.	Description
(1)	An entity is fetched from persistence layer (DB).
(2)	Version of the entity fetched by a different database transaction in advance is compared with the latest version of persistence layer (DB) fetched in step (1).
(3)	If the versions are different, optimistic locking error (<code>org.springframework.dao.ObjectOptimisticLockingFailureException</code>) is generated.

It is important to note the following points when using both row lock function of RDBMS and optimistic locking function.

Warning: **Version must always be updated **in SQL that uses the row lock function of RDBMS in case of applications wherein a process performing exclusive control by using a row lock function of RDBMS and a process performing exclusive control by using an optimistic lock function co-exist, for the same data. If Version is not updated in row SQL that performs exclusive control using row lock function of RDBMS, the contents updated by Query method may be overwritten by the process of a different transaction. Hence, the exclusive control is not performed properly.

An implementation example is given below.

- `xxx-sqlmap.xml`

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

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

</update>
```

Sr. No.	Description
(1)	Version needs to be updated (<code>version = version + 1</code>).

Pessimistic locking

Mybatis does not provide a library with a mechanism to perform pessimistic locking.

Therefore, a keyword for obtaining lock needs to be specified in SQL to perform pessimistic locking.

- xxx-sqlmap.xml

```
<select id="findOneForUpdate" parameterClass="java.lang.String" resultMap="stockResultMap">
    SELECT * FROM m_stock
    WHERE item_code = #itemCode#
    FOR UPDATE <!-- (1) -->
</select>
```

Sr. No.	Description
(1)	A keyword for obtaining pessimistic lock is specified for the SQL which requires pessimistic locking. Keyword varies with the database.

How to handle an exclusive error

Error handling in case of optimistic locking failure

When optimistic locking fails, `org.springframework.dao.OptimisticLockingFailureException` occurs; hence it is necessary to handle it appropriately in Controller.

The handling method varies with the operation specifications of application when optimistic locking error occurs.

When there is no need to change the operation at request level, it is handled by using `@ExceptionHandler` annotation.

```
@ExceptionHandler(OptimisticLockingFailureException.class) // (1)
public String handleOptimisticLockingFailureException(
    OptimisticLockingFailureException e) {
    // (2)
    ExtendedModelMap modelMap = new ExtendedModelMap();
    ResultMessages resultMessages = ResultMessages.warn();
    resultMessages.add(ResultMessage.fromText("Other user updated!!"));
    modelMap.addAttribute(setUpForm());
    String viewName = top(modelMap);
```

```
        return new ModelAndView(viewName, modelMap);
    }
```

Sr. No.	Description
(1)	OptimisticLockingFailureException.class is specified in the value attribute of @ExceptionHandler annotation.
(2)	Error handling is performed. The message to notify error and information required for screen display (form or other model) are generated and ModelAndView that specifies a destination is returned. For details of error handling, refer to <i>exception-handling-how-to-use-codingpoint-contoller-usecase-label</i> .

If there is a need to change the operation at request level, it is handled by using try – catch in the processing method of Controller.

```
@RequestMapping(value = "{itemId}/update", method = RequestMethod.POST)
public String update(StockForm form, Model model, RedirectAttributes attributes) {

    // ...

    try {
        stockService.update(...);
    } catch (OptimisticLockingFailureException e) { // (1)
        // (2)
        ResultMessages resultMessages = ResultMessages.warn();
        resultMessages.add(ResultMessage.fromText("Other user updated!!"));
        model.addAttribute(resultMessages);
        return updateRedo(modelMap);
    }

    // ...
}
```

Sr. No.	Description
(1)	OptimisticLockingFailureException is caught.
(2)	Error handling is performed. The message to notify error and information required for screen display (form or other model) are generated and a destination view name is returned. For details of error handling, refer to <i>exception-handling-how-to-use-codingpoint-contoller-usecase-label</i> .

Error handling in case of pessimistic locking failure

When pessimistic locking fails, `org.springframework.dao.PessimisticLockingFailureException` occurs; hence it is necessary to handle it appropriately in Controller.

The handling method varies with the operation specifications of the application when pessimistic locking error occurs.

If there is no need to change the operation at request level, it is handled using `@ExceptionHandler` annotation.

```
@ExceptionHandler(PessimisticLockingFailureException.class) // (1)
public String handlePessimisticLockingFailureException(
    PessimisticLockingFailureException e) {
    // (2)
    ExtendedModelMap modelMap = new ExtendedModelMap();
    ResultMessages resultMessages = ResultMessages.warn();
    resultMessages.add(ResultMessage.fromText("Other user updated!!"));
    modelMap.addAttribute(setUpForm());
    String viewName = top(modelMap);
    return new ModelAndView(viewName, modelMap);
}
```

SR. No.	Description
(1)	PessimisticLockingFailureException .class is specified in value attribute of @ExceptionHandler annotation.
(2)	Error handling is performed. The message to notify error and information required for screen display (form or other model) are generated and ModelAndView that specifies a destination is returned. For details of error handling, refer to <i>exception-handling-how-to-use-codingpoint-contoller-usecase-label</i> .

If there is need to change the operation at request level, it is handled using try – catch in the processing method of Controller.

```
@RequestMapping(value = "{itemId}/update", method = RequestMethod.POST)
public String update(StockForm form, Model model, RedirectAttributes attributes) {

    // ...

    try {
        stockService.update(...);
    } catch (PessimisticLockingFailureException e) { // (1)
        // (2)
        ResultMessages resultMessages = ResultMessages.warn();
        resultMessages.add(ResultMessage.fromText("Other user updated!!"));
        model.addAttribute(resultMessages);
        return updateRedo(modelMap);
    }

    // ...
}

}
```

Sr. No.	Description
(1)	PessimisticLockingFailureException is caught.
(2)	Error handling is performed. The message to notify error and information required for screen display (form or other model) are generated and destination view name is returned. For details of error handling, refer to <i>exception-handling-how-to-use-codingpoint-contoller-usecase-label</i> .

Todo

It has been found that an unexpected error occurs if JPA (Hibernate) is used.

- When pessimistic locking fails, child class of `org.springframework.dao.UncategorizedDataAccessException` is generated instead of `PessimisticLockingFailureException`.

`UncategorizedDataAccessException` generated at the time of pessimistic 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 an exception notifying the occurrence of pessimistic locking error is saved as 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`
-

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 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)	<p>Assign <code>javax.validation.constraints.NotNull</code> indicating that the target field is not null.</p> <p>In Spring MVC, when form is sent with input fields left blank, empty string instead of null binds to form object by default.</p> <p>This <code>@NotNull</code> checks that name exists as request parameter.</p>
(2)	<p>Assign <code>javax.validation.constraints.Size</code> indicating that the string length (or collection size) of the target field is within the specified size range.</p> <p>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.</p>
(3)	<p>Assign <code>org.hibernate.validator.constraints.Email</code> indicating that the target field is in RFC2822-compliant E-mail format.</p> <p>When E-mail format requirements are flexible than RFC2822-compliant constraints, regular expression should be specified using <code>javax.validation.constraints.Pattern</code> instead of <code>@Email</code>.</p>
(4)	<p>When form is sent without entering any number in input field, <code>null</code> binds to form object so <code>@NotNull</code> indicates mandatory input of age.</p>
(5)	<p>Assign <code>javax.validation.constraints.Min</code> indicating that the target field value must be greater than the specified value.</p>
(6)	<p>Assign <code>javax.validation.constraints.Max</code> indicating that the target field value must be less than the specified value.</p>

Tip: Refer to [Bean Validation 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) */ result) {
        if (result.hasErrors()) { // (4)
            return "user/createForm";
        }
        return "user/createConfirm";
    }

    @RequestMapping(value = "create", method = RequestMethod.POST)
    public String create(@Validated UserForm form, BindingResult result) { // (5)
        if (result.hasErrors()) {
            return "user/createForm";
        }
        // omitted business logic
        return "redirect:/user/create?complete";
    }

    @RequestMapping(value = "create", method = RequestMethod.GET, params = "complete")
    public String createComplete() {

```

```
    return "user/createComplete";
}
}
```

S.No.	Description
(1)	Display “New user registration” form screen.
(2)	Assign <code>org.springframework.validation.annotation.Validated</code> to the form class argument to perform input validation.
(3)	Add <code>org.springframework.validation.BindingResult</code> that stores check result of input validation performed in step (2). This <code>BindingResult</code> must be specified immediately after the form argument. When not specified immediately, <code>org.springframework.validation.BindException</code> is thrown.
(4)	Use <code>BindingResult.hasErrors()</code> method to determine the check result of step (2). When the result of <code>hasErrors()</code> is <code>true</code> , 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 submission 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 recommended 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.

Name:	
Email:	
Age:	
<input type="button" value="Confirm"/>	

Error message is displayed as follows if this form is sent with all the input fields left blank.

Name:		size must be between 1 and 20
Email:		size must be between 1 and 50
Age:		may not be null
<input type="button" value="Confirm"/>		

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.

Name:

Email: not a well-formed email address

Age: must be less than or equal to 200

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.

Name: size must be between 1 and 20

Email: not a well-formed email address
size must be between 1 and 50

Age: -1 must be greater than or equal to 0

```
<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)	<p>By specifying * in path attribute of <code><form:errors></code> in <code><form:form></code> tag, all error messages related to Model specified in <code>modelAttribute</code> attribute of <code><form:form></code> can be output.</p> <p>Tag name including these error messages can be specified in <code>element</code> attribute. By default, it is <code>span</code>. However,</p> <p>specify <code>div</code> to output error message list as block element.</p> <p>Specify CSS class in <code>cssClass</code> attribute.</p>

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;  
}
```

size must be between 1 and 50
may not be null
size must be between 1 and 20

Name:

Email:

Age:

Confirm

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.

About how to define error message, refer to “*Definition of error messages*”.

Note: Error messages are output in random order by default. Order can be controlled using `@GroupSequence` annotation However, cost is high.

If error messages are displayed in a list,

- Field name must be included in the error message.
- Order of the error 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.

Coupon Code:

Receiver

Name:

Postcode:

Address:

Sender

Name:

Postcode:

Address:

Coupon Code:

Receiver

Name: size must be between 1 and 50

Postcode: size must be between 1 and 10

Address: size must be between 1 and 100

Sender

Name: size must be between 1 and 50

Postcode: size must be between 1 and 10

Address: size must be between 1 and 100

```
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="${pageContext.request.contextPath}/resources/vendor/js/jquery-1.10.2.min.js"></script>
<script type="text/javascript"
    src="${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="text">\
    <label for="addresses' + number + '.postcode">Postcode:</label>\
    <input id="addresses' + number + '.postcode" name="addresses[' + number + '].postcode" type="text">\
    <label for="addresses' + number + '.address">Address:</label>\
    <input id="addresses' + number + '.address" name="addresses[' + number + '].address" type="text">\
    <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.

Name:

Email:

Age:

Address1

Name:

Postcode:

Address:

Add 2 address forms by clicking “Add address” button twice.

Name:

Email:

Age:

Address1

Name:

Postcode:

Address:

Address2

Name:

Postcode:

Address:

Address3

Name:

Postcode:

Address:

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.

Name: size must be between 1 and 20
Email: size must be between 1 and 50
Age: may not be null

Address1

Name: size must be between 1 and 50
Postcode: size must be between 1 and 10
Address: size must be between 1 and 100

Address2

Name: size must be between 1 and 50
Postcode: size must be between 1 and 10
Address: size must be between 1 and 100

Address3

Name: size must be between 1 and 50
Postcode: size must be between 1 and 10
Address: size must be between 1 and 100

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
		NG	must be greater than or equal to 20
		NG	must be greater than or equal to 21
	cn	OK	
		NG	must be greater than or equal to 20
		NG	must be greater than or equal to 21
	cn	OK	
		OK	
		NG	must be greater than or equal to 21
21	cn	OK	
		OK	
		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 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 <code>country</code> does not have a value, the request is mapped to a method in which Default group is specified in <code>@Validated</code> annotation.
(2)	When the field <code>country</code> has a value, the request is mapped to a method in which Default group is not included in <code>@Validated</code> 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) {
        // (3)
        Class<?> validationGroup = Default.class;
        // logic to determine validation group
        // if (xxx) {
        //     validationGroup = Xxx.class;
        // }
        smartValidator.validate(form, result, validationGroup); // (4)
        if (result.hasErrors()) {
    
```

```
        return "user/createForm";
    }
    return "user/createConfirm";
}

}
```

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)	Determine validation group. Logic to determine validation group recommend to delegate to Helper class and keep Controller's source code with simple state.
(4)	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	<p>It is easy to create input validation for a particular class.</p> <p>It is inconvenient to use in Controller.</p>	Input validation implementation of unique business requirements depending on specific form
Bean Validation	<p>Creation of input validation is not as easy as Spring Validator.</p> <p>It is easy to use in Controller.</p>	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	<p>Mandatory input</p> <p>8 or more characters</p> <p>Must be same as confirmPassword</p>	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) {

        if (errors.hasFieldErrors("password")) { // (3)
            return;
        }

        PasswordResetForm form = (PasswordResetForm) target;
        String password = form.getPassword();
        String confirmPassword = form.getConfirmPassword();

        if (!password.equals(confirmPassword)) { // (4)
            errors.rejectValue(/* (5) */ "password",
                /* (6) */ "PasswordEqualsValidator.passwordResetForm.password",
                /* (7) */ "password and confirm password must be same.");
        }
    }
}
```

```

    }
}
}
}
```

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)	If occurred error at the target fields during a single item check, does not perform correlation item check by in this Validator. If it is necessary to perform the correlation item check always, this logic is unnecessary.
(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 <i>Messages to be defined in application-messages.properties</i> for message 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 <code>@InitBinder</code> annotation, add Validators using <code>WebDataBinder.addValidators</code> method. By this, the added Validator is called when validation is executed with the use of <code>@Validated</code> 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:label>
<form:password path="confirmPassword"
    cssErrorClass="error-input" />
<form:errors path="confirmPassword" cssClass="error-messages" />
<br>
<form:button>Reset</form:button>
</form:form>
</body>
</html>
```

Error message as shown below is displayed when form is sent by entering different values in password field and confirmPassword fields.

Password: password and confirm password must be same.
Password (Confirm):

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).
About default rule of Spring, refer to “[DefaultMessageCodesResolver](#) \ominus JavaDoc”.
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

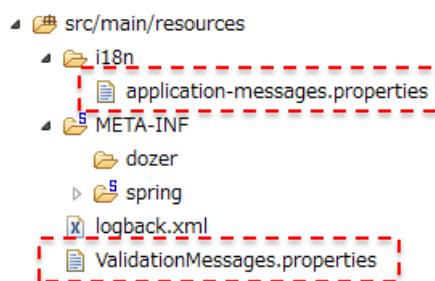
Basically, it is recommended to define error messages in properties file.

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.

Name: size is not in the range 1 through 20.
 Email: size is not in the range 1 through 50.
 Age: is required.

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} through {max}.
javax.validation.constraints.Min.message="{0}" can not be less than {value}.
javax.validation.constraints.Max.message="{0}" can not be greater than {value}.
org.hibernate.validator.constraints.Email.message="{0}" is an invalid e-mail address.

```

Error message is changed as follows.

Name: The size of "name" is not in the range 1 through 20.
Email: The size of "email" is not in the range 1 through 50.
Age: "age" is required.

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.

Name: The size of "Name" is not in the range 1 through 20.
Email: The size of "Email" is not in the range 1 through 50.
Age: "Age" is required.

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}.\nNotNull.userForm.email="{0}" is compulsory.\nSize.userForm.age=The size of "{0}" must be between {2} and {1}.\n# filed names\nname=Name\nemail=Email\nage=Age
```

Value of attributes of the annotation gets inserted after {1} onwards.

Error messages are changed as follows.

Name: The size of "Name" is not in the range 1 through 20.
Email: The size of "Email" must be between 1 and 50.
Age: "Age" is compulsory.

Note: There are other formats as 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;\n\nimport java.lang.annotation.Documented;\nimport java.lang.annotation.Retention;\nimport java.lang.annotation.Target;\nimport javax.validation.Constraint;\nimport javax.validation.Payload;\nimport static java.lang.annotation.ElementType.ANNOTATION_TYPE;\nimport 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 alphanumeric 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 NotNegative {
    String message() default "{com.example.common.validation.NotNegative.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    public @interface List {
```

```
        NotNegative[] 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 document](#) for 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_Number
    static boolean isISBN13Valid(String isbn) {
        if (isbn.length() != 13) {
            return false;
        }
        int check = 0;
        try {
            for (int i = 0; i < 12; i += 2) {
                check += Integer.parseInt(isbn.substring(i, i + 1));
            }
            for (int i = 1; i < 12; i += 2) {
                check += Integer.parseInt(isbn.substring(i, i + 1)) * 3;
            }
            check += Integer.parseInt(isbn.substring(12));
        } catch (NumberFormatException e) {
            return false;
        }
        return check % 10 == 0;
    }
}
```

S.No.	Description
(1)	Specify target annotation and field type in generics parameter.
(2)	Implement initialization process in <code>initialize</code> method.
(3)	Implement input validation in <code>isValid</code> method.
(4)	Consider input value as correct in case of <code>null</code> .
(5)	Check ISBN-13 format.

Tip: Example of *Bean Validation of file upload* 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 <code>org.springframework.beans.BeanWrapper</code> which is very convenient to access JavaBean properties.
(2)	Acquire property value from the form object through <code>BeanWrapper</code> .
(3)	Disable the creation of default <code>ConstraintViolation</code> object.
(4)	Create independent <code>ConstraintViolation</code> object. Define message to be output in <code>ConstraintValidatorContext.buildConstraintViolationWithTemplate</code> . Specify field name to output error message in <code>ConstraintViolationBuilder.addNode</code> . Refer to JavaDoc for 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)	<p>Assign @Confirm annotation at class level.</p> <p>By this, form object is passed as an argument to ConstraintValidator.isValid.</p>

- 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 <code>isValid</code> 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 Bean Validation.

Refer to http://docs.jboss.org/hibernate/validator/4.3/reference/en-US/html_single/#section-builtin-constraints for the annotation list that can be used for validation.

Bean Validation check rules

Bean Validation standard annotations are shown below.

Refer to Chapter 6 [Bean Validation specification](#) for details.

Annotation(javax.validation.*)	Target type	Application	Usage example
@NotNull	Arbitrary	Validates that the target field is not null.	<pre>@NotNull private String id;</pre>
@Null	Arbitrary	Validates that the target field is null. (Example: usage in group validation)	<pre>@Null(groups={Update.class}) private String id;</pre>
@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)	<pre>@Pattern(regexp = "[0-9]+") private String tel;</pre>
@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	Validate whether the value is less than the maximum value.	<pre>@Min(1) @Max(100) private int quantity;</pre>
5.5. Input Validation	(In case of Hibernate Validator implementation, it is also possible to use it with arbitrary CharSequence, Number		611

Hibernate Validator check rules

All the major annotations of Hibernate Validator are shown below.

Refer to [Hibernate Validator specifications](#)for details.

Annotation(org.hibernate.validator.constraints.*)	Description	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.	<pre>@CreditCardNumber private String cardNumber;</pre>
@Email	It is applicable to the any CharSequence inherited class	Validate whether the Email address is complaint with RFC2822.	<pre>@Email private String email;</pre>
@URL	It is applicable to any CharSequence inherited class	Validate whether it is compliant with RFC2396.	<pre>@URL private String url;</pre>
@NotBlank	It is applicable to any CharSequence inherited class	Validate that it is not Null, empty string ("") and space only.	<pre>@NotBlank private String userId;</pre>
@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.	<pre>@NotEmpty private String password;</pre>

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 {max}
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 evaluate to true
org.hibernate.validator.constraints.URL.message       = must be a valid URL
org.hibernate.validator.constraints.br.CNPJ.message   = invalid Brazilian corporate taxpayer identification number
org.hibernate.validator.constraints.br.CPF.message    = invalid Brazilian individual taxpayer identification number
org.hibernate.validator.constraints.br.TituloEleitor.message = invalid Brazilian Voter ID card number

```

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.

Name: Taro Yamada
 Email: yamada@example.com
 Age: ten Failed to convert property value of type java.lang.String to required type
 java.lang.Integer for property age; nested exception is
 java.lang.NumberFormatException: For input string: "ten"

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.

The screenshot shows a web form with three input fields: 'Name' (Taro Yamada), 'Email' (yamada@example.com), and 'Age' (ten). The 'Age' field has a red border around it. Below the 'Age' field, the text "'Age' must be an integer." is displayed in red, indicating an validation error. At the bottom of the form is a 'Confirm' button.

Field name can be inserted in the message by specifying {0} in the message; This is as per the explanation in *Messages to be defined in application-messages.properties*.

Default messages should be defined.

Tip: Refer to [Javadoc](#) for 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 without 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 Logging

Note: This contents described in this chapter are just guidelines that can be customized as per the business requirements.

5.6.1 Overview

When a system is being operated, logs and messages are output as information sources to analyze the business process usage and to determine the causes when the system is down or when a business error etc. occurs.

It is important to output the log since effectiveness of analysis improves significantly if it is output at the time of debugging.

If only the behavior is to be checked, it can be done by debugging in IDE or through a simple output such as `System.out.println`.

However, if the output result is not manually stored, it is not possible to check the results later. This hampers the effectiveness of analysis.

Obtaining the log by implementing logging library is just writing the code to be output.

Log can be verified at any convenient time later on.

Considering operating time, trail and analysis, it is recommended to implement logging library.

There are various log output methods in Java and a number of methods can be selected; however this guideline recommends [SLF4J](#) (interface) + [Logback](#) (implementation) due to simplicity of coding, ease of modification and performance.

Types of Logs

A typical log at the time of application development is shown below.

Log level	Category	Purpose of output	Output contents
TRACE	Performance log	Measurement of request processing time (Should not be output during production environment operations)	Process start and end time, process elapsed time (ms), Information that can identify the execution process etc. (execution controller + method, request URL etc.).
DEBUG	Debug log	Debug at the time of development (Should not be output during production environment operations)	Optional (Executed query, Input parameter, Return value etc.)
INFO	Access log	Assessing business process volume	Information that can identify access date and time, user (IP address, authentication information) Information that can identify execution process (request URL) etc., information for leaving a trail
INFO	External communication log	Analysis of error that occurs while communicating with external system	Send and receive time, send and receive data etc.
WARN	Business error log	Business error records	Business error occurrence time, Message ID and message corresponding to business error Information necessary for analysis such as input information, exception message etc.
ERROR	System error log	Record the events where it is difficult to continue a system operation	System error occurrence time, Message and message ID corresponding to system error Information necessary for analysis such as input information, exception message etc. Basically, framework is output and business process logic is not output.
ERROR 618	Monitoring log	Monitoring the occurrence of an exception	5 Architecture in Detail - TERASOLUNA Global Framework Exception occurrence time, Message ID corresponding to system error Use tools for monitoring and keep the output contents to a minimum.

Debug log, Access log, Communication log, Business error log and System error log are output to same file.

In this guideline, the log file that outputs the above mentioned logs is called as application log.

Note: The sequence of log levels of SLF4J or Logback is TRACE < DEBUG < INFO < WARN < ERROR. It does not include FATAL level provided in commons-loggings or Log4J.

Output contents of log

It is important to note the points given below for the output contents of log.

1. ID to be output to log

When log is to be monitored during the operation, it is recommended to include a message ID in the log to be monitored.

Further, when the business process volume is to be assessed using an access log, the log should be output according to the ID assigned to each business process as explained in Message Management.

This facilitates overall data compilation.

Note: The readability of log is enhanced by including an ID in the log thereby reducing the time required for primary isolation of failure analysis. Refer to *Message Management* for log ID structure. However, there is no need to assign an ID to all the logs. ID is not required at the time of debugging. It is recommended when the system is operational so as to isolate the log quickly.

During failure, when a system user is notified by displaying a log ID (or a message ID) on the error screen and the ID is then notified to the call center, for that user, failure analysis becomes easier.

However, note that the vulnerabilities of the system may be exposed if errors are displayed on the screen along with the failure details.

In common library, the mechanism(component) is provided to include the message ID(exception code) into the log and the screen when an exception is occurred. Details refer to “*Exception Handling*”.

2. Traceability

To improve the traceability, it is recommended to output a unique track ID (hereafter referred to as X-Track) at request level in each log.

Example of logs including X-Track is given below.

date:2013-09-06 19:36:31	X-Track:85a437108e9f4a959fd227f07f72ca20	message: [START C
date:2013-09-06 19:36:31	X-Track:85a437108e9f4a959fd227f07f72ca20	message: [END CON
date:2013-09-06 19:36:31	X-Track:85a437108e9f4a959fd227f07f72ca20	message: [HANDLING
date:2013-09-06 19:36:33	X-Track:948c8b9fd04944b78ad8aa9e24d9f263	message: [START C
date:2013-09-06 19:36:33	X-Track:142ff9674efd486cbd1e293e5aa53a78	message: [START C

date:2013-09-06 19:36:33	X-Track:142ff9674efd486cbd1e293e5aa53a78	message: [END CON
date:2013-09-06 19:36:33	X-Track:142ff9674efd486cbd1e293e5aa53a78	message: [HANDLIN
date:2013-09-06 19:36:33	X-Track:948c8b9fd04944b78ad8aa9e24d9f263	message: [END CON
date:2013-09-06 19:36:33	X-Track:948c8b9fd04944b78ad8aa9e24d9f263	message: [HANDLIN

Logs can be linked together using Track IDs even when the output is irregular.

In the above example, it can be clearly understood that 4th, 8th and 9th rows in the log are pertaining to the same request.

In common library, `org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter` to be added to MDC is provided by generating a unique key for each request.

`XTrackMDCPutFilter` sets Track ID in “X-Track” of HTTP response header as well. X-Track is used as a Track ID label in the log.

Refer to [About MDC](#) for the usage methods.

3. Log mask

If personal information, credit card number etc. are output to the log file as is, the information that has security threat should be masked if needed.

Log output points

Category	Output points
Performance log	<p>The processing time of business process is measured and it is output after executing business process. Request processing time is measured and log is output when response is returned.</p> <p>It is usually implemented in AOP or Servlet filter.</p> <p>Common library provides <code>org.terasoluna.gfw.web.logging.TraceLoggingInterceptor</code> which outputs processing time of SpringMVC Controller method in TRACE log after the execution of processing method of Controller.</p>
Debug log	<p>When it is necessary to output debug information at the time of development, a suitable log output process is implemented in source code.</p> <p>Common library provides <code>org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener</code> listener which outputs DEBUG log at the time of HTTP session creation/destruction/attribute addition.</p>
Access log	<p>INFO log is output at the time of receiving a request and returning the response.</p> <p>It is usually implemented in AOP or Servlet filter.</p>
External communication log	INFO log is output before and after external system linking.
Business error log	<p>WARN log is output when business process exception is thrown.</p> <p>It is usually implemented in AOP.</p> <p>In common library, when <code>org.terasoluna.gfw.common.exception.BusinessException</code> is thrown at the time of business process execution, <code>org.terasoluna.gfw.common.exception.BusinessExceptionLoggingInterceptor</code> that outputs WARN log is provided.</p>
622	Refer to Exception Handling for details. 5 Architecture in Detail - TERASOLUNA Global Framework
System error log	An ERROR log is output when system exception or unexpected exception occurs. It is usually implemented in AOP or Servlet filter.

Note: Note that when the log is output, the contents should not be exactly identical to other logs. This is helpful in easily identifying the location of the output.

5.6.2 How to use

The following are required to output the log in SLF4J + Logback.

1. Settings of Logback
2. Calling API of SLF4J

Settings of Logback

Logback settings are described in logback.xml under the class path. An example of configuration is shown below.

Refer to [Official Manual](#) for the detailed configuration of logback.xml.

Note: Settings of Logback are read automatically as per the rules given below.

1. logback.groovy on class path
2. If file “1” is not found, logback-text.xml on class path
3. If file “2” is not found, logback.xml on class path
4. If file “3” is not found, settings of BasicConfigurator class (console output)

In this guideline, it is recommended to place logback.xml in the class path. Moreover, apart from automatic reading, it is possible to read programmatically through an [API](#) and specify the configuration file in system properties.

logback.xml

```
<!DOCTYPE logback>
<configuration>

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender"> <!-- (1) -->
        <encoder>
            <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss} \tthread:%thread\tx-Track:%X{X-Track}\ttx-Track:&lt;/pattern>
        </encoder>
    </appender>
```

```
<appender name="APPLICATION_LOG_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>log/projectName-application.log</file> <!-- (4) -->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>log/projectName-application-%d{yyyyMMddHH}.log</fileNamePattern> <!--
        <maxHistory>7</maxHistory> <!-- (6) -->
    </rollingPolicy>
    <encoder>
        <charset>UTF-8</charset> <!-- (7) -->
        <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tthread:%thread\tx-Track:%X{X-Track}\tlevel:-5level\trid:%X{X-Track}]]></pattern>
    </encoder>
</appender>

<appender name="MONITORING_LOG_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>log/projectName-monitoring.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>log/projectName-monitoring-%d{yyyyMMdd}.log</fileNamePattern>
        <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder>
        <charset>UTF-8</charset>
        <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss}\tx-Track:%X{X-Track}\tlevel:-5level\trid:%X{X-Track}]]></pattern>
    </encoder>
</appender>

<!-- Application Loggers -->
<logger name="com.example.sample"> <!-- (9) -->
    <level value="debug" />
</logger>

<!-- TERASOLUNA -->
<logger name="org.terasoluna.gfw">
    <level value="info" />
</logger>
<logger name="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor">
    <level value="trace" />
</logger>
<logger name="org.terasoluna.gfw.common.exception.ExceptionLogger">
    <level value="info" />
</logger>
<logger name="org.terasoluna.gfw.common.exception.ExceptionLogger.Monitoring" additivity="false">
    <level value="error" />
    <appender-ref ref="MONITORING_LOG_FILE" />
</logger>

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

<logger name="org.springframework.web.servlet">
```

```
<level value="info" />
</logger>

<root level="warn"> <!-- (11) -->
  <appender-ref ref="STDOUT" /> <!-- (12) -->
  <appender-ref ref="APPLICATION_LOG_FILE" />
</root>

</configuration>
```

Sr. No.	Description
(1)	<p>Definition of appender is specified to output the log on console. It can be selected whether the output destination is standard output or standard error, however when the destination is not specified, it is considered as standard output.</p>
(2)	<p>The output format for log is specified. If the format is not specified, the message alone is output. Time and message level etc. are output according to the business requirements. Here, LTSV (Labeled Tab Separated Value) of “Label:Value<TAB>Label:Value<TAB>...” format is set.</p>
(3)	<p>Definition for appender is specified to output application log. The appender to be used can also be specified in <logger>, however, here it is referred to root (11) since application log is used by default. RollingFileAppender is commonly used at the time of application log output, however, FileAppender may also be used to implement log rotation using another functions such as logrotate.</p>
(4)	<p>A current file name (File name of log being output) is specified. It should be specified when it is necessary to specify a fixed file name. If <file>log file name</file> is not specified, it is output with the name in pattern (5).</p>
(5)	<p>Name of the file after rotation is specified. Usually, date or time format is widely used. Note that 24 hour clock is not set if HH is mistakenly set as hh.</p>
(6)	<p>The number of remaining files for which rotation is performed is specified.</p>
(7)	<p>A character code of log file is specified.</p>
(8)	<p>It is set so as to output the application log by default.</p>
626	<p>The logger name is set so that log output can be controlled by the above debug level.</p>
(10)	<p>A monitoring log is set. Refer to Common Settings of Exception Handling.</p>

Tip: About LTSV(Labeled Tab Separated Value)

LTSV is one of text data formats, and mainly used as the log format.

For log fomart, LTSV is easy to parse using some tools because it has following features.

- It's easy to split the field compared to other delimiters because tabs is used as field delimiters.
- Even if the field definition (changing the position of field or adding the field or removing the field) is changed, it does not affect to parsing because of including a label(name) in the field.

It is also one of features that there are that pasting on the Excel can format it with the least effort.

The following three items should be set in logback.xml.

Type	Overview
appender	“Location” and “Layout” to be used for output
root	Default “Appender” and the minimum “Log level” to be used for output
logger	“Which logger (package or class etc.)” is to be output at which minimum “log level”

In <appender> element, the “location” and the “layout” to be used for output are defined. It is not used at the time of the log output only by defining the appender. It is used for the first time when it is referred in <logger> element or <root> element. There are two attributes, namely, “name” and “class” and both are mandatory.

Attribute	Overview
name	Name of the appender. It is specified by appender-ref. Any name can be assigned as there is no restriction on assigning the name.
class	FQCN of appender implementation class.

The main appenders that are provided are shown below.

Appender	Overview
ConsoleAppender	Console output
FileAppender	File output
RollingFileAppender	File output (Rolling possible)
AsyncAppender	Asynchronous output. It is used for logging in processes with high performance requirement. (It is necessary to set the output destination in other Appender.)

Refer to [Official Manual](#) for detailed Appender types.

Basic log output by calling API of SLF4J

Log is output by calling a method according to each log level of SLF4J logger(`org.slf4j.Logger`).

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

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

@Controller
public class HomeController {

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

    @RequestMapping(value = "/", method = { RequestMethod.GET,
        RequestMethod.POST })
    public String home(Model model) {
        logger.trace("This log is trace log.");
        logger.debug("This log is debug log.");
        logger.info("This log is info log.");
        logger.warn("This log is warn log.");
        logger.error("This log is error log.");
        return "welcome/home";
    }

}
```

Sr. No.	Description
(1)	Logger is generated from <code>org.slf4j.LoggerFactory</code> . If Class object is set as an argument of <code>getLogger</code> , the logger name acts as an FQCN of that class. In this example, the logger name is “com.example.sample.app.welcome.HomeController”.
(2)	The log of TRACE level is output.
(3)	The log of DEBUG level is output.
(4)	The log of INFO level is output.
(5)	The log of WARN level is output.
(6)	The log of ERROR level is output.

Log output results are shown below. Log level of com.example.sample is DEBUG, hence TRACE log is not output.

```
date:2013-11-06 20:13:05      thread:tomcat-http--3 X-Track:5844f073b7434b67a875cb85b131e686
date:2013-11-06 20:13:05      thread:tomcat-http--3 X-Track:5844f073b7434b67a875cb85b131e686
date:2013-11-06 20:13:05      thread:tomcat-http--3 X-Track:5844f073b7434b67a875cb85b131e686
date:2013-11-06 20:13:05      thread:tomcat-http--3 X-Track:5844f073b7434b67a875cb85b131e686
```

The description can be as given below when an argument is to be entered in placeholder of a log message.

```
int a = 1;
logger.debug("a={}", a);
String b = "bbb";
logger.debug("a={}, b={}", a, b);
```

The log given below is output.

```
date:2013-11-06 20:32:45      thread:tomcat-http--3 X-Track:853aa701a401404a87342a574c69efbc
date:2013-11-06 20:32:45      thread:tomcat-http--3 X-Track:853aa701a401404a87342a574c69efbc
```

Warning: Note that string concatenation such as `logger.debug("a=" + a + " , b=" + b);` should not be carried out.

When the exception is to be caught, ERROR log (WARN log in some cases) is output as shown below. Error message and exception generated are passed to the log method.

```
public String home(Model model) {
    // omitted

    try {
        throwException();
    } catch (Exception e) {
        logger.error("Exception happened!", e);
        // omitted
    }
    // omitted
}

public void throwException() throws Exception {
    throw new Exception("Test Exception!");
}
```

Accordingly, stack trace of caused exception is output and the cause of the error can be easily analyzed.

```
date:2013-11-06 20:38:04      thread:tomcat-http--5      X-Track:11d7dbdf64e44782822c5aea4fc4bb4f
java.lang.Exception: Test Exception!
    at com.example.sample.app.welcome.HomeController.throwException(HomeController.java:40) ~[HomeController.class]
    at com.example.sample.app.welcome.HomeController.home(HomeController.java:31) ~[HomeController.class]
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.7.0_40]
    (omitted)
```

However, as shown below, when the exception that is caught is wrapped with other exception and is re-thrown at upper level, there is no need to output the log. This is because usually the error log is output at upper level.

```
try {
    throwException();
} catch (Exception e) {
    throw new SystemException("e.ex.fw.9001", e);
    // no need to log
}
```

Note: When cause exception is to be passed to a log method, a placeholder cannot be used. Only in this case, the message argument can be concatenated using a string.

```
try {
    throwException();
} catch (Exception e) {
    // NG => logger.error("Exception happend! [a={} , b={}]", e, a, b);
```

```
    logger.error("Exception happend! [a=" + a + " , b=" + b + "] ", e);
    // omitted
}
```

Points to be noted for the description of log output

SLF4J Logger internally checks the log level and outputs actual log only for the required levels.

Therefore, the log level check as given below is basically not necessary.

```
if (logger.isDebugEnabled()) {
    logger.debug("This log is Debug.");
}

if (logger.isDebugEnabled()) {
    logger.debug("a={}", a);
}
```

However, the log level should be checked in the cases given below to prevent performance degradation.

1. When there are 3 or more arguments

When arguments of log message are 3 or more, argument array should be passed in the API of SLF4J.

Log level should be checked in order to avoid the cost for generating an array and the array should be generated only when necessary.

```
if (logger.isDebugEnabled()) {
    logger.debug("a={}, b={}, c{}", new Object[] { a, b, c });
}
```

2. When it is necessary to call a method for creating an argument

When it is necessary to call a method while creating an argument for the log message, the log level should be checked to avoid the method execution cost and the method should be executed only when necessary.

```
if (logger.isDebugEnabled()) {
    logger.debug("xxx={}, foo.getXxx()");
}
```

5.6.3 Appendix

Using MDC

A cross-sectional log can be output by using **MDC** (Mapped Diagnostic Context).

Log traceability improves if same information (such as user name or unique request ID) is included in the log to be output in a request.

MDC internally consists of a ThreadLocal map and sets value for the key. The value set in log can be output till it is removed.

The value should be set at the beginning of the request and removed at the time of process termination.

Basic usage method

An example of using MDC is given below.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

public class Main {

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

    public static void main(String[] args) {
        String key = "MDC_SAMPLE";
        MDC.put(key, "sample"); // (1)
        try {
            logger.debug("debug log");
            logger.info("info log");
            logger.warn("warn log");
            logger.error("error log");
        } finally {
            MDC.remove(key); // (2)
        }
        logger.debug("mdc removed!");
    }
}
```

The value added to MDC can be output in log by defining the output format as %X{key name}format in <pattern> of logback.xml.

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss} \t thread:%thread\tdcSample:%X{MDC_SAMPLE}\t]>
  </encoder>
</appender>
```

Execution results are as follows:

date:2013-11-08 17:45:48	thread:main mdcSample:sample	level:DEBUG	message:debug log
date:2013-11-08 17:45:48	thread:main mdcSample:sample	level:INFO	message:info log
date:2013-11-08 17:45:48	thread:main mdcSample:sample	level:WARN	message:warn log
date:2013-11-08 17:45:48	thread:main mdcSample:sample	level:ERROR	message:error log
date:2013-11-08 17:45:48	thread:main mdcSample: sample	level:DEBUG	message:mdc removed!

Note: If `MDC.clear()` is executed, all the added values are deleted.

Setting value in MDC using Filter

Common library provides `org.terasoluna.gfw.web.logging.mdc.AbstractMDCPutFilter` as a base class to add/delete values from MDC using filter.

Further, following are provided as implementation classes.

- `org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter` to set an unique ID for each request in MDC
- `org.terasoluna.gfw.security.web.logging.UserIdMDCPutFilter` to set authentication user name of Spring Security in MDC

If individual values are to be added to MDC using Filter, it is desirable to implement

`AbstractMDCPutFilter`

based on implementation of `org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter`.

How to use MDCCFilter

Definition of MDCCFilter is added to the filter definition of web.xml.

```
<!-- omitted -->

<!-- (1) -->
<filter>
  <filter-name>MDCClearFilter</filter-name>
```

```
<filter-class>org.terasoluna.gfw.web.logging.mdc.MDCClearFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>MDCClearFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- (2) -->
<filter>
  <filter-name>XTrackMDCPutFilter</filter-name>
  <filter-class>org.terasoluna.gfw.web.logging.mdc.XTrackMDCPutFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>XTrackMDCPutFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

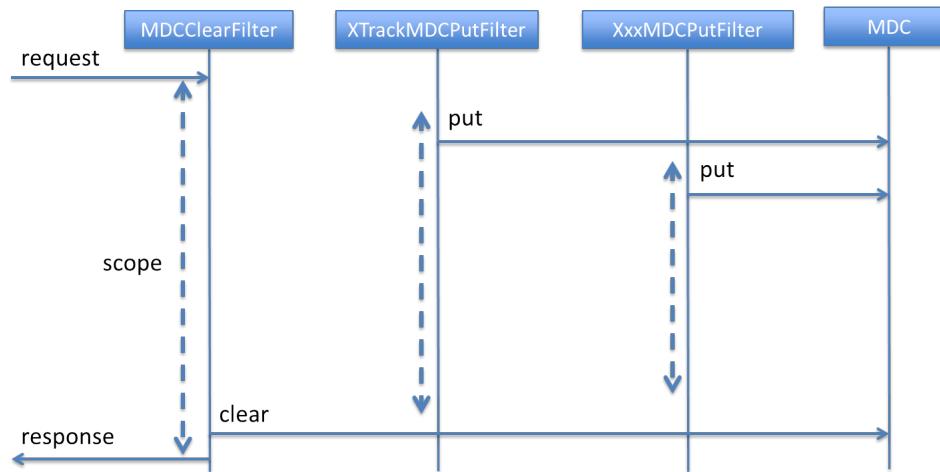
<!-- (3) -->
<filter>
  <filter-name>UserIdMDCPutFilter</filter-name>
  <filter-class>org.terasoluna.gfw.security.web.logging.UserIdMDCPutFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>UserIdMDCPutFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- omitted -->
```

Sr. No.	Description
(1)	MDCClearFilter that clears the contents of MDC is set. Values in MDC added by various MDCPutFilter are deleted by this Filter.
(2)	XTrackMDCPutFilter is set. XTrackMDCPutFilter sets Request ID in key “X-Track”.
(3)	UserIdMDCPutFilter is set. UserIdMDCPutFilter sets User ID in key “USER”.

As shown in the sequence diagram below, MDCClearFilter should be defined prior to each MDCPutFilter to clear the contents of MDC for post-processing.

Request ID and User ID can be output to log by adding %X{X-Track} and %X{USER} in <pattern> of



logback.xml.

```

<!-- omitted -->
<appender name="APPLICATION_LOG_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>log/projectName-application.log</file>
  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
    <fileNamePattern>log/projectName-application-%d{yyyy-MMdd}.log</fileNamePattern>
    <maxHistory>7</maxHistory>
  </rollingPolicy>
  <encoder>
    <charset>UTF-8</charset>
    <pattern><![CDATA[date:%d{yyyy-MM-dd HH:mm:ss} \tthread:%thread\tUSER:%X{USER} \tX-Track:%X{X-Track}]]></pattern>
  </encoder>
</appender>
<!-- omitted -->
  
```

Example of log output

```

date:2013-09-06 23:05:22  thread:tomcat-http--3  USER:      X-Track:97988cc077f94f9d9d435f6f7602742
date:2013-09-06 23:05:22  thread:tomcat-http--3  USER:anonymousUser  X-Track:97988cc077f94f9d9d4
  
```

Note: User information to be set in MDC by `UserIdMDCPutFilter` is created by Spring Security Filter. As mentioned earlier, if `UserIdMDCPutFilter` is defined in `web.xml`, user ID is output to log after the completion of a series of Spring Security processes. If user information is to be output to the log immediately after it is generated, the information should be incorporated in Spring Security Filter as shown below by deleting `web.xml` definition.

The definitions given below are added to `spring-security.xml`.

```
<sec:http auto-config="true" use-expressions="true">
    <!-- omitted -->
    <sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/> <!-- (1) -->
    <!-- omitted -->
</sec:http>

<!-- (2) -->
<bean id="userIdMDCPutFilter" class="org.terasoluna.gfw.security.web.logging.UserIdMDCPutFilter">
</bean>
```

Sr. No.	Description
(1)	UserIdMDCPutFilter defined in Bean is added after “ANONYMOUS_FILTER”.
(2)	UserIdMDCPutFilter is defined.

In blank project, UserIdMDCPutFilter is defined in spring-security.xml.

Log output related functionalities provided by common library

HttpSessionEventLoggingListener

`org.terasoluna.fw.web.logging.HttpSessionEventLoggingListener` is a listener class that outputs debug log at the time of generating, discarding, activating or deactivating session, and adding or deleting session attributes.

The following should be added to web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">
    <listener>
        <listener-class>org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener</listener-class>
    </listener>

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

In logback.xml, `org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener` is set at debug level as shown below.

```
<logger
    name="org.terasoluna.gfw.web.logging.HttpSessionEventLoggingListener"> <!-- (1) -->
```

```
<level value="debug" />
</logger>
```

Debug log as shown below is output.

```
date:2013-09-06 16:41:33    thread:tomcat-http--3    USER:    X-Track:c004ddb56a3642d5bc5f6b5d884e5
```

When the lifecycle of an object is managed using a Session such as @SessionAttribute etc., it is strongly recommended to use this listener to confirm whether the attributes added to the session are deleted as anticipated.

TraceLoggingInterceptor

org.terasoluna.gfw.web.logging.TraceLoggingInterceptor is the HandlerInterceptor to output start and termination of Controller process to the log. When the process is terminated, View name returned by the Controller, attributes added to Model and the time required for Controller process are also output.

TraceLoggingInterceptor is added in <mvc:interceptors> of spring-mvc.xml as shown below.

```
<mvc:interceptors>
  <!-- omitted -->
  <mvc:interceptor>
    <mvc:mapping path="/**" />
    <mvc:exclude-mapping path="/resources/**" />
    <bean
      class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor">
    </bean>
  </mvc:interceptor>
  <!-- omitted -->
</mvc:interceptors>
```

By default, WARN log is output if Controller process takes more than 3 seconds.

When the threshold value is to be changed, it is specified in nano seconds in warnHandlingNanos property.

The following settings should be performed if the threshold value is to be changed to 10 seconds ($10 * 1000 * 1000 * 1000$ nano seconds).

```
<mvc:interceptors>
  <!-- omitted -->
  <mvc:interceptor>
    <mvc:mapping path="/**" />
    <mvc:exclude-mapping path="/resources/**" />
    <bean
      class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor">
      <property name="warnHandlingNanos" value="#{10 * 1000 * 1000 * 1000}" />
    </bean>
  </mvc:interceptor>
</mvc:interceptors>
```

```
</bean>
</mvc:interceptor>
<!-- omitted -->
</mvc:interceptors>
```

In logback.xml, org.terasoluna.gfw.web.logging.TraceLoggingInterceptor is set at trace level as shown below.

```
<logger name="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor"> <!-- (1) -->
  <level value="trace" />
</logger>
```

ExceptionLogger

org.terasoluna.gfw.common.exception.ExceptionLogger is provided as a logger when an exception occurs.

Refer to [*How to use of Exception Handling*](#) for the usage method.

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:

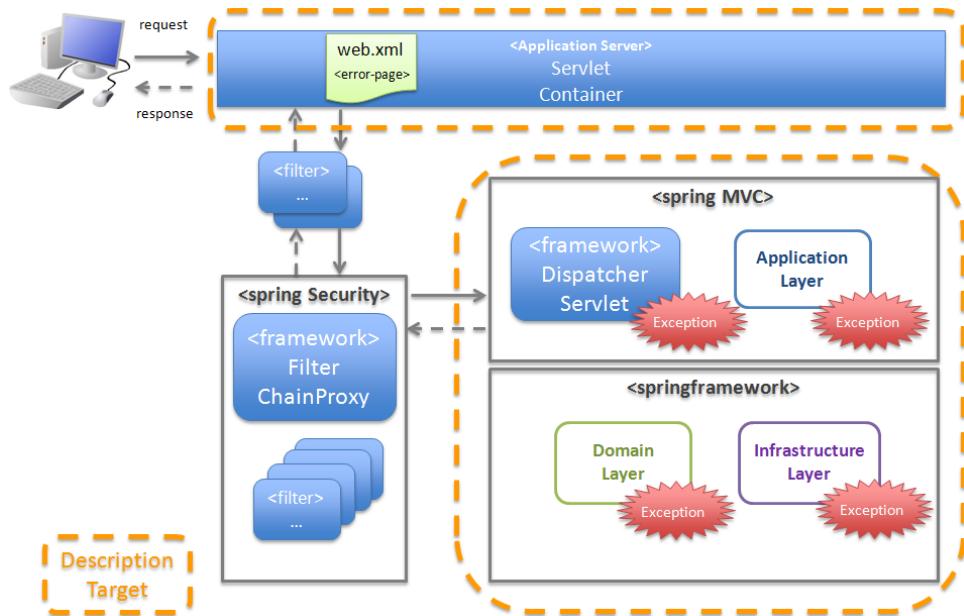


Figure.5.28 Figure - Description Targets

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.5.13 Table - Types of exceptions that occur when an application is running

Sr. No.	Classification	Description	<i>Types of exceptions</i>
(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. <i>Business exception</i> 2. <i>Library exceptions that occurs during normal operation</i>
(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. <i>System Exception</i> 2. <i>Unexpected System Exception</i> 3. <i>Fatal Errors</i>
(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. <i>Framework exception in case of invalid requests</i>

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.5.14 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.	<p>Use this in order to handle exceptions at request (Controller method) level.</p> <p>For details, refer to <i>Basic flow when the Controller class handles the exception at request level.</i></p>	1. <i>When notifying partial redo of a use case (from middle)</i>
(2)	Use @ExceptionHandler annotation to carry out exception handling in application code.	<p>Use this when exceptions are to be handled at use case (Controller) level.</p> <p>For details, refer to <i>Basic flow when the Controller class handles the exception at use case level.</i></p>	1. <i>When notifying redo of a use case (from beginning)</i>
(3)	Use HandlerExceptionResolver mechanism provided by the framework to carry out exception handling.	<p>Use this in order to handle exceptions at servlet level.</p> <p>When <mvc:annotation-driven> is specified, HandlerExceptionResolver uses automatically registered class, and SystemExceptionResolver provided by common library.</p> <p>For details, refer to <i>Basic flow when the framework handles the exception at servlet level.</i></p>	1. <i>When notifying that the system or application is not in a normal state</i> 2. <i>When notifying that the request contents are invalid</i>
5.7(4) Exception Handling	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. <i>When a fatal error has been detected</i> 2. <i>When notifying that an</i>

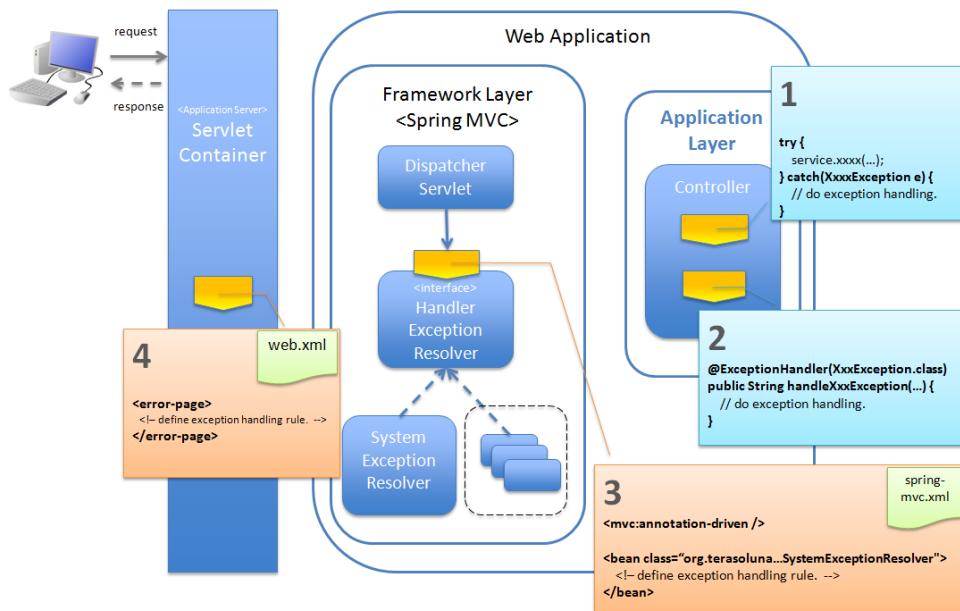


Figure.5.29 Figure - Exception Handling Methods

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)	<p>This is a class for handling the exceptions by calling the methods of Controller class with <code>@ExceptionHandler</code> annotation.</p> <p>This class is necessary for implementing the handling method No. 2.</p>
(2)	ResponseStatusExceptionResolver (order=1)	<p>This is a class for handling the exceptions wherein <code>@ResponseStatus</code> is applied as class level annotation.</p> <p><code>HttpServletResponse#sendError(int sc, String msg)</code> is called with the value specified in <code>@ResponseStatus</code>.</p>
(3)	DefaultHandlerExceptionResolver (order=2)	<p>This is a class for handling framework exceptions in Spring MVC.</p> <p><code>HttpServletResponse#sendError(int sc)</code> is called using the value of HTTP response code corresponding to framework exception.</p> <p>For details on HTTP response code to be set, refer to <i>HTTP response code set by DefaultHandlerExceptionResolver</i>.</p>

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.5.15 Table - Types of Exceptions

Sr. No.	Types of Exceptions	Description
(1)	<i>Business exception</i>	Exception to notify a violation of a business rule
(2)	<i>Library exceptions that occurs during normal operation</i>	Amongst the exceptions that occur in framework and libraries, the exception which is likely to occur when the system is operating normally.
(3)	<i>System Exception</i>	Exception to notify detection of a state which should not occur when the system is operating normally
(4)	<i>Unexpected System Exception</i>	Unchecked exception that does not occur when the system is operating normally
(5)	<i>Fatal Errors</i>	Error to notify an occurrence of a fatal error impacting the entire system (application)
(6)	<i>Framework exception in case of invalid requests</i>	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.UncategorizedDataAccessException 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.5.16 Table - Exception Handling Patterns

Sr. No.	Purpose of handling	Types of exceptions	Handling method	Handling location
(1)	<i>When notifying partial redo of a use case (from middle)</i>	1. <i>Business exception</i>	Application code (try-catch)	Request
(2)	<i>When notifying redo of a use case (from beginning)</i>	1. <i>Business exception</i> 2. <i>Library exceptions that occurs during normal operation</i>	Application code (@ExceptionHandler)	Use case
(3)	<i>When notifying that the system or application is not in a normal state</i>	1. <i>System Exception</i> 2. <i>Unexpected System Exception</i>	Framework (Handling rules are specified in spring-mvc.xml)	Servlet
(4)	<i>When notifying that the request contents are invalid</i>	1. <i>Framework exception in case of invalid requests</i>	Framework	Servlet
(5)	<i>When a fatal error has been detected</i>	1. <i>Fatal Errors</i>	Servlet container (Handling rules are specified in web.xml)	Web application
(6)	<i>When notifying that an exception has occurred in the presentation layer (JSP etc.)</i>	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.

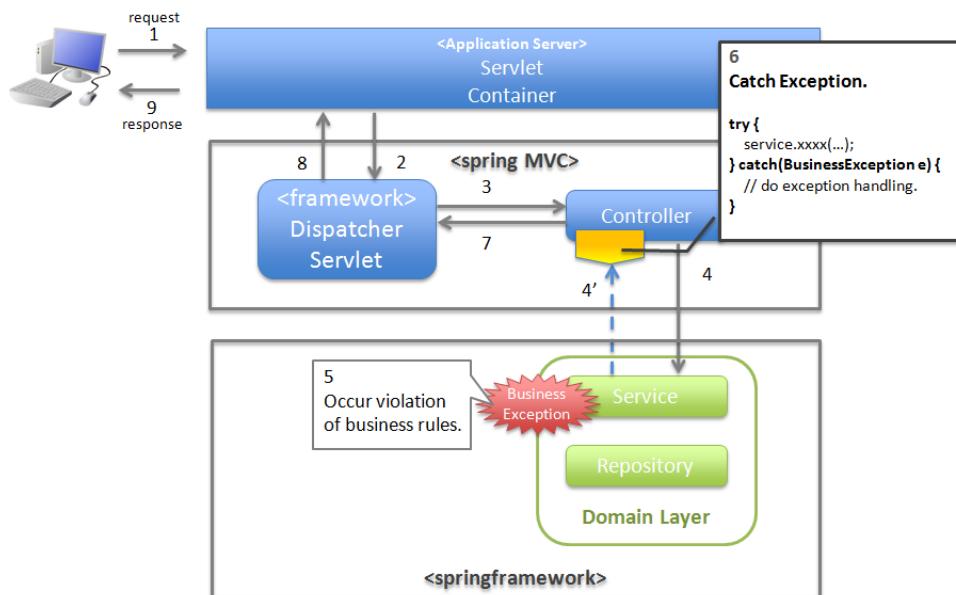


Figure.5.30 Figure - Handling method when notifying partial redo of a use case (from middle)

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.

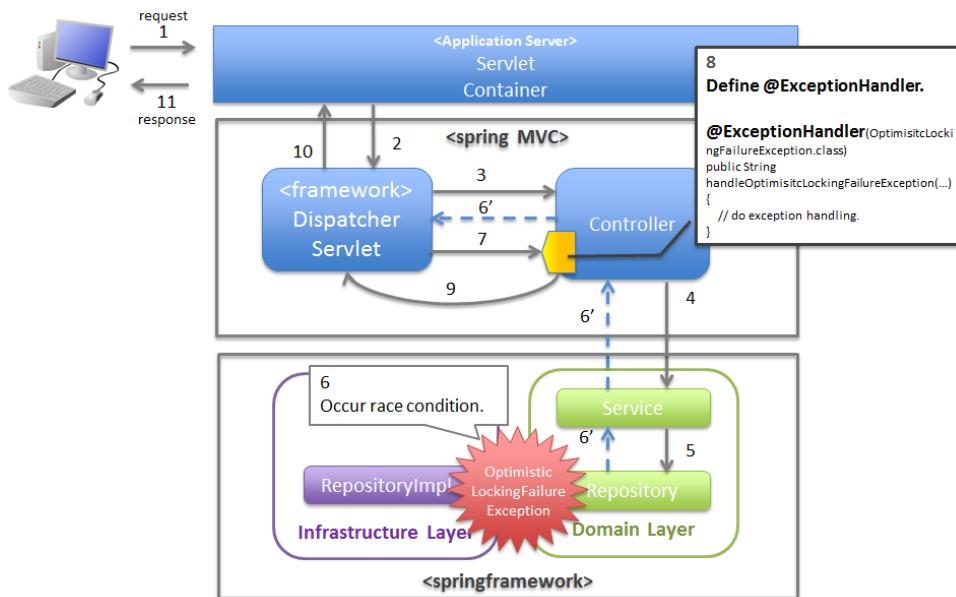


Figure.5.31 Figure - Handling method when notifying redo of a use case (from beginning)

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 ...

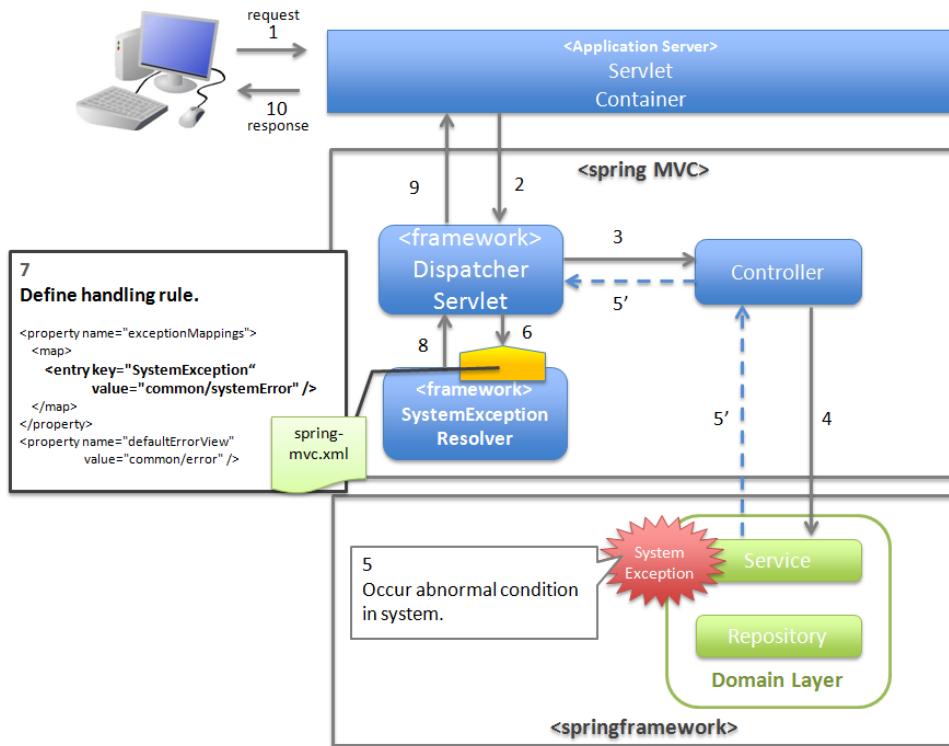


Figure 5.32 Figure - Handling method when an exception to notify that system or application is not in a normal state is detected

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.

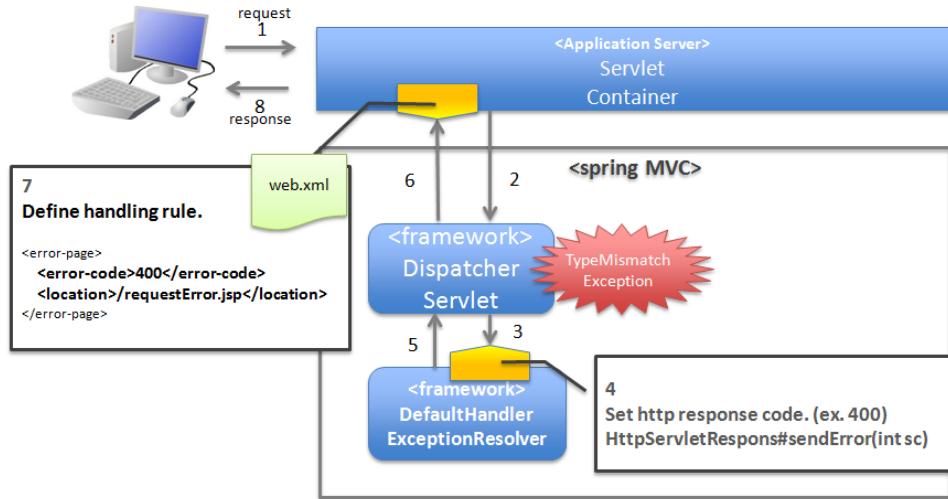


Figure.5.33 Figure - Handling method when notifying that the request contents are invalid

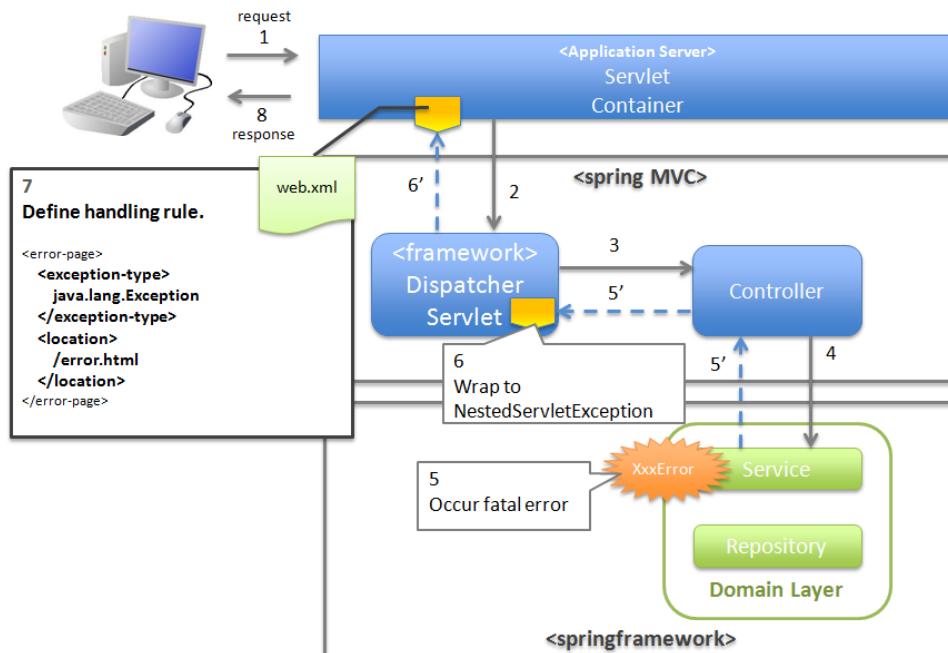


Figure.5.34 Figure - Handling method when a fatal error has been detected

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 common library, refer to [Exception handling classes provided by the](#)

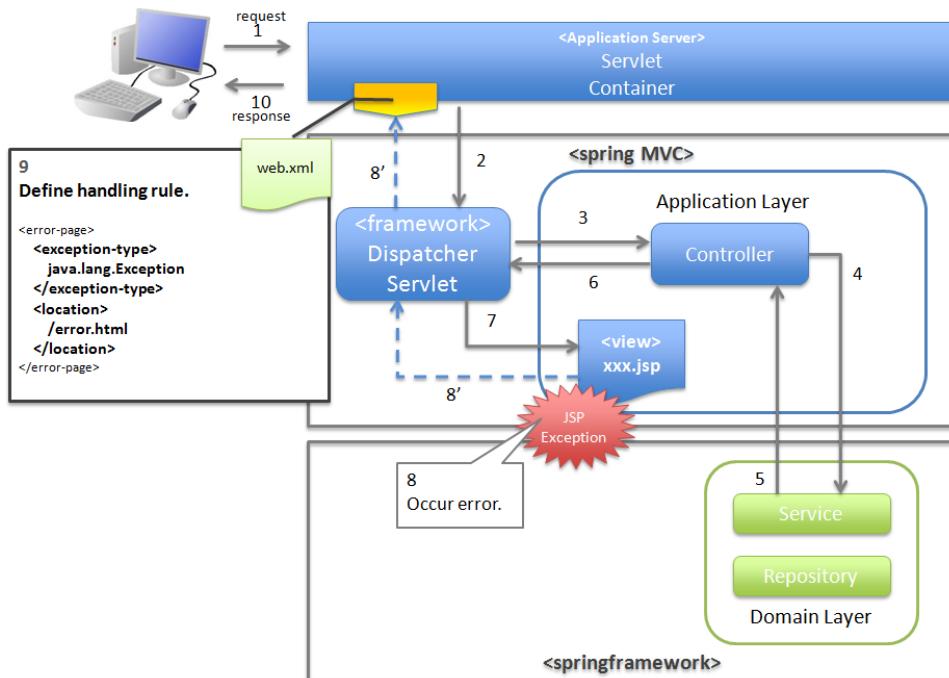


Figure.5.35 Figure - Handling method when notifying an occurrence of exception in the presentation layer (JSP etc.)

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 common library.

Log is output using interceptor

(org.terasoluna.gfw.common.exception.ResultMessagesLoggingInterceptor) which records that an exception holding the result message has occurred.

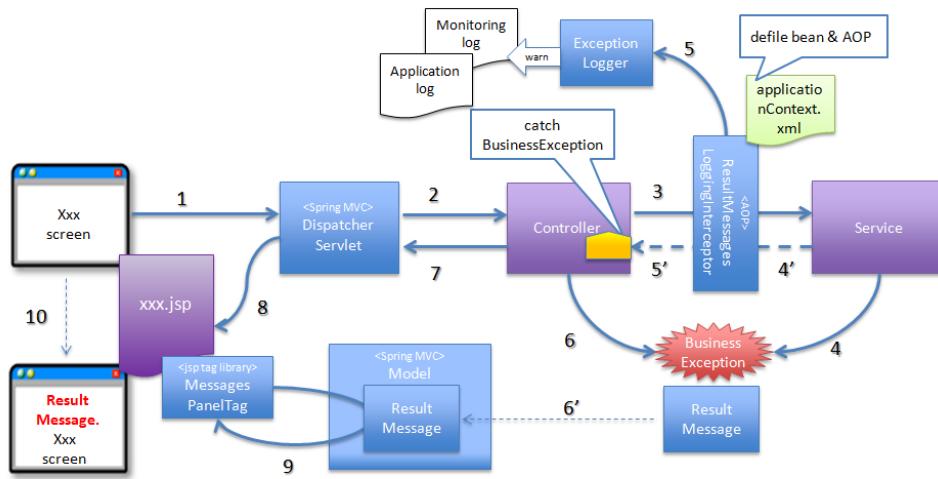


Figure 5.36 Figure - Basic flow when the Controller class handles the exception at request level

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.

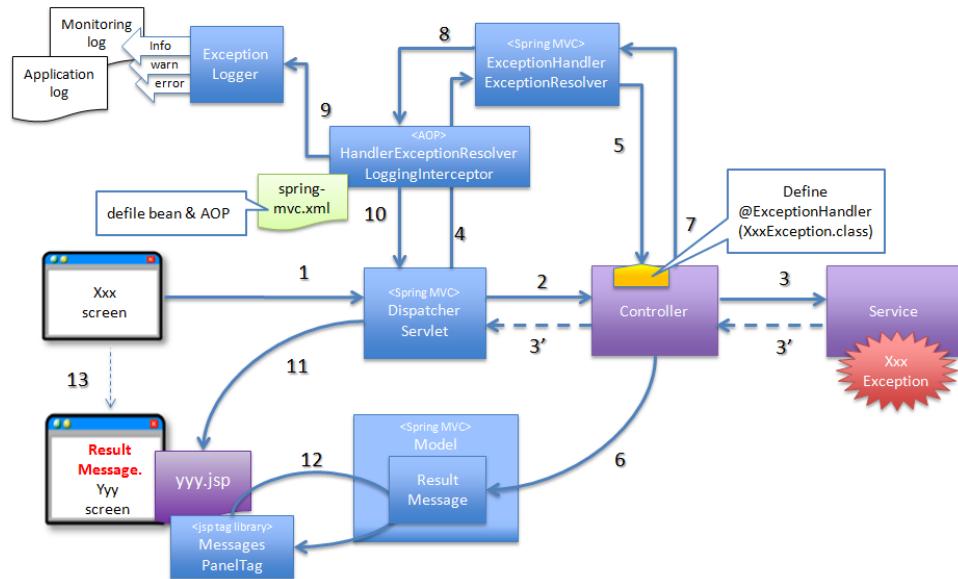


Figure 5.37 Figure - Basic flow when the Controller class handles the exception at use case level

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 common library 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.

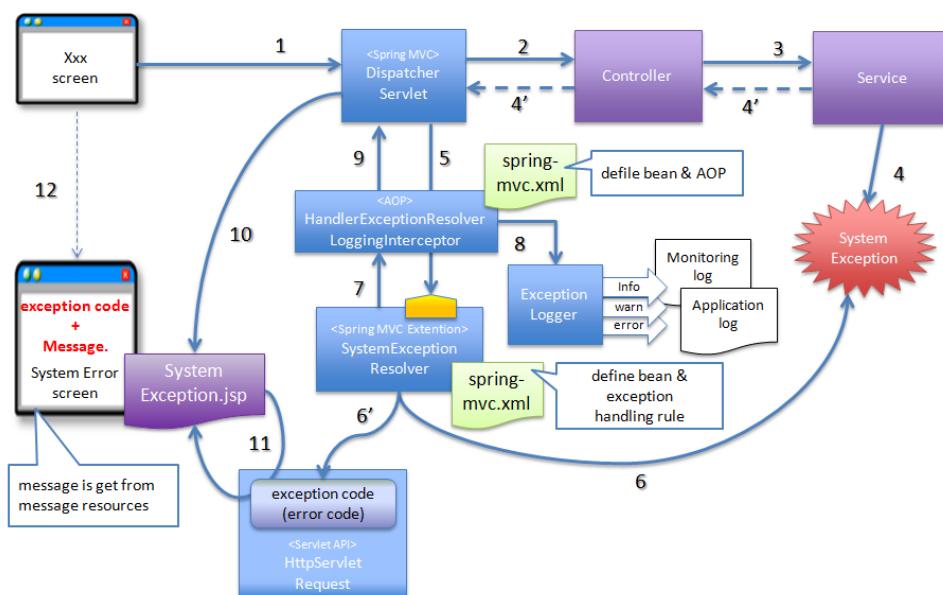


Figure.5.38 Figure - Basic flow when the framework handles the exception at servlet level

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.

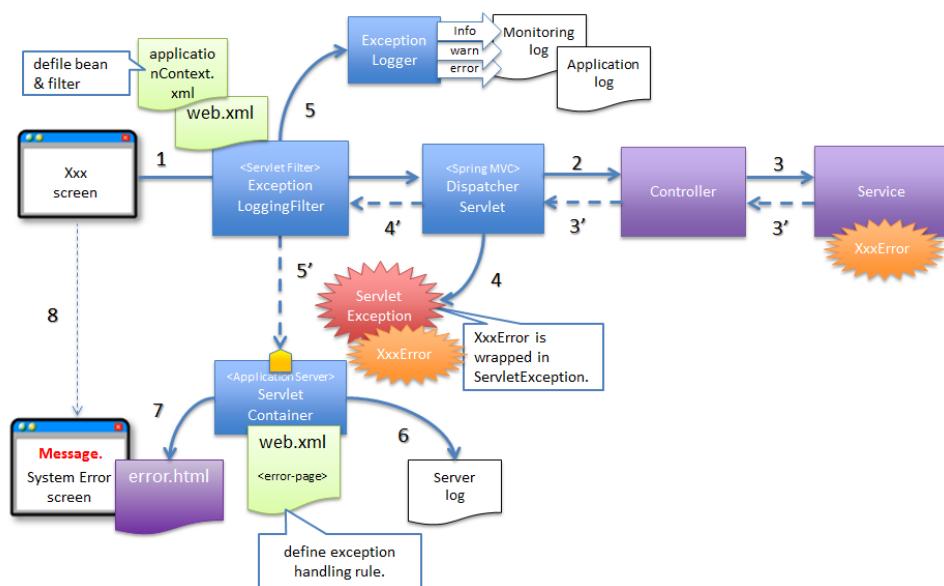


Figure.5.39 Figure - Basic flow when servlet container handles the exception at web application level

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 functionality 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)	<p>Specify mapping between name of the exception to be handled and applicable “Exception Code (Message ID)”.</p> <p>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)”.</p> <hr/> <p>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.</p> <hr/> <p>[Location to be customized for each project]</p>
(3)	<p>Specify default “Exception Code (Message ID)”.</p> <p>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)”.</p>
5.7. Exception Handling	<p>[Location to be customized for each project] 663</p> <hr/> <p>Note: Exception Code (Message ID) Exception code is only to be output to log (It can also be fetched on screen). It is also possible</p>

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]]></pattern>
    </encoder>
</appender>

<logger name="org.terasoluna.gfw.common.exception.ExceptionLogger.Monitoring" additivity="false">
    <level value="error" /> <!-- (3) -->
    <appender-ref ref="MONITORING_LOG_FILE" /> <!-- (4) -->
</logger>
```

Sr. No.	Description
(1)	<p>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.</p> <p>[Location to be customized for each project]</p>
(2)	<p>Specify logger definition for monitoring log. When creating ExceptionLogger, if any logger name is not specified, the above settings can be used as is.</p> <p>Warning: About additivity setting value Specify false. If true is specified, the same log will be output by upper level logger (for example, root).</p>
(3)	<p>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.</p> <p>[Location to be customized for each project]</p>
(4)	<p>Specify the appender which will act as output destination.</p> <p>[Location to be customized for each project]</p>

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}\t]>
</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)	<p>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.</p> <p>[Location to be customized for each project]</p>
(2)	<p>Specify logger definition for application log. When creating ExceptionLogger, if any logger name is not specified, the above settings can be used as is.</p> <hr/> <p>Note: Appender definition for outputting application log</p> <p>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.</p> <hr/>
(3)	<p>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.</p> <p>[Location to be customized for each project]</p>
(4)	<p>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”.</p> <p>[Location to be customized for each project]</p>
5.7. Exception Handling	667

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/transactionTokenError" />
    <entry key=".DataAccessException" value="common/error/dataAccessError" />
  </map>
</property>
<property name="statusCodes"> <!-- (5) -->
  <map>
    <entry key="common/error/resourceNotFoundError" value="404" />
    <entry key="common/error/businessError" value="409" />
    <entry key="common/error/transactionTokenError" value="409" />
    <entry key="common/error/dataAccessError" value="500" />
  </map>
</property>
<property name="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)	<p>Specify the order of priority for handling. The value can be “3”. When <code><mvc:annotation-driven></code> is specified, automatically <i>registered class</i> is given higher priority.</p> <hr/> <p>ヒント: Method to disable exception handling carried out by DefaultHandlerExceptionResolver</p> <p>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 <i>HTTP response code set by DefaultHandlerExceptionResolver</i>.</p> <hr/>
670	<p>(4) 5 Architecture in Detail - TERASOLUNA Global Framework Specify the mapping between name of the exception to be handled and View name.</p> <p>In the above settings, if class name of the exception class (or parent class) includes “.DataAccessException”,</p>

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"> <!-- (1) -->
    <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.ResultMessage 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.ResultMessage. If default settings need to be changed, refer to <i>About HandlerExceptionResolverLoggingInterceptor settings.</i>

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> <!-- (2) -->
</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 <code>ExceptionLoggingFilter</code> defined in <code>applicationContext.xml</code> .
(2)	Specify the filter class. The value should be fixed to <code>org.springframework.web.filter.DelegatingFilter</code>
(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 <code>/ *</code> 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:f94de92148f1489b9ceec3b2f17c969
```

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)	<p>Specify the HTTP Response Code to be handled. [Location to be customized for each project]</p> <p>For HTTP response code for which response is sent using the default exception handling function of Spring MVC, refer to <i>HTTP response code set by DefaultHandlerExceptionResolver</i>.</p>
(2)	<p>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.</p> <p>[Location to be customized for each project]</p>

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)	<p>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.</p> <p>[Location to be customized for each project]</p>

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 occurred.

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 <i>Message Management</i> .
(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=[Res
```

```
// 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

Business Error!

[e.xx.fw.8001] Business error occurred!

- Test example exception

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)	<p>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.</p>
(2)	<p>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.</p>

Application log as shown below is output.

```
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

System Error!

[e.ad.od.9012] System error occurred!

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 occurred. 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 = restTemplate.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 <code>org.terasoluna.gfw.common.exception.ExceptionL</code> 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, <code>log</code> 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 <code>info</code> , <code>warn</code> , <code>error</code> 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 following 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)

System Error!

[e.xx.fw.9010] System error occurred!

- Output Screen (Without exceptionCode)

System Error!

System error occurred!

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 [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.5.17 Table - Classes under org.terasoluna.gfw.common.exception package

Sr. No.	Class	Role
(1)	ExceptionCode Resolver	<p>An interface for resolving exception code (message ID) of exception class.</p> <p>An exception code is used for identifying the exception type and is expected to be output to system error screen and log.</p> <p>It is referenced from <code>ExceptionLogger</code>, <code>SystemExceptionResolver</code>.</p>
(2)	SimpleMapping ExceptionCode Resolver	<p>Implementation class of <code>ExceptionCodeResolver</code> contains the mapping of exception class name and exception code, through which the exception code is resolved.</p> <p>The name of exception class need not be FQCN, it can be a part of FQCN or parent class name.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> Warning: <ul style="list-style-type: none"> • 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. </div>
(3) 5.7. Exception Handling	enums. ExceptionLevel	enum indicating exception level corresponding to exception class. 689 INFO, WARN, ERROR are defined.

Table.5.18 Table - Classes under org.terasoluna.gfw.web.exception package

Sr. No.	Class	Role
(13)	SystemException Resolver	<p>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.</p> <p>It inherits SimpleMappingExceptionResolver provided by Spring MVC and adds the functionality of referencing ResultMessages of exception code from View.</p>
(14)	HandlerException ResolverLogging Interceptor	<p>This is an Interceptor class to output the log of exceptions handled by HandlerExceptionResolver.</p> <p>In this Interceptor class, the output level of log is switched based on the classification of HTTP response code resolved by HandlerExceptionResolver.</p> <ul style="list-style-type: none"> 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. <p>By using this Interceptor, it is possible to output the log of all exceptions which are within the boundary of Spring MVC.</p>
690	5 Architecture in Detail - TERASOLUNA Global Framework	Implementation of the ExceptionLogger.

About SystemExceptionResolver settings

This section describes the settings which are not explained above. The settings should be performed depending on the requirements.

Table.5.19 List of settings not explained above

Sr. No.	Field Name	Property Name	Description	Default Value
(1)	Attribute name of result message	resultMessagesAttribute	<p>Specify the attribute name (String) used for setting message of businessException to Model.</p> <p>This attribute name is used for accessing result message in View(JSP).</p>	resultMessages
(2)	Attribute name of exception code (message ID)	exceptionCode Attribute	<p>Specify the attribute name (String) used for setting the exception code (message ID) to HttpServletRequest.</p> <p>This attribute name is used for accessing the exception code (message ID) in View(JSP).</p>	exceptionCode
(3)	Header name of exception code (message ID)	exceptionCode Header	<p>Specify the header name (String) used for setting the exception code (message ID) to response header of HttpServletResponse.</p>	X-Exception-Code
(4)	Attribute name of exception object	exceptionAttribute	<p>Specify the attribute name (String) used for setting the exception object to Model.</p> <p>This attribute name is used for accessing the exception object.</p>	exception
692	5 Architecture in Detail - TERASOLUNA Global Framework			

(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 “ <code>resultMessagesForExceptionResolver</code> ” in result message attribute name (<code>resultMessagesAttribute</code>).
(2)	Specify attribute name that was set in <code>SystemExceptionResolver</code> , in message attribute name (<code>messagesAttributeName</code>).

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.5.20 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	<p>Amongst the exceptions handled by HandlerExceptionResolverLoggingInterceptor, the exception classes which are not to be logged should be specified in list format.</p> <p>When an exception of the specified exception class and sub class occurs, the same is not logged in this class.</p> <p>Only the exceptions which are logged at a different location (using different mechanism) should be specified here.</p>	ResultMessagesNotificationException, ResultMessagesNotificationException 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</value>
            <!-- (2) -->
            <value>com.example.common.XxxException</value>
        </set>
    </property>
</bean>
```

```

    </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.

TERASOLUNA Global Framework Development Guideline Documentation, Release

1.0.2-SNAPSHOT

Sr. No.	Handled framework exceptions	HTTP Status Code
(1)	org.springframework.web.servlet.mvc.multiaction.NoSuchRequestHandlingMethodException	404
(2)	org.springframework.web.HttpRequestMethodNotSupportedException	405
(3)	org.springframework.web.HttpMediaTypeNotSupportedException	415
(4)	org.springframework.web.HttpMediaTypeNotAcceptableException	406
(5)	org.springframework.web.bind.MissingServletRequestParameterException	400
(6)	org.springframework.web.bind.ServletRequestBindingException	400
(7)	org.springframework.beans.ConversionNotSupportedException	500
(8)	org.springframework.beans.TypeMismatchException	400
(9)	org.springframework.http.converter.HttpMessageNotReadableException	400
(10)	org.springframework.http.converter.HttpMessageNotWritableException	500
(11). 700	org.springframework.http.converter.HttpMessageNotWritableException 5. Architecture in Detail. MERN Stack and Global Framework	400
(12)	org.springframework.web.multipart.support.MissingServletRequestPartException	400

5.8 Session Management

5.8.1 Overview

This chapter explains Session Management in a Web application.

In Web applications, data is exchanged between the client and the server, using HTTP.

HTTP itself does not have a feature to physically maintain a session. Instead, it provides a mechanism wherein, a logical session is maintained by linking a session identifier value (session ID), between the client and the server.

Cookie or request parameter is used to link the session ID between client and server.

The figure below illustrates establishment of logical session.

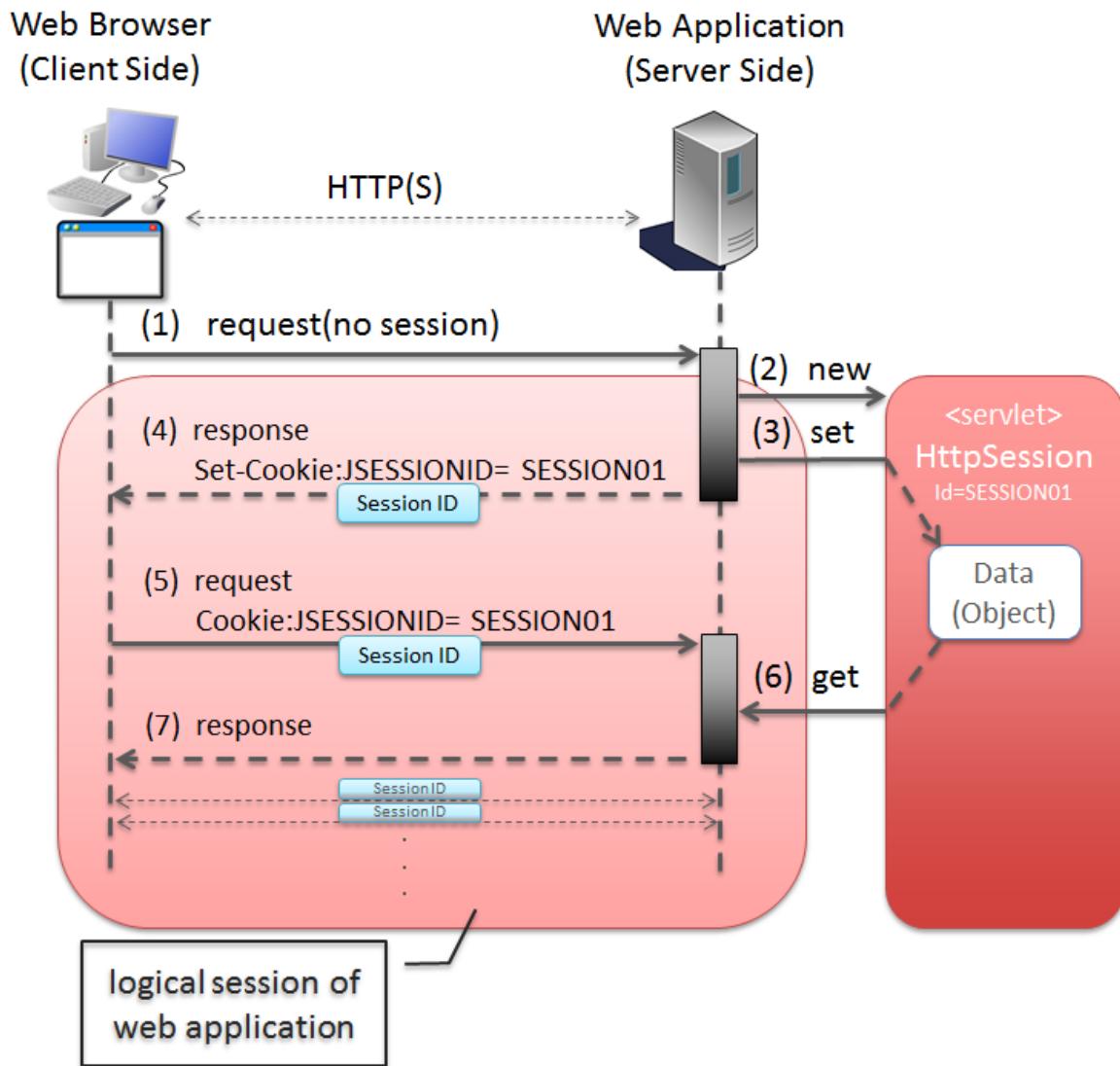


Figure.5.40 Picture - Establishment of logical session

Sr. No.	Description
(1)	Web browser (Client) accesses the Web application (Server) when session is not established.
(2)	Web application creates HttpSession object for storing the session with Web browser. Session ID is issued at the time of HttpSession object creation.
(3)	Web application stores the data sent by the Web browser in HttpSession object.
702	(4) Web application sends a response to the Web browser. By setting "JSESSIONID = Issued session ID" in the "Set-Cookie" header of response, session ID is linked to the Web Browser. Linked session ID is stored in Cookie.

Note: About the parameter name to link session ID

In JavaEE Servlet specifications, the default parameter name to link a session ID is JSESSIONID.

Session lifecycle

Instead of implementing as Controller process, session lifecycle control (create, discard, timeout detection) is implemented by using the processes provided by framework or common library.

Note: "Session" mentioned in the following explanation refers to javax.servlet.http.HttpSession object provided by Servlet API. HttpSession object is the Java object representing the logical session described above.

Generating a session

When Web application is created by the method recommended in this guideline, session can be generated by any one of the following processes.

Sr. No.	Description
1.	<p>Authentication/authorization process provided by Spring Security.</p> <p>Timing and status of session generation can be specified by Spring Security settings.</p> <p>For details on session management in Spring Security, refer to <i>authentication(spring_security)_how_to_use_sessionmanagement</i>.</p>
2.	<p>CSRF token check process provided by Spring Security</p> <p>When a session is already established, new session is not generated.</p> <p>For details on CSRF token check, refer to <i>[coming soon] CSRF(Cross Site Request Forgeries) Countermeasures</i>.</p>
3.	<p>Transaction token check process provided by common library.</p> <p>When a session is already established, new session is not generated.</p> <p>For details on Transaction token check, refer to <i>Double Submit Protection</i>.</p>
4.	<p>Process for passing a model (form object, domain object etc.) to redirect destination request, by using addFlashAttribute method of RedirectAttributes interface.</p> <p>When a session is already established, new session is not generated.</p> <p>For details on RedirectAttributes and Flash scope, refer to <i>Passing data while redirecting request</i>.</p>
5.	<p>Process for storing a model (form object, domain object etc.) in a session, using @SessionAttributes annotation.</p> <p>The specified model (form object, domain object etc.) is stored in session. When a session is already established, new session is not generated.</p> <p>For details on how to use @SessionAttributes annotation, refer to <i>Using @SessionAttributes annotation</i>.</p>
6.	<p>Process that uses session-scoped bean in Spring Framework.</p> <p>When a session is already established, new session is not generated.</p> <p>For details on how to use session-scoped bean, refer to <i>Using session-scoped bean of Spring Framework</i>.</p>

Note: In Sr. Nos. 4, 5, 6 mentioned above, whether the session is used or not is specified by Controller implementation whereas, session generation timing is controlled by the framework. In other words, HttpSession API need not be used directly as Controller process.

Storing attributes in a session

When Web application is created by the methods recommended in this guideline, attributes (objects) are stored in session by any one of the following processes.

Sr. No.	Description
1.	<p>Authentication process provided by Spring Security.</p> <p>Authenticated user information is stored in the session.</p> <p>For details on Spring Security authentication process, refer to [coming soon] Authentication.</p>
2.	<p>CSRF token check process provided by Spring Security.</p> <p>Issued token value is stored in session.</p> <p>For details on CSRF token check, refer to [coming soon] CSRF(Cross Site Request Forgeries) Countermeasures.</p>
3.	<p>Transaction token check process provided by common library.</p> <p>Issued token value is stored in session.</p> <p>For details on Transaction token check, refer to Double Submit Protection.</p>
4.	<p>Process for passing a model (form object, domain object etc.) to redirect destination request, by using addFlashAttribute method of RedirectAttributes interface.</p> <p>Object specified in the addFlashAttribute method of RedirectAttributes interface, is stored in an area called Flash scope in the session.</p> <p>For details on RedirectAttributes and Flash scope, refer to Passing data while redirecting request.</p>
5.	<p>Process for storing a model (form object, domain object etc.) in session, using @SessionAttributes annotation.</p> <p>The specified model (form object, domain object etc.) is stored in session.</p> <p>For the details on how to use @SessionAttributes annotation, refer to, Using @SessionAttributes annotation.</p>
6.	<p>Process that uses session-scoped bean in Spring Framework.</p> <p>Session-scoped bean is stored in session.</p> <p>For details on how to use session-scoped bean, refer to Using session-scoped bean of Spring Framework.</p>

Note: Timing to store the object in session is controlled by the framework. Hence, setAttribute method of HttpSession object is not called as Controller process.

Deleting attributes from a session

When Web application is created by the methods recommended in this guideline, attributes (objects) are deleted from a session by any one of the following processes.

Sr. No.	Description
1.	<p>Logout process provided by Spring Security. Authenticated user information is deleted from the session. For details on Spring Security logout process, refer to [coming soon] Authentication.</p>
2.	<p>Transaction token check process provided by common library. When the value of issued token exceeds the upper limit allocated to NameSpace, token value that is not in use, is deleted from the session. For details on Transaction token check, refer to Double Submit Protection.</p>
3.	<p>Redirect process after the object is stored in Flash scope. Object specified in the addFlashAttribute method of RedirectAttributes interface, is deleted from the Flash scope area of session.</p>
4.	<p>Framework process, after the setComplete method of SessionStatus object is called as Controller process. Object specified by @SessionAttributes annotation is deleted from the session.</p>

Note: Timing to delete the object is controlled by the framework. Hence, removeAttribute method of HttpSession object is not called as Controller process.

Discarding a session

When Web application is created by the methods recommended in this guideline, session is discarded using any one of the following processes.

Sr. No.	Description
1.	Logout process provided by Spring Security. For details on Spring Security logout process, refer to <i>[coming soon] Authentication</i> .
2.	Process for detecting session timeout of application server.

The scenario in which a session is discarded explicitly is illustrated below.

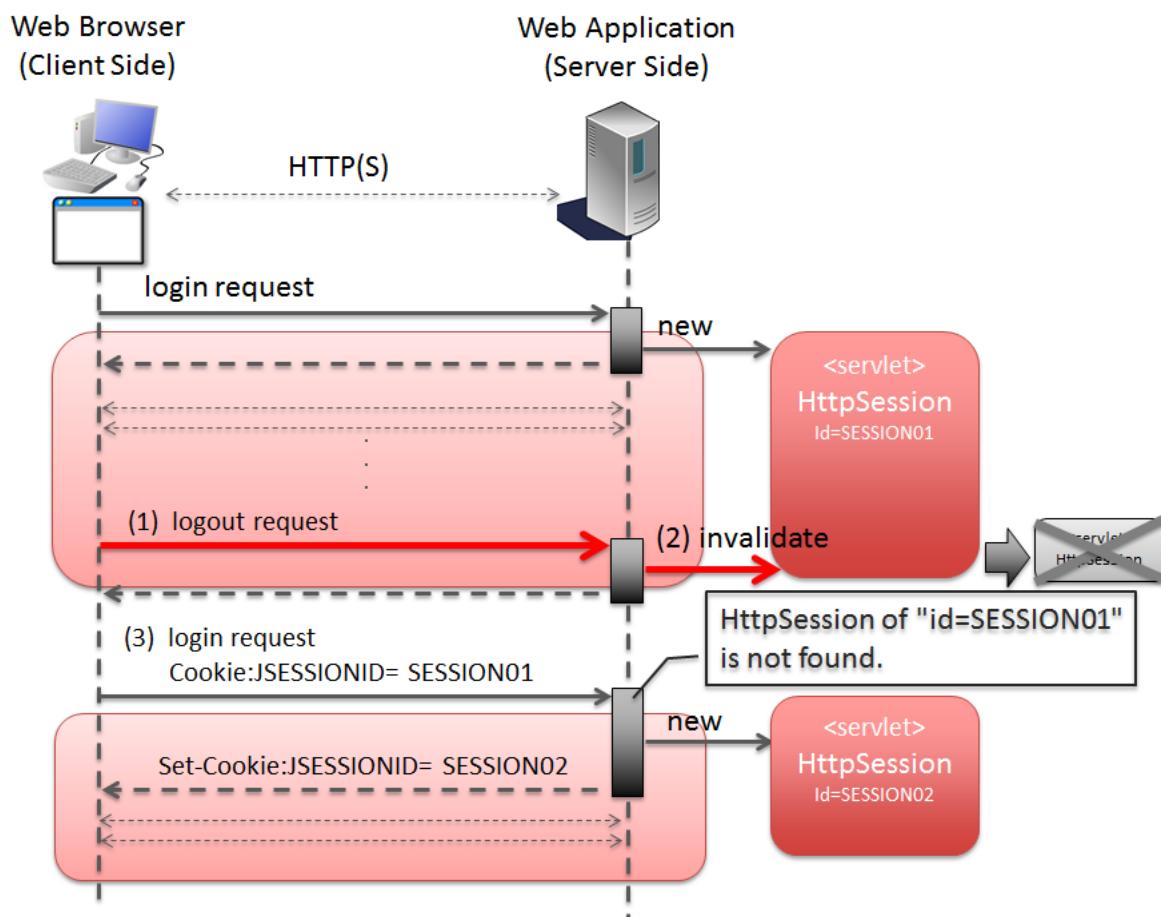


Figure.5.41 Picture - Invalidate session by processing of Web Application

Sr. No.	Description
(1)	<p>Access the process that discards session from Web browser.</p> <p>When using Spring Security, the logout process provided by it is used to discard the session.</p> <p>For details on logout process of Spring Security, refer to [coming soon] Authentication.</p>
(2)	<p>Web application discards HttpSession object corresponding to the session ID linked from Web browser.</p> <p>At this point, HttpSession object with the ID, "SESSION01", disappears from server side.</p>
(3)	<p>When the discarded session is accessed from Web browser using the respective session ID, a new session is generated, as HttpSession object corresponding to the requested session ID does not exist.</p> <p>In the above example, a session with ID "SESSION02" is being generated.</p>

The scenario in which a session is automatically discarded due to timeout is illustrated below.

Sr. No.	Description
(1)	<p>When an established session is not accessed for a particular period, application server detects session timeout.</p>
(2)	<p>Application server discards the session for which session timeout is detected.</p>
(3)	<p>When the session is accessed from Web browser after session timeout occurs, session timeout error is returned to Web browser since HttpSession object corresponding to the session ID requested from the Web browser, does not exist.</p>

Note: Designing session timeout

When data needs to be stored in the session, the design should include ‘session timeout’. It is recommended to set the timeout as short as possible, especially when the data to be stored is large.

Note: Default session timeout period

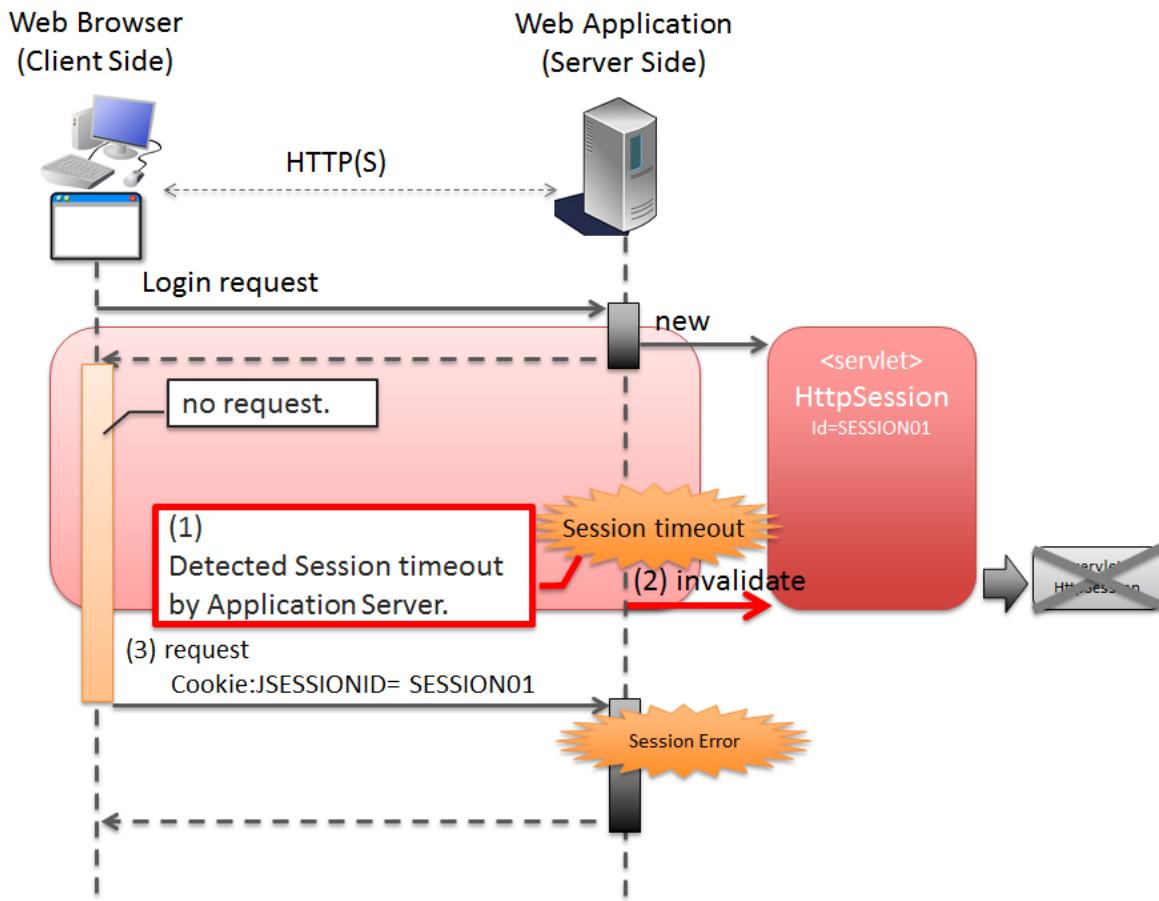


Figure.5.42 Picture - Invalidate session by Application Server

Default session timeout period differs depending on application server.

- Tomcat: 1800 seconds (30 minutes)
- WebLogic: 3600 seconds (60 minutes)
- WebSphere: 1800 seconds (30 minutes)
- Resin: 1800 seconds (30 minutes)

Detecting a request after session timeout

When Web application is created by the method recommended in this guideline, a request subsequent to session timeout is detected by any one of the following processes.

Sr. No.	Description
1.	<p>Session timeout check process provided by Spring Security.</p> <p>Session timeout check is performed as per default settings in Spring Security.</p> <p>Therefore, to store data in session, settings to validate the timeout check process of a Spring Security session, are required.</p> <p>For details on timeout check process in Spring Security, refer to <i>authentication(spring_security)_how_to_use_sessionmanagement</i>.</p>
2.	<p>When not using Spring Security, timeout check process needs to be implemented in Servlet Filter or HandlerInterceptor of Spring MVC.</p>

The scenario in which session timeout is detected using session check process provided by Spring Security, is illustrated below.

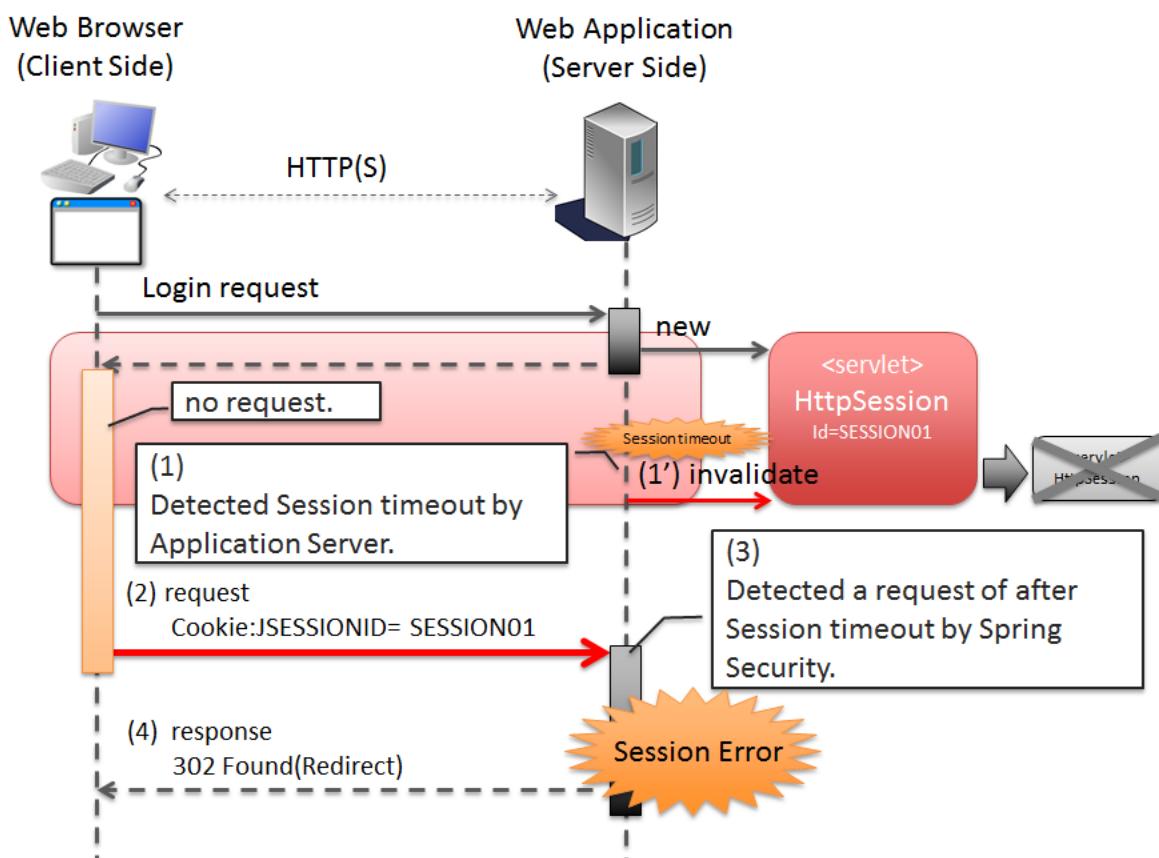


Figure.5.43 Picture - Detected a request after session timeout by Spring Security

Sr. No.	Description
(1)	When an established session is not accessed for a particular period, application server detects session timeout and discards the session.
(2)	Web browser accesses the session after the session timeout occurs.
(3)	Spring Security throws session timeout error as, the <code>HttpSession</code> object corresponding to the session ID linked from client, does not exist. In default Spring Security implementation, response is sent to the request for redirecting to URL to display the error screen.

Note: Necessity of session timeout check

For the processes having precondition, “Data should be stored in session”, session timeout check should always be performed. If this check is not performed, unexpected system errors and operations may occur, as data required by the process cannot be fetched.

Todo

TBD

When POST method is called on enabling CSRF measures provided by Spring Security, CSRF token error is thrown even before session timeout is detected.

Currently, the viewpoint of Spring Security is being inquired in [Spring Security JIRA\(SEC-2422\)](#). As per Spring Security viewpoint, it is necessary to create a servlet filter to detect session timeout. This servlet filter should be defined before defining the ‘servlet filter for CSRF measures’ provided by Spring Security.

About using a session

When data needs to be shared across multiple screens (multiple requests), it can be shared easily by storing this data in session.

However, as against the advantage that the data can be easily shared, storing the data in session also results in application constraints etc. Hence, whether or not to use session should be decided by considering application and system requirements.

Note: Rather than simply storing the data in session, this guideline initially recommends to consider a policy wherein session is not used. Further, if session is used, it recommends storing only the absolutely required data in it.

Note: Storing the data applicable to following conditions has proved better.

- Data that does not provide linkage between use cases, but for such data, the status when it is moved and returned from a different use case needs to be stored.

For example, search condition of a List screen corresponds to this pattern.

When the search condition of a List screen returns from another use case (example: “Change searched data” use case), many a times its status before moving to other use case needs to be stored as a functional requirement.

Search conditions can also be shared in hidden state but this causes excessive dependency between use cases and implementing the application is expected to be complex.

- Data for which linkage between use cases is necessary.

For example, data stored in the shopping site cart corresponds to this pattern.

Data stored in the shopping site cart involves use cases of ” Adding the product to the cart”, ” Displaying the cart”, ” Changing the status of the cart” and ” Purchasing the products in the cart”. It requires the data linkage of all these use cases.

However, when scalability needs to be considered, sometimes it is better to store the data in database instead of session.

Advantages and disadvantages of using session

The advantages and disadvantages of using session are as follows:

- **Advantages**

- Data can be shared across multiple screens (multiple requests) easily when a single process is composed in multiple screens such as a Wizard Screens.
- By storing the fetched data in session, number of executions of data acquisition can be reduced.

- **Disadvantages**

- When the screen with same process is opened in multiple browsers or various tabs, data consistency cannot be maintained, as mutual operations interfere with the data stored in session.
In order to maintain data consistency, control is required so that screen with the same process cannot be opened in multiple browsers or tabs.

This control can be implemented by using Transaction token check provided by common library, however it results in reduced usability.

- Session data is usually stored in application server memory; hence memory usage also increases with increase in data stored in the session.
If unnecessary data that is no longer used in a process is left as it is, it gets excluded from garbage collection process, leading to memory exhaustion. Such data needs to be deleted from session at a stage when it is rendered unnecessary.
The timing to delete unnecessary data from session needs to be designed separately.
- Storing process data in session may lower scalability of application server.

Note: Any one of the following methods is required to scale out the application server.

1. Performing session replication and sharing the session information with all application servers.

When performing session replication, the load on replication process increases in proportion to the data stored in session and number of application servers for replication.

Therefore, issues owing to scale out such as, possible degradation in response time etc. need to be considered.

2. Distributing all requests of the same session to the same application server, using load balancer.

When requests are distributed to the same application server and if the server is down, the process cannot be continued by another application server.

Therefore, it needs to be noted that, this method may not be feasible in applications that demand high availability (service level).

Scale out method should be determined upon considering each of these points.

Advantages and disadvantages of not using session

In order to avoid the disadvantages faced while using session, all the data required for server processing can be implemented by linking as request parameters.

The advantages and disadvantages of not using session are as follows:

- **Advantages**

- As data is not stored at server side, even if multiple browsers and tabs are used, their operations do not interfere with each other. Therefore, multiple screens of the same process can be started without impairing the usability.
- As data is not stored at server side, continuous utilization of memory can be controlled.
- Number of factors that lower the scalability of application server are reduced.

- **Disadvantages**

- Data required for server processing needs to be sent as request parameter. As a result, even the items that are not displayed on the screen need to be specified as hidden items.

Thus, JSP implementation code increases.

This can be minimized by creating JSP tag library.

- Amount of data flowing to the network increases, as all the data required for server processing needs to be sent through requests.
- Data required for screen display needs to be fetched each time. Hence, number of executions of data acquisition increases.

About the data to be stored in session

Following points need to be considered for the data to be stored in session.

- It should be serializable object (object implementing `java.io.Serializable`).
- It should not be a large object that can cause memory exhaustion.

Serializable objects

Data to be stored in the session may be input or output to disk or network under specific conditions.

Therefore the objects need to be serializable.

Cases where data is input/output to disk are as follows:

- When application server is stopped while a session is active, the session and the data stored in it are saved to the disk.

The saved session and data stored in it, are restored with the start of application server.

Support status for this data restoring operation differs depending on application server.

- If session storage area is about to overflow or if the session is not accessed for a particular period of time since the last access, there is a possibility of session swap-out.

The swapped-out session is swapped-in when the session is accessed again.

Conditions etc. for swap-out differ depending on application server.

Cases where data is input/output to network are as follows:

- To perform replication of session in another application server, data stored in the session is sent to that application server via network.

Amount of data to be stored in session

It is recommended that the data to be stored in session should be as compact as possible.

When large amount of data is stored in session, it fatally degrades the performance. Hence, it is recommended to design such that, large amount of data will not be stored in session.

The main causes of performance degradation are as follows:

- When storing large amount of data in session, application server settings need to be enabled such that session is easily swapped-out so as to prevent memory exhaustion.

Swap-out is a “Heavy” process. Hence, it occurring very frequently may affect the overall application performance.

Support status for swap-out operation or setting method, differs depending on application server.

- When carrying out session replication, serialization and deserialization of an object are performed.

Serialization and deserialization of an object with large capacity being “heavy” processes, may affect performance, such as response time.

To make the session data compact, storing the data applicable to the following conditions in request scope instead of session scope, should be considered.

- Read-only data that cannot be edited by screen operations.

If latest data is fetched at a time when it is required, and if the fetched data is displayed in View (JSP) by storing it in request scope, it need not be stored in session.

- Data that can be edited by screen operations. It can be shared only in the screen operations of a use case.

If data can be shared in all screen transitions as hidden HTML fields, it need not be stored in session.

Points to be considered in case of application server clustering

A normal system is rarely composed of a single application server. Considering requirements such as availability, efficiency etc. it consists of multiple servers.

Therefore, when storing the data in session, any one of the following mechanisms needs to be applied in accordance with system requirements.

1. For systems that require high availability (service level), if one AP server is down, it should be possible to continue the processing on another AP server.

To be able to continue processing on another AP server when one AP server is down, session information needs to be shared amongst all AP servers. Hence, session replication needs to be carried out by application servers that are configured as a cluster.

As an alternate method, session information can be shared by setting the session storage location to cache server such as Oracle Coherence or database.

It would be better to consider setting the session storage location to a cache server like Oracle Coherence or database if, number of AP servers, amount of data stored in session and number of sessions that can be pasted simultaneously are in large quantity.

2. For systems that do not require high availability (service level), processing need not be continued on another server if the AP server is down.

Therefore, session information need not be shared amongst all AP servers. Thus, it is alright if all the requests in a same session are distributed to the same AP server, using load balancer functionality.

Warning: When the Web application is created by methods recommended in this guideline, either of the above mechanisms needs to be applied to store the following data in session.

- User information authenticated by the Spring Security authentication process.
- Token values issued by Spring Security CSRF token check.
- Token values issued by the transaction token check provided by common library.

About storage location of session

The session storage location can also be set in the In-memory data grids such as Key-Value Store or Oracle Coherence, other than AP server memory.

There is room for consideration when scalability is required.

The implementation method to change the storage location of session differs according to the AP server and the storage location itself. Therefore, it is omitted in this guideline.

5.8.2 How to use

This guideline recommends using any one of the following methods to store data in session.

1. *Using @SessionAttributes annotation*
2. *Using session-scoped bean of Spring Framework*

Warning: HttpSession API can be called directly by specifying the HttpSession object as an argument of Controller processing method. However, **as a rule, this guideline strongly recommends not to use the HttpSession API directly.**

HttpSession API may be used directly only for those processes that cannot be implemented otherwise. However, in most of the business processes, it is not necessary to use the HttpSession API directly. Therefore, set such that HttpSession object is not specified as an argument of the Controller processing method.

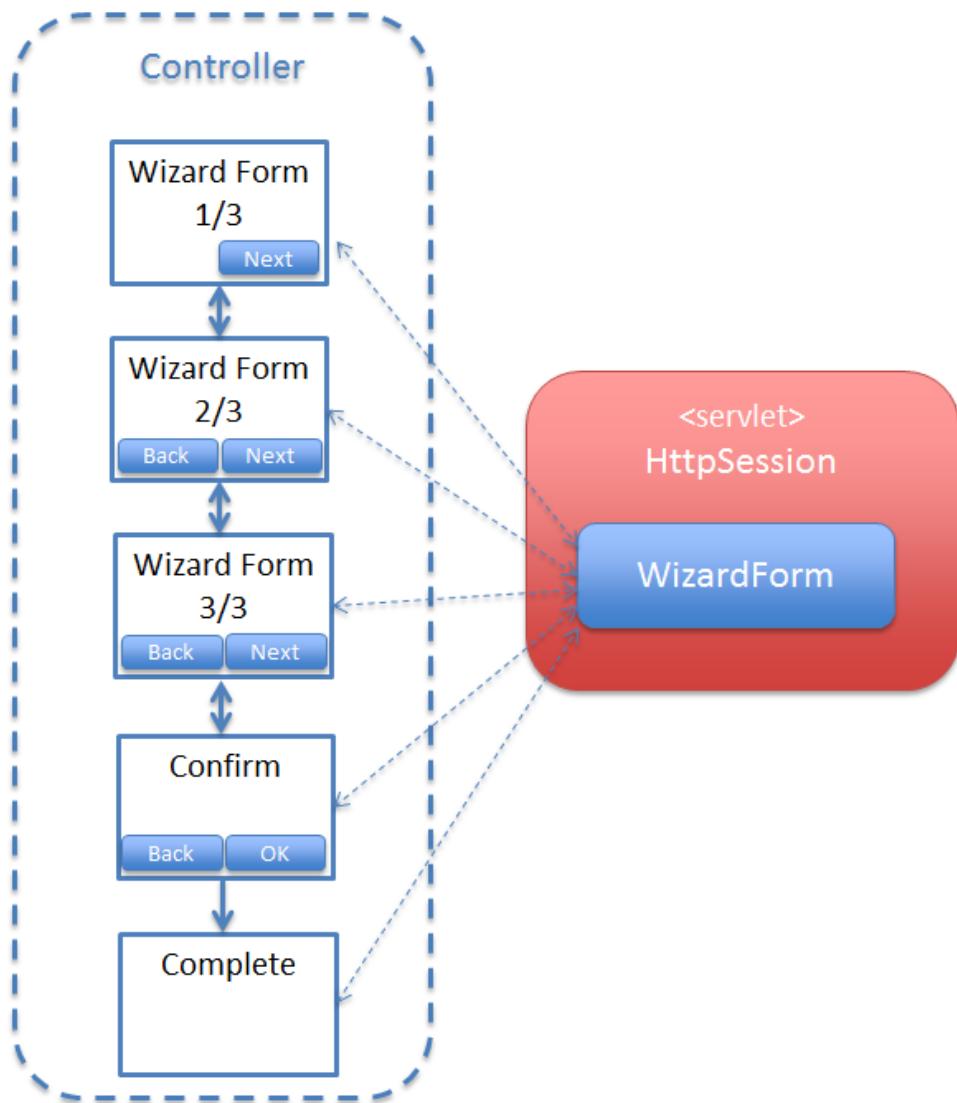
Using @SessionAttributes annotation

@SessionAttributes annotation is used for sharing data during screen transitions in Controller.

However, cases where each of the screens namely, the input screen, confirmation screen and completion screen consists of one page, it is better to share the data using ‘hidden’ HTML field rather than using session.

Cases where the input screen consists of multiple pages or when complicated screen transitions are involved, adopting the method, to store form object in session using `@SessionAttributes` annotation, should be considered.

Application designing and implementation can be simplified by storing the form object in session.



Specifying the object to be stored in session

Specify the object to be stored in session by specifying `@SessionAttributes` annotation in “class”.

```

@Controller
@RequestMapping("wizard")
@SessionAttributes(types = { WizardForm.class, Entity.class }) // (1)
public class WizardController {
    // ...
}

```

Sr. No.	Description
(1)	<p>In “types” attribute of @SessionAttributes annotation, specify the object type to be stored in session.</p> <p>Amongst the objects added to Model object using @ModelAttribute annotation or addAttribute method of Model, the objects matching with the type specified in “types” attribute, are stored in session.</p> <p>In the above example, objects of WizardForm class and Entity class are stored in session.</p>

Note: Management unit for life cycle

Life cycles of the objects stored in session using @SessionAttributes annotation, are managed at Controller level.

When setComplete method of SessionStatus object is called, all the objects specified with “ @ SessionAttribute ” annotation are deleted from session. Therefore, to store the objects having different life cycles in session, it is necessary to divide the Controller.

Warning: Points to be considered when using @SessionAttribute annotation

As described above, life cycle is managed at Controller level. However, when the object with the same attribute name in multiple Controllers, is stored in session using @SessionAttribute annotation, its life cycle is managed across Controllers.

In case of a process wherein, screen can be operated simultaneously by opening another window or tab, it can cause an error as the same object is being accessed. Therefore, to use the class of the same form object in multiple Controllers, first different values (attribute names) should be specified in “value” attribute of @ModelAttributeannotation. Then the same values should be specified in “value” attribute of @SessionAttributes annotation.

The object to be stored in session can also be specified in attribute name.

How to specify is explained below.

```

@Controller
@RequestMapping("wizard")
@SessionAttributes(value = { "wizardcreateForm" }) // (2)

```

```
public class WizardController {  
  
    // ...  
  
    @ModelAttribute(value = "wizardCreateForm")  
    public WizardForm setUpWizardForm() {  
        return new WizardForm();  
    }  
  
    // ...  
}
```

Sr. No.	Description
(2)	<p>Specify the attribute name of the object to be stored in session, in “value” attribute of <code>@SessionAttributes</code> annotation.</p> <p>Among the objects added to <code>Model</code> object using <code>@ModelAttribute</code> annotation or <code>addAttribute</code> method of <code>Model</code>, the objects matching with the attribute name specified in <code>value</code> attribute, are stored in session.</p> <p>In the above example, objects with attribute name "wizardCreateForm" are stored in session.</p>

Adding object to session

Object can be added to session using the following two methods.

- Method with `@ModelAttribute` annotation is used to return the object to be added to the session.
- `addAttribute` method of `Model` object is used to add the object stored in session.

Object added to `Model` object is stored in session as per the type of `@SessionAttributes` annotation and value of “value” attribute. Hence, there is no need to consider the session when implementation is carried out using processing method of Controller.

How to return the object to be stored in session using the method with `@ModelAttribute` annotation, is explained below.

It is recommended to create object using this method when storing form object in session.

```
@ModelAttribute(value = "wizardForm") // (1)  
public WizardForm setUpWizardForm() {  
    return new WizardForm();  
}
```

Sr. No.	Description
(1)	<p>In “value” attribute, specify the attribute name to be stored in <code>Model</code> object.</p> <p>In the above example, the object returned is stored in session, with attribute name "wizardForm".</p> <p>When “value” attribute is specified, method with <code>@ModelAttribute</code> annotation is no longer called by the subsequent requests after the object is stored in session. Thus, it has an advantage wherein, unnecessary objects are not generated.</p>

Warning: Operation in case “value” attribute of `@ModelAttribute` annotation is omitted

When “value” attribute is omitted, method with `@ModelAttribute` annotation is called by all requests in order to generate default attribute name. Therefore, it has a disadvantage of unnecessary objects being generated. Hence, **This method should not be used when storing objects in session.**

```
@ModelAttribute // (1)
public WizardForm setUpWizardForm() {
    return new WizardForm();
}
```

Sr. No.	Description
(1)	<p>Use the method with <code>@ModelAttribute</code> annotation to generate and return the object to be added in session.</p> <p>In the above example, object returned with attribute name "wizardForm" annotation, is stored in session.</p>

How to add object to session using `addAttribute` method of `Model` object is explained below.

When storing the Domain object to session, this method is used to add the object.

```
@RequestMapping(value = "update/{id}", params = "form1")
public String updateForm1(@PathVariable("id") Integer id, WizardForm form,
    Model model) {
    Entity loadedEntity = entityService.getEntity(id);
    model.addAttribute(loadedEntity); // (3)
    beanMapper.map(loadedEntity, form);
    return "wizard/form1";
}
```

Sr. No.	Description
(3)	Add the object to be stored in session using addAttribute method of Model object. In the above example, the object fetched from domain layer with the attribute name "entity" is stored in session.

Fetching the object stored in session

The object stored in session can be received as an argument of Controller processing method.

There is no need to consider the session in Controller processing method, as the object stored session gets stored in Model object as per the attribute value of @SessionAttributes annotation.

```
@RequestMapping(value = "save", method = RequestMethod.POST)
public String save(@Validated({ Wizard1.class, Wizard2.class,
    Wizard3.class }) WizardForm form,    // (1)
    BindingResult result,
    Entity entity,                      // (2)
    RedirectAttributes redirectAttributes) {
    ...
    return "redirect:/wizard/save?complete";
}
```

Sr. No.	Description
(1)	Fetch the object stored in Model object. In the above example, object stored in session scope with attribute name "wizardForm" is passed to argument form. For details on Wizard1.class, Wizard2.class, Wizard3.class specified by @Validated annotation, refer to <i>Example of screen transition implementation using @SessionAttributes in wizard format</i> of Appendix.
(2)	In the above example, object stored in session scope with attribute name "entity", is passed to argument "entity".

When the object to be passed to the argument of Controller processing method does not exist in Model object, the operation changes depending on whether @ModelAttribute annotation is specified or not.

- When @ModelAttribute annotation is not specified, a new object is created and passed as argument. The created object is stored in Model object and subsequently in session as well.

Note: Redirect operations

Created object is not stored in session if it is redirected to a transition destination. Therefore, when referring to the created object in redirect process, it is necessary to store the object in Flash scope using addFlashAttribute method of RedirectAttributes.

- When @ModelAttribute annotation is specified, org.springframework.web.HttpSessionRequiredException occurs.

```
@RequestMapping(value = "save", method = RequestMethod.POST)
public String save(@Validated({ Wizard1.class, Wizard2.class,
    Wizard3.class }) WizardForm form, // (3)
    BindingResult result,
    @ModelAttribute Entity entity, // (4)
    RedirectAttributes redirectAttributes) {
    // ...
    return "redirect:/wizard/save?complete";
}
```

Sr. No.	Description
(3)	<p>Input validation is performed by setting specific verification groups (Wizard1.class, Wizard2.class, Wizard3.class) using @Validated annotation.</p> <p>For details on input validation, refer to Input Validation.</p>
(4)	<p>When @ModelAttribute annotation is specified in argument and when the target object that does not exist in session is called, HttpSessionRequiredException occurs.</p> <p>HttpSessionRequiredException occurs due to client operations such as browser back, directly accessing specified URL etc. Therefore, the exception needs to be handled as client error.</p>

Following are the settings to handle HttpSessionRequiredException as client error.

- spring-mvc.xml

```
<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">
    <property name="exceptionCodeResolver" ref="exceptionCodeResolver" />
    <!-- ... -->
    <property name="exceptionMappings">
        <map>
            <!-- ... -->
            <entry key="HttpSessionRequiredException"
                value="common/error/operationError" /> <!-- (5) -->
```

```
</map>
</property>
<property name="statusCodes">
  <map>
    <!-- ... -->
    <entry key="common/error/operationError" value="400" /> <!-- (6) -->
  </map>
</property>
<!-- ... -->
</bean>
```

Sr. No.	Description
(5)	Add the exception handling definition of HttpSessionRequiredException to exceptionMappings of SystemExceptionResolver provided by common library. In the above example, /WEB-INF/views/common/error/operationError.jsp is specified as the transition destination at the time of exception.
(6)	In statusCodes of SystemExceptionResolver, specify the HTTP response code that is generated when HttpSessionRequiredException occurs. In the above example, Bad Request (400) is specified as the HTTP response code at the time of exception.

- applicationContext.xml

```
<bean id="exceptionCodeResolver"
  class="org.terasoluna.gfw.common.exception.SimpleMappingExceptionCodeResolver">
  <!-- Setting and Customization by project. -->
  <property name="exceptionMappings">
    <map>
      <!-- ... -->
      <entry key="HttpSessionRequiredException" value="w.xx.0003" /> <!-- (7) -->
    </map>
  </property>
  <property name="defaultExceptionCode" value="e.xx.0001" /> <!-- (8) -->
</bean>
```

Sr. No.	Description
(7)	<p>Add the exception handling definition of HttpSessionRequiredException to exceptionMappings of SimpleMappingExceptionCodeResolver provided by common library.</p> <p>In the above example, "w.xx.0003" is specified as the exception code at the time of exception.</p> <p>When this setting is not added, default exception code is output to log.</p>
(8)	Default exception code at the time of exception.

Deleting the object stored in session

To delete the object stored in session using @SessionAttributes, call setComplete method of org.springframework.web.bind.support.SessionStatus from the processing method of Controller.

On calling setComplete method of SessionStatus object, the object specified in attribute value of @SessionAttributes annotation, is deleted from session.

Note: Time when the object is deleted from session

By calling setComplete method of SessionStatus object, the object specified in attribute value of @SessionAttributes annotation is deleted from session. However, the actual time when the object is deleted is different than when setComplete method is called.

setComplete method of SessionStatus object changes only the internal flag, whereas, the actual deletion of object is carried out by the framework, after the processing method of Controller is completed.

Note: Referring object from View (JSP)

Object is deleted from session by calling the setComplete method of SessionStatus object. However, it can be referred from View (JSP) as the same object remains in Model object.

The objects stored in session need to be deleted from the following 3 locations.

- Request to display completion screen. (**Mandatory**)

Delete unnecessary objects, since the objects stored in session are not accessed once completion screen is displayed.

Warning: Reason for deletion

The objects stored in session are outside the scope of garbage collection process. As a result, if unnecessary objects are not deleted, it leads to memory exhaustion. Moreover, storing unnecessary objects in session results in session swap-out which is a heavy process and may affect the overall application performance.

- Request to stop the consecutive screen operations. (**Mandatory**)

Events such as “Go Back to Menu” or “Cancel” etc. to stop the consecutive screen operations, do not access the objects stored in session. Therefore, unnecessary objects should be deleted from session.

- Request for initial display of input screen. (Optional)

Warning: Reason for deletion

When the browser or tab is closed during screen operations, information entered during input, remains in the form object stored in session. As a result, this information is displayed on screen if it is not deleted at the time of initial display. However, in cases where it is alright to display this information on screen, it is not mandatory to delete the information using request for initial display.

Example given below illustrates implementation of object deletion by using a request to display completion screen.

```
// (1)
@RequestMapping(value = "save", method = RequestMethod.POST)
public String save(@ModelAttribute @Validated({ Wizard1.class,
    Wizard2.class, Wizard3.class }) WizardForm form,
    BindingResult result, Entity entity,
    RedirectAttributes redirectAttributes) {
// ...
    return "redirect:/wizard/save?complete"; // (2)
}

// (3)
@RequestMapping(value = "save", params = "complete", method = RequestMethod.GET)
public String saveComplete(SessionStatus sessionStatus) {
    sessionStatus.setComplete(); // (4)
    return "wizard/complete";
}
```

Sr. No.	Description
(1)	Processing method to perform update process.
(2)	Redirect to request (3), in order to display completion screen.
(3)	Processing method to display completion screen.
(4)	Call setComplete method of SessionStatus object and delete the object from session. Display process for View (JSP) is not affected directly as the same object remains in Model object.

Example given below illustrates implementation of object deletion by using a request to stop consecutive operations.

```
// (1)
@RequestMapping(value = "save", params = "cancel", method = RequestMethod.POST)
public String saveCancel(SessionStatus sessionStatus) {
    sessionStatus.setComplete(); // (2)
    return "redirect:/wizard/menu"; // (3)
}
```

Sr. No.	Description
(1)	Processing method to stop the consecutive screen operations.
(2)	Call setComplete method of SessionStatus object and delete the object from session.
(3)	In the above example, user is redirected to Menu Screen.

Example given below illustrates implementation of object deletion by using a request for initial display of input screen.

```
// (1)
@RequestMapping(value = "create", method = RequestMethod.GET)
public String initializeCreateWizardForm(SessionStatus sessionStatus) {
```

```
    sessionStatus.setComplete();           // (2)
    return "redirect:/wizard/create?form1"; // (3)
}

// (4)
@RequestMapping(value = "create", params = "form1")
public String createForm1() {
    return "wizard/form1";
}
```

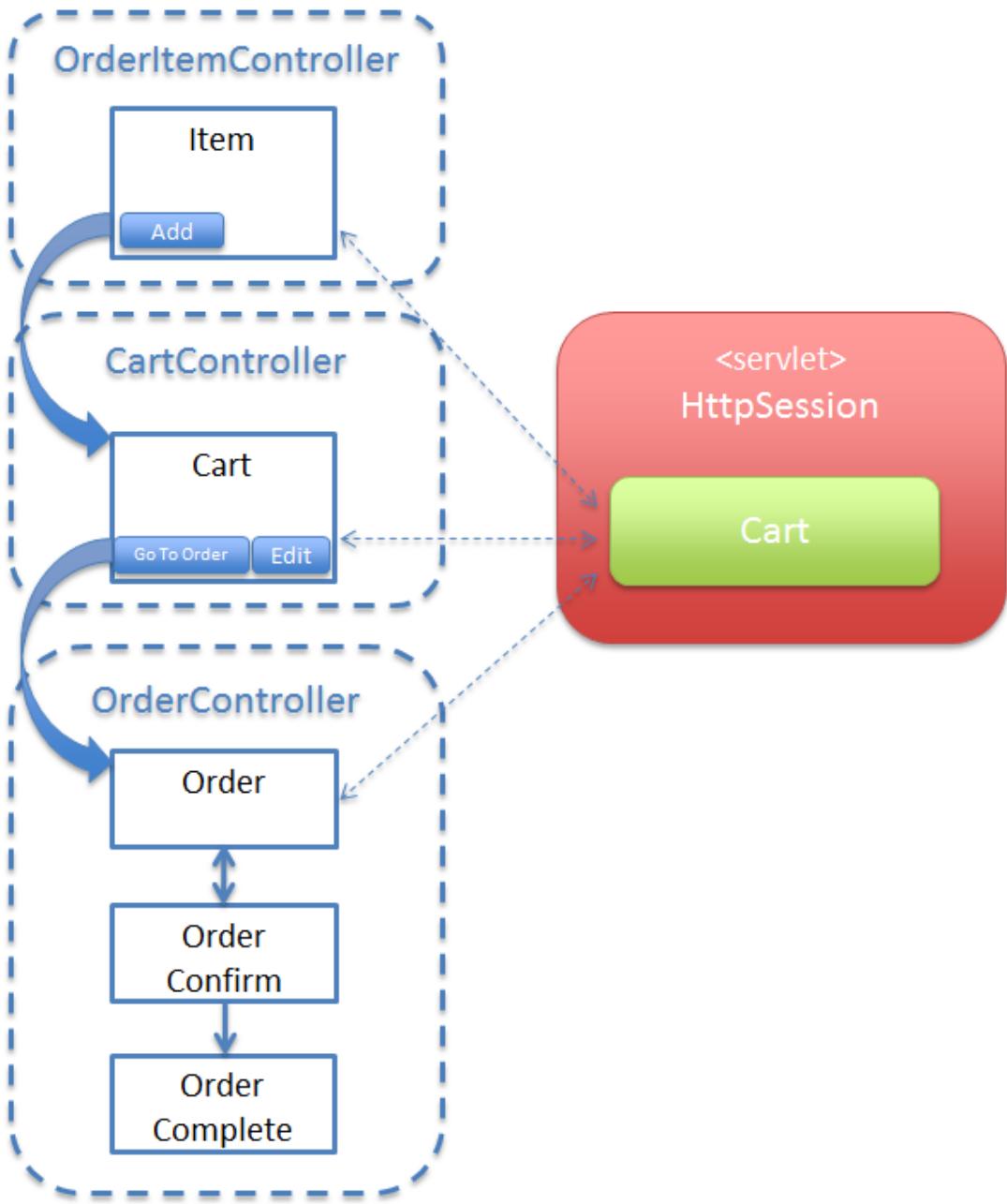
Sr. No.	Description
(1)	Processing method for initial display of input screen.
(2)	Call setComplete method of SessionStatus object.
(3)	Redirect to request (4), that displays input screen. Object is deleted from session by calling the setComplete method of SessionStatus object. However, as the same object remains in Model object, if View (JSP) is called directly, the information being entered is displayed. Therefore, It should be redirected to the request for displaying input screen after deleting from session.
(4)	Processing method to display input screen.

Process implementation using @SessionAttributes

For specific implementation, refer to *Example of screen transition implementation using @SessionAttributes in wizard format* of Appendix.

Using session-scoped bean of Spring Framework

It is recommended to use the session-scoped bean of Spring Framework in order to share the data in screen transition across multiple Controllers.



Bean definition of session scope

Define session-scoped bean of Spring Framework.

There are 2 methods to define the session-scoped bean.

- Define bean using component-scan.
- Define bean in Bean definition file (XML).

How to use component-scan is shown below.

- Class

```
@Component
@Scope("session") // (1)
public class SessionCart implements Serializable {

    private static final long serialVersionUID = 1L;

    private Cart cart;

    public Cart getCart() {
        if (cart == null) {
            cart = new Cart();
        }
        return cart;
    }

    public void setCart(Cart cart) {
        this.cart = cart;
    }
}
```

Sr. No.	Description
(1)	Set Bean scope to "session".

Note: To define Entity class to be handled using JPA as session scope bean, it is recommended to provide a wrapper class instead of defining it directly as session-scoped bean.

If Entity class to be handled using JPA is defined as session-scoped bean, it cannot be handled directly using the API of JPA (throws error if handled directly). Therefore, a process to convert it to Entity object that can be handled in JPA is required.

In the above example, JPA Entity class called `Cart` is wrapped in a wrapper class called `SessionCart` and set as session-scoped bean. With this, the process to convert it to Entity object that can be handled in JPA is no longer required; hence the process to be performed in Controller becomes simple.

- `spring-mvc.xml`

```
<context:component-scan base-package="xxx.yyy.zzz.app"
    scoped-proxy="targetClass" /> // (2)
```

Sr. No.	Description
(2)	Enable scoped-proxy by specifying "targetClass" in the scoped-proxy attribute of <code><context:component-scan></code> element.

Note: Reason for enabling scoped-proxy

scoped-proxy needs to be enabled to inject session-scoped bean in singleton scope Controller.

How to define it in Bean definition file (XML), is shown below.

- JavaBean

```
<beans:bean id="sessionCart" class="xxx.yyy.zzz.app.SessionCart"
    scope="session"> <!-- (3) -->
    <aop:scoped-proxy /> <!-- (4) -->
</beans:bean>
```

Sr. No.	Description
(3)	Set Bean scope to "session".
(4)	Specify “ <code><aop:scoped-proxy /></code> “element and enable scoped-proxy.

Using session-scoped bean

To store and fetch objects in session using session-scoped bean,
inject the session-scoped bean to Controller.

```
@Inject
SessionCart sessionCart; // (1)

// (2)
@ModelAttribute
public SessionCart setUpSessionCart() {
    return sessionCart;
}

@RequestMapping(value = "add")
public String addCart(@Validated ItemForm form, BindingResult result) {
    if (result.hasErrors()) {
        return "item/item";
    }
    CartItem cartItem = beanMapper.map(form, CartItem.class);
```

```
Cart addedCart = cartService.addCartItem(sessionCart.getCart(), // (3)
                                         cartItem);
sessionCart.setCart(addedCart); // (4)
return "redirect:/cart";
}
```

Sr. No.	Description
(1)	Inject session-scoped bean to Controller.
(2)	Add the session-scoped bean to Model object, so that it can be referred from View (JSP).
(3)	On calling the method of session-scoped bean, object stored in session is returned. When there is no object stored in session, newly created object is returned as well as stored in session. In the above example, Service method is called to check inventory etc. before adding to cart.
(4)	In the above example, Cart object passed as an argument of addCartItem method of CartService and the Cart object returned with value, may form a separate instance. Therefore, the returned Cart object is set to session-scoped bean. Session-scoped bean is stored in Model object, by the process explained in (2). As a result, Cart object returned by addCartItem method of CartService can be referred from View (JSP) as well.

Deleting objects stored in session

To delete objects from session using session-scoped bean,
call setComplete method of org.springframework.web.bind.support.SessionStatus from
Controller processing method,
same as when using @SessionAttributes annotation.

To delete objects from session by calling the setComplete method of SessionStatus object,
attribute name of session-scoped bean needs to be specified in the “value” attribute of @SessionAttributes
annotation.

```

@Controller
@RequestMapping("order")
@SessionAttributes("scopedTarget.sessionCart") // (1)
public class OrderController {

    @Inject
    SessionCart sessionCart;

    // ...

    @RequestMapping(method = RequestMethod.POST)
    public String order() {
        // ...
        return "redirect:/order?complete";
    }

    @RequestMapping(params = "complete", method = RequestMethod.GET)
    public String complete(Model model, SessionStatus sessionStatus) {
        sessionStatus.setComplete(); // (2)
        model.addAttribute(sessionCart.getCart()); // (3)
        return "order/complete";
    }
}

```

Sr. No.	Description
(1)	Specify attribute name of session-scoped bean in “value” attribute of <code>@SessionAttributes</code> annotation. Attribute name is "scopedTarget . " + Bean name.
(2)	Call <code>setComplete</code> method of <code>SessionStatus</code> object. In the above example, the object stored with attribute name "scopedTarget.sessionCart", is deleted from session.
(3)	When the object held by session-scoped bean in View (JSP) needs to be referred in View (JSP), the same should be stored in <code>Model</code> object.

Process implementation using session-scoped bean

For a more specific implementation, refer to *Example of screen transition across multiple Controllers using session-scoped bean* of Appendix.

Debug log output of session operations

Class that outputs the operations performed for a session to debug log, is provided by common library.

The log output by this class is effective to check whether session operations are performed as expected.

For details on common library, refer to :ref:*logging_appendix_httpsessioneventlogginglistener*.

Using the JSP implicit object sessionScope

To use JSP implicit object `sessionScope`, the session attribute value of page directive needs to be set to `true`. It is set as `false` in the `include.jsp`, provided by blank project.

`include.jsp` is stored in the `src/main/webapp/WEB-INF/views/common` directory.

- `include.jsp`

```
<%@ page session="true"%>      <%-- (1) --%>  
  
<%-- omitted --%>
```

Sr. No.	Description
(1)	Set the session attribute value of page directive to <code>true</code> .

5.8.3 How to extend

Synchronizing requests in same session

It is recommended to synchronize the requests in the same session in order to use `@SessionAttributes` annotation or session-scoped bean.

If the requests are not synchronized, the object stored in session may be accessed at the same time, causing unexpected errors or operations.

For example, incorrect value may be set for the form object with completed input validation.

To prevent this, it is strongly recommended to set `synchronizeOnSession` of `org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter` to ‘true’ and synchronize the requests in the same session.

It can be implemented by creating BeanPostProcessor as follows and performing bean definition.

- Component

```
package com.example.app.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter;

public class EnableSynchronizeOnSessionPostProcessor
    implements BeanPostProcessor {
    private static final Logger logger = LoggerFactory
        .getLogger(EnableSynchronizeOnSessionPostProcessor.class);

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        // NO-OP
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        if (bean instanceof RequestMappingHandlerAdapter) {
            RequestMappingHandlerAdapter adapter =
                (RequestMappingHandlerAdapter) bean;
            logger.info("enable synchronizeOnSession => {}", adapter);
            adapter.setSynchronizeOnSession(true); // (1)
        }
        return bean;
    }
}
```

Sr. No.	Description
(1)	Requests in the same session can be synchronized by specifying <code>true</code> as an argument of <code>setSynchronizeOnSession</code> method of <code>org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter</code>

- spring-mvc.xml

```
<bean class="com.example.app.config.EnableSynchronizeOnSessionPostProcessor" /> <!-- (2) -->
```

Sr. No.	Description
(2)	Define bean for BeanPostProcessor created in (1).

5.8.4 Appendix

Example of screen transition implementation using @SessionAttributes in wizard format

Screen transition implementation using `@SessionAttributes` annotation in wizard format, is explained below.

Process specifications are as follows:

- Provide screen for registering and updating entity.
- Input screen consists of 3 sub-screens wherein one field is entered on each sub-screen.
- Before saving (registering/updating), the entered values can be verified on confirmation screen.
- Input validation is performed at the time of screen transition. User is returned to input screen in case of error.
- Before saving (registering/updating), perform input validation again for all the input values and display the error screen notifying invalid operations.
- If all the input values are appropriately validated, save the input data to the database.

Basic screen transition is as follows:

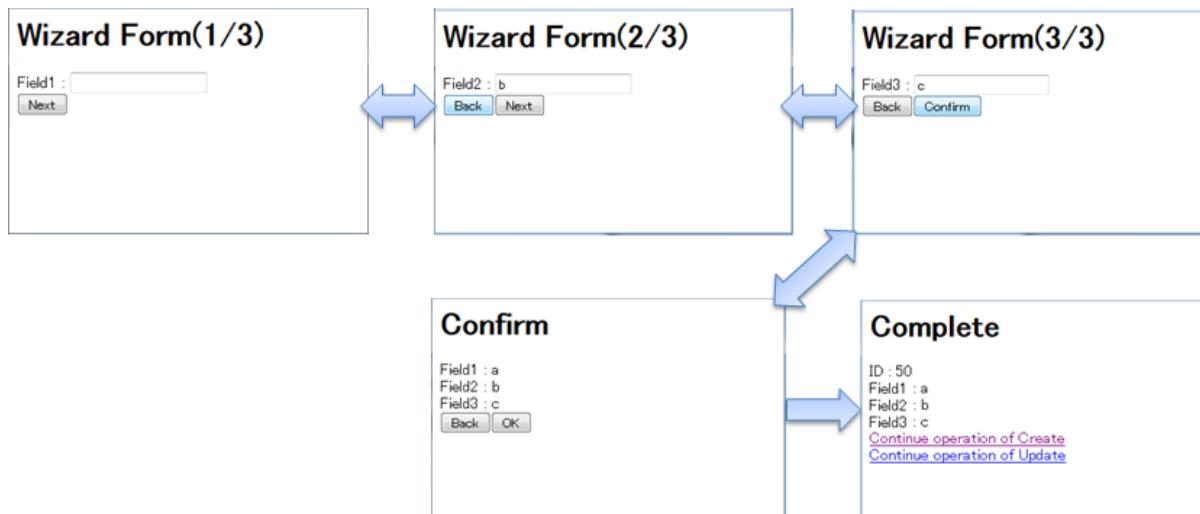
Example of implementation is as follows:

- Form object

```
public class WizardForm implements Serializable {

    private static final long serialVersionUID = 1L;

    // (1)
    @NotEmpty(groups = { Wizard1.class })
    private String field1;
```



```
// (2)
@NotEmpty(groups = { Wizard2.class })
private String field2;

// (3)
@NotEmpty(groups = { Wizard3.class })
private String field3;

// ...

// (4)
public static interface Wizard1 {
}

// (5)
public static interface Wizard2 {
}

// (6)
public static interface Wizard3 {
}

}
```

Sr. No.	Description
(1)	Field to be entered on the first page of input screen.
(2)	Field to be entered on the second page of input screen.
(3)	Field to be entered on the third page of input screen.
(4)	Verification group interface to indicate that it is the field entered on the first page of input screen.
(5)	Verification group interface to indicate that it is the field entered on the second page of input screen.
(6)	Verification group interface to indicate that it is the field entered on the third page of input screen.

Note: Verification group

When performing input validation for screen transition, only the fields on corresponding page need to be validated. In Bean Validation, verification rules can be grouped by setting the interface or class representing the verification group. In this implementation, input validation for each screen is performed by providing verification group for each screen.

- Controller

```
@Controller
@RequestMapping("wizard")
@SessionAttributes(types = { WizardForm.class, Entity.class }) // (7)
public class WizardController {

    @Inject
    WizardService wizardService;

    @Inject
    Mapper beanMapper;
```

Sr. No.	Description
(7)	In the above example, form object (<code>WizardForm.class</code>) and entity (<code>Entity.class</code>) object are stored in session.

```
@ModelAttribute("wizardForm") // (8)
public WizardForm setUpWizardForm() {
    return new WizardForm();
}
```

Sr. No.	Description
(8)	In the above example, form object (<code>WizardForm</code>) to be stored in session is generated. To prevent creation of unnecessary objects, “value” attribute of <code>@ModelAttribute</code> annotation is specified.

```
// (9)
@RequestMapping(value = "create", method = RequestMethod.GET)
public String initializeCreateWizardForm(SessionStatus sessionStatus) {
    sessionStatus.setComplete();
    return "redirect:/wizard/create?form1";
}

// (10)
@RequestMapping(value = "create", params = "form1")
public String createForm1() {
    return "wizard/form1";
}
```

Sr. No.	Description
(9)	Processing method for initial display of input screen for registration. Objects for which operation is in process, may be stored in session; hence such objects are deleted by this processing method.
(10)	Processing method to display the first page of input screen for registration.

```
// (11)
@RequestMapping(value = "{id}/update", method = RequestMethod.GET)
public String initializeUpdateWizardForm(@PathVariable("id") Integer id,
    RedirectAttributes redirectAttributes, SessionStatus sessionStatus) {
    sessionStatus.setComplete();
    redirectAttributes.addAttribute("id", id);
```

```

        return "redirect:/wizard/{id}/update?form1";
    }

// (12)
@RequestMapping(value = "{id}/update", params = "form1")
public String updateForm1(@PathVariable("id") Integer id, WizardForm form,
    Model model) {
    Entity loadedEntity = wizardService.getEntity(id);
    beanMapper.map(loadedEntity, form); // (13)
    model.addAttribute(loadedEntity); // (14)
    return "wizard/form1";
}

```

Sr. No.	Description
(11)	Processing method for initial display of input screen for update.
(12)	Processing method to display input screen for update (first page).
(13)	Set the fetched entity status in form object. In the above example, Bean mapper library called Dozer, is used.
(14)	Add the fetched entity to Model object and store in session. In the above example, it is stored in session with attribute name "entity".

```

// (15)
@RequestMapping(value = "save", params = "form2", method = RequestMethod.POST)
public String saveForm2(@Validated(Wizard1.class) WizardForm form, // (16)
    BindingResult result) {
    if (result.hasErrors()) {
        return saveredoForm1();
    }
    return "wizard/form2";
}

// (17)
@RequestMapping(value = "save", params = "form3", method = RequestMethod.POST)
public String saveForm3(@Validated(Wizard2.class) WizardForm form, // (18)
    BindingResult result) {
    if (result.hasErrors()) {
        return saveredoForm2();
    }
}

```

```

        return "wizard/form3";
    }

// (19)
@RequestMapping(value = "save", params = "confirm", method = RequestMethod.POST)
public String saveConfirm(@Validated(Wizard3.class) WizardForm form, // (20)
    BindingResult result) {
    if (result.hasErrors()) {
        return saveRedoForm3();
    }
    return "wizard/confirm";
}

```

Sr. No.	Description
(15)	Processing method to display second page of input screen.
(16)	Specify the verification group (<code>Wizard1.class</code>) on the first page of input screen, in “value” attribute of <code>@Validated</code> annotation, so as to perform input validation of only the value entered in first page of input screen.
(17)	Processing method to display the third page of input screen.
(18)	Specify the verification group (<code>Wizard2.class</code>) on the second page of input screen, in “value” attribute of <code>@Validated</code> annotation, so as to perform input validation of only the value entered in second page of input screen.
(19)	Processing method to display confirmation screen.
(20)	Specify the verification group (<code>Wizard3.class</code>) on the third page of input screen, in “value” attribute of <code>@Validated</code> annotation, so as to perform input validation of only the value entered in third page of input screen.

```

// (21)
@RequestMapping(value = "save", method = RequestMethod.POST)
public String save(@ModelAttribute @Validated({ Wizard1.class,
    Wizard2.class, Wizard3.class }) WizardForm form, // (22)

```

```
        BindingResult result,
        Entity entity, // (23)
        RedirectAttributes redirectAttributes) {
    if (result.hasErrors()) {
        throw new InvalidRequestException(result); // (24)
    }

    beanMapper.map(form, entity);

    entity = wizardService.saveEntity(entity); // (25)

    redirectAttributes.addFlashAttribute(entity); // (26)

    return "redirect:/wizard/save?complete";
}

// (27)
@RequestMapping(value = "save", params = "complete", method = RequestMethod.GET)
public String saveComplete(SessionStatus sessionStatus) {
    sessionStatus.setComplete();
    return "wizard/complete";
}
```

Sr. No.	Description
(21)	Processing method to execute save process.
(22)	Specify the verification group interface (<code>Wizard1.class</code> , <code>Wizard2.class</code> , <code>Wizard3.class</code>) of each input screen, in the “value” attribute of <code>@Validated</code> annotation, to check all the values entered on input screen.
(23)	Fetch the <code>Entity.class</code> object to be saved. For registration process, newly created object is fetched and for update process, the object stored in session by process (14), is fetched.
(24)	As error does not occur when screen transition is performed using the buttons provided by the application, <code>InvalidRequestException</code> is thrown when an invalid operation is performed. Further, exception class <code>InvalidRequestException</code> needs to be created separately, as it is not provided by common library.
(25)	Save the <code>Entity.class</code> object reflecting the input value.
(26)	<code>Entity.class</code> object saved by processing method of redirect destination is stored in Flash scope so that it can be referred.
(27)	Processing method to display completion screen.

```
// (28)
@RequestMapping(value = "save", params = "redoForm1")
public String saveredoForm1() {
    return "wizard/form1";
}

// (29)
@RequestMapping(value = "save", params = "redoForm2")
```

```
public String saveRedoForm2() {
    return "wizard/form2";
}

// (30)
@RequestMapping(value = "save", params = "redoForm3")
public String saveRedoForm3() {
    return "wizard/form3";
}

}
```

Sr. No.	Description
(28)	Processing method to re-display first page of input screen.
(29)	Processing method to re-display second page of input screen.
(30)	Processing method to re-display third page of input screen.

- Complete Controller source code

```
@Controller
@RequestMapping("wizard")
@SessionAttributes(types = { WizardForm.class, Entity.class })
// (7)
public class WizardController {

    @Inject
    EntityService wizardService;

    @Inject
    Mapper beanMapper;

    @ModelAttribute("wizardForm")
    // (8)
    public WizardForm setUpWizardForm() {
        return new WizardForm();
    }

    // (9)
    @RequestMapping(value = "create", method = RequestMethod.GET)
    public String initializeCreateWizardForm(SessionStatus sessionStatus) {
        sessionStatus.setComplete();
        return "redirect:/wizard/create?form1";
    }
}
```

```
}

// (10)
@RequestMapping(value = "create", params = "form1")
public String createForm1() {
    return "wizard/form1";
}

// (11)
@RequestMapping(value = "{id}/update", method = RequestMethod.GET)
public String initializeUpdateWizardForm(@PathVariable("id") Integer id,
    RedirectAttributes redirectAttributes, SessionStatus sessionStatus) {
    sessionStatus.setComplete();
    redirectAttributes.addAttribute("id", id);
    return "redirect:/wizard/{id}/update?form1";
}

// (12)
@RequestMapping(value = "{id}/update", params = "form1")
public String updateForm1(@PathVariable("id") Integer id, WizardForm form,
    Model model) {
    Entity loadedEntity = wizardService.getEntity(id);
    beanMapper.map(loadedEntity, form); // (13)
    model.addAttribute(loadedEntity); // (14)
    return "wizard/form1";
}

// (15)
@RequestMapping(value = "save", params = "form2", method = RequestMethod.POST)
public String saveForm2(@Validated(Wizard1.class) WizardForm form, // (16)
    BindingResult result) {
    if (result.hasErrors()) {
        return saveRedoForm1();
    }
    return "wizard/form2";
}

// (17)
@RequestMapping(value = "save", params = "form3", method = RequestMethod.POST)
public String saveForm3(@Validated(Wizard2.class) WizardForm form, // (18)
    BindingResult result) {
    if (result.hasErrors()) {
        return saveRedoForm2();
    }
    return "wizard/form3";
}

// (19)
@RequestMapping(value = "save", params = "confirm", method = RequestMethod.POST)
public String saveConfirm(@Validated(Wizard3.class) WizardForm form, // (20)
    BindingResult result) {
```

```
        if (result.hasErrors()) {
            return saveRedoForm3();
        }
        return "wizard/confirm";
    }

// (21)
@RequestMapping(value = "save", method = RequestMethod.POST)
public String save(@ModelAttribute @Validated({ Wizard1.class,
    Wizard2.class, Wizard3.class }) WizardForm form, // (22)
    BindingResult result, Entity entity, // (23)
    RedirectAttributes redirectAttributes) {
    if (result.hasErrors()) {
        throw new InvalidRequestException(result); // (24)
    }

    beanMapper.map(form, entity);

    entity = wizardService.saveEntity(entity); // (25)

    redirectAttributes.addFlashAttribute(entity); // (26)

    return "redirect:/wizard/save?complete";
}

// (27)
@RequestMapping(value = "save", params = "complete", method = RequestMethod.GET)
public String saveComplete(SessionStatus sessionStatus) {
    sessionStatus.setComplete();
    return "wizard/complete";
}

// (28)
@RequestMapping(value = "save", params = "redoForm1")
public String saveRedoForm1() {
    return "wizard/form1";
}

// (29)
@RequestMapping(value = "save", params = "redoForm2")
public String saveRedoForm2() {
    return "wizard/form2";
}

// (30)
@RequestMapping(value = "save", params = "redoForm3")
public String saveRedoForm3() {
    return "wizard/form3";
}
```

- First page of input screen (JSP)

```
<html>
<head>
<title>Wizard Form(1/3) </title>
</head>
<body>
    <h1>Wizard Form(1/3) </h1>
    <form:form action="${pageContext.request.contextPath}/wizard/save"
        modelAttribute="wizardForm">
        <form:label path="field1">Field1</form:label> :
        <form:input path="field1" />
        <form:errors path="field1" />
        <div>
            <form:button name="form2">Next</form:button>
        </div>
    </form:form>
</body>
</html>
```

- Second page of input screen (JSP)

```
<html>
<head>
<title>Wizard Form(2/3) </title>
</head>
<body>
    <h1>Wizard Form(2/3) </h1>
    <%-- (31) --%>
    <form:form action="${pageContext.request.contextPath}/wizard/save"
        modelAttribute="wizardForm">
        <form:label path="field2">Field2</form:label> :
        <form:input path="field2" />
        <form:errors path="field2" />
        <div>
            <form:button name="redoForm1">Back</form:button>
            <form:button name="form3">Next</form:button>
        </div>
    </form:form>
</body>
</html>
```

Sr. No.	Description
(31)	There is no need to hide the input screen fields of first page since the form object is stored in session.

- Third page of input screen (JSP)

```
<html>
<head>
<title>Wizard Form (3/3) </title>
</head>
<body>
    <h1>Wizard Form (3/3) </h1>
    <%-- (32) --%>
    <form:form action="${pageContext.request.contextPath}/wizard/save"
        modelAttribute="wizardForm">
        <form:label path="field3">Field3</form:label> :
        <form:input path="field3" />
        <form:errors path="field3" />
        <div>
            <form:button name="redoForm2">Back</form:button>
            <form:button name="confirm">Confirm</form:button>
        </div>
    </form:form>
</body>
</html>
```

Sr. No.	Description
(32)	There is no need to hide the input screen fields of first and second page since form object is stored in session.

- Confirmation screen (JSP)

```
<html>
<head>
<title>Confirm</title>
</head>
<body>
    <h1>Confirm</h1>
    <%-- (33) --%>
    <form:form action="${pageContext.request.contextPath}/wizard/save"
        modelAttribute="wizardForm">
        <div>
            Field1 : <c:out value="${wizardForm.field1}" />
        </div>
        <div>
            Field2 : <c:out value="${wizardForm.field2}" />
        </div>
        <div>
            Field3 : <c:out value="${wizardForm.field3}" />
        </div>
        <div>
            <form:button name="redoForm3">Back</form:button>
            <form:button>OK</form:button>
        </div>
    </form:form>
</body>
</html>
```

```

</form:form>
</body>
</html>
```

Sr. No.	Description
(33)	There is no need to hide input screen fields since form object is stored in session.

- Completion screen (JSP)

```

<html>
<head>
<title>Complete</title>
</head>
<body>
    <h1>Complete</h1>
    <div>
        <div>
            ID : ${f:h(entity.id)}
        </div>
        <div>
            Field1 : ${f:h(entity.field1)}
        </div>
        <div>
            Field2 : ${f:h(entity.field2)}
        </div>
        <div>
            Field3 : ${f:h(entity.field3)}
        </div>
    </div>
    <div>
        <a href="${pageContext.request.contextPath}/wizard/create">
            Continue operation of Create
        </a>
    </div>
    <div>
        <a href="${pageContext.request.contextPath}/wizard/${entity.id}/update">
            Continue operation of Update
        </a>
    </div>
</body>
</html>
```

- spring-mvc.xml

```

<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">
    <property name="exceptionCodeResolver" ref="exceptionCodeResolver" />
    <!-- ... -->
    <property name="exceptionMappings">
```

```
<map>
<!-- ... -->
<entry key="InvalidRequestException"
       value="common/error/operationError" /> <!-- (34) -->
</map>
</property>
<property name="statusCodes">
  <map>
    <!-- ... -->
    <entry key="common/error/operationError" value="400" /> <!-- (35) -->
  </map>
</property>
<!-- ... -->
</bean>
```

Sr. No.	Description
(34)	<p>Add exception handling definition of <code>InvalidRequestException</code> that notifies detection of invalid request when executing save process, in <code>exceptionMappings</code> of <code>SystemExceptionResolver</code>, provided by common library.</p> <p>In the above example, <code>/WEB-INF/views/common/error/operationError.jsp</code> is specified as the transition destination when exception occurs.</p>
(35)	<p>Specify the HTTP response code when <code>HttpSessionRequiredException</code> occurs in <code>statusCodes</code> of <code>SystemExceptionResolver</code>.</p> <p>In the above example, Bad Request (400) is specified as HTTP response code when exception occurs.</p>

- `applicationContext.xml`

```
<bean id="exceptionCodeResolver"
      class="org.terasoluna.gfw.common.exception.SimpleMappingExceptionCodeResolver">
  <!-- Setting and Customization by project. -->
  <property name="exceptionMappings">
    <map>
      <!-- ... -->
      <entry key="InvalidRequestException" value="w.xx.0004" /> <!-- (36) -->
    </map>
  </property>
  <property name="defaultExceptionCode" value="e.xx.0001" /> <!-- (37) -->
</bean>
```

Sr. No.	Description
(36)	<p>Add exception handling definition of <code>InvalidRequestException</code> to <code>exceptionMappings</code> of <code>SimpleMappingExceptionCodeResolver</code> provided by common library.</p> <p>In the above example, "<code>w.xx.0004</code>" is specified as the exception code when an exception occurs.</p> <p>When this setting is not added, default exception code is output to log.</p>
(37)	Default exception code in case of an exception.

Example of screen transition across multiple Controllers using session-scoped bean.

Implementation using session-scoped bean is explained with example of process of performing screen transitions across multiple Controllers.

Process specifications are as follows:

- Provide a process to add products to cart.
- Provide a process to change the quantity of products added to the cart.
- Provide a process to order the products stored in cart.
- Above 3 processes are provided as independent functions and should be set in separate Controllers (ItemController, CartController, OrderController).
- Store the cart in session as it is shared by the above 3 processes.
- Display the cart screen when products are added to cart.

Screen transition should be as follows:

Implementation is as follows:

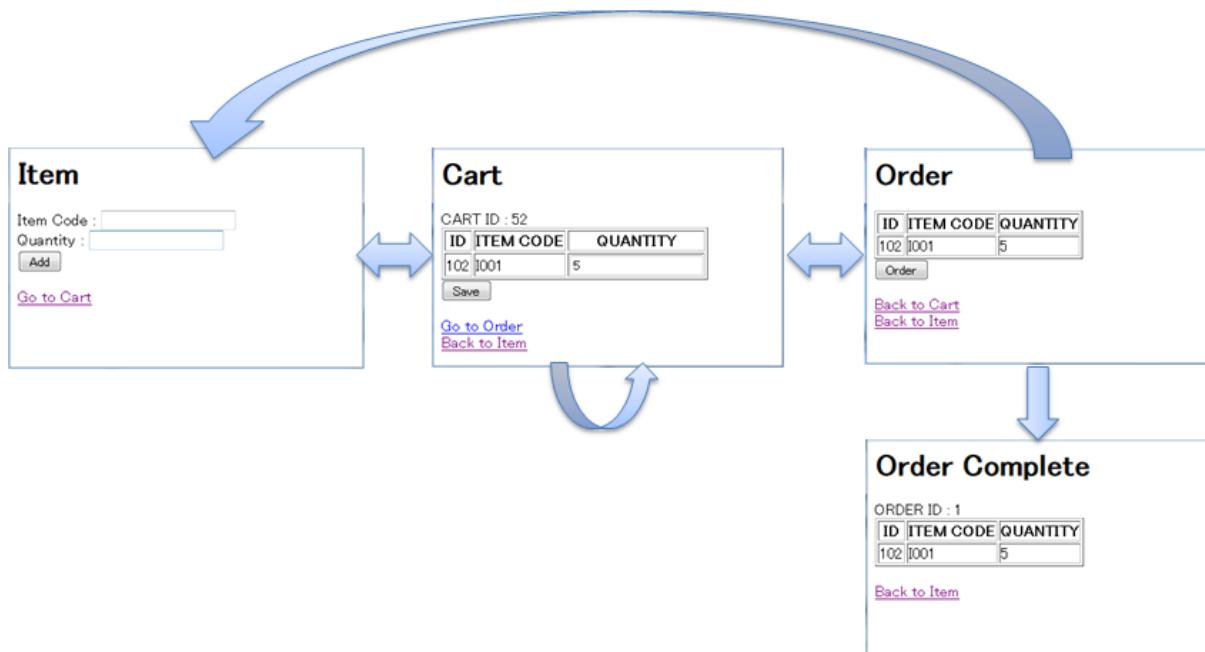
- JavaBean to be defined as session-scoped bean.

```

@Component
@Scope("session")
public class SessionCart implements Serializable {

    private static final long serialVersionUID = 1L;

```



```

private Cart cart; // (1)

public Cart getCart() {
    if (cart == null) {
        cart = new Cart();
    }
    return cart;
}

public void setCart(Cart cart) {
    this.cart = cart;
}
}

```

Sr. No.	Description
(1)	Wrap the Entity (Domain object) called, Cart.

- ItemController

```

@Controller
@RequestMapping("item")
public class ItemController {

    @Inject
    SessionCart sessionCart;

    @Inject
    
```

```
CartService cartService;

@Inject
Mapper beanMapper;

@ModelAttribute
public ItemForm setUpItemForm() {
    return new ItemForm();
}

// (2)
@RequestMapping
public String view(Model model) {
    return "item/item";
}

// (3)
@RequestMapping(value = "add")
public String addCart(@Validated ItemForm form, BindingResult result) {
    if (result.hasErrors()) {
        return "item/item";
    }
    CartItem cartItem = beanMapper.map(form, CartItem.class);
    Cart cart = cartService.addCartItem(sessionCart.getCart(), // (4)
                                         cartItem);
    sessionCart.setCart(cart); // (5)
    return "redirect:/cart"; // (6)
}
}
```

Sr. No.	Description
(2)	Processing method to display product screen.
(3)	Processing method to add the specified products to cart.
(4)	Pass the <code>Cart</code> object stored in session, to service method.
(5)	Reflect <code>Cart</code> object returned by Service method to session-scoped bean. By reflecting in session-scoped bean, it is reflected in session and <code>Model</code> object.
(6)	Redirect to the request to display cart screen after adding products to cart. When transitioning to a separate Controller screen, instead of calling View (JSP) directly, it is recommended to redirect to the request for displaying the screen.

- CartController

```
@Controller
@RequestMapping("cart")
public class CartController {

    @Inject
    SessionCart sessionCart;

    @Inject
    CartService cartService;

    @Inject
    Mapper beanMapper;

    @ModelAttribute
    public CartForm setUpCartForm() {
        return new CartForm();
    }

    // (7)
    @ModelAttribute("sessionCart")
    public SessionCart setUpSessionCart() {
        return sessionCart;
    }
}
```

```
}

// (8)
@RequestMapping
public String cart(CartForm form) {
    beanMapper.map(sessionCart.getCart(), form);
    return "cart/cart";
}

// (9)
@RequestMapping(params = "edit", method = RequestMethod.POST)
public String edit(@Validated CartForm form, BindingResult result,
    Model model) {
    if (result.hasErrors()) {
        return "cart/cart";
    }

    Cart cart = sessionCart.getCart();
    Iterator<CartItemForm> itemForm = form.getCartItems().iterator();
    for (CartItem item : cart.getCartItems()) {
        beanMapper.map(itemForm.next(), item);
    }

    cart = cartService.saveCart(cart);
    sessionCart.setCart(cart); // (10)

    return "redirect:/cart"; // (11)
}

}
```

Sr. No.	Description
(7)	Add to Model object to refer in View (JSP).
(8)	Processing method to display cart screen (quantity change screen).
(9)	Processing method to change quantity.
(10)	Reflect Cart object returned by Service method, to session-scoped bean. By reflecting in session-scoped bean, it is reflected in session and Model object.
(11)	Redirect to the request to display cart screen (quantity change screen) after changing quantity. In case of Update process, instead of calling View (JSP) directly, it is recommended to redirect to the request for displaying the screen.

- OrderController

```

@Controller
@RequestMapping("order")
@SessionAttributes("scopedTarget.sessionCart")
public class OrderController {

    @Inject
    SessionCart sessionCart;

    @ModelAttribute
    public OrderForm setUpOrderForm() {
        return new OrderForm();
    }

    // (12)
    @ModelAttribute("sessionCart")
    public SessionCart setUpSessionCart() {
        return sessionCart;
    }

    // (13)
    @RequestMapping
    public String view() {

```

```

        return "order/order";
    }

    // (14)
    @RequestMapping(method = RequestMethod.POST)
    public String order() {
        ...
        return "redirect:/order?complete";
    }

    // (15)
    @RequestMapping(params = "complete", method = RequestMethod.GET)
    public String complete(Model model, SessionStatus sessionStatus) {
        sessionStatus.setComplete();
        return "order/complete";
    }
}

```

Sr. No.	Description
(12)	Add to Model object to refer in View (JSP).
(13)	Processing method to display Order screen.
(14)	Processing method to place an Order.
(15)	Processing method to display Order Completion screen.

- Product screen (JSP)

```

<html>
<head>
<title>Item</title>
</head>
<body>
    <h1>Item</h1>
    <form:form action="${pageContext.request.contextPath}/item/add"
               modelAttribute="itemForm">
        <form:label path="itemCode">Item Code</form:label> :
        <form:input path="itemCode" />
        <form:errors path="quantity" />
        <br>

```

```
<form:label path="quantity">Quantity</form:label> :
<form:input path="quantity" />
<form:errors path="quantity" />
<div>
    <%-- (15) --%>
    <form:button>Add</form:button>
</div>
</form:form>
<div>
    <a href="${pageContext.request.contextPath}/cart">Go to Cart</a>
</div>
</body>
</html>
```

Sr. No.	Description
(15)	Button to add a product.

- Cart screen (JSP)

```
<html>
<head>
<title>Cart</title>
</head>
<body>
    <h1>Cart</h1>
    <c:choose>
        <c:when test="${ empty sessionCart.cart.cartItems }">
            <div>Cart is empty.</div>
        </c:when>
        <c:otherwise>
            CART ID :
            ${f:h(sessionCart.cart.id)}
            <form:form modelAttribute="cartForm">
                <table border="1">
                    <thead>
                        <tr>
                            <th>ID</th>
                            <th>ITEM CODE</th>
                            <th>QUANTITY</th>
                        </tr>
                    </thead>
                    <tbody>
                        <c:forEach var="item"
                            items="${sessionCart.cart.cartItems}"
                            varStatus="rowStatus">
                            <tr>
                                <td>${f:h(item.id)}</td>
                                <td>${f:h(item.itemCode)}</td>
```

```

<td>
    <form:input
        path="cartItems[${rowStatus.index}].quantity" />
    <form:errors
        path="cartItems[${rowStatus.index}].quantity" />
</td>
</tr>
</c:forEach>
</tbody>
</table>
<%-- (16) --%>
<form:button name="edit">Save</form:button>
</form:>
</c:otherwise>
</c:choose>
<c:if test="${ not empty sessionCart.cart.cartItems }">
    <div>
        <%-- (17) --%>
        <a href="${pageContext.request.contextPath}/order">Go to Order</a>
    </div>
</c:if>
<div>
    <a href="${pageContext.request.contextPath}/item">Back to Item</a>
</div>
</body>
</html>

```

Sr. No.	Description
(16)	Button to update quantity.
(17)	Link to display Order screen.

- Order screen (JSP)

```

<html>
<head>
<title>Order</title>
</head>
<body>
    <h1>Order</h1>
    <table border="1">
        <thead>
            <tr>
                <th>ID</th>
                <th>ITEM CODE</th>
                <th>QUANTITY</th>

```

```
</tr>
</thead>
<tbody>
<c:forEach var="item" items="${sessionCart.cart.cartItems}"
    varStatus="rowStatus">
<tr>
    <td>${f:h(item.id)}</td>
    <td>${f:h(item.itemCode)}</td>
    <td>${f:h(item.quantity)}</td>
</tr>
</c:forEach>
</tbody>
</table>
<form:form modelAttribute="orderForm">
    <%-- (18) --%>
    <form:button>Order</form:button>
</form:form>
<div>
    <a href="${pageContext.request.contextPath}/cart">Back to Cart</a>
</div>
<div>
    <a href="${pageContext.request.contextPath}/item">Back to Item</a>
</div>
</body>
</html>
```

Sr. No.	Description
(18)	Button to place an order.

- Order Completion screen (JSP)

```
<html>
<head>
<title>Order Complete</title>
</head>
<body>
    <h1>Order Complete</h1>
    ORDER ID :
    ${f:h(order.id)}
    <table border="1">
        <thead>
            <tr>
                <th>ID</th>
                <th>ITEM CODE</th>
                <th>QUANTITY</th>
            </tr>
        </thead>
        <tbody>
```

```
<c:forEach var="item" items="${order.orderItems}">
    varStatus="rowStatus">
    <tr>
        <td>${f:h(item.id)}</td>
        <td>${f:h(item.itemCode)}</td>
        <td>${f:h(item.quantity)}</td>
    </tr>
</c:forEach>
</tbody>
</table>
<br>
<div>
    <a href="${pageContext.request.contextPath}/item">Back to Item</a>
</div>
</body>
</html>
```

5.9 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 also recommended to define a error 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

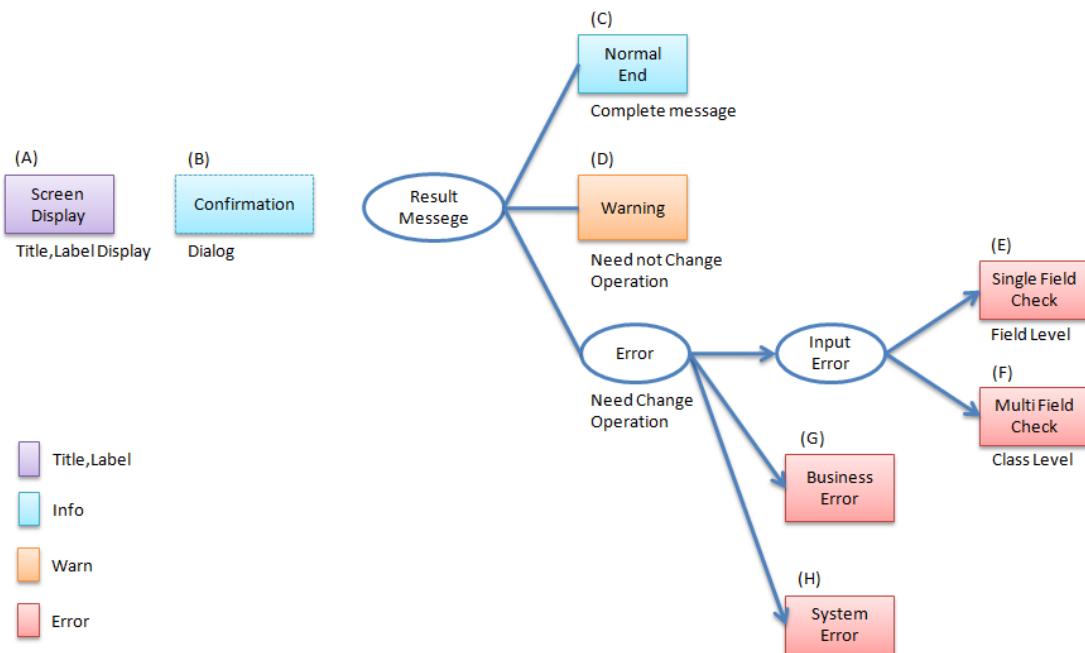
Message that display according to the result of the user's screen operation should be classified into following 3 types depending on the message contents.

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 waring to be focused on; 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	-	<ul style="list-style-type: none"> Employee Registration screen
	Label	Screen field name Report field name Comment Guidance		<ul style="list-style-type: none"> User name Password
(B)	Dialog	Confirmation message	info	<ul style="list-style-type: none"> Are you sure you want to register? Are you sure you want to delete?
(C)	Result message	Successful completion	info	<ul style="list-style-type: none"> Registered. Deleted.
(D)		Warning	warn	<ul style="list-style-type: none"> Password is about to expire. Please change the password. Server is busy. Please try again later.
(E)		Single field validation error	error	<ul style="list-style-type: none"> “User name” is mandatory. Please enter “Name” within 20 characters. Please enter the “Amount” in number.
(F)		Correlation check error	error	<ul style="list-style-type: none"> “Password” and “Confirm Password” do not match.
(G)		Business error	error	<ul style="list-style-type: none"> Failed to cancel the reservation as cancellation period has elapsed. Failed to register as number of allowed registrations exceeded.
(H)		System error	error	<ul style="list-style-type: none"> 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 [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	

Note: When including the field name into validation error message, define messages as follows.

- model attribute name of form + ":" + field name

```
staffForm.staffName = Staff name
```

- filed name

```
staffName = Staff name
```

- 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.
```

Note: 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 password.  
# 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 [Internationalization](#).

Display of result messages

`org.terasoluna.gfw.common.message.ResultMessages` and `org.terasoluna.gfw.common.message.ResultMessage` are provided in common library, as classes storing the result messages which indicate success or failure of process at server side.

Class name	Description
<code>ResultMessages</code>	Class having list of result messages and message type. List of Result messages is expressed in terms of <code>List<ResultMessage></code> and message type is expressed in terms of <code>org.terasoluna.gfw.common.message.ResultMessageType</code> interface.
<code>ResultMessage</code>	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: <code>ResultMessages.error().add(ResultMessage.fromCode("e.ex.an.9001"));</code> 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:

Result Message

- There are inconsistencies in the data.

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 <div> 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("There 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: #ffffce;
    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 />`

- There are inconsistencies in the data.

- Example wherein `ResultMessages.warn().add("w.ex.an.2001")` is output using
`<t:messagesPanel />`

- The recommended change interval has passed password. Please change your password.

- Example wherein `ResultMessages.info().add("i.ex.an.0001", "XXXX")` is output using
`<t:messagesPanel />`

- XXXX upload completed.

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:

Result Message

Messages1

- The recommended change interval has passed password. Please change your password.

Messages2

- There are inconsistencies in the data.

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 <code>getType</code> 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.5.21 <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	<p>Flag for disabling HTML escaping.</p> <p>By setting the flag to true, HTML escaping is no longer performed for the message to be output.</p> <p>This attribute is used to create different message styles by inserting HTML into the message to be output.</p> <p>When the flag is set to true, XSS vulnerable characters should not be included in the message.</p> <p>This attribute can be used with terasoluna-gfw-web 1.0.1.RELEASE or higher version.</p>	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 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="password">Password:</label><input
        type="password" id="password" name="j_password">
</div>
<div>
    <input type="submit" value="Login" />
</div>
</fieldset>
</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 messageType 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:

The screenshot shows a red rectangular alert box with a white background and black text, containing the message "Bad credentials". Below the alert box is a login form with a light gray background. The form has a title "Login Form" and two input fields: "Username:" and "Password:", each with a corresponding text input box. Below the input fields is a "Login" button.

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/application.properties");
        BufferedReader br = new BufferedReader(new InputStreamReader(inputStream));
        Class<?> targetClazz = MessageKeys.class;
        File output = new File("src/main/java/" + targetClazz.getName().replaceAll(Pattern.quote("."), "/") + ".java");
        System.out.println("write " + output.getAbsolutePath());
```

```
PrintWriter pw = new PrintWriter(FileUtils.openOutputStream(output));

try {
    pw.println("package " + targetClazz.getPackage().getName() + ";");
    pw.println("/**");
    pw.println(" * Message Id");
    pw.println(" */");
    pw.println("public class " + targetClazz.getSimpleName() + " {");

    String line;
    while ((line = br.readLine()) != null) {
        String[] vals = line.split("=", 2);
        if (vals.length > 1) {
            String key = vals[0].trim();
            String value = vals[1].trim();
            pw.println("    /** " + key + "=" + value + " */");
            pw.println("    public static final String "
                + key.toUpperCase().replaceAll(Pattern.quote("."), "_")
                .replaceAll(Pattern.quote("-"), "_")
                + " = \\\"" + key + "\\\";");
        }
    }
    pw.println("}");
    pw.flush();
} finally {
    IOUtils.closeQuietly(br);
    IOUtils.closeQuietly(pw);
}
}
```

3. Provision of message properties file

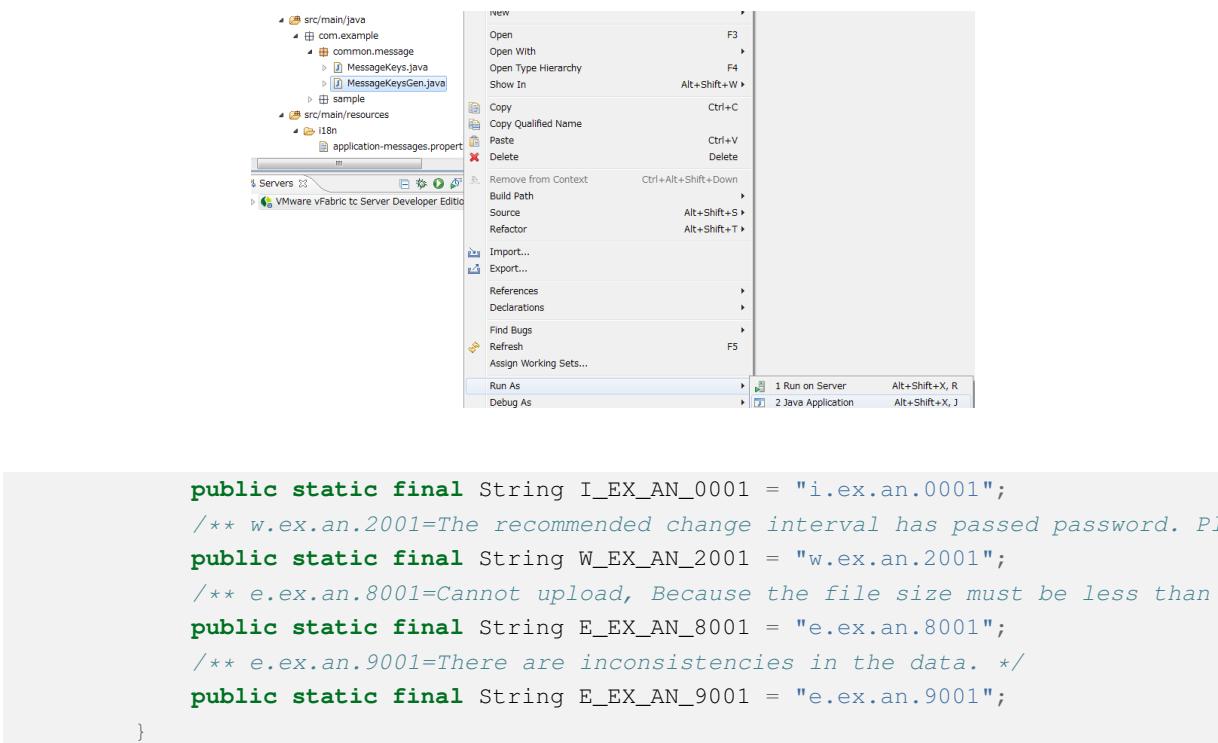
Define the messages in src/main/resource/i18n/application-messages.properties. The settings are carried out as follows:

```
i.ex.an.0001={0} upload completed.
w.ex.an.2001=The recommended change interval has passed password. Please change your password.
e.ex.an.8001=Cannot upload, Because the file size must be less than {0}MB.
e.ex.an.9001=There are inconsistencies in the data.
```

4. 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. */
```



5.10 Properties Management

5.10.1 Overview

This section explains how to manage properties.

Value that needs to be managed as properties can be classified into following two categories.

Sr. No.	Classification	Description	Example
1.	Environment dependent setting	The setting needs to be changed according to the environment on which the application is running. It depends on non-functional requirements such as system configuration.	<ul style="list-style-type: none"> • Database connection information (connection URL, connection user, password, etc) • Destination of the file storage (directory path etc) • more ...
2.	Application settings	The settings that can be customize the application behavior. It depends on functional requirements.	<ul style="list-style-type: none"> • Password valid days • Reservation period days • more ...

Note: In this guideline, it is recommended to manage these settings as properties (properties file).

If an application is mechanized such a way that acquires setting from the properties, there is no need to re-build the application even if there is any changes in these values. Therefore it is possible to release the tested application on production environment.

About how to release the tested application, refer to “[Removing Environment Dependency](#)”.

Tip: Values that are managed as properties can be acquired from JVM system properties (-D option) or OS environment variables. About access order, refer to “[How to use](#)”.

Values that are managed as properties can be used at the following two locations.

- Bean definition file
- Java classes managed by DI container

5.10.2 How to use

About properties file definition

Properties file values in Java class and bean definition file can be accessed by defining `<context:property-placeholder/>` tag in bean definition file.

`<context:property-placeholder/>` tag reads the group of specified properties files and can fetch values for properties files key `xxx` specified in `${xxx}` format in `@Value` annotation or bean definition files.

Note: When specified in `${xxx:defaultValue}` format and when key `xxx` is not set in properties file, `defaultValue` is used.

See the method below for defining a properties file

bean definition file

- `applicationContext.xml`
- `spring-mvc.xml`

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

Sr. No.	Description
(1)	<p>In location, set the resource location path.</p> <p>Multiple paths separated by comma can be specified in location attribute.</p> <p>By performing above settings, read properties file under META-INF/spring directory of class path.</p> <p>Once these settings are done, just add the properties file under META-INF/spring.</p> <p>For details on location value, see Reference.</p>

Note: `<context:property-placeholder>` needs to be defined in both `applicationContext.xml` and `spring-mvc.xml`.

Properties are accessed in the following order by default.

1. System properties of active JVM
2. Environment variables
3. Application definition properties

As per default setting, properties file defined in application is searched and read after all environment related properties (JVM system properties and environment variables) are read.

Read sequence can be changed by setting local-override attribute of
`<context:property-placeholder/>` tag to true.

By performing these settings, the properties defined in application are enabled with higher priority.

bean definition file

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

Sr. No.	Description
(1)	Access properties in the following order when local-override attribute is set to true. 1. Application definition properties 2. System properties of active JVM 3. Environment variables

Note: Normally the above settings are sufficient. When multiple `<context:property-placeholder/>` tags are specified, read order can be defined by setting order attribute value.

bean definition file

```
<context:property-placeholder
    location="classpath:/META-INF/property/extendPropertySources.properties"
    order="1" ignore-unresolvable="true" /> <!-- (1) -->
<context:property-placeholder
    location="classpath*/META-INF/spring/*.properties"
    order="2" ignore-unresolvable="true" /> <!-- (2) -->
```

Sr. No.	Description
(1)	<p>By setting the order attribute to a value which is less than (2), properties file corresponding to location attribute is read before (2).</p> <p>When a key overlapping with the key in properties file read in (2) exists, value fetched in (1) is given preference.</p> <p>By setting ignore-unresolvable attribute to true, error which occurs when key exists only in properties file of (2) can be prevented.</p>
(2)	<p>By setting the order attribute to value greater than (1), properties file corresponding to location attribute is read after (1).</p> <p>When a key overlapping with the key in properties file read in (1) exists, value fetched in (1) is set.</p> <p>By setting ignore-unresolvable attribute to true, error which occurs when key exists only in properties file of (1) can be prevented.</p>

Using properties in bean definition file

See the example below of datasource configuration file.

In the following example, it is assumed that properties file definition (
`<context:property-placeholder/>`) is specified.

Basically, property value can be set by setting properties file key in bean definition file using `{} placeholder`.

Properties file

```
database.url=jdbc:postgresql://localhost:5432/shopping
database.password=postgres
```

```
database.username=postgres
database.driverClassName=org.postgresql.Driver
```

bean definition file

```
<bean id="dataSource"
      destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"
              value="${database.driverClassName}" />  <!-- (1) -->
    <property name="url" value="${database.url}" />  <!-- (2) -->
    <property name="username" value="${database.username}" />  <!-- (3) -->
    <property name="password" value="${database.password}" />  <!-- (4) -->
    <!-- omitted -->
</bean>
```

Sr. No.	Description
(1)	By setting \${database.driverClassName}, the value for read properties file key database.driverClassName gets substituted.
(2)	By setting \${database.url}, the value for read properties file key database.url gets substituted.
(3)	By setting \${database.username}, the value for read properties file key database.username gets substituted.
(4)	By setting \${database.password}, the value for read properties file key database.password gets substituted.

As a result of reading the properties file key, the values are replaced as follows:

```
<bean id="dataSource"
    destroy-method="close"
    class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="org.postgresql.Driver"/>
    <property name="url"
        value="jdbc:postgresql://localhost:5432/shopping"/>
    <property name="username" value="postgres"/>
    <property name="password" value="postgres"/>
    <!-- omitted -->
</bean>
```

Using properties in Java class

It is possible to use properties in Java class by specifying @Value annotation in the field wherein properties values are to be stored.

To use @Value annotation, the corresponding object needs to be stored in DI container of Spring.

In the following example, it is assumed that properties file definition (

<context:property-placeholder/>) is specified.

External reference is possible by adding @Value annotation to variables and setting properties file key in value using \${ } placeholder.

Properties file

```
item.upload.title=list of update file
item.upload.dir=file:/tmp/upload
item.upload.maxUpdateFileNum=10
```

Java class

```
@Value("${item.upload.title}") // (1)
private String uploadTitle;

@Value("${item.upload.dir}") // (2)
private Resource uploadDir;
```

```

@Value("${item.upload.maxUpdateFileNum}") // (3)
private int maxUpdateFileNum;

// Getters and setters omitted

```

Sr. No.	Description
(1)	By setting \${item.upload.title} to @Value annotation value, the value for read properties file key item.upload.title gets substituted. uploadTitle is substituted by “list of update file” in String class.
(2)	By setting \${item.upload.dir} to @Value annotation value, the value for read properties file key item.upload.dir gets substituted. org.springframework.core.io.Resource object created with initial value “/tmp/upload” is stored in uploadDir.
(3)	By setting \${item.upload.maxUpdateFileNum} to @Value annotation value, the value for read properties file key item.upload.maxUpdateFileNum gets substituted. maxUpdateFileNum is substituted by 10.

Warning: There could be cases wherein properties values are to be used in static methods of Utility classes etc.; however properties value cannot be fetched using @Value annotation in classes for which bean definition is not done. In this case, it is recommended to create Helper class with @Component annotation and to fetch properties values using @Value annotation. (This class needs to be included in the component-scan scope.) Classes in which values from properties file is to be used, should not be made Utility classes.

5.10.3 How to extend

Extension of method for fetching properties values is explained below. This can be achieved by extending org.springframework.context.support.PropertySourcesPlaceholderConfigurer class.

The example below illustrates a case wherein encrypted properties file is used.

Decrypting encrypted values and using them

To strengthen security, properties file needs to be encrypted in some cases.

The example below illustrates decryption of encrypted properties values. (No specific encrypting and decrypting methods are mentioned.)

bean definition file

- applicationContext.xml
- spring-mvc.xml

```
<!-- (1) -->
<bean class="com.example.common.property.EncryptedPropertySourcesPlaceholderConfigurer">
    <!-- (2) -->
    <property name="locations"
        value="classpath*:META-INF/spring/*.properties" />
</bean>
```

Sr. No.	Description
(1)	Define the extended PropertySourcesPlaceholderConfigurer instead of <context:property-placeholder/>. <context:property-placeholder/> tag should be deleted.
(2)	Set “locations” in name attribute of property tag and specify the path of the properties file to be read, in value attribute. The method of specifying path of the properties file to be read is same as <i>About properties file definition</i> .

Java class

- Extended PropertySourcesPlaceholderConfigurer

```
public class EncryptedPropertySourcesPlaceholderConfigurer extends
    PropertySourcesPlaceholderConfigurer { // (1)
    @Override
    protected void doProcessProperties(
        ConfigurableListableBeanFactory beanFactoryToProcess,
        StringValueResolver valueResolver) { // (2)
        super.doProcessProperties(beanFactoryToProcess,
```

```

        new EncryptedValueResolver(valueResolver)); // (3)
    }
}

```

Sr. No.	Description
(1)	Inherited PropertySourcesPlaceholderConfigurer, should extend org.springframework.context.support.PropertySourcesPlaceholderConfigurer class.
(2)	Override doProcessProperties method of org.springframework.context.support.PropertySourcesPlaceholderConfigurer class.
(3)	Call doProcessProperties of parent class; however, use valueResolver(EncryptedValueResolver) valueResolver wherein valueResolver is implemented independently. In EncryptedValueResolver class, decrypt when encrypted values of properties file are fetched.

- EncryptedValueResolver.java

```

public class EncryptedValueResolver implements
    StringValueResolver { // (1)

    private final StringValueResolver valueResolver;

    EncryptedValueResolver(StringValueResolver stringValueResolver) { // (2)
        this.valueResolver = stringValueResolver;
    }

    @Override
    public String resolveStringValue(String strVal) { // (3)

        // Values obtained from the property file to the naming
        // as seen with the encryption target
        String value = valueResolver.resolveStringValue(strVal); // (4)
    }
}

```

```
// Target messages only, implement coding
if (value.startsWith("Encrypted:")) { // (5)
    value = value.substring(10); // (6)
    // omitted decryption
}
return value;
}
```

Sr. No.	Description
(1)	Inherited EncryptedValueResolver should implement org.springframework.util.StringValueResolver.
(2)	When EncryptedValueResolver class is created in constructor, set StringValueResolver inherited from EncryptedPropertySourcesPlaceholderConfigurer.
(3)	Override resolveStringValue method of org.springframework.util.StringValueResolver. If the values fetched from properties file are encrypted, these must be decrypted in resolveStringValue method. The process mentioned in steps (5) and (6) is just an example, the process differs depending on type of implementation.
(4)	The value is being fetched by specifying key as an argument of resolveStringValue method of StringValueResolver set in constructor. This value is defined in properties file.
(5)	Check whether values of properties file are encrypted. The method to determine whether the values are encrypted differs depending on type of implementation. Here, the value can be considered encrypted if it starts with “Encrypted:”. If the values are encrypted, decrypt them in step (6) and if they are not encrypted, return them as is.
(6)	Encrypted values of properties file are being decrypted. (No specific decryption process is mentioned.) Decryption method differs depending on type of implementation.

- Helper to fetch properties

```
@Value("${encrypted.property.string}") // (1)
private String testString;

@Value("${encrypted.property.int}") // (2)
private int testInt;

@Value("${encrypted.property.integer}") // (3)
private Integer testInteger;

@Value("${encrypted.property.file}") // (4)
private File testFile;

// Getters and setters omitted
```

Sr. No.	Description
(1)	By setting \${encrypted.property.string} to @Value annotation value, the value for read properties file key encrypted.property.string is decrypted and then substituted. Value decrypted in String class is substituted in testString.
(2)	By setting \${encrypted.property.int} to @Value annotation value, the value for read properties file key encrypted.property.int is decrypted and then substituted. Value decrypted in integer type is substituted in testInt.
(3)	By setting \${encrypted.property.integer} to @Value annotation value, the value for read properties file key encrypted.property.integer is decrypted and then substituted. Value decrypted in Integer class is substituted in testInteger.
(4)	By setting \${encrypted.property.file} to @Value annotation value, the value for read properties file key encrypted.property.file is decrypted and then substituted. In testFile, File object is stored as initial value which is created using the decrypted value (auto conversion).

Properties file

The values encrypted as properties values are prefixed with “Encrypted:” to indicate that they are encrypted. Although one can view the contents of properties file, but cannot understand them as the values are encrypted.

```
encrypted.property.string=Encrypted:ZlpbQRJRWlNAU1FGV0ASRVteXhJQVxJXXFFAS0JGV1Yc
encrypted.property.int=Encrypted:AwI=
encrypted.property.integer=Encrypted:AwICAgI=
encrypted.property.file=Encrypted:YkBdQldARkt/U1xTVVdfV1xGHFpGX14=
```

5.11 Pagination

5.11.1 Overview

This chapter describes the Pagination functionality wherein the data matching the search criteria is divided into pages and displayed.

It is recommended to use pagination functionality when a large amount of data matches the search criteria.

Retrieving and displaying a large amount of data at a time on screen may lead to following problems.

- Memory exhaustion at server side

`java.lang.OutOfMemoryError` occurs when multiple requests are executed simultaneously.

- Network load

Transferring unnecessary data over the network results in increased network load and thereby may affect the response time of the entire system.

- Delay in response on screen

Server process, network traffic process and client rendering process take time to handle a large amount of data; hence the response on screen may get delayed.

Display of list screen at the time of dividing into pages

When a page is divided using pagination, the screen looks as follows:

The screenshot shows a web browser window titled "Article Search" with the URL "localhost:8080/terasoluna-gfw-web-blank/article/list?word=title&_csrf=c36b". The page displays a table of 10 articles with columns: No, Class, Title, Overview, and Published Date. A search bar at the top contains the word "title". Below the table is a pagination control with links for <<, <, 1, 2, 3, 4, 5, 6, >, and >>. A message below the control says "60 results (0.012 seconds) 1 / 6 Pages". Two callout boxes point to specific elements: (1) points to the pagination links, and (2) points to the pagination information message.

No	Class	Title	Overview	Published Date
1	Internal	Internal title1_1	overview1	2013-10-03
2	Internal	Internal title1_2	overview2	2013-10-04
3	Internal	Internal title1_3	overview3	2013-10-05
4	Internal	Internal title1_4	overview4	2013-10-06
5	Internal	Internal title1_5	overview5	2013-10-07
6	Internal	Internal title1_6	overview6	2013-10-08
7	Internal	Internal title1_7	overview7	2013-10-09
8	Internal	Internal title1_8	overview8	2013-10-10
9	Internal	Internal title1_9	overview9	2013-10-11
10	Internal	Internal title1_10	overview10	2013-10-12

<< < 1 2 3 4 5 6 > >>

60 results (0.012 seconds)
1 / 6 Pages

(1)
Pagination Links

(2)
Pagination Information

Sr. No.	Description
(1)	<p>Display the link to navigate to various pages.</p> <p>On clicking link, send a request to display the corresponding page. JSP tag library to display this area is provided as common library.</p>
(2)	<p>Display the information related to pagination (total records, total pages and number of displayed pages etc.).</p> <p>Tag library to display this area does not exist; hence it should be implemented separately as JSP processing.</p>

Page search

For implementing pagination, it is essential to first implement the server-side search processing to make page searching possible.

It is assumed in this guideline that the mechanism provided by Spring Data is used for page search at server side.

Page search functionality of Spring Data

Page search functionality provided by Spring Data is as follows:

Sr. No.	Description
1	<p>Extract the information required for page search (location of page to be searched, number of records to be fetched and sort condition) from request parameter and pass the extracted information as objects of <code>org.springframework.data.domain.Pageable</code> to the argument of Controller.</p> <p>This functionality is provided as <code>org.springframework.data.web.PageableHandlerMethodArgumentResolver</code> class and is enabled by adding to <code><mvc:argument-resolvers></code> element of <code>spring-mvc.xml</code>.</p> <p>For request parameters, refer to “<i>Note column</i>”.</p>
2	<p>Save the page information (total records, data of corresponding page, location of page to be searched, number of records to be fetched and sort condition).</p> <p>This functionality is provided as <code>org.springframework.data.domain.Page</code> interface and <code>org.springframework.data.domain.PageImpl</code> is provided as default implementation class.</p> <p>As per specifications, it fetches the required data from Page object in JSP tag library to output pagination link provided by common library.</p>
3	<p>When Spring Data JPA is used for database access, the information of corresponding page is returned as <code>Page</code> object by specifying <code>Pageable</code> object as an argument of Repository Query method.</p> <p>All the processes such as executing SQL to fetch total records, adding sort condition and extracting data matching the corresponding page are carried out automatically.</p> <p>When Mybatis2 is used for database access, the process that is automatically carried out in Spring Data JPA needs to be carried out in Java or SQL mapping file.</p>

Note: Request parameters for page search

Request parameters for page search provided by Spring Data are as follows:

Sr. No.	Parameter name	Description
1.	page	<p>Request parameter to specify the location of page to be searched</p> <p>Specify value greater than or equal to 0.</p> <p>As per default setting, page location starts from 0 (zero). Hence, specify 0 (zero) to fetch the data of the first page and 1 (one) to fetch the data of the second page.</p>
2.	size	<p>Request parameter to specify the count of fetched records.</p> <p>Specify value greater than or equal to 1.</p> <p>When value specified is greater than the value in <code>maxPageSize</code> of <code>PageableHandlerMethodArgumentResolver</code>, <code>maxPageSize</code> value becomes <code>size</code> value.</p>
3.	sort	<p>Parameter to specify sort condition (multiple parameters can be specified).</p> <p>Specify the value in "<code>{sort item name(, sort order)}</code>" format.</p> <p>Specify either "ASC" or "DESC" as sort order. When nothing is specified, "ASC" is used.</p> <p>Multiple item names can be specified using ", " separator.</p> <p>For example, when <code>"sort=lastModifiedDate, id, DESC&sort=subId"</code> is specified as query string, Order By clause "<code>ORDER BY lastModifiedDate DESC, id DESC, subId ASC</code>" is added to the query.</p>

Warning: Operations at the time of specifying “size=0” in spring-data-commons 1.6.1.RELEASE

spring-data-commons 1.6.1.RELEASE having terasoluna-gfw-common 1.0.0.RELEASE has a bug wherein if “size=0” is specified, all the records matching the specified condition are fetched. As a result, `java.lang.OutOfMemoryError` may occur when a large amount of records are fetched. This problem is handled using JIRA [DATACMNS-377](#) of Spring Data Commons and is being resolved in spring-data-commons 1.6.3.RELEASE. Post modification, if “size<=0” is specified, the default value when size parameter is omitted will be applied.

In cases where terasoluna-gfw-common 1.0.0.RELEASE is used, the version should be upgraded to terasoluna-gfw-common 1.0.1.RELEASE or higher version.

Warning: About operations when invalid values are specified in request parameters of spring-data-commons 1.6.1.RELEASE

spring-data-commons 1.6.1.RELEASE having terasoluna-gfw-common 1.0.0.RELEASE has a bug wherein if an invalid value is specified in request parameters for page search (page, size, sort etc.), `java.lang.IllegalArgumentException` or `java.lang.ArrayIndexOutOfBoundsException` occurs and SpringMVC settings when set to default values leads to system error (HTTP status code=500).

This problem is handled using JIRA [DATACMNS-379](#) and [DATACMNS-408](#) and is being resolved in spring-data-commons 1.6.3.RELEASE. Post modification, if invalid values are specified, the default value when parameters are omitted will be applied.

In cases where terasoluna-gfw-common 1.0.0.RELEASE is used, the version should be upgraded to terasoluna-gfw-common 1.0.1.RELEASE or higher version.

Display of pagination link

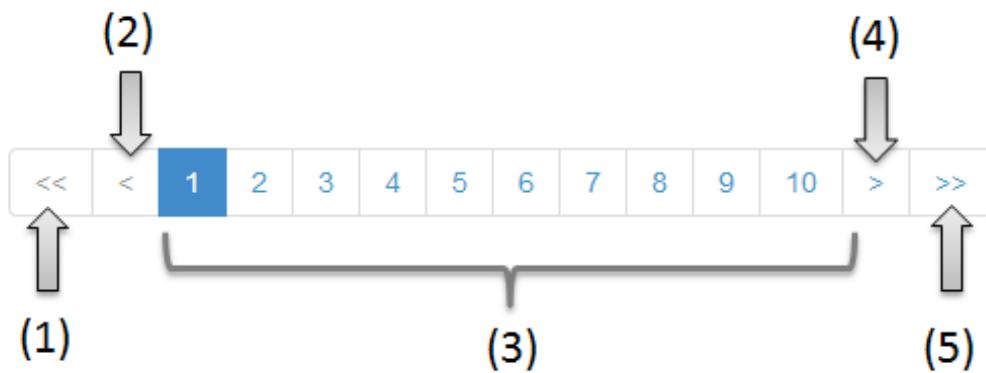
This section describes the pagination link which is output using JSP tag library of common library.

Style sheet to display pagination link is not provided from common library, hence it should be created in each project.

Bootstrap v3.0.0 style sheet is applied for the screens used in the explanation below.

Structure of pagination link

Pagination link consists of the following elements.



Sr. No.	Description
(1)	Link to navigate to the first page.
(2)	Link to navigate to the previous page.
(3)	Link to navigate to the specified page.
(4)	Link to navigate to the next page.
(5)	Link to navigate to the last page.

Pagination link has the following status.



Sr. No.	Description
(6)	<p>Status indicating link where operations cannot be performed on the currently displayed page.</p> <p>The status is specifically “Link to navigate to the first page” and “Link to navigate to the previous page” when the first page is displayed and “Link to navigate to the next page” “Link to navigate to the last page” when the last page is displayed.</p> <p>This status is defined as "disabled" in the JSP tag library of common library.</p>
(7)	<p>Status indicating currently displayed page.</p> <p>This status is defined as "active" in the JSP tag library of common library.</p>

HTML to be output using common library is as follows:

The numbers in figure correspond to serial numbers of “Structure of pagination link” and “Status of pagination link” mentioned above.

- JSP

```
<t:pagination page="${page}" />
```

- HTML to be output

```
<ul>
```

```
    <li class="disabled">  
        <a href="#">&lt;&lt;/a>  
    </li>
```

(1)

```
    <li class="disabled">  
        <a href="#">&lt;</a>  
    </li>
```

(2)

```
    <li class="active">  
        <a href="?page=0&size=6">1</a>  
    </li>
```

(7)

```
    <li>  
        <a href="?page=1&size=6">2</a>  
    </li>
```

(3)

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

```
    <li>  
        <a href="?page=9&size=6">10</a>  
    </li>
```

```
    <li>  
        <a href="?page=1&size=6">&gt;</a>  
    </li>
```

(4)

```
    <li>  
        <a href="?page=9&size=6">&gt;&gt;</a>  
    </li>
```

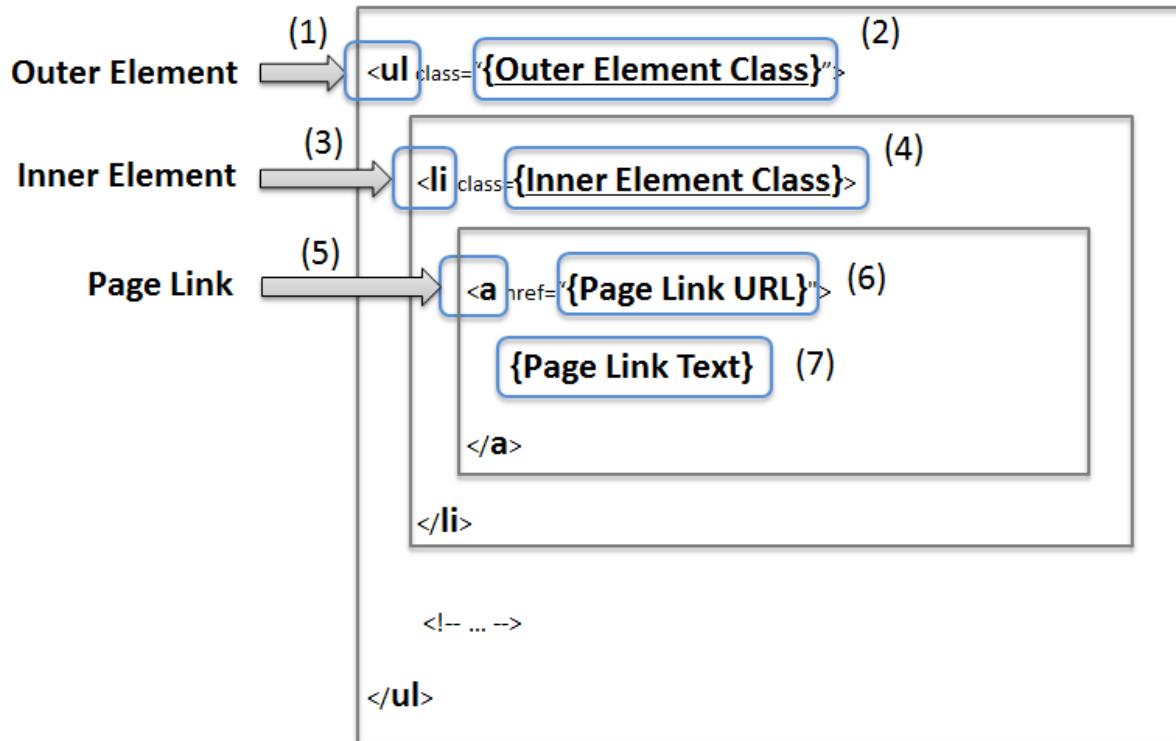
(5)

```
</ul>
```

HTML of pagination link

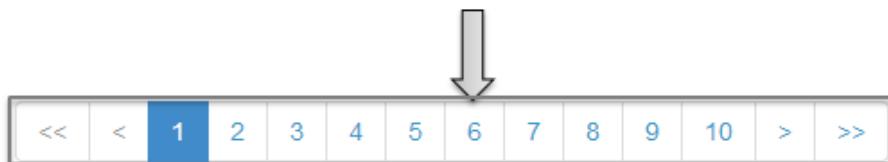
HTML of pagination link to be output using common library is as follows:

- HTML

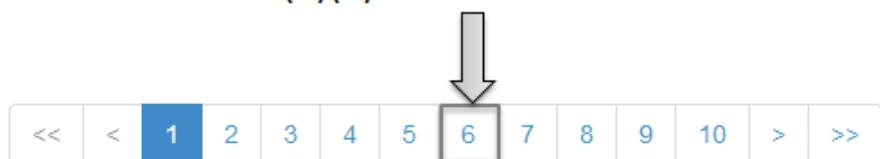


- Screen image

(1)(2) Outer Element



(3)(4) Inner Element



(5)(6)(7) Page Link



Sr. No.	Description	Default values
(1)	<p>Elements to combine the components of pagination link. In common library, this part is called “Outer Element” which stores multiple “Inner Elements”. The elements to be used can be changed using the parameters of JSP tag library.</p>	 element
(2)	<p>Attribute to specify style class of “Outer Element”. In common library, this part is called “Outer Element Class”. The attribute values are specified using the parameters of JSP tag library.</p>	No specification
(3)	<p>Elements to configure pagination link. Call this portion as “Inner Element” and maintain <a> element to send request for navigating the page in common library. The elements can be changed using parameter of JSP tag library.</p>	 elements
818	<p align="center">5 Architecture in Detail - TERASOLUNA Global Framework</p> <p>(4)</p> <p>Attribute to specify style class of “Inner Elements”. In common library, this part is called “Inner Element Class”. The attribute values are switched during JSP tag library processing according to the</p>	Refer to “ <i>Note column</i> ”.

Note: About number of “Inner Elements”

As per default setting, there are maximum 14 “Inner Elements”. Their division is as follows:

- Link to navigate to the first page : 1
- Link to navigate to the previous page : 1
- Link to navigate to the specified page : Maximum 10
- Link to navigate to the next page : 1
- Link to navigate to the last page : 1

The number of “Inner Elements” can be changed by specifying parameters of JSP tag library.

Note: About setting values of “Inner Element Class”

As per default setting, following are the three values depending on location of the page.

- "disabled" : Style class indicating the link which cannot be operated on the currently displayed page.
- "active" : Style class indicating the link of currently displayed page.
- No specification : Indicating the link other than those mentioned above.

"disabled" and "active" values can be changed by specifying the parameters of JSP tag library.

Note: About default values of “Page Link URL”

When link status is "disabled", the default value is "#" and if not "disabled", the default value is "?page={page}&size={size}".

“Page Link URL” can be changed to another value as per the specification of parameters of JSP tag library.

Note: About default values of “Page Link Text”

Sr. No.	Link name	Default values
1.	Link to navigate to the first page	"<<"
2.	Link to navigate to the previous page	"<"
3.	Link to navigate to the specified page	Page number of the corresponding page (cannot be changed)
4.	Link to navigate to the next page	">"
5.	Link to navigate to the last page	">>"

Links other than “Link to navigate to the specified page” can be changed as per the specification of parameters of JSP tag library.

Parameters of JSP tag library

Default operations can be changed by specifying values in parameters of JSP tag library.

List of parameters is shown below.

Parameters to control layout

Sr. No.	Parameter name	Description
1.	outerElement	<p>Specify HTML element name to be used as “Outer Element”.</p> <p>Example: div</p>
2.	outerElementClass	<p>Specify class name of style sheet to be set in “Outer Element Class”.</p> <p>Example: pagination</p>
3.	innerElement	<p>Specify HTML element name to be used as “Inner Element”.</p> <p>Example: span</p>
4.	disabledClass	<p>Specify the value to be set in the class attribute of “Inner Element” with "disabled" status.</p> <p>Example: hiddenPageLink</p>
5.	activeClass	<p>Specify the value to be set in the class attribute of “Inner Element” with "active" status.</p> <p>Example: currentPageLink</p>
6.	firstLinkText	<p>Specify the value to be set in “Page Link Text” of “Link to navigate to the first page”.</p> <p>If "" is specified, “Link to navigate to the first page” itself is not output.</p> <p>Example: First</p>
7.	previousLinkText	<p>Specify the value to be set in “Page Link Text” of “Link to navigate to the previous page”.</p> <p>If "" is specified, “Link to navigate to the previous page” itself is not output.</p> <p>Example: Prev</p>
8.	nextLinkText	
5.11. Pagination		<p>Specify the value to be set in “Page Link Text” of “Link to navigate to the next page”.</p> <p>If "" is specified, “Link to navigate to the next page” itself is not output.</p> <p>Example: Next</p>

Warning: Restrictions related to active page link

The mechanism to disable "active" page link is not provided; hence when a page link is clicked, request is sent to fetch the corresponding page. When a request is not supposed to be sent, it is necessary to either extend JSP tag library of common library or to control the processing so that the request is not sent from JavaScript.

When default values of all parameters to control the layout are changed, the following HTML is output. The numbers in figure correspond to serial numbers in the parameter list mentioned above.

- JSP

```
<t:pagination page="${page}"  
    outerElement="div"  
    outerElementClass="pagination"  
    innerElement="span"  
    disabledClass="hiddenPageLink"  
    activeClass="currentPageLink"  
    firstLinkText="First"  
    previousLinkText="Prev"  
    nextLinkText="Next"  
    lastLinkText="Last"  
    maxDisplayCount="5"  
    />
```

- Output HTML

Parameters to control operations

```
(1) <div class="pagination">
      (2)
      (3) <span class="hiddenPageLink">
          <a href="#">First</a>
        </span>           (4)
      <span class="hiddenPageLink">
          <a href="#">Prev</a>
        </span>           (5)
      <span class="currentPageLink">
          <a href="?page=0&size=6">1</a>
        </span>           (6)
      <!-- .... -->
      <span>
          <a href="?page=4&size=6">5</a>
        </span>
      <span>
          <a href="?page=1&size=6">Next</a>
        </span>           (7)
      <span>
          <a href="?page=9&size=6">Last</a>
        </span>           (8)
    </div>
```

(10)

Sr. No.	Parameter name	Description
1.	disabledHref	<p>Specify the value to be set in “Page Link URL” having “disabled” state.</p> <p>Example: javascript:void(0);</p>
2.	pathTmpl	<p>Specify the template of request path to be set in “Page Link URL”. When request path at the time of page display and the request path for page navigation are different, request path for page navigation needs to be specified in this parameter.</p> <p>In the template of request path to be specified, location of page (page) and number of records to be fetched (size) can be specified as path variables (placeholders).</p> <p>The specified value of URL is encoded in UTF-8.</p>
3.	queryTmpl	<p>Specify the template of query string of “Page Link URL”.</p> <p>Specify the template for generating pagination related query string (page, size, sort parameters) required at the time of page navigation. When setting request parameter name for location of page or the number of records to be fetched to values other than default values, the query string needs to be specified in this parameter.</p> <p>In the template of query string to be specified, location of page (page) and number of records to be fetched (size) can be specified as path variables (placeholders).</p> <p>The specified value of URL is encoded in UTF-8.</p>
4.	criteriaQuery	<p>This attribute is used to generate pagination related query string (page, size, sort parameters); hence query string for adding search conditions should be specified in criteriaQuery attribute.</p>
824		<p>5 Architecture in Detail - TERASOLUNA Global Framework</p> <p>If EL function (<code>f : query(Object)</code>) of common library is used when converting the search conditions stored in form object into encoded URL query string, the search conditions can be added easily.</p>

Note: About setting values of disabledHref

"#" is set in disabledHref by default; hence the focus moves to top of the page on clicking page link. In order to completely disable the operations on clicking page link, it is necessary to specify "javascript:void(0);".

Note: Path variables (placeholders)

Path variables that can be specified in pathTmpl and queryTmpl are as follows:

Sr. No.	Path variable name	Description
1.	page	Path variable for inserting page location.
2.	size	Path variable for inserting number of records to be fetched.
3.	sortOrderProperty	Path variable for inserting sort field of sort condition.
4.	sortOrderDirection	Path variable for inserting sort order of sort condition.

Specify path variables in "`{Path variable name}`" format.

Warning: Constraints related to sort condition

Only one sort condition can be set as a sort condition path variable. Therefore, when the search result obtained by specifying multiple sort condition needs to be displayed using pagination, it is necessary to extend the JSP tag library of common library.

When default parameters to control the operations are changed, the following HTML is output. The numbers in figure correspond to serial numbers of parameter list mentioned above.

- JSP

```
<t:pagination page="${page}"
    disabledHref="javascript:void(0);"
    pathTmpl="${pageContext.request.contextPath}/article/list/{page}/{size}"
    queryTmpl="sort={sortOrderProperty}, {sortOrderDirection}"
    criteriaQuery="${f:query(articleSearchCriteriaForm)}" />
```

- HTML to be output

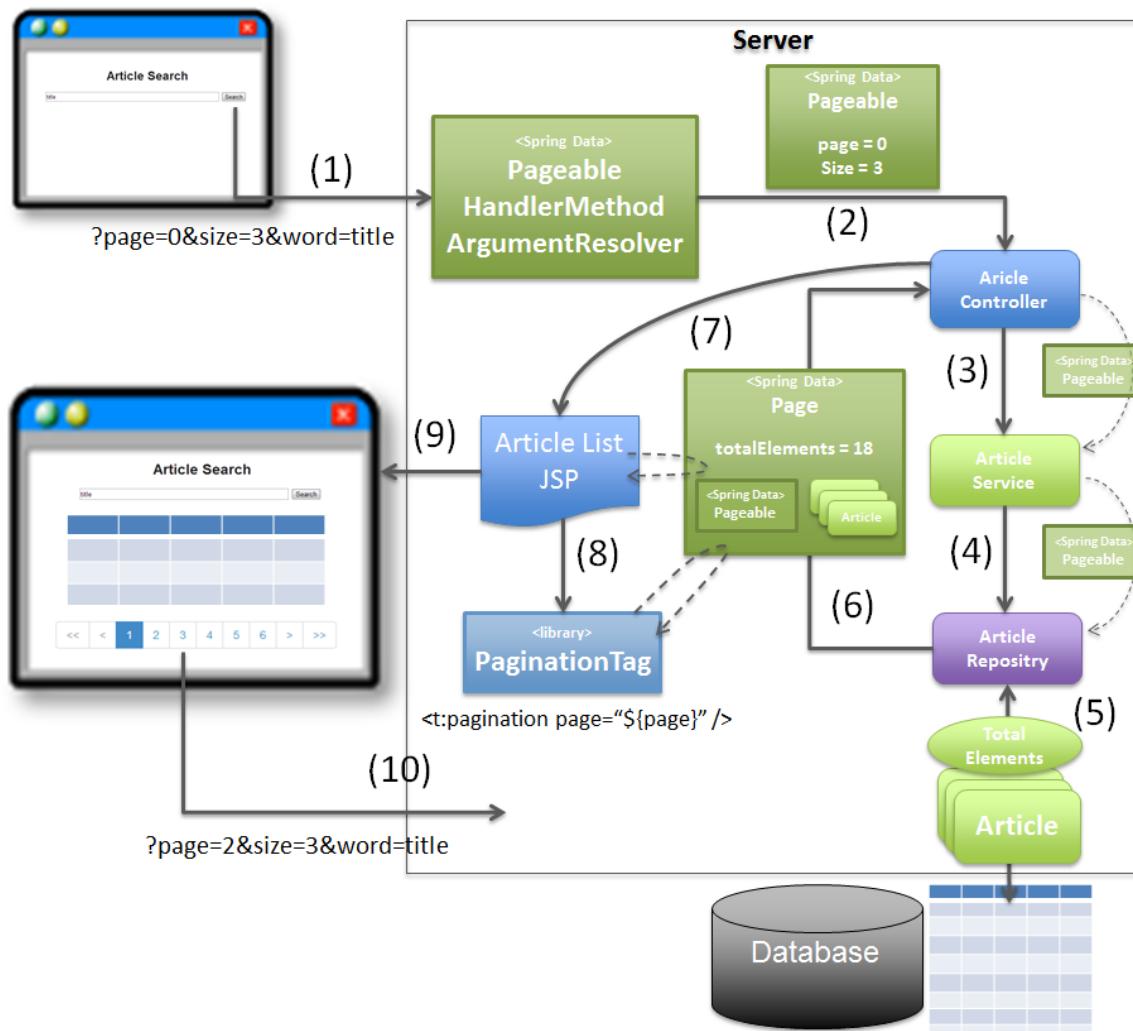
```
<ul>
  <li class="disabled"> (1)
    <a href="javascript:void(0);">&lt;&lt;</a>
  </li>
  <li class="disabled">
    <a href="javascript:void(0);">&lt;</a>
  </li>
  <li class="active"> (2) (3) (4)
    <a href="/webapp/article/list/0/6?sort=publishedDate,DESC&word=title">1</a>
  </li>

  <!-- ... -->

  <li>
    <a href="/webapp/article/list/9/6?sort=publishedDate,DESC&word=title">10</a>
  </li>
  <li>
    <a href="/webapp/article/list/1/6?sort=publishedDate,DESC&word=title">&gt;</a>
  </li>
  <li>
    <a href="/webapp/article/list/9/6?sort=publishedDate,DESC&word=title">&gt;&gt;</a>
  </li>
</ul>
```

Process flow when pagination is used

Process flow when using pagination functionality of Spring Data and JSP tag library of common library is as follows:



Sr. No.	Description
(1)	Apart from search conditions, specify the location of page to be searched (page) and number of records to be fetched (size) as request parameters and send the request.
(2)	<code>PageableHandlerMethodArgumentResolver</code> fetches location of page to be searched (page) and number of records to be fetched (size) specified in request parameter and creates <code>Pageable</code> object. The created <code>Pageable</code> object is set as an argument of Controller processing method.
(3)	Controller passes the <code>Pageable</code> object received as an argument to Service method.
5.11. Pagination (4)	Service passes the <code>Pageable</code> object received as an argument to Query method of Repository.
(5)	Repository fetches total records (<code>totalElements</code>) of data matching the search conditions. It also fetches the required data (size) from the database.

Note: Implementation of Repository

When the method of Repository interface of Spring Data JPA is used, the processes (5) and (6) are carried out automatically using Spring Data JPA functionality.

When using Mybatis2, repository needs to be implemented in Java or SQL mapping file. For example, refer to ?g:doc:DataAccessMybatis2?h.

5.11.2 How to use

The method of using pagination functionality is as follows:

Application settings

Settings for enabling pagination functionality of Spring Data

Location of page to be searched (page), number of records to be fetched (size) and sort condition (sort) are specified in the request parameter. The functionality to set these properties in the argument of Controller as Pageable object should be enabled.

The settings mentioned below are preset in a blank project.

spring-mvc.xml

```
<mvc:annotation-driven>
    <mvc:argument-resolvers>
        <!-- (1) -->
        <bean
            class="org.springframework.data.web.PageableHandlerMethodArgumentResolver" />
    </mvc:argument-resolvers>
</mvc:annotation-driven>
```

Sr. No.	Description
(1)	<p>Specify <code>org.springframework.data.web.PageableHandlerMethodArgumentResolver</code> in <code><mvc:argument-resolvers></code>.</p> <p>For the properties that can be specified in <code>PageableHandlerMethodArgumentResolver</code>, refer to “<i>About property values of PageableHandlerMethodArgumentResolver</i>”.</p>

Page search

The method of implementing page search is as follows:

Implementation of application layer

The information required for page search (such as location of page to be searched, number of records to be fetched and sort condition) is received as an argument and passed to Service method.

- Controller

```
@RequestMapping("list")
public String list(@Validated ArticleSearchCriteriaForm form,
                   BindingResult result,
                   Pageable pageable, // (1)
                   Model model) {

    ArticleSearchCriteria criteria = beanMapper.map(form,
            ArticleSearchCriteria.class);

    Page<Article> page = articleService.searchArticle(criteria, pageable); // (2)

    model.addAttribute("page", page); // (3)

    return "article/list";
}
```

Sr. No.	Description
(1)	Specify <code>Pageable</code> as an argument of processing method. <code>Pageable</code> object stores the information required for page search (such as location of page to be searched, number of records to be fetched and sort condition).
(2)	Specify the <code>Pageable</code> object as an argument of Service method and then call the same.
(3)	Add the search result (<code>Page</code> object) returned by Service to Model. It can be referred from View (JSP) after it is added to Model.

Note: Operations when the information required for page search is not specified in request parameter

Default values are applied when the information required for page search (such as location of page to be searched, number of records to be fetched and sort condition) is not specified in request parameter. Default values are as follows:

- Location of page to be searched: *0* (first page)
- number of records to be fetched: *20*
- Sort condition: *null* (no sort condition)

Default values can be changed using the following two methods.

- Define the default values by specifying `@org.springframework.data.web.PageableDefault` annotation as an argument of `Pageable` of processing method.
- Specify `Pageable` object wherein default values are defined in `fallbackPageable` property of `PageableHandlerMethodArgumentResolver`.

See the method below for specifying default values using `@PageableDefault` annotation.

Use `@PageableDefault` annotation to change the default values for each page search.

```
@RequestMapping("list")
public String list(@Validated ArticleSearchCriteriaForm form,
                   BindingResult result,
                   @PageableDefault( // (1)
                        page = 0,      // (2)
                        size = 50,     // (3)
                        direction = Direction.DESC, // (4)
                        sort = {        // (5)
                            "publishedDate",
                            "articleId"
                        }
                   ) Pageable pageable,
                   Model model) {
    // ...
    return "article/list";
}
```

Sr. No.	Description	Default values
(1)	Specify <code>@PageableDefault</code> annotation as an argument of <code>Pageable</code> .	-
(2)	To change the default value of location of page, specify the value in <code>page</code> attribute of <code>@PageableDefault</code> annotation. Normally it need not be changed.	0 (first page)
(3)	To change the default value of number of records to be fetched, specify the value in <code>size</code> or <code>value</code> attribute of <code>@PageableDefault</code> annotation.	10
(4)	To change the default value of sort condition, specify the value in <code>direction</code> attribute of <code>@PageableDefault</code> annotation.	<code>Direction.ASC</code> (Ascending order)
(5)	Specify the sort fields of sort condition in <code>sort</code> attribute of <code>@PageableDefault</code> annotation. When sorting the records using multiple sort fields, specify the property name to be sorted in array. In the above example, sort condition " <code>ORDER BY publishedDate DESC, articleId DESC</code> " is added to Query.	Empty array (No sort field)

Note: About sort order that can be specified using `@PageableDefault` annotation

Sort order that can be specified using `@PageableDefault` annotation is either ascending or descending; hence when you want to specify different sort order for each field, it is necessary to use `@org.springframework.data.web.SortDefaults` annotation. For example, when sorting the fields using "`ORDER BY publishedDate DESC, articleId ASC`" sort order.

Tip: Specifying annotation when only the default value of number of records to be fetched is to be changed

In order to change only the default value of number of records to be fetched, the annotation can also be specified as `@PageableDefault(50)`. This operation is same as `@PageableDefault(size = 50)`.

See the method below to specify default values using `@SortDefaults` annotation.

`@SortDefaults` annotation is used when sorting needs to be done on multiple fields and in order to have different sort order for each field.

```
@RequestMapping("list")
public String list(
    @Validated ArticleSearchCriteriaForm form,
    BindingResult result,
    @PageableDefault(size = 50)
    @SortDefaults( // (1)
        {
            @SortDefault( // (2)
                sort = "publishedDate", // (3)
                direction = Direction.DESC // (4)
            ),
            @SortDefault(
                sort = "articleId"
            )
        }) Pageable pageable,
    Model model) {
    // ...
    return "article/list";
}
```

Sr. No.	Description	Default values
(1)	Specify <code>@SortDefaults</code> annotation as an argument of <code>Pageable</code> . Multiple <code>@org.springframework.data.web.SortDefault</code> annotations can be specified as arrays in <code>@SortDefaults</code> annotation.	-
(2)	Specify <code>@SortDefault</code> annotation as value attribute of <code>@SortDefaults</code> annotation. Specify as array when specifying multiple annotations.	-
(3)	Specify sort fields in sort or value attribute of <code>@PageableDefault</code> . Specify as array when specifying multiple fields.	Empty array (No sort field)
(4)	Specify value in direction attribute of <code>@PageableDefault</code> to change default sort condition.	<code>Direction.ASC</code> (Ascending)

In the above example, sort condition called "ORDER BY publishedDate DESC, articleId ASC" is added to query.

Tip: Specifying annotation when only the default value of sort fields is to be specified

In order to specify only the records to be fetched, the annotation can also be specified as `@PageableDefault("articleId")`. This operation is same as `@PageableDefault(sort = "articleId")` and `@PageableDefault(sort = "articleId", direction = Direction.ASC)`.

When it is necessary to change the default values of entire application, specify `Pageable` object wherein default values are defined in `fallbackPageable` property of `PageableHandlerMethodArgumentResolver` that is defined in `spring-mvc.xml`.

For description of `fallbackPageable` and example of settings, refer to "[About property values of Pageable-HandlerMethodArgumentResolver](#)".

Implementation of domain layer (JPA)

When accessing the database using JPA (Spring Data JPA), pass Pageable object received from Controller to Repository.

See the example below for simplest way of implementation.

For the details on page search to be implemented in domain layer, refer to *Database Access (JPA)*.

- Service

```
public Page<Article> searchArticle(ArticleSearchCriteria criteria,
    Pageable pageable) { // (1)

    String word = QueryEscapeUtils.toLikeCondition(criteria.getWord());

    Page<Article> page = articleRepository.findPageBy(word, pageable); // (2)

    return page; // (3)
}
```

Sr. No.	Description
(1)	Receive the information required for page search (Pageable) as an argument of Service method.
(2)	Specify the Pageable object as an argument of Repository query method and then call the same.
(3)	Return the search result (Page object) returned by Repository to Controller.

- Repository

```
@Query("SELECT a FROM Article a WHERE a.title LIKE %:freeWord% ESCAPE '~' OR a.overview LIKE
Page<Article> findPageByFreeWord(@Param("freeWord") String word, Pageable pageable); // (4)
```

Sr. No.	Description
(4)	<p>Receive the information required for page search (<code>Pageable</code>) as an argument of Repository query method.</p> <p>Return value type should be <code>Page<Entity></code>.</p> <p>If the above method is created, Spring Data JPA functionality can return <code>Page</code> object by extracting data corresponding to <code>Pageable</code> object status.</p>

Implementation of Service (Mybatis2)

When accessing the database using Mybatis2(TERASOLUNA DAO), extract the necessary information from `Pageable` object received from Controller and call TERASOLUNA DAO method.

SQL to fetch the corresponding data and sort condition needs to be executed using SQL mapping.

For details on page search to be implemented in domain layer, refer to “[Database access \(Mybatis2\)](#)”.

Implementation of JSP (Base version)

The method to display pagination link and pagination information (total records, total pages, number of displayed pages etc.) by displaying the `Page` object fetched during page search on list screen, is described below.

Display of fetched data

An example to display the data fetched during page search is shown below.

- Controller

```
@RequestMapping("list")
public String list(@Validated ArticleSearchCriteriaForm form, BindingResult result,
    Pageable pageable, Model model) {

    if (!StringUtils.hasLength(form.getWord())) {
        return "article/list";
    }

    ArticleSearchCriteria criteria = beanMapper.map(form,
```

```

        ArticleSearchCriteria.class);

Page<Article> page = articleService.searchArticle(criteria, pageable);

model.addAttribute("page", page); // (1)

return "article/list";
}

```

Sr. No.	Description
(1)	Store Page object with the attribute name "page" in Model. In JSP, Page object can be accessed by specifying attribute name "page".

- JSP

```

<%-- ... --%>

<%-- (2) --%>
<c:when test="${page != null && page.totalPages != 0}">



| No                                              | Class                              | Title                  | Overview                  | Published Date                 |
|-------------------------------------------------|------------------------------------|------------------------|---------------------------|--------------------------------|
| \${(page.number * page.size) + rowStatus.count} | \${f:h(article.articleClass.name)} | \${f:h(article.title)} | \${f:h(article.overview)} | \${f:h(article.publishedDate)} |


```

```
</td>
</tr>
</c:forEach>

</table>

<div class="paginationPart">

<%-- ... --%>

</div>
</c:when>

<%-- ... --%>
```

Sr. No.	Description
(2)	In the above example, it is checked whether data matching the specified conditions exists. If there is no such data, the header row is also not displayed. When it is necessary to display the header row even if there is no matching data, this branching is no longer required.
(3)	Display the list of data fetched using <c:forEach> tag of JSTL. The fetched data is stored in a list in content property of Page object.

- Example of screen output in JSP

No	Class	Title	Overview	Published Date
1	Internal	Internal title1_20	overview20	2013-10-29
2	International	International title2_20	overview20	2013-10-29
3	Economy	Economy title3_20	overview20	2013-10-29
4	Internal	Internal title1_19	overview19	2013-10-28
5	International	International title2_19	overview19	2013-10-28
6	Economy	Economy title3_19	overview19	2013-10-28
•				
•				
•				

Display of Pagination link

See the example below to display the link for page navigation (pagination link).

Pagination link is output using JSP tag library of common library.

- include.jsp

Declare the JSP tag library of common library. The settings below are carried out in a blank project.

```
<%@ taglib uri="http://terasoluna.org/tags" prefix="t"%>      <%-- (1) --%>
<%@ taglib uri="http://terasoluna.org/functions" prefix="f"%>  <%-- (2) --%>
```

Sr. No.	Description
(1)	JSP tag to display pagination link is stored.
(2)	EL function of JSP used at the time of using pagination link, is stored.

- JSP

```
<t:pagination page="${page}" /> <%-- (3) --%>
```

Sr. No.	Description
(3)	Use <t:pagination> tag. In page attribute, specify Page object stored in Model of Controller.

- Output HTML

The example below shows the search results obtained upon specifying "?page=0&size=6".

```
<ul>
  <li class="disabled"><a href="#">&lt;&lt;</a></li>
  <li class="disabled"><a href="#">&lt;&lt;</a></li>
  <li class="active"><a href="?page=0&size=6">1</a></li>
  <li><a href="?page=1&size=6">2</a></li>
  <li><a href="?page=2&size=6">3</a></li>
```

```
<li><a href="?page=3&size=6">4</a></li>
<li><a href="?page=4&size=6">5</a></li>
<li><a href="?page=5&size=6">6</a></li>
<li><a href="?page=6&size=6">7</a></li>
<li><a href="?page=7&size=6">8</a></li>
<li><a href="?page=8&size=6">9</a></li>
<li><a href="?page=9&size=6">10</a></li>
<li><a href="?page=1&size=6">&gt; </a></li>
<li><a href="?page=9&size=6">&gt; &gt; </a></li>
</ul>
```

If style sheet for pagination link is not created, the display will be as follows:

As it is visible below, the pagination link is not established.

- <<
- <
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- >
- >>

The display will be as follows if minimal changes are carried out like adding a definition of style sheet for pagination link and changing the JSP.

- Screen image

<< < 1 2 3 4 5 6 7 8 9 10 > >>

- JSP

```
<%-- ... --%>

<t:pagination page="${page}"
    outerElementClass="pagination" /> <%-- (4) --%>

<%-- ... --%>
```

Sr. No.	Description
(4)	Specify the class name indicating that it is a pagination link. By specifying the class name, the applicable range of styles to be specified in style sheet can be restricted to pagination link.

- Style sheet

```
.pagination li {
    display: inline;
}

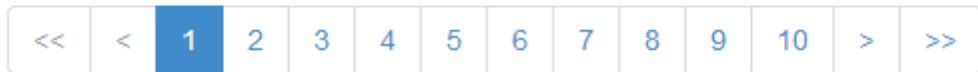
.pagination li>a {
    margin-left: 10px;
}
```

Even after the pagination link is established, the following two problems still persist.

- clickable and non-clickable links cannot be distinguished.
- The location of currently displayed page cannot be identified.

When Bootstrap v3.0.0 style sheet is applied to resolve the above problems, the display is as follows:

- Screen image



- Style sheet

Place the css file of bootstrap v3.0.0 under

`$WEB_APP_ROOT/resources/vendor/bootstrap-3.0.0/css/bootstrap.css.`

Abstract of pagination related style definition.

```
.pagination {  
    display: inline-block;  
    padding-left: 0;  
    margin: 20px 0;  
    border-radius: 4px;  
}  
  
.pagination > li {  
    display: inline;  
}  
  
.pagination > li > a,  
.pagination > li > span {  
    position: relative;  
    float: left;  
    padding: 6px 12px;  
    margin-left: -1px;  
    line-height: 1.428571429;  
    text-decoration: none;  
    background-color: #ffffff;  
    border: 1px solid #dddddd;  
}  
  
.pagination > li:first-child > a,  
.pagination > li:first-child > span {  
    margin-left: 0;  
    border-bottom-left-radius: 4px;  
    border-top-left-radius: 4px;  
}  
  
.pagination > li:last-child > a,  
.pagination > li:last-child > span {  
    border-top-right-radius: 4px;  
    border-bottom-right-radius: 4px;  
}
```

```
.pagination > li > a:hover,
.pagination > li > span:hover,
.pagination > li > a:focus,
.pagination > li > span:focus {
    background-color: #eeeeee;
}

.pagination > .active > a,
.pagination > .active > span,
.pagination > .active > a:hover,
.pagination > .active > span:hover,
.pagination > .active > a:focus,
.pagination > .active > span:focus {
    z-index: 2;
    color: #ffffff;
    cursor: default;
    background-color: #428bca;
    border-color: #428bca;
}

.pagination > .disabled > span,
.pagination > .disabled > a,
.pagination > .disabled > a:hover,
.pagination > .disabled > a:focus {
    color: #999999;
    cursor: not-allowed;
    background-color: #ffffff;
    border-color: #dddddd;
}
```

- JSP

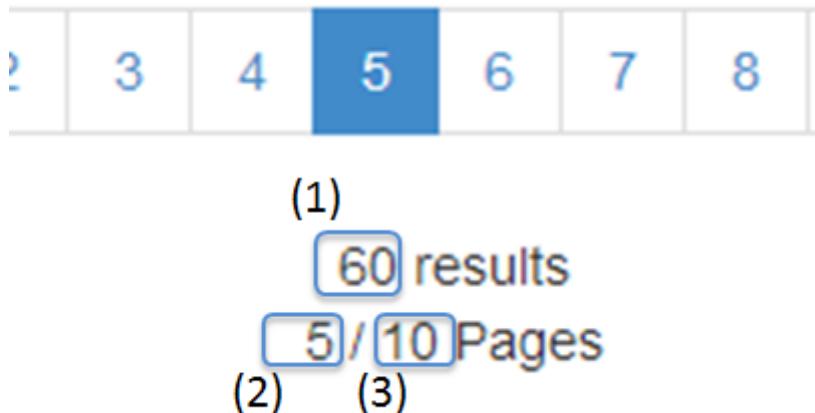
Add a definition to read the css file placed under JSP.

```
<link rel="stylesheet"
      href="${pageContext.request.contextPath}/resources/vendor/bootstrap-3.0.0/css/bootstrap.
      type="text/css" media="screen, projection">
```

Display of pagination information

An example to display the information related to pagination (such as total records, total pages and total displayed pages) is as follows:

- Screen example



- JSP

```
<div>
    <fmt:formatNumber value="${page.totalElements}" /> results <%-- (1) --%>
</div>
<div>
    ${f:h(page.number + 1)} / <%-- (2) --%>
    ${f:h(page.totalPages)} Pages <%-- (3) --%>
</div>
```

Sr. No.	Description
(1)	To display total number of data records matching the search conditions, fetch value from <code>totalElements</code> property of <code>Page</code> object.
(2)	To display number of displayed pages, fetch value from <code>number</code> property of <code>Page</code> object and increment the value by 1. <code>number</code> property of <code>Page</code> object starts with 0; hence value should be incremented by 1 at the time of displaying the page number.
(3)	To display total pages of data matching the search conditions, fetch the value from <code>totalPages</code> property of <code>Page</code> object.

Example to display the display data range of the corresponding page is shown below.

- Example of screen



- JSP

```
<div>
    <%-- (4) --%>
    <fmt:formatNumber value="${(page.number * page.size) + 1}" /> -
    <%-- (5) --%>
    <fmt:formatNumber value="${(page.number * page.size) + page.numberOfElements}" />
</div>
```

Sr. No.	Description
(4)	To display start location, calculate the value using <code>number</code> property and <code>size</code> property of <code>Page</code> object. <code>number</code> property of <code>Page</code> object starts with 0; hence the value needs to be incremented by 1 at the time of displaying data start location.
(5)	To display end location, calculate the value using <code>number</code> property, <code>size</code> property and <code>numberOfElements</code> property of <code>Page</code> object. <code>numberOfElements</code> needs to be calculated since the last page is likely to be a fraction.

Tip: About format of numeric value

When the numeric value to be displayed needs to be formatted, use tag library (`<fmt:formatNumber>`) provided by JSTL.

Carrying forward search conditions using page link

The method of carrying forward the search conditions to the page navigation request is shown below.

- JSP

```
<%-- (1) --%>
<div id="criteriaPart">
    <form:form action="${pageContext.request.contextPath}/article/list" method="get"
               modelAttribute="articleSearchCriteriaForm">
        <form:input path="word" />
        <form:button>Search</form:button>
        <br>
    </form:form>
</div>

<%-- ... --%>

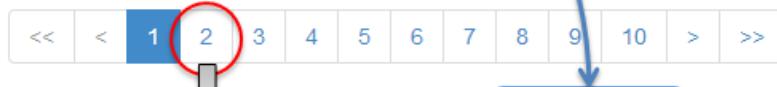
<t:pagination page="${page}"
               outerElementClass="pagination"
               criteriaQuery="${f:query(articleSearchCriteriaForm)}" /> <%-- (2) --%>
```

Sr. No.	Description
(1)	Form to specify search conditions. <code>word</code> is specified as a search condition.
(2)	When carrying forward the search conditions to page navigation request, specify the encoded URL query string in <code>criteriaQuery</code> attribute. When storing the search conditions in form object, conditions can be carried forward easily if EL function (<code>f:query(Object)</code>) provided by common library is used. In the above example, query string of " <code>?page=page location&size=number of records to be fetched&word=input value</code> " format is generated. <code>criteriaQuery</code> attribute can be used in terasoluna-gfw-web 1.0.1.RELEASE or higher version.

Note: Specifications of `f:query(Object)`

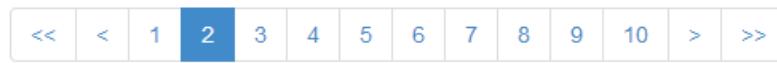
Article Search

No	Class	Title	Overview	Published Date
1	Internal	Internal title1_20	overview20	2013-10-29
2	International	International title2_20	overview20	2013-10-28
3	Economy	Economy title3_20	overview20	
4	International	International title2_19	overview19	2013-10-28
5	Internal	Internal title1_19	overview19	2013-10-28
6	Economy	Economy title3_19	overview19	2013-10-28



localhost:8080/webapp/article/list?page=1&size=6&word=title&sort=publishedDate,DE

No	Class	Title	Overview	Published Date
7	International	International title2_18	overview18	2013-10-27
...				
12	Economy	Economy title3_17	overview17	2013-10-26



JavaBean of form object and Map object can be specified as an argument of `f:query`. In case of JavaBean, property name is treated as request parameter name and in case of Map object, map key name is treated as request parameter. URL of the generated query string is encoded in UTF-8.

Warning: Operations when Query string created using f:query is specified in queryTmpl attribute

It has been found that specifying the query string generated using `f:query` in `queryTmpl` attribute leads to duplication of URL encoding. Thus, special characters are not carried forward correctly.

This URL encoding duplication can be avoided by using `criteriaCuery` attribute which can be used in terasoluna-gfw-web 1.0.1.RELEASE or higher version.

Carrying forward the sort condition using page link

The method to carry forward the sort condition to page navigation request is as follows:

- JSP

```
<t:pagination page="${page}"  
    outerElementClass="pagination"  
    queryTmpl="page={page}&size={size}&sort={sortOrderProperty},{sortOrderDirection}" /> <%
```

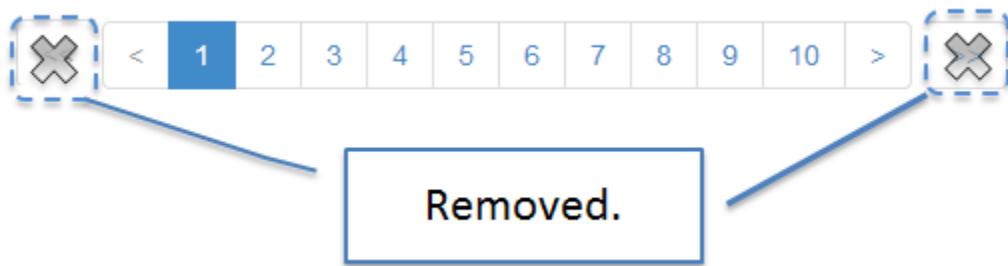
Sr. No.	Description
(1)	To carry forward the sort condition to page navigation request, specify <code>queryTmpl</code> and add sort condition to query string. For parameter specifications to specify sort condition, refer to “ Request parameters for page search “ In the above example, “?page=0&size=20&sort=sort item, sort order(ASC or DESC)” is a query string.

Implementation of JSP (layout change)

Removal of link to navigate to the first page and the last page

Example to remove “Link to navigate to the first page” and “Link to navigate to the last page” is shown below.

- Screen example



- JSP

```
<t:pagination page="${page}"  
    outerElementClass="pagination"  
    firstLinkText=""  
    lastLinkText="" /> <%-- (1) (2) --%>
```

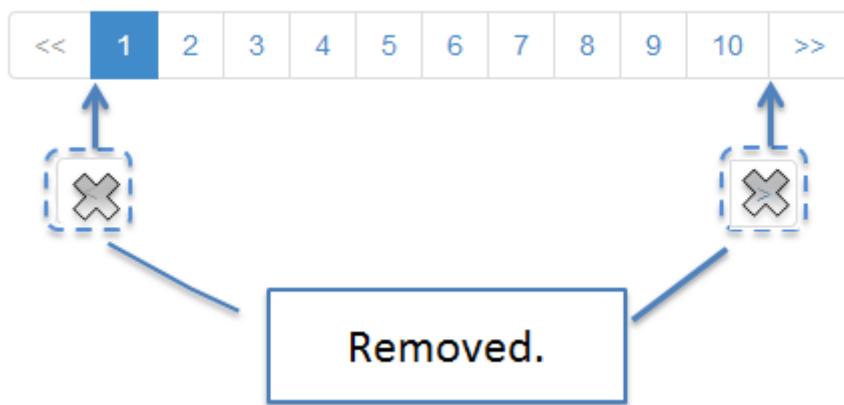
Sr. No.	Description
(1)	Specify "" as firstLinkText attribute of <t:pagination> tag to hide “Link to navigate to the first page”.
(2)	Specify "" as lastLinkText attribute of <t:pagination> tag to hide “Link to navigate to the last page”.

Removal of link to navigate to previous page and next page

Example to remove “Link to navigate to the first page” and “Link to navigate to the last page” is shown below.

- Screen example

```
<t:pagination page="${page}"  
    outerElementClass="pagination"  
    previousLinkText=""  
    nextLinkText="" /> <%-- (1) (2) --%>
```



Sr. No.	Description
(1)	Specify "" as previousLinkText attribute of <t:pagination> tag to hide "Link to navigate to the previous page".
(2)	Specify "" as nextLinkText attribute of <t:pagination> tag to hide "Link to navigate to the next page".

Removal of disabled link

Example to remove the link in "disabled" state is shown below.

Add the following definition to style sheet when the status is "disabled".

- Screen example

- Style sheet

```
.pagination .disabled {  
    display: none; /* (1) */  
}
```

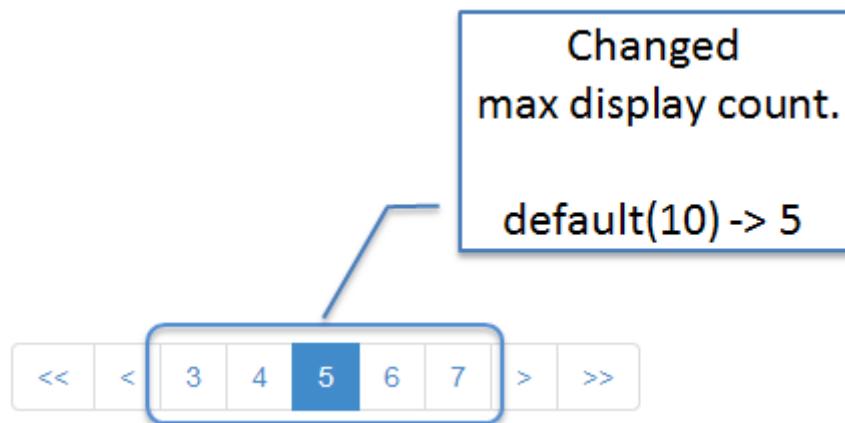


Sr. No.	Description
(1)	Specify "display: none;" as an attribute value of "disabled" class.

Change in maximum number of display links to navigate to the specified page

Example to change maximum number of display links to navigate to the specified page is shown below.

- Screen example



- JSP

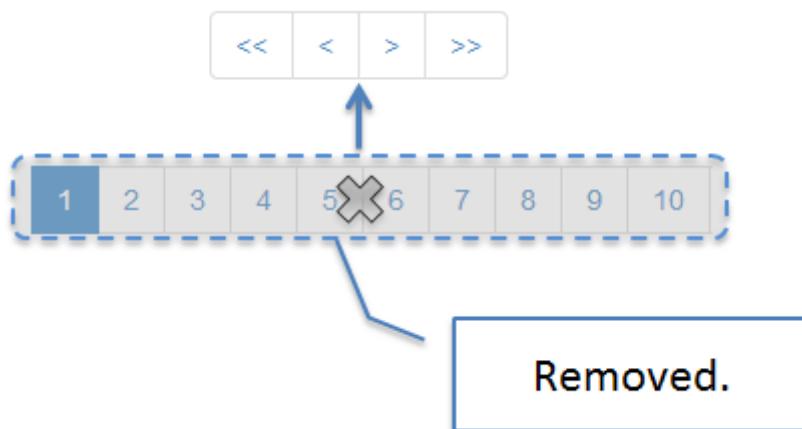
```
<t:pagination page="${page}"
    outerElementClass="pagination"
    maxDisplayCount="5" /> <%-- (1) --%>
```

Sr. No.	Description
(1)	In order to change maximum number of display links to navigate to the specified page, specify value in maxDisplayCount attribute of <t :pagination> tag.

Removal of link to navigate to the specified page

Example to remove link to navigate to the specified page is shown below.

- Screen example



- JSP

```
<t:pagination page="${page}"  
    outerElementClass="pagination"  
    maxDisplayCount="0" /> <%-- (1) --%>
```

Sr. No.	Description
(1)	In order to hide the link to navigate to the specified page, specify "0" as maxDisplayCount attribute of <t :pagination> tag.

Implementation of JSP (Operation)

Specifying sort condition

Example to specify sort condition from client is shown below.

- Screen example

Article Search

The screenshot shows a search interface titled "Article Search". At the top, there is a search bar with the placeholder "title" and a "Search" button. To the right of the search bar is a dropdown menu with three options: "Newest", "Newest" (which is highlighted with a red box), and "Oldest". Below the search area is a table with five columns: "No", "Class", "Title", "Overview", and "Published Date". The first row of the table contains the values: 1, International, International title2_20, overview20, and 2013-10-30. The "Sort" column header is visible above the table.

No	Class	Title	Overview	Published Date
1	International	International title2_20	overview20	2013-10-30

- JSP

```
<div id="criteriaPart">
    <form:form
        action="${pageContext.request.contextPath}/article/search"
        method="get" modelAttribute="articleSearchCriteriaForm">
        <form:input path="word" />
        <%-- (1) --%>
        <form:select path="sort">
            <form:option value="publishedDate,DESC">Newest</form:option>
            <form:option value="publishedDate,ASC">Oldest</form:option>
        </form:select>
        <form:button>Search</form:button>
        <br>
    </form:form>
</div>
```

Sr. No.	Description
(1)	<p>For specifying the sort condition from client, add the corresponding parameters for specifying the sort condition.</p> <p>For parameter specifications to specify sort condition, refer to “Request parameters for page search” .</p> <p>In the above example, publishedDate can be selected in ascending order or descending order from pull-down.</p>

5.11.3 Appendix

About property values of PageableHandlerMethodArgumentResolver

Properties that can be specified in `PageableHandlerMethodArgumentResolver` are as follows:

Values should be changed as required in the application.

Sr. No.	Property name	Description	Default value
1.	maxPageSize	Specify maximum permissible value for the number of records to be fetched. When the specified number of records to be fetched exceeds <code>maxPageSize</code> , only the number of records specified as <code>maxPageSize</code> will be fetched.	2000
2.	fallbackPageable	Specify default values for page location, number of records to be fetched and sort condition of the entire application. When page location, number of records to be fetched and sort condition are not specified, the values set in <code>fallbackPageable</code> are used.	Page location : 0 Number of records to be fetched : 20 Sort condition : <code>null</code>
3.	oneIndexedParameters	Specify start value of page location. When <code>false</code> is specified, start value of page location becomes <code>0</code> and when <code>true</code> is specified, it becomes <code>1</code> .	<code>false</code>
4.	pageParameterName	Specify request parameter name to specify page location.	"page"
5.	sizeParameterName	Specify request parameter name to specify number of records to be fetched.	"size"
6.	prefix	Specify prefix (namespace) of request parameter to specify page location and number of records to be fetched. When there is a conflict between default parameter name and the parameter to be used in the application, it is recommended to specify namespace to avoid this issue. If prefix is specified, request parameter name to specify page location will be <code>prefix + pageParameterName</code> and request parameter name to specify number of records to be fetched will be <code>prefix + sizeParameterName</code> .	" " (No namespace)
5.11. Pagination	7. qualifierDelimiter	+ <code>sizeParameterName</code> . To search multiple pages in the same request, specify request parameter name in <code>qualifier +</code>	855 "_"

Note: Setting value of maxPageSize

Default value is 2000; however it is recommended to change the setting to maximum permissible value for the application. If maximum permissible value for the application is 100, maxPageSize should also be set to 100.

Note: Setting fallbackPageable

To change default values used in the entire application, set Pageable (org.springframework.data.domain.PageRequest) object wherein default value is defined in fallbackPageable property . To change the default sort condition, org.springframework.data.domain.Sort object where default value is defined in fallbackSort property of SortHandlerMethodArgumentResolver.

It is assumed that the fields given below will normally be changed in each application to be developed. The example to change the default values of such fields is given below.

- Maximum permissible value for number of records to be fetched (maxPageSize)
- Default values (fallbackPageable) of page location and number of records to be fetched in the entire application
- Default sort condition (fallbackSort)

```
<mvc:annotation-driven>
    <mvc:argument-resolvers>
        <bean
            class="org.springframework.data.web.PageableHandlerMethodArgumentResolver">
            <!-- (1) -->
            <property name="maxPageSize" value="100" />
            <!-- (2) -->
            <property name="fallbackPageable">
                <bean class="org.springframework.data.domain.PageRequest">
                    <!-- (3) -->
                    <constructor-arg index="0" value="0" />
                    <!-- (4) -->
                    <constructor-arg index="1" value="50" />
                </bean>
            </property>
            <!-- (5) -->
            <constructor-arg index="0">
```

```
<bean class="org.springframework.data.web.SortHandlerMethodArgumentResolver">
    <!-- (6) -->
    <property name="fallbackSort">
        <bean class="org.springframework.data.domain.Sort">
            <!-- (7) -->
            <constructor-arg index="0">
                <list>
                    <!-- (8) -->
                    <bean class="org.springframework.data.domain.Sort.Order">
                        <!-- (9) -->
                        <constructor-arg index="0" value="DESC" />
                        <!-- (10) -->
                        <constructor-arg index="1" value="lastModifiedDate" />
                    </bean>
                    <!-- (8) -->
                    <bean class="org.springframework.data.domain.Sort.Order">
                        <constructor-arg index="0" value="ASC" />
                        <constructor-arg index="1" value="id" />
                    </bean>
                </list>
            </constructor-arg>
        </bean>
    </property>
    </bean>
</constructor-arg>
</bean>
</mvc:argument-resolvers>
</mvc:annotation-driven>
```

Sr. No.	Description
(1)	In the above example, maximum value of number of records to be fetched is set to 100. When value specified in number of records to be fetched (size) is 101 or more, search is performed for 100 records only.
(2)	Create an instance of <code>org.springframework.data.domain.PageRequest</code> and set to <code>fallbackPageable</code> .
(3)	Specify default value of page location as the first argument of constructor of <code>PageRequest</code> . In the above example, 0 is specified, hence the default value is not changed.
(4)	Specify default value of number of records to be fetched as the second argument of constructor of <code>PageRequest</code> . In the above example, the value will be considered 50 when number of records to be fetched is not specified in request parameter.
(5)	Set an instance of <code>SortHandlerMethodArgumentResolver</code> as constructor of <code>PageableHandlerMethodArgumentResolver</code> .
(6)	Create an instance of <code>Sort</code> and set to <code>fallbackSort</code> .
(7)	Set the list of <code>Order</code> objects to be used as default value as the first argument of <code>Sort</code> constructor.
(8)	Create an instance of <code>Order</code> and add to the list of <code>Order</code> objects to be used as default value. In the above example, sort condition of "ORDER BY x.lastModifiedDate DESC, x.id ASC" is added to query when sort condition is not specified in request parameter.
(9)	Specify sort order (ASC/DESC) as the first argument of <code>Order</code> constructor.
858	<p style="text-align: center;">5 Architecture in Detail - TERASOLUNA Global Framework</p> <p>(10) Specify sort item as the second argument of <code>Order</code> constructor.</p>

Property value of SortHandlerMethodArgumentResolver

Properties that can be specified in SortHandlerMethodArgumentResolver are as follows:

Values should be changed as required in the application.

Sr. No.	Property name	Description	Default value
1.	fallbackSort	Specify default sort condition for the entire application. When sort condition is not specified, the value set in fallbackSort is used.	null (No sort condition)
2.	sortParameter	Specify request parameter name to specify the sort condition. When there is a conflict between default parameter name and the parameter to be used in the application, it is recommended to change the request parameter name to avoid this issue.	"sort"
3.	propertyDelimiter	Specify delimiter of sort items and sort order (ASC,DESC).	" , "
4.	qualifierDelimiter	To search multiple pages in the same request, specify request parameter name in “ qualifier + delimiter + sortParameter “ format to distinguish the information required for page search (such as sort condition). For this property, set delimiter value of the above format.	" _ "

5.12 Double Submit Protection

5.12.1 Overview

Problems

If any of the operations given below is performed in a Web application with screens, it leads to same process being executed multiple times.

Sr. No.	Operation	Operation Overview
(1)	Double clicking of ‘Update’ button	Repeatedly clicking the button to perform update process.
(2)	Reloading of screen after completing the update process	Using the ‘Refresh’ button of the browser to reload the screen after completion of update process.
(3)	Invalid screen transition using ‘Back’ button of the browser	Using ‘Back’ button of the browser to go back to the previous page from update process completion screen and clicking the button again to perform update process.

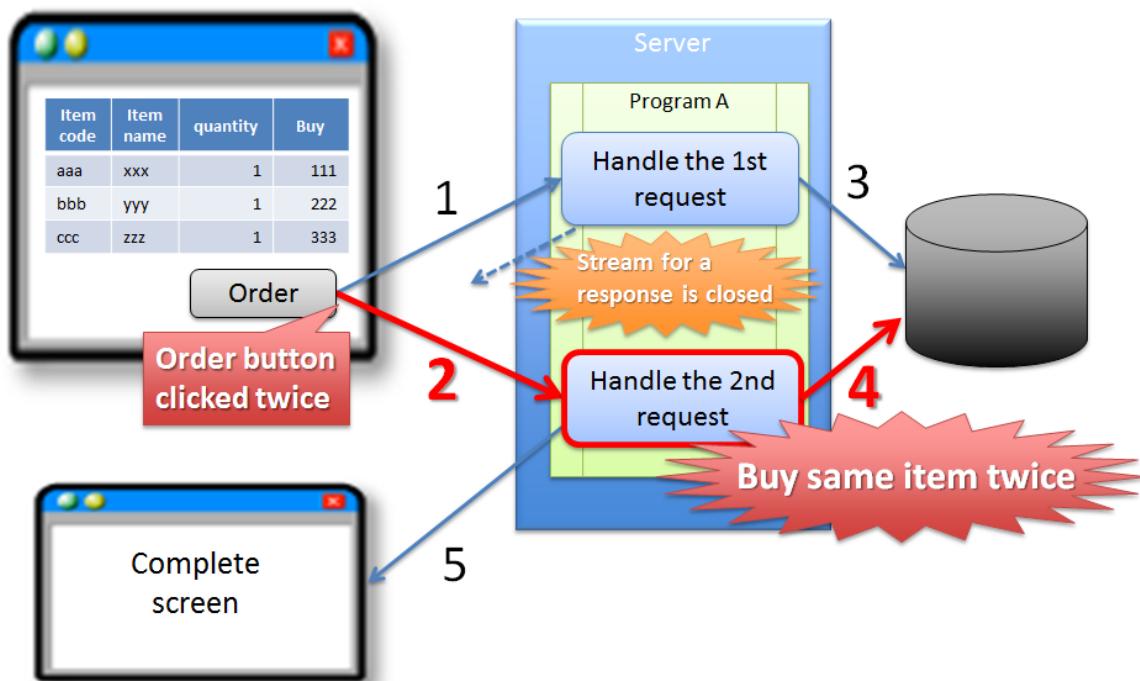
Each problem is described in detail below.

Double clicking of ‘Update’ button

The following problems occur when the button to perform update process is clicked repeatedly.

The example of “Product Purchase” on a shopping site is given below to explain the problems that are likely to occur if the necessary measures are not taken.

Shopping Site



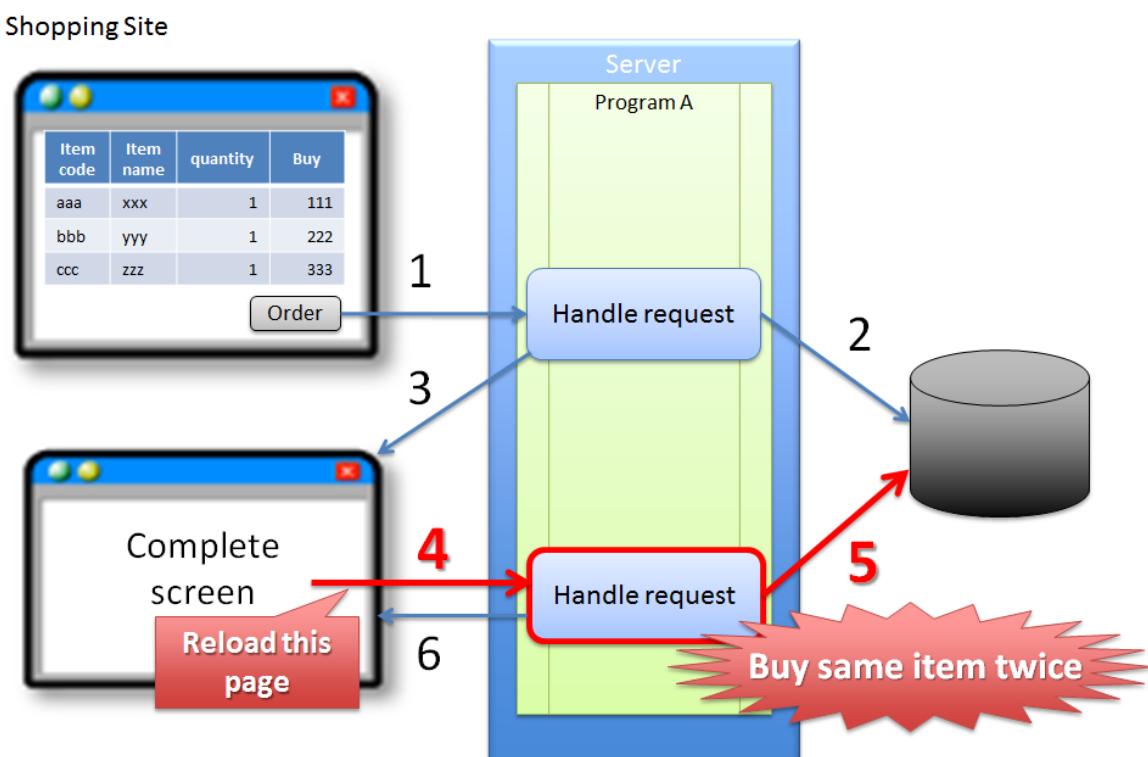
Sr. No.	Description
(1)	Buyer clicks 'Order' button on Product Purchase screen.
(2)	Buyer accidentally clicks 'Order' button again before receiving the response of (1).
(3)	Server updates DB based on purchase of the product received through request (1).
(4)	Server updates DB based on purchase of the product received through request (2).
(5)	Server sends response with a "Purchase Complete" screen for the product received through request (2).

Warning: In the above case, since buyer accidentally clicks ‘Order’ button again, it results in duplicate purchase of same product. Although the problem can be attributed to erroneous operation by the buyer, it is desirable to have the application design such that the above problems do not occur.

Reloading of screen after completion of update process

The following problems occur when the screen is reloaded after completion of update process.

The example of “Product Purchase” on a shopping site is given below to explain the problems that are likely to occur if the necessary measures are not taken.



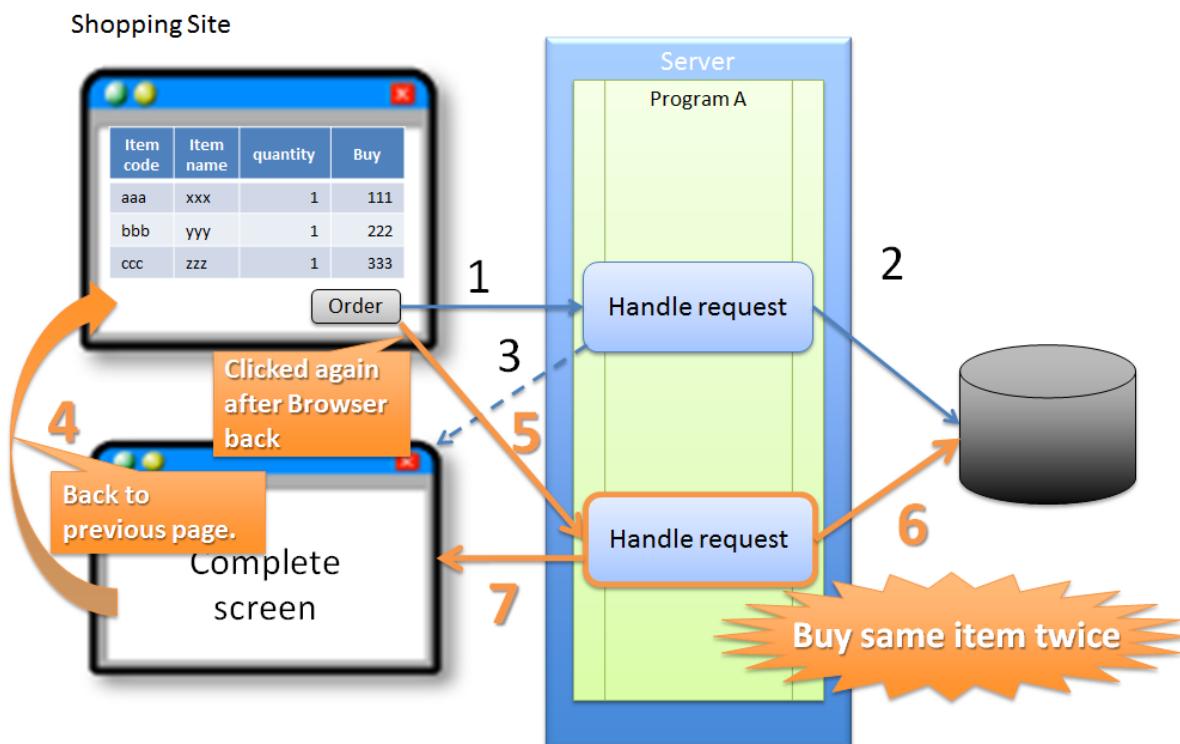
Sr. No.	Description
(1)	Buyer clicks ‘Order’ button on “Product Purchase” screen.
(2)	Server updates DB based on purchase of the product received through request (1).
(3)	Server sends response with a “Purchase complete” screen for the product received through request (1).
(4)	Buyer accidentally executes Reload functionality of the browser.
(5)	Server updates DB based on the purchase of the product received through request (4).
(6)	Server sends response with a “Purchase Complete” screen for the product received through request (4).

Warning: In the above case, since buyer accidentally executes Reload functionality of the browser, **it results in duplicate purchase of same product**. Although the problem can be attributed to erroneous operation by the buyer, it is desirable to have the application design such that the above problems do not occur.

Invalid screen transition using ‘Back’ button of the browser

The following problems occur if invalid screen transition is performed using ‘Back’ button of the browser.

The example of “Product Purchase” on a shopping site is given below to explain the problems that are likely to occur if the necessary measures are not taken.

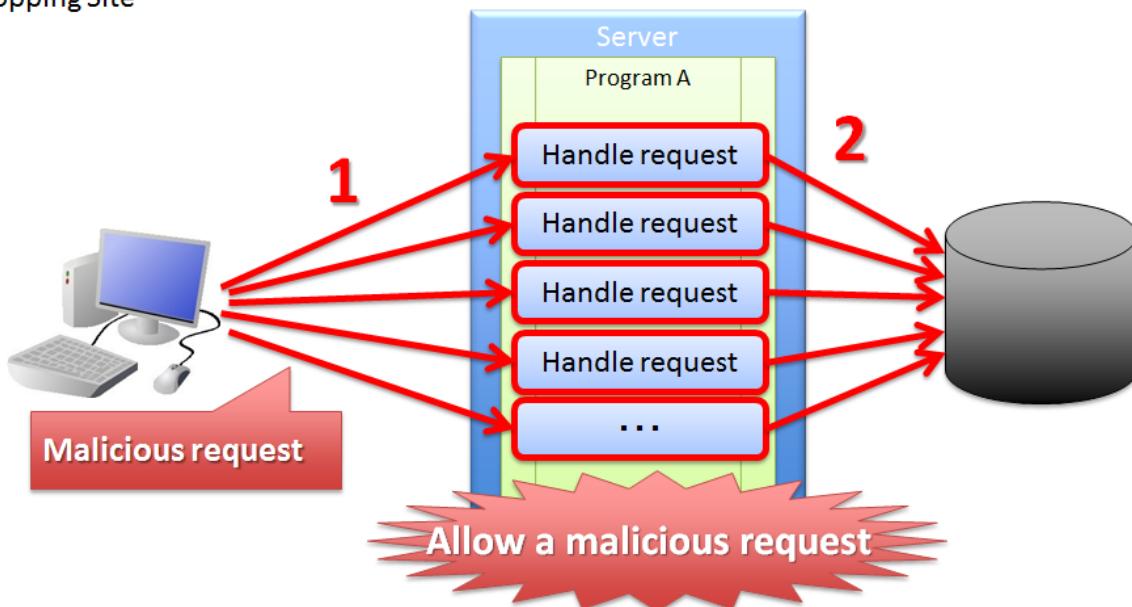


Sr. No.	Description
(1)	Buyer clicks ‘Order’ button on “Product Purchase” screen.
(2)	Server updates DB based on the purchase of the product received through request (1).
(3)	Server sends response with a “Purchase complete” screen of the product received through request (1).
(4)	Buyer uses ‘Back’ button of the browser to go back to “Product Purchase” screen.
(5)	Buyer again clicks ‘Order’ button on “Product Purchase” screen which is re-displayed by clicking ‘Back’ button of the browser.
(6)	Server updates DB based on the purchase of the product received through request (5).
864	<p style="text-align: center;">5 Architecture in Detail - TERASOLUNA Global Framework</p> <p>(7) Server sends response with a “Purchase Complete” screen for the product received through request (5).</p>

Note: In the above case, since buyer does not perform any erroneous operation, the problem is not attributed to the buyer.

However, if update process gets executed even after performing invalid screen operations, the following problems occur.

Shopping Site



Warning: As described above, update process getting executed even after performing invalid screen operations increases the risk of direct updates by a malicious attacker bypassing a valid route.

Sr. No.	Description
(1)	Attacker executes request for processing a direct product purchase without going through a valid screen transition.
(2)	Server cannot detect that the request is getting executed through an invalid route; hence it updates DB based on the purchase of the product received through that request.

Execution of purchase process through an invalid request increases the load on each server resulting in inability to purchase products through a valid route. As a result, the problem causes a ripple effect for the users who purchase the products through valid routes. Hence, it is desirable to have the application design such that the above problems do not occur.

Solutions

The following measures should be taken to resolve the problems described above.

In view of malicious operations such as tampering with requests, **(3) “Applying transaction token check” is mandatory.**

Sr. No.	Solution	Overview
(1)	Preventing double clicking of a button using JavaScript	When a button to perform update process is clicked, the button control using JavaScript prevents the submission of a request if the button is clicked again.
(2)	Applying PRG (Post-Redirect-Get) pattern	A redirect command is returned as a response to the request for performing update process (request by POST method) and then a screen for transition is returned as a response of GET method which is automatically requested from a browser. When a PRG pattern is used, the request generated while reloading the page after the screen is displayed is a GET method; hence re-execution of update process can be prevented.
(3)	Applying transaction token check	Issue a token value for each screen transition and compare the token value sent from browser with the token value stored on the Server to make sure that invalid screen operations do not occur in the transaction. Implementation of transaction token check can prevent re-execution of update process after the page is reloaded using ‘Back’ button of the browser. Deleting the token value stored on the Server after performing the token check can prevent double submission as a Server side process.

Note: When only transaction token check is performed, even a simple operational mistake can lead to transaction token error which in turn results in a low-usability application for the user.

To ensure usability as well as to prevent the problems that occur due to double submission, measures such as “Preventing double clicking of a button using JavaScript” and “Applying PRG (Post-Redirect-Get) pattern” are necessary.

** Although this guideline recommends that you implement all the measures, the decision should be taken depending on application requirements.**

Warning: In Ajax and Web services, since it is difficult to transfer transaction tokens which change for each request, transaction token check need not be used. In Ajax, double submit protection should be performed using only one of the above measures i.e. “Preventing double clicking of a button using JavaScript”.

Todo

TBD

There is further scope for reviewing the check methods in Ajax and Web services.

Preventing double clicking of a button using JavaScript

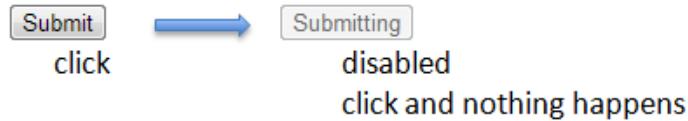
Prevent double clicking of buttons like button to perform update process or button which is used to perform time-consuming search process.

When a button is clicked, use JavaScript to disable that button or link.

Typical examples of control used for disabling a button or link are given below

1. By disabling the button or link so that it cannot be clicked
2. By maintaining a flag for tracking process status and displaying notification “Process in progress” when the button or link is clicked in the middle of the process.

The image when a button is disabled will be as follows:



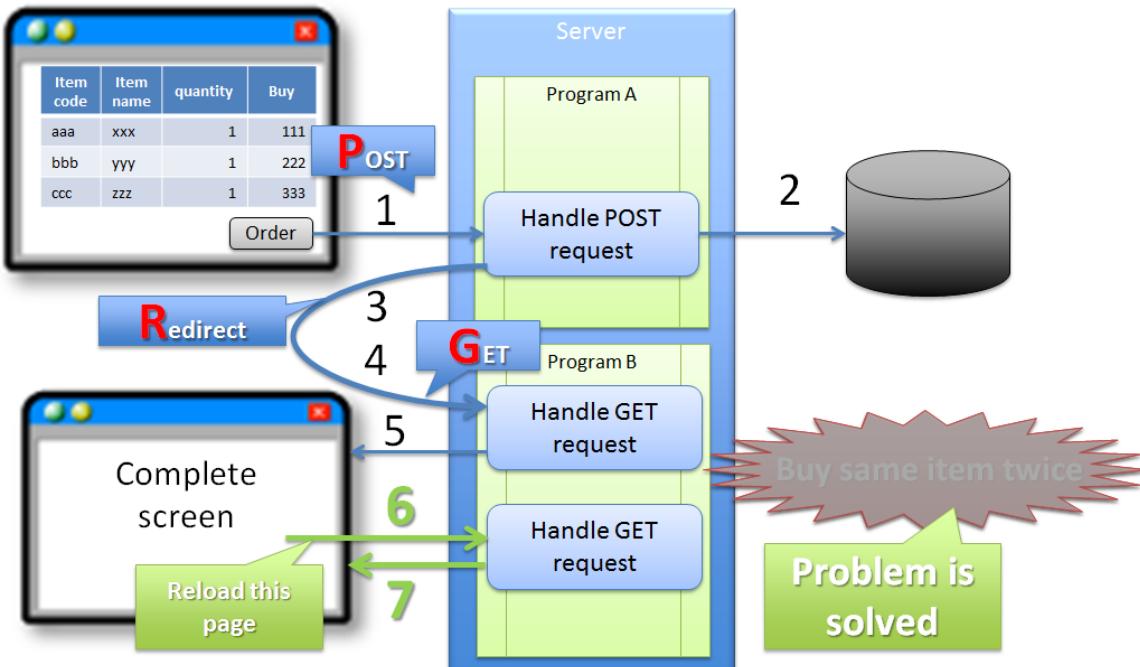
Warning: If all the buttons and links on the screen are disabled, the screen operations can no longer be performed if there is no response from the Server. Therefore, it is recommended not to disable buttons or links that execute events such as “Return to previous screen” or “Go to top screen” etc.

About PRG (Post-Redirect-Get) pattern

A redirect command is returned as a response to the request for performing update process (request by POST method) and then a screen for transition is returned as a response of GET method which is automatically requested from a browser.

When a PRG pattern is used, the request generated while reloading the page after the screen is displayed is a GET method; hence re-execution of update process can be prevented.

Shopping Site



Sr. No.	Description
(1)	Buyer clicks 'Order' button on "Product Purchase" screen. The request is submitted using POST method.
(2)	Server updates DB based on the purchase of the product received through request (1).
(3)	Server sends a redirect response for the URL to display the "Purchase Complete" screen for the product.
(4)	Browser submits request for the URL to display the "Purchase Complete" screen for the product. The request is submitted using GET method.
(5)	Server sends response with a "Purchase Complete" screen for the product.
5.12. Double Submit Protection	869
(6)	Buyer accidentally executes Reload functionality of the browser. The request called by Reload functionality displays "Purchase Complete" screen of the product; hence update process is not re-executed .

Note: It is recommended to use PRG pattern for the processes associated with update process and implement a control so that a request of GET method is sent when ‘Refresh’ button of the browser is clicked.

Warning: In the PRG pattern, the re-execution of update process cannot be prevented by clicking ‘Back’ button of the browser on Completion screen. A transaction token check must be performed to prevent re-execution of update process after an invalid screen transition using ‘Back’ button of the browser.

Transaction Token Check

Transaction token check consists of the following 3 processes.

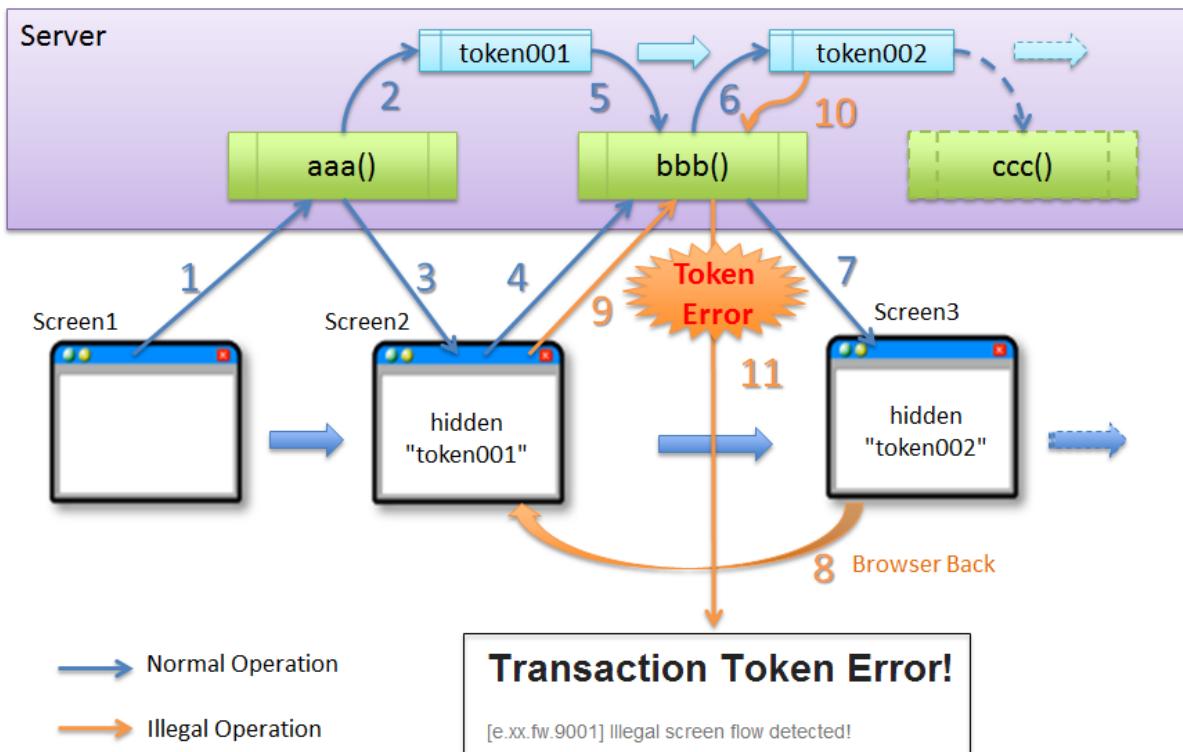
- When a request is received from Client, the Server stores a value (hereafter referred to as transaction token) for uniquely identifying a transaction on the Server.
- Server passes the transaction token to the Client. In case of a Web application with screens, it passes the transaction token to the Client using hidden tag of form.
- When submitting the next request, Client sends the transaction token that was passed from the Server. Server compares the transaction token received from the Client and the transaction token stored on the Server.

When the transaction token sent in the request does not match with the transaction token stored on the Server, it is treated as invalid request and an error is returned.

Warning: Misuse of transaction token check leads to poor usability of the application; hence the scope of its usage should be defined by considering the following points.

- It is not necessary to include reference-type requests that do not involve data update and requests that perform only screen transitions in the scope of transaction token check.
If the scope of transactions is extended unnecessarily, transaction token errors are more likely to occur which in turn reduces the usability of the application.
- Transaction token check is not mandatory for the processes wherein there is no problem even if the data gets updated multiple times from business perspective (user information update etc.).
- Transaction token check is mandatory for the processes such as deposit process or product purchase process etc. wherein there is a risk of duplicate execution.

The process flow when expected operations are performed and process flow when unexpected operations are performed using transaction token check are shown below.



The process flow when expected operations are performed is as follows:

Sr. No.	Description
(1)	Client sends the request.
(2)	Server creates the transaction token (token001) and stores it on the Server.
(3)	Server passes the created transaction token (token001) to the Client.
(4)	Client sends the request along with the transaction token (token001).
(5)	Server checks whether the transaction token (token001) stored on the Server and the transaction token (token001) submitted by the Client are same. Since the values are same, the request is considered as valid.
(6)	Server generates transaction token (token002) to be used in the next request and updates the value stored on the Server. At this point, the transaction token (token001) is discarded.
(7)	Server passes the updated transaction token (token002) to the Client.

The process flow when unexpected operations are performed is as follows:

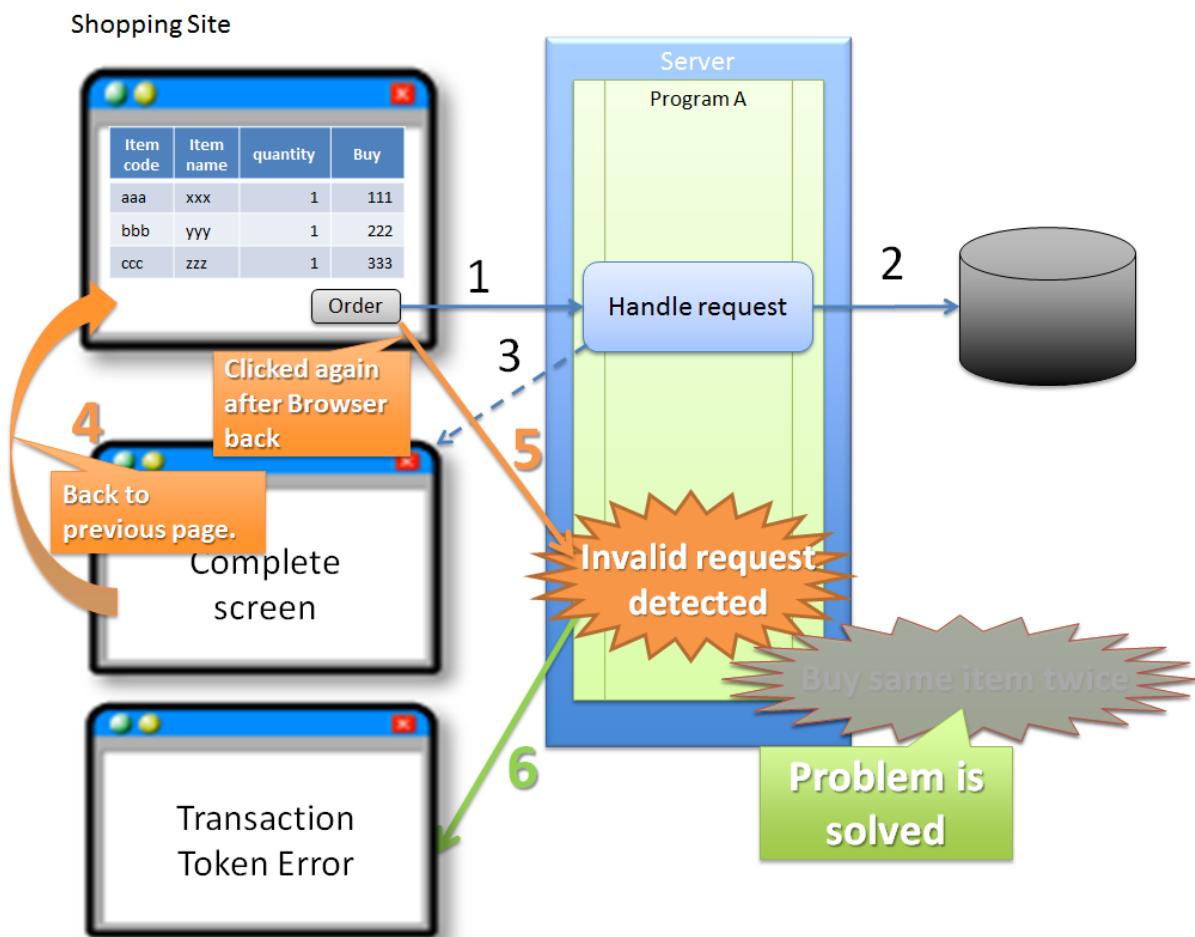
Here, ‘Back’ button of the browser is taken as an example; however, this is also applicable for the direct requests from shortcuts etc.

Sr. No.	Description
(8)	'Back' button of the browser on Client side is clicked.
(9)	Request is sent from Client side along with the transaction token (token001) of the screen which is displayed after clicking 'Back' button.
(10)	Server checks whether the transaction token (token002) stored on the Server and the transaction token (token001) submitted by the Client are same. Since the values are different, the request is considered as invalid and a transaction token error is thrown.
(11)	Server sends response with an error screen to notify that a transaction token error has occurred.

The 3 events described below can be prevented by transaction token check.

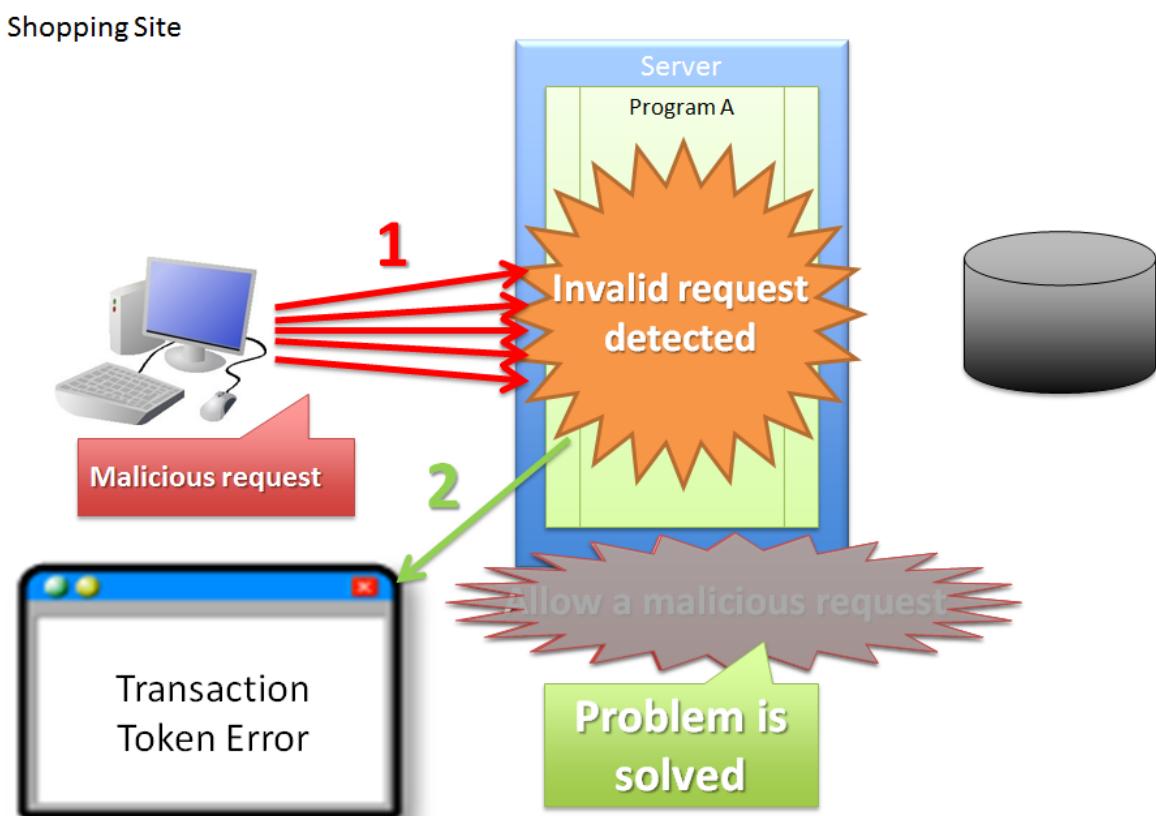
- Invalid screen transition in case of a business process that requires fixed screen transition
- Data update due to invalid requests that do not involve valid screen transitions
- Duplicate execution of update process due to double submission

Invalid screen transition in case of a business process that requires fixed screen transition, can be prevented by the flow shown below.



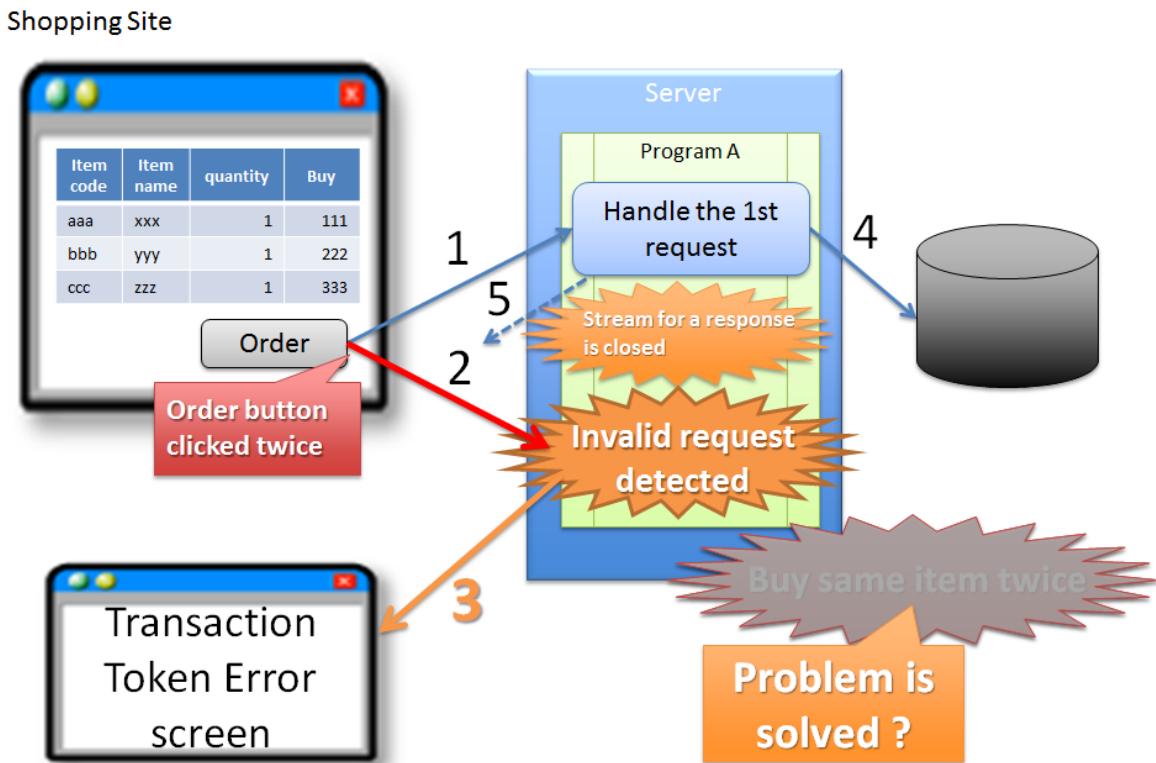
Sr. No.	Description
(1)	<p>Buyer clicks ‘Order’ button on “Product Purchase” screen.</p> <p>Since the transaction token stored on the Server and the transaction token submitted by the Client match, the process to purchase the product is executed.</p> <p>At this time, the value of the transaction token stored on the Server is discarded and updated to a new token value.</p>
(2)	Server updates DB based on the purchase of the product received through request (1).
(3)	Server sends response with a “Purchase Complete” screen for the product received through request (1).
874	<p>5 Architecture in Detail - TERASOLUNA Global Framework</p> <p>(4) Buyer uses ‘Back’ button of the browser to go back to “Product Purchase” screen.</p>

Data updated by an invalid request which does not involve a valid screen transition can be prevented by the flow shown below.



Sr. No.	Description
(1)	<p>Attacker sends a request to purchase the product directly without performing a valid screen transition.</p> <p>Since the request for generating a transaction token is not executed, a transaction token error occurs.</p>
(2)	Server sends response with an error screen to notify that a transaction token error has occurred.

Duplicate execution of update process at the time of double submission can be prevented by the flow shown below.



Sr. No.	Description
(1)	<p>Buyer clicks ‘Order’ button on “Product Purchase” screen.</p> <p>Since the transaction token stored on the server and the transaction token submitted by the Client match, the process to purchase the product is executed.</p> <p>At this time, the value of the transaction token that is stored on the server is discarded and updated to a new token value.</p>
(2)	<p>Buyer accidentally clicks ‘Order’ button again before the response of (1) is returned.</p> <p>Since the transaction token sent by the Client is a value which has already been discarded, a transaction token error occurs when process of (1) is executed.</p>
(3)	<p>Server sends response with an error screen to notify that a transaction token error has occurred for the request (2).</p>
(4)	<p>Server updates DB based on the purchase of the product received through request (1).</p>
(5)	<p>Server attempts to respond with a “Purchase Complete” screen for the product received through request (1); however since the stream for responding to the request of (1) is closed due to the transmission of the request of (2), it fails to send response with a “Purchase Complete” screen.</p>

Warning: This can prevent duplicate execution of update process at the time of double submission; however this does not resolve the problem of server not being able to respond with a screen to notify the completion of process. Therefore, it is also recommended to deal with this problem by preventing double clicking of a button using JavaScript.

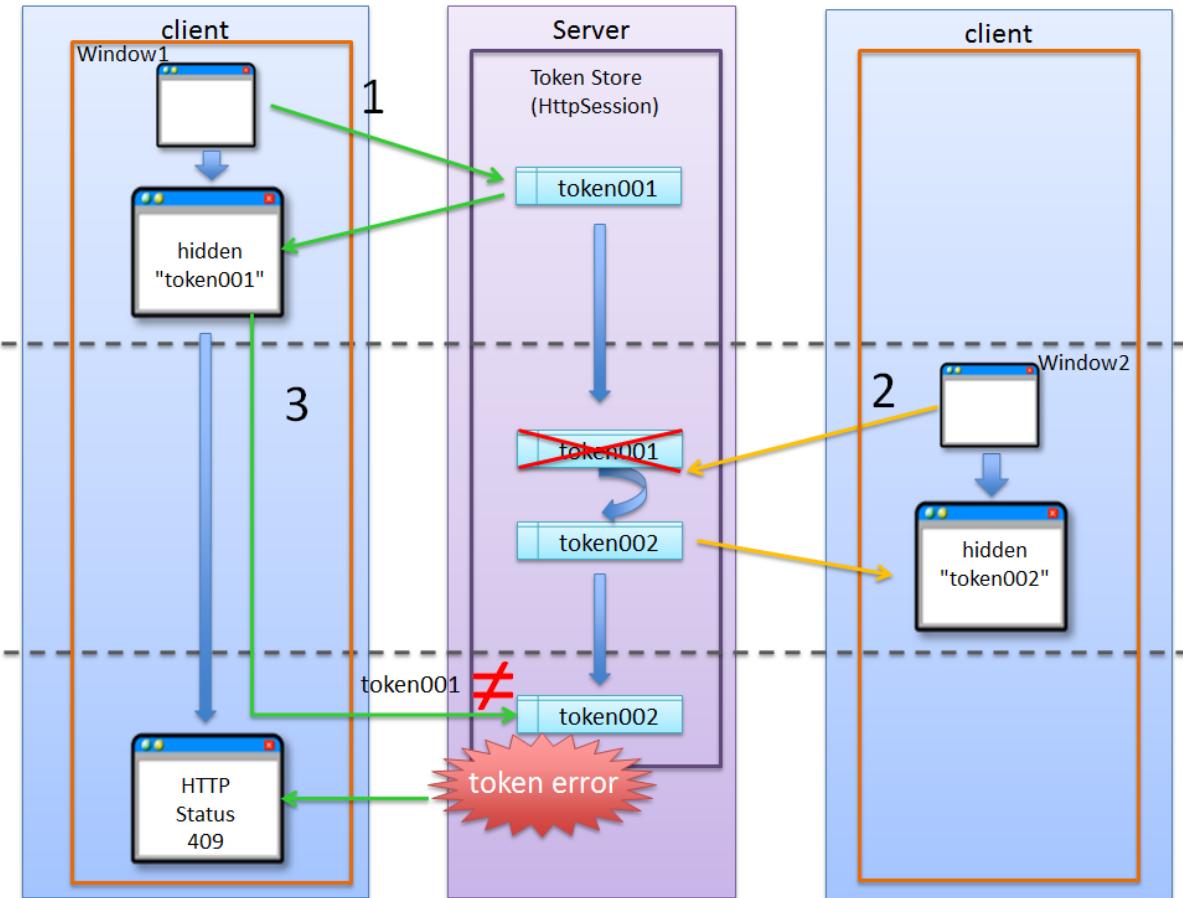
About NameSpace of transaction token

In the transaction token check functionality provided by the common library, it is possible to set a NameSpace in the container for storing transaction token. This enables parallel execution of update process using a tab browser or multiple windows.

Problems that occur when there is no NameSpace

Let us see the problems that occur when there is no NameSpace.

The figure below illustrates an example wherein two clients are arranged side by side; basically 2 windows are launched on same machine.



Sr. No.	Description
(1)	<p>The request is sent from Window 1, then the transaction token (token001) received as a response is stored in the browser.</p> <p>The transaction token stored on the server is considered token001.</p>
(2)	<p>The request is sent from Window 2, then the transaction token (token002) received as a response is stored in the browser.</p> <p>The transaction token stored on the server is considered token002. The transaction token (token001) generated by the process (1) is discarded at this point.</p>
(3)	<p>The request is sent from Window 1 along with the transaction token (token001) stored in the browser.</p> <p>Since the transaction token stored on the server (token002) and the transaction token sent by the request (token001) do not match, the request is considered as invalid resulting in a transaction token error.</p>

Warning: The update process cannot be executed concurrently in absence of NameSpace; hence the application becomes a low-usability application.

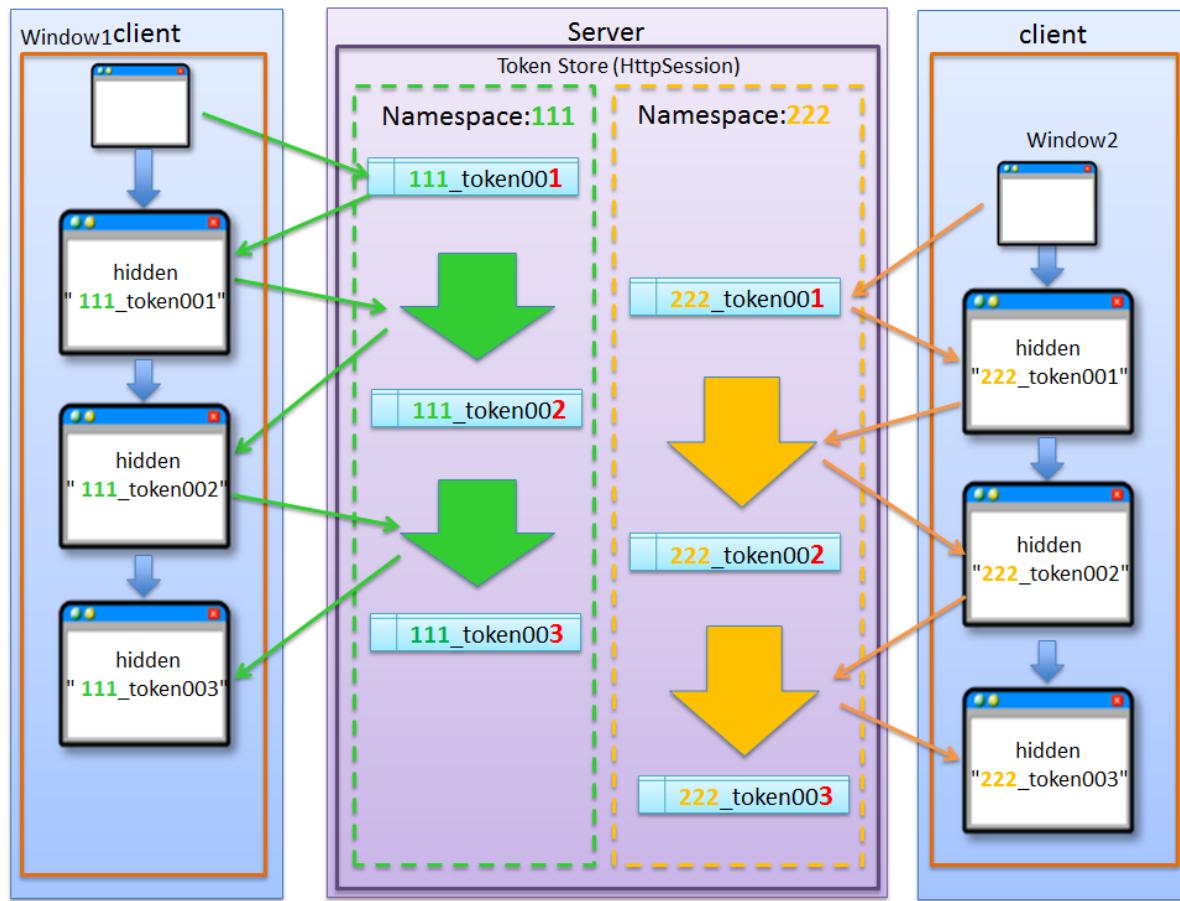
Behavior when NameSpace is specified

Let us now see the behavior when NameSpace is specified.

The problem wherein the update process could not be executed concurrently in the absence of NameSpace can now be resolved by specifying a NameSpace.

The figure below illustrates an example wherein two clients are arranged side by side; basically 2 windows are launched on same machine.

NameSpaces are shown as 111, 222 in the figure.



** When a NameSpace is specified, the transaction token in the NameSpace allocated to the transaction is handled independently. Hence, it does not affect the transactions of another NameSpace.**

Here, even though the browser is explained using different Windows, the tab browser works in the same way. For generated keys and usage method, refer to [Using transaction token check](#).

5.12.2 How to use

Preventing double clicking of button using JavaScript

Double clicking of a button on Client side can be prevented using JavaScript.

Once a button is clicked, it should not be possible to click it again till it is re-generated.

Todo

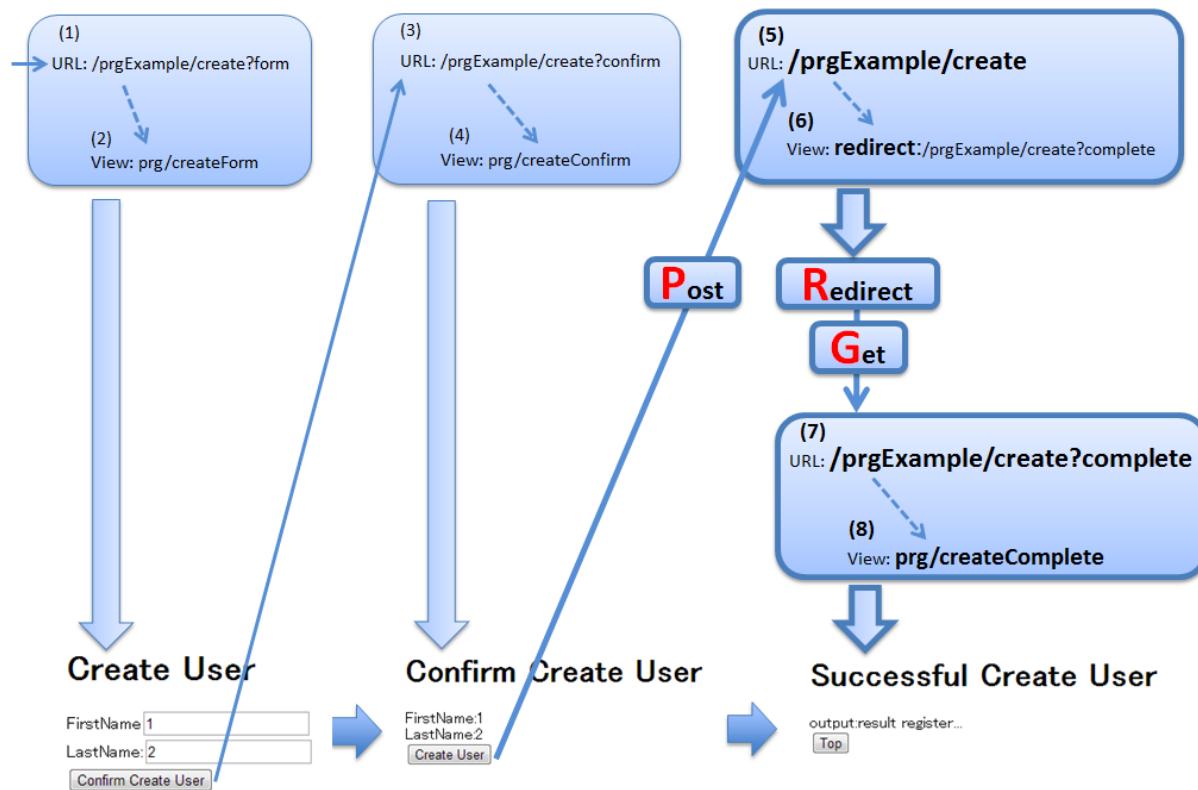
TBD

The check method in JavaScript will be described in detail in subsequent versions.

Using PRG (Post-Redirect-Get) pattern

The example of implementing PRG (Post-Redirect-Get) pattern is given below.

The application which involves a simple screen transition such as Input Screen-> Confirmation Screen-> Completion Screen is taken as an example.



The image numbers and comment number of source code are linked.

However, since (1)-(4) is not directly related to the PRG pattern, the explanation is omitted.

- Controller

```
@Controller
@RequestMapping("prgExample")
public class PostRedirectGetExampleController {

    @Inject
```

```
UserService userService;

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

@RequestMapping(value = "create",
               method = RequestMethod.GET,
               params = "form") // (1)
public String createForm(
    PostRedirectGetForm postRedirectGetForm,
    BindingResult bindingResult) {
    return "prg/createForm"; // (2)
}

@RequestMapping(value = "create",
               method = RequestMethod.POST,
               params = "confirm") // (3)
public String createConfirm(
    @Validated PostRedirectGetForm postRedirectGetForm,
    BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return "prg/createForm";
    }
    return "prg/createConfirm"; // (4)
}

@RequestMapping(value = "create",
               method = RequestMethod.POST) // (5)
public String create(
    @Validated PostRedirectGetForm postRedirectGetForm,
    BindingResult bindingResult,
    RedirectAttributes redirectAttributes) {
    if (bindingResult.hasErrors()) {
        return "prg/createForm";
    }
}

// omitted

String output = "result register..."; // (6)
redirectAttributes.addFlashAttribute("output", output); // (6)
return "redirect:/prgExample/create?complete"; // (6)
}

@RequestMapping(value = "create",
               method = RequestMethod.GET,
               params = "complete") // (7)
public String createComplete() {
    return "prg/createComplete"; // (8)
}
```

```

    }
}

```

Sr. No.	Description
(5)	A processing method to perform a process when ‘Register’ button (Create User button) on Confirmation screen is clicked. The request is received by POST method.
(6)	It is redirected to URL for displaying Completion screen. In the above example, a request is sent to URL "prgExample/create?complete" by GET method. When data is to be delivered to redirect destination, addFlashAttribute method of RedirectAttributes is called and the data to be delivered is added. addAttribute method of Model cannot deliver data to the redirect destination.
(7)	A processing method to display Completion screen. A request is received by GET method.
(8)	View (JSP) is called to display the Completion screen and responds with Completion screen. Since the extension of JSP is assigned by ViewResolver defined in spring-mvc.xml, it is omitted from the return value of the processing method.

Note:

- At the time of redirecting, assign “redirect:” as the prefix of transition information to be returned by the processing method as the return value.
- When the data is to be delivered to the process of redirect destination, call addFlashAttribute method of RedirectAttributes and add the data to be delivered.

- createForm.jsp

```

<h1>Create User</h1>
<div id="prgForm">
  <form:form
    action="${pageContext.request.contextPath}/rpgExample/create"
    method="post" modelAttribute="postRedirectGetForm">

```

```
<form:label path="firstName">FirstName</form:label>
<form:input path="firstName" /><br>
<form:label path="lastName">LastName:</form:label>
<form:input path="lastName" /><br>
<form:button name="confirm">Confirm Create User</form:button>
</form:form>
</div>
```

- createConfirm.jsp

```
<h1>Confirm Create User</h1>
<div id="prgForm">
<form:form
    action="${pageContext.request.contextPath}/rpgExample/create"
    method="post"
    modelAttribute="postRedirectGetForm">
    FirstName:${f:h(postRedirectGetForm.firstName)}<br>
    <form:hidden path="firstName" />
    LastName:${f:h(postRedirectGetForm.lastName)}<br>
    <form:hidden path="lastName" />
    <form:button>Create User</form:button> <%-- (6) --%>
</form:form>
</div>
```

Sr. No.	Description
(6)	When the button to perform update process is clicked, a request is sent by POST method .

- createComplete.jsp

```
<h1>Successful Create User Completion</h1>
<div id="prgForm">
<form:form
    action="${pageContext.request.contextPath}/rpgExample/create"
    method="get" modelAttribute="postRedirectGetForm">
    output:${f:h(output)}<br> <%-- (7) --%>
    <form:button name="backToTop">Top</form:button>
</form:form>
</div>
```

Sr. No.	Description
(7)	When the data delivered from update process is to be referred at the redirect destination, specify the attribute name of the data added by the addFlashAttribute method of RedirectAttributes . In the above example, "output" is the attribute name to refer to the delivered data.

Using transaction token check

The example of implementation using transaction token check is given below.

Transaction token check functionality is provided by the common library and not by Spring MVC.

Transaction token check provided by common library

Transaction token check functionality of common library provides `@org.terasoluna.gfw.web.token.transaction.TransactionTokenCheck` annotation to perform the following tasks:

- Creation of NameSpace for transaction token
- Starting the transaction
- Token value check in the transaction
- Ending the transaction

The transaction token check can be performed declaratively by assigning `@TransactionTokenCheck` annotation for the Controller class and the processing methods of the Controller class.

Attributes of `@TransactionTokenCheck` annotation

The attributes that can be specified in `@TransactionTokenCheck` annotation are explained below.

Table.5.22 @TransactionTokenCheckAnnotation Parameter List

Sr. No.	Attribute Name	Contents	default	Example
1.	value	Any character string. Used as NameSpace.	None	value = "create" If there is only 1 argument, the "value =" part can be omitted.
2.	type	BEGIN A transaction token is created and a new transaction is started. IN Transaction token is validated. When the requested token value and the token value stored on the server match, the token value of transaction token is updated.	IN	type = Transaction-TokenType.BEGIN type = Transaction-TokenType.IN

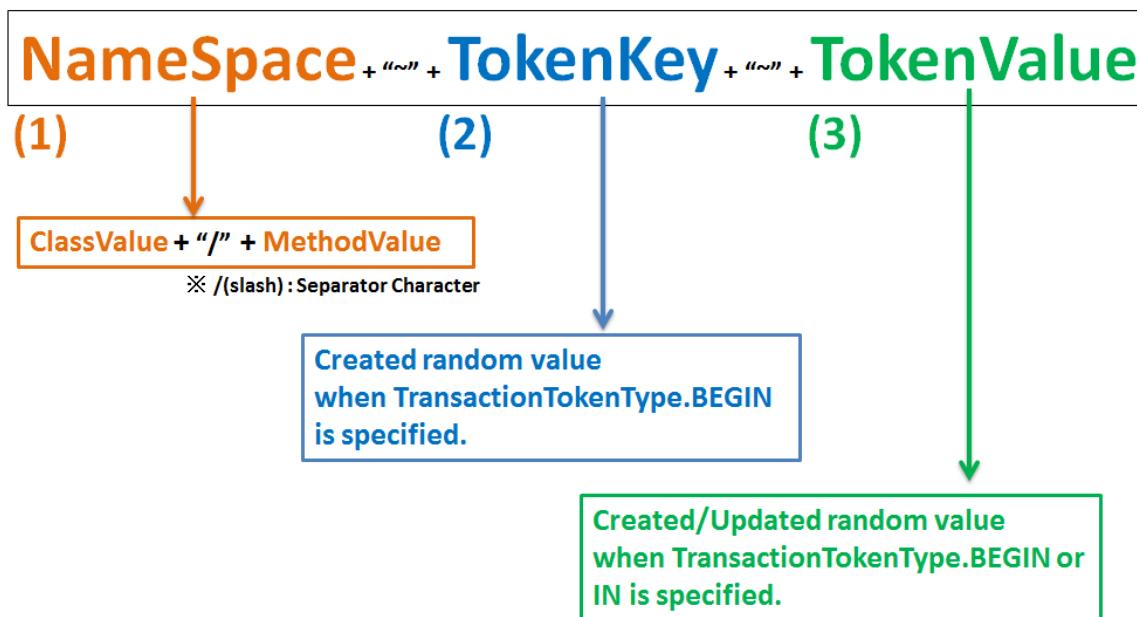
Note: It is recommended that the value to be set in "value" attribute should be same as the config value of "value" attribute for @RequestMapping annotation.

Note: In "type" attribute, **NONE** and **END** can be specified; however, the description is omitted as normally they are not used.

Format of transaction token

Format for the transaction token used in the transaction token check of common library is as follows:

※ ~(tilde) : Separator Character



ex)

admin/staff/create~(Random value of 32 chars)~(Random value of 32 chars) ←

```
@Controller
@RequestMapping("admin/staff")
@TransactionalTokenCheck("admin/staff")
public class StaffController{
    @TransactionalTokenCheck("create", type = TransactionTokenType.BEGIN)
    public String createAbb( ... )
    @TransactionalTokenCheck("create", type = TransactionTokenType.IN)
    public String createBbb( ... )
}
```

Sr. No.	Components	Description
(1)	NameSpace	<ul style="list-style-type: none"> • NameSpace is an element for assigning a logical name to identify a series of screen transitions. • By setting a NameSpace, it is possible to prevent the requests belonging to different Namespaces from interfering with each other and it is also possible to increase the number of screen transitions that can operate in parallel. • The value specified in the “value” attribute of <code>@TransactionTokenCheck</code> annotation is used as the value to be used for NameSpace. • When both “value” attribute of the “class” annotation and “value” attribute of the “method” annotation are specified, the value which concatenates both the values with “/” is used as NameSpace. When the same value is specified in multiple methods, the methods belong to same NameSpace. • When the “value” attribute is specified only in “class” annotation, all the NameSpaces of the transaction tokens generated in that class will be the value specified in “class” annotation. • When the “value” attribute is specified only in “method” annotation, the NameSpace of the generated transaction tokens will be the value specified in “method” annotation. When the same value is specified in multiple methods, the methods belong to same NameSpace. • When both “value” attribute of “class” annotation and “value” attribute of “method” annotation are omitted, the method belong to the global token. For global token, refer to Global Tokens.
(2)	TokenKey	<ul style="list-style-type: none"> • TokenKey is an element for identifying the transactions stored in the Namespace. • TokenKey is generated upon execution of a method wherein <code>TransactionTokenType.BEGIN</code> is declared in the “type” attribute of <code>@TransactionTokenCheck</code> annotation. • A maximum limit exists for the number of multiple TokenKeys which can be concurrently stored and the default number is 10. The count of stored TokenKeys is managed for each NameSpace. • When the number of TokenKeys stored for each NameSpace reaches the maximum value at the time of <code>TransactionTokenType.BEGIN</code>, a new transaction will be stored as a valid transaction by discarding the TokenKey with the oldest date and time of execution (Least Recently Used (LRU)). • When the access is made by using the discarded transaction token, a transaction token error is thrown.
(3)	TokenValue	<ul style="list-style-type: none"> • TokenValue is an element for storing the token value of the transaction. • TokenValue is generated upon execution of the 5 Architecture in Detail - TERASOLUNA Global Framework method wherein <code>TransactionTokenType.BEGIN</code> or <code>TransactionTokenType.IN</code> is declared in the “type” attribute of <code>@TransactionTokenCheck</code> annotation.

Warning: When the “value” attribute is specified only in “method” annotation and if the same value is specified in another Controller, it should be noted that it will be handled as a request for carrying out a series of screen transitions. “value” attribute should be specified by this method only when the screen transitions across Controllers are to be treated as the same transaction.

Basically, it is recommended not to use the method wherein “value” attribute is specified only in “method” annotation.

Note: The method for specifying NameSpace is classified according to creation granularity of the Controller,

- when “value” attributes of both “class” annotation and “method” annotation are specified
 - when “value” attribute is specified only in “class” annotation
1. When a processing method which corresponds to multiple usecases is to be implemented in Controller, “value” attributes of both “class” annotation and “method” annotation are specified.
For example, this pattern is used when registration, change, deletion of users is to be implemented in a single Controller.
 2. When a processing method which corresponds to a single usecase is to be implemented in Controller, “value” attribute is specified only in “class” annotation.
For example, this pattern is used when a Controller is implemented for every registration, modification, deletion of users.
-

Lifecycle of transaction token

The lifecycle (Generate, Update, Discard) control of transaction token is performed in the following scenarios.

Sr. No.	Lifecycle Control	Description
(1)	Token Generation	A new token is generated and transaction is started when the processing of the method wherein <code>TransactionTokenType.BEGIN</code> is specified in “type” attribute of <code>@TransactionTokenCheck</code> annotation, is terminated.
(2)	Token Update	The token (<code>TokenValue</code>) is updated and transaction is continued when the processing of the method wherein <code>TransactionTokenType.IN</code> is specified in “type” attribute of <code>@TransactionTokenCheck</code> annotation, is terminated.
(3)	Token Discard	<p>The tokens are discarded in any of the following scenarios and the transaction is terminated.</p> <p>[1] When the method wherein “<code>TransactionTokenType.BEGIN</code>“ is specified in “type” attribute of <code>@TransactionTokenCheck</code> annotation, is called, the transaction token specified in the request parameter is discarded and unnecessary transaction is terminated.</p> <p>[2] If a new transaction starts when number of transaction tokens (<code>TokenKey</code>) that can be stored in NameSpace has reached the maximum limit, the transaction token with the oldest date and time of execution is discarded and the transaction is forcibly terminated.</p> <p>[3] When exceptions such as system error occur, the transaction token specified in the request parameter is discarded and the transaction is terminated.</p>

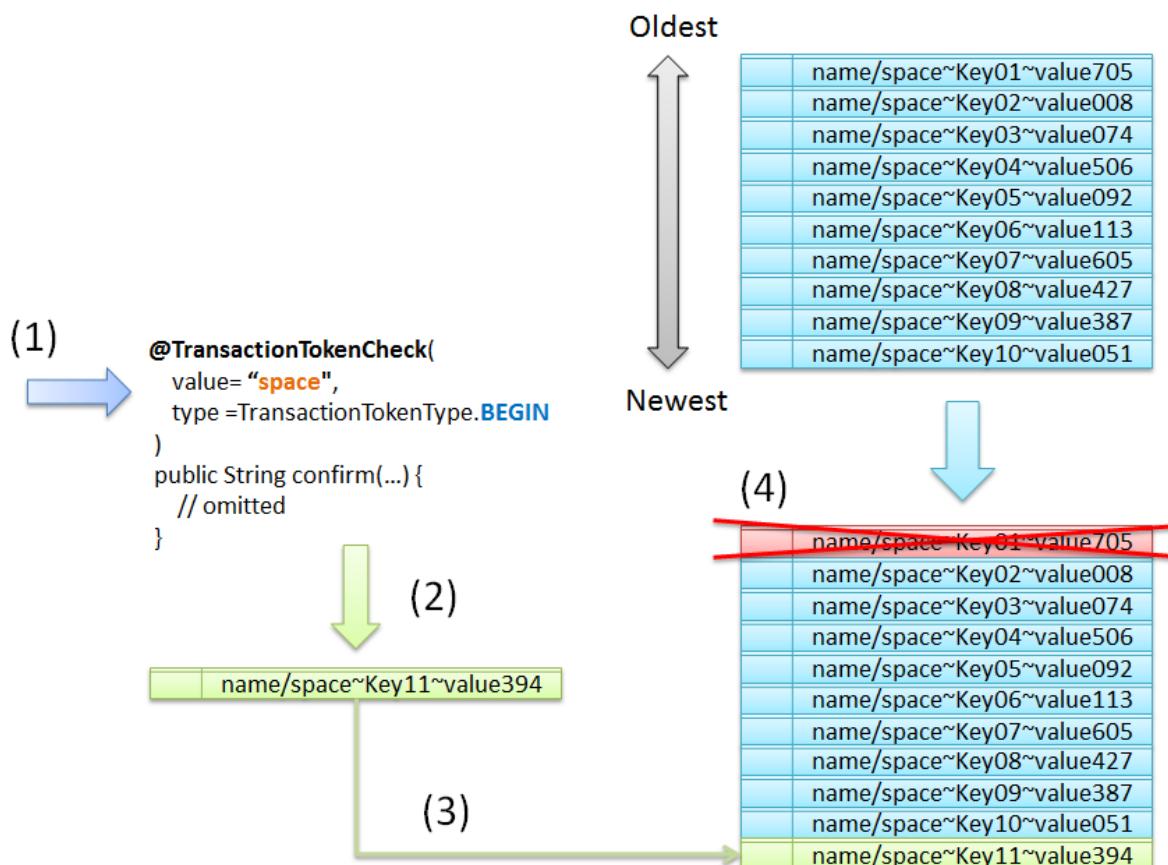
Note: A maximum limit is provided for the number of transaction tokens (`TokenKey`) which can be stored in a NameSpace. When the maximum limit is reached while generating a new transaction token, a new transaction is managed as a valid transaction, by discarding the transaction token which has the `TokenKey` with the oldest date and time of execution (Least Recently Used (LRU)).

The maximum limit of transaction tokens that can be stored for each NameSpace is 10 by default. To change the maximum limit, refer to [How to change the maximum limit of transaction tokens](#).

The behavior when the maximum limit is reached while generating a new transaction token is explained below.

The pre-requisites are as given below.

- A default value (10) is specified as the maximum limit for the number of transaction tokens that can be stored in the NameSpace.
- `@TransactionTokenCheck("name")` is specified as the class annotation of Controller.
- Transaction tokens of the same NameSpace have reached the maximum limit.



Sr. No.	Description
(1)	A request to start a new transaction is received when the transaction tokens of the same NameSpace have reached the maximum limit.
(2)	A new transaction token is generated.
(3)	The generated transaction token is added to the location where tokens are stored. ** At this point, the transaction tokens that exceed the maximum limit are present in the NameSpace.**
(4)	The transaction tokens exceeding the maximum limit are deleted from the NameSpace. The transaction tokens are deleted in a sequence starting with the transaction token with the oldest date and time of execution.

Settings for using a transaction token check

The settings for using the transaction token check provided by the common library are shown below.

- spring-mvc.xml

```
<mvc:interceptors>
    <mvc:interceptor> <!-- (1) -->
        <mvc:mapping path="/**" /> <!-- (2) -->
        <mvc:exclude-mapping path="/resources/**" /> <!-- (2) -->
        <mvc:exclude-mapping path="/**/*.html" /> <!-- (2) -->
    <!-- (3) -->
    <bean
        class="org.terasoluna.gfw.web.token.transaction.TransactionTokenInterceptor" />
</mvc:interceptor>
</mvc:interceptors>

<bean id="requestDataValueProcessor"
    class="org.terasoluna.gfw.web.mvc.support.CompositerequestDataValueProcessor">
    <constructor-arg>
        <util:list>
```

```

<!-- (4) -->
<bean class="org.terasoluna.gfw.web.token.transaction.TransactionTokenRequestData
<!-- omitted -->
</util:list>
</constructor-arg>
</bean>

```

Sr. No.	Description
(1)	Set HandlerInterceptor to generate and check transaction tokens.
(2)	Specify a request path wherein HandlerInterceptor is to be applied. In the above example, it is applicable to all the requests except the requests under /resources and the requests to HTML.
(3)	Specify a class (TransactionTokenInterceptor) to generate and check transaction tokens using @TransactionTokenCheck annotation.
(4)	Set a class (TransactionTokenRequestDataValueProcessor) for automatic embedding of the transaction token to the Hidden area using <form:form> tag of Spring MVC.

Settings for handling transaction token errors

When a transaction token error occurs,

“org.terasoluna.gfw.web.token.transaction.InvalidTransactionTokenException“ is generated.

Therefore, in order to handle transaction token errors, it is necessary to add the handling definition of InvalidTransactionTokenException to the following settings.

- ExceptionCodeResolver defined in applicationContext.xml
- SystemExceptionResolver defined in spring-mvc.xml

For adding the settings, refer to the following:

- *Common Settings*

- *Application Layer Settings*

How to use transaction token check in Controller

In order to perform transaction token check, it is necessary to define the method to start the transaction and the method to carry out the checks in Controller.

The explanation below is about the implementation of processing method required in a single usecase using a controller.

- Controller

```
@Controller
@RequestMapping("transactionTokenCheckExample")
@TransactionTokenCheck("transactionTokenCheckExample") // (1)
public class TransactionTokenCheckExampleController {

    @RequestMapping(params = "first", method = RequestMethod.GET)
    public String first() {
        return "transactionTokenCheckExample/firstView";
    }

    @RequestMapping(params = "second", method = RequestMethod.POST)
    @TransactionTokenCheck(type = TransactionTokenType.BEGIN) // (2)
    public String second() {
        return "transactionTokenCheckExample/secondView";
    }

    @RequestMapping(params = "third", method = RequestMethod.POST)
    @TransactionTokenCheck // (3)
    public String third() {
        return "transactionTokenCheckExample/thirdView";
    }

    @RequestMapping(params = "fourth", method = RequestMethod.POST)
    @TransactionTokenCheck // (3)
    public String fourth() {
        return "transactionTokenCheckExample/fourthView";
    }

    @RequestMapping(params = "fifth", method = RequestMethod.POST)
    @TransactionTokenCheck // (3)
    public String fifth() {
        return "redirect:/transactionTokenCheckExample?complete";
    }

    @RequestMapping(params = "complete", method = RequestMethod.GET)
    public String complete() { // (4)
        return "transactionTokenCheckExample/fifthView";
    }
}
```

}

Sr. No.	Description
(1)	NameSpace is specified in “value” attribute of “class” annotation. In the above example, the value same as the “value” attribute of @RequestMapping which is the recommended pattern of this guideline is specified.
(2)	The transaction is started and a new transaction token is issued. Here, since the transaction tokens are managed at Controller level, “value” attribute of “method” annotation is not specified.
(3)	The transaction token is checked and transaction token value is updated. If the type attribute is omitted, the behavior remains the same as when @TransactionTokenCheck(type = TransactionTokenType.IN) is specified.
(4)	Since it is not necessary to perform transaction token check in the request for displaying a screen which notifies the completion of the usecase, @TransactionTokenCheck annotation is not specified.

Note:

- When BEGIN is specified in the “type” attribute of @TransactionTokenCheck annotation, transaction token is not checked since a new TokenKey is generated.
- When IN is specified in the “type” attribute of @TransactionTokenCheck annotation, it is checked whether the token value specified in the request and the token value stored on the server are the same.

How to use transaction token check in View (JSP)

When transaction token is to be checked, View (JSP) should be implemented in such a way that the issued transaction token is submitted as a request parameter.

A method wherein a transaction is automatically embedded in hidden elements by using <form:form> tag is recommended as a method to submit it as a request parameter after carrying out *Settings for using a transaction token check*.

- firstView.jsp

```
<h1>First</h1>
<form:form method="post" action="transactionTokenCheckExample">
  <input type="submit" name="second" value="second" />
</form:form>
```

- secondView.jsp

```
<h1>Second</h1>
<form:form method="post" action="transactionTokenCheckExample"><!-- (1) -->
  <input type="submit" name="third" value="third" />
</form:form>
```

- thirdView.jsp

```
<h1>Third</h1>
<form:form method="post" action="transactionTokenCheckExample"><!-- (1) -->
  <input type="submit" name="fourth" value="fourth" />
</form:form>
```

- fourthView.jsp

When <form:form>tag is used

```
<h1>Fourth</h1>
<form:form method="post" action="transactionTokenCheckExample"><!-- (1) -->
  <input type="submit" name="fifth" value="fifth" />
</form:form>
```

When “<form>“ tag of HTML is used

```
<h1>Fourth</h1>
<form method="post" action="transactionTokenCheckExample">
  <t:transaction /><!-- (2) -->
  <!-- (3) -->
  <input type="hidden" name="${f:h(_csrf.parameterName)}" value="${f:h(_csrf.token)}"/>
  <input type="submit" name="fifth" value="fifth" />
</form>
```

- fifthView.jsp

```
<h1>Fifth</h1>
<form:form method="get" action="transactionTokenCheckExample">
  <input type="submit" name="first" value="first" />
</form:form>
```

Sr. No.	Description
(1)	When <code><form:form></code> tag is used in JSP and if BEGIN or IN is specified in “type” attribute of <code>@TransactionTokenCheck</code> annotation, the Value of <code>name="__TRANSACTION_TOKEN"</code> is automatically embedded as a hidden tag.
(2)	When <code><form></code> tag of HTML is used, a hidden tag same as (1) is embedded using <code><t:transaction /></code> .
(3)	When <code><form></code> tag of HTML is used, csrf token necessary for CSRF token check provided by Spring Security needs to be embedded as a hidden item. For csrf token necessary for CSRF token check, refer to <i>csrf_formtag-use</i> .

Note: If `<form:form>` tag is used, the parameters necessary for CSRF token check are also automatically embedded. Refer to *csrf_formtag-use* for the parameters necessary for CSRF token check.

Note: `<t:transaction />` is a JSP tag library provided by the common library. For the “t:” used in (2), refer to *Creating common JSP for include*.

- Example of Output HTML

Following observations can be made upon verifying the output HTML.

- For NameSpace, the value specified in “value” attribute of the class annotation is set.

In the above example, "`transactionTokenCheckExample`" (underlined in orange) is the NameSpace.

- For TokenKey, the value that was issued at the start of the transaction is circulated and set.

In the above example, "`c0123252d531d7bafe730cd49fe0422ef`" (underlined in blue) is the TokenKey.

- Value to be set for TokenValue varies depending on request.

In the above example, "`3f610684e1cb546a13b79b9df30a7523`",

`"da770ed81dbca9a694b232e84247a13b"`,

`"bd5a2d88ec446b27c06f6d4f486d4428"` (underlined in green) are TokenValues.



When multiple usecases are to be implemented in one Controller

The implementation example of the transaction token check while carrying out processing of multiple usecases in one controller is given below.

In the example given below, (2), (3), (4) are handled as screen transitions of different usecases.

- Controller

```

@Controller
@RequestMapping("transactionTokenChecFlowkExample")
@TransactionTokenCheck("transactionTokenChecFlowkExample") // (1)
public class TransactionTokenCheckFlowExampleController {

    @RequestMapping(value = "flowOne",
                    params = "first",
                    method = RequestMethod.GET)
    public String flowOneFirst() {
        return "transactionTokenChecFlowkExample/flowOneFirstView";
    }
}

```

```
@RequestMapping(value = "flowOne",
                 params = "second",
                 method = RequestMethod.POST)
@TransactionalTokenCheck(value = "flowOne",
                        type = TransactionTokenType.BEGIN) // (2)
public String flowOneSecond() {
    return "transactionTokenChecFlowkExample/flowOneSecondView";
}

@RequestMapping(value = "flowOne",
                 params = "third",
                 method = RequestMethod.POST)
@TransactionalTokenCheck(value = "flowOne",
                        type = TransactionTokenType.IN) // (2)
public String flowOneThird() {
    return "transactionTokenChecFlowkExample/flowOneThirdView";
}

@RequestMapping(value = "flowTwo",
                 params = "first",
                 method = RequestMethod.GET)
public String flowTwoFirst() {
    return "transactionTokenChecFlowkExample/flowTwoFirstView";
}

@RequestMapping(value = "flowTwo",
                 params = "second",
                 method = RequestMethod.POST)
@TransactionalTokenCheck(value = "flowTwo",
                        type = TransactionTokenType.BEGIN) // (3)
public String flowTwoSecond() {
    return "transactionTokenChecFlowkExample/flowTwoSecondView";
}

@RequestMapping(value = "flowTwo",
                 params = "third",
                 method = RequestMethod.POST)
@TransactionalTokenCheck(value = "flowTwo",
                        type = TransactionTokenType.IN) // (3)
public String flowTwoThird() {
    return "transactionTokenChecFlowkExample/flowTwoThirdView";
}

@RequestMapping(value = "flowThree",
                 params = "first",
                 method = RequestMethod.GET)
public String flowThreeFirst() {
    return "transactionTokenChecFlowkExample/flowThreeFirstView";
}

@RequestMapping(value = "flowThree",
```

```
        params = "second",
        method = RequestMethod.POST)
@TransactionTokenCheck(value = "flowThree",
                       type = TransactionTokenType.BEGIN) // (4)
public String flowThreeSecond() {
    return "transactionTokenChecFlowkExample/flowThreeSecondView";
}

@RequestMapping(value = "flowThree",
               params = "third",
               method = RequestMethod.POST)
@TransactionTokenCheck(value = "flowThree",
                       type = TransactionTokenType.IN) // (4)
public String flowThreeThird() {
    return "transactionTokenChecFlowkExample/flowThreeThirdView";
}

}
```

Sr. No.	Description
(1)	NameSpace is specified in “value” attribute of “class” annotation. In the above example, the value same as the “value” attribute of @RequestMapping which is a recommended pattern of this guideline, is specified.
(2)	The transaction token is checked for processing of usecase with name "flowOne". In the above example, the value same as the “value” attribute of @RequestMapping which is a recommended pattern of this guideline, is specified.
(3)	The transaction token is checked for processing of usecase with name "flowTwo". In the above example, the value same as the “value” attribute of @RequestMapping which is a recommended pattern of this guideline, is specified.
(4)	The transaction token is checked for processing of usecase with name "flowThree". In the above example, the value same as the “value” attribute of @RequestMapping which is a recommended pattern of this guideline, is specified.

Note: Allocating a NameSpace for each usecase enables transaction token check for each usecase.

Typical example of using transaction token check

See the example below wherein transaction token check is applied for the usecase which carries out a simple screen transition such as “Input Screen-> Confirmation Screen-> Completion Screen”.

- Controller

```
@Controller
@RequestMapping("user")
@TransactionTokenCheck("user") // (1)
public class UserController {

    // omitted

    @RequestMapping(value = "create", params = "form")
    public String createForm(UsercreateForm form) { // (2)
        return "user/createForm";
    }

    @RequestMapping(value = "create",
                    params = "confirm",
                    method = RequestMethod.POST)
    @TransactionTokenCheck(value = "create",
                           type = TransactionTokenType.BEGIN) // (3)
    public String createConfirm(@Validated
                                UsercreateForm form, BindingResult result) {

        // omitted

        return "user/createConfirm";
    }

    @RequestMapping(value = "create", method = RequestMethod.POST)
    @TransactionTokenCheck(value = "create") // (4)
    public String create(@Validated
                        UsercreateForm form, BindingResult result) {

        // omitted

        return "redirect:/user/create?complete";
    }

    @RequestMapping(value = "create", params = "complete")
    public String createComplete() { // (5)
        return "user/createComplete";
    }

    // omitted
}
```

Sr. No.	Description
(1)	A NameSpace called "user" is set as "class" annotation. In the above example, the value same as "value" attribute of @RequestMapping annotation which is a recommended pattern, is specified.
(2)	A processing method to display input screen. It is a screen to start a usecase; however since the process only displays data and does not involve data update, it is not necessary to start a transaction. Therefore, @TransactionTokenCheck annotation is not specified in the example given above.
(3)	A processing method to perform input validation and display the Confirmation screen. A transaction is started at this point since a button to perform update process is placed on the Confirmation screen. A View (JSP) is specified for the transition destination.
(4)	A processing method to perform update. Since this method performs update, a transaction token check is performed.
(4)	A processing method to display the Completion screen. Since the method only displays a Completion screen, the transaction token check is not required. Therefore, @TransactionTokenCheck annotation is not specified in the example given above.

Warning: It is necessary to specify the View (JSP) for the transition destination of processing method wherein @TransactionTokenCheck annotation is defined. If other than View (JSP) of the redirect destination is specified as transition destination, the value of TransactionToken changes in the next process always resulting in the TransactionToken error.

Exclusion control of parallel processing while using a session

When a form object is stored in a session using @SessionAttribute annotation, and if multiple screen transitions of the same processing are performed in parallel, screen operations may interfere with each other and the values displayed on the screen and the values stored in the session may no longer match.

Transaction token check function can be used to prevent requests from non-conforming screens as the invalid requests.

The maximum limit of transaction tokens that can be stored for each NameSpace is set to 1.

- spring-mvc.xml

```
<mvc:interceptor>
    <mvc:mapping path="/**" />
    <!-- omitted -->
    <bean
        class="org.terasoluna.gfw.web.token.transaction.TransactionTokenInterceptor">
        <constructor-arg value="1"/> <!-- (1) -->
    </bean>
</mvc:interceptor>
```

Sr. No.	Description
(1)	The number of transaction tokens stored for each NameSpace is set to “1”.

Note: When form objects etc. are stored in session using @SessionAttribute annotation, the requests from the screen displaying old data can be prevented as invalid requests by setting number of transaction token stored for each NameSpace to “1”.

5.12.3 How to extend

How to manage the lifecycle of transaction tokens using a program

It is possible to receive org.terasoluna.gfw.web.token.transaction.TransactionTokenContext as an argument for processing method of Controller and manage the lifecycle of transaction tokens programmatically by adding the settings give below.

- spring-mvc.xml

```
<mvc:annotation-driven>
  <mvc:argument-resolvers>
    <!-- (1) -->
    <bean
      class="org.terasoluna.gfw.web.token.transaction.TransactionTokenContextHandlerMethodArgumentResolver"
    </mvc:argument-resolvers>
</mvc:annotation-driven>
```

Sr. No.	Description
(1)	<p>In <code><mvc:argument-resolvers></code> element, set the class (<code>TransactionTokenContextHandlerMethodArgumentResolver</code>) which passes the object (<code>TransactionTokenContext</code>) managing the lifecycle of transaction tokens programmatically, as an argument for methods of Controller.</p> <p>When it is not necessary to manage the lifecycle of transaction tokens using a program, this setting is not required.</p>

Note: This setting is not required as the transaction tokens that can no longer be used are automatically discarded when the tokens that can be stored in a NameSpace exceeds the maximum limit.

How to change the maximum limit of transaction tokens

The maximum limit of transaction tokens that can be stored on 1 NameSpace can be changed by performing settings given below.

- `spring-mvc.xml`

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**" />
    <mvc:exclude-mapping path="/resources/**" />
    <mvc:exclude-mapping path="/**/*.html" />
    <bean
      class="org.terasoluna.gfw.web.token.transaction.TransactionTokenInterceptor" />
      <constructor-arg value="5"/> <!-- (1) -->
    </mvc:interceptor>
  </mvc:interceptors>
```

Sr. No.	Description
(1)	<p>The maximum limit of transaction tokens that can be stored in a NameSpace is specified as a value of constructor of <code>TransactionTokenInterceptor</code>.</p> <p>The default value (value that is set when the default constructor is used) is 10.</p> <p>In the above example, the default value (10) is changed to 5.</p>

5.12.4 Appendix

Global Tokens

If the “value” attribute of `@TransactionTokenCheck` annotation is not specified, it is handled as global transaction token.

“globalToken” (fixed value) is used for the NameSpace of global transaction tokens.

Note: When only a single screen transition is to be allowed as an overall application, it can be implemented by setting the maximum limit of transaction tokens that can be stored for each NameSpace to 1 and using the global token.

The settings and implementation example when only a single screen transition is to be allowed as an overall application are shown below.

Changing the maximum limit of transaction tokens that can be stored for each NameSpace

The maximum limit of transaction tokens that can be stored for each NameSpace is set to 1.

- `spring-mvc.xml`

```
<mvc:interceptor>
    <mvc:mapping path="/**" />
    <!-- omitted -->
    <bean
        class="org.terasoluna.gfw.web.token.transaction.TransactionTokenInterceptor">
        <constructor-arg value="1"/> <!-- (1) -->
    </bean>
</mvc:interceptor>
```

Sr. No.	Description
(1)	The number of tokens stored for each NameSpace is set to “1”.

Implementation of Controller

The value is not specified in “value” attribute of @TransactionTokenCheck annotation, in order to make it the NameSpace for global tokens.

- Controller

```
@Controller
@RequestMapping("globalTokenCheckExample")
public class GlobalTokenCheckExampleController { // (1)

    @RequestMapping(params = "first", method = RequestMethod.GET)
    public String first() {
        return "globalTokenCheckExample/firstView";
    }

    @RequestMapping(params = "second", method = RequestMethod.POST)
    @TransactionTokenCheck(type = TransactionTokenType.BEGIN) // (2)
    public String second() {
        return "globalTokenCheckExample/secondView";
    }

    @RequestMapping(params = "third", method = RequestMethod.POST)
    @TransactionTokenCheck // (2)
    public String third() {
        return "globalTokenCheckExample/thirdView";
    }

    @RequestMapping(params = "fourth", method = RequestMethod.POST)
    @TransactionTokenCheck // (2)
    public String fourth() {
        return "globalTokenCheckExample/fourthView";
    }

    @RequestMapping(params = "fifth", method = RequestMethod.POST)
    public String fifth() {
        return "globalTokenCheckExample/fifthView";
    }

}
```

Sr. No.	Description
(1)	@TransactionTokenCheck annotation is not specified as the class annotation.
(2)	The “value” attribute of @TransactionTokenCheck annotation to be specified as “method” annotation is not specified.

- Example of Output HTML

JSP used is same as the one created in [How to use transaction token check in View \(JSP\)](#) .

Other details are same except change in action from "transactionTokenCheckExample" to "globalTokenCheckExample" .



Following observations can be made upon verifying the output HTML.

- For NameSpace, a fixed value called "globalToken" is set.
- For TokenKey, the value that was issued while starting the transaction is circulated and set.

In the above example, "9d937be4adc2f5dd2032292d153f1133" (underlined in blue) is the TokenKey.

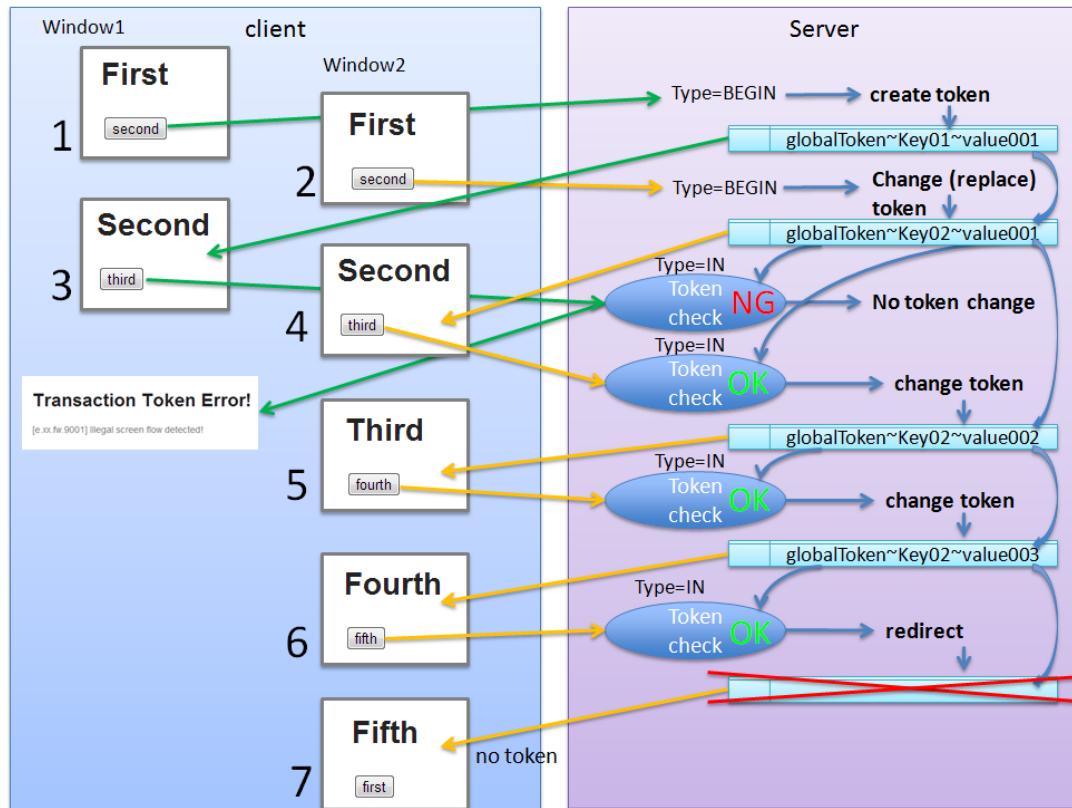
- Value to be set for TokenValue varies depending on request.

In the above example, "9204d7705ce7a17f16ca6cec24cf88b",

"69c809fefcad541dbd00bd1983af2148",

"6b83f33b365f1270ee1c1b263f046719" (underlined in green) are TokenValues.

The behavior when the maximum limit of transaction tokens for each NameSpace is set to 1 and global token is used, is explained below.



Sr. No.	Description
(1)	In window 1 process, TransactionTokenType.BEGIN is performed and global token is generated.
(2)	In window 2 process, the token is updated by TransactionTokenType.BEGIN. Although internally the data is reshuffled rather than getting updated, it gives the impression that the token has been updated since one transaction token can be stored on the server.
(3)	Token value is checked in TransactionTokenType.IN of window 1 process. Transaction token generated in process 1 is submitted as request parameter; however since the specified token does not exist on the server, a transaction token error occurs.
(4)	Token value is checked in TransactionTokenType.IN of window 2 process. The transaction token generated in process 2 is submitted as request parameter and it is checked whether the value matches with the token value stored on the server. If the value matches, the process is continued.
(5)	Similar to (4) .
(6)	Similar to (4) .
(7)	When a screen is to be displayed using redirect, hidden tag for transaction token does not exist.

Note: The transaction token existing on the server is automatically deleted when a new global token is generated.

Quick Reference

The table below illustrates an example of a business application for managing Account and Customer. It shows the required settings for transaction tokens and business limitations.

The usecases assumed in this business application are Create, Update, Delete relating to Account and Customer. By using the table below as reference, the maximum limit of tokens and NameSpace settings should be performed as per the system requirements.

Number	Number of tokens stored for each NameSpace	NameSpace value specified in class	NameSpace value specified in method	Example of generated token	Business limitations
(1)	10 (Default)	account	Not specified	account~key~value	Number of concurrent executions of all Account usecases (create/update/delete) is restricted to 10.
(2)	10 (Default)	account	create	account/create~key~value	Number of concurrent executions of “create” operation of Account usecase is restricted to 10.
(3)	10 (Default)	account	update	account/update~key~value	Number of concurrent executions of “update” operation of Account usecase is restricted to 10.
(4)	10 (Default)	account	delete	account/delete~key~value	Number of concurrent executions of “delete” operation of Account usecase is restricted to 10. (By specifying (2), (3) and (4), the number of concurrent executions of all Account usecases should be 30. Since
5.12. Double Submit Protection					911 this setting value is too high for most applications, a value smaller than default value 10 can also be

5.13 Internationalization

5.13.1 Overview

Internationalization is a process wherein the display of labels and messages in an application is not fixed to a specific language. It supports multiple languages. The language switching can be achieved by specifying a unit called “Locale” expressing language, country and region.

This section explains how to internationalize messages to display on the screen.

In order to internationalization, following requirements should be meted.

- Text elements on the screen(code name, messages, labels of GUI components etc.) should be retrieve from external definitions such as properties file. (should not be hard-coding in the source code)
- The mechanism to specify the locale of the clients should be provided.

Methods to specify the locale are as follows:

- Using standard request header (specify the locale by language settings of browsers)
- Saving the locale into the Cookie using request parameter
- Saving the locale into the Session using request parameter

The image of switching locale is as follows:



Note: For internationalization of Codelist, refer to [Codelist](#).

Tip: The most commonly known abbreviation of internationalization is i18n. Internationalization is abbreviated as i18n because the number of letters between the first “i” and the last “n” is 18 i.e. “nternationalizatio”.

5.13.2 How to use

Configuration to define messages

To internationalize the messages on the screen, use the one of following MessageSource implementations for managing messages.

- org.springframework.context.support.ResourceBundleMessageSource
- org.springframework.context.support.ReloadableResourceBundleMessageSource

The information here explains an example of using the ResourceBundleMessageSource.

applicationContext.xml

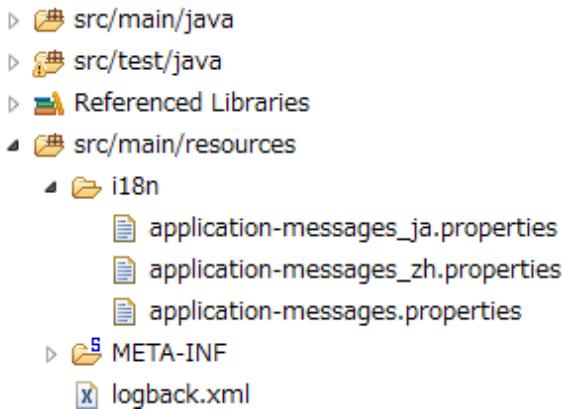
```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>i18n/application-messages</value> <!-- (1) -->
      </list>
    </property>
</bean>
```

Sr. No.	Description
(1)	<p>Specify i18n/application-messages as base name of properties file.</p> <p>It is recommended to store message properties file under i18n directory to support internationalization.</p> <p>For MessageSource details and definition methods, refer to Message Management.</p>

Example of storing properties files

Properties file should be created in accordance with the following rules.

- File name should be defined in application-messages_XX.properties format. (Specify locale in XX portion)
- The messages defined in application-messages.properties should be created in default language.



- Make sure you create `application-messages.properties`. If it does not exist, messages cannot be fetched from `MessageSource` and `JspTagException` occurs while setting the messages in JSP.

When creating property files as above, it is determined which to use the file as follows:

- When the locale resolved by `LocaleResolver` is `zh`, `application-messages_zh.properties` is used.
- when the locale resolved by `LocaleResolver` is `ja`, `application-messages_ja.properties` is used.
- When properties file does not exist corresponding to the locale resolved by `LocaleResolver`, `application-messages.properties` is used as default. (“_XX” portion does not exist in file name)

Note: The locate to use is determined in the following order until a properties file is found corresponding to the locale.

1. Locale sent from clients
2. Locale specified by JVM on which application server runs (it may not be set in some cases)
3. Locale specified by OS on which application server runs

It is frequently misunderstood that default properties file is used when properties file does not exist corresponding to the locale sent from clients . In this case, then it is checked whether the file is available corresponding to the locale specified by the application server. If not found, finally the default properties file is used.

Tip: For description of message properties file, refer to [Message Management](#).

Changing locale as per browser settings

Settings of AcceptHeaderLocaleResolver

If switch the locale using browser settings, use the `AcceptHeaderLocaleResolver`.

spring-mvc.xml

```
<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver" /> <!-- (1) -->
```

Sr. No.	Description
(1)	Specify <code>org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver</code> in <code>id</code> attribute “ <code>localeResolver</code> ” of bean tag. If this <code>LocaleResolver</code> is used, HTTP header “ <code>accept-language</code> ” is added for each request and locale gets specified.

Note: When `LocaleResolver` is not set, `org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver` is used by default; hence `LocaleResolver` need not be set.

Definition of messages

See the example of definition of messages below.

application-messages.properties

```
title.admin.top = Admin Top
```

application-messages_ja.properties

```
title.admin.top = 管理画面 Top
```

Implementation of JSP

See the example of implementation of messages below.

include.jsp(Common jsp file to be included)

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

Sr. No.	Description
(1)	<p>When message is to be output in JSP, it is output using Spring tag library; hence custom tag needs to be defined.</p> <pre><%@taglib uri="http://www.springframework.org/tags" prefix="spring"%> should be defined.</pre>

Note: For details on common jsp files to be included, refer to [Creating common JSP for include](#).

JSP file for screen display

```
<spring:message code="title.admin.top" /> <!-- (2) -->
```

Sr. No.	Description
(2)	<p>Output the message using <code><spring:message></code> which is a Spring tag library of JSP.</p> <p>In code attribute, set the key specified in properties.</p> <p>In this example, if locale is ja, “管理画面 Top” is output and for other locales, “Admin Top” is output.</p>

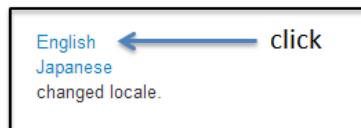
Changing locale depending on screen operations dynamically

The method of dynamic changing the locale depending on screen operations etc. is effective in case of selecting a specific language irrespective of user terminal (browser) settings.

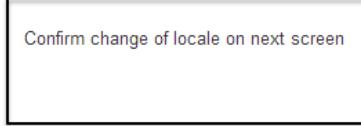
Following is an example of changing locale depending on screen operations.

Change locale on screen

first



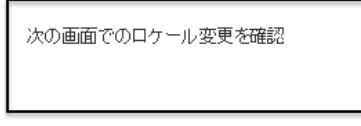
second



first



second



To use the language selected by a user, choose `org.springframework.web.servlet.i18n.LocaleChangeInterceptor`.

`LocaleChangeInterceptor` is an interceptor to save the locale value specified by the request parameter using `org.springframework.web.servlet.LocaleResolver`.

Select the implementation class of `LocaleResolver` from the following table.

Table.5.23 Types of LocaleResolver

No	Implementation class	How to save locale
1.	org.springframework.web.servlet.i18n.SessionLocaleResolver	Save in server(using HttpSession)
2.	org.springframework.web.servlet.i18n.CookieLocaleResolver	Save in client(using Cookie)

Note: When `org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver` is used in `LocaleResolver`, locale cannot be changed dynamically using `org.springframework.web.servlet.i18n.LocaleChangeInterceptor`.

How to define LocaleChangeInterceptor

If switching the locale using the request parameter, use the `LocaleChangeInterceptor`.

spring-mvc.xml

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**" />
    <mvc:exclude-mapping path="/resources/**" />
    <mvc:exclude-mapping path="/**/*.html" />
    <bean
      class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"> <!-- (1) -->
    </bean>
    <!-- omitted -->
  </mvc:interceptor>
</mvc:interceptors>
```

項番	説明
(1)	Define <code>org.springframework.web.servlet.i18n.LocaleChangeInterceptor</code> in interceptor of Spring MVC.

Note: How to change the name of request parameter to specify locale

```
<bean
    class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="lang"/> <!-- (2) -->
</bean>
```

Sr. No.	Description
(2)	In paramName property, specify the name of request parameter. In this example, it is “request URL?lang=xx”. When paramName property is omitted, “locale” gets set. With “request URL?locale=xx”, it becomes <i>enabled</i> .

How to define SessionLocaleResolver

If saving the locale in the server side, use the SessionLocaleResolver.

spring-mvc.xml

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver"> <!-- (2) -->
    <property name="defaultLocale" value="en"/>
</bean>
```

Sr. No.	Description
(1)	<p>Define id attribute of bean tag in “localeResolver” and specify the class wherein <code>org.springframework.web.servlet.LocaleResolver</code> is implemented.</p> <p>In this example,</p> <p><code>org.springframework.web.servlet.i18n.SessionLocaleResolver</code> that stores locale in session is specified.</p> <p><code>id</code> attribute of bean tag should be set as “<code>localeResolver</code>”.</p> <p>By performing these settings, <code>SessionLocaleResolver</code> will be used at the <code>LocaleChangeInterceptor</code>.</p>
(2)	<p>When locale is not specified in request parameter, locale specified in <code>defaultLocale</code> property is enabled. In this case, the value fetched in <code>HttpServletRequest#getLocale</code> is considered.</p>

How to define CookieLocaleResolver

If saving the locale in the client side, use the `CookieLocaleResolver`.

spring-mvc.xml

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver"> <!-- (1) -->
    <property name="defaultLocale" value="en"/> <!-- (2) -->
    <property name="cookieName" value="localeCookie"/> <!-- (3) -->
</bean>
```

Sr. No.	Description
(1)	In id attribute “localeResolver” of bean tag, specify <code>org.springframework.web.servlet.i18n.CookieLocaleResolver</code> . id attribute of bean tag should be set as “localeResolver”. By performing these settings, CookieLocaleResolver will be used at the LocaleChangeInterceptor.
(2)	When locale is not specified, locale specified in <code>defaultLocale</code> property is enabled. In this case, the value fetched in <code>HttpServletRequest#getLocale</code> is considered.
(3)	The value specified in <code>cookieName</code> property is used as cookie name. If not specified, the value of <code>org.springframework.web.servlet.i18n.CookieLocaleResolver.LOCALE</code> is used as default. It is recommended to change not to tell the user explicitly Spring Framework is used.

Messages settings

See the example below for messages settings.

application-messages.properties

```
i.xx.yy.0001 = changed locale
i.xx.yy.0002 = Confirm change of locale at next screen
```

application-messages_ja.properties

```
i.xx.yy.0001 = Locale を変更しました。
i.xx.yy.0002 = 次の画面での Locale 変更を確認
```

Implementation of JSP

See the example of implementation of JSP below.

JSP file for screen display

```
<a href='${pageContext.request.contextPath}?locale=en'>English</a> <!-- (1) -->
<a href='${pageContext.request.contextPath}?locale=ja'>Japanese</a>
<spring:message code="i.xx.yy.0001" />
```

Sr. No.	Description
(1)	<p>Submit the request parameter to switch the locale.</p> <p>Request parameter name is specified in <code>paramName</code> property of <code>LocaleChangeInterceptor</code>. (In the example above, the default parameter name is used)</p> <p>In the above example, it is changed to English locale in English link and to Japanese locale in Japanese link.</p> <p>Hereafter, the selected locale is enabled.</p> <p>As “en” properties file does not exist, English locale is read from properties file by default.</p>

Tip:

- Spring tag library should be defined in common jsp files to be included.
 - For details on common jsp files to be included, refer to [*Creating common JSP for include*](#).
-

5.14 Codelist

5.14.1 Overview

A codelist is a pair comprising of “Code values (Value) and their display names (Label)”.

It is used for mapping code values with the labels to be displayed on screen such as selectbox.

Common library provides the following functionalities:

- Functionality to read and cache the codelist defined in xml file or DB at the time of launching the application.
- Functionality to refer to the codelist from JSP or Java class
- Functionality to perform input validation using codelist

Moreover, it also supports

- internationalization of codelist
- reloading of cached codelist

Note: As per standard specifications, only the codelist defined in DB is reloadable.

Following four types of codelists are implemented in common library.

Table.5.24 Types of codelist

Types	Contents	Reloadable
<code>org.terasoluna.gfw.common.codelist.SimpleCodelist</code>	Use the hardcoded contents from xml file.	NO
<code>org.terasoluna.gfw.common.codelist.NumberRangeCodelist</code>	Use when creating number range list.	NO
<code>org.terasoluna.gfw.common.codelist.JdbcCodelist</code>	Use the codelist by fetching the code from DB using SQL.	YES
<code>org.terasoluna.gfw.common.codelist.i18n.CachedCodelist</code>	Use the codelist corresponding to java.util.Locale.	NO

Common library provides `org.terasoluna.gfw.common.codelist.CodeList` for the interface of above codelists.

Codelist class diagram provided in common library is as follows:

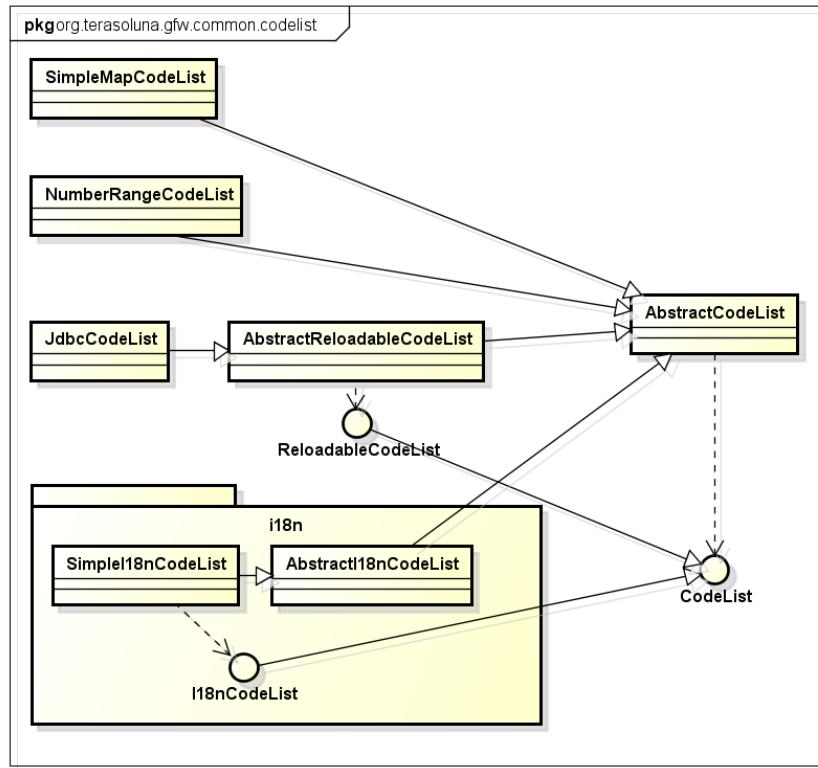


Figure.5.44 Picture - Image of codelist class diagram

5.14.2 How to use

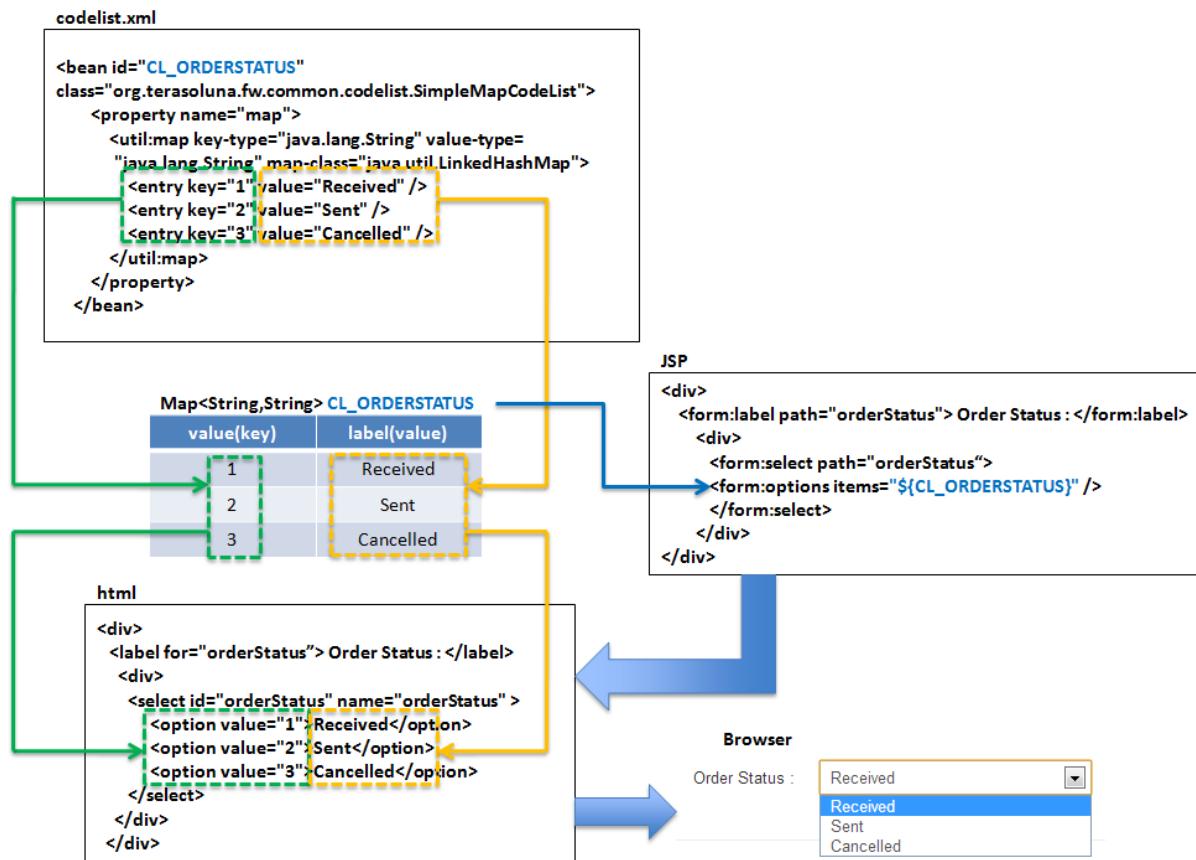
This section describes settings for various codelists and their implementation methods.

- *Using SimpleMapCodeList*
- *Using NumberRangeCodeList*
- *Using JdbcCodeList*
- *How to use SimpleI18nCodeList*
- *Input validation of code value using codelist*

Using SimpleMapCodeList

`org.terasoluna.gfw.common.codelist.SimpleMapCodeList` reads the code values defined in xml file at the time of launching the application and uses them as is.

SimpleMapCodeList image



Example of codelist settings

Definition of Bean definition file(xxx-codelist.xml)

It is recommended to create a bean definition file for codelist.

```
<bean id="CL_ORDERSTATUS" class="org.terasoluna.gfw.common.codelist.SimpleMapCodeList"> <!-- (1)
  <property name="map">
    <util:map>
      <entry key="1" value="Received" /> <!-- (2) -->
      <entry key="2" value="Sent" />
      <entry key="3" value="Cancelled" />
    </util:map>
  </property>
</bean>
```

```
</util:map>
</property>
</bean>
```

Sr. No.	Description
(1)	Define a bean of SimpleMapCodeList class. beanID should have the name matching with the ID pattern of <code>org.terasoluna.gfw.web.codelist.CodeListInterceptor</code> described later.
(2)	Define Key, Value pairs of Map. When map-class attribute is omitted, it is registered in <code>java.util.LinkedHashMap</code> ; hence in the above example, “Name and value” are stored in Map in the order of registration.

Definition of Bean definition file(xxx-domain.xml)

Once the bean definition file for codelist is created, it should be imported to already existing bean definition file.

```
<import resource="classpath:META-INF/spring/projectName-codelist.xml" /> <!-- (3) -->
<context:component-scan base-package="com.example.domain" />

<!-- omitted -->
```

Sr. No.	Description
(3)	Import bean definition file for codelist. Resource information of import is necessary during component-scan; hence import should be set above <code><context:component-scan base-package="com.example.domain" /></code> .

Using codelist in JSP

By using the interceptor of common library, codelist can be set automatically in request scope and can be easily referred from JSP.

Definition of Bean definition file(spring-mvc.xml)

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**" /> <!-- (1) -->
    <bean
      class="org.terasoluna.gfw.web.codelist.CodeListInterceptor"> <!-- (2) -->
      <property name="codeListIdPattern" value="CL_.+" /> <!-- (3) -->
    </bean>
  </mvc:interceptor>

  <!-- omitted -->

</mvc:interceptors>
```

Sr. No.	Description
(1)	Set the applicable path.
(2)	Define a bean of CodeListInterceptor class.
(3)	<p>Set the beanID pattern of codelist which is automatically set in the request scope.</p> <p>In pattern, regular expression used in <code>java.util.regex.Pattern</code> should be set.</p> <p>In the above example, only the data in which id is defined in “CL_XXX” format is targeted. In that case, bean definition wherein id does not start with “CL_” should not be imported.</p> <p>beanID defined in “CL_” can be used in JSP since it is set in the request scope.</p> <p><code>codeListIdPattern</code> property can be omitted.</p> <p>If omitting <code>codeListIdPattern</code> property, all of <code>CodeLists</code> (all beans which implements <code>org.terasoluna.gfw.common.codelist.CodeList</code>) are available in JSP.</p>

Example of implementing the codelist in jsp

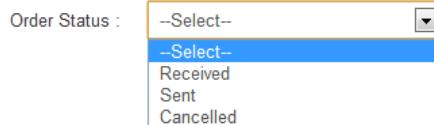
```
<form:select path="orderStatus">
  <form:option value="" label="--Select--" /> <!-- (4) -->
  <form:options items="${CL_ORDERSTATUS}" /> <!-- (5) -->
</form:select>
```

Sr. No.	Description
(4)	When setting dummy value at the top of the selectbox, null characters should be specified in the value.
(5)	Specify the beanID for which codelist is defined.

Output HTML

```
<select id="orderStatus" name="orderStatus">
  <option value="">--Select--</option>
  <option value="1">Received</option>
  <option value="2">Sent</option>
  <option value="3">Cancelled</option>
</select>
```

Output screen



Using codelist in Java class

When using the codelist in Java class, inject the codelist by setting javax.inject.Inject annotation and javax.inject.Named annotation. Specify the codelist name in @Named annotation.

```
import javax.inject.Named;

import org.terasoluna.fw.common.codelist.CodeList;

public class OrderServiceImpl implements OrderService {

    @Inject
    @Named("CL_ORDERSTATUS")
```

```
CodeList orderStatusCodeList; // (1)

public boolean existOrderStatus(String target) {
    return orderStatusCodeList.asMap().containsKey(target); // (2)
}
}
```

Sr. No.	Description
(1)	Inject the codelist with beanID “CL_ORDERSTATUS”.
(2)	Fetch the codelist in <code>java.util.Map</code> format using <code>CodeList#asMap</code> method.

Using NumberRangeCodeList

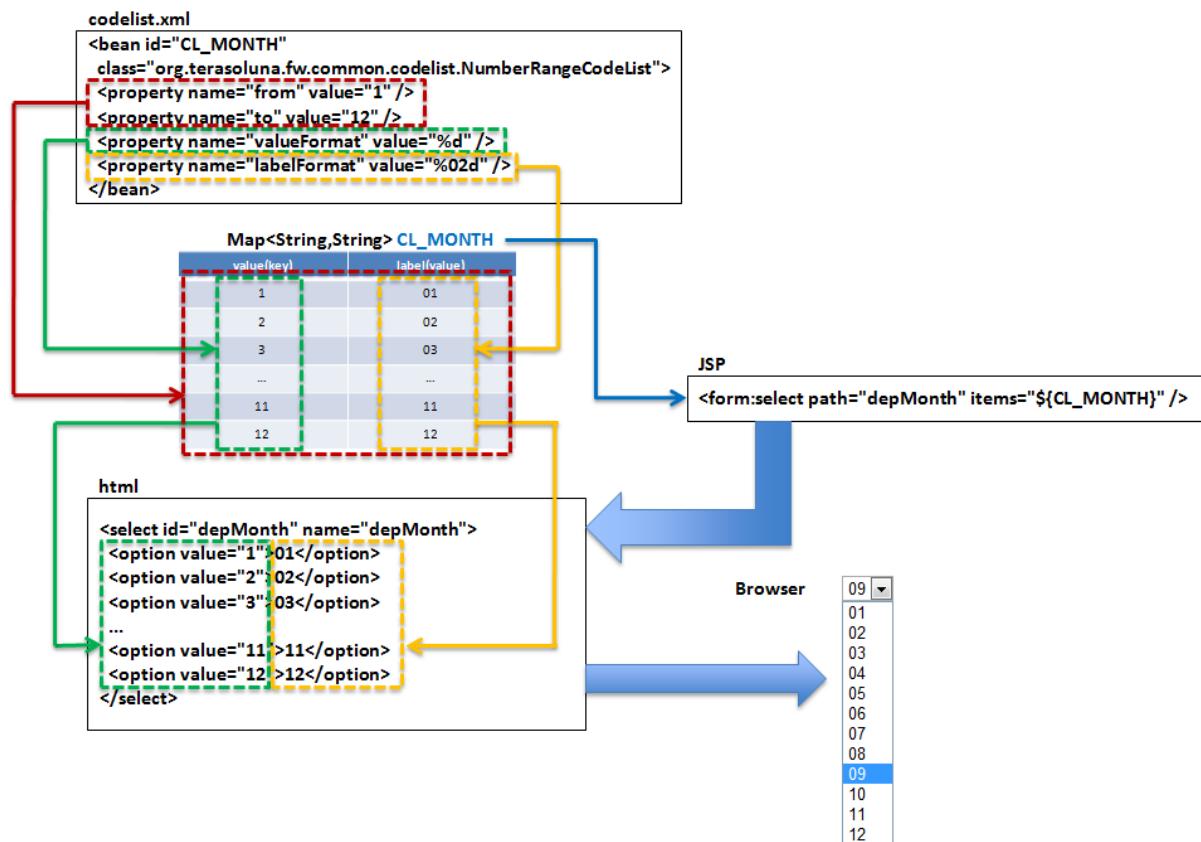
`org.terasoluna.gfw.common.codelist.NumberRangeCodeList` is a codelist that creates the list of numeric values of specified range at the time of launching the application. It is assumed that this codelist will mainly be used in the selectboxes having only numbers i.e. selectbox for month, date etc.

Image of NumberRangeCodeList

Tip: NumberRangeCodeList supports only Arabic numbers and does not support Chinese and Roman numbers. Chinese and Roman numbers can be supported by using JdbcCodeList and SimpleMapCodeList.

NumberRangeCodeList has the following features:

1. In order to set From value < To value, the values increased in accordance with the interval are set in From-To range in ascending order.
2. In order to set To value < From value, the values decreased in accordance with the interval are set in To-From range in descending order.
3. Increment (decrement) can be changed by setting intervals.



The information here describes how to configure the ascending NumberRangeCodeList. For how to create the descending NumberRangeCodeList or change interval, refer to “[Variations of NumberRangeCodeList](#)”. |

Example of codelist settings

Example of setting From value < To value is shown below.

Definition of Bean definition file(xxx-codelist.xml)

```
<bean id="CL_MONTH"
    class="org.terasoluna.gfw.common.codelist.NumberRangeCodeList"> <!-- (1) -->
    <property name="from" value="1" /> <!-- (2) -->
    <property name="to" value="12" /> <!-- (3) -->
    <property name="valueFormat" value="%d" /> <!-- (4) -->
    <property name="labelFormat" value="%02d" /> <!-- (5) -->
    <property name="interval" value="1" /> <!-- (6) -->
</bean>
```

Sr. No.	Description
(1)	Define a bean of NumberRangeCodeList.
(2)	Specify the range start value. When omitted, “0” is set as range start value.
(3)	Specify the range end value. It cannot be blank.
(4)	Specify the format of the code value. Format used should be <code>java.lang.String.format</code> . When omitted, “%s” is set.
(5)	Specify the format of the code name. Format used should be <code>java.lang.String.format</code> . When omitted, “%s” is set.
(6)	Set the increment value. When omitted, “1” is set.

Using codelist in JSP

For details on settings shown below, refer to *Using codelist in JSP* described earlier.

Example of jsp implementation

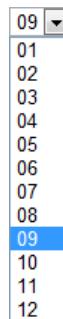
```
<form:select path="depMonth" items="${CL_MONTH}" />
```

Output HTML

```
<select id="depMonth" name="depMonth">
  <option value="1">01</option>
  <option value="2">02</option>
  <option value="3">03</option>
  <option value="4">04</option>
  <option value="5">05</option>
```

```
<option value="6">06</option>
<option value="7">07</option>
<option value="8">08</option>
<option value="9">09</option>
<option value="10">10</option>
<option value="11">11</option>
<option value="12">12</option>
</select>
```

Output screen



Using codelist in Java class

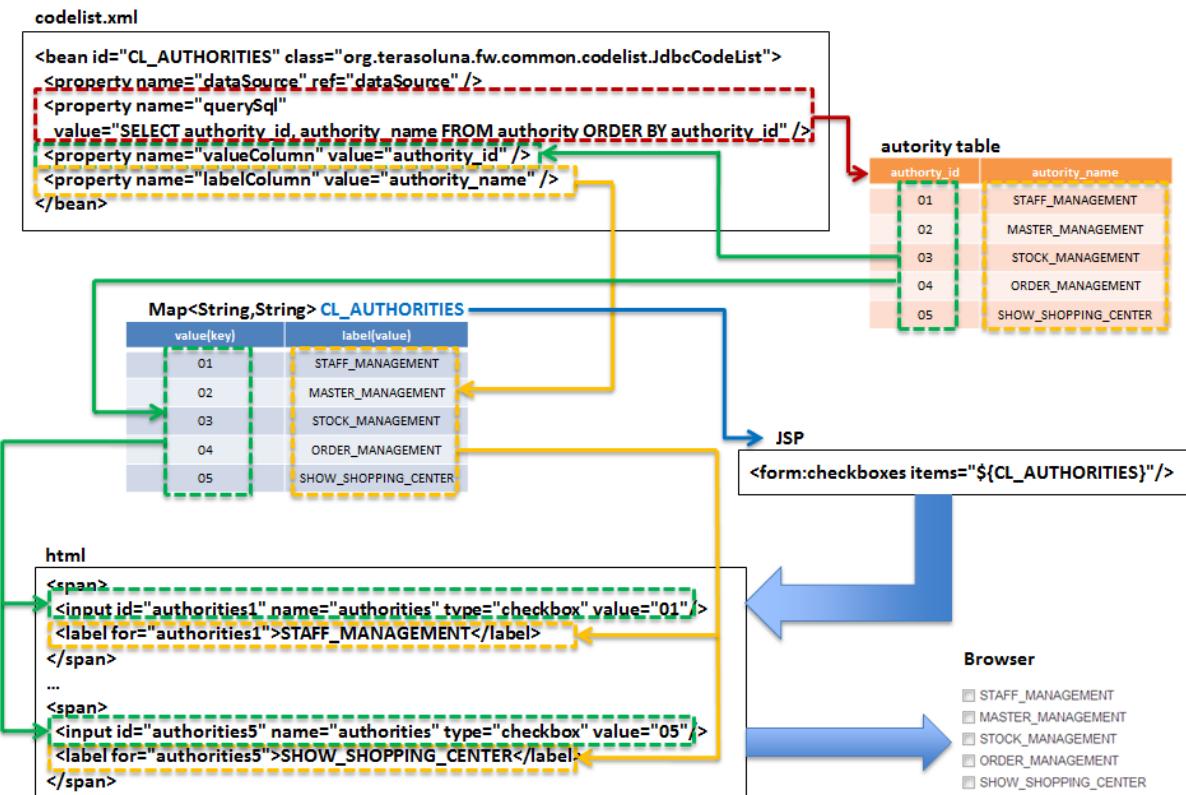
For details on settings shown below, refer to *Using codelist in Java class* described earlier.

Using JdbcCodeList

`org.terasoluna.gfw.common.codelist.JdbcCodeList` is a class for creating codelist by fetching values from DB at the time of launching the application. This list is cached.

The fetched values can be changed dynamically by reloading. For details, refer to *When reloading the codelist*.

JdbcCodeList image



Example of codelist settings

Definition of Table(authority)

authority_id	authority_name
01	STAFF_MANAGEMENT
02	MASTER_MANAGEMENT
03	STOCK_MANAGEMENT
04	ORDER_MANAGEMENT
05	SHOW_SHOPPING_CENTER

Definition of Bean definition file(xxx-codelist.xml)

```
<bean id="CL_AUTHORITIES" class="org.terasoluna.gfw.common.codelist.JdbcCodeList"> <!-- (1) -->
    <property name="dataSource" ref="dataSource" />
    <property name="querySql"
        value="SELECT authority_id, authority_name FROM authority ORDER BY authority_id" /> <!-- (2) -->
    <property name="valueColumn" value="authority_id" /> <!-- (3) -->
    <property name="labelColumn" value="authority_name" /> <!-- (4) -->
</bean>
```

Sr. No.	Description
(1)	Define a bean of JdbcCodeList class.
(2)	Write an SQL for fetching the codelist in querySql property. At that time, make sure to specify ORDER BY clause to define the order. If ORDER BY is not specified, the order gets changed every time when records are fetched using SQL.
(3)	Set the value corresponding to the Key of Map in valueColumn property. In this example, authority_id is set.
(4)	Set the value corresponding to the Value of Map in labelColumn property. In this example, authority_name is set.

Using codelist in JSP

For details on settings shown below, refer to *Using codelist in JSP* described earlier.

Example of jsp implementation

```
<form:checkboxes items="${CL_AUTHORITIES}" />
```

Output HTML

```
<span>
  <input id="authorities1" name="authorities" type="checkbox" value="01"/>
  <label for="authorities1">STAFF_MANAGEMENT</label>
</span>
<span>
  <input id="authorities2" name="authorities" type="checkbox" value="02"/>
  <label for="authorities2">MASTER_MANAGEMENT</label>
</span>
<span>
  <input id="authorities3" name="authorities" type="checkbox" value="03"/>
  <label for="authorities3">STOCK_MANAGEMENT</label>
</span>
<span>
  <input id="authorities4" name="authorities" type="checkbox" value="04"/>
  <label for="authorities4">ORDER_MANAGEMENT</label>
</span>
<span>
  <input id="authorities5" name="authorities" type="checkbox" value="05"/>
  <label for="authorities5">SHOW_SHOPPING_CENTER</label>
</span>
```

Output screen

Authorities STAFF_MANAGEMENT
 MASTER_MANAGEMENT
 STOCK_MANAGEMENT
 ORDER_MANAGEMENT
 SHOW_SHOPPING_CENTER

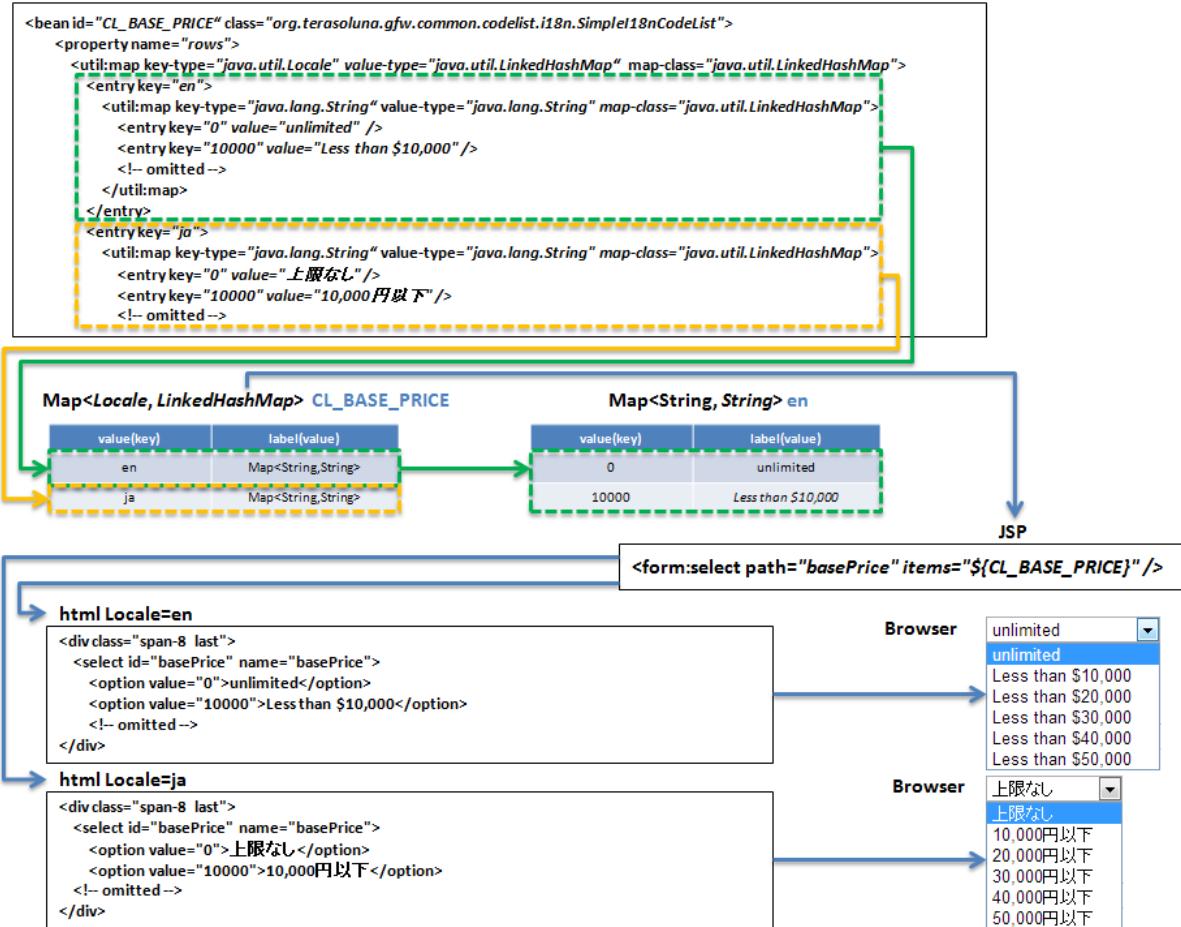
Using codelist in Java class

For details on settings shown below, refer to [Using codelist in Java class](#) described earlier.

How to use SimpleI18nCodeList

`org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList` is a codelist supporting internationalization. By setting the codelist for each locale, the codelist corresponding to locale can be returned.

SimpleI18nCodeList image



Example of setting a codelist

It is easier to understand if you consider `SimpleI18nCodeList` as two dimensional table wherein row is Locale, column contains code values and cell details are labels.

The table would be as follows in case of a selectbox for selecting charges.

row=Locale	column=Code	10000	20000	30000	40000	50000
en	unlimited	Less than \$10,000	Less than \$20,000	Less than \$30,000	Less than \$40,000	Less than \$50,000
ja	上限なし	10,000 円以下	20,000 円以下	30,000 円以下	40,000 円以下	50,000 円以下

For creating a codelist table that supports internationalization, `SimpleI18nCodeList` has been set in following 3 ways.

- Set `CodeList` for each locale by rows.
- Set `java.util.Map(key = code value, value = label)` for each locale by rows.
- Set `java.util.Map(key = locale, value = label)` for each code value by columns.

It is recommended that you set the codelist using “Set `CodeList` for each locale by rows.” method.

The way of setting the `CodeList` for each locale by rows considering the above example of selectbox for selecting charges, is mentioned below. For other setting methods, refer to [Setting SimpleI18nCodeList](#).

Definition of Bean definition file (xxx-codelist.xml)

```
<bean id="CL_I18N_PRICE"
      class="org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList">
  <property name="rowsByCodeList"> <!-- (1) -->
    <util:map>
      <entry key="en" value-ref="CL_PRICE_EN" />
      <entry key="ja" value-ref="CL_PRICE_JA" />
    </util:map>
  </property>
</bean>
```

Sr. No.	Description
(1)	<p>Set Map wherein key is <code>java.lang.Locale</code>, in <code>rowsByCodeList</code> properties.</p> <p>In Map, specify locale in key and a reference link to codelist class corresponding to locale in <code>value-ref</code>.</p> <p>For Map values, refer to codelist class corresponding to each locale.</p>

Definition of Bean definition file(xxx-codelist.xml) when creating SimpleMapCodeList for each locale

```
<bean id="CL_I18N_PRICE"
      class="org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList">
  <property name="rowsByCodeList">
    <util:map>
      <entry key="en" value-ref="CL_PRICE_EN" />
      <entry key="ja" value-ref="CL_PRICE_JA" />
    </util:map>
  </property>
</bean>

<bean id="CL_PRICE_EN" class="org.terasoluna.gfw.common.codelist.SimpleMapCodeList"> <!-- (2) -->
  <property name="map">
    <util:map>
      <entry key="0" value="unlimited" />
      <entry key="10000" value="Less than $10,000" />
      <entry key="20000" value="Less than $20,000" />
      <entry key="30000" value="Less than $30,000" />
      <entry key="40000" value="Less than $40,000" />
      <entry key="50000" value="Less than $50,000" />
    </util:map>
  </property>
</bean>

<bean id="CL_PRICE_JA" class="org.terasoluna.gfw.common.codelist.SimpleMapCodeList"> <!-- (3) -->
  <property name="map">
    <util:map>
      <entry key="0" value="上限なし" />
      <entry key="10000" value="10,000 円以下" />
      <entry key="20000" value="20,000 円以下" />
      <entry key="30000" value="30,000 円以下" />
      <entry key="40000" value="40,000 円以下" />
      <entry key="50000" value="50,000 円以下" />
    </util:map>
  </property>
</bean>
```

Sr. No.	Description
(2)	For bean definition CL_PRICE_EN where locale is “en”, codelist class is set in SimpleMapCodeList.
(3)	For bean definition CL_PRICE_JA where locale is “ja”, codelist class is set in SimpleMapCodeList.

Definition of Bean definition file(xxx-codelist.xml) when creating JdbcCodeList for each locale

```

<bean id="CL_I18N_PRICE"
      class="org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList">
    <property name="rowsByCodeList">
      <util:map>
        <entry key="en" value-ref="CL_PRICE_EN" />
        <entry key="ja" value-ref="CL_PRICE_JA" />
      </util:map>
    </property>
  </bean>

<bean id="CL_PRICE_EN" class="org.terasoluna.gfw.common.codelist.JdbcCodeList"<!-- (4) --&gt;
  &lt;property name="dataSource" ref="dataSource" /&gt;
  &lt;property name="querySql"
            value="SELECT code, label FROM price WHERE locale = 'en' ORDER BY code" /&gt;
  &lt;property name="valueColumn" value="code" /&gt;
  &lt;property name="labelColumn" value="label" /&gt;
&lt;/bean&gt;

&lt;bean id="CL_PRICE_JA" class="org.terasoluna.gfw.common.codelist.JdbcCodeList"<!-- (5) --&gt;
  &lt;property name="dataSource" ref="dataSource" /&gt;
  &lt;property name="querySql"
            value="SELECT code, label FROM price WHERE locale = 'ja' ORDER BY code" /&gt;
  &lt;property name="valueColumn" value="code" /&gt;
  &lt;property name="labelColumn" value="label" /&gt;
&lt;/bean&gt;
</pre>

```

Sr. No.	Description
(4)	For bean definition CL_PRICE_EN where locale is “en”, codelist class is set in JdbcCodeList.
(5)	For bean definition CL_PRICE_JA where locale is “ja”, codelist class is set in JdbcCodeList.

Insert the following data in Table Definition (price table).

locale	code	label
en	0	unlimited
en	10000	Less than \$10,000
en	20000	Less than \$20,000
en	30000	Less than \$30,000
en	40000	Less than \$40,000
en	50000	Less than \$50,000
ja	0	上限なし
ja	10000	10,000 円以下
ja	20000	20,000 円以下
ja	30000	30,000 円以下
ja	40000	40,000 円以下
ja	50000	50,000 円以下

Warning: Currently SimpleI18nCodeList does not support reloadable functionality. It should be noted that even if JdbcCodeList (reloadable CodeList) referred by SimpleI18nCodeList is reloaded, it does not get reflected in SimpleI18nCodeList. In order to make it reloadable, it should be implemented independently. For implementation method, refer to [Customizing the codelist independently](#).

Using codelist in JSP

Description of basic settings is omitted since it is same as [Using codelist in JSP](#) described earlier.

Definition of Bean definition file(spring-mvc.xml)

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**" />
    <bean
      class="org.terasoluna.gfw.web.codelist.CodeListInterceptor">
      <property name="codeListIdPattern" value="CL_.+" />
      <property name="fallbackTo" value="en" />  <!-- (1) -->
    </bean>
  </mvc:interceptor>

  <!-- omitted -->

</mvc:interceptors>
```

Sr. No.	Description
(1)	When request locale is not defined in codelist, codelist is fetched using the locale set in fallbackTo property. When fallbackTo property is not set, default JVM locale is used as fallbackTo property. When codelist cannot be fetched even after using the locale set in fallbackTo property, WARN log is output and empty Map is returned.

Example of jsp implementation

```
<form:select path="basePrice" items="${CL_I18N_PRICE}" />
```

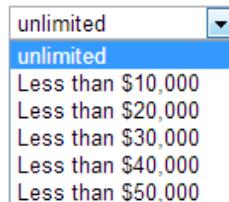
Output HTML lang=en

```
<select id="basePrice" name="basePrice">
  <option value="0">unlimited</option>
  <option value="1">Less than $10,000</option>
  <option value="2">Less than $20,000</option>
  <option value="3">Less than $30,000</option>
  <option value="4">Less than $40,000</option>
  <option value="5">Less than $50,000</option>
</select>
```

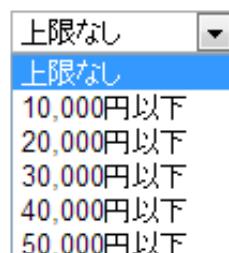
Output HTML lang=ja

```
<select id="basePrice" name="basePrice">
  <option value="0">上限なし</option>
  <option value="1">10,000 円以下</option>
  <option value="2">20,000 円以下</option>
  <option value="3">30,000 円以下</option>
  <option value="4">40,000 円以下</option>
  <option value="5">50,000 円以下</option>
</select>
```

Output screen lang=en



Output screen lang=ja



Using codelist in Java class

Description of basic settings is omitted since it is same as [Using codelist in Java class](#) described earlier.

```
@RequestMapping("orders")
@Controller
public class OrderController {

    @Inject
    @Named("CL_I18N_PRICE")
    I18nCodeList priceCodeList;

    // ...

    @RequestMapping(method = RequestMethod.POST, params = "confirm")
    public String confirm(OrderForm form, Locale locale) {
        // ...
        String priceMassage = getPriceMessage(form.getPriceCode(), locale);
        // ...
    }

    private String getPriceMessage(String targetPrice, Locale locale) {
        return priceCodeList.asMap(locale).get(targetPrice); // (1)
    }

}
```

Sr. No.	Description
(1)	Map of locale corresponding to I18nCodeList#asMap(Locale) can be fetched.

Input validation of code value using codelist

When checking whether the input value is the code value defined in codelist, `org.terasoluna.gfw.common.codelist.ExistInCodeList` annotation for BeanValidation is provided in common library.

For details on BeanValidation and message output method, refer to [Input Validation](#).

Example of @ExistInCodeList settings

See below the example of input validation method using codelist.

Definition of Bean definition file(xxx-codelist.xml)

```
<bean id="CL_GENDER" class="org.terasoluna.gfw.common.codelist.SimpleMapCodeList">
    <property name="map">
        <map>
            <entry key="M" value="Male" />
            <entry key="F" value="Female" />
        </map>
    </property>
</bean>
```

Form object

```
public class Person {
    @ExistInCodeList(codeListId = "CL_GENDER") // (1)
    private String gender;

    // getter and setter omitted
}
```

Sr. No.	Description
(1)	Set @ExistInCodeList annotation for the field for which input is to be validated, and specify the target codelist in codeListId.

As a result of above settings, when characters other than M, F are stored in gender, the system throws an error.

Tip: @ExistInCodeList input validation supports only String or Character data types. Therefore, even if the fields with @ExistInCodeList may contain integer values, they should be defined as String data type. (such as Year/Month/Day)

5.14.3 How to extend

When large number of records need to be read from JdbcCodeList

When large number of records (in hundreds) need to be read from JdbcCodeList, Web application takes time to start.

This is because all records may be fetched at the same time during DB inquiry and it may take time to fetch the list from DB. (fetchSize may be set to Fetch All by default.)

This problem can be resolved by specifying appropriate value for fetchSize. In order to change the fetchSize, it is necessary to set the fetchSize of `org.springframework.jdbc.core.JdbcTemplate`.

See the example below.

Definition of Bean definition file(xxx-infra.xml)

```
<bean id="jdbcTemplateForCodeList" class="org.springframework.jdbc.core.JdbcTemplate" > <!-- (1) -->
    <property name="dataSource" ref="dataSource" />
    <property name="fetchSize" value="1000" /> <!-- (2) -->
</bean>

<bean id="AbstractJdbcCodeList"
    class="org.terasoluna.gfw.common.codelist.JdbcCodeList" abstract="true"> <!-- (3) -->
    <property name="jdbcTemplate" ref="jdbcTemplateForCodeList" /> <!-- (4) -->
</bean>

<bean id="CL_AUTHORITIES" parent="AbstractJdbcCodeList" ><!-- (5) -->
    <property name="querySql"
        value="SELECT authority_id, authority_name FROM authority ORDER BY authority_id" />
    <property name="valueColumn" value="authority_id" />
    <property name="labelColumn" value="authority_name" />
</bean>
```

Sr. No.	Description
(1)	Define a bean of <code>org.springframework.jdbc.core.JdbcTemplate</code> class. It is necessary for setting the <code>fetchSize</code> independently.
(2)	Set an appropriate value for the <code>fetchSize</code> .
(3)	Define a common bean of <code>JdbcCodeList</code> . Common parts of other <code>JdbcCodeList</code> are set. Therefore, for bean definition of basic <code>JdbcCodeList</code> , set this bean definition in parent class. This bean class cannot be instantiated by setting <code>abstract</code> attribute to true.
(4)	Set the <code>jdbcTemplate</code> referring to (1). <code>JdbcTemplate</code> for which <code>fetchSize</code> value is set is stored in <code>JdbcCodeList</code> .
(5)	Bean definition of <code>JdbcCodeList</code> By setting Bean defined in (3) as parent class in <code>parent</code> attribute, <code>JdbcCodeList</code> is set with <code>fetchSize</code> . In this bean definition, only the query related settings are carried out and the required <code>CodeList</code> is created.

When reloading the codelist

Codelist provided in common library is read at the time of launching the application and it is never updated subsequently. However, in some cases, when the master data of the codelist is updated, the codelist also needs to be updated.

Example: Updating the codelist when DB master is updated using `JdbcCodeList`.

Common library provides `org.terasoluna.gfw.common.codelist.ReloadableCodeList` interface. The class implementing the above interface, implements refresh method. Codelist can be updated by calling this refresh method. `JdbcCodeList` implements `ReloadableCodeList` interface; hence it is possible to update the codelist.

Codelist can be updated in following two ways.

1. By using Task Scheduler
2. By calling refresh method in Controller (Service) class

This guideline recommends the method to reload the codelist periodically using [Spring Task Scheduler](#).

However, when it is necessary to arbitrarily refresh the codelist, it is appropriate to call refresh method in Controller class.

Note: For the codelist having ReloadableCodeList interface, refer to [List of codelist types](#).

Using Task Scheduler

Example for setting the Task Scheduler is shown below.

Definition of Bean definition file(xxx-codelist.xml)

```
<task:scheduler id="taskScheduler" pool-size="10"/> <!-- (1) -->

<task:scheduled-tasks scheduler="taskScheduler"> <!-- (2) -->
    <task:scheduled ref="CL_AUTHORITIES" method="refresh" cron="${cron.codelist.refreshTime}"/>
</task:scheduled-tasks>

<bean id="CL_AUTHORITIES" class="org.terasoluna.gfw.common.codelist.JdbcCodeList">
    <property name="dataSource" ref="dataSource" />
    <property name="querySql"
        value="SELECT authority_id, authority_name FROM authority ORDER BY authority_id" />
    <property name="valueColumn" value="authority_id" />
    <property name="labelColumn" value="authority_name" />
</bean>
```

Sr. No.	Description
(1)	<p>Specify the thread pool size in pool-size attribute of <task:scheduler> element. When pool-size attribute is not specified, the value is set to “1”.</p>
(2)	<p>Define <task:scheduled-tasks> element and set <task:scheduler> ID in scheduler attribute.</p>
(3)	<p>Define <task:scheduled> element. Specify refresh method in method attribute. In cron attribute, the value should be mentioned in <code>org.springframework.scheduling.support.CronSequenceGenerator</code> supported format. Reload timing for cron attribute may change with development environment and commercial environment; hence it is recommended to fetch the codelist from property file or environment variable.</p> <p>Example of setting cron attribute</p> <p>Specify in “Seconds Minutes Hours Month Year Day”.</p> <p>execution every second “* * * * *”</p> <p>execution every hour “0 0 * * *”</p> <p>execution every hour 9:00-17:00 on weekdays “0 0 9-17 * * MON-FRI”</p> <p>For details, refer to JavaDoc.</p> <p>http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/scheduling/support/CronSequenceGenerator.html</p>

Calling refresh method in Controller (Service) class

See the example below for directly calling refresh method of JdbcCodeList in Service class.

Definition of Bean definition file(xxx-codelist.xml)

```
<bean id="CL_AUTHORITIES" class="org.terasoluna.gfw.common.codelist.JdbcCodeList">
    <property name="dataSource" ref="dataSource" />
    <property name="querySql"
```

```

        value="SELECT authority_id, authority_name FROM authority ORDER BY authority_id" />
<property name="valueColumn" value="authority_id" />
<property name="labelColumn" value="authority_name" />
</bean>
```

Controller class

```

@Controller
@RequestMapping(value = "codelist")
public class CodeListController {

    @Inject
    CodeListService codeListService; // (1)

    @RequestMapping(method = RequestMethod.GET, params = "refresh")
    public String refreshJdbcCodeList() {
        codeListService.refresh(); // (2)
        return "codelist/jdbcCodeList";
    }
}
```

Sr. No.	Description
(1)	Inject the Service class that executes refresh method of ReloadableCodeList class.
(2)	Execute the refresh method of Service class that executes refresh method of ReloadableCodeList class.

Service class

The description below is given only for the implementation class. Description for interface class has been omitted.

```

@Service
public class CodeListServiceImpl implements CodeListService { // (3)

    @Inject
    @Named(value = "CL_AUTHORITIES") // (4)
    ReloadableCodeList codeListItem; // (5)

    @Override
    public void refresh() { // (6)
        codeListItem.refresh(); // (7)
    }
}
```

Sr. No.	Description
(3)	Implement <code>CodeListService</code> interface for <code>CodeListServiceImpl</code> class.
(4)	Specify the corresponding codelist using <code>@Named</code> annotation at the time of injecting the codelist. ID of the bean to be fetched should be specified in <code>value</code> attribute. Codelist of ID attribute “CL_AUTHORITIES” of bean tag defined in Bean definition file is injected.
(5)	ReloadableCodeList interface should be defined in field type. ReloadableCodeList interface should be implemented for Bean specified in (4).
(6)	refresh method defined in Service class is called from Controller class.
(7)	refresh method of codelist wherein ReloadableCodeList interface is implemented. Codelist is updated by executing refresh method.

Customizing the codelist independently

In order to create a codelist which does not fall under the 4 types provided by the common library, the existing codelist can be customized independently. Refer to the table below for the implementation method and type of codelist that can be created.

Sr. No.	Reloadable	Class to be inherited	Implementation location
(1)	Not required	org.terasoluna.gfw.common.codelist.AbstractCodeList	Override <code>asMap</code>
(2)	Required	org.terasoluna.gfw.common.codelist.AbstractReloadableCodeList	Override <code>retrieveMap</code>

The codelist can be customized by directly implementing `org.terasoluna.gfw.common.codelist.CodeList` and `org.terasoluna.gfw.common.codelist.ReloadableCodeList` interfaces; however extending the abstract class provided in common library minimizes the implementation efforts.

Actual example of independent customization is shown below. It illustrates a codelist for creating a list of current year and the next year. (Example: If current year is 2013, it is stored in codelist in the order of “2013, 2014”.)

Codelist class

```
@Component("CL_YEAR") // (1)
public class DepYearCodeList extends AbstractCodeList { // (2)

    @Inject
    DateFactory dateFactory; // (3)

    @Override
    public Map<String, String> asMap() { // (4)
        DateTime dateTime = dateFactory.newDateTime();
        DateTime nextYearDateTime = dateTime.plusYears(1);

        Map<String, String> depYearMap = new LinkedHashMap<String, String>();

        String thisYear = dateTime.toString("Y");
        String nextYear = nextYearDateTime.toString("Y");
        depYearMap.put(thisYear, thisYear);
        depYearMap.put(nextYear, nextYear);

        return Collections.unmodifiableMap(depYearMap);
    }
}
```

Sr. No.	Description
(1)	Register the codelist as a component using @Component annotation. By specifying "CL_YEAR" in Value, register the codelist as a component using the codelist intercept set in bean definition.
(2)	Inherit org.terasoluna.gfw.common.codelist.AbstractCodeList. When creating the list of current year and next year, reloading is not necessary since it is created dynamically by calculating from system date.
(3)	org.terasoluna.gfw.common.date.DateFactory creating the Date class of system date is injected. Current year and next year can be fetched using DateFactory. Class that implements DateFactory interface should be set in advance in bean definition file.
(4)	Override asMap () method and create the list of current year and next year. Implementation differs with every created codelist.

Example of jsp implementation

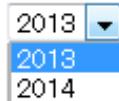
```
<form:select path="mostRecentYear" items="${CL_YEAR}" /> <!-- (5) -->
```

Sr. No.	Description
(5)	"CL_YEAR" registered as component in items attribute should be specified in \${ } placeholder to fetch the corresponding codelist.

Output HTML

```
<select id="mostRecentYear" name="mostRecentYear">
  <option value="2013">2013</option>
  <option value="2014">2014</option>
</select>
```

Output screen



Note: Implementation should be made thread-safe at the time of customizing the reloadable CodeList independently.

5.14.4 Appendix

Setting SimpleI18nCodeList

Apart from the settings mentioned in [How to use SimpleI18nCodeList](#), SimpleI18nCodeList can be set in following 2 ways. The respective setting methods are explained using the example of selectbox for selecting charges.

Set `java.util.Map (key = code value, value = label)` for each locale by rows

Definition of Bean definition file(xxx-codelist.xml)

```
<bean id="CL_I18N_PRICE"
      class="org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList">
    <property name="rows" > <!-- (1) -->
      <util:map>
        <entry key="en">
          <util:map>
            <entry key="0" value="unlimited" />
            <entry key="10000" value="Less than $10,000" />
            <entry key="20000" value="Less than $20,000" />
            <entry key="30000" value="Less than $30,000" />
            <entry key="40000" value="Less than $40,000" />
            <entry key="50000" value="Less than $50,000" />
          </util:map>
        </entry>
        <entry key="ja">
          <util:map>
            <entry key="0" value="unlimited" />
            <entry key="10000" value="10,000 円以下" />
            <entry key="20000" value="20,000 円以下" />
            <entry key="30000" value="30,000 円以下" />
            <entry key="40000" value="40,000 円以下" />
            <entry key="50000" value="50,000 円以下" />
          </util:map>
        </entry>
      </util:map>
    </property>
  </bean>
```

```
        </util:map>
    </entry>
</util:map>
</property>
</bean>
```

Sr. No.	Description
(1)	Set “Map of Map” for rows property. External Map key is java.lang.Locale. Internal Map key is a code value and value is a label corresponding to locale.

Set `java.util.Map(key = locale, value = label)` for each code value by columns

Definition of Bean definition file(xxx-codelist.xml)

```
<bean id="CL_I18N_PRICE"
      class="org.terasoluna.gfw.common.codelist.i18n.SimpleI18nCodeList">
    <property name="columns"> <!-- (1) -->
      <util:map>
        <entry key="0">
          <util:map>
            <entry key="en" value="unlimited" />
            <entry key="ja" value="上限なし" />
          </util:map>
        </entry>
        <entry key="10000">
          <util:map>
            <entry key="en" value="Less than $10,000" />
            <entry key="ja" value="10,000 円以下" />
          </util:map>
        </entry>
        <entry key="20000">
          <util:map>
            <entry key="en" value="Less than $20,000" />
            <entry key="ja" value="20,000 円以下" />
          </util:map>
        </entry>
        <entry key="30000">
          <util:map>
            <entry key="en" value="Less than $30,000" />
            <entry key="ja" value="30,000 円以下" />
          </util:map>
        </entry>
      </util:map>
    </property>
  </bean>
```

```
<entry key="40000">
    <util:map>
        <entry key="en" value="Less than $40,000" />
        <entry key="ja" value="40,000 円以下" />
    </util:map>
</entry>
<entry key="50000">
    <util:map>
        <entry key="en" value="Less than $50,000" />
        <entry key="ja" value="50,000 円以下" />
    </util:map>
</entry>
</util:map>
</property>
</bean>
```

Sr. No.	Description
(1)	Set “Map of Map” for columns property. External Map key is a code value. Internal Map key is java.lang.Locale and value is a label corresponding to locale.

Variations of NumberRangeCodeList

Create the Descending NumberRangeCodeList

Example of setting To value < From value is shown below.

Definition of Bean definition file(xxx-codelist.xml)

```
<bean id="CL_BIRTH_YEAR"
      class="org.terasoluna.gfw.common.codelist.NumberRangeCodeList">
    <property name="from" value="2013" /> <!-- (1) -->
    <property name="to" value="2000" /> <!-- (2) -->
</bean>
```

Sr. No.	Description
(1)	<p>Specify the range start value. Specify a value greater than the one specified in “value” attribute of “to” property.</p> <p>As per this specification, display the values decreased in accordance with the interval in To-From range in descending order.</p> <p>Since interval is not set, default value 1 is applied.</p>
(2)	<p>Specify the range end value.</p> <p>In this example, since 2000 is specified as range end value; the value is reduced by 1 and stored in descending order from 2013 to 2000.</p>

Example of jsp implementation

```
<form:select path="birthYear" items="${CL_BIRTH_YEAR}" />
```

Output HTML

```
<select id="birthYear" name="birthYear">
  <option value="2013">2013</option>
  <option value="2012">2012</option>
  <option value="2011">2011</option>
  <option value="2010">2010</option>
  <option value="2009">2009</option>
  <option value="2008">2008</option>
  <option value="2007">2007</option>
  <option value="2006">2006</option>
  <option value="2005">2005</option>
  <option value="2004">2004</option>
  <option value="2003">2003</option>
  <option value="2002">2002</option>
  <option value="2001">2001</option>
  <option value="2000">2000</option>
</select>
```

Output screen



Change interval of NumberRangeCodeList

Example of setting interval value is shown below.

Definition of Bean definition file(xxx-codelist.xml)

```
<bean id="CL_BULK_ORDER_QUANTITY_UNIT"
      class="org.terasoluna.gfw.common.codelist.NumberRangeCodeList">
    <property name="from" value="10" />
    <property name="to" value="50" />
    <property name="interval" value="10" /> <!-- (1) -->
</bean>
```

Sr. No.	Description
(1)	Specify increment (decrement) value. Then, store the values obtained upon increasing (decreasing) the interval value within From-To range as codelist. In the above example, the values are stored in the order of 10,20,30,40,50 in the codelist.

Example of jsp implementation

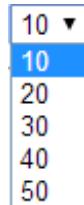
```
<form:select path="quantity" items="${CL_BULK_ORDER_QUANTITY_UNIT}" />
```

Output HTML

```
<select id="quantity" name="quantity">
  <option value="10">10</option>
  <option value="20">20</option>
  <option value="30">30</option>
  <option value="40">40</option>
```

```
<option value="50">50</option>
</select>
```

Output screen



Note: If From-To value exceeds the specified range, then the value increased (decreased) in accordance with interval is not stored in the codelist.

i.e. in case of following definition,

```
<bean id="CL_BULK_ORDER_QUANTITY_UNIT"
      class="org.terasoluna.gfw.common.codelist.NumberRangeCodeList">
    <property name="from" value="10" />
    <property name="to" value="55" />
    <property name="interval" value="10" />
  </bean>
```

5 values of 10,20,30,40,50 are stored in the codelist. The value of subsequent interval 60 and the range threshold value 55 are not stored in the codelist.

5.15 Ajax

5.15.1 Overview

This section explains how to implement applications that use Ajax.

Todo

TBD

Details regarding client side implementation etc. will follow in subsequent versions.

Ajax is the generic term used for group of techniques that perform the following asynchronous processes.

- Screen operations performed on the browser
- HTTP communication with the server triggered by a screen operation and reflecting back the communication results to the user interface

Ajax is often used to improve usability. This is because, screen operations can be continued during HTTP communication.

Typical examples of Ajax are (a) Providing suggestions while searching words and (b) Real time search of a search site.

5.15.2 How to use

Application settings

Settings to enable the Ajax functionality in Spring MVC

Content-Type (such as "application/xml", "application/json" etc.) used in Ajax communication is set such that it can be handled by the processing method of Controller.

- spring-mvc.xml

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

Sr. No.	Description
(1)	If <mvc:annotation-driven> element is specified, the functionality required for Ajax communication is enabled. Therefore, special settings are not necessary for Ajax communication.

Note: Functionalities required for Ajax communication specifically refer to the ones provided by org.springframework.http.converter.HttpMessageConverter class.

HttpMessageConverter performs the following roles.

- Creating Java object from data stored in the request body.
 - Creating the data to be written to the response Body from Java object.
-

The HttpMessageConverter which is enabled by default on specifying <mvc:annotation-driven>, is as follows.

Sr. No.	Class name	Target Format	Description
1.	org.springframework.http.converter.json.MappingJacksonHttpMessageConverter	JSON	HttpMessageConverter to handle JSON as request body or response body. Jackson 1.x system is included in the blank project. Hence, it can be used in its default state.
2.	org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter	XML	HttpMessageConverter to handle XML as request body or response body. JAXB2.0 is included as standard from JavaSE6. Hence it can be used in its default state.

Warning: XXE (XML External Entity) Injection measures

When handling XML format data in Ajax communication, it is necessary to implement [XXE\(XML External Entity\) Injection](#) measure. Subsequent versions above terasoluna-gfw-web 1.0.1.RELEASE are Spring MVC (above 3.2.10.RELEASE) version dependent. As these Spring MVC versions implement XXE Injection measures, it is not necessary to implement them independently.

When using terasoluna-gfw-web 1.0.0.RELEASE, since it is dependent on the Spring MVC version (3.2.4.RELEASE) that does not implement XXE Injection, a class provided by Spring-oxm should be used.

- `spring-mvc.xml`

```

<!-- (1) -->
<bean id="xmlMarshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <property name="packagesToScan" value="com.examples.app" /> <!-- (2) -->
</bean>

<!-- ... -->

<mvc:annotation-driven>

    <mvc:message-converters>
        <!-- (3) -->
        <bean class="org.springframework.http.converter.xml.MarshallingHttpMessageConv
            <property name="marshaller" ref="xmlMarshaller" /> <!-- (4) -->
            <property name="unmarshaller" ref="xmlMarshaller" /> <!-- (5) -->
        </bean>
    </mvc:message-converters>

    <!-- ... -->

</mvc:annotation-driven>

<!-- ... -->

```

Sr. No.	Description
(1)	Perform the bean definition of <code>Jaxb2Marshaller</code> provided by Spring-oxm. <code>Jaxb2Marshaller</code> implements the XXE Injection measures in default state.
(2)	Specify the package name where the JAXB JavaBean (JavaBean assigned with <code>javax.xml.bind.annotation.XmlRootElement annotation</code>) is stored in the <code>packagesToScan</code> property. JAXB JavaBean stored under the specified package is scanned and registered for marshalling or unmarshalling the JavaBean.
5.15. Ajax	It is scanned in the same way as the <code>base-package</code> attribute of <code><context:component-scan></code> .

Implementing Controller

Prerequisites for the sample code explained hereafter, are as follows.

- Response data should be in JSON format.
- JQuery should be used at client side. It should be the latest version of 1.x series (1.10.2), which is used while writing this document.

Fetching data

How to fetch data using Ajax is explained here.

Following example serves as the Ajax communication that returns a list matching with the search word.

- JavaBean for receiving request data

```
// (1)
public class SearchCriteria implements Serializable {
    // omitted

    private String freeWord; // (2)

    // omitted setter/getter
}
```

Sr. No.	Description
(1)	Create the JavaBean that receives request data.
(2)	Match property name with parameter name of request parameter.

- JavaBean for storing the data to be returned

```
// (3)
public class SearchResult implements Serializable {

    // omitted

    private List<XxxEntity> list;

    // omitted setter/getter

}
```

Sr. No.	Description
(3)	Create the JavaBean for storing the data to be returned.

- Controller

```
@RequestMapping(value = "search", method = RequestMethod.GET) // (4)
@ResponseBody // (5)
public SearchResult search(@Validated SearchCriteria criteria) { // (6)

    SearchResult searchResult = new SearchResult(); // (7)

    // (8)
    // omitted

    return searchResult; // (9)
}
```

Sr. No.	Description
(4)	Specify RequestMethod.GET in the method attribute of @RequestMapping annotation.
(5)	Assign @org.springframework.web.bind.annotation.ResponseBody annotation. By assigning this annotation, the returned object is marshalled in JSON format and set in response body.
(6)	Specify the JavaBean that receives request data, as an argument. If input validation is required, specify @Validated. For error handling of input validation, refer to “ <i>Input error handling</i> ”. For details on input validation, refer to “ <i>Input Validation</i> ”.
(7)	Create the JavaBean object to store the data to be returned.
(8)	Search data and store the search result in the object created in (7). In the above example, implementation is omitted.
(9)	Return the object to be marshalled in response body.

- HTML(JSP)

```
<!-- omitted -->

<meta name="contextPath" content="${pageContext.request.contextPath}" />

<!-- omitted -->
```

```
<!-- (10) -->
<form id="searchForm">
  <input name="freeWord" type="text">
  <button onclick="return searchByFreeWord()">Search</button>
</form>
```

Sr. No.	Description
(10)	<p>Form to enter the search condition.</p> <p>In the above example, it has a text box to enter the search condition and a search button.</p>

```
<!-- (11) -->
<script type="text/javascript"
  src="${pageContext.request.contextPath}/resources/vendor/jquery/jquery-1.10.2.js">
</script>
```

Sr. No.	Description
(11)	<p>Read the JQuery JavaScript file.</p> <p>In the above example, request is sent to the /resources/vendor/jquery/jquery-1.10.2.js path, to read the JQuery JavaScript file.</p>

Note: Refer to the settings below to read JQuery JavaScript file. Setting values provided in the blank project are as follows.

- spring-mvc.xml

```
<!-- (12) -->
<mvc:resources mapping="/resources/**"
  location="/resources/, classpath: META-INF/resources/"
  cache-period="#{60 * 60}" />
```

Sr. No.	Description
(12)	Settings for releasing resource files (JavaScript files, Stylesheet files, image files etc.). In the above setting example, when there is a request for path starting with /resources/, the files in /resources/ directory of war file or the /META-INF/resources/ directory of class path are sent as a response.

In the above settings, the JQuery JavaScript file needs to be placed under any one of the following paths.

- /resources/vendor/jquery/jquery-1.10.2.js in war file
It is src/main/webapp/resources/vendor/jquery/jquery-1.10.2.js when indicated by the path in the project.
 - /META-INF/resources/vendor/jquery/jquery-1.10.2.js in class path
It is src/main/resources/META-INF/resources/vendor/jquery/jquery-1.10.2.js when indicated by the path in the project.
-

- **JavaScript**

```
var contextPath = $("meta[name='contextPath']").attr("content");

// (13)
function searchByFreeWord() {
    $.ajax(contextPath + "/ajax/search", {
        type : "GET",
        data : $("#searchForm").serialize(),
        dataType : "json", // (14)

    }).done(function(json) {
        // (15)
        // render search result
        // omitted
    })
}
```

```

});.fail(function(xhr) {
    // (16)
    // render error message
    // omitted

});
return false;
}

```

Sr. No.	Description
(13)	<p>Ajax function that converts search criteria specified in the form to request parameter and sends the request for <i>/ajax/search</i> using GET method.</p> <p>In the above example, clicking the button acts as the trigger for Ajax communication. However, by setting key down or key up of text box as the trigger, real time search can be performed.</p>
(14)	<p>Specify the data format to be received as a response.</p> <p>In the above example, as "json" is specified, "application/json" is set in Accept header.</p>
(15)	<p>Implement the process when Ajax communication ends normally (when Http status code is "200").</p> <p>In the above example, implementation is omitted.</p>
(16)	<p>Implement the process when Ajax communication does not end normally (when Http status code is "4xx" and "5xx").</p> <p>In the above example, implementation is omitted.</p> <p>For error process implementation, refer to <i>Posting form data</i>.</p>

Tip: In the above example, by setting context path (`#{pageContext.request.contextPath}`) of Web application in HTML “`<meta>`“ element. JSP code is deleted from JavaScript code.

Communication is as follows when “Search” button of Search form is clicked.

Main points are highlighted.

- Request data

```
GET /terasoluna-gfw-web-blank/ajax/search?freeWord= HTTP/1.1
Host: localhost:9999
Connection: keep-alive
Accept: application/json, text/javascript, */*; q=0.01
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/30.0.
Referer: http://localhost:9999/terasoluna-gfw-web-blank/ajax/xxe
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,ja;q=0.6
Cookie: JSESSIONID=3A486604D7DEE62032BA6C073FC6BE9F
```

- Response data

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: a8fb8fefaaaf64ee2bffc2b0f77050226
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Fri, 25 Oct 2013 13:52:55 GMT

{"list":[]}
```

Posting form data

How to post form data and fetch processing result using Ajax, is explained here.

Following example is about the Ajax communication of receiving two numbers and returning the calculation result.

- JavaBean to receive form data

```
// (1)
public class CalculationParameters implements Serializable {
```

```
// omitted

private Integer number1;

private Integer number2;

// omitted setter/getter

}
```

Sr. No.	Description
(1)	Create the JavaBean for receiving form data.

- JavaBean that stores processing result

```
// (2)
public class CalculationResult implements Serializable {

    // omitted

    private int resultNumber;

    // omitted setter/getter

}
```

Sr. No.	Description
(2)	Create the JavaBean that stores processing result.

- Controller

```
@RequestMapping("xxx")
@Controller
public class XxxController {

    @RequestMapping(value = "plusForForm", method = RequestMethod.POST) // (3)
    @ResponseBody
    public CalculationResult plusForForm(
        @Validated CalculationParameters params) { // (4)
        CalculationResult result = new CalculationResult();
        int sum = params.getNumber1() + params.getNumber2();
        result.setResultNumber(sum); // (5)
        return result; // (6)
    }

    // omitted
}
```

Sr. No.	Description
(3)	Specify RequestMethod.POST in the method attribute of @RequestMapping annotation.
(4)	Specify the JavaBean for receiving form data as an argument. Specify @Validated when input validation is required. For handling input validation errors, refer to “ <i>Input error handling</i> ”. For details on input validation, refer to “ <i>Input Validation</i> ”.
(5)	Store the processing result in the object created for the same. In the above example, calculation result of the two numbers fetched from form object, is stored.
(6)	Return the object to perform marshalling in response body.

- HTML (JSP)

```

<!-- omitted -->

<meta name="contextPath" content="${pageContext.request.contextPath}" />

<meta name="_csrf" content="${_csrf.token}" />
<meta name="_csrf_header" content="${_csrf.headerName}" />

<!-- omitted -->

<!-- (7) -->
<form id="calculationForm">
    <input name="number1" type="text">+
    <input name="number2" type="text">
    <button onclick="return plus()">=</button>
    <span id="calculationResult"></span> <!-- (8) -->
</form>

```

Sr. No.	Description
(7)	Form to enter the numerical value to be calculated.
(8)	Area to display calculation result. In the above example, calculation result is displayed when communication is successful and it is cleared when the communication fails.

- JavaScript

```

var contextPath = $("meta[name='contextPath']").attr("content");

// (9)
var csrfToken = $("meta[name='_csrf']").attr("content");
var csrfHeaderName = $("meta[name='_csrf_header']").attr("content");
$(document).ajaxSend(function(event, xhr, options) {
    xhr.setRequestHeader(csrfHeaderName, csrfToken);
});

// (10)
function plus() {
    $.ajax(contextPath + "/ajax/plusForForm", {
        type : "POST",

```

```
        data : $("#calculationForm").serialize(),
        dataType : "json"
    }).done(function(json) {
        $("#calculationResult").text(json.resultNumber);

    }).fail(function(xhr) {
        // (11)
        var messages = "";
        // (12)
        if(400 <= xhr.status && xhr.status <= 499) {
            // (13)
            var contentType = xhr.getResponseHeader('Content-Type');
            if (contentType != null && contentType.indexOf("json") != -1) {
                // (14)
                json = $.parseJSON(xhr.responseText);
                $(json.errorResults).each(function(i, errorResult) {
                    messages += ("<div>" + errorResult.message + "</div>");
                });
            } else {
                // (15)
                messages = ("<div>" + xhr.statusText + "</div>");
            }
        } else{
            // (16)
            messages = ("<div>" + "System error occurred." + "</div>");
        }
        // (17)
        $("#calculationResult").html(messages);
    });

    return false;
}
```

Sr. No.	Description
(9)	<p>To send the request using POST method, CSRF token needs to be set to HTTP header.</p> <p>In the above example, the header name and token value are set in the <code><meta></code> element of HTML and value is fetched by JavaScript.</p> <p>For details on CSRF measures, refer to <i>[coming soon] CSRF(Cross Site Request Forgeries) Countermeasures.</i></p>
(10)	<p>Ajax function that converts the numerical value specified in form, to request parameter and sends the request for <code>/ajax/plusForForm</code> using POST method.</p> <p>In the above example, clicking the button acts as the trigger for Ajax communication however, real time calculation can be implemented by setting lost focus of the text box as the trigger.</p>
(11)	<p>Implementation of error handling is shown below.</p> <p>For server side implementation of error handling, refer to <i>Input error handling.</i></p>
(12)	<p>Determine the HTTP status code and type of error.</p> <p>HTTP status code is stored in the <code>status</code> field of XMLHttpRequest object.</p>
(13)	<p>Check whether the response data is in JSON format.</p> <p>In the above example, response data format is checked by referring to the value set in the Content-Type of response header.</p> <p>If the format is not checked and if it a format other than JSON, an error occurs while deserializing to JSON object.</p> <p>If error handling is performed easily at the server side, page may be returned in HTML format.</p>
(14)	<p>Deserialize the response data in JSON object.</p> <p>Response data is stored in the <code>responseText</code> field of XMLHttpRequest object.</p> <p>In the above example, error information is fetched from the serialized JSON object and error message is created.</p>
5.15. Ajax (15)	<p>Perform the process when the response data is not in JSON format.</p> <p>In the above example, HTTP status text is stored in the error message.</p> <p>HTTP status text is stored in the <code>statusText</code> field of XMLHttpRequest object.</p>

Warning: In the above example, processes namely, Ajax communication, DOM operation (rendering) and error handling are performed by the same function. It is recommended to split and implement these processes.

Todo

TBD

Implementation at client side will be explained in detail, in subsequent versions.

Tip: In the above example, JSP code is deleted from JavaScript code by setting CSRF token value and CSRF token header name, in the `<meta>` element of HTML. Please refer, [*csrf_ajax-token-setting*](#).

Please note that, CSRF token value and name of CSRF token header can also be fetched by using `$_csrf.token` and `$_csrf.headerName` respectively.

Following communication occurs when the “=” button of search form is clicked.

Main points are highlighted.

- Request data

```
POST /terasoluna-gfw-web-blank/ajax/plusForForm HTTP/1.1
Host: localhost:9999
Connection: keep-alive
Content-Length: 19
Accept: application/json, text/javascript, */*; q=0.01
Origin: http://localhost:9999
X-CSRF-TOKEN: a5dd1858-8a4f-4ecc-88bd-a326388ab5c9
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/30.0.
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Referer: http://localhost:9999/terasoluna-gfw-web-blank/ajax/xxe
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,ja;q=0.6
Cookie: JSESSIONID=3A486604D7DEE62032BA6C073FC6BE9F

number1=1&number2=2
```

- Response data

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: c2d5066d0fa946f584536775f07d1900
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Fri, 25 Oct 2013 14:27:55 GMT

{"resultNumber":3}
```

- Response data in case of an input error

```
HTTP/1.1 400 Bad Request
Server: Apache-Coyote/1.1
X-Track: cecd7b4d746249178643b7110b0eaa74
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 04 Dec 2013 15:06:01 GMT
Connection: close

{"errorResults":[{"code":"NotNull","message":"\"number2\"maynotbenull.","itemPath":"number2"}]
```

Posting form data in JSON format

How to fetch processing result by converting form data to JSON format and subsequently posting it using Ajax, is explained here.

Difference between this method and “Posting form data” method, is explained.

- Controller

```
@RequestMapping("xxx")
@Controller
public class XxxController {

    @RequestMapping(value = "plusForJson", method = RequestMethod.POST)
    @ResponseBody
```

```
public CalculationResult plusForJson(
    @Validated @RequestBody CalculationParameters params) { // (1)
    CalculationResult result = new CalculationResult();
    int sum = params.getNumber1() + params.getNumber2();
    result.setResultNumber(sum);
    return result;
}

// omitted

}
```

Sr. No.	Description
(1)	<p>Assign <code>@org.springframework.web.bind.annotation.RequestBody</code> as the argument annotation of JavaBean for receiving form data.</p> <p>By assigning this annotation, data in JSON format stored in the request body is unmarshalled and converted to object.</p> <p>Specify <code>@Validated</code> when input validation is required. For error handling of input validation, refer to “Input error handling”.</p> <p>For details on input validation, refer to Input Validation.</p>

- JavaScript/HTML (JSP)

```
// (2)
function toJson($form) {
    var data = {};
    $($form.serializeArray()).each(function(i, v) {
        data[v.name] = v.value;
    });
    return JSON.stringify(data);
}

function plus() {

    $.ajax(contextPath + "/ajax/plusForJson", {
        type : "POST",
        contentType : "application/json; charset=utf-8", // (3)
        data : toJson($("#calculationForm")), // (2)
        dataType : "json",
    })
}
```

```

        beforeSend : function(xhr) {
            xhr.setRequestHeader(csrfHeaderName, csrfToken);
        }

    }).done(function(json) {
        $("#calculationResult").text(json.resultNumber);

    }).fail(function(xhr) {
        $("#calculationResult").text("");

    });
    return false;
}

```

Sr. No.	Description
(2)	Function to change form input field to JSON string format.
(3)	Change the media type of Content-Type to "application/json" as the data stored in request body is in JSON format.

Following communication occurs when “=” button of the search form mentioned above, is clicked.

Main points are highlighted.

- Request data

```

POST /terasoluna-gfw-web-blank/ajax/plusForJson HTTP/1.1
Host: localhost:9999
Connection: keep-alive
Content-Length: 31
Accept: application/json, text/javascript, */*; q=0.01
Origin: http://localhost:9999
X-CSRF-TOKEN: 9d4f1e0c-c500-43f3-9125-a7a131ff88fa
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/30.0.
Content-Type: application/json; charset=UTF-8
Referer: http://localhost:9999/terasoluna-gfw-web-blank/ajax/xxe?
Accept-Encoding: gzip,deflate,sdch

```

```
Accept-Language: en-US,en;q=0.8,ja;q=0.6
Cookie: JSESSIONID=CECD7A6CB0431266B8D1173CCFA66B95

{"number1":"34","number2":"56"}
```

Input error handling

How to perform error handling when an incorrect input value is specified, is explained here.

Input error handling methods are widely classified into the following.

- Method that performs error handling by providing an exception handling method.
- Method that performs error handling by receiving `org.springframework.validation.BindingResult` as an argument of Controller processing method.

Handling BindException

`org.springframework.validation.BindException` is an exception class generated when an incorrect input value is specified while sending the data as request parameter for binding to JavaBean.

To receive request parameter and form data at the time of GET, in `"application/x-www-form-urlencoded"` format, exception handling of `BindException` class needs to be performed.

- Controller

```
@RequestMapping("xxx")
@Controller
public class XxxController {

    // omitted

    @ExceptionHandler(BindException.class) // (1)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST) // (2)
    @ResponseBody // (3)
    public ErrorResults handleBindException(BindException e, Locale locale) { // (4)
        // (5)
        ErrorResults errorResults = new ErrorResults();
    }
}
```

```

    for (FieldError fieldError : e.getBindingResult().getFieldErrors()) {
        errorResults.add(fieldError.getCode(),
                         messageSource.getMessage(fieldError, locale),
                         fieldError.getField());
    }
    for (ObjectError objectError : e.getBindingResult().getGlobalErrors()) {
        errorResults.add(objectError.getCode(),
                         messageSource.getMessage(objectError, locale),
                         objectError.getObjectName());
    }
    return errorResults;
}

// omitted
}

```

Sr. No.	Description
(1)	<p>Define the error handling method in Controller.</p> <p>Assign <code>@org.springframework.web.bind.annotation.ExceptionHandler</code> annotation to the error handling method and specify the exception type to be handled in the <code>value</code> attribute.</p> <p>In the above example, <code>BindException.class</code> is specified as the exception for binding.</p>
(2)	<p>Specify the HTTP status information sent as response.</p> <p>In the above example, <code>400 (Bad Request)</code> is specified.</p>
(3)	<p>Assign <code>@ResponseBody</code> annotation to write the returned object in response body.</p>
(4)	<p>Declare the exception class to be handled as an argument of the error handling method.</p>
(5)	<p>Implement error handling.</p> <p>In the above example, a JavaBean is created to return the error information.</p>

Tip: Locale object can be received as an argument while creating a message for error handling by consid-

ering internationalization.

- JavaBean storing the error information

```
// (6)
public class ErrorResult implements Serializable {

    private static final long serialVersionUID = 1L;

    private String code;

    private String message;

    private String itemPath;

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String getItemPath() {
        return itemPath;
    }

    public void setItemPath(String itemPath) {
        this.itemPath = itemPath;
    }
}

// (7)
public class ErrorResults implements Serializable {

    private static final long serialVersionUID = 1L;
```

```
private List<ErrorResult> errorResults = new ArrayList<ErrorResult>();

public List<ErrorResult> getErrorResults() {
    return errorResults;
}

public void setErrorResults(List<ErrorResult> errorResults) {
    this.errorResults = errorResults;
}

public ErrorResults add(String code, String message) {
    ErrorResult errorResult = new ErrorResult();
    errorResult.setCode(code);
    errorResult.setMessage(message);
    errorResults.add(errorResult);
    return this;
}

public ErrorResults add(String code, String message, String itemPath) {
    ErrorResult errorResult = new ErrorResult();
    errorResult.setCode(code);
    errorResult.setMessage(message);
    errorResult.setItemPath(itemPath);
    errorResults.add(errorResult);
    return this;
}

}
```

Sr. No.	Description
(6)	JavaBean to store one record of error information.
(7)	JavaBean to store multiple JavaBeans, each of which stores one record of error information. JavaBeans mentioned in (6) are stored as a list.

Handling MethodArgumentNotValidException

`org.springframework.web.bind.MethodArgumentNotValidException` is the exception class generated when an incorrect input value is specified while binding the data stored in the request body to JavaBean using `@RequestBody` annotation.

To receive it in formats such as "application/json" or "application/xml" etc., exception handling of `MethodArgumentNotValidException` needs to be performed.

- Controller

```
@ExceptionHandler(MethodArgumentNotValidException.class) // (1)
@ResponseBody(value = HttpStatus.BAD_REQUEST)
public ErrorResults handleMethodArgumentNotValidException(
    MethodArgumentNotValidException e, Locale locale) { // (1)
    ErrorResults errorResults = new ErrorResults();

    // implement error handling.
    // omitted

    return errorResults;
}
```

Sr. No.	Description
(1)	Specify <code>MethodArgumentNotValidException.class</code> as an exception for error handling. Other than this, it is same as <code>BindException</code> .

Handling HttpMessageNotReadableException

`org.springframework.http.converter.HttpMessageNotReadableException` is the exception class generated when a JavaBean could not be created from the data stored in Body, while binding the data stored in the request body to JavaBean, using `@RequestBody` annotation.

To receive it in formats such as "application/json" or "application/xml" etc., exception handling of `MethodArgumentNotValidException` needs to be performed.

Note: Causes of specific errors differ depending on the implementation of `HttpMessageConverter` or library to be used.

In `MappingJacksonHttpMessageConverter` implementation, wherein data in JSON format is to be converted to JavaBean using Jackson, if a string is specified in the Integer field instead of number, `HttpMessageNotReadableException` occurs.

- Controller

```
@ExceptionHandler(HttpMessageNotReadableException.class) // (1)
@ResponseStatus(value = HttpStatus.BAD_REQUEST)
@ResponseBody
public ErrorResults handleHttpMessageNotReadableException(
    HttpMessageNotReadableException e, Locale locale) { // (1)
    ErrorResults errorResults = new ErrorResults();

    // implement error handling.
    // omitted

    return errorResults;
}
```

Sr. No.	Description
(1)	Specify <code>HttpMessageNotReadableException.class</code> as the exception of error handling object. Other than this, it is same as <code>BindException</code> .

Handling by using `BindingResult`

When same type of JavaBean is returned in case of normal termination and in case of input error, error handling can be performed by receiving `BindingResult` as the processing method argument.

This method can be used irrespective of the request data format.

When `BindingResult` is not to be specified as processing method argument, it is necessary to implement error handling by the exception handling method mentioned earlier.

- Controller

```
@RequestMapping(value = "plus", method = RequestMethod.POST)
@ResponseBody
public CalculationResult plus(
    @Validated @RequestBody CalculationParameters params,
    BindingResult bResult) { // (1)
    CalculationResult result = new CalculationResult();
    if (bResult.hasErrors()) { // (2)

        // (3)
        // implement error handling.
        // omitted

        return result; // (4)
    }
    int sum = params.getNumber1() + params.getNumber2();
    result.setResultNumber(sum);
    return result;
}
```

Sr. No.	Description
(1)	Declare BindingResult as a processing method argument. BindingResult needs to be declared immediately after the JavaBean for input validation.
(2)	Check whether there is any input value error.
(3)	If so, perform error handling for input error. In the above example, although error handling is omitted, it is assumed that settings for error message etc. are performed.
(4)	Return processing result.

Note: In the above example, HTTP status code 200 (OK) is returned as response for both normal process as well as error. When it is necessary to classify HTTP status codes as per processing results, it can be implemented by setting `org.springframework.http.ResponseEntity` as the return value. As another approach, error handling can be implemented by the exception handling method mentioned earlier, without specifying BindingResult as the processing method argument.

```
@RequestMapping(value = "plus", method = RequestMethod.POST)
@ResponseBody
public ResponseEntity<CalculationResult> plus(
    @Validated @RequestBody CalculationParameters params,
    BindingResult bResult) {
    CalculationResult result = new CalculationResult();
    if (bResult.hasErrors()) {

        // implement error handling.
        // omitted

        // (1)
        return new ResponseEntity<CalculationResult>(result, HttpStatus.BAD_REQUEST);
    }
    // omitted

    // (2)
    return new ResponseEntity<CalculationResult>(result, HttpStatus.OK);
}
```

Sr. No.	Description
(1)	Return response data and HTTP status in case of input error.
(2)	Return response data and HTTP status in case of normal termination.

Business error handling

How to handle business errors is explained here.

Methods that handle business errors are widely classified as follows.

- Method that performs error handling by providing a business exception handling method.
- Method that catches business exception in the processing method of Controller and performs error handling.

Handling business exception by exception handling method

Business exceptions are handled by providing an exception handling method same as in case of input error.

This method is recommended when it is necessary to implement the same error handling in requests for multiple processing methods.

- Controller

```
@ExceptionHandler(BusinessException.class) // (1)
@ResponseBody(value = HttpStatus.CONFLICT) // (2)
public ErrorResults handleHttpBusinessException(BusinessException e, // (1)
    Locale locale) {
    ErrorResults errorResults = new ErrorResults();

    // implement error handling.
    // omitted

    return errorResults;
}
```

Sr. No.	Description
(1)	Specify <code>BusinessException.class</code> as an exception for error handling. Other than this, it is similar to the input error handling for <code>BindException</code> .
(2)	Specify the HTTP status information sent as response. In the above example, 409 (Conflict) is specified.

Handling business exception in processing method

Business exception is caught by enclosing the process where the business error has occurred, in try clause.

This method is implemented when error handling is different for each request.

- Controller

```

@RequestMapping(value = "plus", method = RequestMethod.POST)
@ResponseBody
public ResponseEntity<CalculationResult> plusForJson(
    @Validated @RequestBody CalculationParameters params) {
    CalculationResult result = new CalculationResult();

    // omitted

    // (1)
    try {

        // call service method.
        // omitted

        // (2)
    } catch (BusinessException e) {

        // (3)
        // implement error handling.
        // omitted

        return new ResponseEntity<CalculationResult>(result, HttpStatus.CONFLICT);
    }

    // omitted

    return new ResponseEntity<CalculationResult>(result, HttpStatus.OK);
}

```

Sr. No.	Description
(1)	Enclose the method call where business exception occurs, in try clause.
(2)	Catch business exception.
(3)	Perform the error handling intended for business exception error. In the above example, although error handling is omitted, it is assumed that settings for error message etc. are performed.

5.16 RESTful Web Service

5.16.1 Overview

This section explains the basic concept of RESTful Web Service and its development by using Spring MVC.

Refer to the following for basic description of architecture, design and implementation of RESTful Web Service

- “*Architecture*“

Basic architecture of RESTful Web Service is explained.

- “*How to design*“

Points to be considered while designing a RESTful Web Service are explained.

- “*How to use*“

Application structure of RESTful Web Service and API implementation methods are explained.

What is RESTful Web Service

REST is an abbreviation of “REpresentational State Transfer”, and is one of the **architecture styles** for building an application, wherein data is exchanged between client and server.

REST architecture style consists of various important fundamental rules and the services which are in accordance with these rules (system etc.) are expressed as **RESTful**.

In other words, “RESTful Web Service” is a Web service that is built in accordance with the fundamental rules of REST.

The concept of “resource” is of prime importance in RESTful Web Service.

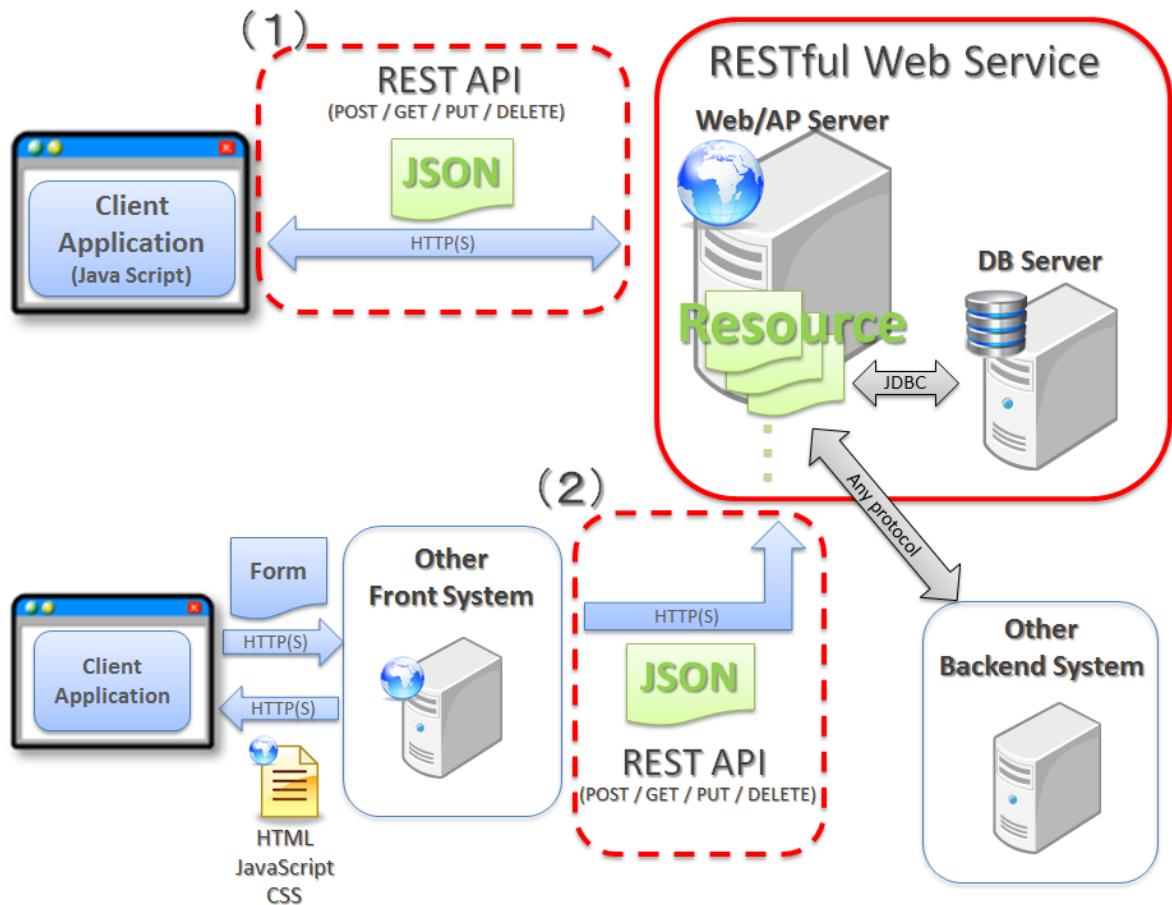
In RESTful Web Service, the information that should be provided to the client is extracted as “resource”, from the information stored in database etc. and the CRUD operation for this extracted “resource” is provided to the client using HTTP methods (POST/GET/PUT/DELETE).

The CRUD operation for “resource” is called “REST API” or “RESTful API” and is mentioned as “REST API” in this guideline.

Further, JSON or XML that have higher message visibility and data structure expressivity, are used as the message formats while exchanging resources between client and server.

System configuration of the application that uses RESTful Web Service mainly consists of following 2 patterns.

The basic architecture for exchanging resources between client and server is explained by using “Architecture”.



Sr. No.	Description
(1)	<p>A resource is directly exchanged between client application with user interface and RESTful Web Service.</p> <p>This pattern is used to separate user interface dependent logic with higher number of requirement & specification changes and the logic for a data model which is more universal with less number of changes.</p>
(2)	<p>Rather than directly exchanging the resource with client applications having user interface, the resource is exchanged between systems.</p> <p>This pattern is used while building a system wherein, the business data stored by each system is managed centrally.</p>

RESTful Web Service development

RESTful Web Service is developed in TERASOLUNA Global Framework using Spring MVC functionalities.

The common functionalities necessary for RESTful Web Service development are built in Spring MVC by default.

As a result, RESTful Web Service development can be initiated without adding any specific settings or implementations.

Main common functionalities built in Spring MVC by default, are given below.

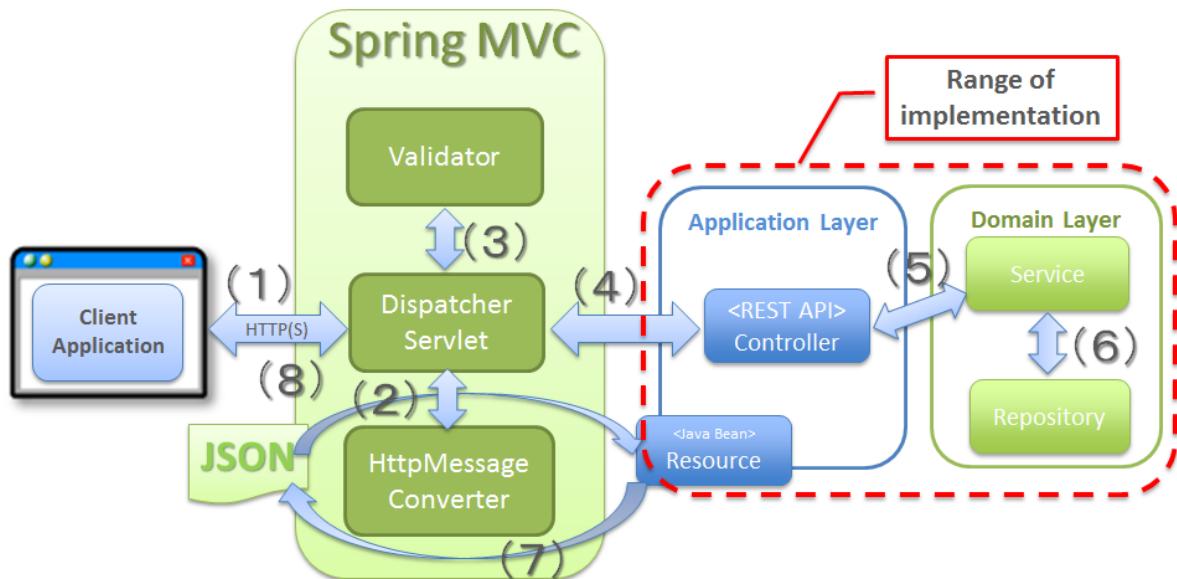
These functionalities can only be enabled by specifying annotations in the methods of the Controller that provides REST API.

Sr. No.	Function overview
(1)	It is a function which converts the JSON or XML format message set in request BODY, to Resource object (JavaBean) and delivers it to the Controller class method (REST API).
(2)	It is a function which implements input validation for the value stored in Resource object (JavaBean) that has been converted from message.
(3)	It is a function which converts Resource object (JavaBean) returned from the Controller class method (REST API) to JSON or XML format and sets it in response BODY.

Note: Exception handling

It is necessary to implement exception handling for each project since a generic functionality for the same is not provided by Spring MVC. For details on exception handling, refer to “[*Implementing exception handling*](#)”.

When RESTful Web Service is developed using Spring MVC, the application is configured as given below. Among these, implementation is necessary for the portion marked with red frame.



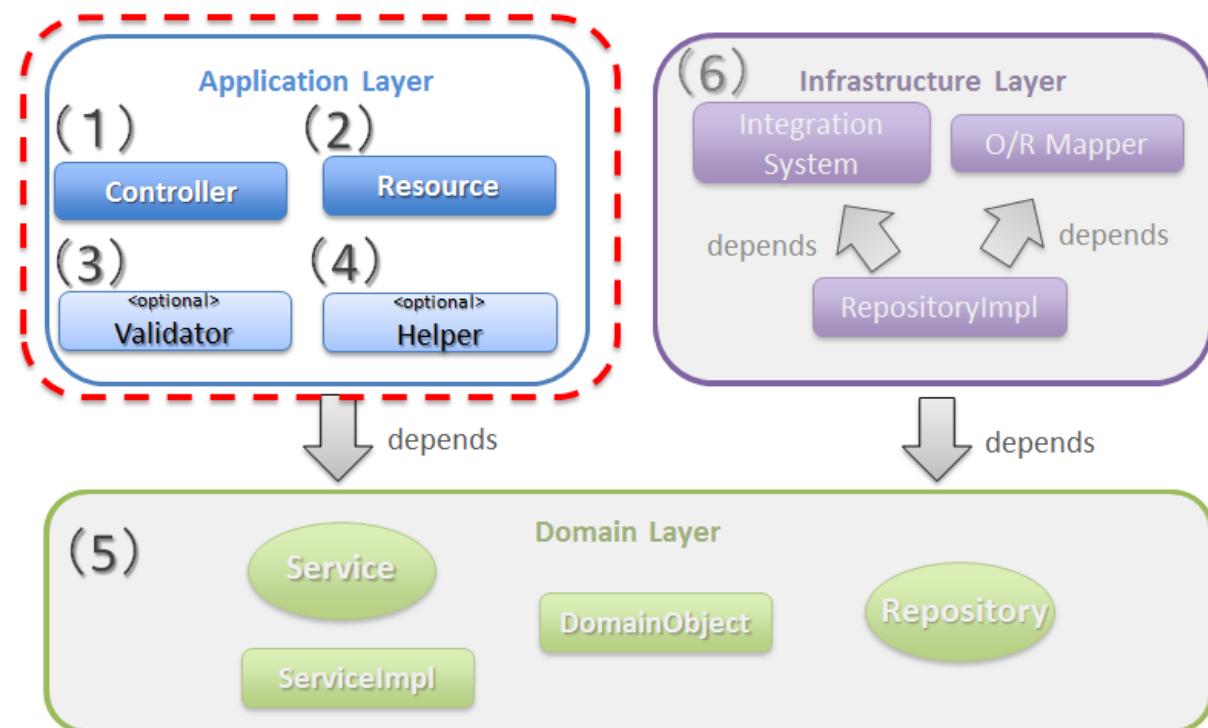
Sr. No.	Process layer	Description
(1)	Spring MVC (Framework)	Spring MVC receives a request from client and determines the REST API (processing method of Controller) to be called.
(2)		Spring MVC converts the JSON format message specified in request BODY to Resource object by using <code>HttpMessageConverter</code> .
(3)		Spring MVC performs input validation for the value stored in Resource object using Validator.
(4)		Spring MVC calls REST API. Here, the Resource that has been converted from JSON and for which input validation is carried out, is delivered to REST API.
(5)	REST API	REST API calls Service method and performs the process for DomainObject such as Entity etc.
(6)		Service method calls the Repository method and performs CRUD process for the DomainObject such as Entity etc.
(7)	Spring MVC (Framework)	Spring MVC converts the Resource object returned from REST API to JSON format message, by using <code>HttpMessageConverter</code> .
(8)		Spring MVC sets JSON format message in response BODY and responds to client.

Configuration for RESTful Web Service module

A lot of RESTful Web Service specific processing can be entrusted to Spring MVC by using the functionalities provided by the framework.

Therefore, configuration of the module to be developed is almost same as the development of conventional Web application that responds with HTML.

Configuration elements of the module are explained below.



- Module for application layer

Sr. No.	Module name	Description
(1)	Controller class	<p>A class that provides REST API.</p> <p>Controller class is created by resource unit and specifies end points (URI) of REST API for each resource.</p> <p>CRUD process for the resource is implemented by delegating it to the Service of domain layer.</p>
(2)	Resource class	<p>Java Bean representing JSON (or XML) that acts as I/O for REST API.</p> <p>Annotation for Bean Validation and annotation for controlling JSON or XML format are specified in this class.</p>
(3)	Validator Class (Optional)	<p>Class that implements correlation validation for input value.</p> <p>If the correlation validation for input value is unnecessary, this class need not be created. Hence, it is considered as optional.</p> <p>For input value correlation validation, refer to “<i>Input Validation</i>”.</p>
(4)	Helper Class (Optional)	<p>Class which implements the process that assists the process to be performed by the Controller.</p> <p>This class is created with the aim of simplifying the Controller processing.</p> <p>Basically, it implements a method that performs conversion of Resource object and DomainObject models.</p> <p>If the model can be converted simply by using copy of the value, “<i>Bean Mapping (Dozer)</i>” may be used without creating the Helper class. Hence, it is considered as optional.</p>

- **Domain layer module**

Sr. No.	Description
(5)	<p>The description is beyond the scope of this section since the module implemented in the domain layer is independent of application type.</p> <p>For role of each module, refer to “<i>Application Layering</i>” and for domain layer development, refer to “<i>Domain Layer Implementation</i>”.</p>

- **Infrastructure layer module**

Sr. No	Description
(6)	<p>The description is beyond the scope of this section since the module implemented in the infrastructure layer is independent of application type.</p> <p>Refer to “<i>Application Layering</i>” for role of each module and “<i>Implementation of Infrastructure Layer</i>” for development of infrastructure layer.</p>

REST API implementation sample

Before giving a detailed explanation, an implementation sample of Resource class and Controller class is given below to let one understand the kind of class created in the application layer.

The implementation sample given below is the REST API of Todo resource which is the topic of `.../TutorialREST/index`.

Note: It is strongly recommended to practice `**:doc:'..TutorialREST/index'**` first, before reading the detailed explanation.

Aim of the tutorial is to emphasize the saying “Practice makes one perfect”. Prior to detailed explanation, the user can gain the experience of actually practicing RESTful Web Service development using TERASOLUNA Global Framework, with the help of this tutorial. When this firsthand experience of RESTful Web Service development is followed by reading the detailed explanation, the user gains a deeper understanding of the development.

Especially when the user does not have any experience of RESTful Web Service development, it is recom-

mended to follow a process in the order namely, “Tutorial practice” → “Detailed explanation of architecture, design and development (described in subsequent sections) → “Tutorial revision (Re-practice)”.

- Resources handled in implementation sample

Resources handled in the implementation sample (Todo resources) are set in following JSON format.

```
{  
    "todoId" : "9aef3ee3-30d4-4a7c-be4a-bc184ca1d558",  
    "todoTitle" : "Hello World!",  
    "finished" : false,  
    "createdAt" : "2014-02-25T02:21:48.493+0000"  
}
```

- Resource class implementation sample

Resource class is created as the JavaBean representing the Todo resources shown above.

```
package todo.api.todo;  
  
import java.util.Date;  
  
import javax.validation.constraints.NotNull;  
import javax.validation.constraints.Size;  
  
public class TodoResource {  
  
    private String todoId;  
  
    @NotNull  
    @Size(min = 1, max = 30)  
    private String todoTitle;  
  
    private boolean finished;  
  
    private Date createdAt;  
  
    public String getTodoId() {  
        return todoId;  
    }
```

```
public void setTodoId(String todoId) {
    this.todoId = todoId;
}

public String getTodoTitle() {
    return todoTitle;
}

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

public boolean isFinished() {
    return finished;
}

public void setFinished(boolean finished) {
    this.finished = finished;
}

public Date getCreatedAt() {
    return createdAt;
}

public void setCreatedAt(Date createdAt) {
    this.createdAt = createdAt;
}
```

- Implementation sample for Controller class (REST API)

Following five REST APIs (Controller processing methods) are created for Todo resource.

Sr. No.	API Name	HTTP Method	Path	Status Code	Description
(1)	GET Todos	GET	/api/v1/todos	200 (OK)	All Todo resources are fetched.
(2)	POST Todos	POST	/api/v1/todos	201 (Created)	A new Todo resource is created.
(3)	GET Todo	GET	/api/v1/todos/{todoId}	200 (OK)	One Todo resource is fetched.
(4)	PUT Todo	PUT	/api/v1/todos/{todoId}	200 (OK)	Todo resource is updated to “completed”.
(5)	DELETE Todo	DELETE	/api/v1/todos/{todoId}	204 (No Content)	Todo resource is deleted.

```
package todo.api.todo;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import javax.inject.Inject;

import org.dozer.Mapper;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
```

```
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

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

@Controller
@RequestMapping("todos")
public class TodoRestController {
    @Inject
    TodoService todoService;
    @Inject
    Mapper beanMapper;

    // (1)
    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public List<TodoResource> getTodos() {
        Collection<Todo> todos = todoService.findAll();
        List<TodoResource> todoResources = new ArrayList<>();
        for (Todo todo : todos) {
            todoResources.add(beanMapper.map(todo, TodoResource.class));
        }
        return todoResources;
    }

    // (2)
    @RequestMapping(method = RequestMethod.POST)
    @ResponseBody
    @ResponseStatus(HttpStatus.CREATED)
    public TodoResource postTodos(@RequestBody @Validated TodoResource todoResource) {
        Todo createdTodo = todoService.create(beanMapper.map(todoResource, Todo.class));
        TodoResource createdTodoResponse = beanMapper.map(createdTodo, TodoResource.class);
        return createdTodoResponse;
    }

    // (3)
    @RequestMapping(value = "{todoId}", method = RequestMethod.GET)
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public TodoResource getTodo(@PathVariable("todoId") String todoId) {
        Todo todo = todoService.findOne(todoId);
        TodoResource todoResource = beanMapper.map(todo, TodoResource.class);
        return todoResource;
    }

    // (4)
    @RequestMapping(value = "{todoId}", method = RequestMethod.PUT)
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
```

```
public TodoResource putTodo(@PathVariable("todoId") String todoId) {
    Todo finishedTodo = todoService.finish(todoId);
    TodoResource finishedTodoResource = beanMapper.map(finishedTodo, TodoResource.class);
    return finishedTodoResource;
}

// (5)
@RequestMapping(value="{todoId}", method = RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteTodo(@PathVariable("todoId") String todoId) {
    todoService.delete(todoId);
}

}
```

5.16.2 Architecture

This section explains the architecture for building a RESTful Web Service.

Resource Oriented Architecture (ROA) is used as the architecture for building RESTful Web Service.

ROA is an abbreviation of “Resource Oriented Architecture” and defines **the basic architecture for building a Web Service in accordance with REST architecture style (rules)**.

It is important to thoroughly understand ROA architecture when creating RESTful Web Service.

This section explains following 7 elements of ROA architecture.

These form important architectural elements for building RESTful Web Service. However, it is not always necessary to apply all of these elements.

Necessary elements should be applied after considering the characteristics of the application to be developed.

Following five architectural elements must be applied regardless of the application characteristics.

Sr. No.	Architecture	Architecture overview
(1)	<i>Publishing as a resource on Web</i>	It is published as a Web resource through which information stored in the system is provided to the client.
(2)	<i>Identifying the resource using URI</i>	URI (Universal Resource Identifier) that can uniquely identify a Web resource is assigned to the resource published to the client.
(3)	<i>Resource operations using HTTP methods</i>	Resource related operations are implemented by using different HTTP methods (GET, POST, PUT and DELETE).
(4)	<i>Using an appropriate format</i>	JSON or XML that represents the data structure, is used as resource format.
(5)	<i>Using the appropriate HTTP status code</i>	Appropriate HTTP status code is set in the response returned to the client.

Following two architectural elements are applied depending on the characteristics of an application.

Sr. No.	Architecture	Architectural elements
(6)	<i>Stateless communication between client and server</i>	This element enables to perform the process only by the information requested from client, without retaining the application status on the server.
(7)	<i>Link to related resource</i>	It includes links to other resources (URI) inside a resource, that are related to the specified resource.

Publishing as a resource on Web

It is published as a resource on Web as the means to provide information stored in the system to client.

It signifies that resources can be accessed using HTTP protocol and URI is used as a method to identify resources.

For example, following information is published on the Web as resource, for a Web system providing shopping site.

- Product information
- Stock information
- Order information
- Member information
- Authentication information for each member (Login ID and password etc.)
- Order history information for each member
- Authentication history information for each member
- and more ...

Identifying the resource using URI

URI (Universal Resource Identifier) that can uniquely identify a resource on the Web, is assigned to the resource to be published to the client.

URL (Uniform Resource Locator), which is a subset of the URI, is actually used.

In ROA, the ability to access a resource on the Web using URI, is called as “Addressability”.

It signifies that on using the same URI, the same resource can be accessed from anywhere.

URI assigned to RESTful Web Service is a combination of “**a noun that indicates the type of resource**” and “**a value (ID etc.) that uniquely identifies a resource**”.

For example, URI of product information handled by a Web system that provides a shopping site, is given below.

- *http://example.com/api/v1/items*

“**items**” portion is the “noun that represents the type of resource”. If there are multiple resources, a plural noun is used.

In the above example, a plural noun is specified to indicate the product information. It forms the URI for batch operation of product information. If replaced to a file system, it corresponds to a directory.

- *http://example.com/api/v1/items/I312-535-01216*

The part “**I312-535-01216**” in the above URI, represents “the value that identifies the resource” and varies for each resource.

In the above example, product ID is specified as the value for uniquely identifying product information. It acts as the URI used to handle specific product information. If replaced by a file system, it corresponds to the files stored in a directory.

Warning: Verbs that indicate operations cannot be included in the URI assigned to RESTful Web Service are as shown below.

- *http://example.com/api/v1/items?get&itemId=I312-535-01216*
- *http://example.com/api/v1/items?delete&itemId=I312-535-01216*

URI mentioned in the above example is not suitable to be assigned to RESTful Web Service since it includes verbs like **get** or **delete**.

In RESTful Web Service, **Resource related operations are represented by using HTTP methods (GET, POST, PUT and DELETE).**

Resource operations using HTTP methods

Resource operations can be performed by using HTTP methods (GET, POST, PUT, DELETE).

In ROA, HTTP methods are called as “Unified interface”.

It implies that HTTP methods can be executed for all the resources published on the Web and that the meaning of HTTP method does not change with each resource.

The association of resource operations assigned to HTTP methods and the post-conditions ensured by each operation, are explained below.

Sr. No.	HTTP method	Resource operations	Post-conditions that the operation should ensure
(1)	GET	Resource is fetched.	Safety, idempotency.
(2)	POST	Resource is created.	Server assigns the URI for created resource, this assigned URI is set to Location header of response and is returned to client.
(4)	PUT	Resource is created or updated.	Idempotency.
(5)	PATCH	Resource difference is updated.	Idempotency.
(6)	DELETE	Resource is deleted.	Idempotency.
(7)	HEAD	Meta information of resource is fetched. Same process as GET is performed and responds with header only.	Safety, Idempotency.
(8)	OPTIONS	Responds with a list of HTTP methods that can be used for resources.	Safety, idempotency.

Note: Ensuring safety and idempotency

When resource operation is performed using HTTP method, it is necessary to ensure “safety” and “idempotency” as post conditions.

[Safety]

It ensures that even if a particular value is multiplied several times by 1, the value does not change. (for example, if 10 is multiplied several times by 1, result remains 10). This guarantees that even if an operation is carried out for several times, resource status does not change.

[Idempotency]

It ensures that even if a value is multiplied a number of times by 0, the value remains 0 (for example, if 10 is multiplied a number of times or just once by 0, the result remains 0). This signifies that once an operation is performed, resource status does not change even if the same operation is performed later for a number of times. However, when another client is modifying the status of the same resource, idempotency need not be ensured and can be handled as a precondition error.

Tip: When client specifies the URI assigned to a resource for creating a resource

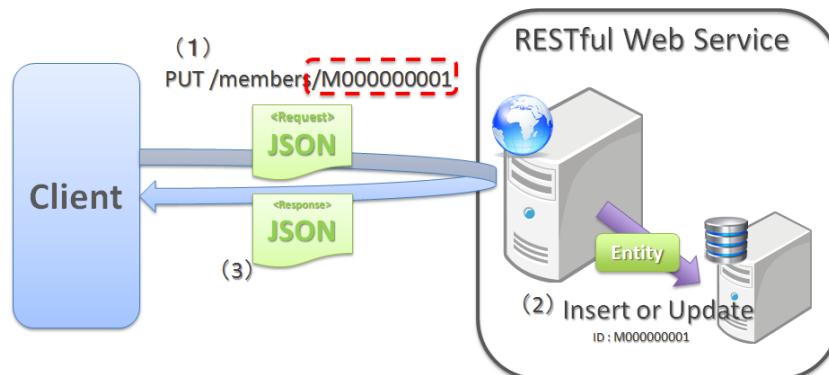
To create a resource, when the URI to be assigned to the resource is specified by client, **PUT method is called for the URI assigned to the resource to be created.**

When creating a resource using PUT method, the general operation is to,

- Create a resource when no resource exists in the specified URI
- Modify resource status when a resource already exists

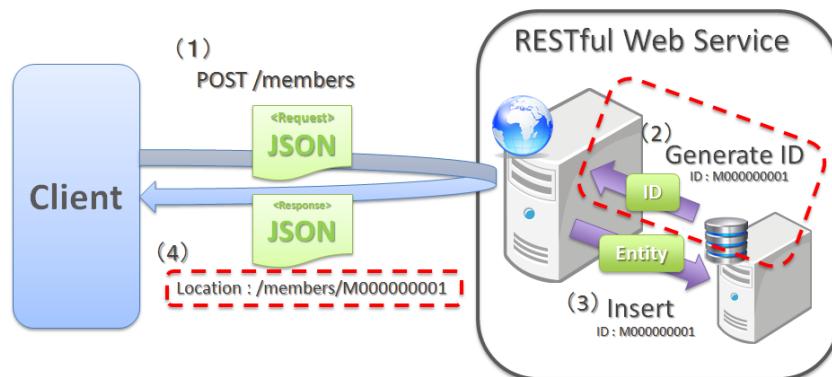
Following is the difference in process images while creating a resource using PUT and POST methods.

[Process image while creating a resource using PUT method]



Sr. No.	Description
(1)	PUT method is called by specifying URI (ID) of the resource to be created in URI.
(2)	Entity is created for the ID specified in URI. If the entity has already been created with same ID, the contents are updated.
(3)	Created or updated resource is sent as a response.

[Process image while creating a resource using POST method]



Sr. No.	Description
(1)	POST method is called.
(2)	ID that identifies the requested resource is generated.
(3)	Entity for the ID generated in (2) is created.
(4)	Created resource is sent as a response. URI for accessing the generated resource is set in the Location header of response.

Using an appropriate format

JSON or XML that indicate data structure, are used for resource format.

However, formats other than JSON or XML can also be used depending on the type of resource.

For example, a resource classified as statistical information can be published with line graph represented in image format (Binary data).

When multiple formats are supported as resource formats, any of the following methods is used to change the format.

- **Changing the format using an extension.**

Response format can be changed by specifying the extension.

This guideline recommends changing the format using extension.

The reasons for recommending this format are, responding format can be easily specified and as the responding format is included in URI, it results in an intuitive URI.

Note: Examples of URI where format is changed using extension

- *http://example.com/api/v1/items.json*
 - *http://example.com/api/v1/items.xml*
 - *http://example.com/api/v1/items/I312-535-01216.json*
 - *http://example.com/api/v1/items/I312-535-01216.xml*
-

- **Changing format by using the MIME type in Accept header of request.**

A typical MIME type used in RESTful Web Service is shown below.

Sr. No.	Format	MIME type
(1)	JSON	application/json
(2)	XML	application/xml

Using the appropriate HTTP status code

Appropriate HTTP status code is set in the response to be returned to the client.

Value indicating the method by which server has processed the request received from the client, is set in HTTP status code.

This is an HTTP specification and it is recommended to conform to the HTTP specifications wherever possible.

Tip: HTTP Specifications

Refer to [RFC 2616 \(Hypertext Transfer Protocol – HTTP/1.1\) - 6.1.1 Status Code and Reason Phrase](#).

In a traditional Web system wherein HTML is returned in the browser, regardless of the process results, it was common that "200 OK" was returned as the response and process results were displayed in entity body (HTML),

In a traditional Web application that returns HTML, there were no issues since an operator (human) determined the process results.

However, if this structure is used to build a RESTful Web Service, following issues may exist potentially. Hence, it is recommended to set appropriate status codes.

Sr. No.	Potential issues
(1)	Even in cases where only the process result (success and failure) is to be determined, unnecessary process has to be performed, as analysis process is mandatory for entity body.
(2)	Since it is mandatory to be aware of the unique error codes defined in the system while handling errors, it may adversely affect the architecture (design and implementation) at the client side.
(3)	Intuitive error analysis may be obstructed when analyzing error causes at client side, since understanding the meaning of unique error codes defined in the system is required for the same.

Stateless communication between client and server

In this communication, only the information requested by the client is processed, without retaining the application status on the server.

In ROA, a state wherein application status is not retained on the server, is called “stateless”.

It signifies that application status is not retained in application server memory (HTTP session etc.) and resource related operations can be completed only by using the requested information.

In this guideline, **it is recommended to retain “stateless” state wherever possible.**

Note: Application status

Web page transition status, selection status for input value, pull down/checkbox/radio buttons and authentication status etc. are included in application status.

Note: Relation with CSRF measures

Please note that the “Stateless” state between client and server cannot be retained when the CSRF measures described in this guideline are implemented for RESTful Web Service as, the token values for CSRF measures are stored in HTTP sessions.

As a result, system availability must be considered while implementing CSRF measures.

Following measures need to be implemented for a system that requires high availability.

- Perform AP server clustering and session replication.
- Use a destination other than AP server memory for storing a session.

However, above measures may affect the performance. Hence, it is necessary to consider performance requirements as well.

For CSRF measures, refer to *[coming soon] CSRF(Cross Site Request Forgeries) Countermeasures*.

Todo

TBD

When high availability is required, it is advisable to review an architecture wherein, “token values for CSRF measures are stored in a destination other than the AP server memory (HTTP session)”.

Basic architecture is currently under review and will be documented in subsequent versions.

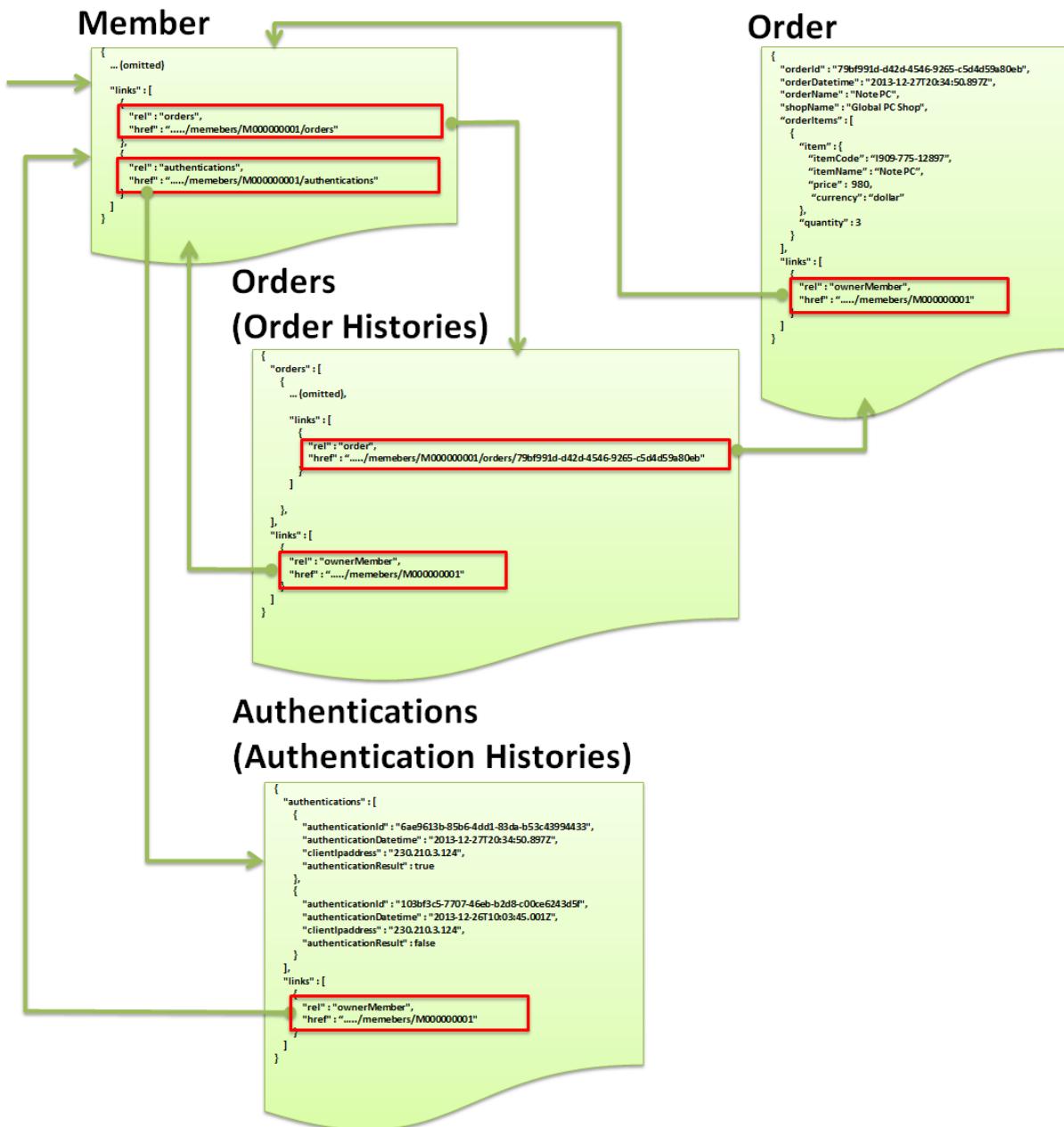
Link to related resource

Hypermedia link (URI) to another resource related to the specified resource, is included in the resource.

In ROA, the process of incorporating a hypermedia link for another resource in the resource status display, is called “Connectivity”.

It signifies that both the linked resources retain this mutual link and all the related resources can be accessed by following this link.

Connectivity of resources is described below, with the example of member information resource of a shopping site.



Sr. No.	Description
(1)	<p>Following JSON is returned when the member information resource is fetched (GET http://example.com/api/v1/members/M000000001).</p> <pre>{ "memberId" : "M000000001", "memberName" : "John Smith", "address" : { "address1" : "45 West 36th Street", "address2" : "7th Floor", "city" : "New York", "state" : "NY", "zipCode" : "10018" }, "links" : [{ "rel" : "self", "href" : ".../members/M000000001" }] }</pre>
5.16. RESTful Web Service	1011

It is not mandatory for a resource to include hypermedia link (URI) to another resource.

When all the endpoints (URI) of REST API are already published, even if the link for related resource is set in the resource, it is highly unlikely that it will be used.

Particularly, it makes no sense to provide links for the REST API that exchanges resources between systems, since REST API end points that are already published can be accessed directly.

There is no need to provide a link where it is not required.

In contrast, when a resource is to be directly exchanged between a client application with user interface and RESTful Web service, the loose coupling between client and server can be enhanced by providing a link.

Following are the reasons for enhancing the coupling between client and server.

Sr. No.	Reasons to enhance the loose coupling
(1)	Client application needs to know only the logical name of the link in advance. Hence, it is not necessary to know the specific URI for calling REST API.
(2)	Since it is not necessary for the client application to know the specific URI, impact on server, owing to change in URI, can be minimized.

Whether to provide a hypermedia link (URI) to other resources should be determined on considering all the points described above.

Tip: Relation with HATEOAS

HATEOAS is an abbreviation for “Hypermedia As The Engine Of Application State” and is one of the architectures for creating a RESTful Web application.

HATEOAS architecture includes the following processes.

- In the resources (JSON or XML) that are exchanged between client and server, the server includes a hypermedia link (URI) to an accessible resource.
- Client fetches required resources from the server through the hypermedia link in the resource display (JSON or XML), and changes application status (screen status etc.).

Therefore, providing a link for related resources is consistent with the HATEOAS architecture.

When loose coupling between server and client is to be enhanced, please review if using the HATEOAS architecture would be beneficial.

5.16.3 How to design

This section explains the design of RESTful Web Service.

Resource extraction

First, the resource published on the Web is extracted.

Precautions while extracting a resource are as given below.

Sr. No.	Precautions while extracting a resource
(1)	<p>Resource published on the web is used as the information managed by database. However, data model of the database must not be published as resource as it is, without careful consideration.</p> <p>It should be closely investigated, as the fields stored in the database may include some fields that should not be disclosed to the client.</p>
(2)	<p>When information type is different in spite of being managed by the same table of the database, publishing it as a separate resource may be considered.</p> <p>There are cases wherein, even if essentially seen as different information, it is managed by the same table, due to same data structure. Hence, such cases need to be reviewed closely.</p>
(3)	<p>In RESTful Web Service, the information operated by an event is extracted as a resource. The event itself should not be extracted as a resource.</p> <p>For example, when creating RESTful Web Service to be called from the events (approve, deny, return etc.) generated by work flow functionality, information for managing the workflow status or the workflow itself, is extracted as a resource.</p>

Assigning URI

URI is assigned to the extracted resource for identifying it.

It is recommended to use following formats for the URI.

- `http(s)://{{Domain name (:Port number)}}/{A value indicating REST API}/{API version}/{path for identifying a resource}`
- `http(s)://{{Domain name indicating REST API(:Port number)}}/{API version}/{path for identifying a resource}`

A typical example is given below.

- `http://example.com/api/v1/members/M000000001`
- `http://api.example.com/v1/members/M000000001`

Assigning a URI that indicates the API as REST API

It is recommended to include `api` within the URI domain or path, to clearly indicate that the URI is intended for RESTful Web Service (REST API).

Typically, the URI is as given below.

- `http://example.com/api/...`
- `http://api.example.com/...`

Assigning a URI for identifying the API version

It is recommended to include a value that identifies the API version, in the URI to be published to the client, since it may be necessary to run RESTful Web Service in multiple versions.

Typically, the URI format is as follows.

- `http://example.com/api/{API version}/{path for identifying a resource}`

- `http://api.example.com/{API version}/{path for identifying a resource}`
-

Todo

TBD

Whether API version should be included in URI, is currently being investigated.

Assigning a path for identifying resource

The 2 URLs given below are assigned for resources that are published on Web.

Following is an example of a URI when publishing member information on Web.

Sr. No.	URI format	Typical example of URI	Description
(1)	<code>/{Noun that represents collection of resources}</code>	<code>/api/v1/members</code>	It is the URI used for batch operations of resources.
(2)	<code>/{Noun that represents collection of resources/resource identifier (ID etc)}</code>	<code>/api/v1/members/M0001</code>	It is the URI used while operating a specific resource.

The URI for related resources published on Web are nested and then displayed.

Following example describes the URI for publishing order information for each member on the Web.

Sr. No.	URI format	Typical example of URI	Description
(3)	{Resource URI}/{Noun representing collection of related resources}	/api/v1/members/M0001/orders	It is the URI used at the time of batch operation of related resources.
(4)	{Resource URI}/{Noun representing collection of related resources}/{Identifier for related resource (ID etc.)}	/api/v1/members/M0001/orders/O0001	It is the URI used when operating a specific related resource.

When the related resource published on Web has a single element, the noun that indicates the related resource should be singular and not plural.

Following is the example of URI to publish credentials of each member on Web.

Sr. No.	URI format	Typical example of URI	Description
(5)	{URI for resource}/{Noun representing related resource}	/api/v1/members/M0001/credential	It is the URI used when operating a related resource with single element.

Assigning HTTP methods

CRUD operation for resources is published as REST API by assigning the following HTTP methods for the URI assigned to each resource.

Note: HEAD and OPTIONS method

Hereafter, HEAD and OPTIONS methods are described as well. However, providing them for REST API is optional.

While creating the REST API conforming to HTTP specifications, it is necessary to provide the HEAD and OPTIONS methods as well. However, it is actually used very rarely and is not required in most of the cases.

Assigning HTTP methods for resource collection URI

Sr. No.	HTTP methods	Overview of the REST API to be implemented
(1)	GET	REST API that fetches collection of resources specified in URI, is implemented.
(2)	POST	REST API that creates and adds the specified resource to the collection is implemented.
(3)	PUT	REST API that performs batch update for resource specified in URI is implemented.
(4)	DELETE	REST API that performs batch deletion for resource specified in URI is implemented.
(5)	HEAD	REST API that fetches meta information of the resource collection specified in URI, is implemented. A process same as GET is performed and only header is sent as response.
(6)	OPTIONS	REST API that responds with the list of HTTP methods (API) supported by resource collection specified in URI, is implemented.

Assigning HTTP methods for URI of specific resources

Sr. No.	HTTP methods	Overview of REST API to be implemented
(1)	GET	REST API that fetches the resource specified in URI is implemented.
(2)	PUT	REST API that creates or updates the resource specified in URI is implemented.
(3)	DELETE	REST API that deletes the resource specified in URI is implemented.
(4)	HEAD	A REST API that fetches meta information of the resource specified in URI is implemented. A process same as GET is performed and only header is sent as a response.
(5)	OPTIONS	REST API that responds with list of HTTP methods (API) supported by the resource specified in URI is implemented.

Resource format

It is recommended to use JSON as the format for displaying a resource.

The explanation hereafter is based on the assumption that JSON is used as the format for displaying a resource.

JSON Field name

It is recommended to use “lower camel case” as the JSON field name.

It is recommended on considering its compatibility with JavaScript, which is assumed as one of the client applications.

The JSON sample with field name set in “lower camel case”, is as given below.

In “lower camel case”, first letter of the word is in lowercase and subsequent first letters of words are in uppercase.

```
{  
    "memberId" : "M000000001"  
}
```

NULL and blank characters

It is recommended to differentiate NULL and blank characters as JSON values.

Although, as application process, NULL and blank characters are often equated, it is advisable to differentiate NULL and blank characters as the value to be set in JSON.

JSON sample wherein NULL and blank characters are differentiated, is given below.

```
{  
    "dateOfBirth" : null,  
    "address1" : ""  
}
```

Date format

It is recommended to use extended ISO-8601 format as the JSON date field format.

Format other than extended ISO-8601 format can be used. However, it is advisable to use the extended ISO-8601 format, if there is no particular reason otherwise.

There are two formats in ISO-8601 namely, basic format and extended format, however, readability is higher in extended format.

Basically, there are following three formats.

1. yyyy-MM-dd

```
{  
    "dateOfBirth" : "1977-03-12"  
}
```

2. yyyy-MM-dd'T'HH:mm:ss.SSSZ

```
{  
    "lastModifiedAt" : "2014-03-12T22:22:36.637+09:00"  
}
```

3. yyyy-MM-dd'T'HH:mm:ss.SSS'Z' (format for UTC)

```
{  
    "lastModifiedAt" : "2014-03-12T13:11:27.356Z"  
}
```

Hypermedia link format

It is recommended to use the following format to create a hypermedia link.

Sample of recommended format is as given below.

```
{  
    "links" : [  
        {  
            "rel" : "ownerMember",  
            "href" : "http://example.com/api/v1/memebers/M000000001"  
        }  
    ]  
}
```

- Link object consisting of 2 fields - "rel" and "href" is retained in collection format.
- Link name for identifying the link is specified in "rel" .
- URI to access the resource is specified in "href" .
- "links" is the field which retains the Link object in collection format.

Format at the time of error response

When an error is detected, it is recommended to use a format that can retain the details of the error occurred.

Detailed error information should be included, especially when there is a possibility of the error being eliminated owing to re-operation by client.

In contrast, detailed error information should not be included when an event that exposes system vulnerability occurs. In such cases, the detailed error information should be output to a log.

Following is an example of the response format when error is detected.

```
{  
  "code" : "e.ex.fw.7001",  
  "message" : "Validation error occurred on item in the request body.",  
  "details" : [ {  
    "code" : "ExistInCodeList",  
    "message" : "\"genderCode\" must exist in code list of CL_GENDER.",  
    "target" : "genderCode"  
  } ]  
}
```

In the above example,

- Error code (code)
- Error message (message)
- Error details list (details)

are provided as error response formats.

It is assumed that the error details list is used when input validation error occurs. It is a format that can retain details like the field in which the error occurred and the information of the error.

HTTP Status Code

HTTP status code is sent as the response, in accordance with the following guidelines.

Sr. No.	Objectives
(1)	When the request is successful, an HTTP status code indicating success or transfer (2xx or 3xx system) is sent as response.
(2)	When the cause of request failure lies at client side, an HTTP status code indicating client error (4xx system) is sent as the response. When client is not responsible for request failure however, when the request may be successful through a re-operation by client, it is still considered as client error.
(3)	When the cause of request failure lies at server side, an HTTP status code indicating server error (5xx system) is sent as the response.

HTTP status codes when the request is successful

When the request is successful, following HTTP status codes are sent as responses, depending on status.

Sr. No.	HTTP Status codes	Description	Applicable conditions
(1)	200 OK	HTTP status code notifying that the request was successful.	It is sent as a response when the resource information corresponding to the request is output in the entity body of response, as a result of successful request,
(2)	201 Created	HTTP status code notifying the creation of a new resource.	It is used when a new resource is created using POST method. URI for created resource is set in the Location header of the response.
(3)	204 No Content	HTTP status code notifying a successful request.	It is sent as a response when the resource information corresponding to request is not output in the entity body of response, as a result of successful request.

Tip: The difference between "200 OK" and "204 No Content" is whether the resource information is output/not output in the response body.

HTTP status code when the cause of request failure lies at client side

When the cause of request failure lies at client side, following HTTP status codes are sent as responses depending on the status.

Status codes that must be identified by individual REST APIs handling the resources, are as given below.

Sr. No.	HTTP Status code	Description	Applicable conditions
(1)	400 Bad Request	HTTP status code notifying that the request syntax or requested value is incorrect.	It is sent as a response when an incomplete JSON or XML format specified in entity body is detected or an incomplete input value is specified in JSON or XML format or in the request parameters.
(2)	404 Not Found	HTTP status code notifying that the specified resource does not exist.	It is sent as a response when resource corresponding to specified URI does not exist in the system.
(3)	409 Conflict	HTTP status code notifying that the process is terminated due to conflict in resource status when the request status is changed by requested contents.	It is sent as a response when an exclusive error or a business error is detected. Conflict details and error details required to resolve the conflict need to be output to the entity body.

Following status codes need not be identified by individual REST APIs which handle the resources.

These status codes need to be identified as the framework or common process.

Sr. No.	HTTP Status codes	Description	Applicable conditions
(4)	405 Method Not Allowed	HTTP status code notifying that the used HTTP method is not supported by the specified resource.	It is sent as a response when an unsupported HTTP method is used. The list of allowed methods is set in the Allow header of response.
(5)	406 Not Acceptable	HTTP status code notifying the inability to receive a request, as the resource status cannot be sent as a response in the specified format.	It is sent as a response when, the format specified in extension or Accept header is not supported as a response format.
(6)	415 Unsupported Media Type	HTTP status code notifying that the request cannot be received, as the format specified in entity body is not supported.	It is sent as a response when an unsupported format is specified in Content-Type header, as request format.

HTTP status code when the cause of request failure lies at server side

When the cause of request failure lies at server side, HTTP status codes given below are sent as responses, depending on the status.

Sr. No.	HTTP Status code	Description	Applicable conditions
(1)	500 Internal Server Error	HTTP status code notifying that an internal error has occurred in the server.	It is sent as a response when an unexpected error has occurred in the server or a status that should not occur during a normal operation is detected.

Authentication and Authorization

Todo

TBD

The guidelines for authentication and authorization control are explained here.

Performing authentication and authorization using OAuth2 protocol will be described in subsequent versions.

Conditional update control of resource

Todo

TBD

The process for conditional update (exclusive control) of a resource using HTTP header is explained here.

Conditional update using headers like Etag/Last-Modified-Since etc. will be described in subsequent versions.

Conditional acquisition control of resource

Todo

TBD

The process for conditional acquisition (304 not modified control) of resource using HTTP header is explained here.

Conditional acquisition using headers like Etag/Last-Modified etc. will be described in subsequent versions.

Cache control of resource

Todo

TBD

Cache control of resources which use HTTP header, is explained here.

Cache control of resources that use headers such as Cache-Control/Pragma/Expires etc. shall be described in subsequent versions.

Versioning

Todo

TBD

Version control of RESTful Web Service and details on performing parallel operations in multiple versions, will be described in subsequent versions.

5.16.4 How to use

This section explains the basic method to create RESTful Web Service.

Web application configuration

While building RESTful Web Service, Web application (war) is built by any one of the following configurations.

It is recommended to build a Web application that is exclusive to RESTful Web Service unless there is a specific reason otherwise.

Sr. No.	Configuration	Description
(1)	An exclusive Web application for RESTful Web Service is built.	<p>It is recommended to build an exclusive Web application (war) for RESTful Web Service when an independence with client application (UI layer application) that uses RESTful Web Service, is to be ensured (is necessary).</p> <p>This method can be used to create RESTful Web Service when there are multiple client applications using RESTful Web Service.</p>
(2)	It is built by providing DispatcherServlet for RESTful Web Service.	<p>When it is not necessary to ensure independence of client application (UI layer application) that uses RESTful Web Service, both the client application and RESTful Web Service can be built as a single Web application (war).</p> <p>However, it is strongly recommended to build it by dividing DispatcherServlet that receives the requests for RESTful Web Service and DispatcherServlet that receives client application requests.</p>

Note: Client application (UI layer application)

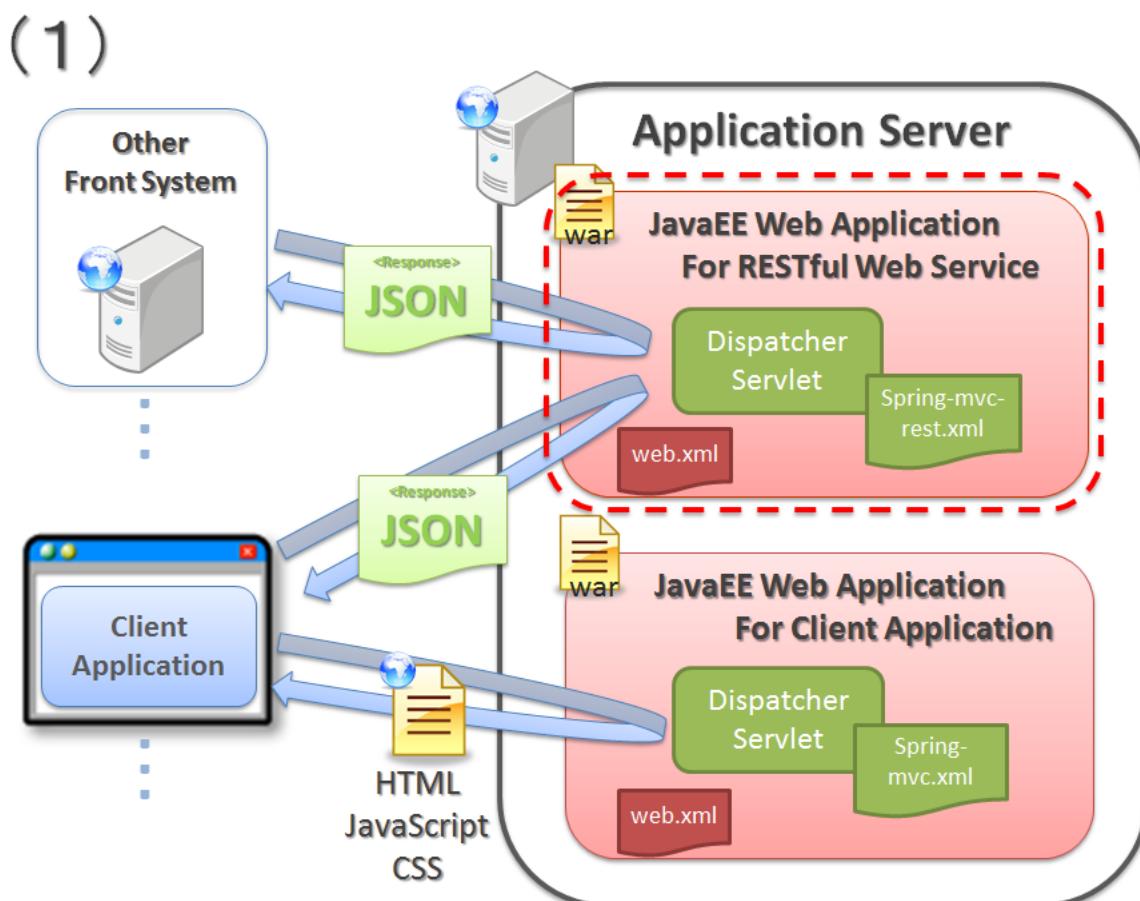
Client application (UI layer application) described here refers to the application that responds with client layer (UI layer) component called CSS (Cascading Style Sheets) and scripts like HTML, JavaScript etc. HTML generated by template engine such as JSP, is also considered.

Note: Why division of DispatcherServlet is recommended

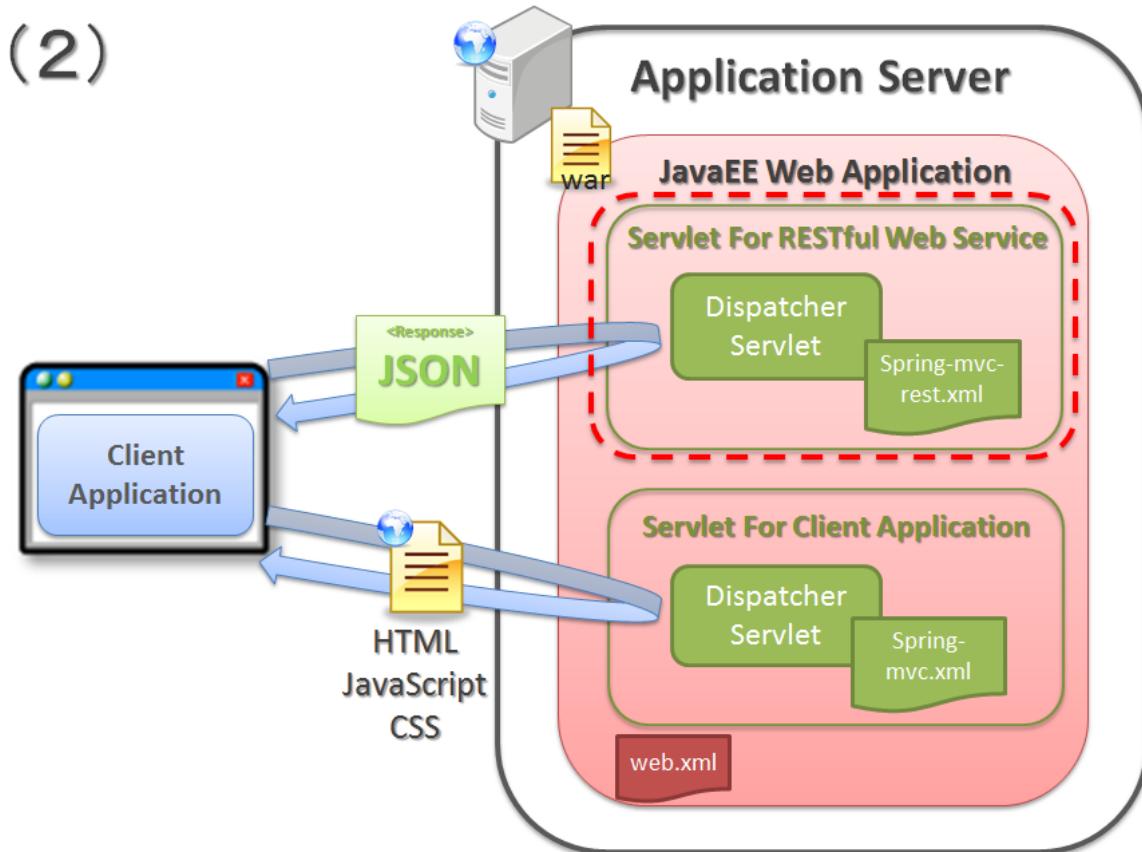
In Spring MVC, operation settings of the application are defined for each `DispatcherServlet`. Therefore, when the requests of RESTful Web Service and client application (UI layer application) are configured to be received from the same `DispatcherServlet`, specific operation settings for RESTful Web Service or client application cannot be defined, thus resulting in complex or cumbersome settings.

In this guideline, when RESTful Web Service and client application are to be configured as same Web application, it is recommended to divide `DispatcherServlet` to avoid occurrence of the issues described above.

Configuration image when building a Web application exclusive to RESTful Web Service, is as follows:



Configuration image when building RESTful Web Service and client application as a single application, is as follows:



Application settings

Application settings for RESTful Web Service are explained below.

Settings for activating the Spring MVC components necessary for RESTful Web Service

A bean definition file is created for RESTful Web Service.

Definitions that are required to operate the sample indicated in the explanation hereafter, are as follows:

- spring-mvc-rest.xml

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

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

    <bean id="jsonMessageConverter"
        class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter">
        <property name="objectMapper">
            <bean id="objectMapper" class="org.codehaus.jackson.map.ObjectMapper">
                <!-- (2) -->
                <property name="dateFormat">
                    <bean class="org.codehaus.jackson.map.util.StdDateFormat" />
                </property>
            </bean>
        </property>
    </bean>

    <!-- Register components of Spring MVC. -->
    <!-- (3) -->
    <mvc:annotation-driven>
        <mvc:message-converters register-defaults="false">
            <ref bean="jsonMessageConverter" />
        </mvc:message-converters>
        <!-- (4) -->
        <mvc:argument-resolvers>
            <bean class="org.springframework.data.web.PageableHandlerMethodArgumentResolver" />
        </mvc:argument-resolvers>
    </mvc:annotation-driven>

    <!-- Register components of interceptor. -->
    <!-- (5) -->
```

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**" />
        <bean class="org.terasoluna.gfw.web.logging.TraceLoggingInterceptor" />
    </mvc:interceptor>
    <!-- omitted -->
</mvc:interceptors>

<!-- Scan & register components of RESTful Web Service. -->
<!-- (6) -->
<context:component-scan base-package="com.example.project.api" />

<!-- Register components of AOP. -->
<!-- (7) -->
<bean id="handlerExceptionResolverLoggingInterceptor"
      class="org.terasoluna.gfw.web.exception.HandlerExceptionResolverLoggingInterceptor">
    <property name="exceptionLogger" ref="exceptionLogger" />
</bean>
<aop:config>
    <aop:advisor advice-ref="handlerExceptionResolverLoggingInterceptor"
                  pointcut="execution(* org.springframework.web.servlet.HandlerExceptionResolver.r
</aop:config>

</beans>
```

Sr. No.	Description
(1)	<p>When the value defined in the property file needs to be referred by an application layer component, the property file should be read by using <code><context:property-placeholder></code> element.</p> <p>For the details of fetching a value from property file, refer to “Properties Management”.</p>
(2)	The settings for handling the JSON date field format as extended ISO-8601 format, are added.
(3)	<p>Bean registration for the Spring MVC framework component necessary for providing RESTful Web Service, is performed.</p> <p>JSON can be used as a resource format by performing these settings.</p> <p>In the above example, resource format is restricted to JSON since the register-defaults attribute of <code><mvc:message-converters></code> element is set as <code>false</code>.</p> <p>To use XML as resource format, <code>MessageConverter</code> for XML, that performs the XXE Injection countermeasure, should be specified. For details on designated methods, refer to “Enabling XXE Injection measures” .</p>
(4)	<p>Settings to enable page search functionality are added.</p> <p>For page search details, refer to “Pagination”.</p> <p>This setting is not required if page search is unnecessary, however, it is alright if defined.</p>
(5)	<p>Bean registration is performed for Spring MVC interceptor.</p> <p>In the above example, only the <code>TraceLoggingInterceptor</code> provided by common library is defined. However, when using JPA as data access, <code>OpenEntityManagerInViewInterceptor</code> setting needs to be added separately.</p> <p>Refer to Database Access (JPA) for <code>OpenEntityManagerInViewInterceptor</code>.</p>
(6)	<p>Application layer components for RESTful Web Service (Controller or Helper class etc.) are scanned and bean registration is performed.</p> <p>The “<code>com.example.project.api</code>” part is the package name for each project.</p>
5.16. RESTful Web Service (7)	<p>AOP definition to output the exception handled by Spring MVC framework to a log, is specified.</p> <p>Refer to Exception Handling for <code>HandlerExceptionResolverLoggingInterceptor</code>.</p>

Servlet settings for RESTful Web Service

When RESTful Web Service and client application are built as separate Web applications, the settings are as follows:

When RESTful Web Service and client application are to be built as same Web application, it is necessary to perform “*Settings when RESTful Web Service and client application are operated as the same Web application*”.

- web.xml

```
<!-- omitted -->

<servlet>
    <!-- (1) -->
    <servlet-name>restAppServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <!-- (2) -->
        <param-value>classpath*:META-INF/spring/spring-mvc-rest.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<!-- (3) -->
<servlet-mapping>
    <servlet-name>restAppServlet</servlet-name>
    <url-pattern>/api/v1/*</url-pattern>
</servlet-mapping>

<!-- omitted -->
```

Sr. No.	Description
(1)	<p>A name which shows that it is a RESTful Web Service servlet, is specified in <servlet-name> element.</p> <p>In the above example, "restAppServlet" is specified as the servlet name.</p>
(2)	<p>A Spring MVC bean definition file used to build DispatcherServlet for RESTful Web Service, is specified.</p> <p>In the above example, META-INF/spring/spring-mvc-rest.xml in class path, is specified as the Spring MVC bean definition file.</p>
(3)	<p>Servlet path pattern to be mapped with the DispatcherServlet of RESTful Web Service, is specified.</p> <p>In the above example, the servlet path under "/api/v1/" is mapped with the DispatcherServlet for RESTful Web Service.</p> <p>Typically, servlet paths like</p> <pre style="margin-left: 40px;">"/api/v1/" "/api/v1/members" "/api/v1/members/xxxxx"</pre> <p>are mapped in the DispatcherServlet("restAppServlet") for RESTful Web Service.</p>

Tip: Value specified in the value attribute of @RequestMapping annotation

For the value to be specified in value attribute of @RequestMapping annotation, value assigned to the part of wild card (*) in <url-pattern> element, is specified.

For example, when @RequestMapping(value = "members") is specified, it is deployed as the method to perform a process for path "/api/v1/members". Therefore, it is not necessary to specify the path ("api/v1") in value attribute of @RequestMapping annotation for mapping to divided servlets.

When @RequestMapping(value = "api/v1/members") is specified, it gets deployed as the method that performs a process for the "/api/v1/api/v1/members" path. Hence, please take note of same.

REST API implementation

How to implement REST API is explained here.

Hereafter, REST API implementation example is explained using member information (Member resource) of a shopping site.

Note: Domain layer implementation is not explained in this section, however, it is sent as attachment “*Source code of the domain layer class created at the time of REST API implementation*”.

Please refer if required.

REST API specifications used in this explanation are as shown below.

Resource format

The resource format of member information is the following JSON format.

In the following example, although all the fields are displayed, they are not used in the requests and responses of all API.

For example, "password" is used only in requests whereas "createdAt" or "lastModifiedAt" are used only in responses.

```
{  
    "memberId" : "M000000001",  
    "firstName" : "Firstname",  
    "lastName" : "Lastname",  
    "genderCode" : "1",  
    "dateOfBirth" : "1977-03-13",  
    "emailAddress" : "user1@test.com",  
    "telephoneNumber" : "09012345678",  
    "zipCode" : "1710051",  
    "address" : "Tokyo",  
    "credential" : {  
        "signId" : "user1@test.com",  
        "password" : "zaq12wsx",  
        "passwordLastChangedAt" : "2014-03-13T04:39:14.831Z",  
        "lastModifiedAt" : "2014-03-13T04:39:14.831Z"  
    }  
}
```

```
},
"createdAt" : "2014-03-13T04:39:14.831Z",
"lastModifiedAt" : "2014-03-13T04:39:14.831Z"
}
```

Note: This section illustrates an example wherein a hypermedia link for related resource is not provided. For details on implementation with hypermedia link, refer to “[Implementing hypermedia link](#)”.

Specifications of resource fields

The specifications for each field of a resource (JSON) are as shown below.

Sr. No.	Item name	Type	I/O specifications	Number of digits (min-max)	Other specifications
(1)	memberId	String	I/O	10-10	It should be "Unspecified" (NULL) at the time of request for POST Members.
(2)	firstName	String	I/O	1-128	-
(3)	lastName	String	I/O	1-128	-
(4)	genderCode	String (Code)	I/O	1-1	"0" : UNKNOWN "1" : MEN "2" : WOMEN
(5)	dateOfBirth	Date	I/O	-	yyyy-MM-dd format (extended ISO-8601 format)
(6)	emailAddress	String (E-mail)	I/O	1-256	-
(7)	telephoneNumber	String	I/O	0-20	-
(8)	zipCode	String	I/O	0-20	-
(9)	address	String	I/O	0-256	-
(10)	credential	Object	I/O	-	It is specified at the time of request for POST Members.
1038	credential/signId	String	I/O	0-256	emailAddress value is applied when not specified.

REST APIs List

APIs given below are used as the REST API to be implemented.

Sr. No.	API name	HTTP Method	Resource path	Status Code	API Overview
(1)	<i>GET Members</i>	GET	/api/v1/members	200 (OK)	Page is searched for Member resource that matches the condition.
(2)	<i>POST Members</i>	POST	/api/v1/members	201 (Created)	One Member resource is created.
(3)	<i>GET Member</i>	GET	/api/v1/members/{memberId}	200 (OK)	One Member resource is fetched.
(4)	<i>PUT Member</i>	PUT	/api/v1/members/{memberId}	200 (OK)	One Member resource is updated.
(5)	<i>DELETE Member</i>	DELETE	/api/v1/members/{memberId}	204 (No Content)	One Member resource is deleted.

Note: This section focuses on the details of CRUD operation for a resource. Hence, HEAD and OPTIONS methods are not explained. To create the RESTful Web Service conforming to HTTP specifications, refer to “[Creating RESTful Web Service conforming to HTTP specifications](#)”.

Creating REST API packages

Package to store REST API class is created.

It is recommended to assign `api` as the package name for the route package that stores REST API class and to create a package for each resource (lower case of resource name) under the same.

Resource name in the explanation is `Member`. Hence, the package name is `org.terasoluna.examples.rest.api.member`.

Note: Usually, following 4 types of classes are stored in the created package. It is recommended to use the following naming rules for name of the class to be created.

- [Resource name]Resource
- [Resource name]RestController
- [Resource name]Validator (created when required)
- [Resource name]Helper (created when required)

In the explanation, name of the resource is `Member`. As a result, the respective names will be as below.

- MemberResource
- MemberRestController
- MemberValidator
- MemberHelper

When handling a related resource, it is advisable to place the class for related resource also in the same package.

It is recommended to create a package named `common` that stores common parts for REST API just under the route package that stores the REST API class and to create sub packages at functionality level.

For example, a sub package that stores common parts which perform error handling is created with the name `error`.

The class for exception handling created in the subsequent explanation, is stored in a package called `org.terasoluna.examples.rest.api.common.error`.

Note: As long as it is clear that the package is storing common parts, it can have a name other than common.

Creating Resource class

In this guideline, it is recommended to provide a Resource class as the class representing the resource published on Web (represent JSON or XML).

Note: Reasons for creating a Resource class

The reason for creating a Resource class regardless of DomainObject class (for example, Entity class) being available is, user interface information (UI) which is used in the I/O with client and information handled by business process do not necessarily match.

If these are mixed and then used, the application layer may affect the domain layer, resulting in deteriorated maintainability. It is recommended to create the DomainObject and Resource class separately and convert data by using BeanMapper like Dozer etc.

Role of Resource class is as follows.

Sr. No.	Roles	Description
(1)	To define the data structure of a resource.	Data structure of the resource published on Web is defined. Generally, it is very rare to publish the data structure managed by persistence layer of database etc. as it is, as a resource on Web.
(2)	To define format.	Definition related to resource format is specified using annotation. Annotation to be used differs according to the resource format (JSON/XML etc.). Jackson annotation is used for JSON format whereas JAXB annotation is used for XML format.
(3)	To define input validation rules.	The input validation rules for single item of each field are specified by using Bean Validation annotation. For input validation details, refer to “ Input Validation ” .

Example of Resource class creation is shown below.

- MemberResource.java

```
package org.terasoluna.examples.rest.api.member;

import java.io.Serializable;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Null;
import javax.validation.constraints.Past;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotEmpty;
import org.joda.time.DateTime;
import org.joda.time.LocalDate;
import org.terasoluna.gfw.common.codelist.ExistInCodeList;
```

```
// (1)
public class MemberResource implements Serializable {

    private static final long serialVersionUID = 1L;

    // (2)
    interface PostMembers {
    }

    interface PutMember {
    }

    @NotNull(groups = PostMembers.class)
    @NotEmpty(groups = PutMember.class)
    @Size(min = 10, max = 10, groups = PutMember.class)
    private String memberId;

    @NotEmpty
    @Size(max = 128)
    private String firstName;

    @NotEmpty
    @Size(max = 128)
    private String lastName;

    @NotEmpty
    @ExistInCodeList(codeListId = "CL_GENDER")
    private String genderCode;

    @NotNull
    @Past
    private LocalDate dateOfBirth;

    @NotEmpty
    @Size(max = 256)
    @Email
    private String emailAddress;

    @Size(max = 20)
    private String telephoneNumber;

    @Size(max = 20)
    private String zipCode;

    @Size(max = 256)
    private String address;

    @NotNull(groups = PostMembers.class)
    @Null(groups = PutMember.class)
    @Valid
    // (3)
```

```
private MemberCredentialResource credential;

@Null
private DateTime createdAt;

@Null
private DateTime lastModifiedAt;

// omitted setter and getter

}
```

Sr. No.	Description
(1)	JavaBean representing the Member resource.
(2)	The interface for specifying validation group of Bean Validation is defined. In this implementation, input validation is grouped, as different input validations are performed for POST and PUT methods. Refer to “ Input Validation ” for grouped validation.
(3)	JavaBean with nested related resource is defined in the field. In this implementation, the member credentials (sign ID and password) are handled as related resources. These are extracted as related resources by considering operations that carry out only the sign ID change and password change. However, REST API implementation for related resources is omitted here.

- MemberCredentialResource.java

```
package org.terasoluna.examples.rest.api.member;

import java.io.Serializable;

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

import org.codehaus.jackson.map.annotate.JsonSerialize;
import org.codehaus.jackson.map.annotate.JsonSerialize.Inclusion;
import org.hibernate.validator.constraints.Email;
import org.joda.time.DateTime;
```

```
// (4)
public class MemberCredentialResource implements Serializable {

    private static final long serialVersionUID = 1L;

    @Size(max = 256)
    @Email
    private String signId;

    // (5)
    @JsonSerialize(include = Inclusion.NON_NULL)
    @NotNull
    @Size(min = 8, max = 32)
    private String password;

    @Null
    private DateTime passwordLastChangedAt;

    @Null
    private DateTime lastModifiedAt;

    // omitted setter and getter
}
```

Sr. No.	Description
(4)	JavaBean that represents Credential resource which is the related resource of Member resource.
(5)	An annotation is specified so that the field itself is not output in JSON when the value is null. It is specified so that the field ‘password’ is not output in responding JSON. In the above example, it is restricted to (<code>Inclusion.NON_NULL</code>) for NULL value, however it can also be specified as (<code>Inclusion.NON_EMPTY</code>) in case of an empty value.

- Adding the mapping definition of Bean

In the subsequent implementations, Entity class and Resource class are copied by using “*Bean Mapping (Dozer)*” .

Joda-Time classes namely, `org.joda.time.DateTime` and `org.joda.time.LocalDate` are included in the JavaBean shown above. However, Joda-Time objects are not correctly copied if “*Bean*

Mapping (Dozer)" is used for copying.

Therefore, it is necessary to apply "[How to copy Joda-Time classes using Dozer](#)" to copy the objects correctly.

Creating Controller class

Controller class is created for each resource.

Refer to [Appendix](#) for the source code when implementation of all APIs is completed.

```
package org.terasoluna.examples.rest.api.member;

// omitted

@RequestMapping("members") // (1)
@Controller
public class MemberRestController {

    // omitted ...

}
```

Sr. No.	Description
(1)	<p>Resource collection URI (Servlet path) is mapped for Controller.</p> <p>Typically, a servlet path indicating a collection of resources is specified in the value attribute of @RequestMapping annotation.</p> <p>In the above example, a servlet path called /api/v1/members is mapped.</p>

Implementing REST API that fetches collection of resources

Implementing the REST API wherein a page search is performed for member resource collection specified by URI.

- Creating the JavaBean for receiving search conditions

When search conditions are necessary to fetch resource collection, a JavaBean is created for receiving the search conditions.

```
// (1)
public class MembersSearchQuery implements Serializable {
    private static final long serialVersionUID = 1L;

    // (2)
    @NotEmpty
    private String name;

    public String getName() {
        return name;
    }

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

Sr. No.	Description
(1)	A JavaBean for receiving search conditions is created. When search conditions are not necessary, JavaBean need not be created.
(2)	Property name is matched with the parameter name of request parameter. In the above example, value "John" is set in the name property of JavaBean for request /api/v1/members?name=John.

- REST API implementation

A process wherein page search is performed for a collection of Member resources is implemented.

```
@RequestMapping("members")
@Controller
public class MemberRestController {

    // omitted

    @Inject
    MemberService memberService;
```

```
@Inject
Mapper beanMapper;

// (3)
@RequestMapping(method = RequestMethod.GET)
// (4)
@ResponseBody
// (5)
@ResponseStatus(HttpStatus.OK)
public Page<MemberResource> getMembers(
    // (6)
    @Validated MembersSearchQuery query,
    // (7)
    Pageable pageable) {

    // (8)
    Page<Member> page = memberService.searchMembers(query.getName(), pageable);

    // (9)
    List<MemberResource> memberResources = new ArrayList<>();
    for (Member member : page.getContent()) {
        memberResources.add(beanMapper.map(member, MemberResource.class));
    }
    Page<MemberResource> responseResource = new PageImpl<>(memberResources,
        pageable, page.getTotalElements());

    // (10)
    return responseResource;
}

// omitted

}
```

Sr. No.	Description
---------	-------------

- (3) RequestMethod.GET is specified in method attribute of @RequestMapping annotation.
- (4) @org.springframework.web.bind.annotation.ResponseBody is assigned as method annotation.
By assigning this annotation, the returned Resource object is marshalled in JSON or XML and set in response BODY.
- (5) @org.springframework.web.bind.annotation.ResponseStatus is assigned as method annotation and the status code returned as response, is specified.
200 (OK) is set in the value attribute of @ResponseStatus annotation.

Tip: How to specify the status code

A fixed status code sent as response is specified in this example using @ResponseStatus annotation. However, it can also be specified in Controller logic.

```
public ResponseEntity<Page<MemberResource>> getMembers (
    @Validated MembersSearchQuery query,
    Pageable pageable) {
    // omitted
    return new ResponseEntity(responseResource, HttpStatus.OK);
}
```

When it is necessary to change the responding status codes based on process details or process results, org.springframework.http.ResponseEntity is used, as shown in the above implementation.

- (6) A JavaBean for receiving search conditions is specified as an argument.
When input validation is necessary, @Validated is assigned as argument annotation. For input validation details, refer to “*Input Validation*”.
- (7) When page search is necessary, org.springframework.data.domain.Pageable is specified as an argument.

- (8) Service method of domain layer is called and resource information (Entity etc.) matching with the condition is fetched.

Response at the time of using PageImpl class is as below.

Highlighted portion shows the fields specific for page search.

```
{
    "content" : [ {
        "memberId" : "M000000001",
        "firstName" : "John",
        "lastName" : "Smith",
        "genderCode" : "1",
        "dateOfBirth" : "1977-03-07",
        "emailAddress" : "john.smith@test.com",
        "telephoneNumber" : "09012345678",
        "zipCode" : "1710051",
        "address" : "Tokyo",
        "credential" : {
            "signId" : "john.smit@test.com",
            "passwordLastChangedAt" : "2014-03-13T10:18:08.003Z",
            "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
        },
        "createdAt" : "2014-03-13T10:18:08.003Z",
        "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
    }, {
        "memberId" : "M000000002",
        "firstName" : "Sophia",
        "lastName" : "Smith",
        "genderCode" : "2",
        "dateOfBirth" : "1977-03-07",
        "emailAddress" : "sophia.smith@test.com",
        "telephoneNumber" : "09012345678",
        "zipCode" : "1710051",
        "address" : "Tokyo",
        "credential" : {
            "signId" : "sophia.smith@test.com",
            "passwordLastChangedAt" : "2014-03-13T10:18:08.003Z",
            "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
        },
        "createdAt" : "2014-03-13T10:18:08.003Z",
        "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
    } ],
    "sort" : [ {
        "direction" : "DESC",
        "property" : "lastModifiedAt",
        "ignoreCase" : false,
        "ascending" : false
    }],
    "lastPage" : false,
    "firstPage" : false,
    "totalElements" : 25,
    "numberOfElements" : 2,
```

```
"totalPages" : 13,  
"size" : 2,  
"number" : 1  
}
```

- Adding Bean mapping definition

In the above implementation, Member object and MemberResource object are copied by using “*Bean Mapping (Dozer)*”.

It is not necessary to add Bean mapping definition when simply a copy of field value can be used.

However, in the above implementation, the setting needs to be such that credential.password is not copied while copying Member object details to MemberResource object.

It is necessary to add Bean mapping definition so that specific fields are not copied.

```
<!-- (11) -->  
<?xml version="1.0" encoding="UTF-8"?>  
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-i  
xsi:schemaLocation="http://dozer.sourceforge.net  
http://dozer.sourceforge.net/schema/beanmapping.xsd">  
  
<mapping type="one-way">  
    <class-a>org.terasoluna.examples.rest.domain.model.MemberCredential</class-a>  
    <class-b>org.terasoluna.examples.rest.api.member.MemberCredentialResource</class-b>  
    <!-- (12) -->  
    <field-exclude>  
        <a>password</a>  
        <b>password</b>  
    </field-exclude>  
</mapping>  
  
</mappings>
```

Sr. No.	Description
(11)	A file that defines mapping rules for Member object and MemberResource object is created. It is recommended to create a mapping definition file of Dozer for each resource. In this implementation, it is stored in <code>/xxx-web/src/main/resources/META-INF/dozer/memberResource-mapping.xml</code> .
(12)	In the above example, password field is not copied while copying the details of MemberCredential which is a related entity of Member, to MemberCredentialResource, a related resource of MemberResource. For Bean mapping definition methods, refer to “ Bean Mapping (Dozer) ” .

- Request example

```
GET /rest-api-web/api/v1/members?name=Smith&page=0&size=2 HTTP/1.1
Accept: text/plain, application/json, application/*+json, /*
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
```

- Response Example

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: fb63a6d446f849feb8ccaa4c9a794333
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 13 Mar 2014 11:10:43 GMT
```

```
{"content": [{"memberId": "M000000001", "firstName": "John", "lastName": "Smith", "genderCode": "1",
```

Tip: When page search is not necessary, Resource class list may be handled directly.

Following is the definition of the Controller method used when handling the list of Resource class directly.

```
@RequestMapping(method = RequestMethod.GET)
@ResponseBody
@ResponseStatus(HttpStatus.OK)
public List<MemberResource> getMembers(
    @Validated MembersSearchQuery query) {
    // omitted
}
```

JSON is as follows when list of Resource class is directly handled.

```
[ {
    "memberId" : "M000000001",
    "firstName" : "John",
    "lastName" : "Smith",
    "genderCode" : "1",
    "dateOfBirth" : "1977-03-07",
    "emailAddress" : "john.smith@test.com",
    "telephoneNumber" : "09012345678",
    "zipCode" : "1710051",
    "address" : "Tokyo",
    "credential" : {
        "signId" : "john.smith@test.com",
        "passwordLastChangedAt" : "2014-03-13T10:18:08.003Z",
        "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
    },
    "createdAt" : "2014-03-13T10:18:08.003Z",
    "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
}, {
    "memberId" : "M000000002",
    "firstName" : "Sophia",
    "lastName" : "Smith",
    "genderCode" : "2",
    "dateOfBirth" : "1977-03-07",
    "emailAddress" : "sophia.smith@test.com",
    "telephoneNumber" : "09012345678",
    "zipCode" : "1710051",
    "address" : "Tokyo",
    "credential" : {
        "signId" : "sophia.smith@test.com",
        "passwordLastChangedAt" : "2014-03-13T10:18:08.003Z",
        "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
    },
    "createdAt" : "2014-03-13T10:18:08.003Z",
    "lastModifiedAt" : "2014-03-13T10:18:08.003Z"
} ]
```

Implementing REST API that adds a resource to collection

Example of implementation of REST API wherein a specified Member resource is created and added to the collection is given below.

- REST API implementation

Process that creates specified Member resource and adds it to the collection, is implemented.

```
@RequestMapping("members")
@Controller
public class MemberRestController {

    // omitted

    // (1)
    @RequestMapping(method = RequestMethod.POST)
    @ResponseBody
    // (2)
    @ResponseStatus(HttpStatus.CREATED)
    public MemberResource postMember(
        // (3)
        @RequestBody @Validated({ PostMembers.class, Default.class })
        MemberResource requestedResource) {

        // (4)
        Member inputMember = beanMapper.map(requestedResource, Member.class);
        Member createdMember = memberService.createMember(inputMember);

        MemberResource responseResource = beanMapper.map(createdMember,
            MemberResource.class);

        return responseResource;
    }

    // omitted
}
```

Sr. No.	Description
(1)	RequestMethod.POST is specified in the method attribute of @RequestMapping annotation.
(2)	@ResponseStatus annotation is assigned as method annotation and responding status code is specified. 201(Created) is set in the value attribute of @ResponseStatus annotation.
(3)	JavaBean (Resource class) that receives information of newly created resource, is specified as an argument. @org.springframework.web.bind.annotation.RequestBody is assigned as argument annotation. By assigning @RequestBody annotation, JSON or XML data set in request Body is unmarshalled in Resource object. @Validated annotation is assigned as argument annotation to enable input validation. For details on input validation, refer to “ <i>Input Validation</i> ” .
(3)	Service method of domain layer is called and a new resource is created. For domain layer implementation, refer to “ <i>Domain Layer Implementation</i> ” .

- Request example

```
POST /rest-api-web/api/v1/members HTTP/1.1
Accept: text/plain, application/json, application/*+json, */*
Content-Type: application/json; charset=UTF-8
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
Content-Length: 248

{"firstName":"John", "lastName": "Smith", "genderCode": "1", "dateOfBirth": "2013-03-13", "emailAdd...
```

- Response example

HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
X-Track: c7e9c8a9aa4f40ff87f3acdb77baccdf
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 13 Mar 2014 10:58:26 GMT

```
{"memberId": "M000000023", "firstName": "John", "lastName": "Smith", "genderCode": "1", "dateOfBirth": "1985-04-15", "emailAddress": "john.smith@example.com", "mobileNumber": "+1234567890", "homeAddress": "123 Main Street, Anytown, USA", "workAddress": "123 Office Park, Anytown, USA", "spouseName": "Jane Doe", "children": [{"name": "Sarah", "age": 10}, {"name": "Michael", "age": 8}], "employment": {"company": "TechCorp", "position": "Software Engineer", "startYear": 2010, "endYear": null}, {"education": [{"degree": "Bachelor's in Computer Science", "year": 2005, "institution": "University of Anywhere"}, {"degree": "Master's in Data Science", "year": 2010, "institution": "Stanford University"}]}, {"hobbies": ["Gardening", "Cooking", "Reading", "Photography"]}]
```

Implementing REST API that fetches specified resource

Implementation of REST API that fetches the Member resource specified by URI, is shown below.

- REST API implementation

Process that fetches the Member resource specified by URI is implemented.

```
@RequestMapping("members")
@Controller
public class MemberRestController {
    // omitted

    // (1)
    @RequestMapping(value = "{memberId}", method = RequestMethod.GET)
    @ResponseBody
    // (2)
    @ResponseStatus(HttpStatus.OK)
    public MemberResource getMember(
        // (3)
        @PathVariable("memberId") String memberId) {

        // (4)
        Member member = memberService.getMember(memberId);

        MemberResource responseResource = beanMapper.map(member,
            MemberResource.class);

        return responseResource;
    }
}
```

```
// omitted
```

```
}
```

Sr. No.	Description
(1)	Path variable (<code>{memberId}</code> in the example above) is specified in value attribute whereas <code>RequestMethod.GET</code> is specified in method attribute of <code>@RequestMapping</code> annotation. A value that uniquely identifies the resource is specified in <code>{memberId}</code> .
(2)	<code>@ResponseStatus</code> annotation is assigned as method annotation and responding status code is specified. <code>200 (OK)</code> is set in value attribute of <code>@ResponseStatus</code> annotation.
(3)	Value that uniquely identifies the resource is fetched from path variable. Value specified in path variable (<code>{memberId}</code>) can be received as method argument by specifying <code>@PathVariable ("memberId")</code> as argument annotation. For details on path variable, refer to " "Retrieving values from URL path" ". In the above example, when URI is <code>/api/v1/members/M12345</code> , "M12345" is stored in <code>memberId</code> of argument.
(4)	Service method of domain layer is called and resource information (Entity etc.) that matches with the ID fetched from path variable is acquired. For domain layer implementation, refer to " "Domain Layer Implementation" ".

- Request example

```
GET /rest-api-web/api/v1/members/M000000003 HTTP/1.1
Accept: text/plain, application/json, application/*+json, /*
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
```

- Response Example

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: 275b4e7a61f946eea47672f272315ac2
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 13 Mar 2014 11:25:13 GMT

```
{"memberId": "M000000003", "firstName": "John", "lastName": "Smith", "genderCode": "L", "dateOfBirth": "1985-04-15", "age": 34, "isEligible": true}
```

Implementing REST API that updates specified resource

Implementation of REST API that updates the Member resource specified in URI, is shown below.

- REST API implementation

Process that updates the Member resource specified in URI, is implemented.

```
@RequestMapping("members")
@Controller
public class MemberRestController {

    // omitted

    // (1)
    @RequestMapping(value = "{memberId}", method = RequestMethod.PUT)
    @ResponseBody
    // (2)
    @ResponseStatus(HttpStatus.OK)
    public MemberResource putMember(
        @PathVariable("memberId") String memberId,
        // (3)
        @RequestBody @Validated({ PutMember.class, Default.class })
        MemberResource requestedResource) {

        // (4)
        Member inputMember = beanMapper.map(
            requestedResource, Member.class);
        Member updatedMember = memberSevice.updateMember(
            memberId, inputMember);

        MemberResource responseResource = beanMapper.map(updatedMember,
```

```

        MemberResource.class);

    return responseResource;
}

// omitted

}

```

Sr. No.	Description
(1)	<p>Path variable ({memberId} in the example above) is specified in value attribute whereas RequestMethod.PUT is specified in “method” attribute of @RequestMapping annotation.</p> <p>Value that uniquely identifies the resource is specified in {memberId}.</p>
(2)	<p>@ResponseStatusannotation is assigned as method annotation and responding status code is specified.</p> <p>200 (OK) is specified in value attribute of @ResponseStatus annotation.</p>
(3)	<p>JavaBean (Resource class) for receiving the details of resource update is specified as an argument.</p> <p>By assigning @RequestBody annotation as argument annotation, JSON or XML data set in request Body is unmarshalled in Resource object.</p> <p>@Validated annotation is assigned as argument annotation to enable input validation.</p> <p>For details on input validation, refer to “Input Validation” .</p>
(4)	<p>Service method of domain layer is called and resource information (Entity etc.) matching with the ID fetched from path variable, is updated.</p> <p>For domain layer implementation, refer to “Domain Layer Implementation” .</p>

- Request example

```
PUT /rest-api-web/api/v1/members/M000000004 HTTP/1.1
Accept: text/plain, application/json, application/*+json, */
Content-Type: application/json; charset=UTF-8
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
Content-Length: 221

{"memberId":"M000000004", "firstName":"John", "lastName":"Smith", "genderCode":"1", "dateOfBirth
```

- Response example

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: 5e8fea3aae044e94bf20a293e155af57
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 13 Mar 2014 11:35:59 GMT

{"memberId":"M000000004", "firstName":"John", "lastName":"Smith", "genderCode":"1", "dateOfBirth
```

Implementing REST API that deletes specified resource

Implementation of REST API that deletes the Member resource specified by URI is as follows:

- REST API implementation

Process that deletes the Member resource specified by URI is implemented.

```
@RequestMapping("members")
@Controller
public class MemberRestController {

    // omitted

    // (1)
    @RequestMapping(value = "{memberId}", method = RequestMethod.DELETE)
```

```
@ResponseBody
// (2)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteMember(
    @PathVariable("memberId") String memberId) {

    // (3)
    memberService.deleteMember(memberId);

}

// omitted

}
```

Sr. No.	Description
(1)	Path variable ({memberId} in the example above) is specified in value attribute and RequestMethod.DELETE is specified in method attribute of @RequestMapping annotation.
(2)	@ResponseStatusannotation is assigned as method annotation and responding status code is specified. 204 (NO_CONTENT) is set in value attribute of @ResponseStatus annotation.
(3)	Service method of domain layer is called and resource information (Entity etc.) matching with the ID fetched from path variable, is deleted. For domain layer implementation, refer to “ <i>Domain Layer Implementation</i> ”.

Note: To set deleted resource information in response BODY, (200) OK is set in the status code.

- Request example

```
DELETE /rest-api-web/api/v1/members/M000000005 HTTP/1.1
Accept: text/plain, application/json, application/*+json, /*
User-Agent: Java/1.7.0_51
Host: localhost:8080
```

Connection: keep-alive

- Response example

```
HTTP/1.1 204 No Content
Server: Apache-Coyote/1.1
X-Track: e06c5bd40c864a299c48d9be3f12b2c0
Date: Thu, 13 Mar 2014 11:40:05 GMT
```

Implementing exception handling

How to handle the exceptions occurring in RESTful Web Service is explained below.

RESTful Web Service oriented generic exception handling feature is not provided in Spring MVC.

Alternately,

(org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler) is provided as the class that assists in implementing exception handling for RESTful Web Service.

This guideline recommends a common exception handling method, by creating an exception handling class that inherits the Spring MVC class and assigning @ControllerAdvice annotation to this exception handling class.

Method that handles exceptions in Spring MVC framework is implemented in advance in ResponseEntityExceptionHandler by using @ExceptionHandler annotation.

Therefore, individual handling of the exceptions in Spring MVC framework need not be implemented.

Further, HTTP status codes corresponding to the exceptions handled by ResponseEntityExceptionHandler are set by the same specifications as DefaultHandlerExceptionResolver.

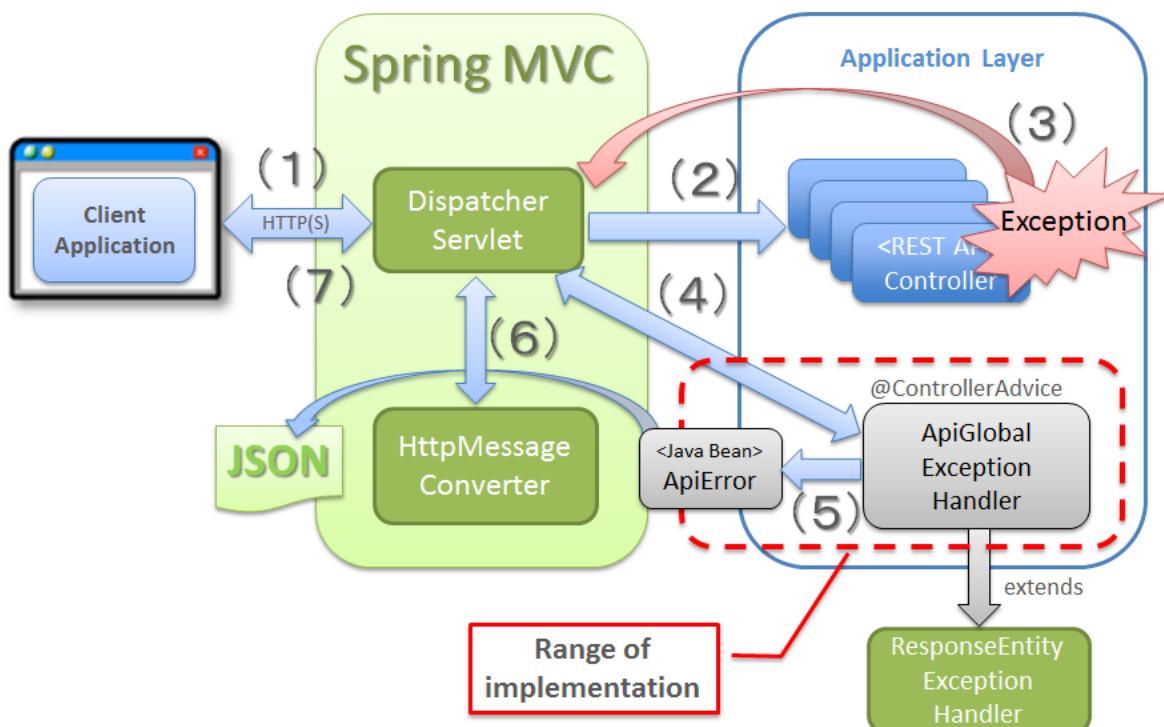
For handled exceptions and HTTP status codes thus set, refer to “*HTTP response code set by DefaultHandlerExceptionResolver*”.

Although an empty response BODY is returned by default implementation of

ResponseEntityExceptionHandler, it can be extended so as to output error information in the response Body.

This guideline recommends to output appropriate error information in the response Body.

Process flow wherein, exception handling class that inherits ResponseEntityExceptionHandler is created and common exception handling is performed, is described before explaining the typical implementation. Please note that, separate implementation is required for the part marked in red frame.



Sr. No.	Processing layer	Description
(1)	Spring MVC	Spring MVC receives a request from client and calls REST API.
(2)	(Framework)	
(3)		An exception occurs during REST API process. The exception occurred is caught by Spring MVC.
(4)		Spring MVC delegates the process to exception handling class.
(5)	Custom Exception Handler (Common Component)	An error object that retains error information is generated in the exception handling class and returned to Spring MVC.
(6)	Spring MVC (Framework)	Spring MVC converts the error object to JSON format message using <code>HttpMessageConverter</code> .
(7)		Spring MVC sets the JSON format error message in response BODY and sends response to the client.

Implementation to output error information in response Body

- Error information is in the following JSON format.

```
{  
  "code" : "e.ex.fw.7001",  
  "message" : "Validation error occurred on item in the request body.",  
  "details" : [ {  
    "code" : "ExistInCodeList",
```

```
        "message" : "\"genderCode\" must exist in code list of CL_GENDER.",
        "target" : "genderCode"
    }
}
```

- JavaBean that retains error information is created.

```
package org.terasoluna.examples.rest.api.common.error;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.codehaus.jackson.map.annotate.JsonSerialize;
import org.codehaus.jackson.map.annotate.JsonSerialize.Inclusion;

// (1)
public class ApiError implements Serializable {

    private static final long serialVersionUID = 1L;

    private final String code;

    private final String message;

    @JsonSerialize(include = Inclusion.NON_EMPTY)
    private final String target; // (2)

    @JsonSerialize(include = Inclusion.NON_EMPTY)
    private final List<ApiError> details = new ArrayList<>(); // (3)

    public ApiError(String code, String message) {
        this(code, message, null);
    }

    public ApiError(String code, String message, String target) {
        this.code = code;
        this.message = message;
        this.target = target;
    }

    public String getCode() {
        return code;
    }

    public String getMessage() {
```

```
        return message;
    }

    public String getTarget() {
        return target;
    }

    public List<ApiError> getDetails() {
        return details;
    }

    public void addDetail(ApiError detail) {
        details.add(detail);
    }

}
```

Sr. No.	Description
(1)	<p>Class is created for retaining error information.</p> <p>In the above example, it is the class that retains lists of error codes, error messages, error targets and detailed error information.</p>
(2)	<p>Field that retains the value for identifying the target where error has occurred.</p> <p>When an error occurs in input validation, there are cases where the value that identifies the field where error has occurred needs to be returned to the client.</p> <p>In such cases, it is necessary to set a field that retains the field name where error has occurred.</p>
(3)	<p>Field that retains the list of detailed error information.</p> <p>When an error occurs in input validation, it is necessary to return all the error information to the client as there are cases with multiple error causes.</p> <p>In such cases, a field that lists and retains detailed error information is necessary.</p>

Tip: When the value is null or empty, it is possible to avoid fields being output to JSON by specifying @JsonSerialize(include = Inclusion.NON_EMPTY) in field. When the condition to disable field output is to be restricted to null, it is advisable to specify @JsonSerialize(include = Inclusion.NON_NULL).

- A class for generating JavaBean that retains error information is created.

Refer to [Appendix](#) for the source code when implementation of all exception handling is completed.

```
// (4)
@Component
public class ApiErrorCreator {

    @Inject
    MessageSource messageSource;

    public ApiError createApiError(WebRequest request, String errorCode,
        String defaultMessage, Object... arguments) {
        // (5)
        String localizedMessage = messageSource.getMessage(errorCode,
            arguments, defaultMessage, request.getLocale());
        return new ApiError(errorCode, localizedMessage);
    }

    // omitted
}
```

Sr. No.	Description
(4)	If needed, class that provides the method for generating error information is created. Creating this class is not mandatory. However, it is recommended to create it so as to clearly define the role division.
(5)	Error message is fetched by <code>MessageSource</code> . For message management methods, refer to “ Message Management ”.

Tip: In the above example, `org.springframework.web.context.request.WebRequest` is received as an argument to support localization of messages. `WebRequest` is not necessary when message localization is not required.

The reason for using `WebRequest` as an argument instead of `java.util.Locale` is due to an additional requirement wherein, HTTP request details are to be embedded in the error message. When there is no such requirement to embed HTTP request details in error, `Locale` can also be used.

- `ResponseEntityExceptionHandler` method is extended and the implementation to output error information in response Body is carried out.

Refer to [Appendix](#) for the source code when implementation for all exception handling is completed.

```
@ControllerAdvice // (6)
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @Inject
    ApiErrorCreator apiErrorCreator;

    @Inject
    ExceptionCodeResolver exceptionCodeResolver;

    // (7)
    @Override
    protected ResponseEntity<Object> handleExceptionInternal(Exception ex,
        Object body, HttpHeaders headers, HttpStatus status,
        WebRequest request) {
        final Object apiError;
        // (8)
        if (body == null) {
            String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
            apiError = apiErrorCreator.createApiError(request, errorCode, ex
                .getLocalizedMessage());
        } else {
            apiError = body;
        }
        // (9)
        return new ResponseEntity<>(apiError, headers, status);
    }

    // omitted
}
```

Sr. No.	Description
(6)	A class that inherits <code>ResponseEntityExceptionHandler</code> provided by Spring MVC is created and <code>@ControllerAdvice</code> annotation is assigned.
(7)	<code>handleExceptionInternal</code> method of <code>ResponseEntityExceptionHandler</code> is overridden.
(8)	<p>When the JavaBean output to response Body is not specified, a JavaBean object that retains error information is generated.</p> <p>In the above example, the exception class changes the error code by using <code>ExceptionCodeResolver</code> provided by common library.</p> <p>For setting example of <code>ExceptionCodeResolver</code>, refer to “<i>Resolving error codes and messages using ExceptionCodeResolver</i>” .</p> <p>When the JavaBean output to response Body is specified, the specified JavaBean is used as it is. This process is implemented considering that error information is generated individually in the error handling process for each exception.</p>
(9)	<p>Error information generated in (8) is set in the ‘Body’ of HTTP Entity for response and then returned.</p> <p>Error information thus returned is converted to JSON using framework and sent as a response.</p> <p>Appropriate values are set in the status code by <code>ResponseEntityExceptionHandler</code> provided by Spring MVC.</p> <p>Refer to “<i>HTTP response code set by DefaultHandlerExceptionResolver</i>” for status codes that are set.</p>

- Response example

```
HTTP/1.1 400 Bad Request
Server: Apache-Coyote/1.1
X-Track: e60b3b6468194e22852c8bfc7618e625
Content-Type: application/json; charset=UTF-8
```

```
Transfer-Encoding: chunked
Date: Thu, 13 Mar 2014 12:16:55 GMT
Connection: close
```

```
{"code":"e.ex.fw.7001","message":"Validation error occurred on item in the request body.", "d
```

Implementing input error exception handling

Implementation for responding to input errors (syntax error, unit item check error, correlated field check error) is explained here.

Following three exceptions need to be handled in order to respond to input errors.

Sr. No.	Exception	Description
(1)	org.springframework.web.bind. MethodArgumentNotValidException	This exception occurs when there is an error during the input validation for JSON or XML specified in request BODY. Basically, it occurs when an invalid value is entered in the resource that is specified when POST or PUT method is performed for the resource.
(2)	org.springframework.validation. BindException	This exception occurs when there is an error during the input validation for request parameter (key=query string of value format). Basically, it occurs when an invalid value is entered in the search conditions specified at the time of GET method of resource collection.
(3)	org.springframework.http.converter. HttpMessageNotReadableException	This exception occurs when there is an error while generating Resource object from JSON or XML. Basically, it occurs in cases such as invalid JSON or XML syntax or violation of schema definition.

Note: org.springframework.beans.TypeMismatchException occurs when there is a type

conversion error for value while fetching values from request parameter, request header and path variable, by using annotation provided by Spring Framework.

When following annotations are specified as arguments of Controller processing method (argument other than `String`), `TypeMismatchException` may occur.

- `@org.springframework.web.bind.annotation.RequestParam`
- `@org.springframework.web.bind.annotation.RequestHeader`
- `@org.springframework.web.bind.annotation.Pathvariable`
- `@org.springframework.web.bind.annotation.MatrixVariable`

`TypeMismatchException` is handled by `ResponseEntityExceptionHandler` resulting in 400 (Bad Request). As a result, individual handling is not required.

Refer to “[Resolving error codes and messages using `ExceptionCodeResolver`](#)” in order to resolve the error codes and error messages to be set in error information.

- Method is created to generate error information for input validation errors.

```
@Component
public class ApiErrorCreator {

    @Inject
    MessageSource messageSource;

    // omitted

    // (1)
    public ApiError createBindingResultApiError(WebRequest request,
                                                String errorCode, BindingResult bindingResult,
                                                String defaultMessage) {
        ApiError apiError = createApiError(request, errorCode,
                                            defaultMessage);
        for (FieldError fieldError : bindingResult.getFieldErrors()) {
            apiError.addDetail(createApiError(request, fieldError, fieldError
                .getField()));
        }
        for (ObjectError objectError : bindingResult.getGlobalErrors()) {
            apiError.addDetail(createApiError(request, objectError, objectError
                .getObjectName()));
        }
        return apiError;
    }
}
```

```
// (2)
private ApiError createApiError(WebRequest request,
    DefaultMessageSourceResolvable messageResolvable, String target) {
    String localizedMessage = messageSource.getMessage(messageResolvable,
        request.getLocale());
    return new ApiError(messageResolvable.getCode(), localizedMessage, target);
}

// omitted

}
```

Sr. No.	Description
(1)	A method is created to generate error information for input validation. In the above example, single field check error (<code>FieldError</code>) and correlated field check error (<code>ObjectError</code>) are added to detailed error information. This method need not be provided when it is not necessary to output error information for each item.
(2)	A common method is created since same process is implemented for single field check error (<code>FieldError</code>) and correlation check error (<code>ObjectError</code>).

- `ResponseEntityExceptionHandler` method is extended and the implementation to output input validation error information in response Body, is performed.

```
@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @Inject
    ApiErrorCreator apiErrorCreator;

    @Inject
    ExceptionCodeResolver exceptionCodeResolver;

    // omitted

    // (3)
    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
```

```
MethodArgumentNotValidException ex, HttpHeaders headers,
HttpStatus status, WebRequest request) {
    return handleBindingResult(ex, ex.getBindingResult(), headers, status,
        request);
}

// (4)
@Override
protected ResponseEntity<Object> handleBindException(BindException ex,
    HttpHeaders headers, HttpStatus status, WebRequest request) {
    return handleBindingResult(ex, ex.getBindingResult(), headers, status,
        request);
}

// (5)
@Override
protected ResponseEntity<Object> handleHttpMessageNotReadable(
    HttpResponseMessageNotFoundException ex, HttpHeaders headers,
    HttpStatus status, WebRequest request) {
    if (ex.getCause() instanceof Exception) {
        return handleExceptionInternal((Exception) ex.getCause(), null,
            headers, status, request);
    } else {
        return handleExceptionInternal(ex, null, headers, status, request);
    }
}

// omitted

// (6)
protected ResponseEntity<Object> handleBindingResult(Exception ex,
    BindingResult bindingResult, HttpHeaders headers,
    HttpStatus status, WebRequest request) {
    String code = exceptionCodeResolver.resolveExceptionCode(ex);
    String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
    ApiError apiError = apiErrorCreator.createBindingResultApiError(
        request, errorCode, bindingResult, ex.getMessage());
    return handleExceptionInternal(ex, apiError, headers, status, request);
}

// omitted
}
```

Sr. No.	Description
(3)	<p>handleMethodArgumentNotValid method of ResponseEntityExceptionHandler is overridden and error handling for MethodArgumentNotValidException is extended.</p> <p>In the above example, the process is delegated to a common method (6) that handles input validation errors.</p> <p>When it is not necessary to output error information for each item, overriding is not required.</p> <p>400 (Bad Request) is set in the status code and presence of some flaw in the field value of the specified resource is notified.</p>
(4)	<p>handleBindException method of ResponseEntityExceptionHandler is overridden and error handling for BindException is extended.</p> <p>In the above example, the process is delegated to a common method (6) that handles input validation error.</p> <p>When it is not necessary to output error information for each item, overriding is not required.</p> <p>400 (Bad Request) is set in the status code and presence of flaw in the specified request parameter is notified.</p>
(5)	<p>handleHttpMessageNotReadable method of ResponseEntityExceptionHandler is overridden and error handling for HttpMessageNotReadableException is extended.</p> <p>In the above example, detailed error handling is performed by using cause exception.</p> <p>If a detailed error handling is not necessary, overriding is not required.</p> <p>400 (Bad Request) is set in the status code and presence of flaw in the specified resource format etc. is notified</p>
(6)	<p>A JavaBean object that retains error information for input validation error is generated.</p> <p>In the above example, a common method is created as same process is implemented in handleMethodArgumentNotValid and handleBindException.</p>

Tip: Error handling when using JSON

When JSON is used as the resource format, following exception is stored as cause exception of

HttpMessageNotReadableException.

Sr. No.	Exception class	Description
(1)	org.codehaus.jackson.JsonParseException	It occurs when an invalid syntax is included for JSON.
(2)	org.codehaus.jackson.map.exc.UnrecognizedPropertyException	It occurs when a field which does not exist in Resource object is specified in JSON.
(3)	org.codehaus.jackson.map.JsonMappingException	It occurs when a value type conversion error occurs while converting from JSON to Resource object.

- Following error response is sent when an input validation error (single field check error, correlated field check error) occurs.

```
HTTP/1.1 400 Bad Request
Server: Apache-Coyote/1.1
X-Track: 13522b3badf2432ba4cad0dc7aeae80
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 05:08:28 GMT
Connection: close
```

```
{"code":"e.ex.fw.7002","message":"Validation error occurred on item in the request parameter"}
```

- Following error response is sent when JSON errors (format error etc.) occur.

```
HTTP/1.1 400 Bad Request
Server: Apache-Coyote/1.1
X-Track: ca4c742a6bfd49e5bc01cd0b124738a1
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
```

```
Date: Wed, 19 Feb 2014 13:32:24 GMT
Connection: close
```

```
{"code": "e.ex.fw.7003", "message": "Request body format error occurred."}
```

Implementing exception handling for “Resource not found” error

When a resource does not exist, implementation for responding to the “resource not found” error, is explained below.

When a resource matching with the ID fetched from path variable is not found, an exception notifying “resource not found” is generated.

`org.terasoluna.gfw.common.exception.ResourceNotFoundException` is provided by common library as an exception notifying “resource not found”.

Implementation is as given below.

- When a resource matching with the ID fetched from path variable is not found, `ResourceNotFoundException` is generated.

```
public Member getMember(String memberId) {
    Member member = memberRepository.findOne(memberId);
    if (member == null) {
        throw new ResourceNotFoundException(ResultMessages.error().add(
            "e.ex.mm.5001", memberId));
    }
    return member;
}
```

- A method is created to generate error information for `ResultMessages`.

```

@Component
public class ApiErrorCreator {

    // omitted

    // (1)
    public ApiError createResultMessagesApiError(WebRequest request,
                                                String rootErrorCode, ResultMessages resultMessages,
                                                String defaultErrorMessage) {
        ApiError apiError;
        if (resultMessages.getList().size() == 1) {
            ResultMessage resultMessage = resultMessages.iterator().next();
            String errorCode = resultMessage.getCode();
            String errorText = resultMessage.getText();
            if (errorCode == null && errorText == null) {
                errorCode = rootErrorCode;
            }
            apiError = createApiError(request, errorCode, errorText,
                                      resultMessage.getArgs());
        } else {
            apiError = createApiError(request, rootErrorCode,
                                      defaultErrorMessage);
            for (ResultMessage resultMessage : resultMessages.getList()) {
                apiError.addDetail(createApiError(request, resultMessage
                    .getCode(), resultMessage.getText(), resultMessage
                    .getArgs()));
            }
        }
        return apiError;
    }

    // omitted
}

```

Sr. No.	Description
(1)	A method for generating error information from process results is created. In the above example, the message information retained by ResultMessages is set in error information.

Note: In the above example, as ResultMessages can retain multiple messages, the process is divided as per when a single message is stored and when multiple messages are stored.

When it is not necessary to support multiple messages, the process wherein the message at the start is generated as error information, may be implemented.

- A method for handling the exception that notifies “resource not found” error is created, in the class that performs error handling.

```
@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @Inject
    ApiErrorCreator apiErrorCreator;

    @Inject
    ExceptionCodeResolver exceptionCodeResolver;

    // omitted

    // (2)
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Object> handleResourceNotFoundException(
        ResourceNotFoundException ex, WebRequest request) {
        return handleResultMessagesNotificationException(ex, null,
            HttpStatus.NOT_FOUND, request);
    }

    // omitted

    // (3)
    private ResponseEntity<Object> handleResultMessagesNotificationException(
        ResultMessagesNotificationException ex, HttpHeaders headers,
        HttpStatus status, WebRequest request) {
        String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
        ApiError apiError = apiErrorCreator.createResultMessagesApiError(
            request, errorCode, ex.getResultMessages(), ex.getMessage());
        return handleExceptionInternal(ex, apiError, headers, status, request);
    }

    // omitted
}
```

Sr. No.	Description
(2)	<p>A method for handling <code>ResourceNotFoundException</code> is added. <code>ResourceNotFoundException</code> exception can be handled if <code>@ExceptionHandler(ResourceNotFoundException.class)</code> is specified as method annotation.</p> <p>In the above example, the process is delegated to method that handles exception of the parent class (<code>ResultMessagesNotificationException</code>) of <code>ResourceNotFoundException</code>.</p> <p>404 (Not Found) is set in the status code and a message stating, ‘specified resource does not exist in the server’ is notified.</p>
(3)	<p>A JavaBean object that retains error information for “resource not found” error and business error is generated.</p> <p>In the above example, a common method is created since the process is same as that for error handling of business error discussed hereafter.</p>

- When resource is not found, following error response is generated.

```

HTTP/1.1 404 Not Found
Server: Apache-Coyote/1.1
X-Track: 5ee563877f3140fd904d0acf52eba398
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 08:46:18 GMT

```

```
{"code": "e.ex.mm.5001", "message": "Specified member not found. member id : M000000001"}
```

Implementing exception handling for business errors

An implementation wherein business error is sent as a response on detecting violation of business rule, is explained here.

Business rule check is performed as Service process and business exception is generated when a business rule violation is detected. For the details on detecting business errors, refer to “[Notifying business error](#)”.

- A method to handle business exception is created in the class that performs error handling.

```
@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    // omitted

    // (1)
    @ExceptionHandler(BusinessException.class)
    public ResponseEntity<Object> handleBusinessException(BusinessException ex,
        WebRequest request) {
        return handleResultMessagesNotificationException(ex, null,
            HttpStatus.CONFLICT, request);
    }

    // omitted
}
```

Sr. No.	Description
(1)	<p>A method for handling <code>BusinessException</code> is added.</p> <p><code>BusinessException</code> can be handled if <code>@ExceptionHandler(BusinessException.class)</code> is specified as method annotation.</p> <p>In the above example, the process is delegated to the method that handles the exception of parent class (<code>ResultMessagesNotificationException</code>) of <code>BusinessException</code>.</p> <p>409 (Conflict) is set in the status code and a message stating that although there are no errors in the resource itself specified by client, all the conditions necessary for operating the resource stored by the server are not in place.</p>

- Following error response is generated when a business error occurs.

```
HTTP/1.1 409 Conflict
Server: Apache-Coyote/1.1
X-Track: 37c1a899d5f74e7a9c24662292837ef7
Content-Type: application/json;charset=UTF-8
```

```
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 09:03:26 GMT
```

```
{"code": "e.ex.mm.8001", "message": "Cannot use specified sign id. sign id : user1@test.com"}
```

Implementing exception handling for exclusive errors

An implementation is explained here wherein, an exclusive error is generated and sent as a response.

Exclusive error handling is necessary when performing exclusive control.

For details of exclusive control, refer to “[Exclusive Control](#)” .

- A method for exclusive error handling is created in the class that performs error handling.

```
@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    // omitted

    // (1)
    @ExceptionHandler({ OptimisticLockingFailureException.class,
        PessimisticLockingFailureException.class })
    public ResponseEntity<Object> handleLockingFailureException(Exception ex,
        WebRequest request) {
        return handleExceptionInternal(ex, null, null,
            HttpStatus.CONFLICT, request);
    }

    // omitted

}
```

Sr. No.	Description
(1)	<p>A method for handling exclusive errors (<code>OptimisticLockingFailureException</code> and <code>PessimisticLockingFailureException</code>) is added.</p> <p>If <code>@ExceptionHandler({ OptimisticLockingFailureException.class, PessimisticLockingFailureException.class })</code> is specified as method annotation, exception handling of exclusive errors (<code>OptimisticLockingFailureException</code> and <code>PessimisticLockingFailureException</code>) can be performed.</p> <p>409(Conflict) is set in status code and a message stating that, ‘although there are no flaws in the resource itself specified by client, the conditions for operating the resource could not be fulfilled due to conflict in the process’ is notified.</p>

- When an exclusive error occurs, following error response is generated.

```
HTTP/1.1 409 Conflict
Server: Apache-Coyote/1.1
X-Track: 85200b5a51be42b29840e482ee35087f
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 16:32:45 GMT

{"code":"e.ex.fw.8002","message":"Conflict with other processing occurred."}
```

Implementing exception handling for system errors

An implementation wherein system error is sent as a response on detecting system abnormality, is explained here.

System exception is generated when any system abnormality is detected. Refer to “[Notifying system error](#)” for the details on how to detect system errors.

- A method to handle system exceptions is created in the class that performs error handling.

```
@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler {
```

```
// omitted

// (1)
@ExceptionHandler(Exception.class)
public ResponseEntity<Object> handleSystemError(Exception ex,
    WebRequest request) {
    return handleExceptionInternal(ex, null, null,
        HttpStatus.INTERNAL_SERVER_ERROR, request);
}

// omitted

}
```

Sr. No.	Description
(1)	<p>A method for handling Exception is added.</p> <p>Exception can be handled if @ExceptionHandler(Exception.class) is specified as method annotation.</p> <p>In the above example, the system exceptions generated from dependent libraries being used, are also handled.</p> <p>500 (Internal Server Error) is set in the status code.</p>

- Following error response is generated when a system error occurs.

```
HTTP/1.1 500 Internal Server Error
Server: Apache-Coyote/1.1
X-Track: 3625d5a040a744e49b0a9b3763a24e9c
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 12:22:33 GMT
Connection: close

{"code":"e.ex.fw.9003","message":"System error occurred."}
```

Warning: Error message at the time of system error

When system error occurs, it is recommended that a simple error message that does not specify the error cause, is set as the message to be returned to the client. If an error message specifying the cause of error is set, it may expose system vulnerabilities to the client resulting in security issues.

Cause of error is output to a log, for error analysis. This log is output by `ExceptionLogger` provided by common library as the default setting of Blank project. As a result, the settings and implementation for log output are not required.

Resolving error codes and messages using `ExceptionCodeResolver`

If `ExceptionCodeResolver` provided by common library is used, error codes can be resolved from the exception class.

Especially, this functionality proves to be convenient, when the cause of error lies at client side and when it is necessary to set the error message corresponding to error cause.

- `applicationContext.xml`

Mapping exception class and error code (exception code).

```
<!-- omitted -->

<bean id="exceptionCodeResolver"
      class="org.terasoluna.gfw.common.exception.SimpleMappingExceptionCodeResolver">
  <property name="exceptionMappings">
    <map>
      <!-- omitted -->
      <entry key="ResourceNotFoundException" value="e.ex.fw.5001" />
      <entry key="HttpRequestMethodNotSupportedException" value="e.ex.fw.6001" />
      <entry key="MediaTypeNotAcceptableException" value="e.ex.fw.6002" />
      <entry key="HttpMediaTypeNotSupportedException" value="e.ex.fw.6003" />
      <entry key="MethodArgumentNotValidException" value="e.ex.fw.7001" />
      <entry key="BindException" value="e.ex.fw.7002" />
      <entry key="JsonParseException" value="e.ex.fw.7003" />
      <entry key="UnrecognizedPropertyException" value="e.ex.fw.7004" />
      <entry key="JsonMappingException" value="e.ex.fw.7005" />
      <entry key="TypeMismatchException" value="e.ex.fw.7006" />
      <entry key="BusinessException" value="e.ex.fw.8001" />
      <entry key="OptimisticLockingFailureException" value="e.ex.fw.8002" />
      <entry key="PessimisticLockingFailureException" value="e.ex.fw.8002" />
      <entry key="DataAccessException" value="e.ex.fw.9002" />
      <!-- omitted -->
    </map>
  </property>
```

```
<property name="defaultExceptionCode" value="e.ex.fw.9001" />
</bean>

<!-- omitted -->
```

Configuration example of message corresponding to error code is shown below.

For message management, refer to “*Message Management*” .

- xxx-web/src/main/resources/i18n/application-messages.properties

Message corresponding to error code (exception code) is set for the error that occurs in application layer.

```
# ---
# Application common messages
# ---
e.ex.fw.5001 = Resource not found.

e.ex.fw.6001 = Request method not supported.
e.ex.fw.6002 = Specified representation format not supported.
e.ex.fw.6003 = Specified media type in the request body not supported.

e.ex.fw.7001 = Validation error occurred on item in the request body.
e.ex.fw.7002 = Validation error occurred on item in the request parameters.
e.ex.fw.7003 = Request body format error occurred.
e.ex.fw.7004 = Unknown field exists in JSON.
e.ex.fw.7005 = Type mismatch error occurred in JSON field.
e.ex.fw.7006 = Type mismatch error occurred in request parameter or header or path variable.

e.ex.fw.8001 = Business error occurred.
e.ex.fw.8002 = Conflict with other processing occurred.

e.ex.fw.9001 = System error occurred.
e.ex.fw.9002 = System error occurred.
e.ex.fw.9003 = System error occurred.

# omitted
```

- xxx-web/src/main/resources/ValidationMessages.properties

Message corresponding to error code is set for the error that occurs in the input validation performed using Bean Validation.

```
# ---
# Bean Validation common messages
# ---
```

```
# for bean validation of standard
javax.validation.constraints.AssertFalse.message = "{0}" must be false.
javax.validation.constraints.AssertTrue.message = "{0}" must be true.
javax.validation.constraints.DecimalMax.message = "{0}" must be less than or equal to {value}.
javax.validation.constraints.DecimalMin.message = "{0}" must be greater than or equal to {value}.
javax.validation.constraints.Digits.message = "{0}" numeric value out of bounds. (<{int}>).
javax.validation.constraints.Future.message = "{0}" must be in the future.
javax.validation.constraints.Max.message = "{0}" must be less than or equal to {value}.
javax.validation.constraints.Min.message = "{0}" must be greater than or equal to {value}.
javax.validation.constraints.NotNull.message = "{0}" may not be null.
javax.validation.constraints.Null.message = "{0}" must be null.
javax.validation.constraints.Past.message = "{0}" must be in the past.
javax.validation.constraints.Pattern.message = "{0}" must match "{regexp}".
javax.validation.constraints.Size.message = "{0}" size must be between {min} and {max}.
```



```
# for bean validation of hibernate
org.hibernate.validator.constraints.CreditCardNumber.message = "{0}" invalid credit card number.
org.hibernate.validator.constraints.Email.message = "{0}" not a well-formed email.
org.hibernate.validator.constraints.Length.message = "{0}" length must be between {min} and {max}.
org.hibernate.validator.constraints.NotBlank.message = "{0}" may not be empty.
org.hibernate.validator.constraints.NotEmpty.message = "{0}" may not be empty.
org.hibernate.validator.constraints.Range.message = "{0}" must be between {min} and {max}.
org.hibernate.validator.constraints.SafeHtml.message = "{0}" may have unsafe html code.
org.hibernate.validator.constraints.ScriptAssert.message = "{0}" script expression "{script}" failed.
org.hibernate.validator.constraints.URL.message = "{0}" must be a valid URL.
```



```
# for common library
org.terasoluna.gfw.common.codelist.ExistInCodeList = "{0}" must exist in code list of {codeList}.
```

- xxx-domain/src/main/resources/i18n/domain-messages.properties

Message corresponding to error code (exception code) is set for the error in domain layer.

```
# omitted

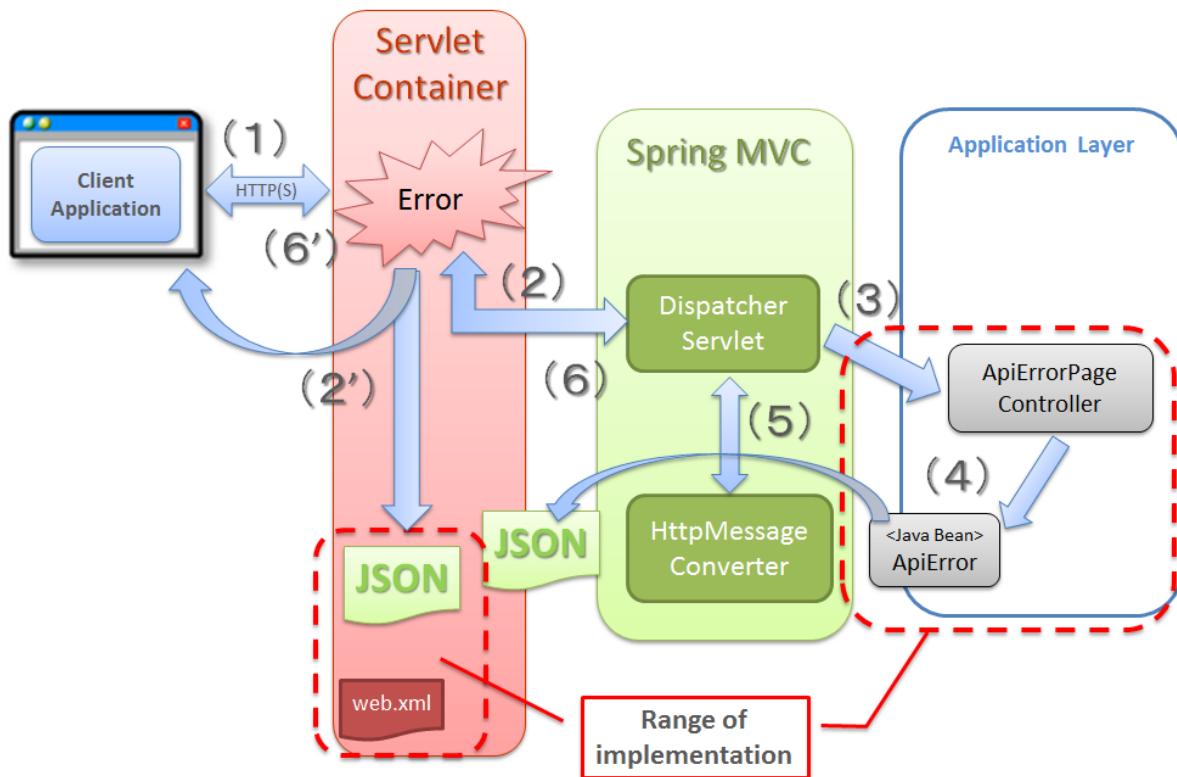
e.ex.mm.5001 = Specified member not found. member id : {0}
e.ex.mm.8001 = Cannot use specified sign id. sign id : {0}

# omitted
```

Implementing the error handling notified to Servlet Container

When an error occurs in Filter or when error responses are sent by using `HttpServletResponse#sendError`, the errors cannot be handled using exception handling feature of Spring MVC. Hence, these errors are notified to Servlet Container.

This section explains how to handle the errors notified to Servlet Container.



Sr. No.	Processing layer	Description
(1)	Servlet Container (AP Server)	Servlet Container receives a request from the client and performs process. Servlet Container detects an error during the process.
(2)		Servlet Container performs error handling according to the error page definition in <code>web.xml</code> . If the error is not fatal, Controller for error handling is called and the error is handled.
(2')		In case of a fatal error, a static JSON file provided in advance is fetched and a response is sent to the client.
(3)	Spring MVC (Framework)	Spring MVC calls the Controller that performs error handling.
(4)	Controller (Common Component)	An error object that retains error information is generated in the Controller and is returned to Spring MVC.
(5)	Spring MVC (Framework)	Spring MVC converts the error object to JSON format message by using <code>HttpMessageConverter</code> .
(6)		Spring MVC sets the JSON format error message in response BODY and sends a response to client.

Implementing the Controller that sends error response

Controller that sends the error response for the error notified to Servlet Container, is created.

```
package org.terasoluna.examples.rest.api.common.error;

import javax.inject.Inject;
import javax.servlet.RequestDispatcher;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.context.request.RequestAttributes;
import org.springframework.web.context.request.WebRequest;

// (1)
@RequestMapping("error")
@Controller
public class ApiErrorPageController {

    @Inject
    ApiErrorCreator apiErrorCreator; // (2)

    // (3)
    @RequestMapping
    @ResponseBody
    public ResponseEntity<ApiError> handleErrorResponse(
        @RequestParam("errorCode") String errorCode, // (4)
        WebRequest request) {
        // (5)
        HttpStatus httpStatus = HttpStatus.valueOf((Integer) request
            .getAttribute(RequestDispatcher.ERROR_STATUS_CODE,
            RequestAttributes.SCOPE_REQUEST));
        // (6)
        ApiError apiError = apiErrorCreator.createApiError(request, errorCode,
            httpStatus.getReasonPhrase());
        // (7)
        return new ResponseEntity<>(apiError, httpStatus);
    }

}
```

Sr. No.	Description
(1)	A Controller class is created for sending error responses. In the above example, it is mapped to Servlet path “/api/v1/error”.
(2)	A class that creates error information is injected.
(3)	Processing method that sends error response is created. Above example is implemented on considering only the case wherein, error page is handled by using a response code (<error-code>). Therefore, separate consideration is required for the case where an error page that is handled by using exception type (<exception-type>) is to be processed using this method.
(4)	Error code is received as request parameter.
(5)	Status code stored in request scope is fetched.
(6)	Error information corresponding to the error code received by request parameter is generated.
(7)	Error information fetched in (5) and (6) is sent as a response.

Creating a static JSON file to be sent as response when a fatal error occurs

A static JSON file is created to be sent as a response when a fatal error occurs.

- unhandledSystemError.json

```
{"code": "e.ex.fw.9999", "message": "Unhandled system error occurred."}
```

Settings for handling an error that is notified to Servlet Container

Settings for handling an error that is notified to Servlet Container are explained here.

- web.xml

```
<!-- omitted -->

<!-- (1) -->
<error-page>
    <error-code>404</error-code>
    <location>/api/v1/error?errorCode=e.ex.fw.5001</location>
</error-page>

<!-- (2) -->
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/WEB-INF/views/common/error/unhandledSystemError.json</location>
</error-page>

<!-- (3) -->
<mime-mapping>
    <extension>json</extension>
    <mime-type>application/json; charset=UTF-8</mime-type>
</mime-mapping>

<!-- omitted -->
```

Sr. No.	Description
(1)	<p>If needed, error page definition for response code is added.</p> <p>In the above example, when error "404 Not Found" occurs, Controller (ApiErrorHandlerController) that is mapped in request "/api/v1/error?errorCode=e.ex.fw.5001" is called and error response is sent.</p>
(2)	<p>Definition for handling a fatal error is added.</p> <p>When a fatal error occurs, it is recommended to respond with the static JSON provided in advance, as double failure may occur during the process that creates response information.</p> <p>In the above example, fixed JSON defined in "/WEB-INF/views/common/error/unhandledSystemError.json" is sent as response.</p>
(3)	<p>MIME type of json is specified.</p> <p>When multi byte characters are included in the JSON file created in (2), junk characters may be displayed at client side if charset=UTF-8 is not specified.</p> <p>When there are no multi byte characters in the JSON file, this setting is not mandatory.</p> <p>However, it is always safe to incorporate this setting.</p>

- When the request is sent to a non-existing path, following error response is sent.

```
HTTP/1.1 404 Not Found
Server: Apache-Coyote/1.1
X-Track: 2ad50fb5ba2441699c91a5b01edef83f
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 23:24:20 GMT

{"code": "e.ex.fw.5001", "message": "Resource not found."}
```

- When a fatal error occurs, following error response is sent.

```
HTTP/1.1 500 Internal Server Error
Server: Apache-Coyote/1.1
X-Track: 69db3854a19f439781584321d9ce8336
Content-Type: application/json
Content-Length: 68
Date: Thu, 20 Feb 2014 00:13:43 GMT
Connection: close

{"code":"e.ex.fw.9999","message":"Unhandled system error occurred."}
```

Security measures

Implementing security measures for RESTful Web Service is explained here.

This guideline recommends using Spring Security for implementation of security measures.

Authentication and Authorization

Todo

TBD

How to implement authentication and authorization using OAuth2 (Spring Security OAuth2), will be explained in subsequent versions.

CSRF measures

- Refer to *CSRF Measures* for the setting methods when CSRF measures are carried out for RESTful Web Service.
- Refer to *Disabling CSRF measures* for the setting methods when CSRF measures are not carried out for RESTful Web Service.

Conditional operations for resource

Todo

TBD

How to implement conditional process control using headers like Etag etc. will be explained in subsequent versions.

Cache control for resource

Todo

TBD

How to implement cache control using headers like Cache-Control/Expires/Pragma etc. will be explained in subsequent versions.

5.16.5 Appendix

Settings when RESTful Web Service and client application are operated as the same Web application

How to set DispatcherServlet for RESTful Web Service

When RESTful Web Service and client application are built as same Web application, it is recommended to divide as DispatcherServlet that receives the requests for RESTful Web Service and DispatcherServlet that receives client application requests.

How to divide DispatcherServlet is explained below.

- web.xml

```
<!-- omitted -->  
<!-- (1) -->
```

```
<servlet>
  < servlet-name>appServlet</servlet-name>
  < servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  < init-param>
    < param-name>contextConfigLocation</param-name>
    < param-value>classpath*:META-INF/spring/spring-mvc.xml</param-value>
  </init-param>
  < load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  < servlet-name>appServlet</servlet-name>
  < url-pattern>/</url-pattern>
</servlet-mapping>

<!-- (2) -->
<servlet>
  < servlet-name>restAppServlet</servlet-name>
  < servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  < init-param>
    < param-name>contextConfigLocation</param-name>
    <!-- (3) -->
    < param-value>classpath*:META-INF/spring/spring-mvc-rest.xml</param-value>
  </init-param>
  < load-on-startup>2</load-on-startup>
</servlet>
<!-- (4) -->
<servlet-mapping>
  < servlet-name>restAppServlet</servlet-name>
  < url-pattern>/api/v1/*</url-pattern>
</servlet-mapping>

<!-- omitted -->
```

Sr. No.	Description
(1)	Request mapping with the <code>DispatcherServlet</code> that receives request for client application.
(2)	Servlet (<code>DispatcherServlet</code>) that receives the request for RESTful Web Service is added. A name indicating it as RESTful Web Service Servlet, is specified in <code><servlet-name></code> element. In the above example, "restAppServlet" is specified as the Servlet name.
(3)	Spring MVC bean definition file to be used while building the <code>DispatcherServlet</code> for RESTful Web Service, is specified. In the above example, <code>META-INF/spring/spring-mvc-rest.xml</code> in class path, is specified as the Spring MVC bean definition file.
(4)	Pattern of servlet path mapped to the <code>DispatcherServlet</code> for RESTful Web Service, is specified. In the above example, the Servlet path under <code>"/api/v1/"</code> is mapped in the <code>DispatcherServlet</code> for RESTful Web Service. Typically, Servlet paths like <code>"/api/v1/"</code> <code>"/api/v1/members"</code> <code>"/api/v1/members/xxxxx"</code> are mapped in the <code>DispatcherServlet</code> ("restAppServlet") for RESTful Web Service.

Tip: Value to be specified in the value attribute of @RequestMapping annotation

Value in the wild card (*) part specified in `<url-pattern>` element, is specified as the value in "value" attribute of `@RequestMapping` annotation.

For example, when `@RequestMapping(value = "members")` is specified, it is deployed as the method that performs process for path `"/api/v1/members"`. Therefore, it is not necessary to specify the path `("api/v1")` that performs mapping in divided Servlets, in the value attribute of `@RequestMapping` annotation.

When specified as `@RequestMapping(value = "api/v1/members")`, it ends up being deployed as the method performing the process for path `"/api/v1/api/v1/members"`. Hence, please take note

of same.

Implementing hypermedia link

Implementation for including a hypermedia link to a related resource in JSON, is explained here.

Implementing common parts

- JavaBean that retains the link information is created.

```
package org.terasoluna.examples.rest.api.common.resource;

import java.io.Serializable;

// (1)
public class Link implements Serializable {

    private static final long serialVersionUID = 1L;

    private final String rel;

    private final String href;

    public Link(String rel, String href) {
        this.rel = rel;
        this.href = href;
    }

    public String getRel() {
        return rel;
    }

    public String getHref() {
        return href;
    }
}
```

Sr. No.	Description
(1)	JavaBean for link information that retains the link name and URL is created.

- An abstract class of Resource that retains collection of link information, is created.

```
package org.terasoluna.examples.rest.api.common.resource;

import java.net.URI;
import java.util.LinkedHashSet;
import java.util.Set;

import org.codehaus.jackson.map.annotate.JsonSerialize;
import org.codehaus.jackson.map.annotate.JsonSerialize.Inclusion;

// (2)
public abstract class AbstractLinksSupportedResource {

    // (3)
    @JsonSerialize(include = Inclusion.NON_EMPTY)
    private final Set<Link> links = new LinkedHashSet<>();

    public Set<Link> getLinks() {
        return links;
    }

    // (4)
    public AbstractLinksSupportedResource addLink(String rel, URI href) {
        links.add(new Link(rel, href.toString()));
        return this;
    }

    // (5)
    public AbstractLinksSupportedResource addSelf(URI href) {
        return addLink("self", href);
    }

    // (5)
    public AbstractLinksSupportedResource addParent(URI href) {
        return addLink("parent", href);
    }

}
```

Sr. No.	Description
(2)	An abstract class (JavaBean) of the Resource that retains collection of link information is created. It is assumed that this is the class inherited by the Resource class supporting hypermedia link.
(3)	A field that retains information of multiple links is defined. In the above example, @JsonSerialize(include = Inclusion.NON_EMPTY) is specified to prevent output to JSON when link is not specified.
(4)	Method to add link information is provided.
(5)	If required, method is provided to add common link information. In the above example, methods to add link information for accessing the resource itself ("self") and to add link information for accessing parent resource ("parent") are provided.

- Creating the class that builds the URI to be set in link information.

```
package org.terasoluna.examples.rest.api.common.resource;

import java.net.URI;

import org.springframework.stereotype.Component;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

@Component
public class ResourceUriBuilder {

    // omitted

    // (6)
    public URI self() {
        return ServletUriComponentsBuilder.fromCurrentRequest().build().toUri();
    }
}
```

```
// omitted
```

```
}
```

Sr. No.	Description
(6)	<p>Method that builds URI for accessing the resource itself.</p> <p>In the above example, URI is built by using <code>org.springframework.web.servlet.support.ServletUriComponentsBuilder</code> provided by Spring MVC.</p> <p>By using <code>ServletUriComponentsBuilder</code> class method, the URI can be built based on request information.</p> <p>Thus, a generic build process which is independent of a resource, can be implemented.</p> <p>For example, when a request is sent for <code>http://example.com/api/v1/members/M000000001</code>, return value of “self” method is same as the requested URI (<code>http://example.com/api/v1/members/M000000001</code>).</p> <p>Method that builds the URI to be set in link information should be implemented as required.</p>

Tip: When building an URI in `ServletUriComponentsBuilder`, by referring `X-Forwarded-Host` header, a configuration with a load balancer or Web server between the client and the application server is considered. However, it should be noted that expected URI will not be built if it is not combined with path configuration.

Implementation for each resource

- An abstract class of Resource that retains collection of link information, is inherited in Resource class.

```
package org.terasoluna.examples.rest.api.member;

// (1)
public class MemberResource extends
AbstractLinksSupportedResource implements Serializable {

    // omitted
```

```
}
```

Sr. No.	Description
(1)	An abstract class of Resource that retains collection of link information, is inherited. By this, the field (<code>links</code>) that retains collection of link information is imported, thus making it a Resource class that supports hypermedia link.

- Adding hypermedia link by REST API process.

```
@RequestMapping("members")
@Controller
public class MemberRestController {

    // omitted

    // (2)
    @Inject
    ResourceUriBuilder resourceUriBuilder;

    // omitted

    @RequestMapping(value = "{memberId}", method = RequestMethod.GET)
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public MemberResource getMember(
        @PathVariable("memberId") String memberId) {

        Member member = memberSevice.getMember(memberId);

        MemberResource responseResource = beanMapper.map(member,
            MemberResource.class);

        // (3)
        responseResource.addSelf(resourceUriBuilder.self());

        return responseResource;
    }

    // omitted
}
```

Sr. No.	Description
(2)	Object that builds the URI to be set in link information is injected.
(3)	Link information is added to resource. In the above example, object method that builds the URI to be set in link information is called and URI for accessing the resource itself, is added to the resource.

- Response example

Following response is obtained in actual operation.

```
GET /rest-api-web/api/v1/members/M000000001 HTTP/1.1
Accept: text/plain, application/json, application/*+json, */
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
```

```
{
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/rest-api-web/api/v1/members/M000000001"
  }],
  "memberId" : "M000000001",
  "firstName" : "John",
  "lastName" : "Smith",
  "genderCode" : "1",
  "dateOfBirth" : "2013-03-14",
  "emailAddress" : "user1394794959984@test.com",
  "telephoneNumber" : "09012345678",
  "zipCode" : "1710051",
  "address" : "Tokyo",
  "credential" : {
    "signId" : "user1394794959984@test.com",
    "passwordLastChangedAt" : "2014-03-14T11:02:41.477Z",
    "lastModifiedAt" : "2014-03-14T11:02:41.477Z"
  },
  "createdAt" : "2014-03-14T11:02:41.477Z",
  "lastModifiedAt" : "2014-03-14T11:02:41.477Z"
}
```

Creating RESTful Web Service conforming to HTTP specifications

A part of REST API implementation explained in this version does not conform to HTTP specifications.

This section explains the implementation that creates the RESTful Web Service conforming to HTTP specifications.

Settings for location header at the time of POST

When conforming to HTTP specifications, it is necessary to set the URI of created resource in the response header of POST method (“Location header”).

Implementation to set the URI of created resource in the response header in case of POST (“Location header”) is explained here.

Implementation of common components

- Class that builds the URI to be set in location header is created.

This class is same as the class explained in “*Implementing hypermedia link*”.

```
package org.terasoluna.examples.rest.api.common.resource;

import java.net.URI;

import org.springframework.stereotype.Component;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

@Component
public class ResourceUriBuilder<T> {

    // omitted

    // (1)
    public URI created(T id) {
        return children(String.valueOf(id));
    }

    public URI children (String... paths) {
        return ServletUriComponentsBuilder.fromCurrentRequestUri().path("/")
            .pathSegment(paths).build().toUri();
    }

    // omitted
}
```

}

Sr. No.	Description
(1)	<p>Method to build the URI of created resource.</p> <p>In this example, URI is built by using <code>org.springframework.web.servlet.support.ServletUriComponentsBuilder</code> provided by Spring MVC.</p> <p>By using <code>ServletUriComponentsBuilder</code> class method, URI can be built based on request information.</p> <p>Thus, a generic build process that is independent of the resource, can be implemented.</p> <p>For example, if POST method is used for <code>http://example.com/api/v1/members</code>, the return value of created method is “requested URI + “ / ” + ID specified as an argument”.</p> <p>Typically, if “M000000001” is specified in ID, it becomes</p> <p><code>http://example.com/api/v1/members/M000000001</code>.</p>

Tip: When building an URI in `ServletUriComponentsBuilder` , by referring X-Forwarded-Host header, a configuration with a load balancer or Web server between the client and the application server is considered. However, it should be noted that expected URI will not be built if it is not combined with path configuration.

Implementation for each resource

- The URI of created resource is set in Location header by REST API process.

```
@RequestMapping("members")
@Controller
public class MemberRestController {

    // omitted

    // (1)
    @Inject
    ResourceUriBuilder resourceUriBuilder;

    // omitted
```

```

@RequestMapping(method = RequestMethod.POST)
@ResponseBody
@ResponseStatus(HttpStatus.CREATED)
public ResponseEntity<MemberResource> postMembers(
    @RequestBody @Validated({ PostMembers.class, Default.class })
    MemberResource requestedResource) {

    Member creatingMember = beanMapper.map(requestedResource, Member.class);

    Member createdMember = memberService.createMember(creatingMember);

    MemberResource responseResource = beanMapper.map(createdMember,
        MemberResource.class);

    // (2)
    URI createdUri = resourceUriBuilder.created(responseResource.getMemberId());

    // (3)
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setLocation(createdUri);

    // (4)
    return new ResponseEntity<>(responseResource, responseHeaders);
}

// omitted
}

```

Sr. No.	Description
(1)	Object to build the URI of created resource is injected.
(2)	URI for created resource is built. In the above example, method of the object that builds URI of created resource is called and the URI is built.
(3)	URI of created resource is set in Location header.
(4)	Information to be set in Resource object and response header is stored in org.springframework.http.HttpEntity and then returned.

- Response example

Following response header is obtained in the actual operation.

```
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
X-Track: 693e132312d64998a7d8d6cabf3d13ef
Location: http://localhost:8080/rest-api-web/api/v1/members/M000000001
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Fri, 14 Mar 2014 12:34:31 GMT
```

Setting to dispatch OPTIONS method request to the Controller

When conforming to HTTP specifications, it is necessary to return the list of HTTP methods that are allowed to be called for each resource. Therefore, it is necessary to add the setting for dispatching OPTIONS method request, to the Controller.

By DispatcherServlet default setting, the request for OPTIONS method is not dispatched in the Controller with the list of methods allowed by DispatcherServlet being set in the Allow header.

- web.xml

```
<!-- omitted -->

<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath*:META-INF/spring/spring-mvc-rest.xml</param-value>
    </init-param>
    <!-- (1) -->
    <init-param>
        <param-name>dispatchOptionsRequest</param-name>
        <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<!-- omitted -->
```

Sr. No.	Description
(1)	Initialization parameter (dispatchOptionsRequest) value of DispatcherServlet that receives RESTful Web Service request is set to true.

Implementing OPTIONS method

When conforming to HTTP specifications, it is necessary to return the list of HTTP methods that are allowed to be called for each resource.

API implementation that responds with list of HTTP methods (REST AP) supported by the resource specified in URI, is shown below.

- REST API implementation

A process is implemented wherein, list of HTTP methods (REST API) supported by the resource specified in URI is sent as response.

```
@RequestMapping("members")
@Controller
public class MembersRestController {

    // omitted

    @RequestMapping(value = "{memberId}", method = RequestMethod.OPTIONS)
    @ResponseBody
    // (1)
    @ResponseStatus(HttpStatus.OK)
    public HttpEntity<Void> optionsMember(
        @PathVariable("memberId") String memberId) {

        // (2)
        memberService.getMember(memberId);

        // (3)
        return RestResponseUtils.createEntityOfOptions(
            HttpMethod.GET, HttpMethod.HEAD, HttpMethod.PUT, HttpMethod.DELETE);
    }

    // omitted
}
```

}

Sr. No.	Description
(1)	@ResponseStatus annotation is assigned as method annotation and the status code sent as response is specified. 200 (OK) is set in the value attribute of @ResponseStatus annotation.
(2)	Domain layer Service method is called and a check is performed to confirm if a resource matching with the ID fetched from path variable exists.
(3)	The HTTP method supported by resource specified in URI, is set in Allow header. In the above example, response entity set with a list of HTTP methods, is generated by calling utility methods.

- Utility methods

The process that responds with a list of HTTP methods (API) supported by resource specified in URI, uses the same logic for all the resources. Hence, it is advisable to adapt them as utility methods.

```
public static HttpEntity<Void> createEntityOfOptions(
    HttpMethod... allowedMethods) {

    Set<HttpMethod> allowedMethodSet = new LinkedHashSet<>(
        Arrays.asList(allowedMethods));
    if (!allowedMethodSet.contains(HttpMethod.OPTIONS)) {
        allowedMethodSet.add(HttpMethod.OPTIONS);
    }

    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setAllow(allowedMethodSet);

    return new HttpEntity<>(responseHeaders);
}
```

- Request example

```
OPTIONS /rest-api-web/api/v1/members/M000000004 HTTP/1.1
Accept: text/plain, application/json, application/*+json, /*
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
```

- Response example

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: 6d7bbc818c7f44e7942c54bc0ddc15bb
Allow: GET,HEAD,PUT,DELETE,OPTIONS
Content-Length: 0
Date: Mon, 17 Mar 2014 01:54:27 GMT
```

Implementing HEAD method

In order to conform to HTTP specifications, when GET method is implemented, HEAD method also needs to be implemented.

API implementation that responds with meta-information of the resource specified in URI, is as follows:

- REST API implementation

A process is implemented wherein meta information of the resource specified in URI is fetched.

```
@RequestMapping("members")
@Controller
public class MemberRestController {

    // omitted

    @RequestMapping(value = "{memberId}",
                    method = { RequestMethod.GET,
                               RequestMethod.HEAD }) // (1)
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public MemberResource getMember(
        @PathVariable("memberId") String memberId) {
```

```
// omitted
}

// omitted

}
```

Sr. No.	Description
(1)	<p>RequestMethod.HEAD is added to the method attribute of REST API @RequestMapping annotation that processes the GET method.</p> <p>HEAD method needs to respond only the header information, by performing the same process as GET method. Therefore, RequestMethod.HEAD is also specified in the method attribute of @RequestMapping annotation.</p> <p>It is advisable to perform a process similar to GET process as the Controller process since, the process for emptying response BODY is performed by standard functionality of Servlet API.</p>

- Request example

```
HEAD /rest-api-web/api/v1/members/M000000001 HTTP/1.1
Accept: text/plain, application/json, application/*+json, /*
User-Agent: Java/1.7.0_51
Host: localhost:8080
Connection: keep-alive
```

- Response example

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Track: 71093a551e624c149867b6bfec486d2c
Content-Type: application/json; charset=UTF-8
Content-Length: 452
Date: Thu, 13 Mar 2014 13:25:23 GMT
```

Disabling CSRF measures

Settings that prevent CSRF measures implementation for RESTful Web Service requests, are explained below.

Tip: Need to use session is eliminated when CSRF measures are not to be implemented.

In the following configuration example, session will not be used in Spring Security process.

CSRF measures are enabled as per the default settings of Blank project. By adding following settings, the process for CSRF measures is prevented for RESTful Web Service requests.

- `spring-security.xml`

```
<!-- omitted -->

<!-- (1) -->
<sec:http pattern="/api/v1/**"
    auto-config="true" use-expressions="true" create-session="stateless">
    <!--<sec:custom-filter ref="csrfFilter" before="LOGOUT_FILTER"/><!-- (2) --&gt;
    &lt;sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/&gt;
    &lt;!--&lt;sec:session-management session-authentication-strategy-ref="sessionAuthenticationStrategy" /--&gt;
&lt;/sec:http&gt;

&lt;sec:http auto-config="true" use-expressions="true"&gt;
    &lt;sec:custom-filter ref="csrfFilter" before="LOGOUT_FILTER"/&gt;
    &lt;sec:custom-filter ref="userIdMDCPutFilter" after="ANONYMOUS_FILTER"/&gt;
    &lt;sec:session-management session-authentication-strategy-ref="sessionAuthenticationStrategy" /&gt;
&lt;/sec:http&gt;

&lt;!-- omitted --&gt;</pre>
```

Sr. No.	Description
(1)	<p>Spring Security definition for REST API is added.</p> <p>URL pattern of REST API request path is specified in <code>pattern</code> attribute of <code><sec:http></code> element.</p> <p>In the above example, request path starting with <code>/api/v1/</code> is handled as the REST API request path.</p> <p>Further, session is no longer used in Spring Security process by setting <code>create-session</code> attribute to <code>stateless</code>.</p>
(2)	<p>Servlet Filter definition of CSRF measures is commented out, so as to disable the CSRF measures. It may be physically deleted in practice.</p>
(3)	<p>The definition used for referring to session management related components is commented out as session is not used. Actually it is better to delete it physically.</p>

Enabling XXE Injection measures

When handling XML format data in RESTful Web Service, it is necessary to implement [XXE \(XML External Entity\) Injection](#) measures.

terasoluna-gfw-web 1.0.1.RELEASE or higher versions are Spring MVC (3.2.10.RELEASE and above) dependent. As these Spring MVC versions implement XXE Injection measures, it is not necessary to implement them individually.

Warning: XXE (XML External Entity) Injection measures

When using terasoluna-gfw-web 1.0.0.RELEASE, a class provided by Spring-oxm should be used, as this version is dependent on the Spring MVC version (3.2.4.RELEASE), that does not implement XXE Injection measures.

Adding Spring-oxm as dependent artifact.

- pom.xml

```
<!-- omitted -->

<!-- (1) -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
    <version>${org.springframework-version}</version> <!-- (2) -->
</dependency>

<!-- omitted -->
```

Sr. No.	Description
(1)	Spring-oxm is added as a dependent artifact.
(2)	Spring version should be fetched from the placeholder (\${org.springframework-version}) that manages version number of Spring defined in pom.xml of terasoluna-gfw-parent.

Bean definition for mutual conversion between XML and object is performed by using the class provided by Spring-oxm.

- spring-mvc-rest.xml

```
<!-- omitted -->

<!-- (1) -->
<bean id="xmlMarshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <property name="packagesToScan" value="com.examples.app" /> <!-- (2) -->
</bean>

<!-- omitted -->

<mvc:annotation-driven>

    <mvc:message-converters>
        <!-- (3) -->
        <bean class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">
            <property name="marshaller" ref="xmlMarshaller" /> <!-- (4) -->
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>
```

```
        <property name="unmarshaller" ref="xmlMarshaller" /> <!-- (5) -->
    </bean>
</mvc:message-converters>

<!-- omitted -->

</mvc:annotation-driven>

<!-- omitted -->
```

Sr. No.	Description
(1)	Bean definition is carried out for <code>Jaxb2Marshaller</code> provided by Spring-oxm. <code>Jaxb2Marshaller</code> implements the XXE injection measure by default.
(2)	Package name with JAXB JavaBean (JavaBean assigned with <code>javax.xml.bind.annotation.XmlRootElement annotation</code>) stored in <code>packagesToScan</code> property, is specified. JAXB JavaBean stored under the specified package is scanned and is registered as JavaBean for marshalling and unmarshalling. JavaBean is scanned by the mechanism same as base-package attribute of <code><context:component-scan></code> .
(3)	Bean definition of <code>MarshallingHttpMessageConverter</code> is added to <code><mvc:message-converters></code> element which is the child element of <code><mvc:annotation-driven></code> element.
(4)	<code>Jaxb2Marshaller</code> bean defined in (1), is specified in <code>marshaller</code> property.
(5)	<code>Jaxb2Marshaller</code> bean defined in (1), is specified in <code>unmarshaller</code> property.

How to copy Joda-Time classes using Dozer

How to copy Joda-Time classes (`org.joda.time.DateTime`, `org.joda.time.LocalDate` etc.) using Dozer is explained here.

A custom converter is created for converting the Joda-Time class.

For details of custom converter, refer to “[Bean Mapping \(Dozer\)](#)”.

- `JodaDateTimeConverter.java`

```
package org.terasoluna.examples.rest.infra.dozer.converter;

import org.dozer.DozerConverter;
import org.joda.time.DateTime;

public class JodaDateTimeConverter extends DozerConverter<DateTime, DateTime> {

    public JodaDateTimeConverter() {
        super(DateTime.class, DateTime.class);
    }

    @Override
    public DateTime convertTo(DateTime source, DateTime destination) {
        // This method not called, because type of from/to is same.
        return convertFrom(source, destination);
    }

    @Override
    public DateTime convertFrom(DateTime source, DateTime destination) {
        return source;
    }
}
```

- `JodaLocalDateConverter.java`

```
package org.terasoluna.examples.rest.infra.dozer.converter;

import org.dozer.DozerConverter;
import org.joda.time.LocalDate;

public class JodaLocalDateConverter extends
        DozerConverter<LocalDate, LocalDate> {

    public JodaLocalDateConverter() {
        super(LocalDate.class, LocalDate.class);
    }
}
```

```
@Override
public LocalDate convertTo(LocalDate source, LocalDate destination) {
    // This method not called, because type of from/to is same.
    return convertFrom(source, destination);
}

@Override
public LocalDate convertFrom(LocalDate source, LocalDate destination) {
    return source;
}

}
```

Custom converter thus created is applied to Dozer.

For details of custom converter, refer to [Bean Mapping \(Dozer\)](#).

```
<!-- (1) -->
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="
               http://dozer.sourceforge.net http://dozer.sourceforge.net/schema/beanmapping.xsd
           ">

    <configuration>
        <custom-converters>
            <!-- (2) -->
            <converter type="org.terasoluna.examples.rest.infra.dozer.converter.JodaDateTimeConverter">
                <class-a>org.joda.time.DateTime</class-a>
                <class-b>org.joda.time.LocalDateTime</class-b>
            </converter>
            <converter type="org.terasoluna.examples.rest.infra.dozer.converter.JodaLocalDateConverter">
                <class-a>org.joda.time.LocalDate</class-a>
                <class-b>org.joda.time.LocalDateTime</class-b>
            </converter>
        </custom-converters>
    </configuration>

</mappings>
```

Sr. No.	Description
(1)	File that defines the operation settings for Dozer, is created. In this implementation, it is stored in <code>/xxx-domain/src/main/resources/META-INF/dozer/dozer-configuration-mapping.xml</code>
(2)	In the above example, custom converter definitions for Joda-Time classes (<code>org.joda.time.DateTime</code> and <code>org.joda.time.LocalDate</code>) are added.

Note: When Dozer is used in domain layer as well, it is recommended to store the file defining Dozer operation settings, in domain layer project (`xxx-domain`).

When Dozer is used only in application layer, it may be stored in application layer project (`xxx-web`).

Source code for application layer

Out of the application layer source codes used in explanation *How to use*, full version of the source code that was pasted in fragments, is attached herewith.

Sr. No.	Section	File name
(1)	<i>REST API implementation</i>	<i>MemberRestController.java</i>
(2)	<i>Implementing exception handling</i>	<i>ApiErrorCreator.java</i>
(3)		<i>ApiGlobalExceptionHandler.java</i>

Following files are excluded.

- JavaBean
- Configuration file

MemberRestController.java

```
java/org/terasoluna/examples/rest/api/member/MemberRestController.java

package org.terasoluna.examples.rest.api.member;

import java.util.ArrayList;
import java.util.List;

import javax.inject.Inject;
import javax.validation.groups.Default;

import org.dozer.Mapper;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotationResponseStatus;
import org.terasoluna.examples.rest.api.member.MemberResource.PostMembers;
import org.terasoluna.examples.rest.api.member.MemberResource.PutMember;
import org.terasoluna.examples.rest.domain.model.Member;
import org.terasoluna.examples.rest.domain.service.member.MemberService;

@RequestMapping("members")
@Controller
public class MemberRestController {

    @Inject
    MemberService memberSevice;

    @Inject
    Mapper beanMapper;

    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
```

```
@ResponseStatus(HttpStatus.OK)
public Page<MemberResource> getMembers (@Validated MembersSearchQuery query,
    Pageable pageable) {

    Page<Member> page = memberSevice.searchMembers(query.getName(), pageable);

    List<MemberResource> memberResources = new ArrayList<>();
    for (Member member : page.getContent()) {
        memberResources.add(beanMapper.map(member, MemberResource.class));
    }
    Page<MemberResource> responseResource =
        new PageImpl<>(memberResources, pageable, page.getTotalElements());

    return responseResource;
}

@RequestMapping(method = RequestMethod.POST)
@ResponseBody
@ResponseStatus(HttpStatus.CREATED)
public MemberResource postMembers (@RequestBody @Validated({
    PostMembers.class, Default.class }) MemberResource requestedResource) {

    Member creatingMember = beanMapper.map(requestedResource, Member.class);

    Member createdMember = memberSevice.createMember(creatingMember);

    MemberResource responseResource = beanMapper.map(createdMember,
        MemberResource.class);

    return responseResource;
}

@RequestMapping(value = "{memberId}", method = RequestMethod.GET)
@ResponseBody
@ResponseStatus(HttpStatus.OK)
public MemberResource getMember (@PathVariable("memberId") String memberId) {

    Member member = memberSevice.getMember(memberId);

    MemberResource responseResource = beanMapper.map(member,
        MemberResource.class);

    return responseResource;
}

@RequestMapping(value = "{memberId}", method = RequestMethod.PUT)
@ResponseBody
@ResponseStatus(HttpStatus.OK)
public MemberResource putMember (
    @PathVariable("memberId") String memberId,
    @RequestBody @Validated({
```

```
        PutMember.class, Default.class }) MemberResource requestedResource) {  
  
    Member updatingMember = beanMapper.map(requestedResource, Member.class);  
  
    Member updatedMember = memberSevice.updateMember(memberId,  
        updatingMember);  
  
    MemberResource responseResource = beanMapper.map(updatedMember,  
        MemberResource.class);  
  
    return responseResource;  
}  
  
@RequestMapping(value = "{memberId}", method = RequestMethod.DELETE)  
@ResponseBody  
@ResponseStatus(HttpStatus.NO_CONTENT)  
public void deleteMember(@PathVariable("memberId") String memberId) {  
  
    memberSevice.deleteMember(memberId);  
  
}  
}
```

ApiErrorCreator.java

```
java/org/terasoluna/examples/rest/api/common/error/ApiErrorCreator.java  
  
package org.terasoluna.examples.rest.api.common.error;  
  
import javax.inject.Inject;  
  
import org.springframework.context.MessageSource;  
import org.springframework.context.support.DefaultMessageSourceResolvable;  
import org.springframework.stereotype.Component;  
import org.springframework.validation.BindingResult;  
import org.springframework.validation.FieldError;  
import org.springframework.validation.ObjectError;  
import org.springframework.web.context.request.WebRequest;  
import org.terasoluna.gfw.common.message.ResultMessage;  
import org.terasoluna.gfw.common.message.ResultMessages;  
  
@Component  
public class ApiErrorCreator {
```

```
@Inject
MessageSource messageSource;

public ApiError createApiError(WebRequest request, String errorCode,
    String defaultErrorMessage, Object... arguments) {
    String localizedMessage = messageSource.getMessage(errorCode,
        arguments, defaultErrorMessage, request.getLocale());
    return new ApiError(errorCode, localizedMessage);
}

public ApiError createBindingResultApiError(WebRequest request,
    String errorCode, BindingResult bindingResult,
    String defaultErrorMessage) {
    ApiError apiError = createApiError(request, errorCode,
        defaultErrorMessage);
    for (FieldError fieldError : bindingResult.getFieldErrors()) {
        apiError.addDetail(createApiError(request, fieldError, fieldError
            .getField()));
    }
    for (ObjectError objectError : bindingResult.getGlobalErrors()) {
        apiError.addDetail(createApiError(request, objectError, objectError
            .getObjectName()));
    }
    return apiError;
}

private ApiError createApiError(WebRequest request,
    DefaultMessageSourceResolvable messageResolvable, String target) {
    String localizedMessage = messageSource.getMessage(messageResolvable,
        request.getLocale());
    return new ApiError(messageResolvable.getCode(), localizedMessage, target);
}

public ApiError createResultMessagesApiError(WebRequest request,
    String rootErrorCode, ResultMessages resultMessages,
    String defaultErrorMessage) {
    ApiError apiError;
    if (resultMessages.getList().size() == 1) {
        ResultMessage resultMessage = resultMessages.iterator().next();
        String errorCode = resultMessage.getCode();
        String errorText = resultMessage.getText();
        if (errorCode == null && errorText == null) {
            errorCode = rootErrorCode;
        }
        apiError = createApiError(request, errorCode, errorText,
            resultMessage.getArgs());
    } else {
        apiError = createApiError(request, rootErrorCode,
            defaultErrorMessage);
        for (ResultMessage resultMessage : resultMessages.getList()) {
            apiError.addDetail(createApiError(request, resultMessage
```

```
        .getCode(), resultMessage.getText(), resultMessage
        .getArgs())));
    }
}
return apiError;
}
}
```

ApiGlobalExceptionHandler.java

```
java/org/terasoluna/examples/rest/api/common/error/ApiGlobalExceptionHandler.java

package org.terasoluna.examples.rest.api.common.error;

import javax.inject.Inject;

import org.springframework.dao.OptimisticLockingFailureException;
import org.springframework.dao.PessimisticLockingFailureException;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.http.converter.HttpMessageNotReadableException;
import org.springframework.validation.BindException;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.method.annotation.ResponseEntityExceptionHandler;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ExceptionCodeResolver;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.exception.ResultMessagesNotificationException;

@ControllerAdvice
public class ApiGlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @Inject
    ApiErrorCreator apiErrorCreator;

    @Inject
    ExceptionCodeResolver exceptionCodeResolver;

    @Override
```

```
protected ResponseEntity<Object> handleExceptionInternal(Exception ex,
    Object body, HttpHeaders headers, HttpStatus status,
    WebRequest request) {
    final Object apiError;
    if (body == null) {
        String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
        apiError = apiErrorCreator.createApiError(request, errorCode, ex
            .getLocalizedMessage());
    } else {
        apiError = body;
    }
    return new ResponseEntity<>(apiError, headers, status);
}

@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(
    MethodArgumentNotValidException ex, HttpHeaders headers,
    HttpStatus status, WebRequest request) {
    return handleBindingResult(ex, ex.getBindingResult(), headers, status,
        request);
}

@Override
protected ResponseEntity<Object> handleBindException(BindException ex,
    HttpHeaders headers, HttpStatus status, WebRequest request) {
    return handleBindingResult(ex, ex.getBindingResult(), headers, status,
        request);
}

private ResponseEntity<Object> handleBindingResult(Exception ex,
    BindingResult bindingResult, HttpHeaders headers,
    HttpStatus status, WebRequest request) {
    String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
    ApiError apiError = apiErrorCreator.createBindingResultApiError(
        request, errorCode, bindingResult, ex.getMessage());
    return handleExceptionInternal(ex, apiError, headers, status, request);
}

@Override
protected ResponseEntity<Object> handleHttpMessageNotReadable(
    HttpResponseMessageNotReadableException ex, HttpHeaders headers,
    HttpStatus status, WebRequest request) {
    if (ex.getCause() instanceof Exception) {
        return handleExceptionInternal((Exception) ex.getCause(), null,
            headers, status, request);
    } else {
        return handleExceptionInternal(ex, null, headers, status, request);
    }
}

@ExceptionHandler(ResourceNotFoundException.class)
```

```
public ResponseEntity<Object> handleResourceNotFoundException(
    ResourceNotFoundException ex, WebRequest request) {
    return handleResultMessagesNotificationException(ex, null,
        HttpStatus.NOT_FOUND, request);
}

@ExceptionHandler(BusinessException.class)
public ResponseEntity<Object> handleBusinessException(BusinessException ex,
    WebRequest request) {
    return handleResultMessagesNotificationException(ex, null,
        HttpStatus.CONFLICT, request);
}

private ResponseEntity<Object> handleResultMessagesNotificationException(
    ResultMessagesNotificationException ex, HttpHeaders headers,
    HttpStatus status, WebRequest request) {
    String errorCode = exceptionCodeResolver.resolveExceptionCode(ex);
    ApiError apiError = apiErrorCreator.createResultMessagesApiError(
        request, errorCode, ex.getResultMessages(), ex.getMessage());
    return handleExceptionInternal(ex, apiError, headers, status, request);
}

@ExceptionHandler({ OptimisticLockingFailureException.class,
    PessimisticLockingFailureException.class })
public ResponseEntity<Object> handleLockingFailureException(Exception ex,
    WebRequest request) {
    return handleExceptionInternal(ex, null, null, HttpStatus.CONFLICT,
        request);
}

@ExceptionHandler(Exception.class)
public ResponseEntity<Object> handleSystemError(Exception ex,
    WebRequest request) {
    return handleExceptionInternal(ex, null, null,
        HttpStatus.INTERNAL_SERVER_ERROR, request);
}
```

Source code of the domain layer class created at the time of REST API implementation

Source code of domain layer class called from REST API explained in [How to use](#) is attached herewith.
Also, infrastructure layer is implemented by using JPA (Spring Data JPA).

Sr. No.	Classification	File name
(1)	model	<i>Member.java</i>
(2)		<i>MemberCredential.java</i>
(3)		<i>Gender.java</i>
(4)	repository	<i>MemberRepository.java</i>
(5)	service	<i>MemberService.java</i>
(6)		<i>MemberServiceImpl.java</i>
(7)	other	<i>DomainMessageCodes.java</i>
(8)		<i>member-mapping.xml</i>

Following files are excluded.

- JavaBean other than Entity
- Configuration files other than Dozer

Member.java

java/org/terasoluna/examples/rest/domain/model/Member.java

```
package org.terasoluna.examples.rest.domain.model;

import java.io.Serializable;

import javax.persistence.Access;
import javax.persistence.AccessType;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;
import javax.persistence.Transient;
import javax.persistence.Version;

import org.joda.time.DateTime;
import org.joda.time.LocalDate;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;

@Table(name = "t_member")
@Entity
public class Member implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private String memberId;

    private String firstName;

    private String lastName;

    @Transient
    private Gender gender;

    private LocalDate dateOfBirth;

    private String emailAddress;

    private String telephoneNumber;

    private String zipCode;

    private String address;
```

```
@CreatedDate
private DateTime createdAt;

@LastModifiedDate
private DateTime lastModifiedAt;

@Version
private long version;

@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "member_id")
private MemberCredential credential;

public String getMemberId() {
    return memberId;
}

public void setMemberId(String memberId) {
    this.memberId = memberId;
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public Gender getGender() {
    return gender;
}

public void setGender(Gender gender) {
    this.gender = gender;
}

@Access(AccessType.PROPERTY)
@Column(name = "gender")
public String getGenderCode() {
    if (gender == null) {
        return null;
    } else {
```

```
        return gender.getCode();
    }
}

public void setGenderCode(String genderCode) {
    this.gender = Gender.getByCode(genderCode);
}

public LocalDate getDateOfBirth() {
    return dateOfBirth;
}

public void setDateOfBirth(LocalDate dateOfBirth) {
    this.dateOfBirth = dateOfBirth;
}

public String getEmailAddress() {
    return emailAddress;
}

public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
}

public String getTelephoneNumber() {
    return telephoneNumber;
}

public void setTelephoneNumber(String telephoneNumber) {
    this.telephoneNumber = telephoneNumber;
}

public String getZipCode() {
    return zipCode;
}

public void setZipCode(String zipCode) {
    this.zipCode = zipCode;
}

public String getAddress() {
    return address;
}

public void setAddress(String address) {
    this.address = address;
}

public DateTime getCreatedAt() {
    return createdAt;
}
```

```
public void setCreatedAt(DateTime createdAt) {
    this.createdAt = createdAt;
}

public DateTime getLastModifiedAt() {
    return lastModifiedAt;
}

public void setLastModifiedAt(DateTime lastModifiedAt) {
    this.lastModifiedAt = lastModifiedAt;
}

public long getVersion() {
    return version;
}

public void setVersion(long version) {
    this.version = version;
}

public MemberCredential getCredential() {
    return credential;
}

public void setCredential(MemberCredential credential) {
    this.credential = credential;
}

}
```

MemberCredential.java

java/org/terasoluna/examples/rest/domain/model/MemberCredential.java

```
package org.terasoluna.examples.rest.domain.model;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;

import org.joda.time.DateTime;
```

```
import org.springframework.data.annotation.LastModifiedDate;

@Table(name = "t_member_credential")
@Entity
public class MemberCredential implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private String memberId;

    private String signId;

    private String password;

    private String previousPassword;

    private DateTime passwordLastChangedAt;

    @LastModifiedDate
    private DateTime lastModifiedAt;

    @Version
    private long version;

    public String getMemberId() {
        return memberId;
    }

    public void setMemberId(String memberId) {
        this.memberId = memberId;
    }

    public String getSignId() {
        return signId;
    }

    public void setSignId(String signId) {
        this.signId = signId;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getPreviousPassword() {
        return previousPassword;
    }
}
```

```
}

public void setPreviousPassword(String previousPassword) {
    this.previousPassword = previousPassword;
}

public DateTime getPasswordLastChangedAt() {
    return passwordLastChangedAt;
}

public void setPasswordLastChangedAt(DateTime passwordLastChangedAt) {
    this.passwordLastChangedAt = passwordLastChangedAt;
}

public DateTime getLastModifiedAt() {
    return lastModifiedAt;
}

public void setLastModifiedAt(DateTime lastModifiedAt) {
    this.lastModifiedAt = lastModifiedAt;
}

public long getVersion() {
    return version;
}

public void setVersion(long version) {
    this.version = version;
}

}
```

Gender.java

java/org/terasoluna/examples/rest/domain/model/Gender.java

```
package org.terasoluna.examples.rest.domain.model;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import org.springframework.util.Assert;

public enum Gender {
```

```
UNKNOWN("0"), MEN("1"), WOMEN("2");

private static final Map<String, Gender> genderMap;

static {
    Map<String, Gender> map = new HashMap<>();
    for (Gender gender : values()) {
        map.put(gender.code, gender);
    }
    genderMap = Collections.unmodifiableMap(map);
}

private final String code;

private Gender(String code) {
    this.code = code;
}

public static Gender getByCode(String code) {
    Gender gender = genderMap.get(code);
    Assert.notNull(gender, "gender code is invalid. code : " + code);
    return gender;
}

public String getCode() {
    return code;
}

}
```

MemberRepository.java

```
java/org/terasoluna/examples/rest/domain/repository/member/MemberRepository.java

package org.terasoluna.examples.rest.domain.repository.member;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.terasoluna.examples.rest.domain.model.Member;

public interface MemberRepository extends JpaRepository<Member, String> {
```

```
@Query("SELECT m FROM Member m"
    + " WHERE m.firstName LIKE :name% ESCAPE '~'"
    + " OR m.lastName LIKE :name% ESCAPE '~'")
Page<Member> findPageByContainsName(@Param("name") String name,
    Pageable pageable);

}
```

MemberService.java

java/org/terasoluna/examples/rest/domain/service/member/MemberService.java

```
package org.terasoluna.examples.rest.domain.service.member;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.terasoluna.examples.rest.domain.model.Member;

public interface MemberService {

    Page<Member> searchMembers(String name, Pageable pageable);

    Member getMember(String memberId);

    Member createMember(Member creatingMember);

    Member updateMember(String memberId, Member updatingMember);

    void deleteMember(String memberId);

}
```

MemberServiceImpl.java

java/org/terasoluna/examples/rest/domain/service/member/MemberServiceImpl.java

```
package org.terasoluna.examples.rest.domain.service.member;

import javax.inject.Inject;
```

```
import javax.inject.Named;

import org.dozer.Mapper;
import org.joda.time.DateTime;
import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.util.StringUtils;
import org.terasoluna.examples.rest.domain.message.DomainMessageCodes;
import org.terasoluna.examples.rest.domain.model.Member;
import org.terasoluna.examples.rest.domain.model.MemberCredential;
import org.terasoluna.examples.rest.domain.repository.member.MemberRepository;
import org.terasoluna.gfw.common.date.DateFactory;
import org.terasoluna.gfw.common.exception.BusinessException;
import org.terasoluna.gfw.common.exception.ResourceNotFoundException;
import org.terasoluna.gfw.common.message.ResultMessages;
import org.terasoluna.gfw.common.query.QueryEscapeUtils;
import org.terasoluna.gfw.common.sequencer.Sequencer;

@Transactional
@Service
public class MemberServiceImpl implements MemberService {

    @Inject
    MemberRepository memberRepository;

    @Inject
    @Named("memberIdSequencer")
    Sequencer<String> sequencer;

    @Inject
    DateFactory dateFactory;

    @Inject
    PasswordEncoder passwordEncoder;

    @Inject
    Mapper beanMapper;

    @Transactional(readOnly = true)
    public Page<Member> searchMembers(String name, Pageable pageable) {

        // escape to like condition value
        String escapedName = QueryEscapeUtils.toLikeCondition(name);

        // find members that matches with search criteria
        return memberRepository.findPageByContainsName(escapedName, pageable);
    }
}
```

```
@Transactional(readOnly = true)
public Member getMember(String memberId) {
    // find member
    Member member = memberRepository.findOne(memberId);
    if (member == null) {
        // If member is not exists
        throw new ResourceNotFoundException(ResultMessages.error().add(
            DomainMessageCodes.E_EX_MM_5001, memberId));
    }
    return member;
}

public Member createMember(Member creatingMember) {

    MemberCredential creatingCredential = creatingMember.getCredential();

    // get processing current date time
    DateTime currentDateTime = dateFactory.newDateTime();

    // set id
    String newMemberId = sequencer.getNext();
    creatingMember.setMemberId(newMemberId);
    creatingCredential.setMemberId(newMemberId);

    // decide sign id(email-address)
    String signId = creatingCredential.getSignId();
    if (!StringUtils.hasLength(signId)) {
        signId = creatingMember.getEmailAddress();
        creatingCredential.setSignId(signId.toLowerCase());
    }

    // encrypt password
    String rawPassword = creatingCredential.getPassword();
    creatingCredential.setPassword(passwordEncoder.encode(rawPassword));
    creatingCredential.setPasswordLastChangedAt(currentDateTime);

    // save member & member credential
    try {
        return memberRepository.saveAndFlush(creatingMember);
    } catch (DataIntegrityViolationException e) {
        // If sign id is already used
        throw new BusinessException(ResultMessages.error().add(
            DomainMessageCodes.E_EX_MM_8001,
            creatingCredential.getSignId(), e));
    }
}

public Member updateMember(String memberId, Member updatingMember) {
    // get member
    Member member = getMember(memberId);
```

```
// override updating member attributes
beanMapper.map(updatingMember, member, "member.update");

// save updating member
return memberRepository.save(member);
}

public void deleteMember(String memberId) {

    // delete member
    memberRepository.delete(memberId);

}
}
```

DomainMessageCodes.java

java/org/terasoluna/examples/rest/domain/message/DomainMessageCodes.java

```
package org.terasoluna.examples.rest.domain.message;

/**
 * Message codes of domain layer message.
 * @author DomainMessageCodesGenerator
 */
public class DomainMessageCodes {

    private DomainMessageCodes() {
        // NOP
    }

    /** e.ex.mm.5001=Specified member not found. member id : {0} */
    public static final String E_EX_MM_5001 = "e.ex.mm.5001";

    /** e.ex.mm.8001=Cannot use specified sign id. sign id : {0} */
    public static final String E_EX_MM_8001 = "e.ex.mm.8001";
}
```

member-mapping.xml

In the implemented Service class, “*Bean Mapping (Dozer)*” is used while copying the value specified by client in Member object.

When it is alright to simply copy the field values, Bean mapping definition need not be added. However, in this implementation it needs to be ensured that, items which are not to be updated (memberId, credential, createdAt and version) are not copied.

Bean mapping definition needs to be added in order to ensure that specific fields are not copied.

resources/META-INF/dozer/member-mapping.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
           http://dozer.sourceforge.net/schema/beanmapping.xsd">

    <mapping map-id="member.update">
        <class-a>org.terasoluna.examples.rest.domain.model.Member</class-a>
        <class-b>org.terasoluna.examples.rest.domain.model.Member</class-b>
        <field-exclude>
            <a>memberId</a>
            <b>memberId</b>
        </field-exclude>
        <field-exclude>
            <a>credential</a>
            <b>credential</b>
        </field-exclude>
        <field-exclude>
            <a>createdAt</a>
            <b>createdAt</b>
        </field-exclude>
        <field-exclude>
            <a>lastModifiedAt</a>
            <b>lastModifiedAt</b>
        </field-exclude>
        <field-exclude>
            <a>version</a>
            <b>version</b>
        </field-exclude>
    </mapping>

</mappings>
```

5.17 File Upload

5.17.1 Overview

This chapter explains how to upload files.

Files are uploaded using the File Upload functionality supported by Servlet3.0 and classes provided by Spring Web.

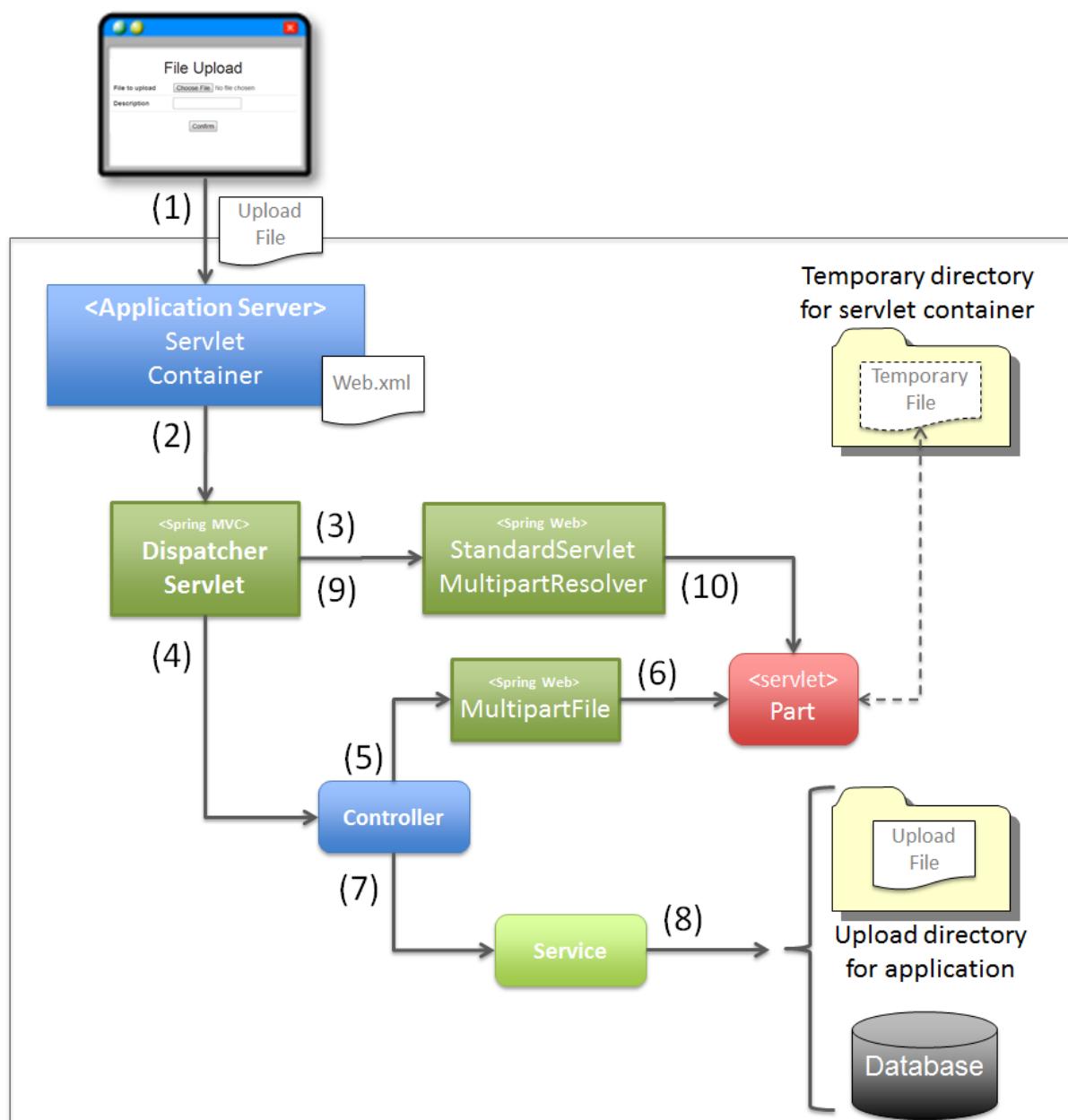
Note: In this chapter, File Upload functionality supported by Servlet3.0 is used; hence, Servlet version 3.0 or above is a prerequisite here.

Warning: In the application server to be used, if implementation of File Upload is dependent on the implementation of Apache Commons FileUpload, vulnerabilities of security mentioned in [CVE-2014-0050](#) may occur. Hence ensure that there are no such vulnerabilities in the application server to be used.

It is necessary to use version 7.0.52 or above for Tomcat series 7, and version 8.0.3 or above for series 8.

Basic flow of upload process

Basic flow of uploading files using File Upload functionality supported by Servlet3.0, and classes of Spring Web, is as shown below.



Sr. No.	Description
(1)	Select and upload the target files.
(2)	Servlet container receives multipart/form-data request and calls <code>org.springframework.web.servlet.DispatcherServlet</code> .
5.17. File Upload	<p>(3) <code>DispatcherServlet</code> calls the method of <code>org.springframework.web.multipart.support.StandardServletMultipartResolver</code> to enable File Upload functionality of Servlet3.0 in Spring MVC.</p> <p><code>StandardServletMultipartResolver</code> generates <code>org.springframework.web.multipart.MultipartFile</code> object that wraps the</p>

Note: Controller performs the process for `MultipartFile` object of Spring Web; hence implementation which is dependent on the File Upload API provided by Servlet3.0 can be excluded.

About classes provided by Spring Web

Classes provided by Spring Web for uploading a file are as follows:

Sr. No.	Class name	Description
1.	org.springframework.web.multipart. MultipartFile	Interface indicating uploaded file. It plays a role in abstraction of file objects handled by the File Upload functionality to be used.
2.	org.springframework.web.multipart.support. StandardMultipartHttpServletRequest\$ StandardMultipartFile	MultipartFile class of File Upload functionality introduced through Servlet3.0. Process is delegated to the Part object introduced through Servlet3.0.
3.	org.springframework.web.multipart. MultipartResolver	Interface that resolves the analysis method of multipart/form-data request. It plays a role in generating MultipartFile object corresponding to implementation of File Upload functionality.
4.	org.springframework.web.multipart.support. StandardServletMultipartResolver	MultipartResolver class for File Upload functionality introduced through Servlet3.0.
5.	org.springframework.web.multipart.support. MultipartFilter	Class that enables fetching of request parameters in Servlet Filter process, at the time of multipart/form-data request. If this class is not used, request parameters cannot be fetched in Servlet Filter; hence CSRF Token Check functionality provided by Spring Security does not work correctly. To be more precise, as CSRF token cannot be fetched, it always throws CSRF token error leading to file upload failure.

Tip: In this guideline, it is a prerequisite to use File Upload functionality implemented through

Servlet 3.0. However, Spring Web also provides an implementation class for “Apache Commons FileUpload”. The difference in implementation of upload processes is absorbed by MultipartResolver and MultipartFile objects; hence it does not affect Controller implementation. It should be used when a servlet container (Tomcat6 etc.) not supported by Servlet3.0 needs to be used.

Warning: When using Apache Commons FileUpload, vulnerabilities of security mentioned in [CVE-2014-0050](#) may occur. Ensure that there are no such vulnerabilities in the Apache Commons FileUpload version to be used.

When using Apache Commons FileUpload, version 1.2.1 or above of the 1.2 series and 1.3.1 or above of the 1.3 series should be used.

5.17.2 How to use

.._file-upload_how_to_usr_application_settings:

Application settings

Settings to enable Servlet3.0 upload functionality

Perform the following settings to enable upload functionality of Servlet3.0.

- web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee
version="3.0"> <!-- (1) (2) -->

<servlet>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <!-- omitted -->
  <multipart-config> <!-- (3) -->
    <max-file-size>5242880</max-file-size> <!-- (4) -->
    <max-request-size>27262976</max-request-size> <!-- (5) -->
    <file-size-threshold>0</file-size-threshold> <!-- (6) -->
  </multipart-config>
</servlet>

<!-- omitted -->
```

</web-app>

Sr. No.	Description
(1)	Specify the XSD file of Servlet3.0 or above in <code>xsi:schemaLocation</code> attribute of <code><web-app></code> element.
(2)	Specify version 3.0 or above in the <code>version</code> attribute of <code><web-app></code> element.
(3)	Add <code><multipart-config></code> element to <code><servlet></code> element of the Servlet using the File Upload functionality.
(4)	<p>Specify the maximum file size of 1 upload-permissible file in bytes. If not specified, -1 (no limit) is set by default. If it exceeds the specified value, exception <code>org.springframework.web.multipart.MultipartException</code> occurs.</p> <p>In the above example, a file size of 5MB is specified.</p>
(5)	<p>Specify the maximum Content-Length value of multipart/form-data request. If not specified, -1 (no limit) is set by default. If it exceeds the specified value, exception <code>org.springframework.web.multipart.MultipartException</code> occurs.</p> <p>Value to be set in this parameter should be calculated by the following formula.</p> <p>(“maximum file size of 1 file to be uploaded” * “Number of files allowed to be uploaded simultaneously”) + “Data size of other form fields” + “Meta information size of multipart/form-data request”</p> <p>In the above example, parameter value of 26MB is specified. Its breakup is, 25MB (5MB * 5 files) and 1MB (number of bytes of meta information + number of bytes of form fields).</p>
(6)	Specify the threshold value (number of bytes for 1 file) if the contents of uploaded file are to be saved as a temporary file.
1144	<p>5 Architecture in Detail - TERASOLUNA Global Framework</p> <p>If this parameter is not specified explicitly, there are application servers wherein values specified for elements <code><max-file-size></code> and <code><max-request-size></code> are considered invalid; hence default value (0) is being specified explicitly.</p>

Warning: In order to increase the resistance against Dos attack, max-file-size and max-request-size should be specified without fail.

For Dos attack, refer to [Dos attack with respect to upload functionality](#).

Note:

Uploaded file is by default output as temporary file. However, its output can be controlled using the configuration value of <file-size-threshold> element, which is the child element of <multipart-config>.

```
<!-- omitted -->

<multipart-config>
    <!-- omitted -->
    <file-size-threshold>32768</file-size-threshold> <!-- (7) -->
</multipart-config>

<!-- omitted -->
```

Sr. No.	Description
(7)	<p>Specify the threshold file size (number of bytes of 1 file) if contents of uploaded file are to be saved as a temporary file.</p> <p>If not specified, 0 is set.</p> <p>If uploaded file size exceeds the specified value, it is output as a temporary file to the disk and deleted when the request is completed.</p> <p>In the above example, 32KB is specified.</p>

Warning: This parameter shows a trade-off relationship as indicated by the following points.

Hence, **configuration value corresponding to system characteristics should be specified..**

- Increasing the configuration value improves processing performance as, processing gets completed within available memory. However, there is a high possibility that `OutOfMemoryError` may occur due to Dos attack.
- If configuration value is reduced, memory utilization can be controlled to the minimum, thereby avoiding the possibility of `OutOfMemoryError` due to Dos attack etc. However, there is a high possibility of performance degradation since the frequency of disk IO generation is high.

To change output directory of temporary files, specify directory path in <location> element, which is the child element of <multipart-config>.

```
<!-- omitted -->

<multipart-config>
    <location>/tmp</location> <!-- (8) -->
    <!-- omitted -->
</multipart-config>

<!-- omitted -->
```

Sr. No.	Description
(8)	<p>Specify the directory path for outputting temporary files.</p> <p>When omitted, they are output to the directory that stores temporary files of application server.</p> <p>In the above example, /tmp is specified.</p>

Warning: The directory specified in `<location>` element is the one used by the application server (servlet container) and **cannot be accessed from application**.

When the files uploaded as application are to be saved as temporary files, they should be output to a directory other than the directory specified in `<location>` element.

Settings to enable fetching of request parameters in Servlet Filter processing

Perform the following settings to fetch request parameters in Servlet Filter processing at the time of multipart/form-data request.

- web.xml

```
<!-- (1) -->
<filter>
    <filter-name>MultipartFilter</filter-name>
    <filter-class>org.springframework.web.multipart.support.MultipartFilter</filter-class>
</filter>
<!-- (2) -->
<filter-mapping>
    <filter-name>MultipartFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Sr. No.	Description
(1)	Define MultipartFilter as the Servlet Filter.
(2)	Specify the URL pattern for applying MultipartFilter.

Warning: **MultipartFilter needs to be defined before the Servlet Filter that accesses request parameters.**

When security measures are to be carried out using Spring Security, it should be defined before `springSecurityFilterChain`. Further, when request parameters are accessed by a project-specific Servlet Filter, MultipartFilter should be defined before that Servlet Filter.

Settings to link Spring MVC with upload functionality of Servlet3.0

Perform the following settings to link Spring MVC with Servlet3.0 upload functionality.

- `spring-mvc.xml`

```
<bean id="multipartResolver"
      class="org.springframework.web.multipart.support.StandardServletMultipartResolver"> <!--
</bean>
```

Sr. No.	Description
(1)	<p>Define a bean for <code>StandardServletMultipartResolver</code> which is a <code>MultipartResolver</code> for Servlet3.0.</p> <p>BeanID should be "multipartResolver".</p> <p>By performing these settings, the uploaded file can be treated as <code>org.springframework.web.multipart.MultipartFile</code> and received as a Controller argument and form object property.</p>

Settings for exception handling

Add the exception handling definition of `MultipartException` which occurs when a request for file or multipart with non-permissible size is sent.

`MultipartException` is an exception caused due to file size specified by the client; hence it is

recommended to handle it as a client error (HTTP response code=4xx).

If exception handling is not added for individual exception, it is eventually treated as system error; hence make sure that it is defined without fail.

Settings for handling `MultipartException` differ depending upon whether `MultipartFilter` is used or not.

In case of using `MultipartFilter`, exception handling is carried out by using the `<error-page>` functionality of servlet container.

Example of settings is shown below.

- `web.xml`

```
<error-page>
  <!-- (1) -->
  <exception-type>org.springframework.web.multipart.MultipartException</exception-type>
  <!-- (2) -->
  <location>/WEB-INF/views/common/error/fileUploadError.jsp</location>
</error-page>
```

Sr. No.	Description
(1)	Specify <code>MultipartException</code> as the exception class for handling.
(2)	Specify the file displayed when <code>MultipartException</code> occurs. In the above example, "/WEB-INF/views/common/error/fileUploadError.jsp" is specified.

- `fileUploadError.jsp`

```
<%-- (3) --%>
<% response.setStatus(HttpServletRequest.SC_BAD_REQUEST); %>
<!DOCTYPE html>
<html>

  <!-- omitted -->

</html>
```

Sr. No.	Description
(3)	<p>Set HTTP status code by calling the API of <code>HttpServletResponse</code>.</p> <p>In the above request, "400" (Bad Request) is set.</p> <p>When not set explicitly, the HTTP status code is considered as "500" (Internal Server Error).</p>

When not using `MultipartFilter`, carry out exception handling by using `SystemExceptionResolver`.

Example of settings is shown below.

- `spring-mvc.xml`

```

<bean class="org.terasoluna.gfw.web.exception.SystemExceptionResolver">
    <!-- omitted -->
    <property name="exceptionMappings">
        <map>
            <!-- omitted -->
            <!-- (4) -->
            <entry key="MultipartException"
                  value="common/error/fileUploadError" />

        </map>
    </property>
    <property name="statusCodes">
        <map>
            <!-- omitted -->
            <!-- (5) -->
            <entry key="common/error/fileUploadError" value="400" />

        </map>
    </property>
    <!-- omitted -->
</bean>

.. tabularcolumns:: |p{0.10\linewidth}|p{0.90\linewidth}|
.. list-table::
:header-rows: 1
:widths: 10 90

* - Sr. No.
  - Description
* - | (4)
  - | In \ ``exceptionMappings``\ of \ ``SystemExceptionResolver``\ , add the definition f

```

```
|  
| In the above example, \ \"common/error/fileUploadError\"\ is specified.  
* - | (5)  
- | Add the definition of HTTP status code which is received as response when 'Multipart  
|  
| In the above example, \ \"400\"\ (Bad Request) is specified.  
| By specifying client error (HTTP response code = 4xx),  
| the level of log which is output by the class ('HandlerExceptionResolverLoggingInter
```

Add exception code settings when setting an exception code for `MultipartException`.

Exception code is output to the log which is output using log output functionality of common library.

Exception code can also be referred from View (JSP).

For referring to exception code from View (JSP), refer to *Method to display system exception code on screen*.

- `applicationContext.xml`

```
<bean id="exceptionCodeResolver"  
      class="org.terasoluna.gfw.common.exception.SimpleMappingExceptionCodeResolver">  
  <property name="exceptionMappings">  
    <map>  
      <!-- (6) -->  
      <entry key="MultipartException" value="e.xx/fw.6001" />  
      <!-- omitted -->  
    </map>  
  </property>  
  <property name="defaultExceptionCode" value="e.xx/fw.9001" />  
  <!-- omitted -->  
</bean>
```

Sr. No.	Description
(6)	<p>In <code>exceptionMappings</code> of <code>SimpleMappingExceptionCodeResolver</code>, add the exception code to be applied when <code>MultipartException</code> occurs.</p> <p>In the above example, "e.xx/fw.6001" is specified.</p> <p>When it is not defined individually, exception code specified in <code>defaultExceptionCode</code> is applied.</p>

Uploading a single file

The explanation about uploading a single file is given below.

File Upload

File to upload	<input type="button" value="Choose File"/> No file chosen
Description	<input type="text"/>
<input type="button" value="Upload"/>	

There are 2 methods to upload a single file. One is by binding `org.springframework.web.multipart.MultipartFile` object to the form object and the other is by receiving it directly as Controller argument. However, this guideline recommends the first method wherein it is received after it is bound with the form object.

The reason for this being, single field check of the uploaded file can be performed using Bean Validation.

How to receive a single file by binding it to form object is explained below.

Implementing form

```
public class FileUploadForm implements Serializable {  
  
    // omitted  
  
    private MultipartFile file; // (1)  
  
    @NotNull  
    @Size(min = 0, max = 100)  
    private String description;  
  
    // omitted getter/setter methods.  
  
}
```

Sr. No.	Description
(1)	Define properties of <code>org.springframework.web.multipart.MultipartFile</code> in form object.

Implementing JSP

```
<form:form
    action="${pageContext.request.contextPath}/article/uploadFile" method="post"
    modelAttribute="fileUploadForm" enctype="multipart/form-data"> <!-- (1) (2) -->
    <table>
        <tr>
            <th width="35%">File to upload</th>
            <td width="65%">
                <form:input type="file" path="file" /> <!-- (3) -->
                <form:errors path="file" />
            </td>
        </tr>
        <tr>
            <th width="35%">Description</th>
            <td width="65%">
                <form:input path="description" />
                <form:errors path="description" />
            </td>
        </tr>
        <tr>
            <td>&ampnbsp</td>
            <td><form:button>Upload</form:button></td>
        </tr>
    </table>
</form:form>
```

Sr. No.	Description
(1)	Specify "multipart/form-data" in the enctype attribute of <form:form> element.
(2)	Specify attribute name of form object in the modelAttribute of <form:form> element. In the above example, "fileUploadForm" is specified.
(3)	Specify "file" in type attribute of <form:input> element and specify MultipartFile property name in path attribute. In the above example, the uploaded file is stored in "file" property of FileUploadForm object.

Implementing Controller

```

@RequestMapping("article")
@Controller
public class ArticleController {

    @Value("${upload.allowableFileSize}")
    private int uploadAllowableFileSize;

    // omitted

    // (1)
    @ModelAttribute
    public FileUploadForm setFileUploadForm() {
        return new FileUploadForm();
    }

    // (2)
    @RequestMapping(value = "upload", method = RequestMethod.GET, params = "form")
    public String uploadForm() {
        return "article/uploadForm";
    }

    // (3)
    @RequestMapping(value = "upload", method = RequestMethod.POST)
    public String upload(@Validated FileUploadForm form,
                        BindingResult result, RedirectAttributes redirectAttributes) {

        if (result.hasErrors()) {
            return "article/uploadForm";
        }
    }
}

```

```
}

MultipartFile uploadFile = form.getFile();

// (4)
if (!StringUtils.hasLength(uploadFile.getOriginalFilename())) {
    result.rejectValue(uploadFile.getName(), "e.xx.at.6002");
    return "article/uploadForm";
}

// (5)
if (uploadFile.isEmpty()) {
    result.rejectValue(uploadFile.getName(), "e.xx.at.6003");
    return "article/uploadForm";
}

// (6)
if (uploadAllowableFileSize < uploadFile.getSize()) {
    result.rejectValue(uploadFile.getName(), "e.xx.at.6004",
        new Object[] { uploadAllowableFileSize }, null);
    return "article/uploadForm";
}

// (7)
// omit processing of upload.

// (8)
redirectAttributes.addFlashAttribute(ResultMessages.success().add(
    "i.xx.at.0001"));

// (9)
return "redirect:/article/upload?complete";
}

@RequestMapping(value = "upload", method = RequestMethod.GET, params = "complete")
public String uploadComplate() {
    return "article/uploadComplete";
}

// omitted

}
```

Sr. No.	Description
(1)	<p>Method of storing the form object for file upload in Model.</p> <p>In the above example, the attribute name for storing form object in Model is "fileUploadForm".</p>
(2)	Processing method for displaying upload screen.
(3)	Processing method for uploading files.
(4)	<p>It is checked whether the files for upload are selected.</p> <p>To check if the files are selected, call <code>MultipartFile#getOriginalFilename</code> method and decide on the basis of whether file name is specified or not.</p> <p>In the above example, input validation error is thrown if the files are not selected.</p>
(5)	<p>It is checked whether an empty file is selected.</p> <p>To check if the selected file is not empty, call <code>MultipartFile#isEmpty</code> method to check for presence of contents.</p> <p>In the above example, input validation error is thrown if an empty file is selected.</p>
(6)	<p>It is checked whether the file size is within allowable range.</p> <p>To check the size of selected file, call <code>MultipartFile#getSize</code> method and check whether the size is within the allowable range.</p> <p>In the above example, input validation error is thrown if the file size exceeds the allowable range.</p>
(7)	<p>Implement upload process.</p> <p>The above example does not cover any specific implementation; however process to store the file on a shared disk or database is performed.</p>
(8)	As per the requirement, the processing result message notifying about successful upload is stored.
5.17. File Upload	1155
(9)	Once upload is complete, redirect to upload completion screen.

Note: Preventing duplicate upload

When uploading files, it is recommended to perform transaction token check and screen transition based on PRG pattern. With this, upload of same files caused due to double submission can be prevented.

For more details on how to prevent double submission, refer to *Double Submit Protection*.

Note: About MultipartFile

Methods to operate the uploaded file are provided in `MultipartFile`. For details about using each method, refer to ‘JavaDoc <<http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/web/multipart/MultipartFile.html>> of `MultipartFile` class’.

Bean Validation of file upload

In the above implementation example, uploaded file is validated as a Controller process. However, here the uploaded file is validated using Bean Validation.

For validation details, refer to *Input Validation*.

Note: It is recommended to use Bean Validation since this makes maintenance of Controller processes easier.

Implementing validation to verify that the file is selected

```
// (1)
@Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = UploadFileRequiredValidator.class)
public @interface UploadFileRequired {
    String message() default "{com.examples.upload.UploadFileRequired.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    @Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    @interface List {
        UploadFileRequired[] value();
    }
}
```

```
// (2)
public class UploadFileRequiredValidator implements
    ConstraintValidator<UploadFileRequired, MultipartFile> {

    @Override
    public void initialize(UploadFileRequired constraint) {
    }

    @Override
    public boolean isValid(MultipartFile multipartFile,
        ConstraintValidatorContext context) {
        return multipartFile != null &&
            StringUtils.hasLength(multipartFile.getOriginalFilename());
    }

}
```

Sr. No.	Description
(1)	Create annotation to verify that the file is selected.
(2)	Create implementation class to verify that the file is selected.

Implementing validation to verify that the file is not empty

```
// (3)
@Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = UploadFileNotEmptyValidator.class)
public @interface UploadFileNotEmpty {
    String message() default "{com.examples.upload.UploadFileNotEmpty.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    @Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    @interface List {
        UploadFileNotEmpty[] value();
    }
}

// (4)
public class UploadFileNotEmptyValidator implements
    ConstraintValidator<UploadFileNotEmpty, MultipartFile> {
```

```
    @Override
    public void initialize(UploadFileNotEmpty constraint) {
    }

    @Override
    public boolean isValid(MultipartFile multipartFile,
        ConstraintValidatorContext context) {
        if (multipartFile == null ||
            !StringUtils.hasLength(multipartFile.getOriginalFilename())) {
            return true;
        }
        return !multipartFile.isEmpty();
    }

}
```

Sr. No.	Description
(3)	Create annotation to verify that the file is not empty.
(4)	Create implementation class to verify that the file is not empty.

Implementing validation to verify that file size is within allowable range

```
// (5)
@Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = UploadFileMaxSizeValidator.class)
public @interface UploadFileMaxSize {
    String message() default "{com.examples.upload.UploadFileMaxSize.message}";
    long value() default (1024 * 1024);
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    @Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    @interface List {
        UploadFileMaxSize[] value();
    }
}
```

```
// (6)
public class UploadFileMaxSizeValidator implements
    ConstraintValidator<UploadFileMaxSize, MultipartFile> {

    private UploadFileMaxSize constraint;

    @Override
    public void initialize(UploadFileMaxSize constraint) {
        this.constraint = constraint;
    }

    @Override
    public boolean isValid(MultipartFile multipartFile,
        ConstraintValidatorContext context) {
        if (constraint.value() < 0 || multipartFile == null) {
            return true;
        }
        return multipartFile.getSize() <= constraint.value();
    }

}
```

Sr. No.	Description
(5)	Create annotation to verify that the file size is within allowable range.
(6)	Create implementation class to verify that the file size is within allowable range.

Implementing form

```
public class FileUploadForm implements Serializable {

    // omitted

    // (7)
    @UploadFileRequired
    @UploadFileNotEmpty
    @UploadFileMaxSize
    private MultipartFile file;

    @NotNull
    @Size(min = 0, max = 100)
    private String description;

    // omitted getter/setter methods.
}
```

}

Sr. No.	Description
(7)	Assign annotation to <code>MultipartFile</code> field for validating uploaded file.

Implementing Controller

```
@RequestMapping(value = "uploadFile", method = RequestMethod.POST)
public String uploadFile(@Validated FileUploadForm form,
    BindingResult result, RedirectAttributes redirectAttributes) {

    // (8)
    if (result.hasErrors()) {
        return "article/uploadForm";
    }

    MultipartFile uploadFile = form.getFile();

    // omit processing of upload.

    redirectAttributes.addFlashAttribute(ResultMessages.success().add(
        "i.xx.at.0001"));

    return "redirect:/article/upload";
}
```

Sr. No.	Description
(8)	Validation result of uploaded file is stored in <code>BindingResult</code> .

Uploading multiple files

This section explains about simultaneously uploading multiple files.

In order to upload multiple files simultaneously, it is necessary to receive `org.springframework.web.multipart.MultipartFile` object by binding it to the form object.

The explanation that has already been covered under single file upload has been omitted to avoid duplication.

Files Upload

File to upload	<input type="button" value="Choose File"/> No file chosen
Description	<input type="text"/>
File to upload	<input type="button" value="Choose File"/> No file chosen
Description	<input type="text"/>
<input type="button" value="Upload"/>	

Implementing form

```
// (1)
public class FileUploadForm implements Serializable {

    // omitted

    @UploadFileRequired
    @UploadFileNotEmpty
    @UploadFileMaxSize
    private MultipartFile file;

    @NotNull
    @Size(min = 0, max = 100)
    private String description;

    // omitted getter/setter methods.

}
```

```
public class FilesUploadForm implements Serializable {

    // omitted

    @Valid // (2)
    private List<FileUploadForm> fileUploadForms; // (3)

    // omitted getter/setter methods.

}
```

Sr. No.	Description
(1)	Class that maintains the information of each file (uploaded file itself and related form fields). In the above example, form object that was originally created to explain about single file upload, is re-used.
(2)	To carry out input validation through Bean Validation for the object maintained in list, assign @Valid annotation.
(3)	Define the object that maintains information of each file (uploaded file itself and related form fields) as List property.

Note: When only files are to be uploaded, `MultipartFile` object can also be defined as List property; however, for input validation of uploaded files using Bean Validation, there is better compatibility if the object that maintains information of each file, is defined as List property.

Implementing JSP

```
<form:form
    action="${pageContext.request.contextPath}/article/uploadFiles" method="post"
    modelAttribute="filesUploadForm" enctype="multipart/form-data">
    <table>
        <tr>
            <th width="35%">File to upload</th>
            <td width="65%">
                <form:input type="file" path="fileUploadForms[0].file" /> <!-- (1) -->
                <form:errors path="fileUploadForms[0].file" />
            </td>
        </tr>
        <tr>
            <th width="35%">Description</th>
            <td width="65%">
                <form:input path="fileUploadForms[0].description" />
                <form:errors path="fileUploadForms[0].description" />
            </td>
        </tr>
    </table>
    <table>
        <tr>
            <th width="35%">File to upload</th>
            <td width="65%">
```

```

<form:input type="file" path="fileUploadForms[1].file" /> <!-- (1) -->
<form:errors path="fileUploadForms[1].file" />
</td>
</tr>
<tr>
<th width="35%">Description</th>
<td width="65%">
    <form:input path="fileUploadForms[1].description" />
    <form:errors path="fileUploadForms[1].description" />
</td>
</tr>
</table>
<div>
    <form:button>Upload</form:button>
</div>
</form:>

```

Sr. No.	Description
(1)	Specify the binding position of the uploaded file in List. Specify the binding position within List in []. Start position begins with 0.

Implementing Controller

```

@RequestMapping(value = "uploadFiles", method = RequestMethod.POST)
public String uploadFiles(@Validated FilesUploadForm form,
                           BindingResult result, RedirectAttributes redirectAttributes) {

    if (result.hasErrors()) {
        return "article/uploadForm";
    }

    // (1)
    for (FileUploadForm fileUploadForm : form.getFileUploadForms()) {

        MultipartFile uploadFile = fileUploadForm.getFile();

        // omit processing of upload.

    }

    redirectAttributes.addFlashAttribute(ResultMessages.success().add(
        "i.xx.at.0001"));

    return "redirect:/article/upload?complete";
}

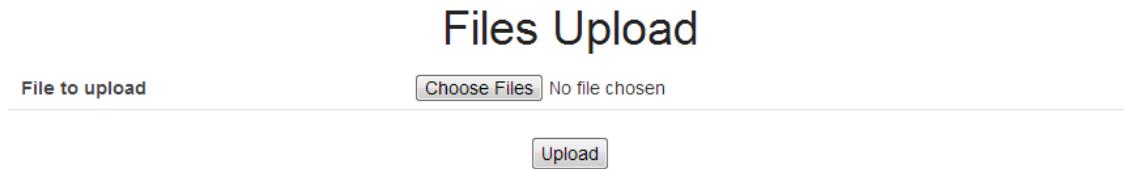
```

Table.5.25 :header-rows: 1 :widths: 10 90

Sr. No.	Description
(1)	<p>Fetch <code>MultipartFile</code> from the object that maintains information of each file (uploaded file itself and related form fields) and implement upload process.</p> <p>The above example does not cover any specific implementation; however process to store the file on a shared disk or database is performed.</p>

Uploading multiple files using the “multiple” attribute of HTML5

The method to simultaneously upload multiple files using “multiple” attribute of input tag supported by HTML5, is explained below.



The explanation that has already been covered under single file upload and multiple file upload has been omitted.

Implementing form

When uploading multiple files simultaneously using “multiple” attribute of HTML5 input tag, it is necessary to receive collection of `org.springframework.web.multipart.MultipartFile` object by binding it to form object.

```
// (1)
public class FilesUploadForm implements Serializable {

    // omitted

    // (2)
    @UploadFileNotEmpty
    private List<MultipartFile> files;

    // omitted getter/setter methods.

}
```

Sr. No.	Description
(1)	Form object that maintains the multiple uploaded files.
(2)	Declare <code>MultipartFile</code> class as list. In the above example, the annotation to verify that the file is not empty, is specified as input validation. Principally, a file size check or other mandatory checks are also required; however, they have been omitted in the above example.

Implementing Validator

When carrying out input validation for multiple `MultipartFile` objects stored in collection, it is necessary to implement Validator for Collection.

The section explains about creating Validator for Collection using the Validator created for single file.

```
// (1)
public class UploadFileNotEmptyForCollectionValidator implements
    ConstraintValidator<NotEmptyUploadFile, Collection<MultipartFile>> {

    // (2)
    private final UploadFileNotEmptyValidator validator =
        new UploadFileNotEmptyValidator();

    // (3)
    @Override
    public void initialize(NotEmptyUploadFile constraintAnnotation) {
        validator.initialize(constraintAnnotation);
    }

    // (4)
    @Override
    public boolean isValid(Collection<MultipartFile> values,
        ConstraintValidatorContext context) {
        for (MultipartFile file : values) {
            if (!validator.isValid(file, context)) {
                return false;
            }
        }
        return true;
    }
}
```

Sr. No.	Description
(1)	Class for performing implementation to verify that none of the files is empty. Specify Collection<MultipartFile> as the type of value to be verified.
(2)	In order to delegate the actual process to a Validator for single file, create an instance for that Validator.
(3)	Initialize the Validator. In the above example, Validator for single file that implements the actual process is initialized.
(4)	Verify that none of the file is empty. In the above example, each file is verified by calling the method of Validator for single file.

```

@Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy =
    {UploadFileNotEmptyValidator.class,
     UploadFileNotEmptyForCollectionValidator.class}) // (5)
public @interface UploadFileNotEmpty {

    // omitted

}

```

Sr. No.	Description
(5)	Add the Validator class that carries out checks with respect to multiple files, to the annotation used for verification. Specify the class created in step (1) in the “validatedBy” attribute of @Constraint annotation. With this, the class created in step (1) is executed when validating the property with @NotEmptyUploadFile annotation.

Implementing JSP

```
<form:form
    action="${pageContext.request.contextPath}/article/uploadFiles" method="post"
    modelAttribute="filesUploadForm2" enctype="multipart/form-data">
    <table>
        <tr>
            <th width="35%">File to upload</th>
            <td width="65%">
                <form:input type="file" path="files" multiple="multiple" /> <!-- (1) -->
                <form:errors path="files" />
            </td>
        </tr>
    </table>
    <div>
        <form:button>Upload</form:button>
    </div>
</form:form>
```

Sr. No.	Description
(1)	In “path” attribute, specify “multiple” attribute by indicating property name of form object. By specifying “multiple” attribute, multiple files can be selected and uploaded using browser supporting HTML5.

Implementing Controller

```
@RequestMapping(value = "uploadFiles", method = RequestMethod.POST)
public String uploadFiles(@Validated FilesUploadForm form,
    BindingResult result, RedirectAttributes redirectAttributes) {
    if (result.hasErrors()) {
        return "article/uploadForm";
    }

    // (1)
    for (MultipartFile file : form.getFiles()) {

        // omit processing of upload.

    }

    redirectAttributes.addFlashAttribute(ResultMessages.success().add(
        "i.xx.at.0001"));

    return "redirect:/article/upload?complete";
}
```

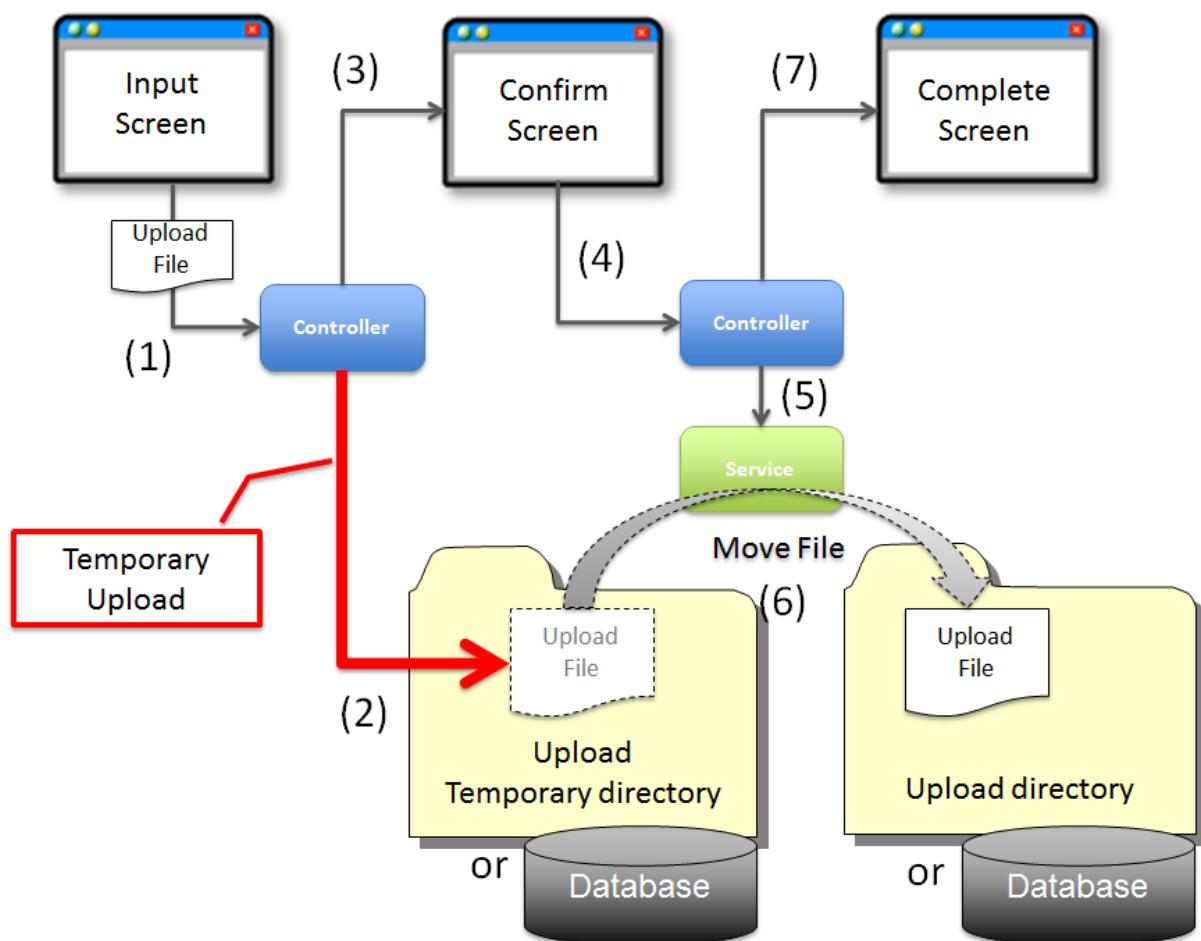
Sr. No.	Description
(1)	<p>Implement upload process by fetching the list which stores <code>MultipartFile</code> objects from form object.</p> <p>The above example does not cover any specific implementation; however process to store the file on a shared disk or database is performed.</p>

Temporary upload

Temporary upload is required when a file is to be uploaded midway through screen transitions like upload result confirmation screen etc.

Note: Contents of file stored in `MultipartFile` object may be deleted once the upload request is completed. Therefore, when the file contents are to be handled across requests, these contents and meta information (file name etc.) maintained in `MultipartFile` object need to be saved in a file or form.

The contents of file stored in `MultipartFile` object are deleted when step (3) of the following processing flow is completed.



Sr. No.	Description
(1)	On Input Screen, select the file to be uploaded and send a request for displaying Confirm Screen.
(2)	Controller temporarily saves contents of uploaded file in the temporary directory for application.
(3)	Controller returns View name of Confirm Screen and then displays the Confirm Screen.
(4)	On Confirm screen, send a request for executing the process.
5.17. ⁽⁵⁾File Upload	Controller calls Service method and executes process.
(6)	Service moves the temporary file saved in temporary directory to this directory or database.

Note: Temporary upload process is the responsibility of application layer; hence it is executed by Controller or Helper class.

Implementing Controller

Example for temporarily saving the uploaded file in a temporary directory, is shown below.

```
@Component
public class UploadHelper {

    // (2)
    @Value("${app.upload.temporaryDirectory}")
    private File uploadTemporaryDirectory;

    // (1)
    public String saveTemporaryFile(MultipartFile multipartFile)
        throws IOException {

        String uploadTemporary fileId = UUID.randomUUID().toString();
        File uploadTemporaryFile =
            new File(uploadTemporaryDirectory, uploadTemporary fileId);

        // (2)
        FileUtils.copyInputStreamToFile(multipartFile.getInputStream(),
            uploadTemporaryFile);

        return uploadTemporary fileId;
    }

}
```

Sr. No.	Description
(1)	Create a method for executing temporary upload in Helper class. When there are multiple processes that perform file upload, it is recommended to have a common temporary upload process by creating a common Helper method.
(2)	Save the uploaded file as a temporary file. In the above example, contents of uploaded file are saved to a file by calling copyInputStreamToFile method of org.apache.commons.io.FileUtils class.

```
// omitted

@Inject
UploadHelper uploadHelper;

@RequestMapping(value = "upload", method = RequestMethod.POST, params = "confirm")
public String uploadConfirm(@Validated FileUploadForm form,
                            BindingResult result) throws IOException {

    if (result.hasErrors()) {
        return "article/uploadForm";
    }

    // (3)
    String uploadTemporary fileId = uploadHelper.saveTemporaryFile(form
        .getFile());

    // (4)
    form.setUploadTemporary fileId(uploadTemporary fileId);
    form.setFileName(form.getFile().getOriginalFilename());

    return "article/uploadConfirm";
}

.. tabularcolumns:: |p{0.10\linewidth}|p{0.90\linewidth}|
.. list-table::
:header-rows: 1
:widths: 10 90

* - Sr. No.
  - Description
* - | (3)
  - | Call the Helper method to temporarily save the uploaded file.
    | In the above example, ID by which the temporarily saved file is identified, is returned
* - | (4)
  - | Save the meta information of uploaded file (ID by which the file is identified, file name)
    | In the above example, name of the uploaded file and ID by which the temporarily saved
```

Note: Directory of temporary directories should be fetched from external properties as it may differ with the environment in which the application is deployed. For details on external properties, refer to [Properties Management](#).

Warning: In the above example, it is a file saved temporarily on the local disk of application server. However, when the application server is clustered, it needs to be saved in the database or on a shared disk. As a result, it is necessary to design a storage destination by considering even the non-functional requirements.

Transaction management is necessary in case of saving the file to the database. As a result, the process to save it to the database will be delegated to Service method.

5.17.3 How to extend

Housekeeping of unnecessary files at the time of temporary upload

When uploading files using the temporary upload method, there is a possibility of unnecessary files piling up in temporary directory.

The cases are as follows:

- When there is interruption in screen operations after temporary upload
- When system error occurs during the screen operations after temporary upload
- When server stops during the screen operations after temporary upload etc ...

Warning: A mechanism should be provided to delete unnecessary files as the disk may run out of space if such files are left to pile up.

This guideline explains about deleting unnecessary files using the “Task Scheduler” functionality provided by Spring Framework. For details on “Task Scheduler”, refer to the [official website “Task Execution and Scheduling”](#).

Note: Although this guideline explains about how to use “Task Scheduler” functionality provided by Spring Framework; its usage is not mandatory. In an actual project, the infrastructure team may provide batch application (Shell application) to delete unnecessary files. In such cases, it is recommended to delete unnecessary files using the batch application created by infrastructure team.

Implementing component class to delete unnecessary files

Implement a component class to delete unnecessary files.

```
package com.examples.common.upload;

import java.io.File;
import java.util.Collection;
```

```
import java.util.Date;

import javax.inject.Inject;

import org.apache.commons.io.FileUtils;
import org.apache.commons.io.filefilter.FileFilterUtils;
import org.apache.commons.io.filefilter.IOFileFilter;
import org.springframework.beans.factory.annotation.Value;
import org.terasoluna.gfw.common.date.DateFactory;

// (1)
public class UnnecessaryFilesCleaner {

    @Inject
    DateFactory dateFactory;

    @Value("${app.upload.temporaryFileSavedPeriodMinutes}")
    private int savedPeriodMinutes;

    @Value("${app.upload.temporaryDirectory}")
    private File targetDirectory;

    // (2)
    public void cleanup() {

        // calculate cutoff date.
        Date cutoffDate = dateFactory.newDateTime().minusMinutes(
            savedPeriodMinutes).toDate();

        // collect target files.
        IOFileFilter fileFilter = FileFilterUtils.ageFileFilter(cutoffDate);
        Collection<File> targetFiles = FileUtils.listFiles(targetDirectory,
            fileFilter, null);

        if (targetFiles.isEmpty()) {
            return;
        }

        // delete files.
        for (File targetFile : targetFiles) {
            FileUtils.deleteQuietly(targetFile);
        }

    }
}
```

Sr. No.	Description
(1)	Create component class to delete unnecessary files.
(2)	Implement the method to delete unnecessary files. In the above example, the files that have not been updated for a certain period of time from the last update, are treated as unnecessary files and are deleted.

Note: Directory path in which files to be deleted are stored or the time criteria for deletion etc. may differ depending upon the environment in which application is to be deployed. Hence they should be fetched from external properties. For details on external properties, refer to *Properties Management*.

Scheduling settings of the process for deleting unnecessary files

Carry out bean registration and task schedule settings for the POJO class that deletes unnecessary files.

- applicationContext.xml

```
<!-- omitted -->

<!-- (3) -->
<bean id="uploadTemporaryFileCleaner"
      class="com.examples.common.upload.UnnecessaryFilesCleaner" />

<!-- (4) -->
<task:scheduler id="fileCleanupTaskScheduler" />

<!-- (5) -->
<task:scheduled-tasks scheduler="fileCleanupTaskScheduler">
    <!-- (6) (7) (8) -->
    <task:scheduled ref="uploadTemporaryFileCleaner"
                    method="cleanup"
                    cron="${app.upload.temporaryFilesCleaner.cron}" />
</task:scheduled-tasks>

<!-- omitted -->
```

Sr. No.	Description
(3)	POJO class that deletes unnecessary files should be registered in bean. In the above example, it is registered with "uploadTemporaryFileCleaner" ID.
(4)	Register the bean for task scheduler that executes the process to delete unnecessary files. In the above example, as pool-size attribute is omitted, this task scheduler executes the task in a single thread . When multiple tasks need to be executed simultaneously, some number should be specified in pool-size attribute.
(5)	Add the task to the task scheduler that deletes unnecessary files. In the above example, task is added to the task scheduler for which bean is registered in step (4).
(6)	In ref attribute, specify the bean that executes the process of deleting unnecessary files. In the above example, the bean registered in step (3) is specified.
(7)	In method attribute, specify the name of method executing the process of deleting unnecessary files. In the above example, cleanup method of bean registered in step (3) is specified.
(8)	In cron attribute, specify execution time of the process to delete unnecessary files. In the above example, cron definition is fetched from external properties.

Note: Specify the configuration value of cron attribute in “seconds minutes hour month year day” format.

Example:

- 0 */15 * * * : Executed in 0 minute, 15 minutes, 30 minutes and 45 minutes every hour.
- 0 0 * * * : Executed in 0 minute every hour.
- 0 0 9-17 * * MON-FRI : Executed in 0 minute every hour from 9:00~17:00 on weekdays.

For details regarding specified value of cron, refer to [CronSequenceGenerator - JavaDoc](#).

Execution time should be fetched from external properties as it may differ depending on the environment in which the application is to be deployed. For details on external properties, refer to [Properties Management](#).

Tip: In the above example, cron is used as a trigger for executing tasks. However, other triggers namely fixed-delay and fixed-rate are also set by default and should be selectively used as per requirement.

When the default triggers do not satisfy the requirements, an independent trigger can be set by specifying the bean implementing `org.springframework.scheduling.Trigger` in trigger attribute.

5.17.4 Appendix

Following security issues need to be considered when providing File Upload functionality.

1. *Dos attack with respect to upload functionality*
2. *Attack by executing uploaded files on Web Server*

Security measures are explained below.

Dos attack with respect to upload functionality

Dos attack with respect to upload functionality is when load on the server is increased by continuously uploading large files, thereby crashing the server or reducing its response speed.

When there is no limit on the size of files to be uploaded and multipart request, the resistance to Dos attack becomes weak.

In order to enhance the resistance towards Dos attack, size limit needs to be set for a request, by using `<multipart-config>` element explained in [file-upload_how_to_usr_application_settings](#).

Attack by executing uploaded files on Web Server

In this attack, the files on Web Server can be viewed/ altered/ deleted by uploading and executing the script files (php, asp, aspx, jsp etc.) that are executable on Web Server (Application Server).

With Web Server as a platform, another server present in the same network as the Web server, is also vulnerable to such attack.

Measures to be taken against this attack are as follows:

- To view the file contents through a process that displays the contents, without placing uploaded files in the public directory of Web Server (Application Server).
- To ensure that executable script file cannot be uploaded on Web server (Application Server) by restricting the extension of files that can be uploaded.

The attacks can be prevented by implementing either of the above measures; however it is always recommended to implement both the measures.

5.18 File Download

5.18.1 Overview

This chapter explains the functionality to download a file from server to client, using Spring.

It is recommended to use Spring MVC View for rendering the files.

Note: It is not recommended to include file rendering logic in controller class.

This is because, it deviates from the role of a controller. Moreover, a View can be easily changed when it is isolated from the controller.

Overview of File Download process is given below.

1. DispatchServlet sends file download request to the controller.
2. Controller fetches file display information.
3. Controller selects View.
4. File rendering is performed in View.

In order to perform file rendering in a Spring based Web application;

this guideline recommends implementation of custom view.

To implement custom view, `org.springframework.web.servlet.View` interface is provided in Spring framework.

For PDF files

`org.springframework.web.servlet.view.document.AbstractPdfView` class of Spring is used as a subclass when rendering PDF files using model information.

For Excel files

`org.springframework.web.servlet.view.document.AbstractExcelView` class of Spring is used as a subclass when rendering Excel files using model information.

For file formats other than those specified above, various types of View implementations are provided in Spring.

For technical details on View, refer to [Spring Reference View technologies](#).

`org.terasoluna.gfw.web.download.AbstractFileDownloadView` provided by common library is the

abstract class to download arbitrary files.

Define this class as a subclass to render files in formats other than PDF or Excel.

5.18.2 How to use

Downloading PDF files

For rendering PDF files, it is necessary to create a class that

inherits `org.springframework.web.servlet.view.document.AbstractPdfView` provided by Spring.

The procedure to download a PDF file using controller is explained below.

Implementation of Custom View

Implementation of class that inherits AbstractPdfView

```
@Component // (1)
public class SamplePdfView extends AbstractPdfView { // (2)

    @Override
    protected void buildPdfDocument(Map<String, Object> model,
        Document document, PdfWriter writer, HttpServletRequest request,
        HttpServletResponse response) throws Exception { // (3)

        document.add(new Paragraph((Date) model.get("serverTime")).toString());
    }
}
```

Sr. No.	Description
(1)	In this example, this class comes under the scope of component scanning by using @Component annotation. It will also come under the scope of org.springframework.web.servlet.view.BeanNameViewResolver which is described later.
(2)	Inherit AbstractPdfView.
(3)	Execute buildPdfDocument method.

AbstractPdfView uses iText for PDF rendering.

Therefore, it is necessary to add itext definition to pom.xml of Maven.

```
<dependencies>
    <!-- omitted -->
    <dependency>
        <groupId>com.lowagie</groupId>
        <artifactId>iText</artifactId>
        <version>${com.lowagie.itext.version}</version>
        <exclusions>
            <exclusion>
                <artifactId>xml-apis</artifactId>
                <groupId>xml-apis</groupId>
            </exclusion>
            <exclusion>
                <artifactId>bctsp-jdk14</artifactId>
                <groupId>org.bouncycastle</groupId>
            </exclusion>
            <exclusion>
                <artifactId>jfreechart</artifactId>
                <groupId>jfree</groupId>
            </exclusion>
            <exclusion>
                <artifactId>dom4j</artifactId>
                <groupId>dom4j</groupId>
            </exclusion>
            <exclusion>
                <groupId>org.swinglabs</groupId>
```

```
        <artifactId>pdf-renderer</artifactId>
    </exclusion>
</exclusions>
</dependency>
</dependencies>

<properties>
    <!-- omitted -->
    <com.lowagie.itext.version>4.2.1</com.lowagie.itext.version>
</properties>
```

Note: Spring 3.2 does not support iText version 5.

Definition of ViewResolver

`org.springframework.web.servlet.view.BeanNameViewResolver` is the class stored in Spring context. It selects the View to be executed on the basis of bean name.

Normally `InternalResourceViewResolver` is used; however while using `BeanNameViewResolver`, it should be called before `InternalResourceViewResolver` by setting `order` property.

Note: Spring provides various types of View Resolvers and it allows chaining of multiple Resolvers; hence some unintended View may get selected under certain conditions. To avoid such a situation, priority can be set by specifying `order` property. Lower the specified value of `order` property, earlier it is executed.

bean definition file

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>

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

Sr. No.	Description
(1)	Define BeanNameViewResolver.
(2)	Set 0 in order property. Priority should be higher than that of InternalResourceViewResolver.

Specifying View in controller

With the help of BeanNameViewResolver, by returning “samplePDFView” in Controller, a view named “samplePDFView” gets used from the BeanIDs stored in Spring Context.

Java source code

```
@RequestMapping(value = "home", params= "pdf", method = RequestMethod.GET)
public String homePdf(Model model) {
    model.addAttribute("serverTime", new Date());
    return "samplePdfView"; // (1)
}
```

Sr. No.	Description
(1)	With “samplePdfView” as the return value of method, SamplePdfView class stored in Spring context is executed.

Following PDF file can be opened after executing the above procedure.

Downloading Excel files

For rendering EXCEL files, it is necessary to create a class that

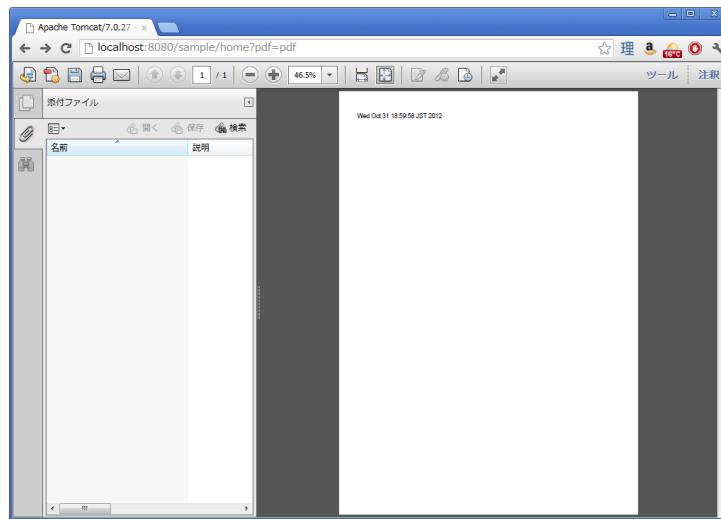


Figure.5.45 Picture - FileDownload PDF

inherits `org.springframework.web.servlet.view.document.AbstractExcelView` provided by Spring.

The procedure to download an EXCEL file using controller is explained below.

Implementation of Custom View

Implementation of class that inherits AbstractExcelView

```
@Component // (1)
public class SampleExcelView extends AbstractExcelView { // (2)

    @Override
    protected void buildExcelDocument(Map<String, Object> model,
        HSSFWorkbook workbook, HttpServletRequest request,
        HttpServletResponse response) throws Exception { // (3)
        HSSFSheet sheet;
        HSSFCell cell;

        sheet = workbook.createSheet("Spring");
        sheet.setDefaultColumnWidth(12);

        // write a text at A1
        cell = getCell(sheet, 0, 0);
        setText(cell, "Spring-Excel test");

        cell = getCell(sheet, 2, 0);
        setText(cell, (Date) model.get("serverTime")).toString();
    }
}
```

Sr. No.	Description
(1)	In this example, this class comes under the scope of component scanning by using @Component annotation. It will also come under the scope of org.springframework.web.servlet.view.BeanNameViewResolver which is described earlier.
(2)	Inherit AbstractExcelView.
(3)	Execute buildExcelDocument method.

AbstractExcelView uses [Apache POI](#) to render EXCEL file.

Therefore, it is necessary to add POI definition to the pom.xml file of Maven.

```
<dependencies>
    <!-- omitted -->
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>${org.apache.poi.poi.version}</version>
    </dependency>
</dependencies>

<properties>
    <!-- omitted -->
    <org.apache.poi.poi.version>3.9</org.apache.poi.poi.version>
</properties>
```

Definition of ViewResolver

Settings are same as that for PDF file rendering. For details, refer to [Definition of ViewResolver](#).

Specifying View in controller

With the help of BeanNameViewResolver, by returning “sampleExcelView” in Controller, a view named “sampleExcelView” gets used from the BeanIDs stored in Spring Context.

Java source

```
@RequestMapping(value = "home", params= "excel", method = RequestMethod.GET)
public String homeExcel(Model model) {
    model.addAttribute("serverTime", new Date());
    return "sampleExcelView"; // (1)
}
```

Sr. No.	Description
(1)	With “sampleExcelView”as the return value of method, SampleExcelView class stored in Spring context is executed.

EXCEL file can be opened as shown below after executing the above procedures.

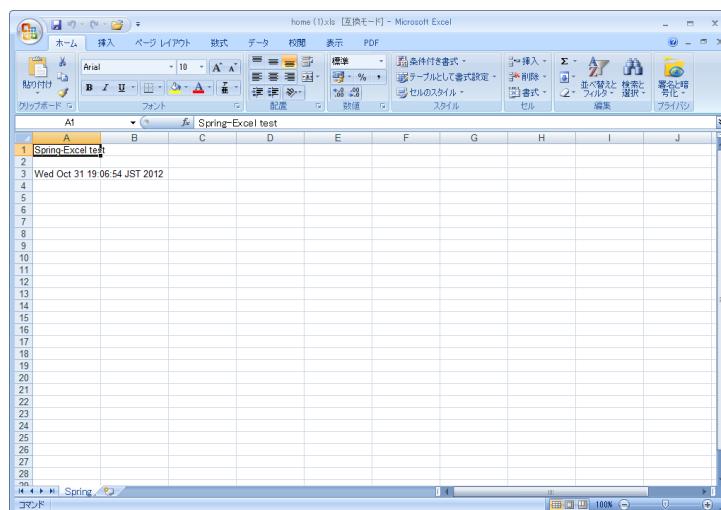


Figure.5.46 Picture - FileDownload EXCEL

Downloading arbitrary files

To download files in formats other than PDF or EXCEL, class that inherits `org.terasoluna.gfw.web.download.AbstractFileDownloadView` provided by common library can be implemented.

Following steps should be implemented in `AbstractFileDownloadView` to render files in other format.

1. Fetch InputStream in order to write to the response body.
2. Set information in HTTP header.

How to implement file download using controller is explained below.

Implementation of Custom View

The example of text file download is given below.

Implementation of class that inherits AbstractFileDialogView

```
@Component // (1)
public class TextFileDialogView extends AbstractFileDialogView { // (2)

    @Override
    protected InputStream getInputStream(Map<String, Object> model,
                                         HttpServletRequest request) throws IOException { // (3)
        Resource resource = new ClassPathResource("abc.txt");
        return resource.getInputStream();
    }

    @Override
    protected void addResponseHeader(Map<String, Object> model,
                                     HttpServletRequest request, HttpServletResponse response) { // (4)
        response.setHeader("Content-Disposition",
                           "attachment; filename=abc.txt");
        response.setContentType("text/plain");
    }
}
```

Sr. No.	Description
(1)	In this example, this class comes under the scope of component scanning by using @Component annotation. It will also come under the scope of org.springframework.web.servlet.view.BeanNameViewResolver which is described earlier.
(2)	Inherit AbstractFileDownloadView.
(3)	Execute getInputStream method. InputStream to be downloaded should be returned.
(4)	Execute addResponseHeader method. Set Content-Disposition or ContentType as per the file to be downloaded.

Definition of ViewResolver

Setting is same as that of PDF file rendering. For details, refer to [Definition of ViewResolver](#).

Specifying View in controller

With the help of BeanNameViewResolver, by returning “textFileDownloadView” in Controller, a view named “textFileDownloadView” gets used from the BeanIDs stored in Spring Context.

Java source

```
@RequestMapping(value = "download", method = RequestMethod.GET)
public String download() {
    return "textFileDownloadView"; // (1)
}
```

Sr. No.	Description
(1)	With “textFileDownloadView” as the return value of method, TextFileDownloadView class stored in Spring context is executed.

Tip: As described above, Model information can be rendered in various types of Views using Spring. Spring supports rendering engine such as Jasper Reports and returns various types of views. For details, refer to the official Spring website [Spring reference](#).

5.19 Screen Layout using Tiles

5.19.1 Overview

When developing a Web application with common layouts such as header, footer and side menu, maintaining the layouts becomes complicated if the common parts are coded in all JSPs.

For example, if the header design needs to be modified, the same modifications must be done for all JSPs.

In JSP development, when the same layout is used in many screens, it is recommended to use [Apache Tiles](#) (hereafter referred to as Tiles).

Reasons for using Tiles are as follows:

1. To eliminate layout errors by designer
2. To reduce redundant codes
3. To change oversized layouts easily

Tiles can combine different JSPs by defining an integrated screen layout.

As a result, the need to describe extra code in each JSP file is eliminated, thereby facilitating developer operations.

For example, if multiple screens have the following layout structure,

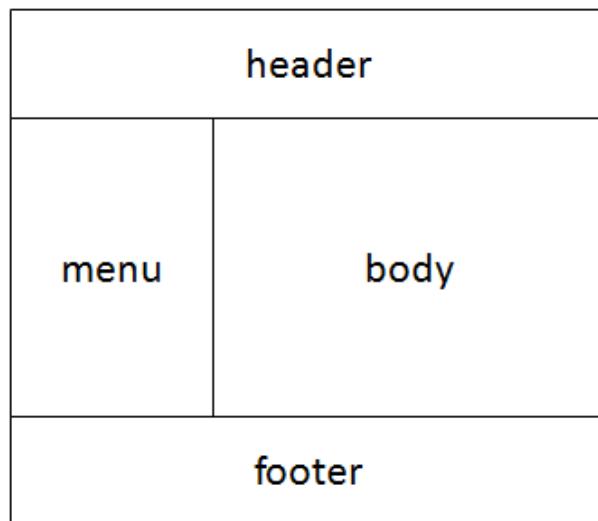


Figure.5.47 Picture - Image of screen layout

By using Tiles, one can focus only on creating the body without having to include and specify the sizes of header, menu and footer, in all the screens with the same layout.

Actual JSP file is as follows:

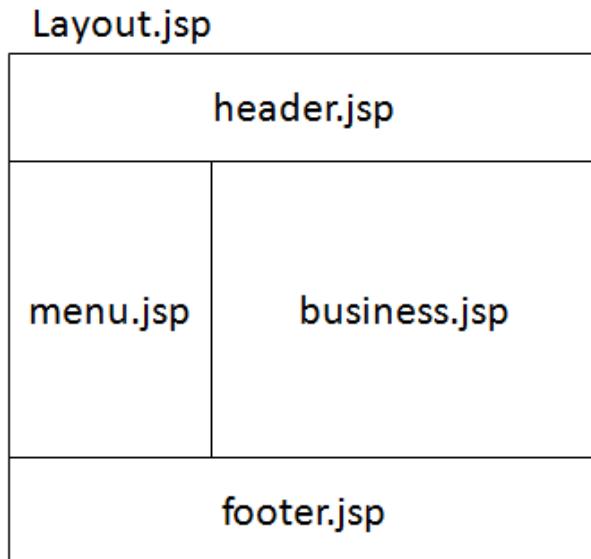


Figure.5.48 Picture - Image of layout jsp

Therefore, after configuring the screen layout using Tiles, only the JSP file corresponding to business process (**business.jsp**) may be created for each screen.

Note: In some cases, it is better to avoid using Tiles. For example, using Tiles in an error screen is not recommended due to the following reasons.

- If an error occurs due to Tiles during error screen display, analyzing the errors becomes difficult. (In case of double failure)
 - Tiles Template is not necessarily always used to display screens in the JSP set by the `<error-pages>` tag of web.xml.
-

5.19.2 How to use

pom.xml setting

To use Tiles in Maven, following dependency should be added to pom.xml.

```
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-recommended-web-dependencies</artifactId><!-- (1) -->
    <type>pom</type><!-- (2) -->
</dependency>
```

Sr. No.	Description
(1)	Add terasoluna-gfw-recommended-web-dependencies defined for the group of web related libraries, to dependency.
(2)	Dependencies such as terasoluna-gfw-recommended-web-dependencies are defined only in pom file; hence <type>pom</type> needs to be specified.

Note: It is assumed that pom.xml has the following terasoluna-gfw-parent settings.

```
<parent>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-parent</artifactId>
    <version>x.y.z</version>
</parent>
```

Therefore, the <version> of terasoluna-gfw-recommended-web-dependencies need not be specified.

Integration of Spring MVC and Tiles

`org.springframework.web.servlet.view.tiles2.TilesViewResolver` can be used to integrate Spring MVC and Tiles.

Implementation of Spring MVC Controller (returning View name) need not be changed.

How to configure is shown below.

Defining Bean (ViewResolver, TilesConfigurer)

- `spring-mvc.xml`

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

<bean id="tilesViewResolver"
    class="org.springframework.web.servlet.view.tiles2.TilesViewResolver"> <!-- (2) -->
    <property name="order" value="1" />
</bean>

<bean id="tilesConfigurer"
    class="org.springframework.web.servlet.view.tiles2.TilesConfigurer"> <!-- (3) -->
    <property name="definitions">
        <list>
            <value>/WEB-INF/tiles/tiles-definitions.xml</value>
        </list>
    </property>
</bean>
```

Sr. No.	Description
(1)	<p>Define InternalResourceViewResolver.</p> <p>jsp file saved under /WEB-INF/views/ is considered as the target.</p> <p>When it does not correspond to the configuration file tiles-definitions.xml that defines Tiles, by specifying “2” in ‘order’ of property definition, define the priority order so as to use org.springframework.web.servlet.view.InternalResourceViewResolver.</p> <p>When file name of the jsp which is read does not match with the name attribute pattern of definition tag in Tiles definition file,</p> <p>org.springframework.web.servlet.view.InternalResourceViewResolver is used.</p>
(2)	<p>Define TilesViewResolver.</p> <p>Specify “1” in ‘order’ of property definition and define the priority so that configuration file tiles-definitions.xml is used first.</p> <p>When file name of the jsp which is read matches with the name attribute pattern of definition tag in Tiles definition file,</p> <p>org.springframework.web.servlet.view.tiles2.TilesViewResolver is used.</p>
(3)	<p>Specify the location of Tiles definition file.</p> <p>Read /WEB-INF/tiles/tiles-definitions.xml.</p>

Tiles Definition

- tiles-definitions.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
"http://tiles.apache.org/dtds/tiles-config_2_1.dtd"> <!-- (1) -->

<tiles-definitions>
  <definition name="layouts"
    template="/WEB-INF/views/layout/template.jsp"> <!-- (2) -->
    <put-attribute name="header"
      value="/WEB-INF/views/layout/header.jsp" /> <!-- (3) -->
    <put-attribute name="footer"
      value="/WEB-INF/views/layout/footer.jsp" /> <!-- (4) -->
  </definition>

```

```
<definition name="*/*" extends="layouts"> <!-- (5) -->
  <put-attribute name="title" value="title.{1}.{2}" /> <!-- (6) -->
  <put-attribute name="body" value="/WEB-INF/views/{1}/{2}.jsp" /> <!-- (7) -->
</definition>
</tiles-definitions>
```

Sr. No.	Description
(1)	Define dtd of tiles.
(2)	Define the parent layout structure. In ‘template’ attribute, specify the jsp file where layout is defined.
(3)	Specify the jsp file that defines header.
(4)	Specify the jsp file that defines footer.
(5)	Layout definition which is called when it is same as name pattern at the time of ‘create’ request. Extended layouts definition is also applied.
(6)	Specify title. Fetch the value from properties incorporated in spring-mvc. (In the following description, it is set in application-messages.properties.) {1},{2} correspond to the 1st and 2nd “*” of “*/*” request.
(7)	Design the location of jsp file that defines the body such that, request path matches with {1} and JSP name matches with {2}. With this, the efforts to describe definition for each request can be saved.

Note: For the screens where Tiles is not to be applied (error screen etc.), it is necessary to set a file structure that excludes use of Tiles. In Blank project, /WEB-INF/views/common/error/xxxError.jsp format is used so that InternalResourceViewResolver can be used (and so that it does not change to the “*/*” format) on

error screen.

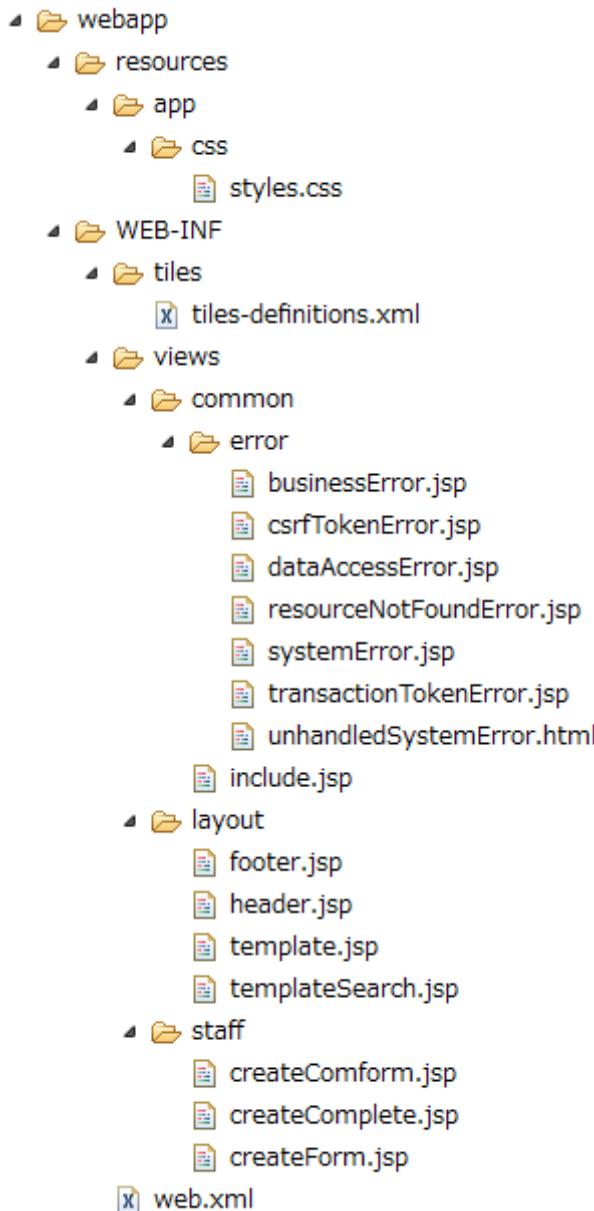
- *application-messages.properties*

```
title.staff.createForm = Create Staff Information
```

Note: For details on message properties file, refer to [Message Management](#).

Following is the file structure when Tiles is set.

- tiles File Path



Custom tag settings

Custom tag (TLD) needs to be set to use Tiles.

- /WEB-INF/views/common/include.jsp

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

Sr. No.	Description
(1)	Add the custom tag (TLD) definition for Tiles.

- web.xml

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>false</el-ignored>
    <page-encoding>UTF-8</page-encoding>
    <scripting-invalid>false</scripting-invalid>
    <include-prelude>/WEB-INF/views/common/include.jsp</include-prelude> <!-- (1) -->
  </jsp-property-group>
</jsp-config>
```

Sr. No.	Description
(1)	Based on web.xml settings, when jsp file (~.jsp) is to be read, include.jsp can be read in advance.

Note: Custom tag can also be set in template.jsp. However, it is recommended to create custom tag definition in common jsp include file. For details, refer to [Creating common JSP for include](#).

Creating layout

Create jsp (template) that forms frame of a layout and jsp to be embedded in the layout.

- template.jsp

```
<!DOCTYPE html>
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>    <html class="no-js lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]>    <html class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!-->
<html class="no-js">
<!--<![endif]-->
<head>
<meta charset="utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
<meta name="viewport" content="width=device-width" />
<link rel="stylesheet"
      href="${pageContext.request.contextPath}/resources/app/css/styles.css"
      type="text/css" media="screen, projection">
<script type="text/javascript">

</script> <!-- (1) -->
<c:set var="titleKey"> <!-- (2) -->
    <tiles:insertAttribute name="title" ignore="true" />
</c:set>
<title><spring:message code="${titleKey}" text="Create Staff Information" /></title><!-- (3)
</head>
<body>
    <div id="header">
        <tiles:insertAttribute name="header" /> <!-- (4) -->
    </div>
    <div id="body">
        <tiles:insertAttribute name="body" /> <!-- (5) -->
    </div>
    <div id="footer">
        <tiles:insertAttribute name="footer" /> <!-- (6) -->
    </div>
</body>
</html>
```

Sr. No.	Description
(1)	Mention the common contents that need to be described, above step (1).
(2)	Fetch the value of <code>title</code> specified in step (6) of tiles-definitions.xml and set it to <code>titleKey</code> .
(3)	Set title. When <code>titleKey</code> cannot be fetched, display the title defined in <code>text</code> attribute.
(4)	Read the “header” defined in tiles-definitions.xml.
(5)	Read the “body” defined in tiles-definitions.xml.
(6)	Read the “footer” defined in tiles-definitions.xml.

- header.jsp

```
<h1>
    <a href="${pageContext.request.contextPath}">Staff Management
        System</a>
</h1>
```

- createForm.jsp(example of body section)

The developer is able to focus only on the body section and describe the same without having to mention the extra source code for header and footer.

```
<h2>Create Staff Information</h2>
<table>
    <tr>
        <td>Staff First Name</td>
        <td><input type="text" /></td>
    </tr>
    <tr>
        <td>Staff Family Name</td>
        <td><input type="text" /></td>
    </tr>
    <tr>
```

```

<td rowspan="5">Staff Authorities</td>
<td><input type="checkbox" name="sa" value="01" /> Staff
Management</td>
</tr>
<tr>
<td><input type="checkbox" name="sa" value="02" /> Master
Management</td>
</tr>
<tr>
<td><input type="checkbox" name="sa" value="03" /> Stock
Management</td>
</tr>
<tr>
<td><input type="checkbox" name="sa" value="04" /> Order
Management</td>
</tr>
<tr>
<td><input type="checkbox" name="sa" value="05" /> Show Shopping
Management</td>
</tr>
</table>

<input type="submit" value="cancel" />
<input type="submit" value="confirm" />

```

- footer.jsp

```
<p style="text-align: center; background: #e5eCf9;">Copyright &copy;
20XX CompanyName</p>
```

Creating Controller

While creating Controller, when the request is <contextPath>/staff/create?form, perform the settings such that “staff/createForm” is returned from the Controller.

- StaffCreateController.java

```
@RequestMapping(value = "create", method = RequestMethod.GET, params = "form")
public String createForm() {
    return "staff/createForm"; // (1)
}
```

Sr. No.	Description
(1)	With staff as {1} and createForm as {2}, fetch the title name from properties and identify the JSP.

Creating screen

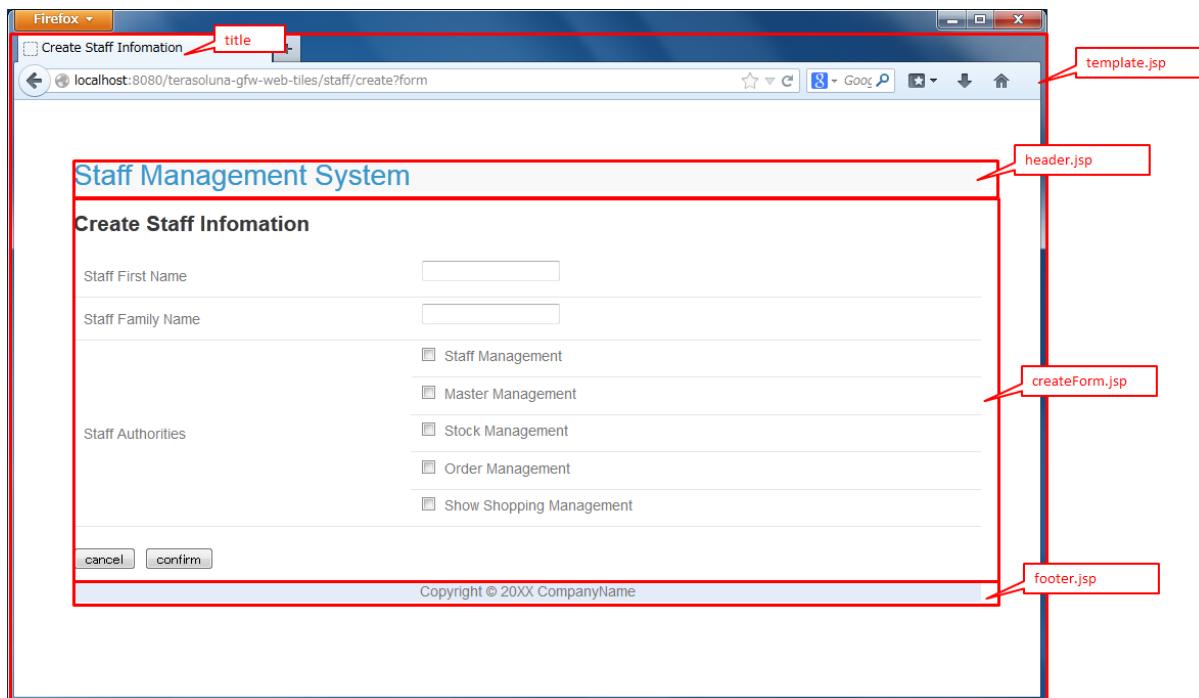
When <contextPath>/staff/create?form is called in request, Tiles constructs the layout and creates screen, as shown below.

```
<definition name="layouts"
    template="/WEB-INF/views/layout/template.jsp"> <!-- (1) -->
    <put-attribute name="header"
        value="/WEB-INF/views/layout/header.jsp" /> <!-- (2) -->
    <put-attribute name="footer"
        value="/WEB-INF/views/layout/footer.jsp" /> <!-- (3) -->
</definition>

<definition name="*/*" extends="layouts">
    <put-attribute name="title" value="title.{1}.{2}" /> <!-- (4) -->
    <put-attribute name="body"
        value="/WEB-INF/views/{1}/{2}.jsp" /> <!-- (5) -->
</definition>
```

Sr. No.	Description
(1)	In case of corresponding request, “layouts” which is a parent layout is called and template is set to /WEB-INF/views/layout/template.jsp.
(2)	/WEB-INF/views/layout/header.jsp is set in header within the template /WEB-INF/views/layout/template.jsp.
(3)	/WEB-INF/views/layout/footer.jsp is set in footer within the template /WEB-INF/views/layout/template.jsp.
(4)	With title.staff.createForm as key, fetch the value from properties incorporated in spring-mvc where staff is {1} and createForm is {2}.
(5)	/WEB-INF/views/staff/createForm.jsp is set in body within template /WEB-INF/views/layout/template.jsp with staff as {1} and createForm as {2}.

As a result, it is output to the browser by combining header.jsp, createForm.jsp and footer.jsp in the above template.jsp.



5.19.3 How to extend

Setting multiple layouts

When creating actual business application, display layout may be divided depending on business process contents.

This time, it is assumed that the staff search functionality menu is required to be displayed on left side of the screen.

Configuration is shown below based on :ref:`How-To-Use`.

Tiles Definition

- tiles-definitions.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
"http://tiles.apache.org/dtds/tiles-config_2_1.dtd">

<tiles-definitions>
<definition name="layoutsOfSearch"
template="/WEB-INF/views/layout/templateSearch.jsp"> <!-- (1) -->
<put-attribute name="header"
```

```
        value="/WEB-INF/views/layout/header.jsp" />
<put-attribute name="menu"
               value="/WEB-INF/views/layout/menu.jsp" />
<put-attribute name="footer"
               value="/WEB-INF/views/layout/footer.jsp" />
</definition>

<definition name="*/search*" extends="layoutsOfSearch"> <!-- (2) -->
<put-attribute name="title" value="title.{1}.search{2}" /> <!-- (3) -->
<put-attribute name="body" value="/WEB-INF/views/{1}/search{2}.jsp" /> <!-- (4) -->
</definition>

<definition name="layouts"
            template="/WEB-INF/views/layout/template.jsp">
<put-attribute name="header"
               value="/WEB-INF/views/layout/header.jsp" />
<put-attribute name="footer"
               value="/WEB-INF/views/layout/footer.jsp" />
</definition>

<definition name="*/*" extends="layouts">
<put-attribute name="title" value="title.{1}.{2}" />
<put-attribute name="body" value="/WEB-INF/views/{1}/{2}.jsp" />
</definition>
</tiles-definitions>
```

Sr. No.	Description
(1)	<p>Define the parent layout structure to be added.</p> <p>When using a different layout, ensure that name attribute of definition tag does not duplicate with the existing layout definition i.e. “layouts”.</p>
(2)	<p>Layout definition called when the layout to be added is same as the name pattern at the time of ‘create’ request.</p> <p>This layout definition is read when the request corresponds to <contextPath>/*/search*. Extended layout definition “layoutsOfSearch” is also applied.</p>
(3)	<p>Specify the title to be used in the layout to be added.</p> <p>Fetch the value from properties incorporated in spring-mvc. (In the following description, it is set to application-messages.properties.)</p> <p>{1} is the 1st “*” of “*/search*” of request.</p> <p>It is necessary that {2} starts with “search” as it corresponds to the “search*” of “*/search*” request.</p>
(4)	<p>Place the jsp file in which the body is defined such that, the request path matches with {1} and JSP file name beginning with “search”, matches with {2}.</p> <p>The value of ‘value’ attribute needs to be changed according to the configuration of JSP file location.</p>

Note: When multiple requests correspond to name attribute patterns of definition tag, the verification is done sequentially from the top and the very first pattern that matches with the request is applied. In the above case, as the request for staff search screen corresponds to multiple patterns, the layout is defined at the top.

- *application-messages.properties*

```
title.staff.createForm = Create Staff Information
title.staff.searchStaff = Search Staff Information # (1)
```

Sr. No.	Description
(1)	<p>Message to be added.</p> <p>“staff” is the 1st “*” of “*/search*” request.</p> <p>As “searchStaff” corresponds to “search*” part of “*/search*” request, it is necessary that it begins with “search”.</p>

Creating layout

Create the jsp (template) that forms the frame of the layout and jsp to be embedded in layout.

- templateSearch.jsp

```
<!DOCTYPE html>
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>     <html class="no-js lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]>     <html class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!-->
<html class="no-js">
<!--<![endif]-->
<head>
<meta charset="utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
<meta name="viewport" content="width=device-width" />
<link rel="stylesheet"
      href="${pageContext.request.contextPath}/resources/app/css/styles.css"
      type="text/css" media="screen, projection">
<script type="text/javascript">

</script>
<c:set var="titleKey">
    <tiles:insertAttribute name="title" ignore="true" />
</c:set>
<title><spring:message code="${titleKey}" text="Search Staff Information" /></title>
</head>
<body>
    <div id="header">
        <tiles:insertAttribute name="header" />
    </div>
    <div id="menu">
        <tiles:insertAttribute name="menu" /> <!-- (1) -->
    </div>
    <div id="body">
        <tiles:insertAttribute name="body" />
    </div>
    <div id="footer">
        <tiles:insertAttribute name="footer" />
    </div>
</body>
```

</html>

Sr. No.	Description
(1)	Read the “menu” defined in tiles-definitions.xml. Rest is same as <i>How to use</i> .

- styles.css

```
div#menu { /* (1) */
    float: left;
    width: 20%;
}

div#searchBody { /* (2) */
    float: right;
    width: 80%;
}

div#footer { /* (3) */
    clear: both;
}
```

Sr. No.	Description
(1)	Set the Menu style. Here, Menu Screen is left aligned using float:left and is displayed with 20% width.
(2)	Set the Body style. Here, the Business Screen is right aligned using float:right and displayed with 80% width. Name is specified as searchBody. This is because duplication in existing layout and name can have an impact on the existing layout style.
(3)	Set the Footer style. Float effect of menu and body is initialized. By this, the footer is displayed below menu and body.

- header.jsp

Same as *How to use*.

- menu.jsp

```
<table>
  <tr>
    <td><a href="${pageContext.request.contextPath}/staff/create?form">Create Staff Info</a>
  </tr>
  <tr>
    <td><a href="${pageContext.request.contextPath}/staff/search">Search Staff Information</a>
  </tr>
</table>
```

- searchStaff.jsp (example of body section)

```
<h2>Search Staff Information</h2>
<table>
  <tr>
    <td>Staff First Name</td>
    <td><input type="text" /></td>
  </tr>
  <tr>
    <td>Staff Family Name</td>
    <td><input type="text" /></td>
  </tr>
  <tr>
    <td rowspan="5">Staff Authorities</td>
    <td><input type="checkbox" name="sa" value="01" /> Staff Management</td>
  </tr>
  <tr>
    <td><input type="checkbox" name="sa" value="02" /> Master Management</td>
  </tr>
  <tr>
    <td><input type="checkbox" name="sa" value="03" /> Stock Management</td>
  </tr>
  <tr>
    <td><input type="checkbox" name="sa" value="04" /> Order Management</td>
  </tr>
  <tr>
    <td><input type="checkbox" name="sa" value="05" /> Show Shopping Management</td>
  </tr>
</table>

<input type="submit" value="Search" />
```

- footer.jsp

Same as *How to use*.

Creating Controller

While creating Controller, if the request is <contextPath>/staff/search, set such that “staff/searchStaff” is returned from the Controller.

- StaffSearchController.java

```
@RequestMapping(value = "search", method = RequestMethod.GET)
public String createForm() {
    return "staff/searchStaff"; // (1)
}
```

Sr. No.	Description
(1)	With staff as {1} and searchStaff as {2}, fetch the title name from properties and identify the JSP.

Creating screen

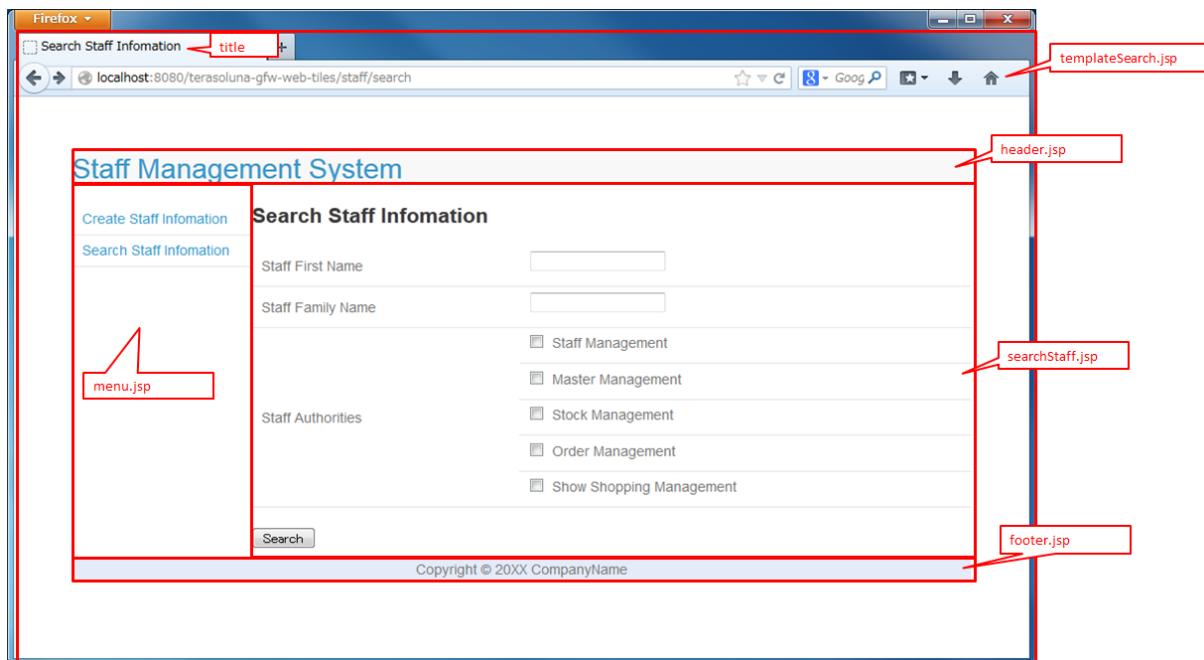
When <contextPath>/staff/search is called in request, screen is generated through another layout as shown below.

```
<definition name="layoutsOfSearch"
    template="/WEB-INF/views/layout/templateSearch.jsp"> <!-- (1) -->
    <put-attribute name="header"
        value="/WEB-INF/views/layout/header.jsp" /> <!-- (2) -->
    <put-attribute name="menu"
        value="/WEB-INF/views/layout/menu.jsp" /> <!-- (3) -->
    <put-attribute name="footer"
        value="/WEB-INF/views/layout/footer.jsp" /> <!-- (4) -->
</definition>

<definition name="*/search*" extends="layoutsOfSearch"> <!-- (5) -->
    <put-attribute name="title" value="title.{1}.search{2}" /> <!-- (6) -->
    <put-attribute name="body" value="/WEB-INF/views/{1}/search{2}.jsp" /> <!-- (7) -->
</definition>
```

Sr. No.	Description
(1)	In case of a corresponding request, “layoutsOfSearch” which is a parent layout is called and template is set in /WEB-INF/views/layout/templateSearch.jsp.
(2)	/WEB-INF/views/layout/header.jsp is set in header within the template /WEB-INF/views/layout/templateSearch.jsp.
(3)	/WEB-INF/views/layout/menu.jsp is set in menu within the template /WEB-INF/views/layout/templateSearch.jsp.
(4)	/WEB-INF/views/layout/footer.jsp is set in footer within the template /WEB-INF/views/layout/templateSearch.jsp.
(5)	This layout definition is read when the request corresponds to <contextPath>/*/search*. In that case, “layoutsOfSearch” which is a parent layout is also read.
(6)	With title.staff.searchStaff as key, fetch the value from properties incorporated in spring-mvc, where staff is {1} and searchStaff is “search{2}”.
(7)	/WEB-INF/views/staff/searchStaff.jsp is set in body within the template /WEB-INF/views/layout/templateSearch.jsp where staff is {1} and searchStaff is “search{2}”.

As a result, it is output to the browser by combining header.jsp, menu.jsp, searchStaff.jsp and footer.jsp in the above templateSearch.jsp file.



5.20 System Date

5.20.1 Overview

In application development, testing may need to be carried out at any random date and time and not necessarily at system time of the server. Even in Production environment, there could be cases wherein recovery is performed by shifting the date to earlier date.

Therefore, it is desirable to have a setting that can fetch any date and time at development and operation sides instead of system time of the server.

Common library provides `org.terasoluna.gfw.common.date.DateFactory` interface that can arbitrarily change the time to be returned. Using `DateFactory`, date can be returned in the following 5 formats.

method	type
<code>newDateTime</code>	<code>org.joda.time.DateTime</code>
<code>newTimestamp</code>	<code>java.sql.Timestamp</code>
<code>newDate</code>	<code>java.util.Date</code>
<code>newSqlDate</code>	<code>java.sql.Date</code>
<code>newTime</code>	<code>java.sql.Time</code>

Note: About the Joda Time, refer to [Date Operations \(Joda Time\)](#).

Tip: When testing is to be carried out by changing the date and time using JUnit etc., any date and time can be set by replacing the Factory implementation class with mock class.

5.20.2 How to use

Define implementation class of `DateFactory` in bean definition file and inject it in Java class by `DateFactory` interface.

Depending on the intended use, select from the following implementation classes.

Class name	Overview	Remarks
<code>org.terasoluna.gfw.common.date.DefaultDateFactory</code>	Returns system time of application server.	Time cannot be changed as the value is equivalent to that fetched by new <code>DateTime()</code> .
<code>org.terasoluna.gfw.common.date.JdbcFixedDateFactory</code>	Returns the fixed time registered in DB.	<p>It is assumed to be used in Integration Test environment that requires a completely fixed time.</p> <p>It is not used in Performance Test environment and Production environment.</p> <p>Table definition is needed.</p>
<code>org.terasoluna.gfw.common.date.JdbcAdjustedDateFactory</code>	Returns the time fetched by adding difference (milliseconds) between the time registered in DB and system time of application server.	<p>It is assumed to be used in Integration Test environment and System Test environment.</p> <p>It can be used in Production environment by setting the difference value to 0.</p> <p>Table definition is needed.</p>

Note: It is recommended to define the bean definition file that sets each class, in [projectName]-env.xml so that it can be changed according to the environment. Using DateFactory, the date and time can be changed just by changing the settings of bean definition file, without having to change the source code. Example of Bean definition file is described later.

Returning system time of server

Use `org.terasoluna.gfw.common.date.DefaultDateFormat`. For details, refer to the following.

Bean definition file ([projectname]-env.xml)

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.DefaultDateFormat" /> <!-- (1) -->
```

Sr. No.	Description
(1)	Define DefaultDateFormat class in bean.

Java class

```
@Inject
DateFactory dateFactory; // (2)

public TourInfoSearchCriteria setUpTourInfoSearchCriteria() {

    DateTime dateTime = dateFactory.newDateTime(); // (3)

    // omitted
}
```

Sr. No.	Description
(2)	Inject DateFactory in the class to be used.
(3)	Call the method that returns the class instance of the date to be used. In above example, fetch as the <code>org.joda.time.DateTime</code> instance.

Returning the fixed time fetched from DB

Use `org.terasoluna.gfw.common.date.JdbcFixedDateFactory`. For details, refer to the following.

Bean definition file

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.JdbcFixedDateFactory" > <!-- (1) -->
    <property name="dataSource" ref="dataSource" /> <!-- (2) -->
    <property name="currentTimestampQuery" value="SELECT now FROM system_date" /> <!-- (3) -->
</bean>
```

Sr. No.	Description
(1)	Define <code>org.terasoluna.gfw.common.date.JdbcFixedDateFactory</code> in bean.
(2)	Datasource (<code>javax.sql.DataSource</code>) settings.
(3)	Settings related to SQL for fetching fixed time <code>currentTimestampQuery</code> . Set the SQL query that returns the date and time specified in table.

Example of Table settings

Records need to be added by creating a table as shown below.

```
CREATE TABLE system_date(now timestamp NOT NULL);
INSERT INTO system_date(now) VALUES (current_date);
```

Record number	now
1	2013-01-01 01:01:01.000

Java class

```
@Inject
DateFactory dateFactory;
```

```

@RequestMapping(value="datetime", method = RequestMethod.GET)
public String listConfirm(Model model) {

    for (int i=0; i < 3; i++) {
        model.addAttribute("jdbcFixedDateFactory" + i, dateFactory.newDateTime()); // (4)
        model.addAttribute("DateTime" + i, new DateTime()); // (5)
    }

    return "date/dateTimeDisplay";
}

```

Sr. No.	Description
(4)	Pass the JdbcFixedDateFactory.newDateTime() result to screen. The fixed value set in DB is output.
(5)	Pass the new DateTime() result to screen, for confirmation. Output result shows a different value each time.

Execution result

Server Time

```

(1)jdbcFixedDateFactory.newDateTime() first
2013-01-01 01:01:01.000

(2)new DateTime() first
2013-10-10 14:09:18.687

(1)jdbcFixedDateFactory.newDateTime() second
2013-01-01 01:01:01.000

(2)new DateTime() second
2013-10-10 14:09:18.688

(1)jdbcFixedDateFactory.newDateTime() third
2013-01-01 01:01:01.000

(2)new DateTime() third
2013-10-10 14:09:18.689

```

SQL log

```
16. SELECT now FROM system_date {executed in 0 msec}  
17. SELECT now FROM system_date {executed in 1 msec}  
18. SELECT now FROM system_date {executed in 0 msec}
```

Access log is output to DB using `JdbcFixedDateFactory.newDateTime()`. In order to output SQL log, `Log4jdbcProxyDataSource` described in [Database Access \(Common\)](#) is used.

Returning time obtained by adding the difference registered in DB to the server system time

Use `org.terasoluna.gfw.common.date.JdbcAdjustedDateFactory`. Fetch the difference in time by executing SQL set in `adjustedValueQuery` property .

For details, refer to the following.

Bean definition file

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.JdbcAdjustedDateFactory" >  
  <property name="dataSource" ref="dataSource" />  
  <!-- <property name="adjustedValueQuery" value="SELECT diff FROM operation_date" /> --><!-- (1)-->  
  <!-- <property name="adjustedValueQuery" value="SELECT diff * 1000 FROM operation_date" /> --><!--  
  <property name="adjustedValueQuery" value="SELECT diff * 60 * 1000 FROM operation_date" /><!--  
  <!-- <property name="adjustedValueQuery" value="SELECT diff * 60 * 60 * 1000 FROM operation_date" />  
  <!-- <property name="adjustedValueQuery" value="SELECT diff * 24 * 60 * 60 * 1000 FROM operation_date" />  
</bean>
```

Sr. No.	Description
(1)	SQL when the difference registered in operation_date table is in “milliseconds”
(2)	SQL when the difference registered in operation_date table is in “seconds”
(3)	SQL when the difference registered in operation_date table is in “minutes”
(4)	SQL when the difference registered in operation_date table is in “hours”
(5)	SQL when the difference registered in operation_date table is in “days”

Example of table settings

Records need to be added by creating a table as shown below.

```
CREATE TABLE operation_date(diff bigint NOT NULL);
INSERT INTO operation_date(diff) VALUES (-1440);
```

Record number	diff
1	-1440

In this example, the difference is in “minutes”. (DB data is specified as -1440 minutes = previous day)

By converting the retrieved result into milliseconds (integer value), the unit for DB value can be set to any one of the units namely, hours, minutes, seconds or milliseconds.

Note: Above SQL is for PostgreSQL. For Oracle, it is better to use NUMBER (19) instead of BIGINT.

Java class

```
@Inject  
DateFactory dateFactory;  
  
@RequestMapping(value="datetime", method = RequestMethod.GET)  
public String listConfirm(Model model) {  
  
    model.addAttribute("firstExpectedDate", new DateTime()); // (6)  
    model.addAttribute("serverTime", dateFactory.newDateTime()); // (7)  
    model.addAttribute("lastExpectedDate", new DateTime()); // (8)  
  
    return "date/dateTimeDisplay";  
}
```

Sr. No.	Description
(6)	For verification purpose, pass a time that is prior to the <code>DateTime</code> generated by <code>dateFactory</code> , to screen.
(7)	Pass the result of <code>JdbcAdjustedDateFactory.newDateTime()</code> to screen. Fetched time is the time derived by subtracting 1440 minutes from execution time.
(8)	For verification purpose, set a time that is later than the <code>DateTime</code> generated by <code>dateFactory</code> .

Execution result

Server Time

```
(1)new DateTime() first  
2013-10-10 15:21:04.225  
  
(2)minuteJdbcAdjustedDateFactory.newDateTime()  
2013-10-09 15:21:04.229  
  
(3)new DateTime() last  
2013-10-10 15:21:04.229
```

SQL log

```
17. SELECT diff * 60 * 1000 FROM operation_date {executed in 1 msec}
```

Access log is output to DB using `dateFactory.newDateTime()`.

Caching and reloading the difference

When the difference value is set to 0 and used in production environment, performance deteriorates as the difference is fetched each time from DB. Therefore, in `JdbcAdjustedDateFactory`, it is possible to cache the acquisition result. Once the value fetched at booting is cached, table is not accessed for each request.

Bean definition file

```
<bean id="dateFactory" class="org.terasoluna.gfw.common.date.JdbcAdjustedDateFactory" >
  <property name="dataSource" ref="dataSource" />
  <property name="adjustedValueQuery" value="SELECT diff * 60 * 1000 FROM operation_date" />
  <property name="useCache" value="true" /> <!-- (1) -->
</bean>
```

Sr. No.	Description
(1)	When it is ‘true’, the value fetched from table is cached. By default it is ‘false’ so the value is not cached. When it is ‘false’, SQL is executed each time when DateFactory is used.

When the difference value is to be changed after setting cache, cache value can be reloaded by executing `JdbcAdjustedDateFactory.reload()` method after changing the table value.

Java class

```
@Controller
@RequestMapping(value = "reload")
public class ReloadAdjustedValueController {

  @Inject
  JdbcAdjustedDateFactory dateFactory;

  // omitted
```

```
@RequestMapping(method = RequestMethod.GET)
public String reload() {

    long adjustedValue = dateFactory.reload(); // (2)

    // omitted
}
}
```

Sr. No.	Description
(2)	By executing reload method of JdbcAdjustedDateFactory, difference can be reloaded from table.

5.20.3 Testing

When carrying out testing, it may be necessary to change to another date and time instead of the current date and time.

Environment	DateFactory to be used	Test details
Unit Test	DefaultDateFactory	Mock for DataFactory is created for date related testing
Integration Test	DefaultDateFactory	Testing not relating to date
	JdbcFixedDateFactory	When testing is carried out by having a fixed date and time
	JdbcAdjustedDateFactory	When linked with an external system and testing is done for multiple days considering the date flow of a testing for a single day
System Test	JdbcAdjustedDateFactory	When testing is carried out by specifying the testing date or for a future date
Production	DefaultDateFactory	When there is no possibility of change in actual time
	JdbcAdjustedDateFactory	When the possibility to change the time is to be retained in an operation. Normally the difference is set as 0. It is provided only if required. <i>useCache</i> should always be set to ‘true’.

Unit Test

In Unit Test, sometimes it needs to be verified whether the time is registered and the registered time has been updated as expected.

In such cases, if the server time is registered as it is during the process, it becomes difficult to perform regression test in JUnit, as the value differs with each test execution. Here, by using DateFactory, the time to be registered can be fixed to any value.

Use mock to match the time in milliseconds. An example wherein fixed date is returned by setting a value in dateFactory, is shown below. In this example, `mockito` is used for mock.

Java class

```
import org.terasoluna.gfw.common.date.DateFactory;

// omitted

@Inject
StaffRepository staffRepository;

@Inject
DateFactory dateFactory;

@Override
public Staff staffUpdateTel(String staffId, String tel) {

    // ex staffId=0001
    Staff staff = staffRepository.findOne(staffId);

    // ex tel = "0123456789"
    staff.setTel(tel);

    // set ChangeMillis
    staff.setChangeMillis(dateFactory.newDateTime()); // (1)

    staffRepository.save(staff);

    return staff;
}

// omitted
```

JUnit source

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import static org.mockito.Mockito.*;

import org.joda.time.DateTime;
import org.junit.Before;
import org.junit.Test;
import org.terasoluna.gfw.common.date.DateFactory;

public class StaffServiceTest {
```

```
StaffService service;

StaffRepository repository;

DateFactory dateFactory;

DateTime now;

@Before
public void setUp() {
    service = new StaffService();
    dateFactory = mock(DateFactory.class);
    repository = mock(StaffRepository.class);
    now = new DateTime();
    service.dateFactory = dateFactory;
    service.staffRepository = repository;
    when(dateFactory.newDateTime()).thenReturn(now); // (2)
}

@After
public void tearDown() throws Exception {
}

@Test
public void testStaffUpdateTel() {

    Staff setDataStaff = new Staff();
    when(repository.findOne("0001")).thenReturn(setDataStaff);

    // execute
    Staff staff = service.staffUpdateTel("0001", "0123456789");

    //assert
    assertThat(staff.getChangeMillis(), is(now)); // (3)

}
}
```

Sr. No.	Description
(1)	Value specified in (2) of mock is fetched and set.
(2)	Set the date and time to the return value of DateFactory in mock.
(3)	success is returned since it is same as the fixed value that has been set.

Example wherein process changes with date

The example below illustrates a Service class which is implemented with the specification of “Reserved tour cannot be cancelled if the cancellation is sought less than 7 days before the departure day”.

Java class

```
import org.terasoluna.gfw.common.date.DateFactory;

// omitted

@Inject
DateFactory dateFactory;

// omitted

@Override
public void cancel(String reserveNo) throws BusinessException {
    // omitted

    LocalDate today = dateFactory.newDateTime().toLocalDate(); // (1)
    LocalDate cancelLimit = tourInfo.getDepDay().minusDays(7); // (2)

    if (today.isAfter(cancelLimit)) { // (3)
        // omitted (4)
    }

    // omitted
}
```

Sr. No.	Description
(1)	Fetch current date and time. For <code>LocalDate</code> , refer to Date Operations (Joda Time) .
(2)	Calculate the last date up to which the tour can be cancelled.
(3)	Check if today's date is later than the last date for cancellation.
(4)	<code>BusinessException</code> is thrown if the date exceeds the last date for cancellation.

JUnit source

```
@Before
public void setUp() {
    service = new ReserveServiceImpl();

    // omitted

    Reserve reserveResult = new Reserve();
    reserveResult.setDepDay(new LocalDate(2012, 10, 10)); // (5)
    when(reserveRepository.findOne((String) anyObject())).thenReturn(
        reserveResult);
    dateFactory = mock(DateFactory.class);
    service.dateFactory = dateFactory;
}

@Test
public void testCancel01() {

    // omitted

    now = new DateTime(2012, 10, 1, 0, 0, 0, 0);
    when(dateFactory.newDateTime()).thenReturn(now); // (6)

    // run
    service.cancel(reserveNo); // (7)

    // omitted
```

```
}
```

```
@Test(expected = BusinessException.class)
public void testCancel02() {

    // omitted

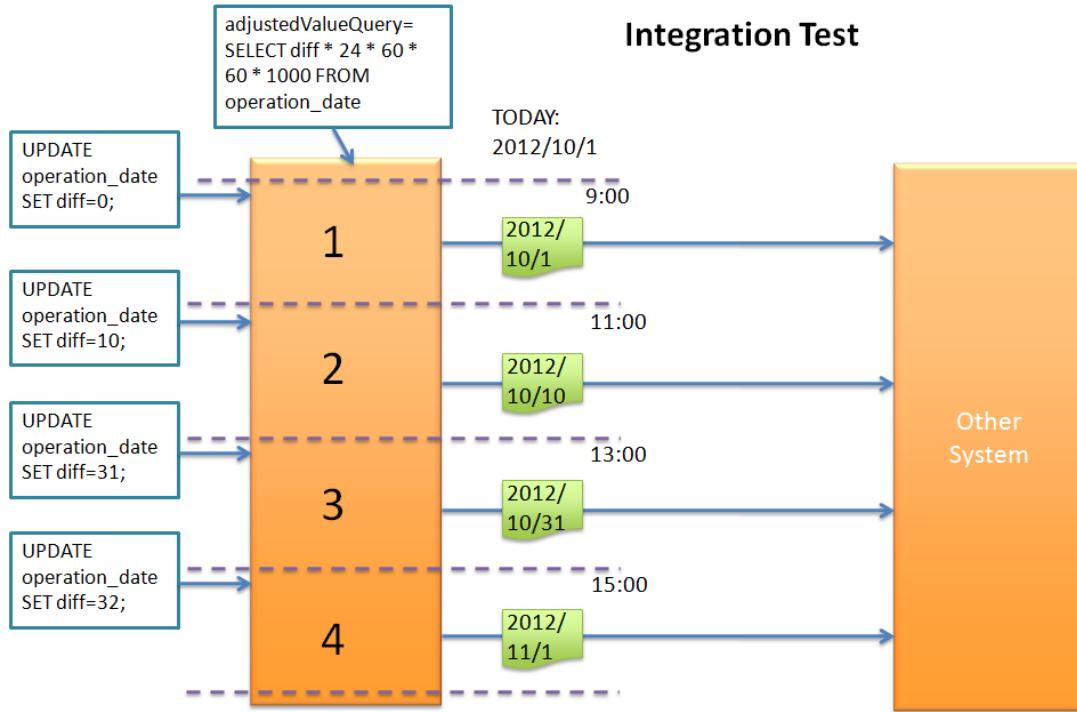
    now = new DateTime(2012, 10, 9, 0, 0, 0);
    when(dateFactory.newDateTime()).thenReturn(now); // (8)

    try {
        // run
        service.cancel(reserveNo); // (9)
        fail("Illegal Route");
    } catch (BusinessException e) {
        // assert message if required
        throw e;
    }
}
```

Sr. No.	Description
(5)	Set the departure date to 2012/10/10 in the tour reservation information to be fetched from Repository class.
(6)	Set the Return value of dateFactory.newDateTime() to 2012/10/1.
(7)	Execute Cancel. Cancellation is successful as the date is prior to the last date for cancellation.
(8)	Return value of dateFactory.newDateTime() should be 2012/10/9.
(9)	Execute Cancel. Cancellation fails as the date falls after the last date for cancellation.

Integration Test

In Integration Test, there may be cases wherein data of several days (for example: files) is created and transferred in a single day, for communicating with the system.



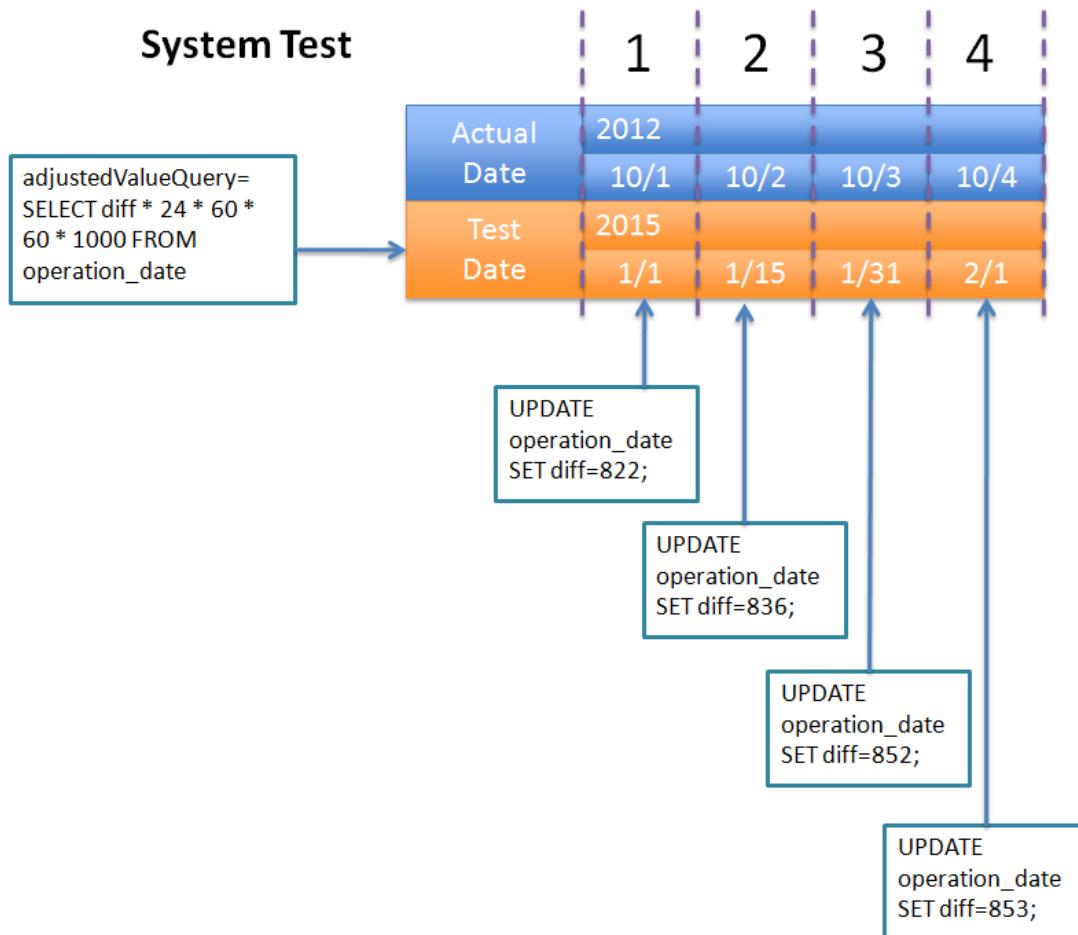
When the actual date is 2012/10/1, Use JdbcAdjustedDateFactory and set the SQL to calculate the difference with test execution date.

Sr. No.	Description
1	Set the difference between 9:00-11:00 as “0 days” and return value of dateFactory as 2012/10/1.
2	Set the difference between 11:00-13:00 as “0 days” and return value of dateFactory as 2012/10/10.
3	Set the difference between 13:00-15:00 as “30 days” and return value of dateFactory as 2012/10/31.
4	Set the difference between 15:00-17:00 as “31 days” and return value of dateFactory as 2012/11/1.

Date can be changed only by changing the table value.

System Test

In System Test, testing may be carried out by creating test scenarios assuming the operation date.



Use JdbcAdjustedDateFactory and set SQL that calculates the date difference. Create a mapping table for actual date and operation date like 1, 2, 3 and 4 as shown in the figure. Testing can be carried out on the desired date, only by changing the difference value in the table.

Production

By setting the difference value to '0' using JdbcAdjustedDateFactory, the return value of dateFactory can be set to the date same as the actual date, without changing the source. Even the bean definition file need not be changed

from System Test onwards. Further, even if the need to change date and time arises, return value of dateFactory can be changed by changing the table value.

Warning: When using in Production environment, verify that the difference value in the table used in Production environment is 0.

Configuration example

Execute the following

- When using the table for the first time in Production environment
 - INSERT INTO operation_date (diff) VALUES (0);
- When test execution is completed in Production environment
 - UPDATE operation_date SET diff=0;

useCache should always be set to ‘true’.

When there is no change in time, it is recommended to change the configuration file to DefaultDateFormat.

5.21 Utilities

5.21.1 Bean Mapping (Dozer)

Overview

Bean mapping is a process to copy the field values of one Bean to another Bean.

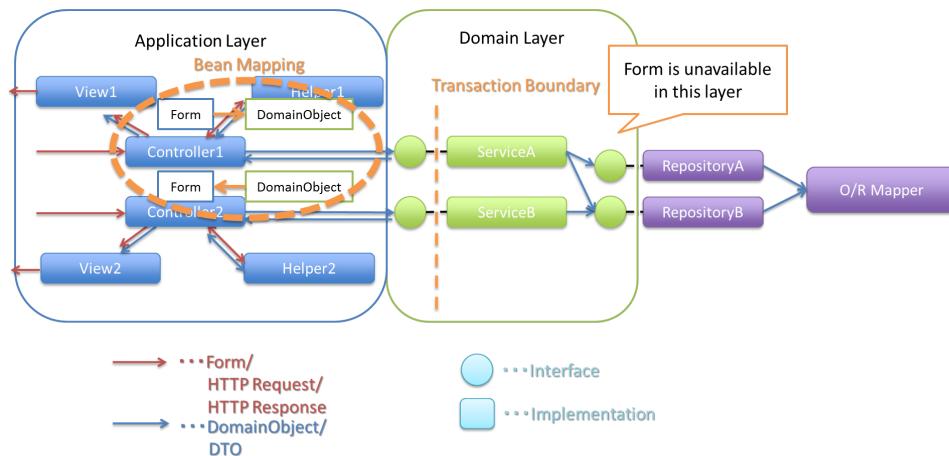
There are many cases where bean mapping is necessary while passing the data between different layers of the application (application layer and domain layer).

For example, AccountForm object of application layer is converted to Account object of domain layer.

Since the domain layer should not depend on the application layer, AccountForm object cannot be used as it is in the domain layer.

Hence, AccountForm object is Bean mapped to Account object and Account object is used in the domain layer.

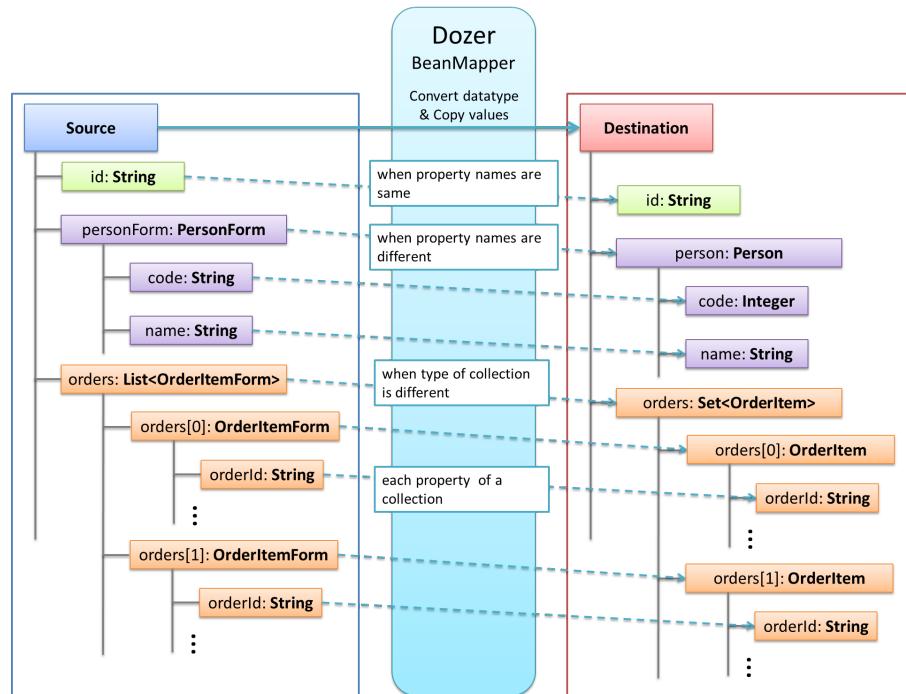
Thus, linear dependency relation can be maintained between application layer and domain layer.



These objects can be mapped by calling getter/setter of Bean and passing the data.

However, since a complex process results in deterioration of the program readability, in this guideline, it is recommended to use [Dozer](#) a Bean mapping library available in OSS.

By using Dozer, different types of copies for copy source class and copy destination class and copy of nested Beans can be easily performed as shown in the following figure.



Code examples with/without using Dozer are given below.

- Example of a complex process that results in the program readability deterioration

```
User user = userService.findById(userId);

XxxOutput output = new XxxOutput();

output.setUserId(user.getUserId());
output.setFirstName(user.getFirstName());
output.setLastName(user.getLastName());
output.setTitle(user.getTitle());
output.setBirthDay(user.getBirthDay());
output.setGender(user.getGender());
output.setStatus(user.getStatus());
```

- Example when Dozer is used

```
User user = userService.findById(userId);

XxxOutput output = beanMapper.map(user, XxxOutput.class);
```

How to use Dozer is explained below.

How to use

Dozer is a mapping function library of JavaBean. Values are copied recursively (nested structure) from conversion source Bean to conversion destination Bean.

Bean definition for using Dozer

Create an instance of `org.dozer.Mapper` when Dozer is to be used independently as follows.

```
Mapper mapper = new DozerBeanMapper();
```

Creating instance of Mapper each time deteriorates efficiency, hence `org.dozer.spring.DozerBeanMapperFactoryBean` provided by Dozer should be used.

Define `org.dozer.spring.DozerBeanMapperFactoryBean`, a Factory class that creates Mapper in Bean definition file (`applicationContext.xml`).

```
<bean class="org.dozer.spring.DozerBeanMapperFactoryBean">
    <property name="mappingFiles"
        value="classpath*: /META-INF/dozer/**/*-mapping.xml" /><!-- (1) -->
</bean>
```

Sr. No.	Description
(1)	<p>Specify mapping definition XML files in <code>mappingFiles</code>. <code>org.dozer.spring.DozerBeanMapperFactoryBean</code> maintains <code>org.dozer.Mapper</code> as an interface. Therefore, specify Mapper for <code>@Inject</code>.</p> <p>In this example, all the (any value)-mapping.xmls in any folder of <code>/META-INF/dozer</code> under the class path are read. The contents of these XML files are explained later.</p>

Mapper should be injected in the class to perform Bean mapping.

```
@Inject
Mapper beanMapper;
```

Mapping when the field name and the type between Beans is same

Dozer can perform mapping by default without creating mapping definition XML file if the field name between the Beans is same.

Bean definition of conversion source

```
public class Source {  
    private int id;  
    private String name;  
    // omitted setter/getter  
}
```

Bean definition of conversion destination

```
public class Destination {  
    private int id;  
    private String name;  
    // omitted setter/getter  
}
```

Perform Bean mapping using `map` method of Mapper as given below. After executing the method given below, a new Destination object is created and each field value of source is copied to the created Destination object.

```
Source source = new Source();  
source.setId(1);  
source.setName("SourceName");  
Destination destination = beanMapper.map(source, Destination.class); // (1)  
System.out.println(destination.getId());  
System.out.println(destination.getName());
```

Sr. No.	Description
(1)	Pass the object to be copied as the first argument and the Bean class where it is to be copied as the second argument.

The output of the above code is as given below. The value of object to be copied is set in the created object.

```
1  
SourceName
```

To copy the field of `source` object to already existing `destination` object, perform the following

```
Source source = new Source();  
source.setId(1);  
source.setName("SourceName");  
Destination destination = new Destination();  
destination.setId(2);  
destination.setName("DestinationName");  
beanMapper.map(source, destination); // (1)  
System.out.println(destination.getId());  
System.out.println(destination.getName());
```

Sr. No.	Description
(1)	Pass the object to be copied as the first argument and the object where it is to be copied as the second argument.

The output of the above code is as given below. The value of object to be copied is reflected in the destination where it is to be copied.

```
1
SourceName
```

Note: The value that does not exist in Source class does not change before and after copying to the field of Destination class.

Bean definition of conversion source

```
public class Source {
    private int id;
    private String name;
    // omitted setter/getter
}
```

Bean definition of conversion destination

```
public class Destination {
    private int id;
    private String name;
    private String title;
    // omitted setter/getter
}
```

Example of Mapping

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
Destination destination = new Destination();
destination.setId(2);
destination.setName("DestinationName");
destination.setTitle("DestinationTitle");
beanMapper.map(source, destination);
System.out.println(destination.getId());
System.out.println(destination.getName());
System.out.println(destination.getTitle());
```

The output of the above code is as given below. Since there is no title field in the Source class, the value of title field of Destination object remains the same as the field value before copying.

```
1
SourceName
DestinationTitle
```

Mapping when the field name is the same and type between Beans is different

When field type of Bean is different in copy source and copy destination, mapping of the type where type conversion is supported can be done automatically.

The conversion given below is an example where it is possible to convert without a mapping definition XML file.

Example: String -> BigDecimal

Bean definition of conversion source

```
public class Source {
    private String amount;
    // omitted setter/getter
}
```

Bean definition of conversion destination

```
public class Destination {
    private BigDecimal amount;
    // omitted setter/getter
}
```

Example of Mapping

```
Source source = new Source();
source.setAmount("123.45");
Destination destination = beanMapper.map(source, Destination.class);
System.out.println(destination.getAmount());
```

The output of the above code is as given below. The value can be copied even when the type is different.

```
123.45
```

Refer to [Manual](#) for the supported type conversions.

Mapping when the field name between Beans is different

When field name of copy source is different from the copy destination, creating mapping definition XML file and defining the field for Bean mapping enables conversion.

Bean definition of conversion source

```
public class Source {  
    private int id;  
    private String name;  
    // omitted setter/getter  
}
```

Bean definition of conversion destination

```
public class Destination {  
    private int destinationId;  
    private String destinationName;  
    // omitted setter/getter  
}
```

To define *Bean definition for using Dozer*, create mapping definition XML file called (any value)-mapping.xml in the src/main/resources/META-INF/dozer folder.

```
<?xml version="1.0" encoding="UTF-8"?>  
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://dozer.sourceforge.net  
        http://dozer.sourceforge.net/schema/beanmapping.xsd">  
  
<mapping>  
    <class-a>com.xx.xx.Source</class-a><!-- (1) -->  
    <class-b>com.xx.xx.Destination</class-b><!-- (2) -->  
    <field>  
        <a>id</a><!-- (3) -->  
        <b>destinationId</b><!-- (4) -->  
    </field>  
    <field>  
        <a>name</a>  
        <b>destinationName</b>  
    </field>  
</mapping>  
  
</mappings>
```

Sr. No.	Description
(1)	Specify fully qualified class name (FQCN) of copy source Bean in <class-a> tag.
(2)	Specify fully qualified class name (FQCN) of copy destination Bean in <class-b> tag.
(3)	Specify field name for mapping of copy source Bean in <a> tag of <field> tag.
(4)	Specify field name for mapping of copy destination Bean corresponding to (3) in tag of <field> tag.

Example of Mapping

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
Destination destination = beanMapper.map(source, Destination.class); // (1)
System.out.println(destination.getDestinationId());
System.out.println(destination.getDestinationName());
```

Sr. No.	Description
(1)	Pass object to be copied as the first argument and the Bean class where it is to be copied as the second argument. (no difference with basic mapping.)

The output of the above code is as given below.

```
1
SourceName
```

Mapping definition XML file existing under META-INF/dozer of class path is read in `mappingFiles` property in accordance with the setting of [Bean definition for using Dozer](#). File name should be (any value)-mapping.xml. The settings are applied if mapping between Source class and Destination class is defined in any file.

Note: It is recommended to create a mapping definition XML file in each Controller and name the file as -mapping.xml (value obtained by removing Controller from Controller name). For example, mapping definition XML file of TodoController is created in src/main/resources/META-INF/dozer/todo-mapping.xml.

One-way/Two-way mapping

The mapping defined in mapping XML is two-way mapping by default. In the above example, mapping is done from Source object to Destination object however, mapping can also be done from Destination object to Source object.

To specify only one-way mapping, set "one-way" in the type attribute of <mapping> tag in the mapping field definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
                               http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping type="one-way">
        <class-a>com.xx.xx.Source</class-a>
        <class-b>com.xx.xx.Destination</class-b>
        <field>
            <a>id</a>
            <b>destinationId</b>
        </field>
        <field>
            <a>name</a>
            <b>destinationName</b>
        </field>
    </mapping>
    <!-- omitted -->
</mappings>
```

Bean definition of conversion source

```
public class Source {
    private int id;
    private String name;
    // omitted setter/getter
}
```

Bean definition of conversion destination

```
public class Destination {
    private int destinationId;
    private String destinationName;
    // omitted setter/getter
}
```

Example of Mapping

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
Destination destination = beanMapper.map(source, Destination.class);
System.out.println(destination.getDestinationId());
System.out.println(destination.getDestinationName());
```

The output of the above code is as given below.

```
1
SourceName
```

If one-way is specified, an error does not occur even if the mapping is done in the reverse direction. The copy process is ignored. This is because Source field corresponding to Destination field does not exist if mapping is not defined.

```
Destination destination = new Destination();
destination.setDestinationId(2);
destination.setDestinationName("DestinationName");

Source source = new Source();
source.setId(1);
source.setName("SourceName");

beanMapper.map(destination, source);

System.out.println(source.getId());
System.out.println(source.getName());
```

The output of the above code is as given below.

```
1
SourceName
```

Mapping of nested fields

It is possible to map the field of Bean to be copied to the field with Nested attributes of the Bean where it is to be copied. (In Dozer terminology, it is called Deep Mapping.)

Bean definition of conversion source

```
public class EmployeeForm {
    private int id;
    private String name;
    private String deptId;
    // omitted setter/getter
}
```

Bean definition of conversion destination

```
public class Employee {
    private Integer id;
    private String name;
    private Department department;
    // omitted setter/getter
}
```

```
public class Department {
    private String deptId;
    // omitted setter/getter and other fields
}
```

Example: Define as below to map the deptId with EmployeeForm object to deptId of Department with Employee object.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
                               http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping map-empty-string="false" map-null="false">
        <class-a>com.xx.aa.EmployeeForm</class-a>
        <class-b>com.xx.bb.Employee</class-b>
        <field>
            <a>deptId</a>
            <b>department.deptId</b><!-- (1) -->
        </field>
    </mapping>
    <!-- omitted -->
</mappings>
```

Sr. No.	Description
(1)	Specify the field of Employee object for deptId of Employee form.

Example of Mapping

```
EmployeeForm source = new EmployeeForm();
source.setId(1);
source.setName("John");
source.setDeptId("D01");

Employee destination = beanMapper.map(source, Employee.class);
System.out.println(destination.getId());
System.out.println(destination.getName());
System.out.println(destination.getDepartment().getDeptId());
```

The output of the above code is as given below.

```
1
John
D01
```

In the above case, a new instance of Employee, a class of conversion destination is created. The newly created Department instance is set in the department field of Employee and deptId of the EmployeeForm is copied.

When the Department object is already set in the department field of Employee, a new instance is not created but the deptId of EmployeeForm is copied to the deptId field of the existing Department object.

```
EmployeeForm source = new EmployeeForm();
source.setId(1);
source.setName("John");
source.setDeptId("D01");

Employee destination = new Employee();
Department department = new Department();
destination.setDepartment(department);

beanMapper.map(source, destination);
System.out.println(department.getDeptId());
System.out.println(destination.getDepartment() == department);
```

The output of the above code is as given below.

```
D01
true
```

Collection mapping

Dozer supports two-way auto-mapping of the following Collection types. When field name is same, mapping definition XML file is not required.

- java.util.List <=> java.util.List
- java.util.List <=> Array
- Array <=> Array
- java.util.Set <=> java.util.Set
- java.util.Set <=> Array
- java.util.Set <=> java.util.List

Bean mapping with a collection of the following classes is considered.

```
package com.example.dozer;

public class Email {
    private String email;

    public Email() {
    }

    public Email(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public String toString() {
        return email;
    }

    // generated by Eclipse
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((email == null) ? 0 : email.hashCode());
        return result;
    }

    // generated by Eclipse
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Email other = (Email) obj;
        if (email == null) {
            if (other.email != null)
                return false;
        } else if (!email.equals(other.email))
            return false;
    }
}
```

```
        return true;
    }

}
```

Conversion source Bean

```
package com.example.dozer;

import java.util.List;

public class AccountForm {
    private List<Email> emails;

    public void setEmails(List<Email> emails) {
        this.emails = emails;
    }

    public List<Email> getEmails() {
        return emails;
    }
}
```

Conversion destination Bean

```
package com.example.dozer;

import java.util.List;

public class Account {
    private List<Email> emails;

    public void setEmails(List<Email> emails) {
        this.emails = emails;
    }

    public List<Email> getEmails() {
        return emails;
    }
}
```

Example of Mapping

```
AccountForm accountForm = new AccountForm();

List<Email> emailsSrc = new ArrayList<Email>();

emailsSrc.add(new Email("a@example.com"));
emailsSrc.add(new Email("b@example.com"));
emailsSrc.add(new Email("c@example.com"));
```

```
accountForm.setEmails(emailsSrc);

Account account = beanMapper.map(accountForm, Account.class);

System.out.println(account.getEmails());
```

The output of the above code is as given below.

```
[a@example.com, b@example.com, c@example.com]
```

There is no particular change in the explanation given so far.

As shown in the example below, **it is necessary to exercise caution when the element is already added to the Collection field of copy destination Bean.**

```
AccountForm accountForm = new AccountForm();
Account account = new Account();

List<Email> emailsSrc = new ArrayList<Email>();
List<Email> emailsDest = new ArrayList<Email>();

emailsSrc.add(new Email("a@example.com"));
emailsSrc.add(new Email("b@example.com"));
emailsSrc.add(new Email("c@example.com"));

emailsDest.add(new Email("a@example.com"));
emailsDest.add(new Email("d@example.com"));
emailsDest.add(new Email("e@example.com"));

accountForm.setEmails(emailsSrc);
account.setEmails(emailsDest);

beanMapper.map(accountForm, account);

System.out.println(account.getEmails());
```

The output of the above code is as given below.

```
[a@example.com, d@example.com, e@example.com, a@example.com, b@example.com, c@example.com]
```

All the elements of Collection of copy source Bean are added to the Collection of copy destination Bean. Two Email objects with a@example.com are “Equal”, but are added easily.

(“Equal” here signifies true when compared by Email.equals and has the same value as Email.hashCode.)

The above behavior is called as **cumulative** in Dozer terminology and is a default behavior at the time of Collection mapping.

This behavior can be changed in mapping definition XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
                           http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping>
        <class-a>com.example.dozer.AccountForm</class-a>
        <class-b>com.example.dozer.Account</class-b>
        <field relationship-type="non-cumulative"><!-- (1) -->
            <a>emails</a>
            <b>emails</b>
        </field>
    </mapping>
    <!-- omitted -->
</mappings>
```

Sr. No.	Description
(1)	<p>Specify non-cumulative in relationship-type attribute of <field> tag. Default value is cumulative.</p> <p>To specify non-cumulative in all the fields of Bean to be mapped, specify non-cumulative in the relationship-type attribute of <mapping> tag.</p>

The output of the above code is as given below based on this setting.

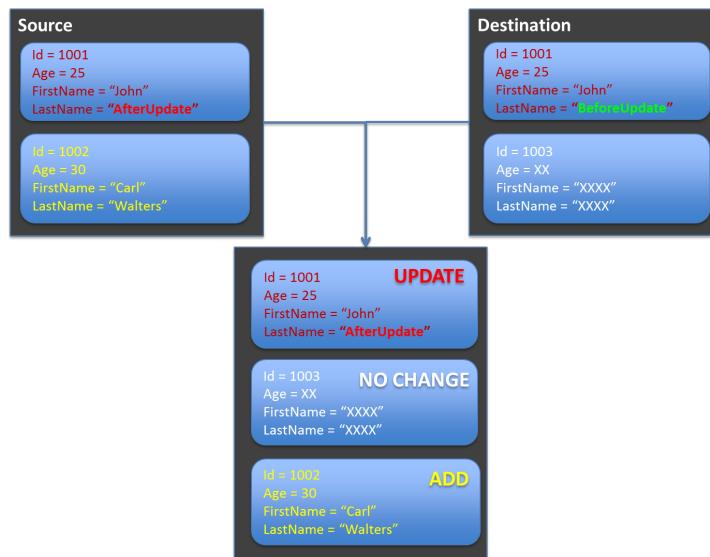
```
[a@example.com, d@example.com, e@example.com, b@example.com, c@example.com]
```

Duplication of equivalent objects is eliminated.

Note: It is necessary to exercise caution for updating conversion source object with conversion destination object. In the above example, a@example.com of AccountForm is stored in the copy destination.

It can be implemented using the mapping definition XML file settings even to remove the fields existing only in the copy destination collection.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
                           http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping>
        <class-a>com.example.dozer.AccountForm</class-a>
        <class-b>com.example.dozer.Account</class-b>
        <field relationship-type="non-cumulative" remove-orphans="true" ><!-- (1) -->
```



```

<a>emails</a>
<b>emails</b>
</field>
</mapping>
<!-- omitted -->
</mappings>

```

Sr. No.	Description
(1)	Set <code>true</code> in the <code>remove-orphans</code> attribute of <code><field></code> tag. Default value is <code>false</code> .

The output of the above code is as given below based on this setting.

```
[a@example.com, b@example.com, c@example.com]
```

Only the objects to be copied are reflected in the collection where they are to be copied.

The same result is obtained even with the settings given below.

```

<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
           http://dozer.sourceforge.net/schema/beanmapping.xsd">
<!-- omitted -->
<mapping>
    <class-a>com.example.dozer.AccountForm</class-a>
    <class-b>com.example.dozer.Account</class-b>
    <field copy-by-reference="true"><!-- (1) -->
        <a>emails</a>
        <b>emails</b>
    </field>
</mapping>

```

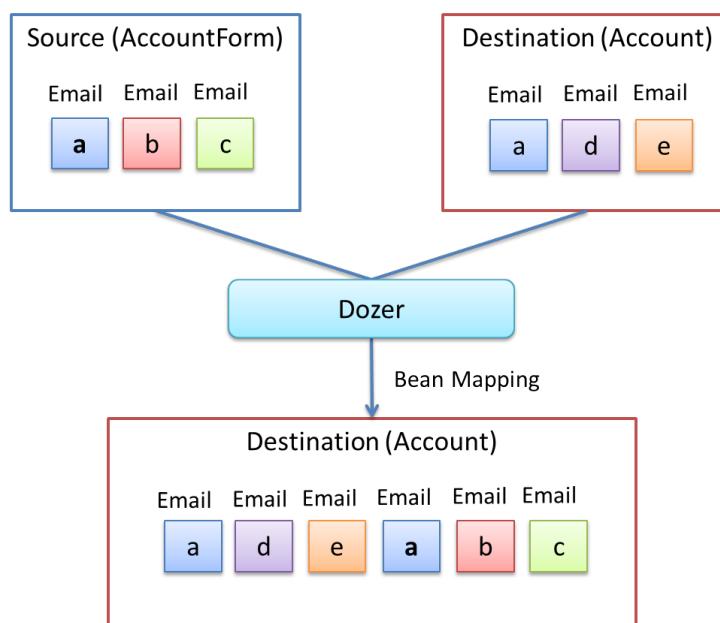
```
<!-- omitted -->
</mappings>
```

Sr. No.	Description
(1)	Set <code>true</code> in the <code>copy-by-reference</code> attribute of <code><field></code> tag. Default value is <code>false</code> .

The behavior explained so far is given in the figure.

- Default behavior (cumulative)

default behavior(cumulative)

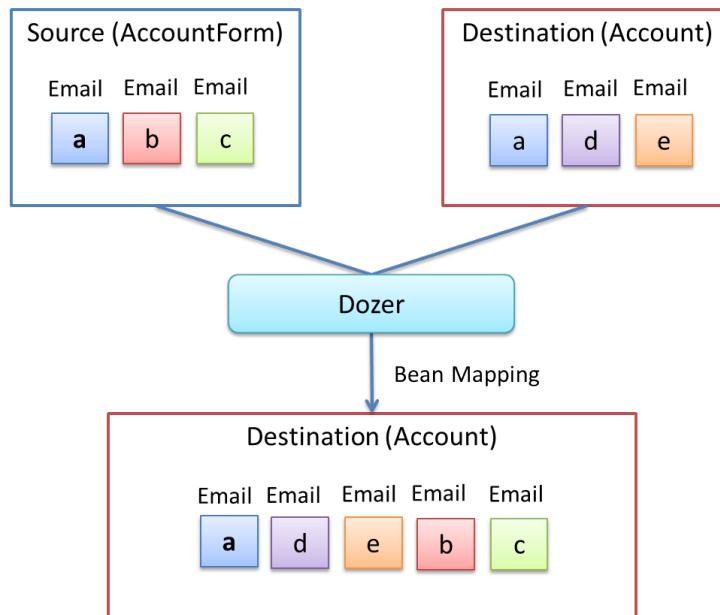


- non-cumulative
 - non-cumulative and remove-orphans=true
- copy-by-reference is also of this pattern.

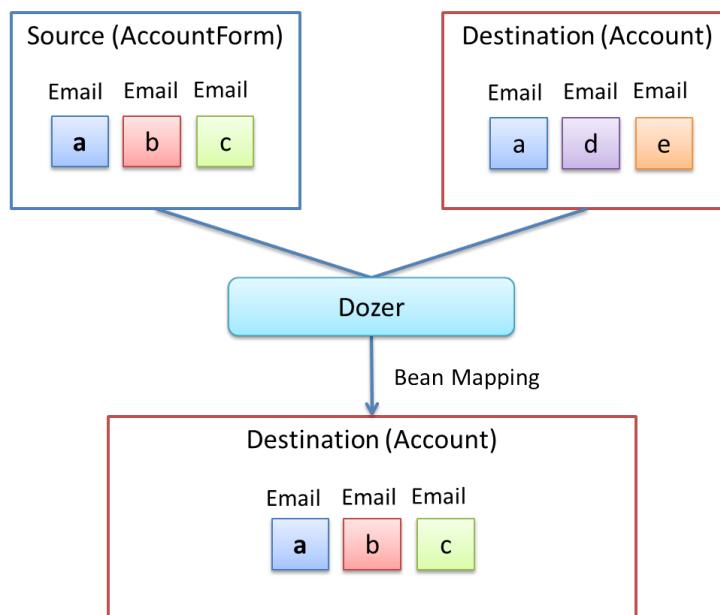
Note: The difference in “non-cumulative and remove-orphans=true” pattern and “copy-by-reference” pattern is whether the container of Collection after Bean conversion is copy destination or copy source.

In case of “non-cumulative and remove-orphans=true” pattern, the container of Collection after Bean conversion is copy destination whereas in case of “copy-by-reference”, the container is copy source. It is explained in the figure given below.

non-cumulative

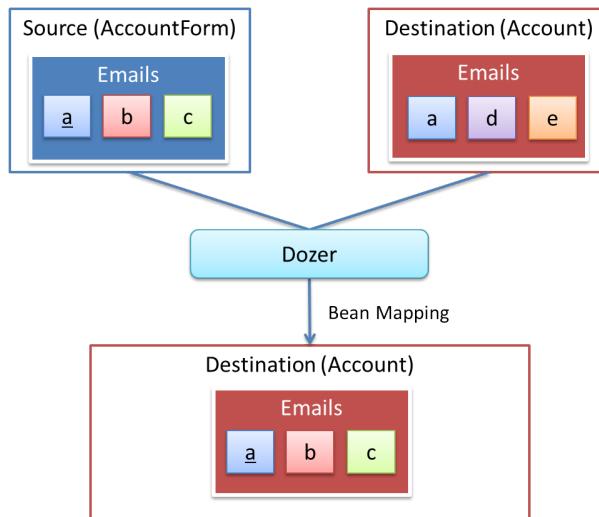


non-cumulative & remove-orphans



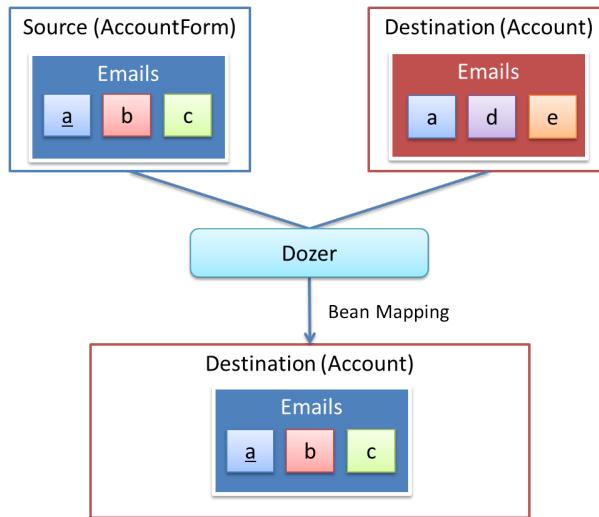
- non-cumulative and remove-orphans=true

non-cumulative & remove-orphans



- copy-by-reference

copy-by-reference



It is necessary to exercise caution when the copy destination has one-to many and many-to-many relationship in the entity of JPA (Hibernate). An unexpected issue may occur when copy destination entity is under EntityManager control. For example, an SQL of DELETE ALL + INSERT ALL may be issued when the container of collection is changed and an SQL to UPDATE (DELETE or INSERT when the number of elements differ) the changes may be issued when copied with “non-cumulative and remove-orphans=true”. Any pattern can be used depending on the requirements.

Warning: When the Bean to be mapped has String collection, a bug wherein the behavior is not as expected is generated.

```
StringListSrc src = new StringListSrc;
StringListDest dest = new StringListDest();

List<String> stringsSrc = new ArrayList<String>();
List<String> stringsDest = new ArrayList<String>();

stringsSrc.add("a");
stringsSrc.add("b");
stringsSrc.add("c");

stringsDest.add("a");
stringsDest.add("d");
stringsDest.add("e");

src.setStrings(stringsSrc);
dest.setStrings(stringsDest);

beanMapper.map(src, dest);

System.out.println(dest.getStrings());
```

If code given above is executed in the non-cumulative and remove-orphans=true settings, following is expected to be output.

```
[a, b, c]
```

However, following is output.

```
[b, c]
```

and **duplicate String is removed**.

When the code is executed in the copy-by-reference="true" settings, following is output.

```
[a, b, c]
```

Tip: In Dozer, the mapping can be performed even between the lists which do not use Generics. At this time, data type of the object included in conversion source and conversion destination can be specified as HINT. Refer to [Manual](#) for details. (Using Hints for Collection Mapping)

Todo

It is checked that the mapping between Beans that uses Collection<T> fails.

Example :

```
public class ListNestedBean<T> {
    private List<T> nest;
    // omitted other declarations
}
```

Execution result :

```
java.lang.ClassCastException: sun.reflect.generics.reflectiveObjects.TypeVariableImpl can
```

How to extend

Creating custom convertor

Mapping of the data type not supported by Dozer can be performed through a custom converter.

- Example : `java.lang.String <=> org.joda.time.DateTime`

Custom converter is a class that implements `org.dozer.CustomConverter` provided by Dozer.

Custom converter can be specified in the following 3 patterns.

- Global Configuration
- Class level
- Field level

Global Configuration is recommended to perform conversion using the same logic in the entire application.

It is convenient to inherit `org.dozer.DozerConverter` to implement a custom converter.

```
package com.example.yourproject.common.bean.converter;

import org.dozer.DozerConverter;
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.springframework.util.StringUtils;

public class StringToJodaDateTimeConverter extends
        DozerConverter<String, DateTime> { // (1)
    public StringToJodaDateTimeConverter() {
        super(String.class, DateTime.class); // (2)
    }

    @Override
```

```

public DateTime convertTo(String source, DateTime destination) { // (3)
    if (!StringUtils.hasLength(source)) {
        return null;
    }
    DateTimeFormatter formatter = DateTimeFormat
        .forPattern("yyyy-MM-dd HH:mm:ss");
    DateTime dt = formatter.parseDateTime(source);
    return dt;
}

@Override
public String convertFrom(DateTime source, String destination) { // (4)
    if (source == null) {
        return null;
    }
    return source.toString("yyyy-MM-dd HH:mm:ss");
}
}

```

Sr. No.	Description
(1)	Inherit org.dozer.DozerConverter.
(2)	Set 2 target classes in the constructor.
(3)	Describe the logic to convert from String to DateTime. In this example, Locale is used by default.
(4)	Describe the logic to convert from DateTime to String. In this example, Locale is used by default.

The created custom converter should be defined for mapping.

dozer-configuration-mapping.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
           http://dozer.sourceforge.net/schema/beanmapping.xsd">

    <configuration>
        <custom-converters><!-- (1) -->
            <!-- these are always bi-directional -->
            <converter

```

```
        type="com.example.yourproject.common.bean.converter.StringToJodaDateTimeConverter"
        <class-a>java.lang.String</class-a><!-- (3) -->
        <class-b>org.joda.time.DateTime</class-b><!-- (4) -->
    </converter>
</custom-converters>
</configuration>
<!-- omitted -->
</mappings>
```

Sr. No.	Description
(1)	Define custom-converters to which all the custom converters belong.
(2)	Define converter for performing individual conversion. Specify fully qualified class name (FQCN) of implementation class in the converter type.
(3)	Fully qualified class name (FQCN) of conversion source Bean
(4)	Fully qualified class name (FQCN) of conversion destination Bean

When `java.lang.String <=> org.joda.time.DateTime` conversion needs to be performed in the entire application by using the mapping given above, the mapping is performed by calling custom converter instead of standard mapping.

Example :

Bean definition of conversion source

```
public class Source {
    private int id;
    private String date;
    // omitted setter/getter
}
```

Bean definition of conversion destination

```
public class Destination {
    private int id;
    private DateTime date;
    // omitted setter/getter
}
```

Mapping (two-way)

```
Source source = new Source();
source.setId(1);
source.setDate("2012-08-10 23:12:12");

DateTimeFormatter formatter = DateTimeFormat.forPattern("yyyy-MM-dd HH:mm:ss");
DateTime dt = formatter.parseDateTime(source.getDate());

// Source to Destination Bean Mapping (String to org.joda.time.DateTime)
Destination destination = dozerBeanMapper.map(source, Destination.class);
assertThat(destination.getId(), is(1));
assertThat(destination.getDate(), is(dt));

// Destination to Source Bean Mapping (org.joda.time.DateTime to String)
dozerBeanMapper.map(destination, source);

assertThat(source.getId(), is(1));
assertThat(source.getDate(), is("2012-08-10 23:12:12"));
```

Refer to [Manual](#) for the details of custom converter.

Note: The conversion from String to the standard date/time object such as java.util.Date is explained in “[Mapping from string to date/time object](#)”.

Appendix

The options that can be specified in the mapping definition XML file are explained.

All options can be confirmed in [Dozer Manual](#).

Field exclusion settings (field-exclude)

The fields that are not to be copied can be excluded at the time of Bean conversion.

Bean conversion is as given below.

Bean definition of conversion source

```
public class Source {
    private int id;
    private String name;
    private String title;
    // omitted setter/getter
}
```

Bean definition of copy destination

```
public class Destination {  
    private int id;  
    private String name;  
    private String title;  
    // omitted setter/getter  
}
```

Define as follows to exclude any field of copy source Bean from mapping.

Carry out the settings of field exclusion in the mapping definition XML file as given below.

```
<?xml version="1.0" encoding="UTF-8"?>  
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:schemaLocation="http://dozer.sourceforge.net  
http://dozer.sourceforge.net/schema/beanmapping.xsd">  
<!-- omitted -->  
<mapping>  
    <class-a>com.xx.xx.Source</class-a>  
    <class-b>com.xx.xx.Destination</class-b>  
    <field-exclude><!-- (1) -->  
        <a>title</a>  
        <b>title</b>  
    </field-exclude>  
</mapping>  
<!-- omitted -->  
</mappings>
```

Sr. No.	Description
(1)	Set the field you want to exclude in the <field-exclude> element. In this example, if map method is executed after specification, the title value of destination is not overwritten while copying Destination object from Source object.

```
Source source = new Source();  
source.setId(1);  
source.setName("SourceName");  
source.setTitle("SourceTitle");  
  
Destination destination = new Destination();  
destination.setId(2);  
destination.setName("DestinationName");  
destination.setTitle("DestinationTitle");  
beanMapper.map(source, destination);  
System.out.println(destination.getId());  
System.out.println(destination.getName());  
System.out.println(destination.getTitle());
```

The output of the above code is as given below.

```
1
SourceName
DestinationTitle
```

The title value of destination after mapping is same as in the previous state.

Specifying mapping (map-id)

The mapping indicated in *Field exclusion settings (field-exclude)* is applied while performing Bean conversion in the entire application. To specify the applicable range of mapping, define map-id as given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
           http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping map-id="mapidTitleFieldExclude">
        <class-a>com.xx.xx.Source</class-a>
        <class-b>com.xx.xx.Destination</class-b>
        <field-exclude>
            <a>title</a>
            <b>title</b>
        </field-exclude>
    </mapping>
    <!-- omitted -->
</mappings>
```

When the above settings are carried out, title can be excluded from copying by passing map-id (mapidTitleFieldExclude) to map method. When map-id is not specified, all the fields are copied without applying the settings.

The example of passing map-id to map method is shown below.

```
Source source = new Source();
source.setId(1);
source.setName("SourceName");
source.setTitle("SourceTitle");

Destination destination1 = new Destination();
destination1.setId(2);
destination1.setName("DestinationName");
destination1.setTitle("DestinationTitle");
beanMapper.map(source, destination1); // (1)
System.out.println(destination1.getId());
System.out.println(destination1.getName());
System.out.println(destination1.getTitle());
```

```
Destination destination2 = new Destination();
destination2.setId(2);
destination2.setName("DestinationName");
destination2.setTitle("DestinationTitle");
beanMapper.map(source, destination2, "mapidSourceBeanFieldExclude"); // (2)
System.out.println(destination2.getId());
System.out.println(destination2.getName());
System.out.println(destination2.getTitle());
```

Sr. No.	Description
(1)	Normal mapping.
(2)	Pass map-id as the third argument and apply the specific mapping rules.

The output of the above code is as given below.

```
1
SourceName
SourceTitle

1
SourceName
DestinationTitle
```

Tip: map-id can be specified not only by mapping items but also by defining the field. Refer to [Manual](#) for details.

Note: The same form object can be used for both creating new/updating the existing Web applications. Form object is copied (mapped) to domain object, however a field may exist where the object is not to be copied depending on the operations. In this case, use <field-exclude>.

- Example: userId is included in the New form whereas it is not included in the Update form.

In this case, null is set in the userId at the time of update if the same form object is used. If form object is copied as it is by fetching copy destination object from DB, userId of copy destination becomes null. In order to avoid this, provide map-id for update and carry out the settings to exclude field of userId at the time of update.

Settings to exclude null/empty field of copy source (map-null, map-empty)

null or empty field of copy source Bean can be excluded from mapping. Set in the mapping definition XML file as given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
           http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping map-null="false" map-empty-string="false"><!-- (1) -->
        <class-a>com.xx.xx.Source</class-a>
        <class-b>com.xx.xx.Destination</class-b>
    </mapping>
    <!-- omitted -->
</mappings>
```

Sr. No.	Description
(1)	To exclude the null field of copy source Bean from mapping, set false in the map-null attribute. Default value is true. To exclude the empty field of copy source Bean from mapping, set false in the map-empty-string attribute. Default value is true.

Bean definition of conversion source

```
public class Source {
    private int id;
    private String name;
    private String title;
    // omitted setter/getter
}
```

Bean definition of conversion destination

```
public class Destination {
    private int id;
    private String name;
```

```
private String title;  
// omitted setter/getter  
}
```

Mapping example

```
Source source = new Source();  
source.setId(1);  
source.setName(null);  
source.setTitle("");  
  
Destination destination = new Destination();  
destination.setId(2);  
destination.setName("DestinationName");  
destination.setTitle("DestinationTitle");  
beanMapper.map(source, destination);  
System.out.println(destination.getId());  
System.out.println(destination.getName());  
System.out.println(destination.getTitle());
```

The output of the above code is as given below.

```
1  
DestinationName  
DestinationTitle
```

name and title field of copy source Bean are null or empty, and are excluded from mapping.

Mapping from string to date/time object

The copy source string field can be mapped to the copy destination date/time field.

Following 6 types of conversions are supported.

Date/time

- java.lang.String <=> java.util.Date
- java.lang.String <=> java.util.Calendar
- java.lang.String <=> java.util.GregorianCalendar
- java.lang.String <=> java.sql.Timestamp

Date only

- java.lang.String <=> java.sql.Date

Time only

- java.lang.String <=> java.sql.Time

Perform date/time conversion as follows.

For example, the conversion to `java.util.Date` is explained.

Conversions for `java.util.Calendar`,`java.util.GregorianCalendar`,`java.sql.Timestamp` can also be performed by the same method.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
                               http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <mapping>
        <class-a>com.xx.xx.Source</class-a>
        <class-b>com.xx.xx.Destination</class-b>
        <field>
            <a date-format="yyyy-MM-dd HH:mm:ss:SS">date</a><!-- (1) -->
            <b>date</b>
        </field>
    </mapping>
    <!-- omitted -->
</mappings>
```

Sr. No.	Description
(1)	Specify field name and date format of copy source.

Bean definition of conversion source

```
public class Source {
    private String date;
    // omitted setter/getter
}
```

Bean definition of conversion destination

```
public class Destination {
    private Date date;
    // omitted setter/getter
}
```

Mapping

```
Source source = new Source();
source.setDate("2013-10-10 11:11:11.111");
Destination destination = beanMapper.map(source, Destination.class);
assert(destination.getDate().equals(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").parse("2013-10-10 11:11:11.111")));
```

There are many cases where it is preferable to set a common date format in the project rather than setting an individual date for each mapping definition.

In such a case, it is recommended to set in the Global configuration file of Dozer.

The date format set in the mapping of the entire application is applied.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
                               http://dozer.sourceforge.net/schema/beanmapping.xsd">
    <!-- omitted -->
    <configuration>
        <date-format>yyyy-MM-dd HH:mm:ss.SSS</date-format>
        <!-- omitted other configuration -->
    </configuration>
    <!-- omitted -->
</mappings>
```

There are no restrictions for the file name however,

src/main/resources/META-INF/dozer/dozer-configuration-mapping.xml is recommended.

Global Configuration that impacts the entire application in this configuration file can be used within the scope of settings for dozer-configuration-mapping.xml.

Refer to the [Manual](#) for the details of items that can be configured.

Mapping error

If mapping process fails during mapping, org.dozer.MappingException (runtime exception) is thrown.

The typical examples of MappingException being thrown are given below.

- map-id that does not exist in the map method is passed.
- map-id existing in map method is passed however, the source/target type passed to map process differs from the definition specified in map-id.
- When conversion is not supported by Dozer and custom converter for conversion does not exist as well.

Since these are normal program bugs, the sections called by map method should be modified appropriately.

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 [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 <code>java.util.Date</code> to <code>DateTime</code> by passing <code>java.util.Date</code> as an argument to <code>DateTime</code> constructor.
(2)	Convert <code>DateTime</code> to <code>java.util.Date</code> using <code>DateTime#toDate</code> 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 <code>toString</code> , 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)	<code>isAfter</code> method returns <code>true</code> when the target date and time is later than the date and time of argument.
(2)	<code>isBefore</code> method returns <code>true</code> when the target date and time is prior to the date and time of argument.
(3)	<code>isEqual</code> method returns <code>true</code> 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 5.26 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="#">Prev <a href="#">Next
    &rarr; <br>
    <joda:format value="#">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>&nbsp;</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 Spring Security Overview

6.1.1 Overview

Two main functionalities namely, “Authentication” and “Authorization” are provided by Spring Security for the security of applications.

Authentication functionality identifies a user and thus prevents unauthorized access through spoofing. Authorization functionality controls the access to system resources according to the authority of the authenticated (logged-in) user.

Spring Security overview is shown in diagram below.

Spring Security implements authorization and authentication processes with help of a group of ServletFilters that interact across several levels. Further, it also provides password hashing functionality, JSP authorized tag library etc.

Authentication

Authentication is the action that checks validity of a request. When connecting to the network or server, through combination of user name and password, it further verifies whether the user has the required authority and also whether the person to be authenticated is really the user himself.

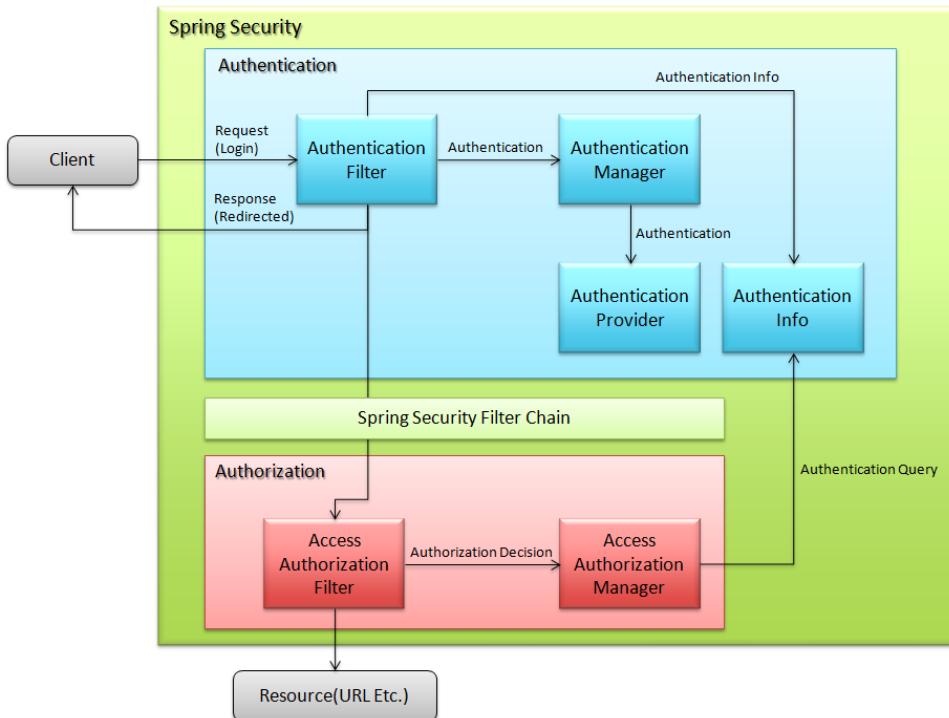


Figure.6.1 Picture - Spring Security Overview

For the details on how to use authentication in Spring Security, refer to [\[coming soon\] Authentication](#).

Password hashing

In password hashing, the original password is replaced with a hash value that is derived from the plaintext password using hash function.

For the details on how to use it in Spring Security, refer [Password Hashing](#).

Authorization

Authorization is the functionality to verify whether the authenticated user is allowed to use the resource that he is trying to access,
using access control process.

For the details on how to use authorization in Spring Security, refer to [\[coming soon\] Authorization](#).

6.1.2 How to use

Following settings need to be defined for using Spring Security.

pom.xml settings

To use Spring Security, following dependency needs to be added to pom.xml.

```
<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-security-core</artifactId>    <!-- (1) -->
</dependency>

<dependency>
    <groupId>org.terasoluna.gfw</groupId>
    <artifactId>terasoluna-gfw-security-web</artifactId>    <!-- (2) -->
</dependency>
```

Sr. No.	Description
(1)	terasoluna-gfw-security-core is not web dependent. As a result, when using from a domain layer project, only terasoluna-gfw-security-core should be added to dependency.
(2)	terasoluna-gfw-web provides web related functionalities. It is dependent on terasoluna-gfw-security-core as well. Hence, for Web projects, only terasoluna-gfw-security-web should be added to dependency.

Web.xml settings

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>    <!-- (1) -->
        classpath*:META-INF/spring/applicationContext.xml
        classpath*:META-INF/spring/spring-security.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<filter>
    <filter-name>springSecurityFilterChain</filter-name>    <!-- (2) -->
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>    <!-- (3) -->
</filter>
<filter-mapping>
```

```
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern> <!-- (4) -->
</filter-mapping>
```

Sr. No.	Description
(1)	In addition to applicationContext.xml, add the Spring Security configuration file to the class path in contextConfigLocation. In this guideline, it is “spring-security.xml” file.
(2)	filter-name should be defined as the Bean name to be used internally in Spring Security, namely, “springSecurityFilterChain”.
(3)	Spring Security filter settings to enable various functionalities.
(4)	Enable the settings for all requests.

spring-security.xml settings

spring-security.xml is placed under the path specified in web.xml.

Normally, it is set in src/main/resources/META-INF/spring/spring-security.xml.

Please refer subsequent chapters for detailed explanation, as the following example is just a template.

- spring-mvc.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:sec="http://www.springframework.org/schema/security"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/security
                           http://www.springframework.org/schema/security/spring-security.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd">
    <sec:http use-expressions="true"> <!-- (1) -->
    <!-- omitted -->
    </sec:http>
</beans>
```

Sr. No.	Description
(1)	Spring EL expressions of access attribute can be enabled by setting use-expressions="true".

Note: For the Spring EL expressions enabled by use-expressions="true", please refer the following.

[Expression-Based Access Control](#)

6.2 [coming soon] Spring Security Tutorial

Coming soon ...

6.3 [coming soon] Authentication

Coming soon ...

6.4 Password Hashing

6.4.1 Overview

Password hashing is one of the key considerations when designing a secure application.

In a normal system, hashing is mandatory as it is not possible to register a password in plaintext.

However, if a weak algorithm is selected, the original data of hashed password can be easily cracked by “Offline Brute Force Attack” or “Rainbow Crack” etc.

Spring Security provides

`org.springframework.security.crypto.PasswordEncoder` interface, as the password hashing mechanism.

The following classes,

- `org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder`
- `org.springframework.security.crypto.password.StandardPasswordEncoder`

etc. are provided as its implementation classes.

As `PasswordEncoder` mechanism, password is hashed by `encode (String rawPassword)` method and the encoded password is verified by

the `matches (String rawPassword, String encodedPassword)` method.

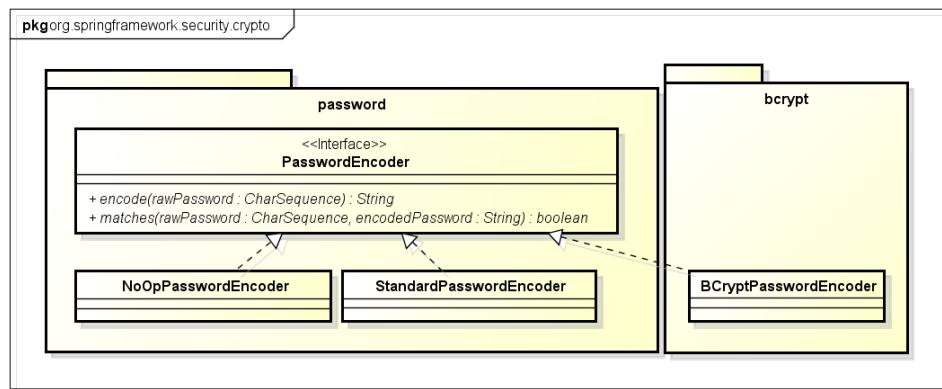


Figure.6.2 Picture - PasswordEncoder Class Diagram

6.4.2 How to use

Implementation classes of PasswordEncoder provided by Spring Security, are explained in this section.

List of PasswordEncoder implementation classes

PasswordEncoder implementation classes	Overview
<code>org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder</code>	Encoder that performs hashing using “bcrypt” algorithm
<code>org.springframework.security.crypto.password.StandardPasswordEncoder</code>	Encoder that performs hashing with “SHA-256” algorithm + 1024 rounds of stretching
<code>org.springframework.security.crypto.password.NoOpPasswordEncoder</code>	Encoder that does not perform hashing (for testing)

It is recommended to use `BCryptPasswordEncoder` when hashing is not required. However, the calculation time taken by `BCryptPasswordEncoder` to improve counter-attacks being more, if the performance requirements at the time of authentication are not fulfilled, `StandardPasswordEncoder` should be considered.

In case of any restrictions concerning salt and hashed algorithm owing to the relation with existing system, implementation class of `org.springframework.security.authentication.encoding.PasswordEncoder` interface, which will be described later, should be used. For details, refer [How to extend](#).

BCryptPasswordEncoder

`BCryptPasswordEncoder` is the class that implements `PasswordEncoder` and provides password hashing.

It is the encoder which uses random 16 bytes of salt and bcrypt algorithm.

Note: In Bcrypt algorithm, number of calculations have been intentionally increased to more than the calculations of the standard algorithms. Therefore, compared to standard algorithms (SHA, MD5), it has strong features to resist “offline brute force attack”.

Configuration example of BCryptPasswordEncoder

- applicationContext.xml

```
<bean id="passwordEncoder"
      class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" /> <!-- (1)
```

Sr. No.	Description
(1)	<p>Specify BCryptPasswordEncoder in passwordEncoder class.</p> <p>Number of salt hash rounds can be specified as constructor argument. Values from 4 to 31 can be set.</p> <p>Longer the salt value, stronger the hashing. However, as number of calculations increase exponentially, it is necessary to exercise caution from performance perspective.</p> <p>When no value is specified, “10” is set.</p>

Tip: It is described later in ‘How to extend’; however, DaoAuthenticationProvider can set the implementation class of `org.springframework.security.crypto.password.PasswordEncoder` as well as the implementation class of `org.springframework.security.authentication.encoding.PasswordEncoder`. Therefore, when the existing PasswordEncoder (authentication package) is changed to a new PasswordEncoder, it can be handled only by changing the passwordEncoder of DaoAuthenticationProvider, after changing the user password.

Warning: When DaoAuthenticationProvider is set in authentication provider and `UsernameNotFoundException` is thrown, without letting the person operating the system know that user does not exist, password is intentionally hashed after `UsernameNotFoundException` is thrown. (Side channel attack countermeasure)

To create values for this hashing, `encode` method is once executed internally when starting the application.

Warning: When SecureRandom is used in Linux environment, the process may be delayed or timeout may occur. The cause of this issue is random number generation and is described in the following Java Bug Database.

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6202721

It has been corrected in the JDK 7 version b20 and above.

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6521844

This issue can be avoided by setting the following as JVM boot arguments.

`-Djava.security.egd=file:///dev/urandom`

- Java class

```

@.Inject
PasswordEncoder passwordEncoder; // (1)

public String register(Customer customer, String rawPassword) {
    // omitted
    // Password Hashing
    String password = passwordEncoder.encode(rawPassword); // (2)
    customer.setPassword(password);
    // omitted
}

public boolean matches(Customer customer, String rawPassword) {
    return passwordEncoder.matches(rawPassword, customer.getPassword()); // (3)
}

```

Sr. No.	Description
(1)	Inject PasswordEncoder for which bean definition is carried out.
(2)	Password hashing example By specifying the plaintext password as an argument of encode method, hashed password is returned.
(3)	Password verification example By specifying plaintext password as the first argument and hashed password as the second argument, 'matches' method checks whether both the passwords match.

StandardPasswordEncoder

StandardPasswordEncoder uses SHA-256 as the hashing algorithm and performs 1024 rounds of stretching.

It assigns randomly generated salt of 8 bytes.

encode(String rawPassword) method and matches(String rawPassword, String encodedPassword) method of the StandardPasswordEncoder are described below.

encode(String rawPassword) method

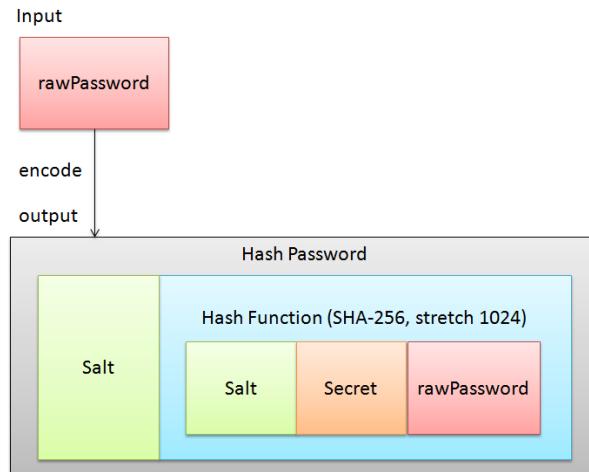


Figure.6.3 Picture - encode method

Hashing is carried out by randomly generated salt of 8 bytes + secret key + the password specified in argument.

Return value of method is the value wherein, salt used for hashing is assigned in the beginning of the above hashed value.

matches(String rawPassword, String encodedPassword) method

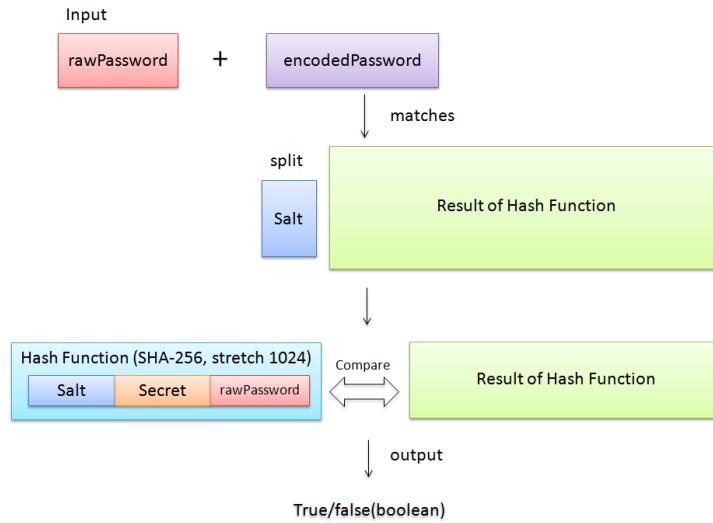


Figure.6.4 Picture - matches method

The salt passed from argument and assigned at the beginning of encodedPassword, is split and the two values

namely, the value hashed by salt + secret + rawPassword
and the value without salt assigned at the beginning of encodedPassword, are compared.

Configuration example of StandardPasswordEncoder

- applicationContext.xml

```
<bean id="passwordEncoder"
      class="org.springframework.security.crypto.password.StandardPasswordEncoder">
    <!-- from properties file -->
    <constructor-arg value="${password.encoder.secret}" /> <!-- (1) -->
</bean>
```

Sr. No.	Description
(1)	<p>Specify the secret key for hashing. When specified, password is hashed with “internally generated salt” + “specified secret key” + “password”. It is recommended to specify secret key, as the strength against rainbow table attack reduces if not specified.</p> <p>About secret key Secret key should be handled as confidential information. Therefore, instead of specifying it directly in the Spring Security configuration file, fetch it from properties file or environment variable etc. Here, example of fetching the secret key from properties file is enabled. Further, care should be taken regarding the storage location of properties file in a production environment.</p>

Tip:

When secret key is fetched from environment variables

It can be fetched by performing the following settings in <constructor-arg> of StandardPasswordEncoder bean definition.

```
<bean id="passwordEncoder"
      class="org.springframework.security.crypto.password.StandardPasswordEncoder">
    <!-- from environment variable -->
    <constructor-arg value="#{systemEnvironment['PASSWORD_ENCODER_SECRET']}"/> <!-- (1) -->
</bean>
```

Sr. No.	Description
(1)	Fetch value from environment variable PASSWORD_ENCODER_SECRET.

Refer to [Configuration example of BCryptPasswordEncoder](#), as example of Java class is the same as BCryptPasswordEncoder.

NoOpPasswordEncoder

NoOpPasswordEncoder is the encoder that returns the specified value as a string without any change. It must not be used except for the un-hashed strings to be used at the time of unit testing etc.

As its configuration example is same as that of BCryptPasswordEncoder, it is omitted here.

6.4.3 How to extend

Depending on business requirements, it may not be possible to implement password hashing using the class that implements PasswordEncoder mentioned above.

Especially when the hashing system used in the existing account information is to be followed, often the PasswordEncoder mentioned above, does not fulfill the requirements.

For example we may consider a case wherein the existing hashing system is as follows:

- Algorithm used is SHA-512.
- There are 1000 rounds of stretching.
- Salt is stored in account table column and needs to be passed externally from PasswordEncoder.

In this case, it is recommended to use the class that implements

`org.springframework.security.authentication.encoding.PasswordEncoder` of a different package

rather than the class that implements

`org.springframework.security.crypto.password.PasswordEncoder`.

Warning: In versions prior to Spring Security 3.1.4, the class that implements org.springframework.security.authentication.encoding.PasswordEncoder was used for hashing. However, it has been deprecated from Spring Security version 3.1.4 onwards. Therefore, it differs from the pattern recommended by Spring.

Example where ShaPasswordEncoder is used

When business requirements are as follows wherein,

SHA-512 algorithm is used and 1000 rounds of stretching are performed.

It is explained here with an example of authentication process that uses DaoAuthenticationProvider described in *[coming soon] Authentication*.

- applicationContext.xml

```
<bean id ="passwordEncoder"
      class="org.springframework.security.authentication.encoding ShaPasswordEncoder"> <!-- (1)
      <constructor-arg value="512" /> <!-- (2) -->
      <property name="iterations" value="1000" /> <!-- (3) -->
</bean>
```

Sr. No.	Description
(1)	Specify org.springframework.security.authentication.encoding ShaPasswordEncoder as the passwordEncoder. The class to be specified in passwordEncoder should change according to the algorithm to be used.
(2)	Set the SHA algorithm type as constructor argument. The values “1, 256, 384, 512” can be set. When omitted, “1” is set.
(4)	Specify the number of stretching rounds at the time of hashing. When omitted, it is 0.

- spring-mvc.xml

```
<bean id="authenticationProvider"
      class="org.springframework.security.authentication.dao.DaoAuthenticationProvider">
      <!-- omitted -->
```

```
<property name="saltSource" ref="saltSource" /> <!-- (1) -->
<property name="userDetailsService" ref="userDetailsService" />
<property name="passwordEncoder" ref="passwordEncoder" /> <!-- (2) -->
</bean>

<bean id="saltSource"
      class="org.springframework.security.authentication.dao.ReflectionSaltSource"> <!-- (3) -->
    <property name="userPropertyToUse" value="username" /> <!-- (4) -->
</bean>
```

Sr. No.	Description
(1)	When salt is to be defined externally, set BeanId of the class that implements org.springframework.security.authentication.dao.SaltSource. In this example, org.springframework.security.authentication.dao.ReflectionSaltSource that fetches the value set in user information class by reflection, is defined.
(2)	Specify org.springframework.security.authentication.encoding.ShaPasswordEncoder as the passwordEncoder. The class to be specified in passwordEncoder should change according to the algorithm to be used.
(3)	Specify org.springframework.security.authentication.dao.SaltSource that decides how to create salt. Here, ReflectionSaltSource resource that fetches UserDetails object property by reflection, is used.
(4)	username property of UserDetails object is used as salt.

- Java class

```
@Inject
PasswordEncoder passwordEncoder;

public String register(Customer customer, String rawPassword, String userSalt) {
    // omitted
    String password = passwordEncoder.encodePassword(rawPassword,
```

```
        userSalt); // (1)
customer.setPassword(password);
// omitted
}

public boolean matches(Customer customer, String rawPassword, String userSalt) {
    return passwordEncoder.isPasswordValid(customer.getPassword(),
        rawPassword, userSalt); // (2)
}
```

Sr. No.	Description
(1)	To hash password, specify password and salt string as the argument of encodePassword method of the class that implements <code>org.springframework.security.authentication.encoding.PasswordEncoder</code> .
(2)	To verify password, Using isPasswordValid method, hashed password, plain text password and salt string are specified as the argument and the hashed password and plaintext passwords are compared.

6.4.4 Appendix

Note: About stretch

By repeating the hash function computation, information regarding password to be stored can be repeatedly encoded. This is done to prolong the time required to crack a password, and thus acts as a countermeasure against the brute force attack. However, since stretching affects system performance, it is necessary to decide the stretch count on considering the system performance.

Note: About salt

Salt is the string assigned to the original data to be encoded. By assigning salt to a password, the length of the password is increased and thus makes it difficult to crack passwords using rainbow attacks etc. Further, if the same salt is used for multiple users and if there are users who have the same password, it will be obvious from the hash value that same password is used. Therefore, it is recommended to use a different salt (random value etc.) for each user.

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
&	&
<	<
>	>
"	"
,	'

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.

The screenshot shows a registration form with the following fields:

- Birthday(Required): 1980 / 01 / 01
- Job(Required): <script>alert("XSS Attack")</script>
- E-mail: (empty)
- Tel(Required): 09099999999 (Half-width 10 digits to 13 digits numeric only) Ex. 262-0002

Figure.6.5 Picture - Input HTML Tag

It is recognized as <script> tag and dialog box is displayed.

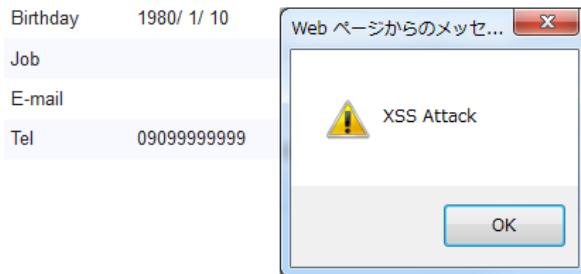


Figure.6.6 Picture - No Escape Result

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.

The screenshot shows a registration form titled 'Register Customer'. It includes fields for Birthday (set to 1980/01/01), Job (containing the value '<script>alert("XSS Attack")</script>'), E-mail (empty), and Tel (set to 09099999999). A note below the Tel field specifies '(Half-width 10 digits to 13 digits numeric only) Ex. 262-0002'.

Figure.6.7 Picture - Input HTML Tag

By escaping special characters, input value is output as is without being recognized as <script> tag.

Birthday	1980/ 1/ 10
Job	<script>alert("XSS Attack")</script>
E-mail	
Tel	09099999999

Figure.6.8 Picture - Escape Result

Output result

```
<!-- omitted -->
<tr>
    <td>Job</td>
    <td>&lt;script&gt;alert("XSS Attack")&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.

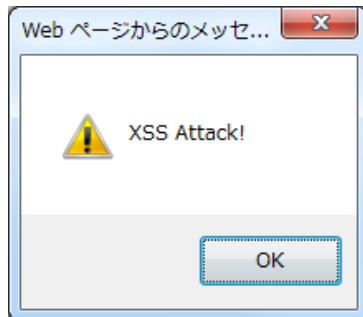


Figure.6.9 Picture - No Escape Result

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 <code>f:js()</code> 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</script>';
    document.write(aaa);
</script>
```

Event handler Escaping

To escape the value of event handler of javascript, `f:h:js()` 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.6.10 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 <code>f:hjs()</code> is set as an argument of javascript event handler.

XSS dialog is not output on mouse over.



Figure.6.11 Picture - Escape Result

Output result

```
<!-- omitted -->
<input type="text" onmouseover="alert('output is '); alert('XSS Attack!'); // .
<!-- omitted -->
```

6.7 [coming soon] CSRF(Cross Site Request Forgeries) Counter-measures

Coming soon ...

7

Appendix

7.1 Create New Project From Blank

Method to create a new project from blank is explained below.

- Prerequisites
 - Operational Spring Tool Suite.
 - Maven should be executable in command line.
 - Internet connectivity.

The contents in this chapter have been validated on the following versions.

Product	Version
JDK	1.7.0_40
Spring Tool Suite (STS)	3.4.0
Maven	3.0.4

Note: If proxy server is required to connect to the internet, STS proxy settings and [Maven proxy settings](#) are required.

7.1.1 Creating a new project

1. Move to the target folder using command prompt.

```
cd C:\work
```

2. Create a project template using Maven archetype.

```
mvn archetype:generate -B^
-DarchetypeCatalog=http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-
-DarchetypeGroupId=org.terasoluna.gfw.blank^
```

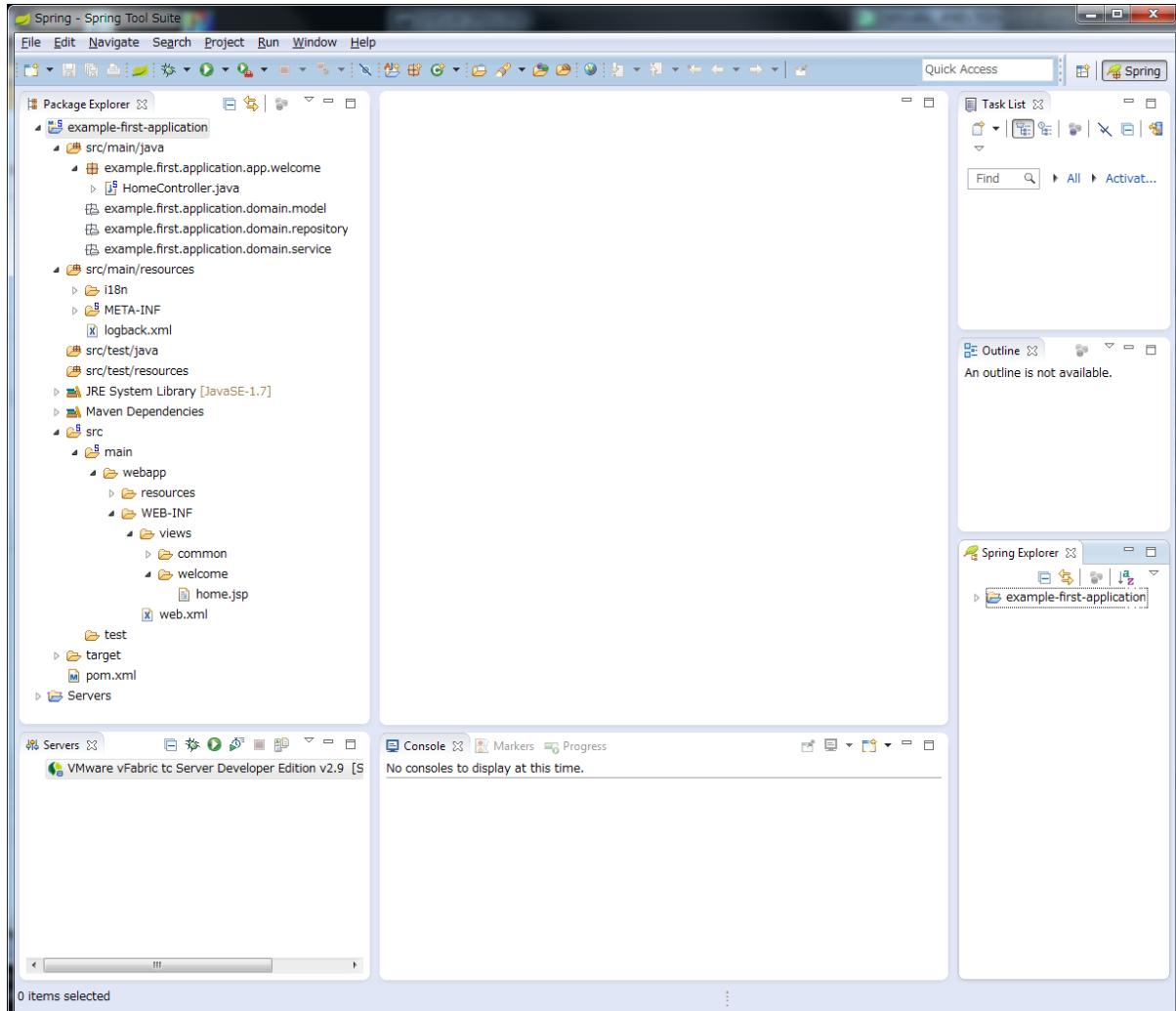
```
-DarchetypeArtifactId=terasoluna-gfw-web-blank-archetype^
-DarchetypeVersion=1.0.1.RELEASE^
-DgroupId=example.first.application^
-DartifactId=example-first-application^
-Dversion=1.0.0-SNAPSHOT
```

Parameter	Description
-B	batch mode (skips interaction)
-DarchetypeCatalog	Creates project using Maven archetype. Specifies TERASOLUNA Global Framework repository.
-DarchetypeGroupId	terasoluna.gfw blank project GroupID is fixed to org.terasoluna.gfw.blank
-DarchetypeArtifactId	terasoluna-gfw-web-blank-archetype = when creating project without using JPA, MyBatis2 terasoluna-gfw-web-blank-jpa-archetype = when creating project using JPA terasoluna-gfw-web-blank-mybatis2-archetype = when creating project using MyBatis2
-DarchetypeVersion	Specifies terasoluna.gfw version.
-DgroupId	Specifies the groupId of the created project.
-DartifactId	Specifies the artifactId of the created project.
-Dversion	Specifies the version of the created project.

3. Import the project in Spring Tool Suite.

Specify the project created using Maven archetype in [STS] -> [File] -> [Import] -> [Maven] -> [Existing Maven Projects] -> [Browse...]. -> Confirm that the check box of one pom.xml is checked and click [Finish]

The status will be as given below.



4. Add the created project to the Application Server.

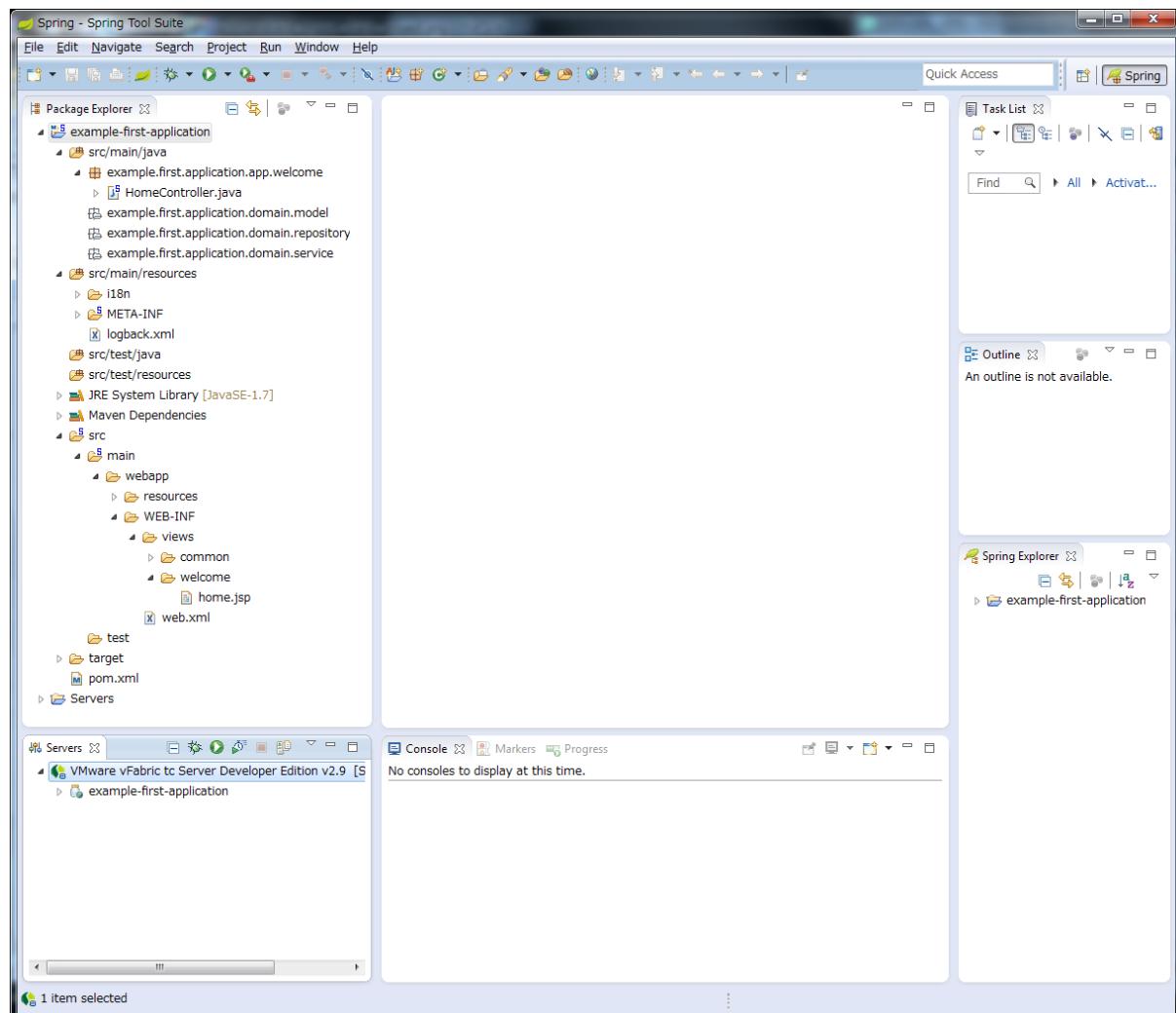
Here, VMware vFabric tc Server Developer Edition v2.9 which is provided with STS is used by default.

Right click on [VMware vFabric tc Server Developer Edition v2.9] -> [Add and Remove] -> select created project and [Add] -> [Finish]

The status will be as given below.

5. Launch the Application Server.

Launch by clicking Start button of server. If the following is displayed on Console, it can be considered that server is launched successfully.



```
FrameworkServlet 'appServlet': initialization completed
```

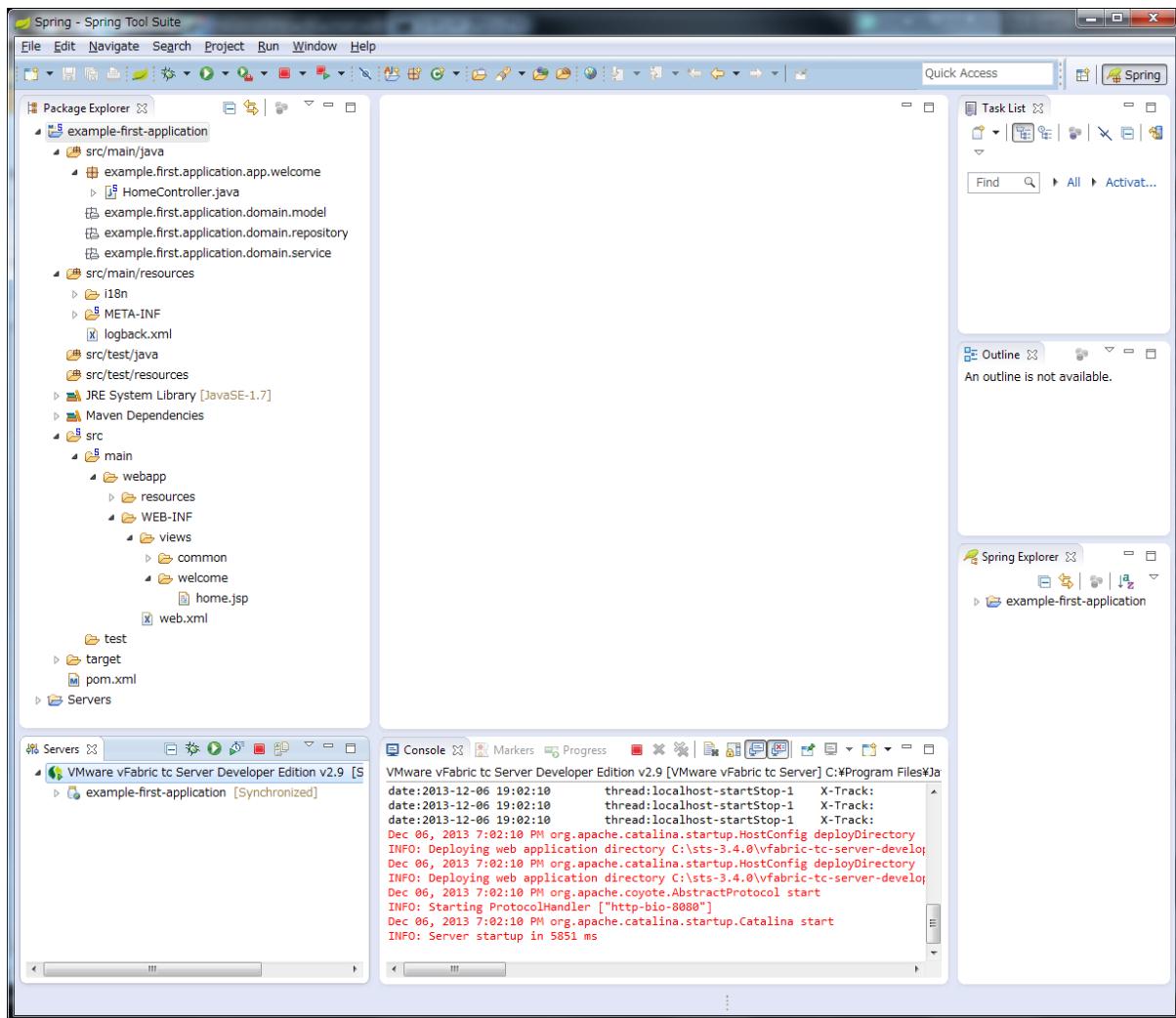
As for the following example, log is output on Console; however “FrameworkServlet ‘appServlet’: initialization completed” is displayed on the line above the line of red characters (It is not displayed on screen capture).

6. Access the launched application.

Access <http://localhost:8080/example-first-application/> on browser.

The screen given below is displayed.

If “Hello world!” is displayed, it can be considered that a new project is created successfully. Necessary functions should be added as per the guidelines.



Hello world!

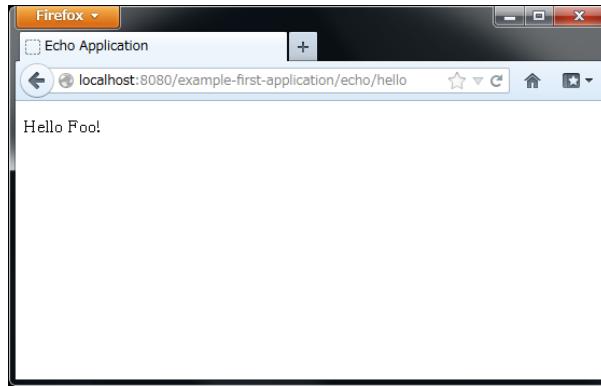
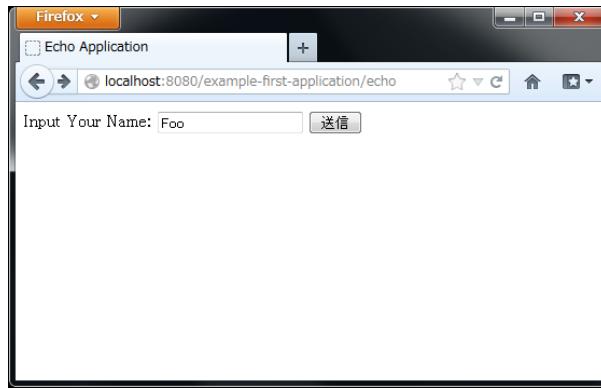
The time on the server is 2013/12/06 19:05:36 JST.

7.1.2 Creating simple Echo project

The steps are basically same as [Create an Echo Application](#) explained in [First application based on Spring MVC](#), so source is skipped.

In [First application based on Spring MVC](#), <context:component-scan base-package="com.example.helloworld" /> is set in spring-mvc.xml; however, when created from Blank Project, it is set as <context:component-scan base-package="example.first.application.app" />.

EchoController should be created in example.first.application.app.echo package.



Todo

TBD

Currently, way of creating a single project structure is explained; however it is assumed to be mainly used for verification purpose. Actually, it is necessary to build the project using [multi-project structure](#). How to create a multi-project structure will be explained later.

Warning: The following settings defined in pom.xml of Blank project are simply for testing the sample application. Use of these settings in actual development is not assumed. In the actual project, these settings should be deleted.

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

7.2 Maven Repository Management using NEXUS

Sonatype [NEXUS](#) is the package repository manager software. OSS version as well as commercial version of NEXUS is available. However its OSS version also has adequate functionalities.

This chapter explains the role and configuration method of OSS version of NEXUS.

7.2.1 Why NEXUS?

When there is only one developer, a central repository on internet and a local repository on developer's machine can be developed using Maven or ant+ivy.

However, when a Java application is to be divided into multiple sub-projects and development is to be carried out in a team, library dependency resolution becomes complex; hence this dependency resolution needs to be automated. For this, availability of package repository server is essential.

The following package repositories are required in Java application development project.

- **Proxy repository** creates proxy to access external repository server including the central repository
- **3rd party repository** for distributing the artifacts provided by others within the organization that are not available in the repository on internet
- **Private repository** for storing the artifacts developed within the project
- **Group repository** for consolidating access to artifacts of different multiple repositories into a single repository URL

In case of NEXUS, operations of such multiple repositories can be easily managed.

7.2.2 Install and Start-up

The machine on which NEXUS is to be installed should satisfy the following conditions.

- JRE6 or later should be installed
- Http access to the following URLs
- URL starting with <http://repo1.maven.org/> (Central repository)
- URL starting with <http://repo.terasoluna.org/> (Terasoluna repository)

The installation procedure is as follows:

1. Download [NEXUS OSS](#) and deploy archive.
2. Start NEXUS by executing bin/nexus or bin/nexus.bat.

3. Access [http://\[IP or FQDN\]:8081/nexus/](http://[IP or FQDN]:8081/nexus/) and check whether welcome screen of NEXUS is displayed.

Some repositories are provided by default. Except for a few cases, they are used as is for development. Repository list is displayed on clicking Repositories on the menu on the left side of screen.

The screenshot shows the Sonatype Nexus web interface. On the left, there is a sidebar with the following navigation items:

- Sonatype™ Servers
- Nexus (selected)
- Artifact Search
- Advanced Search
- Views/Repositories (highlighted with a red oval)
- Repositories (highlighted with a red oval)
- Help

The main content area is titled "Welcome" and "Repositories". It features a "Refresh" button and a "User Managed Repositories" dropdown. Below these are two tabs: "Welcome" and "Repositories" (selected). The "Repositories" tab displays a table of available repositories:

Repository	Type	Quality	Format
Public Repositories	group	ANALYZE	maven2
3rd party	hosted	ANALYZE	maven2
Apache Snapshots	proxy	ANALYZE	maven2
Central	proxy	ANALYZE	maven2
Central M1 shadow	virtual	ANALYZE	maven1
Codehaus Snapshots	proxy	ANALYZE	maven2
Releases	hosted	ANALYZE	maven2
Snapshots	hosted	ANALYZE	maven2

- **Central** = This repository plays a role of proxy to the central repository on internet (<http://repo1.maven.org/maven2/>).
- **3rd party** = This repository stores third-party libraries required in development but not available in the repositories on internet.
- **Releases** = This repository stores the work products of release version of the applications developed internally.
- **Snapshots** = This repository stores the work products of SNAPSHOT version of the applications developed internally.
- **Public Repositories** This group repository is used for enabling access to the above repositories through a single URL.

7.2.3 Add TERASOLUNA Global Framework repository

When an application is to be developed using TERASOLUNA Global Framework, TERASOLUNA Global Framework repository needs to be added in addition to the above repositories.

Todo

Add proxy repository to <http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-releases/> and

<http://repo.terasoluna.org/nexus/content/repositories/terasoluna-gfw-3rdparty/> . Then write the method to add it to public repository group and provide a screen capture along with it.

7.2.4 settings.xml

In order to use the created NEXUS using Maven command, settings.xml file needs be created in the home directory of local development environment of the user.

- Windows: C:/Users/[OSaccount]/.m2/settings.xml
- Unix: \$HOME/.m2/settings.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>

    <mirrors>
        <mirror>
            <id>myteam-nexus</id>
            <mirrorOf>*</mirrorOf>
            <!-- CHANGE HERE by your team own nexus server -->
            <url>http:// IP or FQDN /nexus/content/groups/public </url>
        </mirror>
    </mirrors>

    <activeProfiles>
        <activeProfile>myteam-nexus</activeProfile>
    </activeProfiles>

    <profiles>
        <profile>
            <id>myteam-nexus</id>
            <repositories>
                <repository>
                    <id>central</id>
                    <url>http://central</url>
                    <releases><enabled>true</enabled></releases>
                    <snapshots><enabled>true</enabled></snapshots>
                </repository>
            </repositories>
            <pluginRepositories>
                <pluginRepository>
                    <id>central</id>
                    <url>http://central</url>
                    <releases><enabled>true</enabled></releases>
                    <snapshots><enabled>true</enabled></snapshots>
                </pluginRepository>
            </pluginRepositories>
        </profile>
    </profiles>
```

```
</profiles>

</settings>
```

Note: Additionally, also refer to: [Configuring Maven to Use a Single Nexus Group / Documentation Sonatype.com](#)

7.2.5 mvn deploy how to

Use mvn deploy command to upload jar/war file to the package repository (NEXUS) as an artifact.

A state in which anyone can deploy the application in package repository should be avoided as it causes confusion. Therefore, it is desirable that mvn deployment for package repository is possible only by Jenkins.

In addition to the contents mentioned earlier, add the following to .m2/settings.xml under the home directory of the user executing Jenkins or Jenkins server.

```
<servers>
  <server>
    <id>releases</id>
    <username>deployment</username>
    <password>deployment123</password>
  </server>
  <server>
    <id>snapshots</id>
    <username>deployment</username>
    <password>deployment123</password>
  </server>
</servers>
```

‘deployment’ is the account (set in NEXUS by default) having deployment permission and its password is deployment123. It is recommended to change the password in advance on NEXUS GUI screen.

Note: To avoid saving the password in plain text in settings.xml, it is advisable to use password encryption function of Maven. Refer to [Maven - Password Encryption](#) for details.

Carry out the mvn deployment procedure in the build job of Jenkins as follows:

Todo

Provide screen capture of build job of Jenkins

7.2.6 pom.xml

In case of the project managed in Maven, package repository in which artifact is stored, should be specified using <distributionManagement> tag of pom.xml.

```
<distributionManagement>
  <repository>
    <id>releases</id>
    <!-- CHANGE HERE by your team nexus server -->
    <url>http://192.168.0.1:8081/nexus/content/repositories/releases/</url>
  </repository>
  <snapshotRepository>
    <id>snapshots</id>
    <!-- CHANGE HERE by your team nexus server -->
    <url>http://192.168.0.1:8081/nexus/content/repositories/snapshots/</url>
  </snapshotRepository>
</distributionManagement>
```

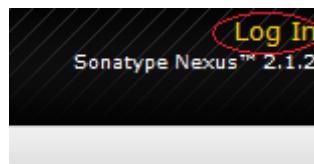
The mvn deploy command uploads the artifact with HTTP PUT for the URL specified using <distributionManagement> tag.

7.2.7 Upload 3rd party artifact (ex. ojdbc6.jar)

In the 3rd party repository, store the artifact which is not disclosed in external remote repository.

Typical example is JDBC driver (ojdbc*.jar) of oracle. Oracle should be used as RDBMS; however, central repository is not stored in the public repository on internet. Therefore, it should be stored in the package repository in the organization.

1. Login as admin user. (Default password is admin123)



2. Select 3rdParty repository and select **Artifact Upload** tab.

3. Enter GAV information. (GAV = groupId, artifactId, version)

4. Select ojdbc6.jar on local PC and click **Add Artifact** button.

The screenshot shows the Nexus Repository Manager interface. The top navigation bar includes 'Welcome', 'Repositories', 'Refresh', 'Add...', 'Delete', 'Trash...', and 'User Managed Repositories'. The main table lists various repositories:

Repository	Type	Quality	Format	Policy	Reposit
Public Repositories	group	ANALYZE	maven2		
3rd party	hosted	ANALYZE	maven2	Release	In Servi
Apache Snapshots	proxy	ANALYZE	maven2	Snapshot	In Servi
Central	proxy	ANALYZE	maven2	Release	In Servi
Central M1 shadow	virtual	ANALYZE	maven1	Release	In Servi
Codehaus Snapshots	proxy	ANALYZE	maven2	Snapshot	In Servi
Releases	hosted	ANALYZE	maven2	Release	In Servi
Snapshots	hosted	ANALYZE	maven2	Snapshot	In Servi

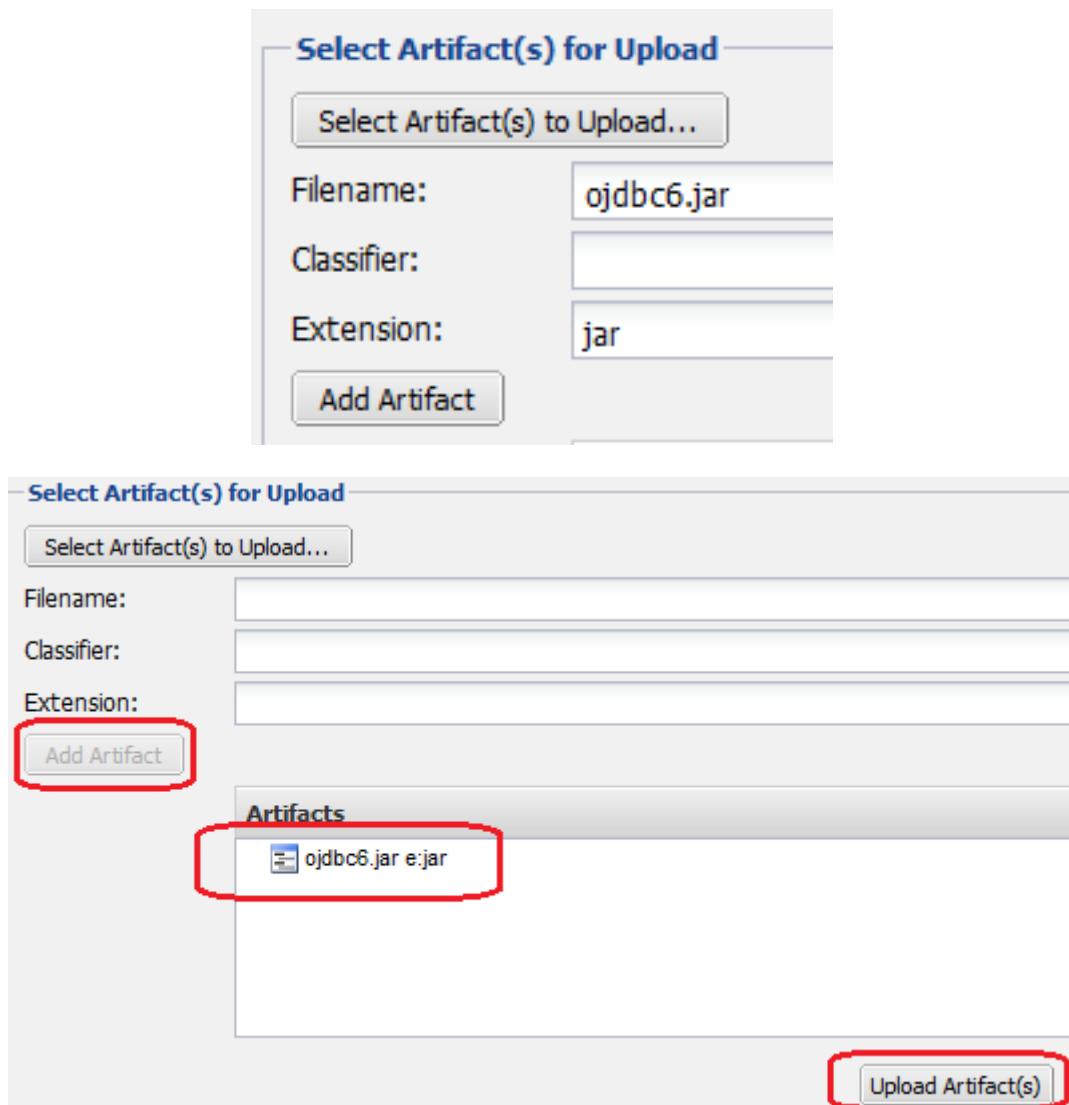
A modal dialog titled '3rd party' is open, showing tabs for 'Browse Storage', 'Browse Index', 'Configuration', 'Mirrors', 'Summary', and 'Artifact Upload'. The 'Artifact Upload' tab is highlighted with a red box. Below it, a section titled 'Select GAV Definition Source' contains a dropdown labeled 'GAV Definition: * Select...' and a note: 'Select a source for the GAV definition. GAV can be specified either manually or from a POM file. Thes'.

The 'Select GAV Definition Source' dialog is also shown, containing fields for 'GAV Definition: * GAV Parameters' (with a dropdown arrow), 'Auto Guess: ?' (checked), 'Group: * com.oracle', 'Artifact: * ojdbc6', 'Version: * 11.2.0.3', and 'Packaging: * jar'.

5. At the end, click **Upload Artifact(s) button to save the jar file in repository.

With this uploading is completed.

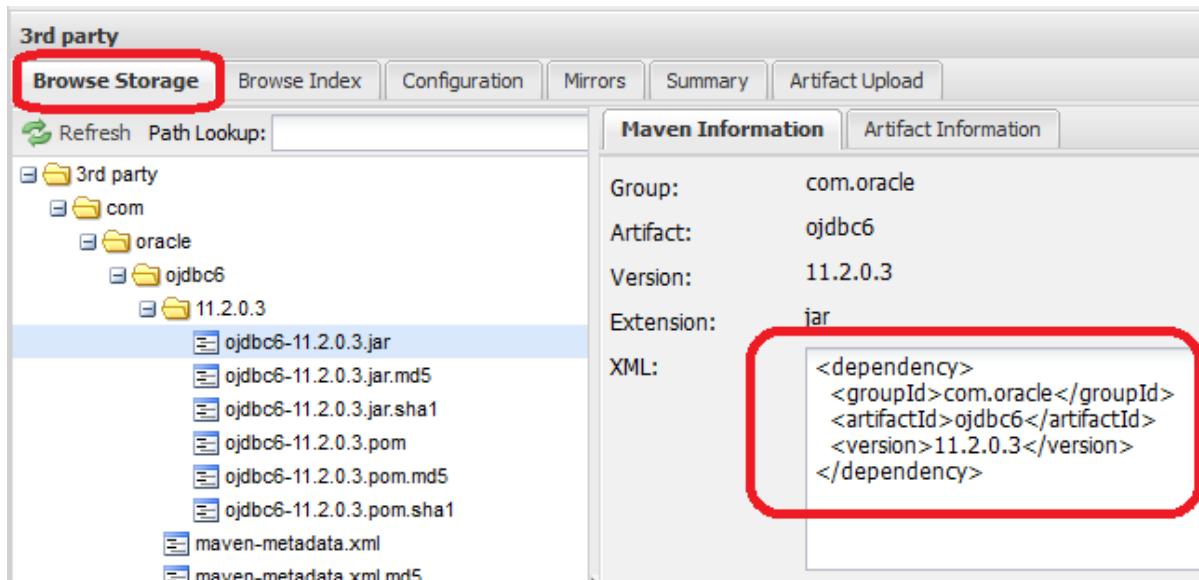
Note: Uploading artifacts using NEXUS GUI screen is a manual task which can easily lead to operational errors. Hence it is not recommended. The method explained here should be used only for simple configurable libraries having 1 or more 3rd party files such as ojdbc6.jar. **mvn deploy** command should be used for other cases.



use artifact

In order to add ojdbc6 of 3rd party repository to project dependency management, just add dependency tag to pom.xml of the corresponding project.

If the target artifact is selected from **Browse Storage** tab, sample of dependency tag is displayed on the right side of the screen. It just needs to be copied and pasted in pom.xml.



7.3 Removing Environment Dependency

Todo

Rewrite.

The problem of environment dependency always arises in Web application development projects.

If `jdbcurl=hdbe:mysql:127.0.0.1...` is written in `datasource.xml` file or `level="DEBUG"` is written in `logback.xml` file and if these files are included in war file, then the Web application runs normally only on your local PC; it cannot be released on test server.

Surprisingly earlier such simple problems were ignored in most of the development projects. Moreover, the problem of difficulty in running the developed application on test server used to get noticed just before the integration test and its resolution also used to take lot of time.

This chapter explains the rules and some specific ways to overcome the problem of environment dependency.

7.3.1 Objective

All the source codes or binary files to be developed hereafter by your team should run seamlessly in all of the following scenarios.

- Running of application in AP server set in IDE (Eclipse) on all developers' PCs
- Execution of test using JUnit plugin of IDE on all developers' PCs
- Execution of test using build tools (Maven/ant) on all developers' PCs

- Execution of test on CI server
- Packaging on CI server (creation of jar/war files)
- Running of application on test server
- Running of application on production server

7.3.2 Rules

To achieve the above objective, consider the following project structure.

1. Make sure to have a multi-project structure.
2. As far as possible, consolidate the configuration files (ex. logback.xml, jdbc.properties) having environment dependency in one project. **Hereafter, this project is expressed as *-env.**
 - ex. terasoluna-tourreservation-env
3. Projects other than *-env never have a setting for environment dependency.
 - However, it is allowed to store the environment dependency configuration files for test under src/test/resources.
4. Manage all software packaged binaries by storing them in package repository.
 - Deploy not only *.jar files but also *.war files as work products in package repository. Consequently, these jar/war files should not include environment dependencies.
5. Configure *-env project as follows:
 - Store the configuration values required for operations on the developer's PC in the file under src/main/resources, as default values.
 - Store the configuration files that differ with each environment such as test server, production server, in the folder other than src/main/resources (ex. configs/test-server). Then, use "profile" function of Maven that automatically replaces the configuration values depending on environment in order to build *-env-x.y.z.jar file.

The above structure facilitates proper development in all the scenarios of software development lifecycle.

1. In local development environment, check out both main project and *-env project and include env project in build path of the main project so as to do coding and testing in the local development environment.
2. On CI server, execute the test and perform packaging using a build tool (Maven) and deploy the artifact in package repository whenever required.
3. Application can work on test server and production server by adding *-env project built as per the application release environment, to the main project which is stored in package repository.

For details, refer to [Sample application](#).

7.3.3 Deployment

Deployment in Tomcat

Perform the following procedure to release the Web application in Tomcat.

1. Specify the profile of Maven as per the AP server environment in which the application is to be released and build *-env project.
2. Place *-env-x.y.z.jar file built above in the folder of AP server decided in advance. ex. /etc/foo/bar/abcd-env-x.y.z.jar
3. Unjar the *.war file deployed in package repository under [CATALINA_HOME]/webapps.
4. Add /etc/foo/bar/*.jar to class path using VirtualWebappLoader function of Tomcat.
 - The above setting should be described in [CATALINA_HOME]/conf/[contextPath].xml file.
 - Refer to configs folder of terasoluna-shopping-env sample and <http://tomcat.apache.org/tomcat-7.0-doc/api/org/apache/catalina/loader/VirtualWebappLoader.html> for details.
 - VirtualWebappLoader can also be used with Tomcat6.x.

Note:

- autoDeploy attribute of Host tag of [CATALINA_HOME]/conf/server.xml should be set to false. Otherwise [CATALINA_HOME]/conf/[contextPath].xml gets deleted each time web application is restarted.
 - When autoDeploy is disabled, Web application does not start by just placing the war file in [CATALINA_HOME]/webapps. war file should always be unjarred (unzipped).
-

Deployment to other application server

When releasing the Web application on application servers (Example: Resin, Websphere, Weblogic) where a mechanism for adding a class path for each web application (which is provided in VirtualWebappLoader of Tomcat) is not provided, the method to release it after adding *-env-x.y.z.jar file under WEB-INF/lib of war file is the easiest.

1. Specify profile of Maven as per the AP server environment in which application is to be released and build *-env project.
2. Copy *.war file deployed in the package repository to the working directory.
3. Add it under WEB-INF/lib of war file using add option of jar command as follows.

4. Release foo-x.y.z.war on AP server.

Continuous deployment

Continuous deployment is constantly releasing the target software through continuous looping of project (source code tree) structure, version control, inspection, build operations and lifecycle management.

During development, release the software of SNAPSHOT version in the package repository and development AP server and execute the test. To release the software officially, tagging to source code tree in VCS needs be performed after assigning a version number. In this way, the flow of build and deployment slightly differs in the snapshot release and official release.

To deploy the application on AP server that provides Web service, irrespective of snapshot version or official release version, a group of environment dependency configuration files and *.war file should be deployed in a set as per the target release AP server environment.

Separating the operation of registering libraries (jar, war) without environment dependency settings, in Maven repository and the operation of actually deploying them on AP server facilitates deployment.

Note: In Maven, it is automatically distinguished whether it is a SNAPSHOT version or RELEASE version according to the contents of <version> tag of pom.xml.

- It is considered as SNAPSHOT if it ends with -SNAPSHOT. Example: <version>1.0-SNAPSHOT</version>
- It is considered as RELEASE if it does not end with -SNAPSHOT. Example: <version>1.0</version>

Please note that there are 2 types of repositories in Maven package repository i.e. snapshot repository and release repository with a few limitations.

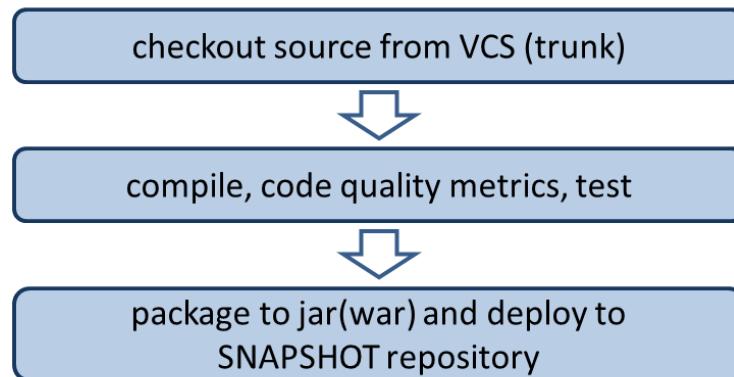
- Software of SNAPSHOT version cannot be registered in release repository. release repository also cannot be registered in snapshot repository.
 - In release repository, artifact having the same GAV information can be registered only once. (GAV=groupId, artifactId, version)
 - In snapshot repository, artifact having the same GAV information can be re-registered many times.
-

Operations of SNAPSHOT version

A simple delivery flow of SNAPSHOT version software is as shown in the following figure.

1. Check out the source code from development trunk.
2. Compile, measure the code metrics and execute test.

Delivery flow for library (SNAPSHOT version)



- In case of compilation error, certain violations of code metrics or in case the test fails, the subsequent operations should be stopped.
 - 3. Upload (mvn deploy) the artifact (jar, war file) on Maven package repository server.
-

Todo

Screen capture needs to be added later on

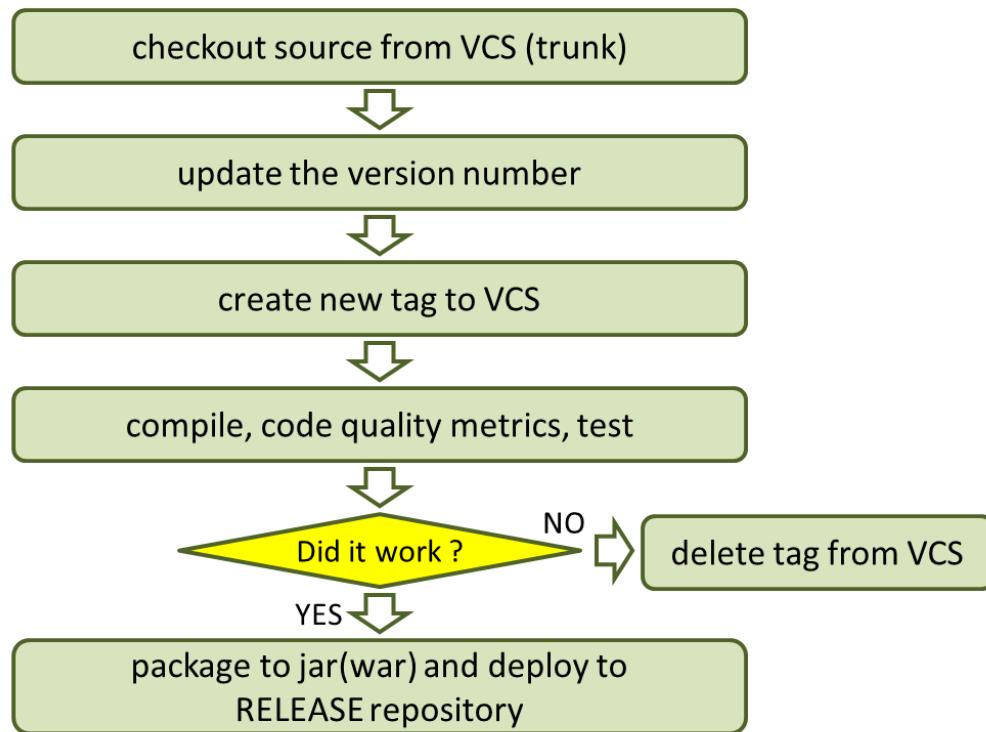
Operations of RELEASE version

In case of official release, since it is necessary to assign the version number, the flow becomes slightly more complex than the SNAPSHOT release.

1. Decide the version number to be assigned for release. (Example : 1.0.1)
 2. Check out the source code from development trunk (or release branch).
 3. Change <version> tag of pom.xml. (Example : <version>1.0.1</version>)
 4. Assign tag to VCS. (Example : tags/1.0.1)
 5. Compile, measure the code metrics and execute test.
 - In case of compilation error, certain violations of code metrics or in case the test fails, the subsequent operations should be stopped.
 - If the test fails, delete the tag of VCS.
 6. Upload (mvn deploy) the artifact (jar, war file) on Maven package repository server.
-

Todo

Delivery flow for library (RELEASE version)



Here, should the version tag of pom.xml of trunk source tree be written at the end till it is replaced by the next SNAPSHOT version and committed?

Note: <version> tag of pom.xml file can be changed in `versions-maven-plugin`.

```
mvn versions:set -DnewVersion=1.0.0
```

Version tag in pom.xml can be edited as <version>1.0.0</version> by the above commands.

Todo

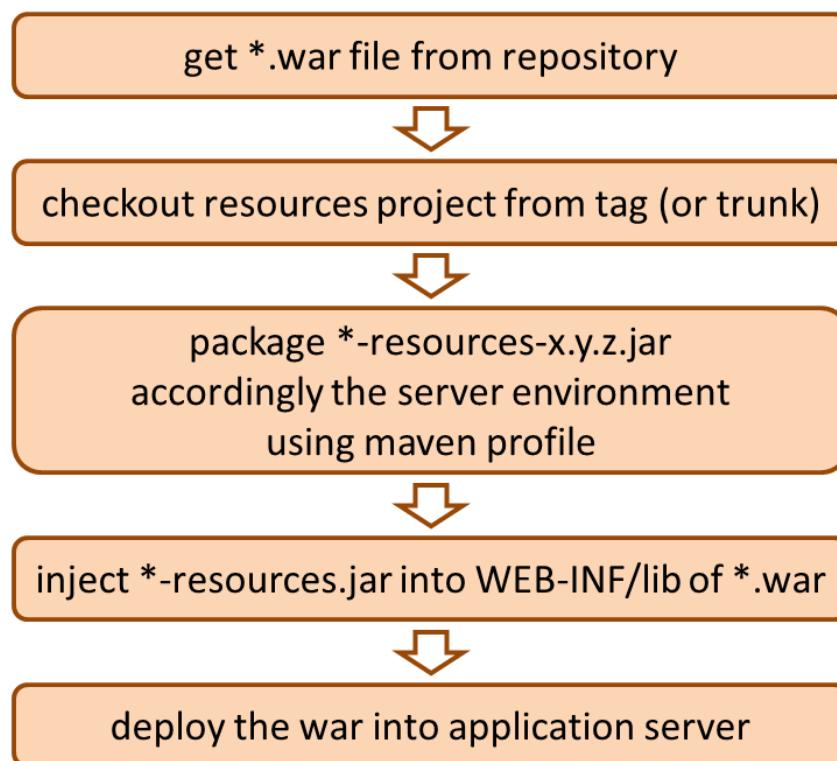
Screen capture needs to be added later on

Release on Application Server

To release the application on AP server that provides Web service, release the *.war file registered in Maven package repository and the group of environment dependency configuration files in a set according to the target release AP server environment. This has same flow irrespective of snapshot release or official release.

1. Download war file of the version to be released from Maven package repository.

Delivery flow for web app to AP server



2. Check out *-resources project (that consolidates environment dependency configuration files) from VCS.
3. Using “profile” function of Maven, replace the contents with group of configuration files according to the target release environment, package the resources project and create *-resources-x.y.z.jar.
4. Add the created *-resources-x.y.z.jar file under WEB-INF/lib folder of war file.
 - In case of Tomcat, instead of adding *-resources-x.y.z.jar to war file, copy it to any path of Tomcat server and specify that path in the extended class path of VirtualWebappLoader. Refer to *Removing Environment Dependency* for details.
5. Deploy the war file on application server.

Note: War file can be downloaded from Maven package repository with “get goal” of maven-dependency-plugin.

```
mvn org.apache.maven.plugins:maven-dependency-plugin:2.5:get \
-DgroupId=com.example \
-DartifactId=mywebapp \
-Dversion=0.0.1-SNAPSHOT \
-Dpackaging=war \
-Ddest=${WORKSPACE}/target/mywebapp.war
```

With this, mywebapp.war file is downloaded under the target directory.

Package of environment dependency configuration files can be added to mywebapp.war file using the following commands.

```
mkdir -p $WORKSPACE/target/WEB-INF/lib
cd $WORKSPACE/target
cp ./mywebapp-resources*.jar WEB-INF/lib
jar -ufv mywebapp.war WEB-INF/lib
```

Todo

Screen capture needs to be added later on

7.4 Project Structure Standard

Todo

Rewrite.

It is recommended to split the software source code tree into multiple projects so as to have a multi-project structure.

Note: Here, it is assumed that Maven is used as a build tool. When Maven is used, the standard multi-project structure consists of Hierarchical project layout. However, Eclipse is used as development environment and it supports only the Flat layout and not the Hierarchical layout, hence use Flat layout.

7.4.1 Simple pattern

A simplest project structure of Web application development project “foo” is as follows:

- foo-parent
- foo-initdb
- foo-domain
- foo-web
- foo-env
- foo-selenium

The details of each project are as follows:

- foo-parent

Project called parent-pom (parent POM). A simple project consisting of only pom.xml file. It never contains other source code or configuration files. By specifying this foo-parent project in the <parent> tag of pom of other project, the common configuration information specified in the parent POM itself can be reflected.

- foo-initdb

Stores the SQL statement to INSERT the initial data and table definition (DDL) of RDBMS. This is also managed as Maven project. By defining *sql-maven-plugin* <<http://mojo.codehaus.org/sql-maven-plugin/>> setting in pom.xml, the execution of DDL statement of any RDBMS and INSERT statement of initial data during the build lifecycle can be automatically executed.

- foo-domain

Stores the classes used as domain layer such as service class, repository class etc. The class of this domain layer is used to create a class of application layer in foo-web.

- foo-web

Stores application layer classes, jsps, configuration files, unit test cases etc. Finally *.war file is created as Web application.

- foo-env

Consolidates only configuration files having environment dependency. foo-web has dependency on foo-env. Refer to [Removing Environment Dependency](#) for details.

- foo-selenium

Stores test cases using [Selenium WebDriver](#).

7.4.2 Complex pattern

The project structure of development project “bar” where 2 Web applications and 1 common library are required is as follows:

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

The details of each project are as follows:

- bar-parent (same as foo-parent)
- bar-initdb (same as foo-initdb)
- bar-common

Stores common library in the project. This is web independent and web related classes are placed under bar-common-web.

- bar-common-web

Stores common web library in the project.

- bar-domain

Stores java classes and unit test cases of domain layer of Domain ‘a’. Finally *.jar file is created.

- bar-domain

Class of domain layer of Domain ‘b’.

- bar-web-a

Stores application layer java classes, jsps, configuration files, unit test cases etc. Finally *.war file is created as the Web application. bar-web-a has dependency on bar-common and bar-env.

- bar-web-b

This is a Web application as one more subsystem. Its structure is same as the bar-web-a.

- bar-env

Collects only the configuration files having environment dependency. Refer to *Removing Environment Dependency* for details.

- bar-web-a-selenium

Stores test cases using [Selenium WebDriver](#) for web-a project.

- bar-web-b-selenium

Stores test cases using [Selenium WebDriver](#) for web-b project.

Todo

Additionally, describe about splitting a JSP.

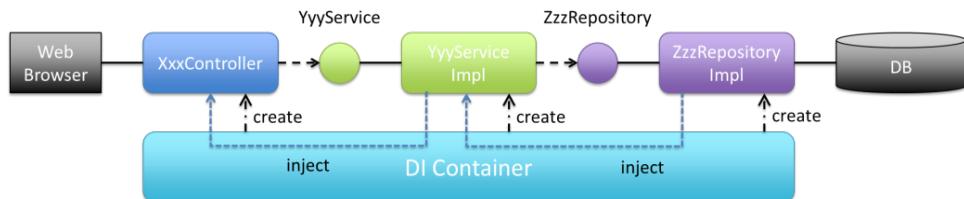
7.5 Reference Books

This guideline was prepared after referring to the following books. Refer to them as per the requirements.

Book name	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	Gihyo Digital Publishing	Japanese
Beginning Java EE 6 GlassFish 3 で始めるエンタープライズ Java	Shoiesha Co., Ltd	Japanese
Seasar2 と Hibernate で学ぶデータベースアクセス JPA 入門	Mynavi Corporation	Japanese

7.6 Spring Framework Comprehension Check

1. Fill (1)-(4) such that the Bean dependency relation is as follows. Skip import statement.



```
@Controller
public class XxxController {
    (1)
    protected (2) yyyService;

    // omitted
}
```

```
@Service
@Transactional
public class YyyServiceImpl implements YyyService {
    (1)
    protected (4) zzzRepository;

    // omitted
}
```

Note: `@Service`, `@Controller` are `org.springframework.stereotype` package annotations and `@Transactional` is the annotation of `org.springframework.transaction.annotation`.

2. Explain when to use `@Controller`, `@Service`, and `@Repository` respectively.

Note: Each of them is `org.springframework.stereotype` package annotation.

3. Explain the difference between `@Resource` and `@Inject`.

Note: `@Resource` is `javax.annotation` package annotation, and `@Inject` is `javax.inject` package annotation.

4. Explain the difference between singleton scope and prototype scope.

5. Fill (1)-(3) in the following Scope related description. However, either of “singleton” or “prototype” can be entered in (1) and (2), but same value cannot be entered for both. Skip the import statement.

```
@Component  
(3)  
public class XxxComponent {  
    // omitted  
}
```

Note: @Component is org.springframework.stereotype.Component.

Scope of bean with @Component is (1) by default. When changing scope to (2), it is better to add (3) (refer to above source code).

6. In case of following Bean definition, what type of Bean is registered in DI container?

```
<bean id="foo" class="xxx.yyy.zzz.Foo" factory-method="create">  
    <constructor-arg index="0" value="aaa" />  
    <constructor-arg index="1" value="bbb" />  
</bean>
```

7. Fill (1)-(3) of the following Bean definition such that contents in com.example.domain package becomes target of component scan.

```
<context:(1) (2)="(3)" />
```

Note: Bean definition file should include the definition of
xmlns:context="http://www.springframework.org/schema/context"

8. Fill (1)-(2) in the following Properties file related description. Skip import statement.

It is possible to read the Bean definition file in \${key} format by removing the settings in properties file, if properties file path is set in the locations attribute of <context:property-placeholder> element. Specify as shown in (1) to read any properties file under META-INF/spring directory under the class path. Moreover, @ (2) annotation should be added as shown in the following codes where the read properties value can also be injected in Bean.

```
<context:property-placeholder locations="(1)" />
```

```
emails.min.count=1  
emails.max.count=4
```

```
@Service  
@Transactional  
public class XxxServiceImpl implements XxxService {  
    @(2)("${emails.min.count}")  
    protected int emailsMinCount;  
    @(2)("${emails.max.count}")  
    protected int emailsMaxCount;
```

```
// omitted  
}
```

Note: Bean definition file should include the definitions of

xmlns:context="http://www.springframework.org/schema/context"

9. Fill (1)-(5) in the following description for AOP Advice of Spring. The contents of (1)-(5) are all different.

Note: Advice (1) should be used when interrupting a process before calling a specific method, Advice (2) should be used when interrupting a process after calling a specific method. Advice (3) should be used when interrupting a process before and after calling a specific method. Advice (4) should be used only when the process is ended normally and Advice (5) should be used when there is an exception.

10. Insert (*) of following Bean definition for performing transaction management using @Transactional annotation.

```
<tx: (*) />
```

Note: Bean definition file should include the definitions of

xmlns:tx="http://www.springframework.org/schema/tx"
