TERASOLUNA Batch Framework for Java (5.x) Development Guideline

NTT DATA Corporation.

Version 5.0.0.RELEASE, 2017-3-17

Table of Contents

Chapter 1. Introduction

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 Batch Framework for Java (5.x) 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. Introduction

1.2.1. Goal of guideline

This guideline provides best practices to develop high maintainability Batch applications using full stack framework focusing on Spring Framework, Spring Batch and MyBatis.

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

1.2.2. Target readers

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

- Basic knowledge of DI and AOP of Spring Framework
- Application development experience using Java
- Knowledge of SQL
- Have experiences on using Maven

This guideline is not for beginners.

In order to check whether one has enough basic knowledge to understand the document, refer to Spring Framework Comprehension Check If one is not able to answer 40% of the comprehension test, then it is recommended to study the following books separately.

- Spring徹底入門 (翔泳社) [日本語]
- ［改訂新版］Spring入門――Javaフレームワーク・より良い設計とアーキテクチャ [日本語]
- Pro Spring 4th Edition (Apress)

1.2.3. Structure of guideline

The most important thing is that the guideline is considered as the subset of TERASOLUNA Server Framework for Java (5.x) Development Guideline (hereafter, referred to as TERASOLUNA Server 5.x Development Guideline). By using TERASOLUNA Server 5.x Development Guideline, you can eliminate duplication in explanation and reduce the cost of learning as much as possible. Since it indicates reference to TERASOLUNA Server 5.x Development Guideline everywhere, we would like you to proceed with the development by using both guides.

TERASOLUNA Batch Framework for Java (5.x)concept

Explains the basic concept of batch processing and the basic concept of TERASOLUNA Batch Framework for Java (5.x) and the overview of Spring Batch.

Methodology of application development

Explains the knowledge and method to be kept in mind while developing an application using TERASOLUNA Batch Framework for Java (5.x).

Running a Job

Explains how to running a job as Synchronous, Asynchronous and provide job parameters.

Input/output of data
Explains how to provide Input/Output to various resources such as Database, File access etc.

Handling for abnormal condition
Explains how to handle the abnormal conditions like Input checks, Exceptions.

Job management
Explains how to manage the Job execution.

Flow control and parallel/multiple processing
Explains the processing of parallel/multiple Job execution.

1.2.4. How to read guideline

It is strongly recommended for all the developers to read the following contents for using TERASOLUNA Batch Framework for Java (5.x).

- TERASOLUNA Batch Framework for Java (5.x) concept
- Methodology of application development

The following contents are usually required, so they should be read in advance. It is better to select according to the development target.

- Start of job
- Input/output of data
- Support for abnormal system
- Job management

Refer to the following contents for the first time when proceeding with the implementation.

- Flow control and parallel/multiple processing

1.2.4.1. Notations in guideline

This section describe for the notations of this guideline.

About Windows command prompt and Unix terminal
If command syntax in Windows and Unix are different, it describe both. Otherwise, standardize the notations of Unix.

Prompt sign
Describe as `$` in Unix.

Prompt notation example

```
$ java -version
```

About defining properties and constructor of Bean definition

In this guideline, it is described by using namespace of `p` and `c`. The use of namespace helps in simplifying and clarifying the description of Bean definition.

Description wherein namespace is used

```
<bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
    <property name="lineTokenizer">
        <bean
class="org.terasoluna.batch.item.file.transform.FixedByteLengthLineTokenizer"
              c:ranges="1-6, 7-10, 11-12, 13-22, 23-32"
              c:charset="MS932"
              p:names="branchId,year,month,customerId,amount"/>
    </property>
</bean>
```

For your reference, the description not using namespace is shown.

Description not using namespace

```
<bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
    <property name="lineTokenizer">
        <bean
class="org.terasoluna.batch.item.file.transform.FixedByteLengthLineTokenizer">
            <constructor-arg index="0" value="1-6, 7-10, 11-12, 13-22, 23-32"/>
            <constructor-arg index="1" value="MS932"/>
            <property name="names" value="branchId,year,month,customerId,amount"/>
    </property>
</bean>
```

This guideline does not force the user to use namespace. We would like to consider it for simplifying the explanation.

1.2.5. Tested environments of guideline

For tested environments of contents described in this guideline, refer to " Tested Environment".

1.3. Change Log

Modified on	Modified locations	Modification details
2017-03-17	-	released `5.0.0 RELEASE` version

Chapter 2. TERASOLUNA Batch Framework for Java (5.x) concept

2.1. Batch Processing in General

2.1.1. Introduction to Batch Processing

The term of "Batch Processing" refers to the execution or process of a series of jobs in a computer program without manual intervention (non-interactive).
It is often a process of reading, processing and writing a large number of records from a database or a file.
Batch processing consists of following features and is a processing method which prioritizes process throughput than the responsiveness, as compared to online processing.

Commons of batch processing

- Process large number of data is collected and processed.
- Uninterruptible process for certainty of time is done in a fixed sequence.
- Process runs in accordance with the schedule.

Objective of batch processing is given below.

Enhanced throughput

Process throughput can be enhanced by processing the data sets collectively in a batch.
File or database does not input or output data one by one, and instead sums up data of a fixed quantity thus dramatically reducing overheads of waiting for I/O resulting in the increased efficiency. Even though waiting period for I/O of a single record is insignificant, cumulative accumulation while processing a large amount of data result in fatal delay.

Ensuring responsiveness

Processes which are not required to be processed immediately are cut for batch processing in order to ensure responsiveness of online processing.
For example, when the process results are not required immediately, the processing is done by online processing till its acceptance and batch processing is performed in the background. The processing method is generally called "delayed processing".

Response to time and events

Processes corresponding to specific period and events are naturally implemented by batch processing.
For example, aggregating business data sets per month on the next 1st weekend,
taking backup every Sunday at 2a.m in accordance with the system operation rules,
and so on.

Restriction for coordination with external system

Batch processing is also used due to restrictions of interface like files with interactions of external systems.

File sent from the external system is a summary of data collected for a certain period. Batch processing is better suited for the processes which incorporate these files, than the online processing.

It is very common to combine various techniques to achieve batch processing. Major techniques are introduced here.

Job Scheduler

A single execution unit of a batch processing is called a job. A job scheduler is a middleware to manage this job.
A batch system rarely has several jobs, and usually the number of jobs can reach hundreds or even thousands at times. Hence, an exclusive system to define the relation with the job and manage execution schedule becomes indispensable.

Shell script

One of the methods to implement a job. A process is achieved by combining the commands implemented in OS and middleware.
Although the method can be implemented easily, it is not suitable for writing complex business logic. Hence, it is primarily used in simple processes like copying a file, backup, clearing a table etc. Further, shell script performs only the pre-start settings and post-execution processing while executing a process implemented in another programming language.

Programming language

One of the methods to implement a job. Structured code can be written rather than the shell script and is advantageous for securing development productivity, maintainability and quality. Hence, it is commonly used to implement business logic that processes data of file or database which tend to be relatively complex with logic.

2.1.2. Requirements for batch processing

Requirements for batch processing in order to implement business process is as given below.

- Performance improvement
  - A certain quantity of data can be processed in a batch.
  - Jobs can be executed in parallel/in multiple.
- Recovery in case of an abnormality
  - Jobs can be reexecuted (manual/schedule).
  - At the time of reprocessing, it is possible to process only unprocessed records by skipping processed records.
- Various activation methods for running jobs
  - Synchronous execution possible.
  - Asynchronous execution possible.
    - DB polling, HTTP requests can be used as opportunities for execution.

- Various input and output interfaces
  - Database
  - File
    - Variable length like CSV or TSV
    - Fixed length
    - XML

Specific details for the above requirements are given below.

A large amount of data can be efficiently processed using certain resources (Performance improvement)

Processing time is reduced by processing the data collectively. Important part here is **"Certain resources"** part.
Processing can be done by using a CPU and memory for 100 or even 1 million records and the processing time is ideally extended slowly and linearly according to number of records.
Transaction is started and terminated for certain number of records to perform a process collectively. Resources to be used must be levelled in order to perform I/O collectively.
Still, when a large amount of data is to be handled which is yet to be processed, a system wherein hardware resources are used till the limit going a step further. Data to be processed is divided into records or groups and multiple processing is done by using multiple processes and multiple threads. Moving ahead, distributed processing using multiple machines is also implemented. When resources are used upto the limit, it becomes extremely important to reduce as much as possible.

Continue the processing as much as possible (Recovery at the time of occurrence of abnormality)

When a large amount of data is to be processed, the countermeasures when an abnormality occurs in input data or system itself must be considered.
A large amount of data takes a long time to finish processing, however if the time till recovery after occurrence of error is prolonged, it is likely to affect the system a great deal.
For example, consider a data consisting of 1 billion records to be processed. Operation schedule would be obviously affected a great deal if error is detected in 999 millionth record and the processing so far is to be performed all over again.
To control this impact, process continuity unique to batch processing becomes very important Hence a system wherein error data skipped and next data record is processed, a system to restart the process and a system which attempts auto-recovery become necessary. Further, it is important to simplify a job as much as possible and enable its easy execution later.

Can be executed flexibly according to triggers of execution (various activation methods)

A system to respond to various execution triggers is necessary when triggered by time, or by connecting online or connecting with external system. various systems are widely known such as synchronous processing wherein processing starts when job scheduler reaches scheduled time, asynchronous processing wherein the process is kept resident and batch processing is performed as per the events.

Handles various input and output interfaces (Various input output interfaces)

It is important to handle various files like CSV/XML as well as databases for linking online and external systems. Further, if a method which transparently handles respective input and output method exists, implementation becomes easier and to deal with various formats becomes more quickly.

2.1.3. Rules and precautions to be considered in batch processing

Important rules while building a batch processing system and a few considerations are shown.

- Simplify unit batch processing as much as possible and avoid complex logical structures.
- Keep process and data in physical proximity (Save data at the location where process is executed).
- Minimise the use of system resources (especially I/O) and execute operations in in-memory as much as possible.
- Further, review I/O of application (SQL etc) to avoid unnecessary physical I/O.
- Do not repeat the same process for multiple jobs.
  - For example, in case of counting and reporting process, avoid repetition of counting process during reporting process.
- Always assume the worst situation related to data consistency. Verify data to check and to maintain consistency.
- Review backups carefully. Difficulty level of backup will be high especially when system is operational seven days a week.

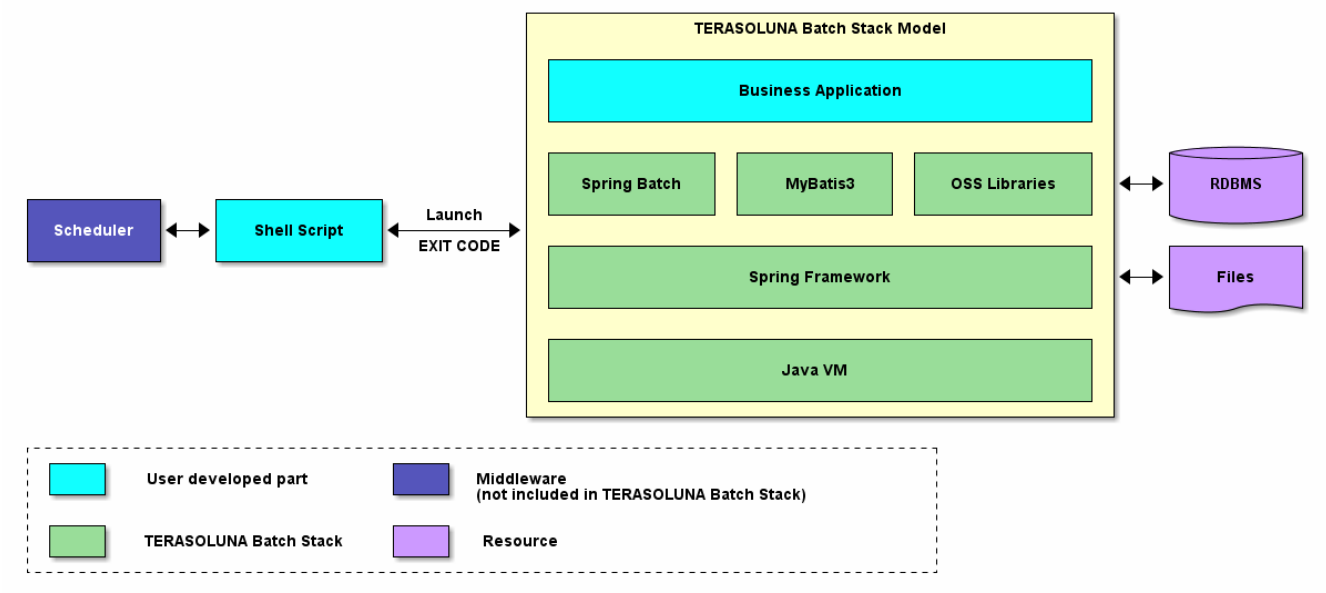
2.2. TERASOLUNA Batch Framework for Java (5.x) stack

2.2.1. Overview

TERASOLUNA Batch Framework for Java (5.x)configuration is explained and TERASOLUNA Batch Framework for Java (5.x) scope of responsibility.

2.2.2. TERASOLUNA Batch Framework for Java (5.x) stack

Software Framework used in TERASOLUNA Batch Framework for Java (5.x) is a combination of OSS focusing on Spring Framework (Spring Batch) A stack schematic diagram of TERASOLUNA Batch Framework for Java (5.x) is shown below.



TERASOLUNA Batch Framework for Java (5.x) stack - schematic diagram

Descriptions for products like job scheduler and database are excluded from this guideline.

2.2.2.1. OSS version to be used

List of OSS versions to be used in {revnumber-index} of TERASOLUNA Batch Framework for Java (5.x) is given below.

> OSS version to be used in TERASOLUNA Batch Framework for Java (5.x) as a rule conforms to definition of Spring IO platform. Note that, version of Spring IO platform in {revnumber-index} is Athens-SR2.+ For details of Spring IO platform, refer OSS version to be used of TERASOLUNA Server Framework for Java (5.x).

OSS version list

Type	GroupId	ArtifactId	Version	Spring IO platform	Remarks
Spring	org.springframework	spring-aop	4.3.5.RELEASE	*	
Spring	org.springframework	spring-beans	4.3.5.RELEASE	*	
Spring	org.springframework	spring-context	4.3.5.RELEASE	*	
Spring	org.springframework	spring-expression	4.3.5.RELEASE	*	
Spring	org.springframework	spring-core	4.3.5.RELEASE	*	
Spring	org.springframework	spring-tx	4.3.5.RELEASE	*	
Spring	org.springframework	spring-jdbc	4.3.5.RELEASE	*	
Spring Batch	org.springframework.batch	spring-batch-core	3.0.7.RELEASE	*	
Spring Batch	org.springframework.batch	spring-batch-infrastructure	3.0.7.RELEASE	*	
Spring Retry	org.springframework.retry	spring-retry	1.1.5.RELEASE	*	
Java Batch	javax.batch	javax.batch-api	1.0.1	*	
Java Batch	com.ibm.jbatch	com.ibm.jbatch-tck-spi	1.0	*	
MyBatis3	org.mybatis	mybatis	3.4.2		
MyBatis3	org.mybatis	mybatis-spring	1.3.1		
MyBatis3	org.mybatis	mybatis-typehandlers-jsr310	1.0.2		
DI	javax.inject	javax.inject	1	*	
Log output	ch.qos.logback	logback-classic	1.1.8	*	
Log output	ch.qos.logback	logback-core	1.1.8	*	*1
Log output	org.slf4j	jcl-over-slf4j	1.7.22	*	
Log output	org.slf4j	slf4j-api	1.7.22	*	
Input check	javax.validation	validation-api	1.1.0.Final	*	

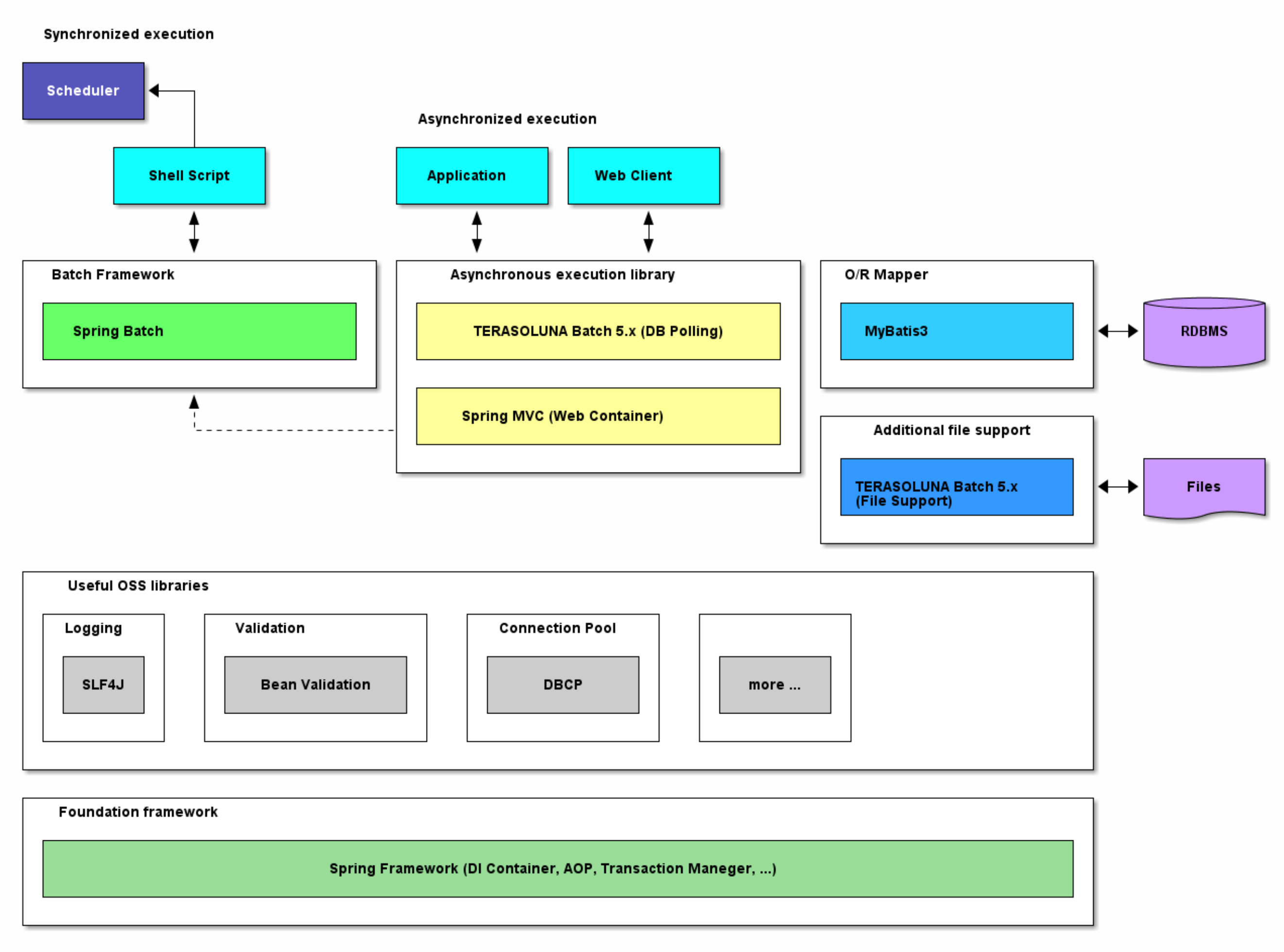
Type	GroupId	ArtifactId	Version	Spring IO platform	Remarks
Input check	org.hibernate	hibernate-validator	5.2.4.Final	*	
Input check	org.jboss.logging	jboss-logging	3.3.0.Final	*	*1
Input check	com.fasterxml	classmate	1.3.3	*	*1
Connection pool	org.apache.commons	commons-dbcp2	2.1.1	*	
Connection pool	org.apache.commons	commons-pool2	2.4.2	*	
Expression Language	org.glassfish	javax.el	3.0.0	*	
In-memory database	com.h2database	h2	1.4.193	*	
XML	com.thoughtworks.xstream	xstream	1.4.9	*	*1
XML	xmlpull	xmlpull	1.1.3.1		*1
XML	xpp	xpp3_min	1.1.4c		*1
XML	xpp	xpp3_min	1.1.4c		*1
JSON	org.codehaus.jettison	jettison	1.2	*	*1

Remarks

1. Libraries which individually depend on libraries supported by Spring IO platform

2.2.3. Structural elements of TERASOLUNA Batch Framework for Java (5.x)

Software Framework structural elements of TERASOLUNA Batch Framework for Java (5.x) are explained.



Schematic diagram of Software Framework structural elements

Overview of each element is shown below.

Foundation framework

Spring Framework is used as a framework foundation. Various functions are applied starting with DI container.

- Spring Framework 4.3

Batch framework

Spring Batch is used as a batch framework.

- Spring Batch 3.0

Asynchronous execution

Following functions are used as a method to execute asynchronous execution.

Periodic activation by using DB polling

A library offered by TERASOLUNA Batch Framework for Java (5.x) is used.

- Asynchronous execution (DB polling)

Web container activation

Link with Spring Batch using Spring MVC.

- Spring MVC 4.3

O/R Mapper

Use MyBatis, and use MyBatis-Spring as a library to coordinate with Spring Framework.

- MyBatis 3.4
- MyBatis-Spring

File access

In addition to Function offered from Spring Batch, TERASOLUNA Batch Framework for Java (5.x) is used as an auxiiliary function.

- File access

Logging

Logger uses SLF4J in API and Logback in the implementation.

- SLF4J
- Logback

Validation

Unit item check

Bean Validation is used in unit item check and Hibernate Validator is used for implementation.

- Bean Validation 1.1
- Hibernate Validator 5.2

Correlation check

Bean Validation or Spring Validation is used for correlation check.

- Spring Validation

Connection pool

DBCP is used in the connection pool.

- DBCP 2
- Commons Pool 2

2.2.3.1. A function wherein TERASOLUNA Batch Framework for Java (5.x) provides implementation

A function wherein TERASOLUNA Batch Framework for Java (5.x) provides implementation is given below.

A function list wherein TERASOLUNA Batch Framework for Java (5.x) offers implementation

Function name	Overview

Asynchronous execution (DB polling)	Asynchronous execution using DB polling is implemented.
File access	Read fixed-length file without line breaks by number of bytes.
	Break down a fixed length record in individual field by number of bytes.
	Control output of enclosed characters by variable length records.

2.3. Spring Batch Architecture

2.3.1. Overview

Spring Batch architecture acting as a base for TERASOLUNA Server Framework for Java (5.x) is explained.

2.3.1.1. What is Spring Batch

Spring Batch, as the name implies is a batch application framework. Following functions are offered based on DI container of Spring, AOP and transaction control function.

Functions to standardize process flow

Tasket model

Simple process

It is a method to freely describe a process. It is used in a simple cases like issuing SQL once, issuing a command etc and the complex cases like performing processing while accessing multiple database or files, which are difficult to standardize.

Chunk model

Efficient processing of large amount of data

A method to collectively input / process / output a fixed amount of data. Process flow of data input / processing and output is standardized and job can be implemented by implementing only a part of it.

Various activation methods

Execution is achieved by various triggers like command line execution, execution on Servlet and other triggers.

I/O of various data formats

Input and output for various data resources like file, database, message queue etc can be performed easily.

Efficient processing

Multiple execution, parallel execution, conditional branching are done based on the settings.

Job execution control

Permanence of execution, restart operation using data records as a standard can be performed.

2.3.1.2. Hello, Spring Batch！

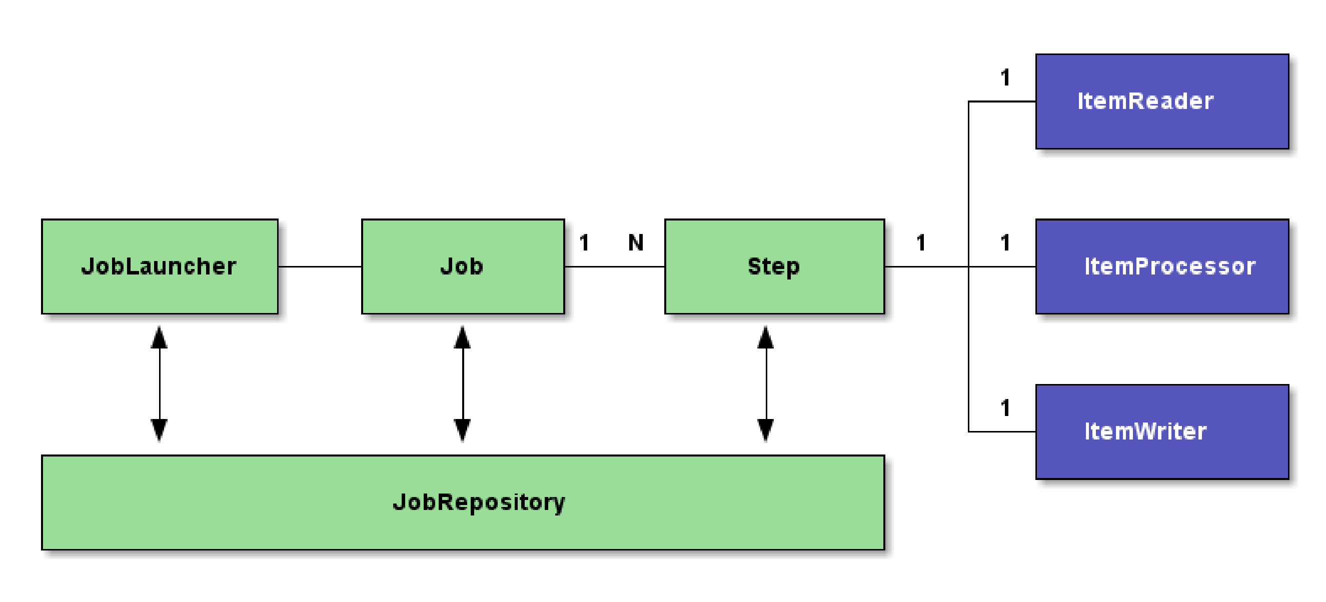
If Spring Batch is not covered in understanding of Spring Batch architecture so far, the official documentation given below should be read. We would like you to get used to Spring Batch through creating simple application.

Creating a Batch Service

2.3.1.3. Basic structure of Spring Batch

Basic structure of Spring Batch is explained.

Spring Batch defines structure of batch process. It is recommended to perform development after understanding the structure.



Primary components appearing in Spring Batch

Primary components appearing in Spring Batch

Components	Roles
Job	A single execution unit that summarises a series of processes for batch application in Spring Batch.
Step	A unit of processing which constitutes Job. 1 job can contain 1~N steps Reusing a process, parallelization, conditional branching can be performed by dividing 1 job process in multiple steps. Step is implemented by either chunk model or tasket model(will be described later).
JobLauncher	An interface for running a Job. JobLauncher can be directly used by the user, however, a batch process can be started simply by starting `CommandLineJobRunner` from java command. `CommandLineJobRunner` undertakes various processes for starting JobLauncher.

Components	Roles
ItemReader ItemProcessor ItemWriter	An interface for dividing into three processes - input / processing / output of data while implementing chunk model. Batch application consists of processing of these 3 patterns and in Spring Batch, implementation of these interfaces is utilized primarily in chunk model. User describes business logic by dividing it according to respective roles. Since ItemReader and ItemWriter responsible for data input and output are often the processes that perform conversion of database and files to Java objects and vice versa, a standard implementation is provided by Spring Batch. In general batch applications which perform input and output of data from file and database, conditions can be satisfied just by using standard implementation of Spring Batch as it is. ItemProcessor which is responsible for processing data implements input check and business logic. In Tasket model, ItemReader/ItemProcessor/ItemWriter substitutes a single Tasklet interface implementation.
JobRepository	A system to manage condition of Job and Step. The management information is persisted on the database based on the table schema specified by Spring Batch.

2.3.2. Architecture

Basic structure of Spring Batch is briefly explained in Overview.

Following points are explained on this basis.

- Overall process flow
- Running a Job
- Execution of business logic
- Metadata schema of JobRepository

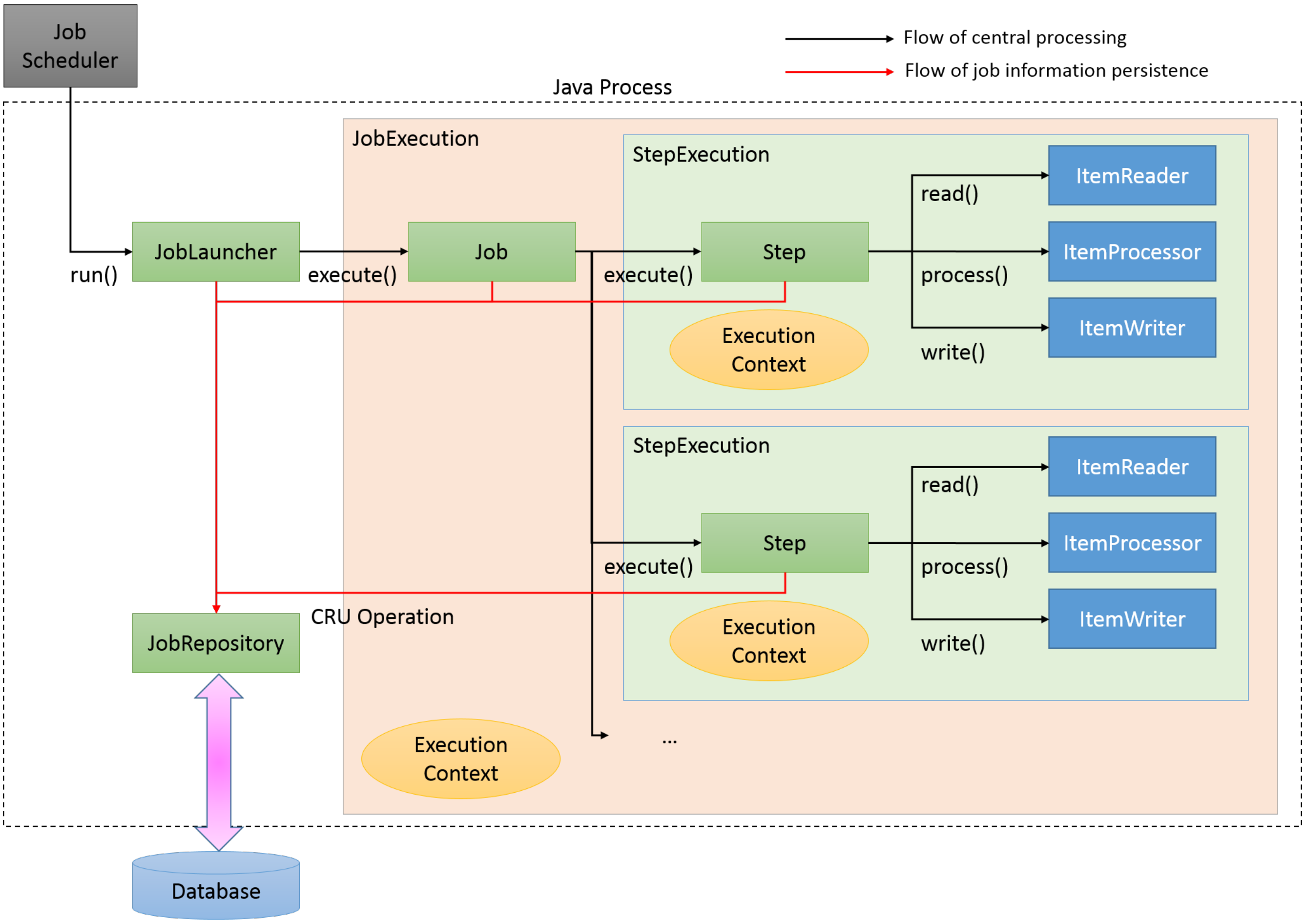
In the end, performance tuning points of batch application which use Spring Batch are explained.

- Typical performance tuning points

2.3.2.1. Overall process flow

Primary components of Spring Batch and overall process flow is explained. Further, explanation is also given about how to manage meta data of execution status of jobs.

Primary components of Spring Batch and overall process flow (chunk model) are shown in the figure below.



Primary components of Spring Batch and overall process flow

Main processing flow (black line) and the flow which persists job information (red line) are explained.

Main processing flow

1. JobLauncher is initiated from the job scheduler.
2. Job is executed from JobLauncher.
3. Step is executed from Job.
4. Step fetches input data by using ItemReader.
5. Step processes input data by using ItemProcessor.
6. Step outputs processed data by using ItemWriter.

A flow for persisting job information

1. JobLauncher registers JobInstance in Database through JobRepository.
2. JobLauncher registers that Job execution has started in Database through JobRepository.
3. JobStep updates miscellaneous information like counts of I/O records and status in Database through JobRepository.
4. JobLauncher registers that Job execution has completed in Database through JobRepository.

Components and JobRepository focusing on persistence are explained freshly again.

Components related to persistence

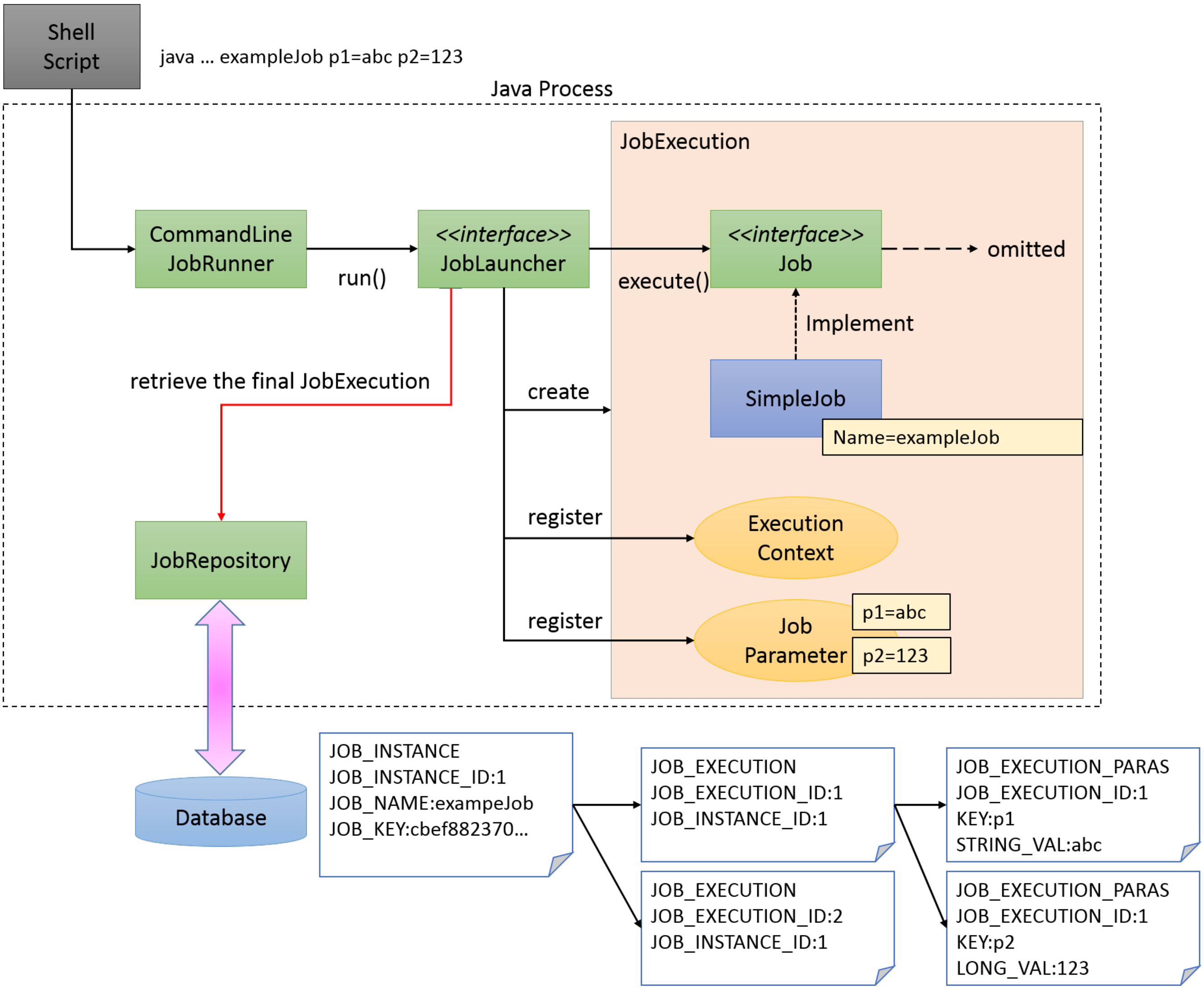
Components	Roles
JobInstance	Spring Batch indicates "logical" execution of a Job. JobInstance is identified by Job name and arguments. In other words, execution with identical Job name and argument is identified as execution of identical JobInstance and Job is executed as a continuation from previous activation. When the target Job supports re-execution and the process was suspended in between due to error in the previous execution, the job is executed from the middle of the process. On the other hand, when the target job does not support re-execution or when the target JobInstance has already been successfully processed, exception is thrown and Java process is terminated abnormally. For example, JobInstanceAlreadyCompleteException is thrown when the process has already been completed successfully.
JobExecution ExecutionContext	JobExecution indicates "physical" execution of Job. Unlike JobInstance, it is termed as another JobExecution even while re-executing identical Job. As a result, JobInstance and JobExecution shows one-to-many relationship. ExecutionContext is considered as an area for sharing metadata such as progress of a process in identical JobExecution. ExecutionContext is primarily used for enabling Spring Batch to record framework status, however, means to access ExecutionContext by the application is also provided. The object stored in the JobExecutionContext must be a class which implements `java.io.Serializable`.
StepExecution ExecutionContext	StepExecution indicates "physical" execution of Step. JobExecution and StepExecution shows one-to-many relationship. Similar to JobExecution, ExecutionContext is an area for sharing data in Step. From the viewpoint of localization of data, information which is not required to be shared by multiple steps should use ExecutionContext of target step instead of using ExecutionContext of Job. The object stored in StepExecutionContext must be a class which implements `java.io.Serializable`.
JobRepository	A function to manage and persist data for managing execution results and status of batch application like JobExecution or StepExecution is provided. In general batch applications, the process is started by starting a Java process and Java process is also terminated along with termination of process. Hence, since the data is likely to be referred across Java process, it is stored in volatile memory as well as permanent layers like database. When data is to be stored in the database, database objects like table or sequence are required for storing JobExecution or StepExecution. It is necessary to generate a database object based on schema information provided by Spring Batch.

Spring Batch heavily manages metadata in order to perform re-execution. A snapshot at the time of earlier execution must be retained and metadata and JobRepository should be used as a base in order to re-execute a batch process.

2.3.2.2. Running a Job

How to run a Job is explained.

A scenario is considered wherein a batch process is started immediately after starting Java process and Java process is terminated after completing a batch process. Figure below shows a process flow from starting a Java process till starting a batch process.



Process flow from starting a Java process till starting a batch process

Starting a Java process and starting a Job

A shell script to start Java is generally described to start a Job defined on Spring Batch, along with starting a Java process. When CommandLineJobRunner offered by Spring Batch is used, Job on Spring Batch defined by the user can be easily started.

Start command of the Job which use CommandLineJobRunner is as shown below.

Start command when a Bean is defined by using XML

```
java -cp ${CLASSPATH}
org.springframework.batch.core.launch.support.CommandLineJobRunner <jobPath> <jobName>
<JobArgumentName1>=<value1> <JobArgumentName2>=<value2> ...
```

Specifying a Job parameter

CommandLineJobRunner can pass arguments (job parameters) as well along with Job name to be started. Arguments are specified in `<Job argument name>=<Value>` format as per the example described earlier. All the arguments are stored in JobExecution after conversion to JobParameters after interpreting and checking by CommandLineJobRunner or JobLauncher. For details, refer to running parameter of Job.

Register and restore JobInstance

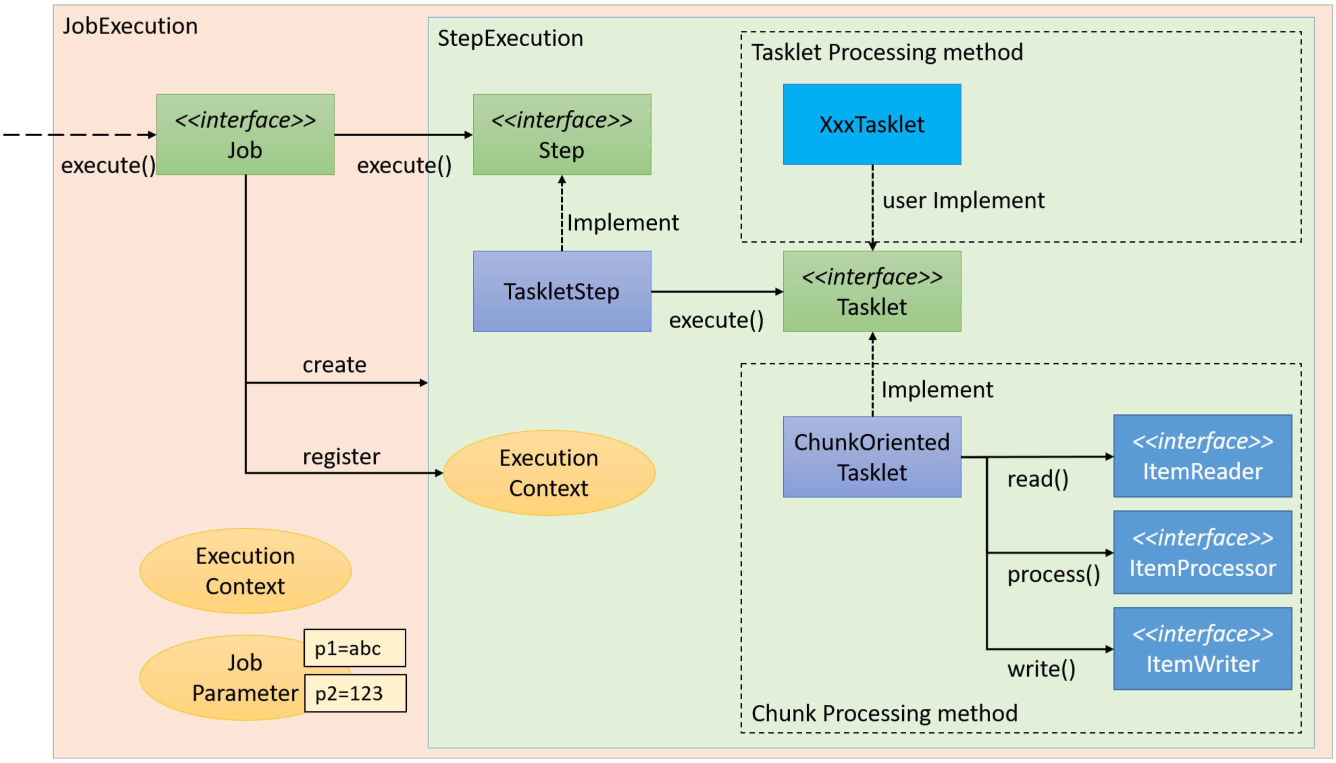
JobLauncher fetches Job name from JobRepository and JobInstance matching with the argument from the database.

- When corresponding JobInstance does not exist, JobInstance is registered as new.
- When corresponding JobInstance exists, the associated JobExecution is restored.
  - In Spring Batch, for the jobs that can be executed repeatedly like daily execution etc, a method to add arguments for making the JobInstance unique is listed. For example, adding system date or random number to arguments are listed.
    For the method recommended in this guideline, refer parameter conversion class.

2.3.2.3. Execution of business logic

Job is divided into smaller units called steps in Spring Batch. When Job is started, Job activates already registered steps and generates StepExecution. Step is a framework for dividing the process till the end and execution of business logic is delegated to Tasket called from Step.

Flow from Step to Tasklet is shown below.



Process flow from Step to Tasklet

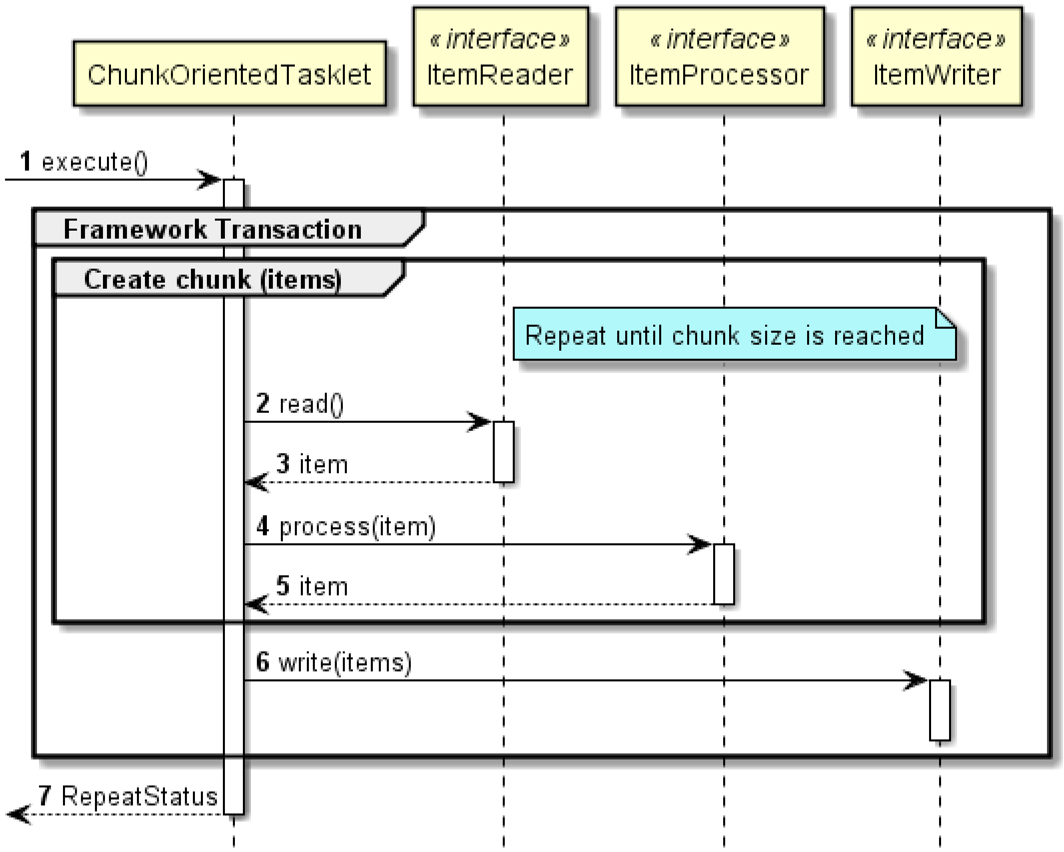
A couple of methods can be listed as the implementation methods of Tasklet - "Chunk model" and "Tasket model". Since the overview has already been explained, the structure will be now explained here.

2.3.2.3.1. Chunk model

As described above, chunk model is a method wherein the processing is performed in a certain number of units (chunks) rather than processing the data to be processed one by one unit. ChunkOrientedTasklet acts as a concrete class of Tasklet which supports the chunk processing. Maximum records of data to be included in the chunk (hereafter referred as "chunk size") can be adjusted by using setup value called commit-interval of this class. ItemReader, ItemProcessor and ItemWriter are all the interfaces based on chunk processing.

Next, explanation is given about how ChunkOrientedTasklet calls the ItemReader, ItemProcessor and ItemWriter.

A sequence diagram wherein ChunkOrientedTasklet processes one chunk is shown below.



Chunk processing by using ChunkOrientedTasklet

ChunkOrientedTasklet repeatedly executes ItemReader and ItemProcessor by the chunk size, in other words, reading and processing of data. After completing reading all the data of chunks, data writing process of ItemWriter is called only once and all the processed data in the chunks is passed. Data update processing is designed to be called once for chunks to enable easy organising like addBatch and executeBatch of JDBC.

Next, ItemReader, ItemProcessor and ItemWriter which are responsible for actual processing in chunk processing are introduced. Although it is assumed that the user handles his own implementation for each interface, it can also be covered by a generic concrete class provided by Spring Batch.

Especially, since ItemProcessor describes the business logic itself, the concrete classes are hardly provided by Spring Batch. ItemProcessor interface is implemented while describing the business logic. ItemProcessor is designed to allow types of objects used in I/O to be specified in respective generics so that typesafe programming is enabled.

An implementation example of a simple ItemProcessor is shown below.

Implementation example of ItemProcessor

```
public class MyItemProcessor implements
      ItemProcessor<MyInputObject, MyOutputObject> {  // (1)
  @Override
    public MyOutputObject process(MyInputObject item) throws Exception {  // (2)

        MyOutputObject processedObject = new MyOutputObject();  // (3)

        // Coding business logic for item of input data

    return processedObject; // (4)
  }
}
```

Sr. No.	Description
(1)	Implement ItemProcessor interface which specifies the types of objects used for input and output.
(2)	Implement `process` method. Argument item is input data.
(3)	Create output object and store business logic results processed for the input data item.
(4)	Return output object.

Various concrete classes are offered by Spring Batch for ItemReader or ItemWriter and these are used quite frequently. However, when a file of specific format is to be input or output, a concrete class which implements individual ItemReader or ItemWriter can be created and used.

For implementation of business logic while developing actual application, refer application development flow.

Representative concrete classes of ItemReader, ItemProcessor and ItemWriter offered by Spring Batch are shown in the end.

Representative concrete classes of ItemReader, ItemProcessor and ItemReader offered by Spring Batch

Interface	Concrete class name	Overview
ItemReader	FlatFileItemReader	Read flat files (non-structural files) like CSV file. Mapping rules for delimiters and objects can be customised by using Resource object as input.
	StaxEventItemReader	Read XML file. As the name implies, it is an implementation which reads a XML file based on StAX.
	JdbcPagingItemReader JdbcCursorItemReader	Execute SQL by using JDBC and read records on the database. When a large amount of data is to be processed on the database, it is necessary to avoid reading all the records on memory, and to read and discard only the data necessary for one processing. JdbcPagingItemReader is implemented by dividing SELECT SQL for each page by using JdbcTemplate and then issuing the same. On the other hand, JdbcCursorItemReader is implemented by issuing one SELECT SQL by using JDBC cursor. ⚠ Using MyBatis is considered as a base in TERASOLUNA Batch 5.x.
	MyBatisCursorItemReader MyBatisPagingItemReader	Read records on the database in coordination with MyBatis. Spring coordination library offered by MyBatis is provided by MyBatis-Spring. For the difference between Paging and Cursor, it is same as JdbcXXXItemReader except for using MyBatis for implementation.+ In addition, JpaPagingItemReader, HibernatePagingItemReader and HibernateCursor are provided which reads records on the database by coordinating with ItemReaderJPA implementation or Hibernate. ⚠ Using `MyBatisCursorItemReader` is considered as a base in TERASOLUNA Batch 5.x.
	JmsItemReader AmqpItemReader	Receive messages from JMS or AMQP and read the data contained in the same.

Interface	Concrete class name	Overview
ItemProcessor	PassThroughItemProcessor	No operation is performed. It is used when processing and modification of input data is not required.
	ValidatingItemProcessor	Performs input check. It is necessary to implement Spring Batch specific org.springframework.batch.item.validator.Validator for the implementation of input check rules, however, SpringValidator which is an adaptor of a generalorg.springframework.validation.Validator offered by Spring is provided and rules of org.springframework.validation.Validator can be used. ⚠ Use of ValidatingItemProcessor is prohibited in TERASOLUNA Batch 5.x. For details, refer Input check.
	CompositeItemProcessor	Sequentially execute multiple ItemProcessor for identical input data. It is enabled when business logic is to be executed after performing input check using ValidatingItemProcessor.
ItemWriter	FlatFileItemWriter	Write processed Java object as a flat file like CSV file. Mapping rules for file lines can be customised from delimiters and objects.
	StaxEventItemWriter	Write processed Java object as a XML file.
	JdbcBatchItemWriter	Execute SQL by using JDBC and output processed Java object to database. Internally JdbcTemplate is used.
	MyBatisBatchItemWriter	Coordinate with MyBatis and output processed Java object to the database. It is provided by Spring coordination library MyBatis-Spring offered by MyBatis. ⚠ JPA implementation or JpaItemWriter and HibernateItemWriter for Hibernate is not used in TERASOLUNA Batch 5.x.
	JmsItemWriter AmqpItemWriter	Send a message of a processed Java object with JMS or AMQP.

PassThroughItemProcessor omitted

When a job is defined in XML, ItemProcessor setting can be omitted. When it is omitted, input data is passed to ItemWriter without performing any operation similar to PassThroughItemProcessor.

ItemProcessor omitted

```
<batch:job id="exampleJob">
    <batch:step id="exampleStep">
        <batch:tasklet>
            <batch:chunk reader="reader" writer="writer" commit-
interval="10" />
        </batch:tasklet>
    </batch:step>
</batch:job>
```

2.3.2.3.2. Tasket model

Chunk model is a framework suitable for batch applications that read multiple input data one by one and perform a series of processing. However, a process which does not fit with the type of chunk processing is also implemented. For example, when system command is to be executed and when only one record of table for control is to be updated.

In such a case, merits of efficiency obtained by chunk processing are very less and demerits owing to difficult design and implementation are significant. Hence, it is rational to use tasket model.

It is necessary for the user to implement Tasket interface provided by Spring Batch while using a Tasket model. Further, following concrete class is provided in Spring Batch, subsequent description is not given in TERASOLUNA Batch 5.x.

Concrete class of Tasket offered by Spring Batch

Class name	Overview
SystemCommandTasklet	Tasket to execute system commands asynchronously. Command to be specified in the command property is specified. Since the system command is executed by a thread different from the thread for calling, it is possible to set a timeout and cancel the execution thread of the system command during the process.
MethodInvokingTaskletAdapter	Tasket for executing specific methods of POJO class. Specify Bean of target class in targetObject property and name of the method to be executed in targetMethod property. POJO class can return batch process termination status as a return value of the method, however then the ExitStatus described later must be set as a return value. When a value of another type is returned, the status is considered as "normal termination (ExistStatus: COMPLETED) regardless of the return value.

2.3.2.4. Metadata schema of JobRepository

Metadata schema of JobRepository is explained.

Note that, overall picture is explained including the contents explained in Spring Batch reference Appendix B. Meta-Data Schema

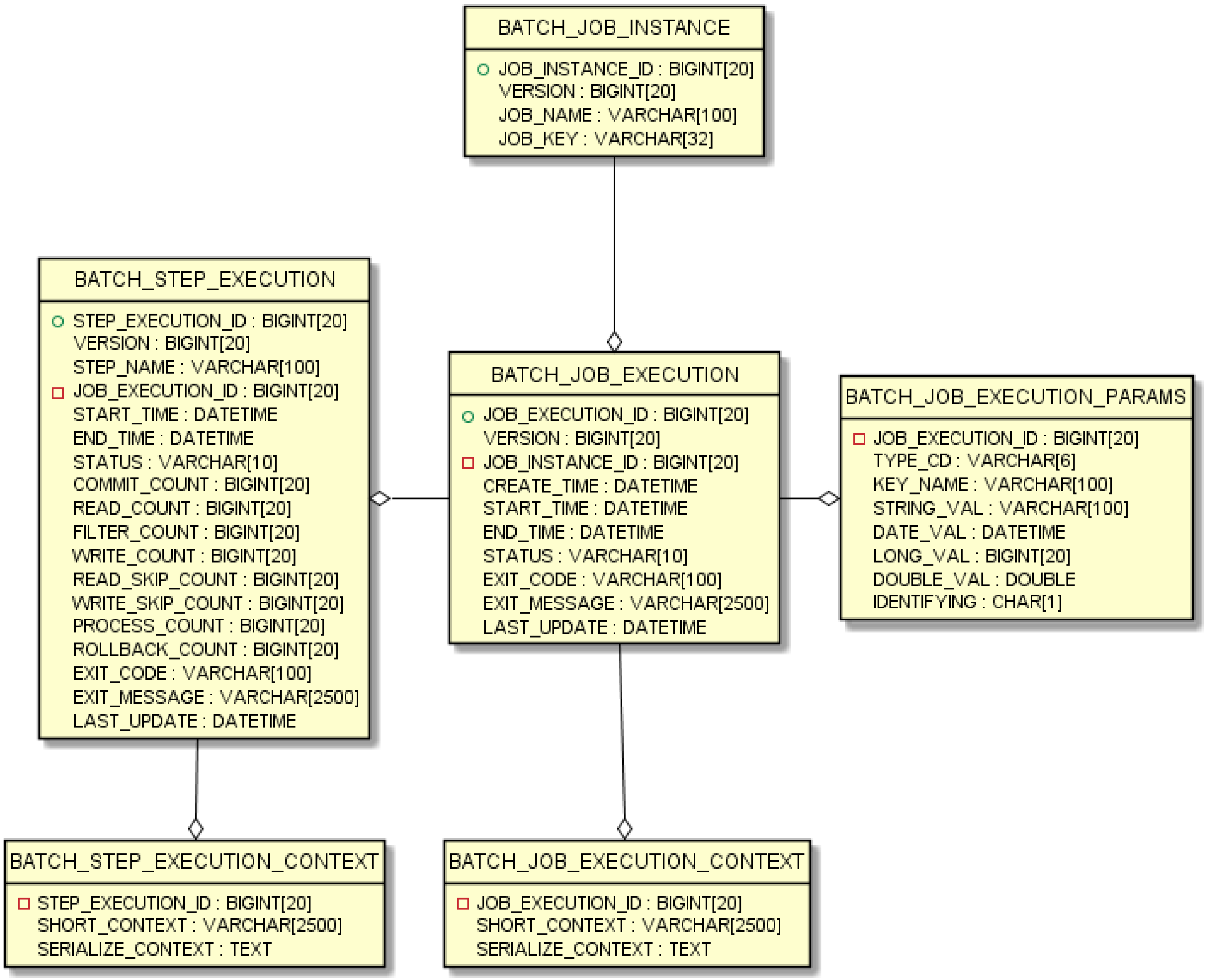
Spring Batch metadata table corresponds to a domain object (Entity object) which are represented by Java.

Correspondence list

Table	Entity object	Overview
BATCH_JOB_INSTANCE	JobInstance	Retains the string which serialises job name and job parameter.
BATCH_JOB_EXECUTION	JobExecution	Retains job status and execution results.
BATCH_JOB_EXECUTION_PARAMS	JobExecutionParams	Retains job parameters assigned at the startup.
BATCH_JOB_EXECUTION_CONTEXT	JobExecutionContext	Retains the context inside the job.
BATCH_STEP_EXECUTION	StepExecution	Retains status and execution results of step, number of commits and rollbacks.
BATCH_STEP_EXECUTION_CONTEXT	StepExecutionContext	Retains context inside the step.

JobRepository is responsible for accurately storing the contents stored in each Java object, in the table.

6 ERD models of all the tables and interrelations are shown below.



ER diagram

2.3.2.4.1. Version

Majority of database tables contain version columns. This column is important since Spring Batch adopts an optimistic locking strategy to handle updates to database. This record signifies that it is updated when the value of the version is incremented. When JobRepository updates the value and the version number is changed, an OptimisticLockingFailureException which indicates an occurrence of simultaneous access error is thrown. Other batch jobs may be running on a different machines, however, all the jobs use the same database, hence this check is required.

2.3.2.4.2. ID (Sequence) definition

BATCH_JOB_INSTANCE, BATCH_JOB_EXECUTION and BATCH_STEP_EXECUTION all contain column ending with _ID. These fields act as a primary key for respective tables. However, these keys are not generated in the database but are rather generated in a separate sequence. After inserting one of the domain objects in the database, the keys which assign the domain objects should be set in the actual objects so that they can be uniquely identified in Java.
Sequences may not be supported depending on the database. In this case, a table is used instead of each sequence.

2.3.2.4.3. Table definition

Explanation is given for each table item.

BATCH_JOB_INSTANCE

BATCH_JOB_INSTANCE table retains all the information related to JobInstance and is at top level of the overall hierarchy.

BATCH_JOB_INSTANCE definition

Column name	Description
JOB_INSTANCE_ID	A primary key which is a unique ID identifying an instance.
VERSION	Refer Version.
JOB_NAME	Job name. A non-null value since it is necessary for identifying an instance.
JOB_KEY	JobParameters which are serialised for uniquely identifying same job as a different instance. JobInstances with the same job name must contain different JobParameters (in other words, varying JOB_KEY values).

BATCH_JOB_EXECUTION

BATCH_JOB_EXECUTION table retains all the information related to JobExecution object. When a job is executed, new rows are always registered in the table with new JobExecution.

BATCH_JOB_EXECUTION definition

Column name	Description
JOB_EXECUTION_ID	Primary key that uniquely identifies this job execution.
VERSION	Refer Version.
JOB_INSTANCE_ID	Foreign key from BATCH_JOB_INSTANCE table which shows an instance wherein the job execution belongs. Multiple executions are likely to exist for each instance.
CREATE_TIME	Time when the job execution was created.
START_TIME	Time when the job execution was started.
END_TIME	Indicates the time when the job execution was terminated regardless of whether it was successful or failed. Even though the job is not running currently, the column value is empty which indicates there are several error types and the framework was unable to perform last save operation.
STATUS	A character string which indicates job execution status. It is a character string output by BatchStatus enumeration object.

Column name	Description
EXIT_CODE	A character string which indicates an exit code of job execution. When it is activated by CommandLineJobRunner, it can be converted to a numeric value.
EXIT_MESSAGE	A character string which indicates detailed explanation of job end status. When a failure occurs, a character string that includes as many as stack traces as possible is likely.
LAST_UPDATED	Time when job execution of the record was last updated.

BATCH_JOB_EXECUTION_PARAMS

BATCH_JOB_EXECUTION_PARAMS table retains all the information related to JobParameters object. It contains a pair of 0 or more keys passed to the job and the value and records the parameters by which the job was executed.

BATCH_JOB_EXECUTION_PARAMS definition

Column name	Description
JOB_EXECUTION_ID	Foreign key from BATCH_JOB_EXECUTION table which executes this job wherein the job parameter belongs.
TYPE_CD	A character string which indicates that the data type is string, date, long or double.
KEY_NAME	Parameter key.
STRING_VAL	Parameter value when data type is string.
DATE_VAL	Parameter value when data type is date.
LONG_VAL	Parameter value when data type is an integer.
DOUBLE_VAL	Parameter value when data type is a real number.
IDENTIFYING	A flag which indicates that the parameter is a value to identify that the job instance is unique.

Constraints of job parameter

- Since it is stored in BATCH_JOB_EXECUTION_PARAMS, the values that can be incorporated in the parameter have certain size limits.
- When multibyte characters are used, consider adjusting the size of STRING_VAL depending on the encoding used.

BATCH_JOB_EXECUTION_CONTEXT

BATCH_JOB_EXECUTION_CONTEXT table retains all the information related to ExecutionContext of Job. It contains all the job level data required for execution of specific jobs. The data indicates the status that must be fetched when the process is to be executed again after a job failure and enables the failed job to start from the point where processing has stopped.

BATCH_JOB_EXECUTION_CONTEXT definition

Column name	Description
JOB_EXECUTION_ID	A foreign key from BATCH_JOB_EXECUTION table which indicates job execution wherein ExecutionContext of Job belongs.
SHORT_CONTEXT	A string representation of SERIALIZED_CONTEXT.
SERIALIZED_CONTEXT	Overall serialised context.

BATCH_STEP_EXECUTION

BATCH_STEP_EXECUTION table retains all the information related to StepExecution object. This table very similar to BATCH_JOB_EXECUTION table in many ways. When each JobExecution is created, at least one entry exists for each Step.

BATCH_STEP_EXECUTION definition

Column name	Description
STEP_EXECUTION_ID	Primary key that uniquely identifies the step execution.
VERSION	Refer Version.
STEP_NAME	Step name.
JOB_EXECUTION_ID	Foreign key from BATCH_JOB_EXECUTION table which indicates JobExecution wherein StepExecution belongs
START_TIME	Time when step execution was started.
END_TIME	Indicates time when step execution ends regardless of whether it is successful or failed. Even though the job is not running currently, the column value is empty which indicates there are several error types and the framework was unable to perform last save operation.
STATUS	A character string that represents status of step execution. It is a string which outputs BatchStatus enumeration object.
COMMIT_COUNT	Number of times a transaction is committed.
READ_COUNT	Data records read by ItemReader.
FILTER_COUNT	Data records filtered by ItemProcessor.
WRITE_COUNT	Data records written by ItemWriter.
READ_SKIP_COUNT	Data records skipped by ItemReader.
WRITE_SKIP_COUNT	Data records skipped by ItemWriter.
PROCESS_SKIP_COUNT	Data records skipped by ItemProcessor.
ROLLBACK_COUNT	Number of times a transaction is rolled back.
EXIT_CODE	A character string which indicates exit code for step execution. When it is activated by using CommandLineJobRunner, it can be changed to a numeric value.

Column name	Description
EXIT_MESSAGE	A character string which indicates detailed explanation of the end state of the step. When a failure occurs, a character string that includes as many as stack traces as possible is likely.
LAST_UPDATED	Time when the step execution of the record was last updated.

BATCH_STEP_EXECUTION_CONTEXT

BATCH_STEP_EXECUTION_CONTEXT table retains all the information related to ExecutionContext of Step. It contains all the step level data required for execution of specific steps. The data indicates the status that must be fetched when the process is to be executed again after a job failure and enables the failed job to start from the point where processing has stopped.

BATCH_STEP_EXECUTION_CONTEXT definition

Column name	Description
STEP_EXECUTION_ID	Foreign key from BATCH_STEP_EXECUTION table which indicates job execution wherein ExecutionContext of Step belongs.
SHORT_CONTEXT	String representation of SERIALIZED_CONTEXT.
SERIALIZED_CONTEXT	Overall serialized context.

2.3.2.4.4. DDL script

JAR file of Spring Batch Core contains a sample script which creates a relational table corresponding to several database platforms. These scripts can be used as it is or additional index or constraints can be changed as required.
The script is included in the package of org.springframework.batch.core and the file name is configured by `schema-*.sql`. "*" is the short name for Target Database Platform..

2.3.2.5. Typical performance tuning points

Typical performance tuning points in Spring Batch are explained.

Adjustment of chunk size

Chunk size is increased to reduce overhead occurring due to resource output.
However, if chunk size is too large, it increases load on the resources resulting in deterioration in the performance. Hence, chunk size must be adjusted to a moderate value.

Adjustment of fetch size

Fetch size (buffer size) for the resource is increased to reduce overhead occurring due to input from resources.

Reading of a file efficiently

When BeanWrapperFieldSetMapper is used, a record can be mapped to the Bean only by sequentially specifying Bean class and property name. However, it takes time to perform complex operations internally. Processing time can be reduced by using dedicated FieldSetMapper interface implementation which performs mapping.

For file I/O details, refer File access.

Parallel processing, Multiple processing

Spring Batch supports parallel processing of Step execution and multiple processing by using data distribution. Parallel processing or multiple processing can be performed and the performance can be improved by running the processes in parallel. However, if number of parallel processes and multiple processes is too large, load on the resources increases resulting in deterioration of performance. Hence, size must be adjusted to a moderate value.
For details of parallel and multiple processing, refer "parallel processing and multiple processing.

Reviewing distributed processing

Spring Batch also supports distributed processing across multiple machines. Guidelines are same as parallel and multiple processing.
Distributed processing will not be explained in this guideline since the basic design and operational design are complex.

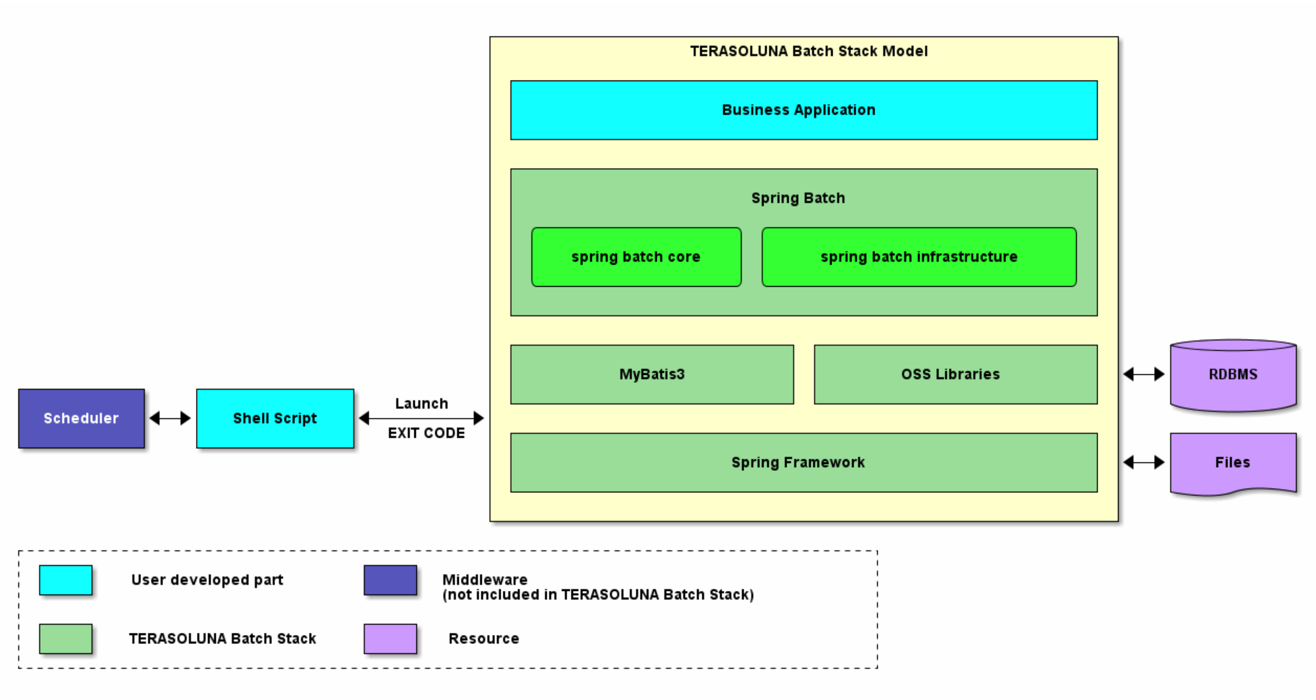
2.4. Architecture of TERASOLUNA Batch Framework for Java (5.x)

2.4.1. Overview

Overall architecture of TERASOLUNA Batch Framework for Java (5.x) is explained.

In TERASOLUNA Batch Framework for Java (5.x), as described in General batch processing system, it is implemented by using OSS combination focused on Spring Batch.

A configuration schematic diagram of TERASOLUNA Batch Framework for Java (5.x) including hierarchy architecture of Spring Batch is shown below.



Configuration schematic diagram of TERASOLUNA Batch Framework for Java (5.x)

Description of hierarchy architecture of Spring Batch

Business Application

All job definitions and business logic written by developers.

spring batch core

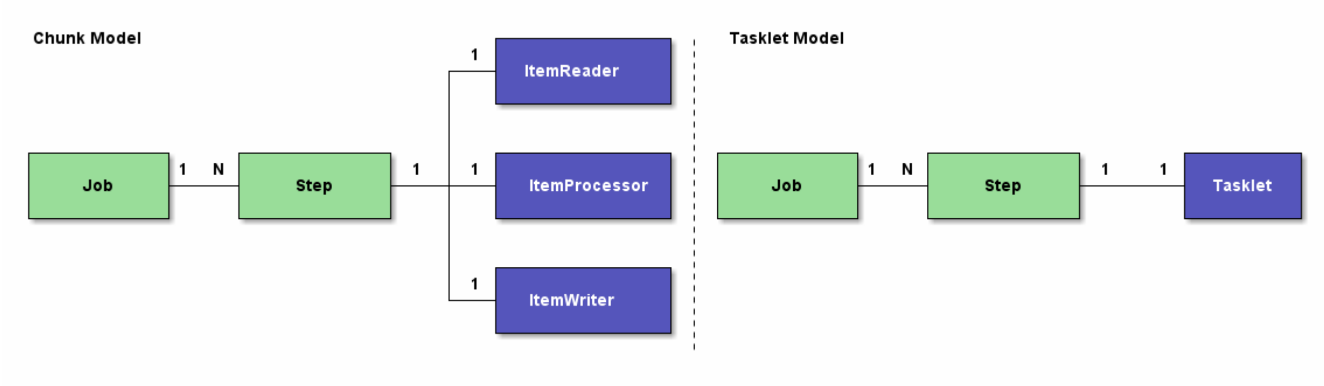
A core runtime class required to start and control batch jobs offered by Spring Batch.

spring batch infrastructure

Implementation of general ItemReader/ItemProcessor/ItemWriter offered by Spring Batch which are used by developers and core framework itself.

2.4.2. Structural elements of job

A configuration schematic diagram of jobs is shown below in order to explain structural elements of the job.



Configuration schematic diagram of job

This section also talks about guidelines which should be finely configured for job and step.

2.4.2.1. Job

A job is an entity that encapsulates entire batch process and is a container for storing steps.
A job can consist of one or more steps.

A job is defined in the Bean definition file by using XML. Multiple jobs can be defined in the job definition file, however, managing jobs tend to become complex.

Hence, TERASOLUNA Batch Framework for Java (5.x) uses following guidelines.

1 job = 1 job definition file

2.4.2.2. Step

Step defines information required for controlling a batch process. A chunk model and a tasket model can be defined in the step.

Chunk model

- It is configured by ItemReader, ItemProcessor and ItemWriter.

Tasket model

- It is configured only by Tasklet.

As given in Rules and precautions to be considered in batch processing, it is necessary to simplify as much as possible and avoid complex logical structures in a single batch process.

Hence, TERASOLUNA Batch Framework for Java (5.x) uses following guidelines.

1 step = 1 batch process = 1 business logic

Distribution of business logic in chunk model

If a single business logic is complex and large-scale, the business logic is divided into units. As clear from the schematic diagram, since only one ItemProcessor can be set in 1 step, it looks like the division of business logic is not possible. However, since CompositeItemProcssor which is an ItemProcessor consisting of multiple ItemProcessors exist, the business logic can be divided and executed by using this implementation.

2.4.3. How to implement Step

2.4.3.1. Chunk model

Definition of chunk model and purpose of use are explained.

Definition

ItemReader, ItemProcessor and ItemWriter implementation and number of chunks are set in ChunkOrientedTasklet. Respective roles are explained.

- ChunkOrientedTasklet…Call ItemReader/ItemProcessor and create a chunk. Pass created chunk to ItemWriter.
- ItemReader…Read input data.
- ItemProcessor…Process read data.
- ItemWriter…Output processed data in chunk units.

For overview of chunk model, refer Chunk model.

How to set a job in chunk model

```
<batch:job id="exampleJob">
    <batch:step id="exampleStep">
        <batch:tasklet>
            <batch:chunk reader="reader"
                         processor="processor"
                         writer="writer"
                         commit-interval="100" />
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Purpose of use

Since it handles a certain amount of data collectively, it is used while handling a large amount of data.

2.4.3.2. Tasket model

Definition of tasket model and purpose of use are explained.

Definition

Only Tasklet implementation is set.
For overview of Tasket model, refer Tasket model.

How to set a job in Tasket model

```
<batch:job id="exampleJob">
    <batch:step id="exampleStep">
        <batch:tasklet ref="myTasklet">
    </batch:step>
</batch:job>
```

Purpose of use

It can be used for executing a process which is not associated with I/O like execution of system commands etc.
Further, it can also be used while committing the data in batches.

2.4.3.3. Function difference between chunk model and Tasket model

Explanation is given for the function difference between chunk model and Tasket model. Here, only outline is given. Refer section for each function for details.

List of function differences

Function	Chunk model	Tasket model
Structural elements	Configured by ItemReader/ItemProcessor/ItemWriter /ChunkOrientedTasklet.	Configured only by Takslet.
Transaction	A transaction is generated in a chunk unit.	Processed in 1 transaction.
Recommended reprocessing method	Re-run and re-start can be used.	As a rule, only re-run is used.
Exception handling	Handling process becomes easier by using a listener. Individual implementation is also possible.	Individual implementation is required.

2.4.4. Running a job method

Running a job method is explained. This contains following.

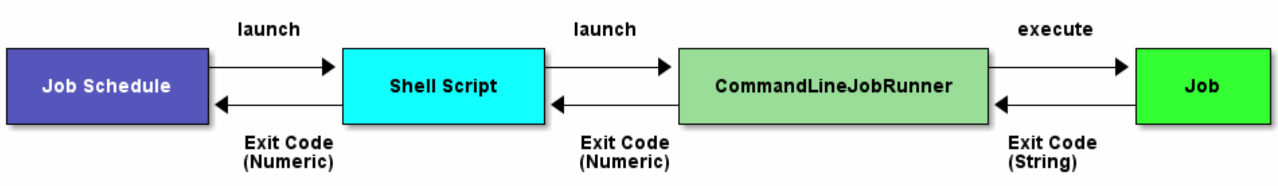
- Synchronous execution method
- Asynchronous execution method

Respective methods are explained.

2.4.4.1. Synchronous execution method

Synchronous execution method is an execution method wherein the control is not given back to the boot source from job start to job completion.

A schematic diagram which starts a job from job scheduler is shown.



Schematic diagram for synchronous execution

1. Start a shell script to run a job from job scheduler.
   Job scheduler waits until the exit code (numeric value) is returned.
2. Start `CommandLineJonRunner` to run a job from shell script.
   Shell script waits until `CommandLineJonRunner` returns an exit code (numeric value).
3. `CommandLineJonRunner` runs a job. Job returns an exit code (string) to `CommandLineJonRunner` after processing is completed.
   `CommandLineJonRunner` converts exit code (string) returned from the job to exit code (numeric value) and returns it to the shell script.

2.4.4.2. Asynchronous execution method

Asynchronous execution method is an execution method wherein the control is given back to boot source immediately after running a job, by executing a job on a different execution base than boot source (a separate thread etc). In this method, it is necessary to fetch job execution results by a means different from that of running a job.

Following 2 methods are explained in TERASOLUNA Batch Framework for Java (5.x).

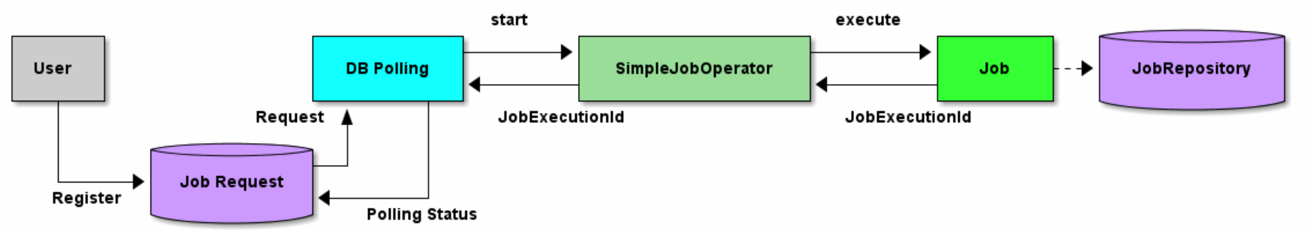
- Asynchronous execution method (DB polling)
- Asynchronous execution method (Web container)

> *Other asynchronous execution methods*
>
> Asynchronous execution can also be performed by using messages like MQ, however since the job execution points are identical, description will be omitted in {batch5_guide}.

2.4.4.2.1. Asynchronous execution method (DB polling)

Asynchronous execution (DB polling) is a method wherein a job execution request is registered in the database, polling of the request is done and job is executed.

TERASOLUNA Batch Framework for Java (5.x) supports DB polling function. The schematic diagram of start by DB polling offered is shown.



DB polling schematic diagram

1. User registers a job request to the database.
2. DB polling function periodically monitors the registration of the job request and executes the corresponding job when the registration is detected.
   - Run the job from SimpleJobOperator and receive `JobExecutionId` after completion of the job.
   - JobExecutionId is an ID which uniquely identifies job execution and execution results are browsed from JobRepository by using this ID.
   - Job execution results are registered in JobRepository by using Spring Batch system.
   - DB polling is itself executed asynchronously.
3. DB polling function updates JobExecutionId returned from SimpleJobOperator and the job request that started the status.
4. Job process progress and results are referred separately by using JobExecutionId.

2.4.4.2.2. Asynchronous execution method (Web container)

Asynchronous execution (Web container) is a method wherein a job is executed asynchronously using the request sent to web application on the web container as a trigger.* A Web application can return a response immediately after starting without waiting for the job to end.

Web container schematic diagram

1. Send a request from a client to Web application.
2. Web application asynchronously executes the job requested from a request.
   - Receive `` `JobExecutionId `` immediately after starting a job from SimpleJobOperator.
   - Job execution results are registered in JobRepository by using Spring Batch system.
3. Web application returns a response to the client without waiting for the job to end.
4. Job process progress and results are browsed separately by using JobExecutionId.

Further, it can also be linked with Web application configured by TERASOLUNA Server Framework for Java (5.x).

2.4.5. Points to consider while using

Points to consider while using TERASOLUNA Batch Framework for Java (5.x) are shown.

Running a job method

Synchronous execution method

It is used when job is run as per schedule and batch processing is carried out by combining multiple jobs.

Asynchronous execution method (DB polling)

It is used in delayed processing, continuous execution of jobs with a short processing time, aggregation of large quantity of jobs.

Asynchronous execution method (Web container)

Similar to DB polling, however it is used when an immediate action is required for the startup.

Implementation method

Chunk model

It is used when a large quantity of data is to be processed efficiently.

Tasket model

It is used for simple processing, processing that is difficult to standardize and for the processes wherein data is to be processed collectively.

Chapter 3. Methodology of application development

3.1. Development of batch application

The development of batch application is explained in the following flow.

- What is blank project
- Creation of project
- Project structure
- Flow of development
- Build of application

3.1.1. What is blank project

Blank project is the template of development project wherein various settings are made in advance such as Spring Batch, MyBatis3 and is the start point of application development.
In this guideline, a blank project with a single project structure is provided.
Refer to Project structure for the explanation of structure.

> *Difference from TERASOLUNA Server 5.x*
>
> Multi-project structure is recommended forTERASOLUNA Server 5.x. The reason is mainly to enjoy the following merits.
>
> - Makes the environmental differences easier to absorb
> - Makes separation of business logic and presentation easier
>
> However, in this guideline, a single project structure is provided unlike TERASOLUNA Server 5.x.
>
> This point should be considered for batch application also, however, by providing single project structure, accessing the resources related to one job is given priority.
> In case of batch application, one of the reason is that there are many cases when environment differences can be switched by property file or environment variables.

3.1.2. Creation of project

How to create a project using `archetype:generate` of `Maven Archetype Plugin` is explained.

Regarding prerequisites of creating environment

Prerequisites are explained below.

- Java SE Development Kit 8
- Apache Maven 3.x
  - Internet should be connected
  - When connecting to the Internet via proxy, Maven proxy setting should be done
- IDE
  - Spring Tool Suite / Eclipse etc.

Execute the following commands in the directory where project is created.

Command prompt(Windows)

```
C:\xxx> mvn archetype:generate^
  -DarchetypeGroupId=org.terasoluna.batch^
  -DarchetypeArtifactId=terasoluna-batch-archetype^
  -DarchetypeVersion=5.0.0.RELEASE
```

Bash(Unix, Linux, ...)

```
$ mvn archetype:generate \
  -DarchetypeGroupId=org.terasoluna.batch \
  -DarchetypeArtifactId=terasoluna-batch-archetype \
  -DarchetypeVersion=5.0.0.RELEASE
```

Next, set the following to Interactive mode in accordance with the status of the user.

- groupId
- artifactId
- version
- package

An example of setting and executing the value is shown below.

Explanation of each element of blank project

Item name	Setting example
groupId	com.example.batch
artifactId	batch
version	1.0.0-SNAPSHOT

Item name	Setting example
package	com.example.batch

Execution example

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.4:generate (default-cli) > generate-sources @
standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.4:generate (default-cli) < generate-sources @
standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.4:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode

(.. omitted)

Define value for property 'groupId': : com.example.batch
Define value for property 'artifactId': : batch
Define value for property 'version':  1.0-SNAPSHOT: : 1.0.0-SNAPSHOT
Define value for property 'package':  com.example.batch: :
Confirm properties configuration:
groupId: com.example.batch
artifactId: batch
version: 1.0.0-SNAPSHOT
package: com.example.batch
 Y: : y
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: terasoluna-
batch-archetype:5.0.0-SNAPSHOT
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.example.batch
[INFO] Parameter: artifactId, Value: batch
[INFO] Parameter: version, Value: 1.0.0-SNAPSHOT
[INFO] Parameter: package, Value: com.example.batch
[INFO] Parameter: packageInPathFormat, Value: com/example/batch
[INFO] Parameter: package, Value: com.example.batch
[INFO] Parameter: version, Value: 1.0.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.example.batch
[INFO] Parameter: artifactId, Value: batch
[INFO] project created from Archetype in dir: C:\workspaces\zzz\batch
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 02:56 min
[INFO] Finished at: 2017-02-07T17:09:52+09:00
[INFO] Final Memory: 16M/240M
[INFO] ------------------------------------------------------------------------
```

The creation of project is completed by the above execution.

It can be confirmed whether the project was created properly by the following points.

Confirm that the project was created properly(Bash)

```
$ mvn clean dependency:copy-dependencies -DoutputDirectory=lib package
$ java -cp 'lib/*:target/*'
org.springframework.batch.core.launch.support.CommandLineJobRunner \
    META-INF/jobs/job01/job01.xml job01
```

It is created properly if the following output is obtained.

Output example

```
$ mvn clean dependency:copy-dependencies -DoutputDirectory=lib package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building TERASOLUNA Batch Framework for Java (5.x) Blank Project 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]

(.. omitted)

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.618 s
[INFO] Finished at: 2017-02-07T17:32:27+09:00
[INFO] Final Memory: 26M/250M
[INFO] ------------------------------------------------------------------------

$ java -cp 'lib/*;target/*'
org.springframework.batch.core.launch.support.CommandLineJobRunner META-
INF/jobs/job01/job01.xml job01
[2017/02/07 17:35:26] [main] [o.s.c.s.ClassPathXmlApplicationContext] [INFO ]
Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@62043840: startup
date [Tue Feb 07 17:35:26 JST 2017]; root of context hierarchy
(.. ommited)
[2017/02/07 17:35:27] [main] [o.s.b.c.l.s.SimpleJobLauncher] [INFO ] Job: [FlowJob:
[name=job01]] launched with the following parameters: [{jsr_batch_run_id=1}]
[2017/02/07 17:35:27] [main] [o.s.b.c.j.SimpleStepHandler] [INFO ] Executing step:
[job01.step01]
[2017/02/07 17:35:27] [main] [o.s.b.c.l.s.SimpleJobLauncher] [INFO ] Job: [FlowJob:
[name=job01]] completed with the following parameters: [{jsr_batch_run_id=1}] and the
following status: [COMPLETED]
[2017/02/07 17:35:27] [main] [o.s.c.s.ClassPathXmlApplicationContext] [INFO ] Closing
org.springframework.context.support.ClassPathXmlApplicationContext@62043840: startup
date [Tue Feb 07 17:35:26 JST 2017]; root of context hierarchy
```

3.1.3. Project structure

Project structure that was created above, is explained. Project structure should be made by considering the following points.

- Implement the job independent of startup method
- Save the efforts of performing various settings such as Spring Batch, MyBatis
- Make the environment dependent switching easy

The structure is shown and each element is explained below.

(It is explained based on the output at the time of executing the above `mvn archetype:generate` to easily understand.)

```
<application-root>
└ src
  ├ main
  │ ├ java
  │ │ └ com // (1)
  │ │   └ example
  │ │     └ batch
  │ │       └ job01 // (2)
  │ │         ├ Employee.java
  │ │         ├ EmployeeProcessor.java
  │ │         └ EmployeeRepository.java
  │ └ resources
  │   ├ batch-application.properties // (3)
  │   ├ logback.xml // (4)
  │   ├ ValidationMessages.properties // (5)
  │   ├ com
  │   │ └ example
  │   │   └ batch
  │   │     └ job01
  │   │       └ EmployeeRepository.xml // (6)
  │   ├ i18n
  │   │ └ application-messages.properties // (7)
  │   └ META-INF
  │     ├ jobs // (8)
  │     │ └ job01
  │     │   └ job01.xml
  │     └ spring // (9)
  │       ├ async-batch-daemon.xml // (10)
  │       ├ job-base-context.xml // (11)
  │       └ launch-context.xml // (12)
  └ test
    ├ java
    └ resources
```

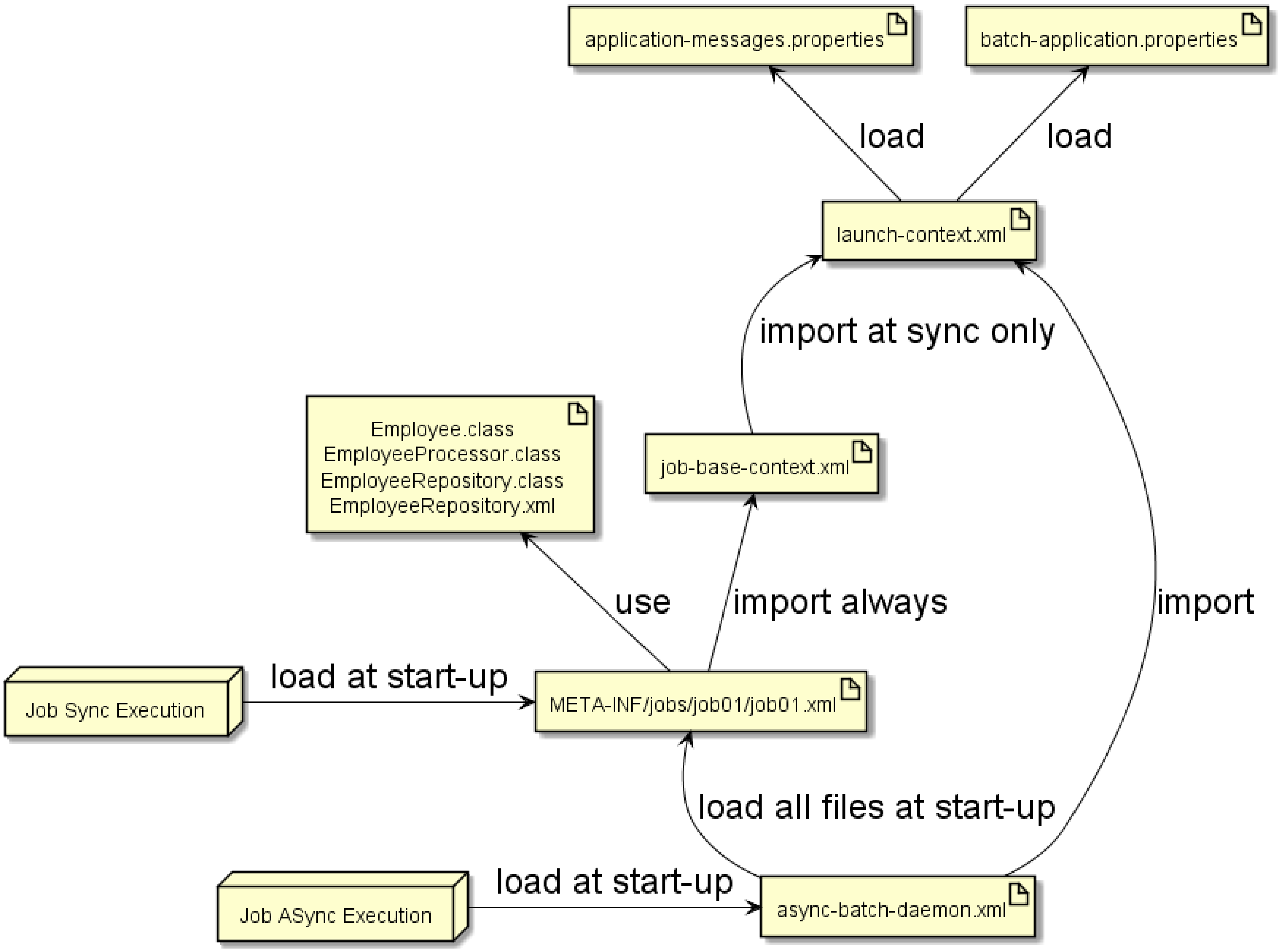
Directory configuration of project

Explanation of each element of blank project

Sr. No.	Explanation
(1)	root package that stores various classes of the entire batch application.
(2)	Package that stores various classes of 1 job. It stores DTO, implementation of Tasklet and Processor, Mapper interface of MyBatis3. Since there are no restrictions on how to store in this guideline, refer to this as an example. You can customize it with reference to default state however, consider making it easier to judge the resources specific to job.
(3)	Configuration file of the entire batch application. In the default state, the settings related to database connection and asynchronous execution are set up. You can add by referring default.
(4)	Configuration file of Logback(log output).

Sr. No.	Explanation
(5)	Configuration file that defines messages to be displayed when an error occurs during the input check using BeanValidation. In the default state, after defining default messages of BeanValidation and HibernateValidator that is its implementation, Comment-out All is done. In this state, since default messages are used, it should be modified to any message by Comment-in only when you want to customize the messages.
(6)	Mapper XML file that pairs with Mapper interface of MyBatis3.
(7)	Property file that defines messages used mainly for log output.
(8)	Directory that stores job-specific Bean definition file. The hierarchical structure can be configured according to the number of jobs.
(9)	Directory that stores Bean definition file related to the entire batch application. It is set to start a job regardless of default setting of Spring Batch or MyBatis or start trigger such as synchronous / asynchronous.
(10)	Bean definition file that describes settings related to asynchronous execution (DB polling) function.
(11)	Bean definition file to reduce various settings by importing in a job-specific Bean definition file. By importing this, the job can absorb the difference in the Bean definition by the start trigger.
(12)	Bean definition file for setting Spring Batch behavior and common jobs.

Relation figure of each file is shown below.



Relation figure of each file

3.1.4. Flow of development

Series of flow of developing job is explained.
Here, we will focus on understanding general flow and not the detailed explanation.

3.1.4.1. Import to IDE

Since the generated project is as per the project structure of Maven, import as Maven project using various IDEs.
Detailed procedures are omitted.

3.1.4.2. Setting of entire application

Customize as follows depending on user status.

- Project information of pom.xml
- Database related settings

How to customize settings other than these by individual functions is explained.

3.1.4.2.1. Project information of pom.xml

As the following information is set with temporary values in the POM of the project, values should be set as per the status.

- Project name(name element)
- Project description(description element)
- Project URL(url element)
- Project inception year(inceptionYear element)
- Project license(licenses element)
- Project organization(organization element)

3.1.4.2.2. Database related settings

Database related settings are at many places, so each place should be modified.

pom.xml

```xml
<!-- (1) -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

batch-application.properties

```
# (2)
# Admin DataSource settings.
admin.jdbc.driver=org.h2.Driver
admin.jdbc.url=jdbc:h2:mem:batch-admin;DB_CLOSE_DELAY=-1
admin.jdbc.username=sa
admin.jdbc.password=

# (2)
# Job DataSource settings.
#jdbc.driver=org.postgresql.Driver
#jdbc.url=jdbc:postgresql://localhost:5432/postgres
#jdbc.username=postgres
#jdbc.password=postgres
jdbc.driver=org.h2.Driver
jdbc.url=jdbc:h2:mem:batch;DB_CLOSE_DELAY=-1
jdbc.username=sa
jdbc.password=

# (3)
# Spring Batch schema initialize.
data-source.initialize.enabled=true
spring-batch.schema.script=classpath:org/springframework/batch/core/schema-h2.sql
terasoluna-batch.commit.script=classpath:org/terasoluna/batch/async/db/schema-
commit.sql
```

META-INF/spring/launch-context.xml

```xml
<!-- (3) -->
<jdbc:initialize-database data-source="adminDataSource"
                          enabled="${data-source.initialize.enabled:false}"
                          ignore-failures="ALL">
    <jdbc:script location="${spring-batch.schema.script}" />
    <jdbc:script location="${terasoluna-batch.commit.script}" />
</jdbc:initialize-database>

<!-- (4) -->
<bean id="adminDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${admin.jdbc.driver}"
      p:url="${admin.jdbc.url}"
      p:username="${admin.jdbc.username}"
      p:password="${admin.jdbc.password}"
      p:maxTotal="10"
      p:minIdle="1"
      p:maxWaitMillis="5000"
      p:defaultAutoCommit="false"/>

<!-- (4) -->
<bean id="jobDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${jdbc.driver}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}"
      p:maxTotal="10"
      p:minIdle="1"
      p:maxWaitMillis="5000"
      p:defaultAutoCommit="false" />

<!-- (5) -->
<bean id="jobSqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"
      p:dataSource-ref="jobDataSource" >
    <property name="configuration">
        <bean class="org.apache.ibatis.session.Configuration"
            p:localCacheScope="STATEMENT"
            p:lazyLoadingEnabled="true"
            p:aggressiveLazyLoading="false"
            p:defaultFetchSize="1000"
            p:defaultExecutorType="REUSE" />
    </property>
</bean>
```

META-INF/spring/async-batch-daemon.xml

```
<!-- (5) -->
<bean id="adminSqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"
      p:dataSource-ref="adminDataSource" >
    <property name="configuration">
        <bean class="org.apache.ibatis.session.Configuration"
              p:localCacheScope="STATEMENT"
              p:lazyLoadingEnabled="true"
              p:aggressiveLazyLoading="false"
              p:defaultFetchSize="1000"
              p:defaultExecutorType="REUSE" />
    </property>
</bean>
```

Each element in database related settings is explained

Sr. No.	Explanation
(1)	In pom.xml, define dependency relation of JDBC driver for connecting to the database to be used. In the default state, H2 Database(in-memory database) and PostgreSQL are set, however add/delete should be performed whenever required.
(2)	Set JDBC driver connection. - `admin.jdbc.xxx` is used by Spring Batch and TERASOLUNA Batch 5.x - `jdbc.xxx~` is used in individual job
(3)	Define whether or not to execute the initialization of database used by Spring Batch or TERASOLUNA Batch 5.x, and the script to be used. Since Spring Batch accesses **JobRepository** and TERASOLUNA Batch 5.x accesses **job request table** in the **asynchronous execution(DB Polling)**, database is mandatory. Whether to enable it, is based on the following. - Enable it when H2 Database is to be used. If disabled, **JobRepositoryandjob request table*** cannot be accessed and an error occurs. - When not using H2 Database, disable it to prevent accidents.
(4)	Set datasource. Tune the number of connections as necessary.
(5)	Set MyBatis behavior. Tune fetch size as necessary.

3.1.5. Creation of job

Refer to the following for how to create a job.

- Creation of chunk model job
- Creation of tasklet model job

3.1.6. Build and execution of project

Build and execution of project is explained.

3.1.6.1. Build of application

Move to the root directory of the project and execute the following command.

Build(Windows/Bash)

```
$ mvn clean dependency:copy-dependencies -DoutputDirectory=lib package
```

The following is generated by this.

- <Root directory>/target/<archetypeId>-<version>.jar
  - Jar of the created batch application is generated
- <Root directory>/lib/(Dependent Jar file)
  - A set of dependent Jar files is copied

When deploying to the test environment and the commercial environment, these Jar files can be copied to an arbitrary directory.

3.1.6.2. Switching of configuration file according to the environment

In the pom.xml of the project, the following Profile is set as the default value.

Profiles settings of pom.xml

```
<profiles>
    <!-- Including application properties and log settings into package. (default) -->
    <profile>
        <id>IncludeSettings</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <exclude-property/>
            <exclude-log/>
        </properties>
    </profile>

    <!-- Excluding application properties and log settings into package. -->
    <profile>
        <id>ExcludeSettings</id>
        <activation>
            <activeByDefault>false</activeByDefault>
        </activation>
        <properties>
            <exclude-property>batch-application.properties</exclude-property>
            <exclude-log>logback.xml</exclude-log>
        </properties>
    </profile>
</profiles>
```

Here, `Whether to include environment dependent configuration file` is switched. By utilizing this setting, it is possible to absorb the environmental difference by separately placing the configurationfile at the time of environment deployment. Moreover, by applying this, it is possible to change the configuration file to be included in Jar in the test environment and the commercial environment. An example is shown below.

Description example of pom.xml for switching configuration file for each environment

```
<build>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
        </resource>
        <resource>

<directory>${project.root.basedir}/${project.config.resource.directory.rdbms}</directo
ry>
        </resource>
    </resources>
</build>

<profiles>
    <profile>
        <id>postgresql9-local</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <dependencies>
            <dependency>
                <groupId>org.postgresql</groupId>
                <artifactId>postgresql</artifactId>
                <scope>runtime</scope>
            </dependency>
        </dependencies>
        <properties>
            <project.config.resource.directory.rdbms>
config/rdbms/postgresql9/local</project.config.resource.directory.rdbms>
        </properties>
    </profile>
    <profile>
        <id>postgresql9-it</id>
        <dependencies>
            <dependency>
                <groupId>org.postgresql</groupId>
                <artifactId>postgresql</artifactId>
                <scope>runtime</scope>
            </dependency>
        </dependencies>
        <properties>
            <project.config.resource.directory.rdbms>
config/rdbms/postgresql9/it</project.config.resource.directory.rdbms>
        </properties>
    </profile>
</profiles>
```

Maven Profile can be activated at the time of executing command as follows.

Multiple Profiles can be activated. Use effectively whenever required.

Example of activating Maven Profile

```
$ mvn -P profile-1,profile-2
```

3.1.6.2.1. Execution of application

An example of executing the job based on the above-mentioned build result, is shown.
Replace `archetypeId` and `version` in accordance with the user's environment.

Command prompt(Windows)

```
C:\xxxx> java -cp target\archetypeId-version.jar;lib\*^
    org.springframework.batch.core.launch.support.CommandLineJobRunner^
    META-INF/jobs/job01.xml job01
```

Shell(Unix, Linux, ...)

```
$ java -cp 'target/archetypeId-version.jar:lib/*' \
    org.springframework.batch.core.launch.support.CommandLineJobRunner \
    META-INF/jobs/job01.xml job01
```

Necessity to handle exit code returned by java command

In the actual system, rather than issuing a java command directly when issuing a job from the job scheduler, It is common to start by inserting shell script for starting java.

This is for setting the environment variables before starting the java command and for handling the exit code of the java command. It is recommended that `Handling of the exit code of the java command` should always be done for the following reasons.

- The normal exit code of the java command is `0` and abnormal is `1`. The job scheduler judges the success / failure of the job within the range of the exit code. Depending on the settings of the job scheduler, it judges as 'Normal end' irrespective of the fact that the java commandended abnormally.
- The exit code that can be handled by OS and job scheduler has finite range.
  - It is important to define the range of the exit code to be used by the user according to the specifications of the OS and job scheduler.
  - Generally, it is in the range of 0 to 255 which is defined by the POSIX standards.
    - In {batch 5 _ shortname}, it is set to return the normal exit code as `0` or otherwise, `255`.

An example of handling exit code is shown below.

Example of handling exit code

```
#!/bin/bash

# ..omitted.

java -cp ...
RETURN_CODE=$?
if [ $RETURN_CODE = 1 ]; then
   return 255
else
   return $RETURN_CODE
fi
```

3.2. Creation of chunk model job

3.2.1. Overview

How to create chunk model job is explained. Refer to Spring Batch architecture for the architecture of chunk model.

The components of chunk model job is explained here.

3.2.1.1. Components

The components of chunk model job are shown below. Implement 1 job by combining these components in job Bean definition file.

Components of chunk model job

Sr. No.	Name	Role	Mandatory settings	Mandatory implementation
1	ItemReader	Interface to fetch data from various resources. Since implementation for flat files and database is provided by Spring Batch, there is no need for the user to create it.	✔	−
2	ItemProcessor	Interface for processing data from input to output. The user `implements` this interface whenever required and implements business logic.	−	−
3	ItemWriter	Interface for the output of data to various resources. An interface paired with `ItemReader`. Since implementation for flat files and database is provided by Spring Batch, there is no need for the user to create it.	✔	−

The points in this table are as follows.

- If the data is to be only transferred from input resource to output resource in a simple way, it can be implemented only by setting.
- `ItemProcessor` should be implemented whenever required.

Hereafter, how to implement the job using these components, is explained.

3.2.2. How to use

How to implement chunk model job is explained in the following order here.

- Job configuration
- Implementation of components

3.2.2.1. Job configuration

Define a way to combine the elements that constitutes chunk model job in the Bean definition file. An example is shown below and the relation between components is explained.

Example of Bean definition file (Chunk model)

```
<?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:batch="http://www.springframework.org/schema/batch"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
       xsi:schemaLocation="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
             http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch.xsd
             http://mybatis.org/schema/mybatis-spring
http://mybatis.org/schema/mybatis-spring.xsd">

    <!-- (1) -->
    <import resource="classpath:META-INF/spring/job-base-context.xml"/>

    <!-- (2) -->
    <context:annotation-config/>

    <!-- (3) -->
    <context:component-scan
        base-package="org.terasoluna.batch.functionaltest.app.common" />

    <!-- (4) -->
    <mybatis:scan
        base-package="org.terasoluna.batch.functionaltest.app.repository.mst"
        factory-ref="jobSqlSessionFactory"/>

    <!-- (5) -->
    <bean id="reader"
          class="org.mybatis.spring.batch.MyBatisCursorItemReader" scope="step"

p:queryId="org.terasoluna.batch.functionaltest.app.repository.mst.CustomerRepository.f
indAll"
          p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

    <!-- (6) -->
    <!-- Item Processor -->
    <!-- Item Processor in order that based on the Bean defined by the annotations,
not defined here -->
```

```
    <!-- (7) -->
    <bean id="writer"
          class="org.springframework.batch.item.file.FlatFileItemWriter"
          scope="step"
          p:resource="file:#{jobParameters[outputFile]}">
        <property name="lineAggregator">
            <bean
class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
                <property name="fieldExtractor">
                    <bean
class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"

p:names="customerId,customerName,customerAddress,customerTel,chargeBranchId"/>
                </property>
            </bean>
        </property>
    </bean>

    <!-- (8) -->
    <batch:job id="jobCustomerList01" job-repository="jobRepository"> <!-- (9) -->
        <batch:step id="jobCustomerList01.step01"> <!-- (10) -->
            <batch:tasklet transaction-manager="jobTransactionManager"> <!-- (11) -->
                <batch:chunk reader="reader"
                             processor="processor"
                             writer="writer"
                             commit-interval="10" /> <!-- (12) -->
            </batch:tasklet>
        </batch:step>
    </batch:job>
</beans>
```

Configuration of ItemProcessor implementation class

```
@Component("processor") // (6)
public class CustomerProcessor implement ItemProcessor<Customer, Customer> {
  // omitted
}
```

Sr. No.	Explanation
(1)	Import the settings to always read the required Bean definition when using TERASOLUNA Batch 5.x.
(2)	Enable Bean definition using annotation. Use it with (3) when implementing ItemProcessor, Listener etc.
(3)	Set base package of component scan target. When defining Bean using annotation, use it with (2).
(5)	ItemReader configuration. For the details of ItemReader, refer to Database access and File access.

Sr. No.	Explanation
(6)	ItemProcessor can be defined by annotation in (2), (3), so there is no need to define in the Bean definition file.
(7)	ItemWriter configuration. For the details of ItemWriter, refer to Database access and File access.
(8)	Job configuration. The value set in the id attribute should be unique in the range of all jobs included in one batch application.
(9)	`JobRepository` configuration. The value set in the `job-repository` attribute should be fixed to `jobRepository` unless there is a special reason. This will allow all the jobs to be managed by 1 `JobRepository`. Resolve Bean definition of `jobRepository` by (1).
(10)	Step configuration. The value set in the id attribute should be unique in 1 job. By adding id attribute set in (8) as a prefix, if the step is made unique in the range of all jobs included in the batch application same as job, it can be effectively used in various situations such as for log output, identification of error occurrence. Therefore, it should be `<Job id>.<step name>`.
(11)	Tasklet configuration. The value set in the `transaction-manager` attribute should be fixed to `jobTransactionManager` unless there is a special reason. This will allow the transaction to be managed for each `commit-interval` of (12). For details, refer to Transaction control. Resolve Bean definition of `jobTransactionManager` by (1).
(12)	Chunk model job configuration. Specify Ben ID for `ItemReader`、 `ItemProcessor`、 `ItemWriter` defined in the previous steps for each attribute of `reader`, `processor` and `writer`. Set input data count per chunk in `commit-interval` attribute.

Tuning of commit-interval

`commit-interval` is the performance tuning point in chunk model job.

In the above example, 10 records are used however, exact count differs with the characteristics of available machine resource and job. In case of a job that processes data by accessing multiple resources, the process throughput may reach to 100 records from 10 records. If input/output resource is of 1:1 correspondence and there is a job of transferring data, then the process throughput may increase to 5000 records or even to 10000 records.

Temporarily set `commit-interval`` to 100 records at the time of implementing the job, and then perform tuning of each job as per the result of performance measurement.

3.2.2.2. Implementation of components

How to implement mainly ItemProcessor is explained here.

Refer to the following for other components.

- ItemReader、ItemWriter
  - Database access、 File access
- Listener
  - Listener

3.2.2.2.1. Implementation of ItemProcessor

How to implement ItemProcessor is explained.

ItemProcessor is responsible for creating **1 record** data for the output resource based on the **1 record** data fetched from the input resource as shown in the interface below. In other words, ItemProcessor is where business logic for **1 record** data is implemented.

ItemProcessor interface

```
public interface ItemProcessor<I, O> {
    O process(I item) throws Exception;
}
```

The interface indicating `I` and `O` can be of same type or of different type as shown below. Same type means modifying input data partially. Different type means to generate output data based on the input data.

Example of implementation of ItemProcessor(Input/Output is of same type)

```
@Component
public class AmountUpdateItemProcessor implements
        ItemProcessor<SalesPlanDetail, SalesPlanDetail> {

    @Override
    public SalesPlanDetail process(SalesPlanDetail item) throws Exception {
        item.setAmount(new BigDecimal("1000"));
        return item;
    }
}
```

Example of implementation of ItemProcessor(Input/Output is of different type)

```
@Component
public class UpdateItemFromDBProcessor implements
        ItemProcessor<SalesPerformanceDetail, SalesPlanDetail> {

    @Inject
    CustomerRepository customerRepository;

    @Override
    public SalesPlanDetail process(SalesPerformanceDetail readItem) throws Exception {
        Customer customer = customerRepository.findOne(readItem.getCustomerId());

        SalesPlanDetail writeItem = new SalesPlanDetail();
        writeItem.setBranchId(customer.getChargeBranchId());
        writeItem.setYear(readItem.getYear());
        writeItem.setMonth(readItem.getMonth());
        writeItem.setCustomerId(readItem.getCustomerId());
        writeItem.setAmount(readItem.getAmount());
        return writeItem;
    }
}
```

Explanation of return of null from ItemProcessor

Return of null from ItemProcessor means the data is not passed to the subsequent process (Writer). In other words, the data is filtered. This can be effectively used to validate the input data. For detail, refer to Input check.

To increase process throughput of ItemProcessor

As shown in the previous implementation example, the implementation class of ItemProcessor should access resources such as DB and files. Since ItemProcessor is executed for each record of input data, even if there is small I/O, large I/O occurs in the entire job, so it is important to suppress I/O as much as possible for increasing process throughput.

One method is to store the required data in memory in advance by utilizing Listener to be mentioned later and implement most of the processing in ItemProcessor so that it completes between CPU/ memory. However, since it consumes a large amount of memory per job, its not that anything can be stored in the memory. The data to be stored in memory based on I/O frequency and data size should be studied.

This point is introduced even in Input/Output of data.

Use multiple ItemProcessors at the same time

If a general ItemProcessor is provided to apply to each job, it can be implemented by using `CompositeItemProcessor` provided by Spring Batch and linking it.

Linking of multiple ItemProcessor by CompositeItemProcessor

```
<bean id="processor"

class="org.springframework.batch.item.support.CompositeItemProcessor">
    <property name="delegates">
        <list>
            <ref bean="commonItemProcessor"/>
            <ref bean="businessLogicItemProcessor"/>
        </list>
    </property>
</bean>
```

Note that it is processed in the order specified in the delegates attribute.

3.3. Creation of tasklet model job

3.3.1. Overview

How to create tasklet model job is explained. Refer to Spring Batch architecture for the architecture of tasklet model.

3.3.1.1. Components

Tasklet model job does not register multiple components. It only implements `org.springframework.batch.core.step.tasklet.Tasklet` and sets it in Bean definition. `ItemReader` and `ItemWriter` which are components of the chunk model can also be used as components as the advanced implementation means.

3.3.2. HowToUse

How to implement tasklet model job is explained in the following order here.

- Job configuration
- Implementation of tasklet

3.3.2.1. Job configuration

Define tasklet model job in Bean definition file. An example is shown below.

Example of Bean definition file (Tasklet model)

```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:batch="http://www.springframework.org/schema/batch"
       xsi:schemaLocation="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
             http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch.xsd">

    <!-- (1) -->
    <import resource="classpath:META-INF/spring/job-base-context.xml"/>

    <!-- (2) -->
    <context:annotation-config/>

    <!-- (3) -->
    <context:component-scan
          base-package="org.terasoluna.batch.functionaltest.app.common"/>

    <!-- (4) -->
    <batch:job id="jobSimpleJob" job-repository="jobRepository"> <!-- (5) -->
        <batch:step id="simpleJob.step01"> <!-- (6) -->
            <batch:tasklet transaction-manager="jobTransactionManager"
                           ref="simpleJobTasklet"/> <!-- (7) -->
        </batch:step>
    </batch:job>

</beans>
```

Example of tasklet implementation class

```java
package org.terasoluna.batch.functionaltest.app.common;

@Component // (3)
public class SimpleJobTasklet implements Tasklet {
  // omitted
}
```

S. No.	Explanation
(1)	Import the settings to always read the required Bean definition when using TERASOLUNA Batch 5.x.
(2)	Enable Bean definition using annotation. Use it with (3).

S. No.	Explanation
(3)	Set base package of component-scan target. Use it with (2). In the tasklet model, Bean is defined by annotation however, Bean definition of tasklet implementation class is not required in XML.
(4)	Job configuration. The value set in the id attribute should be unique in the range of all jobs included in one batch application.
(5)	`JobRepository` configuration. The value set in the `job-repository` attribute should be fixed to `jobRepository` unless there is a special reason. This will allow all the jobs to be managed in one `JobRepository`. Resolve Bean definition of `jobRepository` by (1).
(6)	Step configuration. The value set in the id attribute should be unique in one job. By adding id attribute set in (4) as a prefix if the step is made unique in the range of all jobs included in the batch application same as job, it can be effectively used in various situations such as for log output, identification of error occurrence. Therefore, it should be `<Job id>.<step name>`.
(7)	Tasklet configuration. The value set in the `transaction-manager` attribute should be fixed to `jobTransactionManager` unless there is a special reason. This will manage the processes of the entire tasklet in one transaction. For details, refer to Transaction control. Resolve Bean definition of `jobTransactionManager` by (1). The value to be set in `ref` attribute is the Bean name resolved in (3). `SimpleJobTasklet`, the tasklet implementation class name should be `simpleJobTasklet` with the first letter in lower case.

> *Bean name when using annotation*
>
> Bean name when using `@Component` annotation is generated through `org.springframework.context.annotation.AnnotationBeanNameGenerator`. Refer to Javadoc of this class when you want to confirm the naming rules.

3.3.2.2. Implementation of tasklet

First, understand the overview with simple implementation, then proceed to implementation using the components of the chunk model.

It is explained in the following order.

- Implementation of simple tasklet
- Implementation of tasklet using the components of chunk model

3.3.2.3. Implementation of simple tasklet

Basic points are explained through tasklet implementation only for log output.

Example of simple tasklet implementation class

```java
package org.terasoluna.batch.functionaltest.app.common;

// omitted

@Component
public class SimpleJobTasklet implements Tasklet { // (1)

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

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception {  // (2)
        logger.info("called tasklet."); // (3)
        return RepeatStatus.FINISHED; // (4)
    }
}
```

Sr. No.	Explanation
(1)	Implement `org.springframework.batch.core.step.tasklet.Tasklet` interface using `implements`.
(2)	Implement `execute` method to be defined by `Tasklet` interface. Arguments `StepContribution`, `ChunkContext` are used however, they are not explained here.
(3)	Implement any process. INFO log is output here.
(4)	Return whether or not the tasklet process is completed. Always specify as `return RepeatStatus.FINISHED;`.

3.3.2.4. Implementation of tasklet using the components of chunk model

Spring Batch does not mention using various components of chunk model during tasklet implementation. In TERASOLUNA Batch 5.x, you can select this depending on the following situations.

- When multiple resources are combined and processed, it is difficult to be as per the chunk model format
- When processes are implemented at various places in the chunk model, tasklet model is better to understand the overall image easily
- When recovery is made simple and you want to use batch commit of tasklet model instead of intermediate commit of chunk model

The tasklet implementation that uses `ItemReader` and `ItemWriter` which are the components of the chunk model is explained below.

Tasklet implementation example that uses the components of chunk model

```
@Component()
@Scope("step") // (1)
public class SalesPlanChunkTranTask implements Tasklet {

    @Inject
    @Named("detailCSVReader") // (2)
    ItemStreamReader<SalesPlanDetail> itemReader; // (3)

    @Inject
    SalesPlanDetailRepository repository; // (4)

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception {

        SalesPlanDetail item;

        try {
            itemReader.open(chunkContext.getStepContext().getStepExecution()
                    .getExecutionContext()); // (5)

            while ((item = itemReader.read()) != null) { // (6)

                // do some processes.

                repository.create(item); // (7)
            }
        } finally {
            itemReader.close(); // (8)
        }
        return RepeatStatus.FINISHED;
    }
}
```

Bean definition example 1

```
<!-- omitted -->
<import resource="classpath:META-INF/spring/job-base-context.xml"/>

<context:annotation-config/>

<context:component-scan
    base-package="org.terasoluna.batch.functionaltest.app.plan" />
<context:component-scan
    base-package="org.terasoluna.batch.functionaltest.ch05.transaction.component" />

<!-- (9) -->
<mybatis:scan
    base-package="org.terasoluna.batch.functionaltest.app.repository.plan"
    factory-ref="jobSqlSessionFactory"/>

<!-- (10) -->
<bean id="detailCSVReader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[inputFile]}">
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                      p:names="branchId,year,month,customerId,amount"/>
            </property>
            <property name="fieldSetMapper">
                <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

p:targetType="org.terasoluna.batch.functionaltest.app.model.plan.SalesPlanDetail"/>
            </property>
        </bean>
    </property>
</bean>

<!-- (11) -->
<batch:job id="createSalesPlanChunkTranTask" job-repository="jobRepository">
    <batch:step id="createSalesPlanChunkTranTask.step01">
        <batch:tasklet transaction-manager="jobTransactionManager"
                       ref="salesPlanChunkTranTask"/>
    </batch:step>
</batch:job>
```

Sr. No.	Explanation
(1)	Set the same step scope as the Bean scope of ItemReader to be used in this class.

Sr. No.	Explanation
(2)	Access input resources (flat files in this example) through `ItemReader`. Specify Bean name as `detailCSVReader` but it is optional for clarity purpose.
(3)	Define the type as `ItemStreamReader` that is sub-interface of `ItemReader`. This is because it is necessary to open/close the resource of (5), (8). It is supplemented later.
(4)	Access output resources (database in this example) through Mapper of MyBatis. Mapper is directly used for the sake of simplicity. There is no need to always use `ItemWriter`. Of course, `MyBatisBatchItemWriter` can be used.
(5)	Open input resource.
(6)	Loop all input resources sequentially. `ItemReader#read` returns `null` when it reads all the input data and reaches the end.
(7)	Output to database.
(8)	Resource should be closed without fail. Exception handling should be implemented. When an exception occurs, the transactions of the entire tasklet are rolled-backed, stack trace of exception is output and the job terminates abnormally.
(9)	To output to database, add the settings of `mybatis:scan`. The details are not explained here.
(10)	To enter from file, add Bean definition of `FlatFileItemReader`. The details are not explained here.
(11)	Since all the components are resolved by annotation, it is same as Implementation of simple tasklet.

On unification of scope

The scope of tasklet implementation class and Bean to be Injected should have the same scope.

For example, if `FlatFileItemReader` receives an input file path from an argument, the Bean scope should be `step`. In this case, the scope of tasklet implementation class should also be `step`.

If the scope of tasklet implementation class is set to `singleton` temporarily, after instantiating the tasklet implementation class at the time of generating `ApplicationContext` at application startup if it tries to Inject by resolving the instance of `FlatFileItemReader`, `FlatFileItemReader` will be in the `step` scope, however will not exist yet. because it is to be generated at the time of step execution. As the result, the tasklet implementation class cannot be instantiated and fails to generate `ApplicationContext`.

Regarding the type of field assigned with @Inject

Any one of the following type depending on the implementation class to be used.

- ItemReader/ItemWriter
  - Used when there is no need to open/close the target resource.
- ItemSteamReader/ItemStreamWriter
  - Used when there is a need to open/close the target resource.

It should be decided which type to use after confirming javadoc. Typical examples are shown below.

In case of FlatFileItemReader/Writer
: handle by ItemSteamReader/ItemStreamWriter

In case of MyBatisCursorItemReader
: handle by ItemStreamReader

In case of MyBatisBatchItemWriter
: handle by ItemWriter

Another example is shown when `ItemReader` and `ItemWriter` are used at the same time.

Tasklet implementation example 2 that uses the components of chunk model

```
@Component
@Scope("step")
public class SalesPerformanceTasklet implements Tasklet {


    @Inject
    ItemStreamReader<SalesPerformanceDetail> reader;

    @Inject
    ItemWriter<SalesPerformanceDetail> writer; // (1)

    int chunkSize = 10; // (2)

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception {

        try {
            reader.open(chunkContext.getStepContext().getStepExecution()
                    .getExecutionContext());

            List<SalesPerformanceDetail> items = new ArrayList<>(chunkSize); // (2)
            SalesPerformanceDetail item = null;
            do {
```

```java
                // Pseudo operation of ItemReader
                for (int i = 0; i < chunkSize; i++) { // (3)
                    item = reader.read();
                    if (item == null) {
                        break;
                    }
                    // Pseudo operation of ItemProcessor
                    // do some processes.

                    items.add(item);
                }

                // Pseudo operation of ItemWriter
                if (!items.isEmpty()) {
                    writer.write(items); // (4)
                    items.clear();
                }
            } while (item != null);
        } finally {
            try {
                reader.close();
            } catch (Exception e) {
                // do nothing.
            }
        }

        return RepeatStatus.FINISHED;
    }
}
```

Bean definition example 2

```xml
<!-- omitted -->
<import resource="classpath:META-INF/spring/job-base-context.xml"/>

<context:annotation-config/>
<context:component-scan
    base-package="org.terasoluna.batch.functionaltest.app.common,
        org.terasoluna.batch.functionaltest.app.performance,
        org.terasoluna.batch.functionaltest.ch06.exceptionhandling"/>
<mybatis:scan
    base-package="org.terasoluna.batch.functionaltest.app.repository.performance"
    factory-ref="jobSqlSessionFactory"/>

<bean id="detailCSVReader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[inputFile]}">
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                      p:names="branchId,year,month,customerId,amount"/>
            </property>
            <property name="fieldSetMapper">
                <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

p:targetType="org.terasoluna.batch.functionaltest.app.model.performance.SalesPerforman
ceDetail"/>
            </property>
        </bean>
    </property>
</bean>

<!-- (1) -->
<bean id="detailWriter"
      class="org.mybatis.spring.batch.MyBatisBatchItemWriter"

p:statementId="org.terasoluna.batch.functionaltest.app.repository.performance.SalesPer
formanceDetailRepository.create"
      p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"/>


<batch:job id="jobSalesPerfTasklet" job-repository="jobRepository">
    <batch:step id="jobSalesPerfTasklet.step01">
        <batch:tasklet ref="salesPerformanceTasklet"
                       transaction-manager="jobTransactionManager"/>
    </batch:step>
</batch:job>
```

Sr. No.	Explanation
(1)	Use `MyBatisBatchItemWriter` as the implementation of `ItemWriter`.
(2)	`ItemWriter` outputs a fixed number of records collectively. It processes and output 10 records each.
(3)	As per the behavior of chunk model, it should be read→process→read→process→…→write.
(4)	Output through `ItemWriter` collectively.

Decide each time whether to use the implementation class of `ItemReader` or `ItemWriter`. For file access, the implementation class of `ItemReader` and `ItemWriter` can be used. For other than this such as database access, there is no need to use compulsorily. It can be used to improve performance.

3.4. How to choose chunk model or tasklet model

Here, how to choose chunk model and tasklet model is explained by organizing each feature. Refer to the following chapters which are explained in detail appropriately.

Understand the following contents as examples of concepts without any constraints or recommendations. Refer to it while creating a job depending on the characteristics of the users and systems.

The main differences between the chunk model and the tasklet model are given below.

Comparison of chunk model and tasklet model.

Item	Chunk	Tasklet
Components	It consists of 3 components mainly `ItemReader`, `ItemProcessor` and `ItemWriter`.	It is consolidated in one `Tasklet`.
Transaction	A certain number of records are processed by issuing intermediate commit. Batch commit cannot be done. It can be processed by specific machine resources regardless of the data count. If an error occurs in the midway, then unprocessed data and processed data will get mixed.	The data is entirely processed by batch commit. There is a need for the user to implement intermediate commit. If the data to be processed is large, machine resources may get exhausted. If an error occurs in the midway, only the unprocessed data is rolled back.
Restart	It can be restarted based on the record count.	It cannot be restarted based on the record count.

Based on this, we will introduce some examples of using each one as follows.

To make recovery as simple as possible

When the job having error, is to be recovered by only re-running the target job, tasklet model can be chooseed to make recovery simple.
In chunk model, it should be dealt by returning the processed data to the state before executing the job and by creating a job to process only the unprocessed data.

To consolidate the process contents

When you want to prioritize the outlook of job such as 1 job in 1 class, tasklet can be chooseed.

To process large data stably

When performing batch process of 10 million records, consider to use chunk model in case the record count that influences the resources is the target. It means stabilizing the process by intermediate commit. Even in tasklet model, intermediate commit can be used, but it is simpler to implement in chunk model.

To restart based on the record count for the recovery after error

When batch window is difficult and you want to resume from error data onwards, chunk model should be chooseed to use restart based on the record count provided by Spring Batch. This eliminates the need to create that mechanism for each job.

Chunk model and tasklet model are basically used in combination.
It is not necessary to implement only one model in all jobs in the batch system.
Use one model based on the characteristics of jobs of the entire system and use the other model in accordance with the situation.

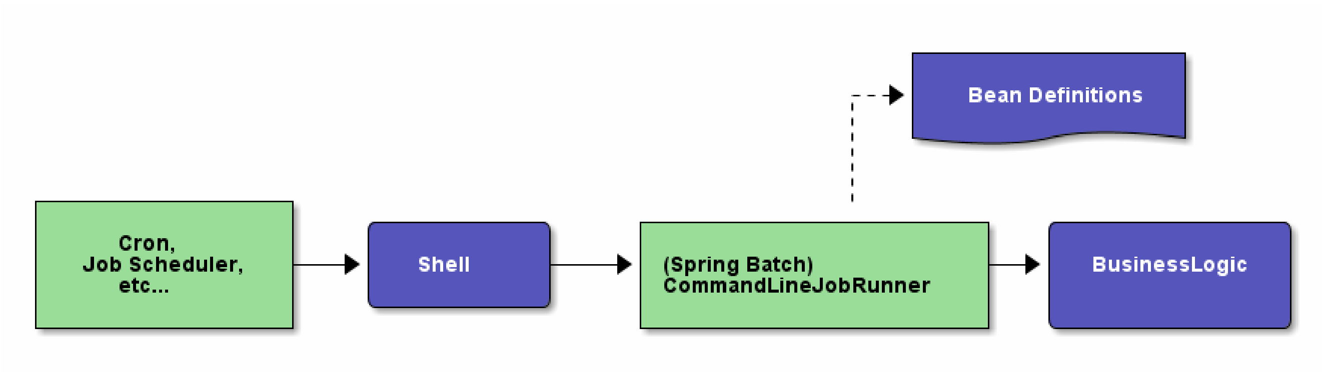
For example, in most cases it is to choose a tasklet model if there is a margin in the number of processesing records and processing time. In a very small number of cases, choosing a chunk model for jobs that process large numbers of records.

Chapter 4. Running a job

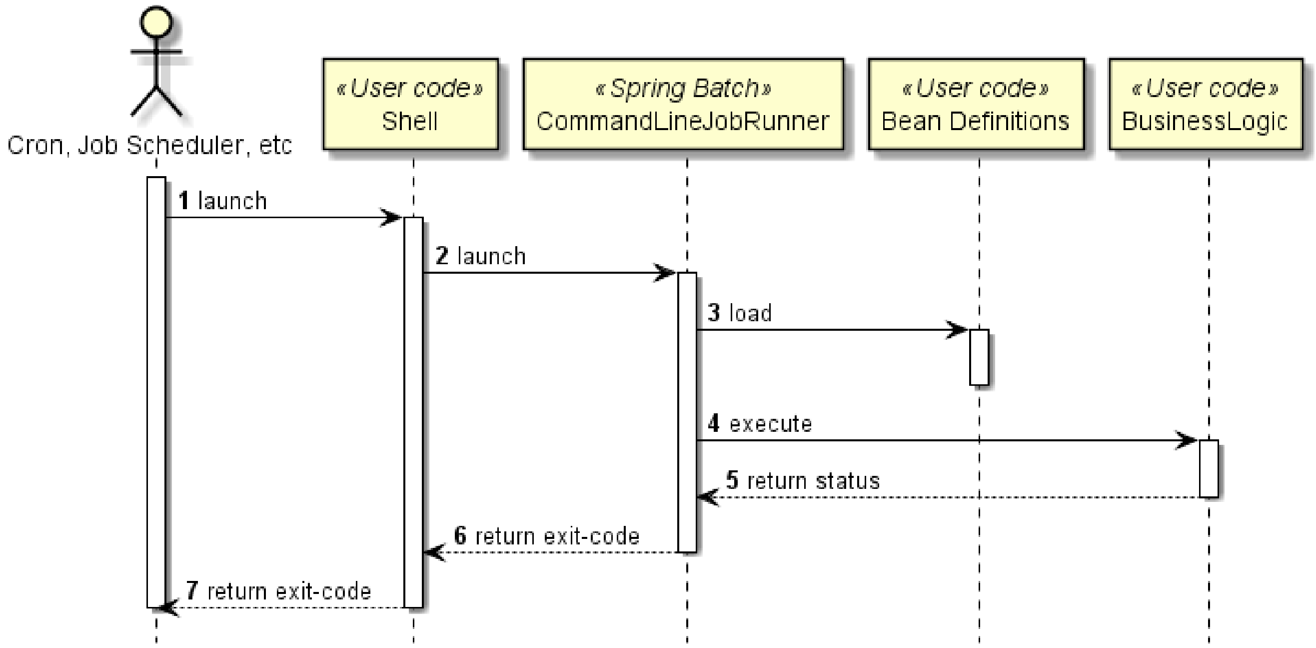
4.1. Synchronous job

4.1.1. Overview

Synchronous job is explained. Synchronous job is the execution method of launching a new process through shell by job scheduler and returning the execution result of the job to the caller.



Overview of synchronous job



Sequence of synchronous job

The usage method of this function is same in the chunk model as well as tasklet model.

4.1.2. How to use

How to running a job by `CommandLineJobRunner` is explained.

Refer to Create project for building and executing the application. Refer to Job parameters for how to specify and use job parameters. Some explanation given in the above reference and in this section overlap however, the elements of synchronous job are mainly explained.

4.1.2.1. How to run

In TERASOLUNA Batch 5.x, run the synchronous job using `CommandLineJobRunner` provided by Spring Batch. Start `CommandLineJobRunner` by issuing java command as shown below.

CommandLineJobRunner syntax

```
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner <jobPath>
<options> <jobIdentifier> <jobParameters>
```

Items to be specified by the arguments

Items to be specified	Explanation	Required
jobPath	Bean definition file path where the settings of the job to be run are described. Specify by relative path from classpath.	✔
options	Specify various options (stop, restart etc.) at the time of launching.	
jobIdentifier	Specify job name in Bean definition or job run ID after job execution as the job identifier. Normally, specify job name. Specify job run ID only when specifying stop and restart options.	✔
jobParameters	Specify job arguments. Specify in `key=value` format.	

The execution example when only the required items are specified, is shown below.

```
. Execution example 1 of CommandLineJobRunner
----
$ java -cp 'target/archetypeId-version.jar:lib/*' \ # (1)
    org.springframework.batch.core.launch.support.CommandLineJobRunner \ # (2)
    META-INF/jobs/job01.xml job01 # (3)
----
```

Settings of Bean definition(Abstract)

```xml
<batch:job id="job01" job-repository="jobRepository"> <!-- (3) -->
    <batch:step id="job01.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="employeeReader"
                         processor="employeeProcessor"
                         writer="employeeWriter" commit-interval="10" />
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Items list of setting contents

Sr. No.	Explanation
(1)	Specify the batch application jar and dependent jar in `classpath` at the time of executing `java` command. Here, it is specified by command arguments however, environment variables can also be used.
(2)	Specify `CommandLineJobRunner` with FQCN in the class to be run.
(3)	Pass the run arguments along the `CommandLineJobRunner`. Here, 2 job names are specified as `jobPath` and `jobIdentifier`.

Execution example when launch parameters are specified as the optional items, is shown below.

Execution example 2 of CommandLineJobRunner

```
$ java -cp 'target/archetypeId-version.jar:lib/*' \
    org.springframework.batch.core.launch.support.CommandLineJobRunner \
    META-INF/jobs/setupJob.xml setupJob target=server1 outputFile=/tmp/result.csv #
(1)
```

Items list of setting contents

Sr. No.	Explanation
(1)	`target=server1` and `outputFile=/tmp/result.csv` are specified as job running parameters.

4.1.2.2. Options

Supplement the options indicated in CommandLineJobRunner syntax.

In `CommandLineJobRunner`, the following 4 launch options can be used. Here, only the overview of each option is explained.

-restart

Restarts the failed job. Refer to Reprocessing for the details.

-stop

Stops a running job. Refer to Job management for the details.

-abandon

Abandons a stopped job. The abandoned job cannot be restarted. In TERASOLUNA Batch 5.x, there is no case of using this option, hence it is not explained.

-next

Runs the job executed once in the past, again. However, in TERASOLUNA Batch 5.x, this option is not used.
In TERASOLUNA Batch 5.x, it is for avoiding the restriction "Running the job by the same parameter is recognized as the same job and the same job can be executed only once" that is given by default in Spring Batch.
The details are explained in regarding parameter conversion class.
For using this option, implementation class of `JobParametersIncrementer` interface is required, it

is not set inTERASOLUNA Batch 5.x.
Therefore, when this option is specified and launched, an error occurs because the required Bean definition does not exist.

4.2. Job parameters

4.2.1. Overview

This section explains about using the job parameter (hereafter referred to as 'parameter').

The usage method of this function is same in the chunk model as well as tasklet model.

A parameter is used to flexibly switch the operation of the job according to the execution environment and execution timing as shown below.

- File path of process target
- System operation date and time

The following explanation is about assigning parameters.

1. Assign from command-line arguments
2. Redirect from file to standard input

The specified parameters can be referred in Bean definition or in Java under Spring management.

4.2.2. How to use

4.2.2.1. Regarding parameter conversion class

In Spring Batch, the received parameters are processed in the following sequence.

1. The implementation class of `JobParametersConverter` convert to `JobParameters`.
2. Refer to the parameters from `JobParameters` in Bean definition and Java under Spring management.

Regarding implementation class of parameter conversion class

Multiple implementation classes of the above mentioned `JobParametersConverter` are provided. The features of each class are shown below.

- DefaultJobParametersConverter
  - It can specify the data type of parameters(4 types; String, Long, Date, Double).
- JsrJobParametersConverter
  - It cannot specify the data type of parameters (Only String).
  - It assigns ID (RUN_ID) that identifies job execution to parameter with the name `jsr_batch_run_id` automatically.
    - It increments the RUN_ID each time the job is executed. Since it uses SEQUENCE (name is `JOB_SEQ`) of the database for incrementing, the name does not overlap.
    - In Spring Batch, runnaing the job by the same parameters is identified as the same job and the same job can be executed only once. Whereas, adding a unique value to the parameter name `jsr_batch_run_id` will recognize it as a separate job. Refer to Spring

Batch architecture for details.

In Spring Batch, when the implementation class of `JobParametersConverter` to be used in Bean definition, is not specified, `DefaultJobParametersConverter` is used.
However, in TERASOLUNA Batch 5.x, `DefaultJobParametersConverter` is not used due to the following reasons.

- It is common to run one job by the same parameter at different timing.
- It is possible to specify the time stamp of the start time and manage them as different jobs, but it is complicated to specify job parameters only for that purpose.
- `DefaultJobParametersConverter` can specify data types for parameters, but handling becomes complicated when type conversion fails.

In TERASOLUNA Batch 5.x, by using `JsrJobParametersConverter`, RUN_ID is automatically assigned without the user knowledge. By this, the same job is handled as a different job in Spring Batch as seen by the user.

About setting of parameter conversion class

In TERASOLUNA Batch 5.x, it is set in advance so as to use `JsrJobParametersConverter` in `launch-context.xml`.
Therefore, when TERASOLUNA Batch 5.x is used with the recommended setting, there is no need to set `JobParametersConverter`.

META-INF\spring\launch-context.xml

```
<bean id="jobParametersConverter"
      class="org.springframework.batch.core.jsr.JsrJobParametersConverter"
      c:dataSource-ref="adminDataSource" />

<bean id="jobOperator"
      class="org.springframework.batch.core.launch.support.SimpleJobOperator"
      p:jobRepository-ref="jobRepository"
      p:jobRegistry-ref="jobRegistry"
      p:jobExplorer-ref="jobExplorer"
      p:jobParametersConverter-ref="jobParametersConverter"
      p:jobLauncher-ref="jobLauncher" />
```

The following description assumes that JsrJobParametersConverter is used.

4.2.2.2. Assign from command-line arguments

Firstly, how to assign from the most basic command-line arguments, is explained.

Assignment of parameters

Command-line arguments are enumerated in the `<Parameter name>=<Value>` format after 3rd argument of `CommandLineJobRunner`.

The number and length of parameters are not restricted in Spring Batch and {batch 5 _ shortname}.

However, there are restrictions in the length of command arguments in the OS.
Therefore, when a large number of arguments is required, the method of Redirect from file to standard input and Using parameters and properties together should be used.

Example of setting parameters as command-line arguments

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
    JobDefined.xml JOBID param1=abc outputFileName=/tmp/result.csv
```

Refer to parameters

Parameters can be referred in Bean definition or in Java as shown below.

- Refer in Bean definition
  - It can be referred by `#{jobParamaters[xxx]}`
- Refer in Java
  - It can be referred by `@Value("#{jobParameters[xxx]}")`

> *The scope of the Bean that refers to JobParameters should be Step scope*
>
> When referring to `JobParameters`, the scope of the Bean to be referred should be set to `Step` scope. This is for using the mechanism of **late binding** of Spring Batch when `JobParameters` is to be referred.
>
> As its name implies, **late binding** is setting of the delayed value. `ApplicationContext` of Spring Framework generates an instance of `ApplicationContext` after resolving the properties of various Beans by default. Spring Batch does not resolve the property at the time of generating an instance of `ApplicationContext`. It has a function to resolve the property when various Beans are required. This is what the word **Delay** means. With this function, after generating and executing `ApplicationContext` required for executing the Spring Batch itself, it is possible to alter the behavior of various Beans according to parameters.
>
> `Step` scope is a unique scope of Spring Batch and a new instance is generated for each Step execution. The value with **late binding** can be resolved by using SpEL expression in Bean definition.

@StepScope annotation cannot be used for specifying Step scope

In Spring Batch, `@StepScope` is provided as the annotation that specifies `Step` scope. However, this is an annotation that can only be used in JavaConfig.

Therefore, specify the `Step` scope in TERASOLUNA Batch 5.x by any one of the following methods.

1. In Bean definition, assign `scope="step"` to Bean.
2. In Java, assign `@Scope("step")` to class.

Example of referring to the parameter assigned by the command-line arguments in Bean definition

```
<!-- (1) -->
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[inputFile]}">  <!-- (2) -->
    <property name="lineMapper">
        <!-- omitted settings -->
    </property>
</bean>
```

Items list of setting contents

Sr. No.	Explanation
(1)	Specify scope as scope attribute in bean tag.
(2)	Specify the parameter to be referred.

Example of referring to the parameter assigned by the command-line arguments in Java

```
@Component
@Scope("step")  // (1)
public class ParamRefInJavaTasklet implements Tasklet {

    /**
     * Holds a String type value
     */
    @Value("#{jobParameters[str]}")  // (2)
    private String str;

    // omitted execute()
}
```

Items list of setting contents

Sr. No.	Explanation
(1)	Specify scope by assigning `@Scope` annotation in class.

Sr. No.	Explanation
(2)	Specify the parameter to be referred by using `@Value` annotation.

4.2.2.3. Redirect from file to standard input

How to redirect from file to standard input is explained.

Creation of file for defining parameters

Define the parameters in the files as follows.

params.txt

```
param1=abc
outputFile=/tmp/result.csv
```

Redirect the files wherein parameters are defined to standard input

Redirect the files wherein parameters are defined as command-line arguments.

Execution method

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
    JobDefined.xml JOBID < params.txt
```

Refer to parameters

How to refer to the parameters is same as the Assign from command-line arguments method.

4.2.2.4. Set the default value of parameter

When parameters are optional, default values can be set in the following format.

- `#{jobParameters[Parameter name] ?: Default value}`

However, in the item where the value is set using parameters, the default values can also differ with the environment and execution timing same as the parameters.

Firstly, how to hardcode the default values in source code is explained. However, there are many cases where it is better to use Using parameters and properties together, so refer them also.

Refer to the parameter wherein default value is set

When the relevant parameter is not set, the value set as the default value is referred.

Example of referring to the parameter assigned by the command-line arguments in Bean definition

```
<!-- (1) -->
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParamaters[inputFile] ?: /input/sample.csv}">  <!-- (2)
-->
    <property name="lineMapper">
        // omitted settings
    </property>
</bean>
```

Items list of setting contents

Sr. No.	Explanation
(1)	Specify the scope as scope attribute in the bean tag.
(2)	Specify the parameter to be referred. `/input/sample.csv` is set as the default value.

Example of referring to the parameter assigned by the command-line arguments in Java

```
@Component
@Scope("step")  // (1)
public class ParamRefInJavaTasklet implements Tasklet {

    /**
     * Holds a String type value
     */
    @Value("#{jobParameters[str] ?: xyz}")  // (2)
    private String str;

    // omitted execute()
}
```

Items list of setting contents

Sr. No.	Explanation
(1)	Specify the scope by assigning `@Scope` annotation in class.
(2)	Specify the parameter to be referred by using `@Value` annotation. `xyz` is set as the default value.

4.2.2.5. Validation of parameters

Validation of the parameters is required at job launch in order to prevent operation errors or unintended behavior.
Validation of parameters can be implemented by using the `JobParametersValidator` provided by Spring Batch.

Since parameters are referred at various places such as ItemReader/ItemProcessor/ItemWriter, validation is performed immediately after the job is launched.

There are two ways to verify the validity of a parameter, and it differs with the degree of complexity of the verification.

- Simple validation
  - Application example
    - Verify that the required parameters are set
    - Verify that the unspecified parameters are not set
  - Validator to be used
    - `DefaultJobParametersValidator` provided by Spring Batch
- Complex validation
  - Application example
    - Numerical value range verification and complex verification such as correlation check between parameters
    - Verification that cannot be done by `DefaultJobParametersValidator` provided by Spring Batch
  - Validator to be used
    - Class wherein `JobParametersValidator` is implemented independently

How to verify the validity of Simple validation and Complex validation is explained respectively.

4.2.2.5.1. Simple validation

Spring Batch provides `DefaultJobParametersValidator` as the default implementation of `JobParametersValidator`.
This validator can verify the following as per the settings.

- Required parameters should be set
- Parameters other than required or optional should not be specified

Definition example is shown as follows.

Definition of validation that uses DefaultJobParametersValidator

```xml
<!-- (1) -->
<bean id="jobParametersValidator"
      class="org.springframework.batch.core.job.DefaultJobParametersValidator">
  <property name="requiredKeys">  <!-- (2) -->
    <list>
        <value>jsr_batch_run_id</value>  <!-- (3) -->
        <value>inputFileName</value>
        <value>outputFileName</value>
    </list>
  </property>
  <property name="optionalKeys">  <!-- (4) -->
    <list>
        <value>param1</value>
        <value>param2</value>
    </list>
  </property>
</bean>

<batch:job id="jobUseDefaultJobParametersValidator" job-repository="jobRepository">
  <batch:step id="jobUseDefaultJobParametersValidator.step01">
    <batch:tasklet ref="sampleTasklet" transaction-manager="jobTransactionManager"/>
  </batch:step>
  <batch:validator ref="jobParametersValidator"/>  <!-- (5) -->
</batch:job>
```

Items list of setting contents

Sr. No.	Explanation
(1)	Define Bean for `DefaultJobParametersValidator`.
(2)	Set the required parameters to property `requiredKeys`. Multiple parameter names of the required parameters can be specified using list tag.
(3)	Set `jsr_batch_run_id` to the required parameters. In TERASOLUNA Batch 5.x, this setting is mandatory when using `DefaultJobParametersValidator`. The reason for making the setting mandatory is explained later.
(4)	Set optional parameters to property `optionalKeys`. Multiple parameter names of the optional parameters can be specified using list tag.
(5)	Apply the validator to the job using validator tag in the job tag.

Required parameters that cannot be omitted in TERASOLUNA Batch 5.x

`JsrJobParametersConverter` is used for parameter conversion in {batch 5 _ shortname}, so the following parameters are always set.

- `jsr_batch_run_id`

Therefore, `jsr_batch_run_id` should be included in the `requiredKeys`.
Refer to Regarding parameter conversion class for detailed explanation.

Example of parameter definition

```
<bean id="jobParametersValidator"

class="org.springframework.batch.core.job.DefaultJobParametersValidato
r">
  <property name="requiredKeys">
    <list>
        <value>jsr_batch_run_id</value>  <!-- mandatory -->
        <value>inputFileName</value>
        <value>outputFileName</value>
    </list>
  </property>
  <property name="optionalKeys">
    <list>
        <value>param1</value>
        <value>param2</value>
    </list>
  </property>
</bean>
```

OK case and NG case when DefaultJobParametersValidator is used

An example when the verification result is OK and NG are shown to understand the verification possible conditions in `DefaultJobParametersValidator`.

DefaultJobParametersValidator definition example

```
<bean id="jobParametersValidator"
    class="org.springframework.batch.core.job.DefaultJobParametersValidator"
    p:requiredKeys="outputFileName"
    p:optionalKeys="param1"/>
```

NG case1

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
    JobDefined.xml JOBID param1=aaa
```

NG as the required parameter `outputFile` is not set.

NG case 2

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
    JobDefined.xml JOBID outputFileName=/tmp/result.csv param2=aaa
```

NG as the parameter `param2` which is not specified for either the required parameter or the optional parameter is set.

OK case 1

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
    JobDefined.xml JOBID param1=aaa outputFileName=/tmp/result.csv
```

OK as the parameters specified as required and optional are set.

OK case 2

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
    JobDefined.xml JOBID fileoutputFilename=/tmp/result.csv
```

OK as the required parameters are set and there is no need to set optional parameters.

4.2.2.5.2. Complex validation

Implementing `JobParametersValidator` interface independently helps in verifying the parameters as per requirements.

Implement `JobParametersValidator` class as follows.

- Implement `JobParametersValidator` class and override validate method
- Implement validate method as follows
  - Fetch each parameter from `JobParameters` and verify
    - If the verification result is OK, there is no need to perform any operation
    - If verification result is NG, throw `JobParametersInvalidException`

Implementation example of `JobParametersValidator` class is shown. In this case, it is verified that the length of the string specified by `str` is less than or equal to the number specified by `num`.

Implementation example of JobParametersValidator interface

```
public class ComplexJobParametersValidator implements JobParametersValidator {  // (1)
    @Override
    public void validate(JobParameters parameters) throws
JobParametersInvalidException {
        Map<String, JobParameter> params = parameters.getParameters();  // (2)

        String str = params.get("str").getValue().toString();  // (3)
        int num = Integer.parseInt(params.get("num").getValue().toString()); // (4)

        if(str.length() > num){
            throw new JobParametersInvalidException(
            "The str must be less than or equal to num. [str:"
                    + str + "][num:" + num + "]");  // (5)
        }
    }
}
```

Items list of setting contents

Sr. No.	Explanation
(1)	Implement `JobParametersValidator` class and override validate method.
(2)	Receive the parameters as arguments in `JobParameters` type. By setting `parameters.getParameters()`, it is easier to refer the parameters by fetching them in Map format.
(3)	Get parameters by specifying key.
(4)	Convert parameters to int type. When handling parameters of other than String type, they should be appropriately converted.
(5)	Validation result is NG when the string length of the parameter `str` exceeds the value of parameter `num`.

Job definition example

```
<batch:job id="jobUseComplexJobParametersValidator" job-repository="jobRepository">
    <batch:step id="jobUseComplexJobParametersValidator.step01">
        <batch:tasklet ref="sampleTasklet" transaction-manager=
"jobTransactionManager"/>
    </batch:step>
    <batch:validator>  <!-- (1) -->
        <bean
class="org.terasoluna.batch.functionaltest.ch04.jobparameter.ComplexJobParametersValid
ator"/>
    </batch:validator>
</batch:job>
```

Items list of setting contents

Sr. No.	Explanation
(1)	Apply validator in the job by using validator tag in the job tag.

Regarding validation of parameters at asynchronous start

By the asynchronous start method (DB polling and Web container), it is possible to verify the parameters at the job launch in the same way, however, it is desirable to verify them before launching the job at the following timing.

- DB polling
  - Before INSERTing to job request table
- Web container
  - At the time of calling Controller (assign @Validated)

In case of asynchronous start, since it is necessary to confirm the result separately, errors such as parameter settings should be responded quickly and job requests should be rejected.

For validation in this case, there is no need to use `JobParametersValidator`. The function to INSERT to the job request table and the controller in the Web container mostly should not depend on Spring Batch and it is better to avoid depending on Spring Batch since only `JobParametersValidator` is used.

4.2.3. How to extend

4.2.3.1. Using parameters and properties together

Spring Framework based on Spring Batch is equipped with the property management function to enable it to handle the values set in the environment variables and property files. For details, refer to Property management of TERASOLUNA Server 5.x Development Guideline.

By combining properties and parameters, it is possible to overwrite some parameters after making common settings for most jobs in the property file.

About when parameters and propertis are resolved

As mentioned above, parameters and properties are different components that provide the function.
Spring Batch has a function of parameter management and Spring Framework has a function of property management.
This difference appears in the description method.

- In case of function possessed by Spring Batch
  - `#{jobParamaters[xxx]}`
- In case of function possessed by Spring Framework
  - `@Value("${xxx}")`

The timing of resolving each value is different.

- In case of function possessed by Spring Batch
  - It is set when the job is executed after generating Application Context.
- In case of function possessed by Spring Framework
  - It is set at the time of generating Application Context.

Therefore, the parameter value is given priority by Spring Batch.
Note that since the application is effective when they are combined together, both of them should be treated individually

How to set by combining properties and parameters, is explained.

In addition to the setting by environment variables, when additional settings is done by command-line arguments

In addition to the setting by environment variables, how to set the parameters using command-line arguments, is explained.
It is possible to refer to it in the same manner as Bean definition.

Example of setting parameters by command-line arguments in addition to environment variables

```
# Set environment variables
$ export env1=aaa
$ export env2=bbb

# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
    JobDefined.xml JOBID param3=ccc outputFile=/tmp/result.csv
```

Example of referring environment variables and parameters in Java

```
@Value("${env1}")  // (1)
private String param1;

@Value("${env2}")  // (1)
private String param2;

private String param3;

@Value("#{jobParameters[param3]")  // (2)
public void setParam3(String param3) {
    this.param3 = param3;
}
```

Items list of setting contents

Sr. No.	Explanation
(1)	Specify the environment variables to be referred by using `@Value` annotation. The format for reference is `${Environment variable name}`.
(2)	Specify the parameters to be referred by using `@Value` annotation. The format for reference is `#{jobParameters[Parameter name]`.

Example when environment variables are default

```
# Set environment variables
$ export env1=aaa

# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
    JobDefined.xml JOBID param1=bbb outputFile=/tmp/result.csv
```

Example of referring parameters by setting default values for environment variables in Java

```
@Value("#{jobParameters[param1] ?: '${env1}'}")  // (1)
public void setParam1(String param1) {
    this.param1 = param1;
}
```

Items list of setting contents

Sr. No.	Explanation
(1)	Specify the parameters to be referred by using `@Value` annotation by setting default values in environment variables. When parameters are not set, the value of environment variables are set.

How to set incorrect default values

When the following is defined and param1 is not set by command-line arguments, note that null is set in param1 irrespective of the fact that you want to set env1 value.

Setting method example of incorrect default value

```
@Value("${env1}")
private String param1;

@Value("#{jobParameters[param1]}")
public void setParam1(String param1) {
  this.param1 = param1;
}
```

4.3. Asynchronous execution (DB polling)

4.3.1. Overview

Running a job using DB polling is explained.

The usage method of this function is same in the chunk model as well as tasklet model.

4.3.1.1. What is asynchronous execution by using DB polling?

A dedicated table which registers jobs to be executed asynchronously (hereafter referred to as Job-request-table) is monitored periodically and job is asynchronously executed based on the registered information.
In TERASOLUNA Batch 5.x, a module which monitors the table and starts the job is defined with the name asynchronous batch daemon. Asynchronous batch daemon runs as a single Java process and executes by assigning threads in the process for each job.

4.3.1.1.1. Functions offered by TERASOLUNA Batch 5.x

TERASOLUNA Batch 5.x offers following functions as **Asynchronous execution (DB polling)**.

List of asynchronous execution (DB polling) functions

Function	Description
Asynchronous batch daemon function	A function which permanently executes Job-request-table polling function
Job-request-table polling function	A function which asynchronously executes the job based on information registered in the Job-request-table. It also offers a table definition of Job-request-table.

Usage premise

Only job requests are managed in Job-request-table. Execution status and results of requested job is entrusted to `JobRepository`. It is assumed that job status is managed through these two factors.

Further, if in-memory database is used in `JobRepository`, `JobRepository` is cleared after terminating asynchronous batch daemon and job execution status and results cannot be referred. Hence, it is assumed that a database that is ensured to be persistent is used in `JobRepository`.

Using in-memory database

When job execution results success or failure can be obtained without referring `JobRepository`, in-memory database can be used.
When long term continuous operations are performed in in-memory database, a large quantity of memory resources are likely to get consumed resulting in adverse effect on job execution.
In other words, in-memory database is not suitable for long term continuous operations and should be restarted periodically.
However, if it is to be used for long term continuous operations, maintenance work like deleting data periodically from `JobRepository` is necessary.
In case of a restart, if initialization is enabled, it gets recreated at the time of restart. Hence, maintenance is not required. For initialization, refer Database related settings.

4.3.1.1.2. Usage scene

A few scenes which use asynchronous execution (DB polling).

List of application scenes

Usage scene	Description
Delayed processing	When it is not necessary to complete the operation immediately in coordination with online processing and the operation which takes time to process is to be extracted as a job.
Continuous execution of jobs with short processing time	When continuous processing is done for a few seconds or a few tens of seconds for 1 job. It is possible to avoid compression of resources by start and stop of Java process for 1 job, by using asynchronous execution (DB polling). Further, since it leads to omission of start and end processing, it is possible to reduce execution time of the job.
Aggregation of large number of jobs	Same as continuous execution of jobs with short processing time.

Points to choose asynchronous execution(DB polling) instead of asynchronous execution (Web container)

Points to choose asynchronous execution(DB polling) instead of Asynchronous execution (Web container) are shown below.

- A hurdle in the introduction of WebAP server in batch processing
- Consider only database while ensuring availability
  - Alternatively, since the access is concentrated in the database, scale is not likely to be like asynchronous execution (Web container).

Reasons not to use Spring Batch Integration

The same function can be implemented by using Spring Batch Integration. However, when Spring Batch Integration is used, it is necessary to understand and fetch technical elements including the elements other than that of asynchronous execution.
Accordingly, application of Spring Batch Integration is deferred in order to avoid difficulty in understanding / use / customization of this function.

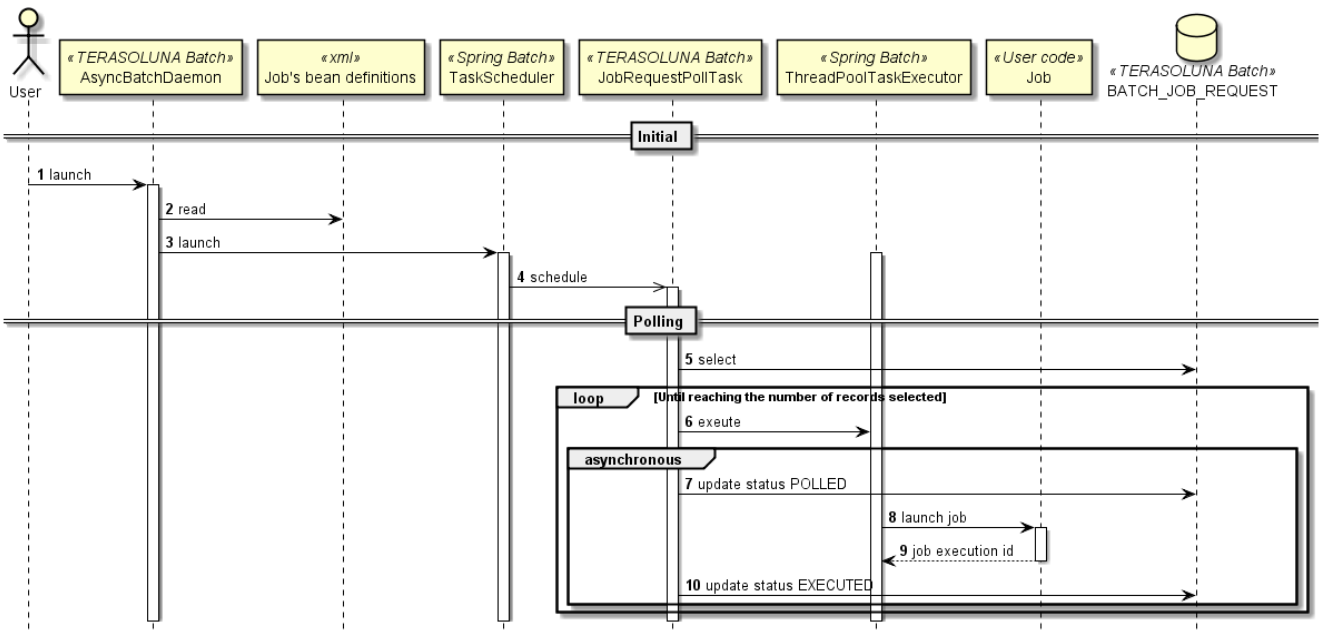
Precautions in asynchronous execution (DB polling)

When a large number of super short batches which are less than several seconds for 1 job are to be executed, database including `JobRepository` is accessed every time. Since performance degradation can occur at this point of time, mass processing of super short batches is not suitable for asynchronous execution (DB polling). This point must be adequately reviewed while using this function to check whether target performance is met.

4.3.2. Architecture

4.3.2.1. Processing sequence of DB polling

Processing sequence of DB polling is explained.



Processing sequence diagram of DB polling

1. Launch `AsyncBatchDaemon` from sh etc.
2. `AsyncBatchDaemon` reads all Bean definition files which defines the jobs at the startup.
3. `AsyncBatchDaemon` starts `TaskScheduler` for polling at regular intervals.
    - `TaskScheduler` starts a specific process at regular interval.
4. `TaskScheduler` starts `JobRequestPollTask` (a process which performs polling of Job-request-table).

5. `JobRequestPollTask` fetches a record for which the polling status is "not executed" (INIT), from Job-request-table.
    - Fetch a fixed number of records collectively. Default is 3 records.
    - When the target record does not exist, perform polling at regular intervals. Default is 5 seconds interval.
6. `JobRequestPollTask` allocates jobs to thread and executes them based on information of records.
7. `JobRequestPollTask` updates polling status of the Job-request-table to "polled" (POLLED).
    - When number of synchronous execution jobs is achieved, the record which cannot be activated from the fetched records is discarded and the record is fetched again at the time of next polling process.
8. Job assigned to the thread run a job with `JobOperator`.
9. Fetch job execution ID of executed jobs (Job execution id).
10. `JobRequestPollTask` updates the polling status of the Job-request-table to "Executed" (EXECUTED) based on job execution ID fetched at the time of job execution.

> *Supplement of processing sequence*
>
> Spring Batch reference shows that asynchronous execution can be implemented by setting `AsyncTaskExecutor` in `JobLauncher`. However, when this method is adopted, `AsyncTaskExecutor` cannot detect the state wherein job execution cannot be performed. This issue occurs when there is no thread assigned to the job and it is likely to lead to following events.
>
> - Even though the job cannot be executed, it tries to run the job and continues to perform unnecessary operation.
> - The job does not run in the polling sequence but appears to be started randomly on the Job-request-table depending on the time when the thread is free.
>
> The processing sequence described earlier is used in order to avoid this phenomenon.

4.3.2.2. About the table to be polled

Explanation is given about table which performs polling in asynchronous execution (DB polling).

Following database objects are necessary.

- Job-request-table (Required)
- Job sequence (Required for some database products)
    - It is necessary when database does not support auto-numbering of columns.

4.3.2.2.1. Job-request-table structure

PostgreSQL from database products corresponding to TERASOLUNA Batch 5.x is shown. For other databases, refer DDL included in jar of TERASOLUNA Batch 5.x.

batch_job_request (In case of PostgreSQL)

Column Name	Data type	Constraint	Description
job_seq_id	bigserial (Use bigint to define a separate sequence)	NOT NULL PRIMARY KEY	A number to determine the sequence of jobs to be executed at the time of polling. Use auto-numbering function of database.
job_name	varchar(100)	NOT NULL	Job name to be executed. Required parameters for job execution.
job_parameter	varchar(2000)	-	Parameters to be passed to jobs to be executed. Single parameter format is same as synchronous execution, however, when multiple parameters are to be specified, each parameter must be separated by a comma (see below) unlike blank delimiters of synchronous execution. {Parameter name}={parameter value},{Parameter name}={Parameter value}…
job_execution_id	bigint	-	ID to be paid out at the time of job execution. Refer `JobRepository` using ID as a key.
polling_status	varchar(10)	NOT NULL	Polling process status. INIT : Not executed POLLED: Polled EXECUTED : Job executed
create_date	TIMESTAMP	NOT NULL	Date and time when the record of the job request is registered.
update_date	TIMESTAMP	-	Date and time when the record of job request is updated.

DDL is as below.

```sql
CREATE TABLE IF NOT EXISTS batch_job_request (
    job_seq_id bigserial PRIMARY KEY,
    job_name varchar(100) NOT NULL,
    job_parameter varchar(200),
    job_execution_id bigint,
    polling_status varchar(10) NOT NULL,
    create_date timestamp NOT NULL,
    update_date timestamp
);
```

4.3.2.2.2. Job request sequence structure

When the database does not support auto-numbering of database columns, numbering according to sequence is required.
A PostgreSQL from database products corresponding to TERASOLUNA Batch 5.x is shown.
For other databases, refer DDL included in jar of TERASOLUNA Batch 5.x.

DDL is as below.

```sql
CREATE SEQUENCE batch_job_request_seq MAXVALUE 9223372036854775807 NO CYCLE;
```

> Since PostgreSQL supports auto-numbering of columns, job request sequence is not defined in DDL included in jar of TERASOLUNA Batch 5.x.

4.3.2.2.3. Transition pattern of polling status (polling_status)

Transition pattern of polling status is shown in the table below.

Transition pattern list of polling status

Transition source	Transition destination	Description
INIT	INIT	When the number of synchronous executions has been achieved and execution of job is denied, status remains unchanged. It acts as a record for polling at the time of next polling.
INIT	POLLED	Transition is done when the job is successfully started. Status when the job is running.
POLLED	EXECUTED	Transition occurs when job execution is completed.

4.3.2.3. About job running

Running method of job is explained.

Job is run by `start` method of `JobOperator` offered by Spring Batch in Job-request-table polling function of TERASOLUNA Batch 5.x.

With TERASOLUNA Batch 5.x, guidelines explain the restart of jobs started by asynchronous execution (DB polling) from the command line. Hence, `JobOperator` also contains startup methods like `restart` etc besides `start`, however, only `start` method is used.

Arguments of start method

jobName

Set the value registered in `job_name` of Job-request-table.

jobParametrers

Set the value registered in `job_parameters` of Job-request-table.

4.3.2.4. When abnormality is detected in DB polling process.

Explanation is given for when an abnormality is detected in DB polling process.

4.3.2.4.1. Database connection failure

Describe behaviour for the processing performed at the time of failure occurrence.

When records of Job-request-table are fetched

- `JobRequestPollTask` results in an error, however, `JobRequestPollTask` is executed again in next polling.

While changing the polling status from INIT to POLLED

- `JobRequestPollTask` terminates with an error prior to executing job by `JobOperator`. Polling status remains unchanged as INIT.
- In the polling process performed after connection failure recovery, the job becomes a target for execution as there is no change in the Job-request-table and the job is executed at the next polling.

While changing polling status from POLLED to EXECUTED

- `JobRequestPollTask` terminates with an error since the job execution ID cannot be updated in the Job-request-table. Polling status remains unchanged as POLLED.
- It is out of the scope for the polling process to be performed after connection failure recovery and the job at the time of failure is not executed.
- Since a job execution ID cannot be identified from a Job-request-table, final status of the job is determined from log or `JobRepository` and re-execute the job as a process of recovery when required.

> Even if an exception occurs in `JobRequestPollTask`, it is not restored immediately. Reason is given below.
>
> 1. Since `JobRequestPollTask` is started at regular intervals, auto-restoration is possible (not immediate) by delegating the operation to `JobRequestPollTask`.
> 2. It is very rare to be able to recover after retrying immediately at the time of failure occurrence, in addition, it is likely to generate load due to attempt of retry.

4.3.2.4.2. Abnormal termination of asynchronous batch daemon process

When a process of asynchronous batch daemon terminates abnormally, transaction of the job being executed is rolled back implicitly.
State of the polling status is same as status at the time of database connection failure.

4.3.2.5. Stopping DB polling process

Asynchronous batch daemon (`AsyncBatchDaemon`) stops by generation of a file. After confirming that the file has been generated, make the polling process idle, wait as long as possible to job being started and then stop the process.

4.3.2.6. About application configuration specific to asynchronous execution

Configuration specific to asynchronous execution is explained.

4.3.2.6.1. ApplicationContext configuration

Asynchronous batch daemon reads `async-batch-daemon.xml` dedicated to asynchronous execution as ApplicationContext. Configuration below is added besides `launch-context.xml` used in synchronous execution as well.

Asynchronous execution settings
: A Bean necessary for asynchronous execution like `JobRequestPollTask` etc. is defined.

Job registration settings
: Job executed as an asynchronous execution registers by `org.springframework.batch.core.configuration.support.AutomaticJobRegistrar`. Context for each job is modularized by using `AutomaticJobRegistrar`. When modularization is done, it does not pose an issue even of Bean ID used between the jobs is duplicated.

> *What is modularization*
>
> Modularization is a hierarchical structure of "Common definition - Definition of each job" and the Bean defined in each job belongs to an independent context between jobs. If a reference to a Bean which is not defined in each job definition exists, it refers to a Bean defined in common definition.

4.3.2.6.2. Bean definition structure

Bean definition of a job can be same as Bean definition of synchronous execution. However, following precautions must be taken.

- When job is to be registered by `AutomaticJobRegistrar`, Bean ID of the job is an identifier, and hence should not be duplicated.
- It is also desirable to not to duplicate Bean ID of step.
  - Only the job ID should be uniquely designed by designing naming rules of Bean ID as `{Job ID}.{Step ID}`.

Import of job-base-context.xml in the Bean definition of job varies for synchronous and asynchronous execution.

* In synchronous execution, launch-context.xml is imported from job-base-context.xml.
* In asynchronous execution, launch-context.xml is not imported from job-base-context.xml. Alternatively, import launch-context.xml from async-batch-daemon.xml which AsyncBatchDaemon loads.

This is because various Beans required for starting Spring Batch need not be instantiated for each job. Only one bean should be created in common definition (async-batch-daemon.xml) which acts as a parent for each job, from various Beans required for starting Spring Batch.

4.3.3. How to use

4.3.3.1. Various settings

4.3.3.1.1. Settings for polling process

Use batch-application.properties for settings required for asynchronous execution.

batch-application.properties

```
#(1)
# Admin DataSource settings.
admin.jdbc.driver=org.postgresql.Driver
admin.jdbc.url=jdbc:postgresql://localhost:5432/postgres
admin.jdbc.username=postgres
admin.jdbc.password=postgres

# TERASOLUNA AsyncBatchDaemon settings.
# (2)
async-batch-daemon.schema.script=classpath:org/terasoluna/batch/async/db/schema-
postgresql.sql
# (3)
async-batch-daemon.job-concurrency-num=3
# (4)
async-batch-daemon.polling-interval=5000
# (5)
async-batch-daemon.polling-initial-delay=1000
# (6)
async-batch-daemon.polling-stop-file-path=/tmp/end-async-batch-daemon
```

Setup details item list

Sr. No.	Description
(1)	Connection settings for database wherein Job-request-table is stored. JobRepository settings are used by default.

Sr. No.	Description
(2)	A path for DDL which defines Job-request-table. It is auto-generated when Job-request-table does not exist at the time of starting asynchronous batch daemon. This is primarily a test function and execution can be set by `data-source.initialize.enabled` of+ `batch-application.properties`. For detailed definition, refer `<jdbc:initialize-database>>` in `async-batch-daemon.xml`.
(3)	Setting for records which are fetched collectively at the time of polling. This setup value is also used as a synchronous parallel number.
(4)	Polling cycle settings. Unit is milliseconds.
(5)	Polling initial start delay time settings. Unit is milliseconds.
(6)	Exit file path settings.

Changing setup value using environment variable

Setup value of `batch-application.properties` can be changed by defining environment variable with same name.
When an environment variable is set, it is prioritized over property value.
This happens due to Bean definition below.

Settings for launch-context.xml

```
<context:property-placeholder location="classpath:batch-application.properties"
        system-properties-mode="OVERRIDE"
        ignore-resource-not-found="false"
        ignore-unresolvable="true"
        order="1"/>
```

For details, refer How to define a property file of TERASOLUNA Server 5.x Development Guideline.

4.3.3.1.2. Job settings

Job to be executed asynchronously is set in `automaticJobRegistrar` of `async-batch-daemon.xml`. Default settings are shown below.

async-batch-daemon.xml

```xml
<bean id="automaticJobRegistrar"

class="org.springframework.batch.core.configuration.support.AutomaticJobRegistrar">
    <property name="applicationContextFactories">
        <bean
class="org.springframework.batch.core.configuration.support.ClasspathXmlApplicationCon
textsFactoryBean">
            <property name="resources">
                <list>
                    <value>classpath:/META-INF/jobs/**/*.xml</value>  <!-- (1) -->
                </list>
            </property>
        </bean>
    </property>
    <property name="jobLoader">
        <bean
class="org.springframework.batch.core.configuration.support.DefaultJobLoader"
              p:jobRegistry-ref="jobRegistry" />
    </property>
</bean>
```

Setting details item list

Sr.No.	Description
(1)	A path for Bean definition of a job executed asynchronously.

About registered jobs

For registering jobs, jobs which are designed and implemented on the premise that they are executed asynchronously should be specified. If the jobs which are not supposed to be executed asynchronously are included, exceptions may occur due to unintended references at the time of job registration.

Example of Narrowing down

```
<bean id="automaticJobRegistrar"

class="org.springframework.batch.core.configuration.support.AutomaticJ
obRegistrar">
    <property name="applicationContextFactories">
        <bean
class="org.springframework.batch.core.configuration.support.ClasspathX
mlApplicationContextsFactoryBean">
            <property name="resources">
                <list>
                    <!-- For the async directory and below -->
                    <value>classpath:/META-
INF/jobs/aysnc/**/*.xml</value>
                    <!-- For a specific job -->
                    <value>classpath:/META-
INF/jobs/CASE100/SpecialJob.xml</value>
                </list>
            </property>
        </bean>
    </property>
    <property name="jobLoader">
        <bean
class="org.springframework.batch.core.configuration.support.DefaultJob
Loader"
            p:jobRegistry-ref="jobRegistry" />
    </property>
</bean>
```

Input value verification for job parameters

`JobPollingTask` does not validate the records obtained from Job-request-table. Hence, the job name and job parameter must be verified for the table registration.
If the job name is incorrect, job is not detected even if it has started and an exception occurs.
If the job parameter is incorrect, an erroneous operation is performed even if the job has started.
Only job parameters can be verified once the job is started. For verification of job parameters, refer Validity verification of parameters.

> *Job design considerations*
>
> As a characteristic of asynchronous execution (DB polling), the same job can be executed in parallel. It is necessary to prevent the same job to create an impact when the jobs are run in parallel.

4.3.3.2. From start to end of asynchronous execution

Start and end of asynchronous batch daemon and how to register in Job-request-table are explained.

4.3.3.2.1. Start of asynchronous batch daemon

Start `AsyncBatchDaemon` offered by TERASOLUNA Batch 5.x.

Start of AsyncBatchDaemon

```
# Start AsyncBatchDaemon
$ java -cp dependency/* org.terasoluna.batch.async.db.AsyncBatchDaemon
```

In this case, `META-INF/spring/async-batch-daemon.xml` is read and various Beans are generated.

Further, when `async-batch-daemon.xml` customised separately, it is implemented by specifying first argument and starting `AsyncBatchDaemon`.
Bean definition file specified in the argument must be specified as a relative path from the class path.
Note that, the second and subsequent arguments are ignored.

When customised META-INF/spring/customized-async-batch-daemon.xml is used,

```
# Start AsyncBatchDaemon
$ java -cp dependency/* org.terasoluna.batch.async.db.AsyncBatchDaemon \
    META-INF/spring/customized-async-batch-daemon.xml
```

Customisation of `async-batch-daemon.xml` can be modified directly by changing some of the settings.
However, when significant changes are added or when multiple settings are managed in Multiple runningsdescribed later, it is easier to manage and create separate files.
It should be choosed according to user's situation..

> It is assumed that jar expressions necessary for execution are stored under dependency.

4.3.3.2.2. Job request

Register in Job-request-table by issuing SQL of INSERT statement.

In case of PostgreSQL

```
INSERT INTO batch_job_request(job_name,job_parameter,polling_status,create_date)
VALUES ('JOB01', 'param1=dummy,param2=100', 'INIT', current_timestamp);
```

4.3.3.2.3. Stopping asynchronous batch daemon

Keep exit file set in `batch-application.properties`.

```
$ touch /tmp/end-async-batch-daemon
```

> *When the exit file exists prior to starting asynchronous batch daemon*
>
> When the exit file exists prior to starting asynchronous batch daemon, asynchronous batch daemon terminates immediately. Asynchronous batch daemon must be started in the absence of exit file.

4.3.3.3. Confirm job status

Job status management is performed with `JobRepository` offered by Spring Batch and the job status is not managed in the Job-request-table. Job-request-table has a column of `job_execution_id` and job status corresponding to individual requests can be confirmed by the value stored in this column. Here, a simple example wherein SQL is issued directly and job status is confirmed is shown. For details of job status confirmation, refer Status confirmation.

In case of PostgreSQL

```
SELECT job_execution_id FROM batch_job_request WHERE job_seq_id = 1;

job_execution_id
----------------
              2
(1 row)

SELECT * FROM batch_job_execution WHERE job_execution_id = 2;

job_execution_id | version | job_instance_id |       create_time       |
start_time         |        end_time         |  status   | exit_code | exit_message |
ocation
------------------+---------+-----------------+-------------------------+
-------------------------+-------------------------+-----------+-----------+
--------------+-
--------
              2 |       2 |               2 | 2017-02-06 20:54:02.263 | 2017-02-06 20
:54:02.295 | 2017-02-06 20:54:02.428 | COMPLETED | COMPLETED |              |
(1 row)
```

4.3.3.4. Recovery after a job is terminated abnormally

For basic points related to recovery of a job which is terminated abnormally, refer Re-execution of process. Here, points specific to asynchronous execution are explained.

4.3.3.4.1. Re-run

Job which is terminated abnormally is re-run by inserting it as a separate record in Job-request-table.

4.3.3.4.2. Restart

When the job which is terminated abnormally is to be restarted, it is executed as a synchronous execution job from the command line. The reason for executing from the command line is "since it is difficult to determine whether the restart is intended or whether it is an unintended duplicate execution resulting in chaotic operation."
For restart methods, refer Job restart.

4.3.3.4.3. Termination

1. When the process has not terminated even after exceeding the expected processing time, attempt terminating the operation from the command line. For methods of termination, refer Job stop.
2. When the termination is not accepted even from a command line, asynchronous batch daemon should be terminated by Stopping asynchronous batch daemon.
3. If even an asynchronous batch daemon cannot be terminated, process of asynchronous batch daemon should be forcibly terminated.

> Adequate care should be taken not to impact other jobs when an asynchronous batch daemon is being terminated.

4.3.3.5. About environment deployment

Building and deploying job is same as a synchronous execution. However, it is important to narrow down the jobs which are executed asynchronously as shown in Job settings.

4.3.3.6. Evacuation of cumulative data

If you run an asynchronous batch daemon for a long time, a huge amount of data is accumulated in JobRepository and the Job-request-table. It is necessary to clear this cumulative data for the following reasons.

- Performance degradation when data is retrieved or updated for a large quantity of data
- Duplication of ID due to circulation of ID numbering sequence

For evacuation of table data and resetting a sequence, refer manual for the database to be used.

List of tables and sequences for evacuation is shown below.

List for evacuation

Table/Sequence	Framework offered
batch_job_request	TERASOLUNA Batch 5.x
batch_job_request_seq	
batch_job_instance	Spring Batch
batch_job_exeution	
batch_job_exeution_params	
batch_job_exeution_context	
batch_step_exeution	
batch_step_exeution_context	
batch_job_seq	
batch_job_execution_seq	
batch_step_execution_seq	

> *Auto-numbering column sequence*
>
> Since a sequence is created automatically for an auto-numbering column, remember to include this sequence while evacuating data.

> *About database specific specifications*
>
> Note that Oracle uses database-specific data types in some cases, such as using CLOB for data types.

4.3.4. How to extend

4.3.4.1. Customising Job-request-table

Job-request-table can be customised by adding a column in order to change extraction conditions of fetched records. However, only `BatchJobRequest` can be passed as an item while issuing SQL from `JobRequestPollTask`.

Extension procedure by customising the Job-request-table is shown below.

1. Customising Job-request-table
2. Creating an extension interface of `BatchJobRequestMapper` interface
3. Defining SQLMap which uses customised table
4. Modifying Bean definition of `async-batch-daemon.xml`

Examples of customization are as below.

- Example of controlling job execution sequence by priority column
- Distributed processing by multiple processes using a group ID

Hereafter, the extension procedure will be described for these two examples.

4.3.4.1.1. Example of controlling job execution sequence by priority column

1. Customising Job-request-table

Add a priority column (priority) in Job-request-table.

Adding a priority column (In case of PostgreSQL)

```
CREATE TABLE IF NOT EXISTS batch_job_request (
    job_seq_id bigserial PRIMARY KEY,
    job_name varchar(100) NOT NULL,
    job_parameter varchar(200),
    priority int NOT NULL,
    job_execution_id bigint,
    polling_status varchar(10) NOT NULL,
    create_date timestamp NOT NULL,
    update_date timestamp
);
```

2. Create extension interface of `BatchJobRequestMapper` interface

An interface which extends `` `BatchJobRequestMapper `` interface is created.

Extension interface

```
// (1)
public interface CustomizedBatchJobRequestMapper extends BatchJobRequestMapper {
    // (2)
}
```

Extension points

Sr. No.	Description
(1)	Extend `BatchJobRequestMapper`.
(2)	Do not add a method.

3. Definition of SQLMap which use a customised table

Define SQL in SQLMap with priority as a condition for order.

SQLMap definition (CustomizedBatchJobRequestMapper.xml)

```xml
<!-- (1) -->
<mapper
namespace="org.terasoluna.batch.extend.repository.CustomizedBatchJobRequestMapper">

    <select id="find" resultType=
"org.terasoluna.batch.async.db.model.BatchJobRequest">
        SELECT
            job_seq_id AS jobSeqId,
            job_name AS jobName,
            job_parameter AS jobParameter,
            job_execution_id AS jobExecutionId,
            polling_status AS pollingStatus,
            create_date AS createDate,
            update_date AS updateDate
        FROM
            batch_job_request
        WHERE
            polling_status = 'INIT'
        ORDER BY
            priority ASC,   <!--(2) -->
            job_seq_id ASC
        LIMIT #{pollingRowLimit}
    </select>

    <!-- (3) -->
    <update id="updateStatus">
        UPDATE
            batch_job_request
        SET
            polling_status = #{batchJobRequest.pollingStatus},
            job_execution_id = #{batchJobRequest.jobExecutionId},
            update_date = #{batchJobRequest.updateDate}
        WHERE
            job_seq_id = #{batchJobRequest.jobSeqId}
        AND
            polling_status = #{pollingStatus}
    </update>

</mapper>
```

Extension points

Sr. No.	Description
(1)	Set extended interface of `BatchJobRequestMapper`in namespace by FQCN.
(2)	Add priority to ORDER clause.
(3)	Do not change updated SQL.

4. Modifying Bean definition of `async-batch-daemon.xml`

Set extended interface created in (2) in `batchJobRequestMapper`.

async-batch-daemon.xml

```
 <!--(1) -->
<bean id="batchJobRequestMapper"
      class="org.mybatis.spring.mapper.MapperFactoryBean"

p:mapperInterface="org.terasoluna.batch.extend.repository.CustomizedBatchJobRequestMap
per"
      p:sqlSessionFactory-ref="adminSqlSessionFactory" />
```

Extension points

Sr. No.	Description
(1)	Set extended interface of `BatchJobRequestMapper` in `mapperInterface` property by FQCN.

4.3.4.1.2. Distributed processing by multiple processes using a group ID

Specify group ID by using environment variable while starting `AsyncBatchDaemon` and narrow down the target job.

1. Customizing Job-request-table

Add group ID column (group_id) to Job-request-table.

Adding group ID column (In case of PostgreSQL)

```
CREATE TABLE IF NOT EXISTS batch_job_request (
    job_seq_id bigserial PRIMARY KEY,
    job_name varchar(100) NOT NULL,
    job_parameter varchar(200),
    group_id varchar(10) NOT NULL,
    job_execution_id bigint,
    polling_status varchar(10) NOT NULL,
    create_date timestamp NOT NULL,
    update_date timestamp
);
```

2. Creating extended interface of `BatchJobRequestMapper` interface

   - Same as Example of controlling job execution sequence by priority column

3. Definition of SQLMap which use customised table

Define SQL in SQLMap with the group ID as the extraction condition.

SQLMap definition (CustomizedBatchJobRequestMapper.xml)

```
<!-- (1) -->
<mapper
namespace="org.terasoluna.batch.extend.repository.CustomizedBatchJobRequestMapper">

    <select id="find" resultType=
"org.terasoluna.batch.async.db.model.BatchJobRequest">
        SELECT
            job_seq_id AS jobSeqId,
            job_name AS jobName,
            job_parameter AS jobParameter,
            job_execution_id AS jobExecutionId,
            polling_status AS pollingStatus,
            create_date AS createDate,
            update_date AS updateDate
        FROM
            batch_job_request
        WHERE
            polling_status = 'INIT'
        AND
            group_id = #{groupId}  <!--(2) -->
        ORDER BY
            job_seq_id ASC
        LIMIT #{pollingRowLimit}
    </select>

    <!-- ommited -->
</mapper>
```

Extension points

Sr. No.	Description
(1)	Set extended interface of `BatchJobRequestMapper` in namespace by FQCN.
(2)	Add groupId to extraction conditions.

4. Modifying Bean definition of `async-batch-daemon.xml`

Set extended interface created in (2) in `batchJobRequestMapper` and set the group ID assigned by environment variable in `jobRequestPollTask` as a query parameter.

async-batch-daemon.xml

```
 <!--(1) -->
<bean id="batchJobRequestMapper"
      class="org.mybatis.spring.mapper.MapperFactoryBean"

p:mapperInterface="org.terasoluna.batch.extend.repository.CustomizedBatchJobRequestMap
per"
      p:sqlSessionFactory-ref="adminSqlSessionFactory" />

    <bean id="jobRequestPollTask"
          class="org.terasoluna.batch.async.db.JobRequestPollTask"
          c:transactionManager-ref="adminTransactionManager"
          c:jobOperator-ref="jobOperator"
          c:batchJobRequestMapper-ref="batchJobRequestMapper"
          c:daemonTaskExecutor-ref="daemonTaskExecutor"
          c:automaticJobRegistrar-ref="automaticJobRegistrar"
          p:optionalPollingQueryParams-ref="pollingQueryParam" /> <!-- (2) -->

   <bean id="pollingQueryParam"
         class="org.springframework.beans.factory.config.MapFactoryBean">
        <property name="sourceMap">
            <map>
                <entry key="groupId" value="${GROUP_ID}"/>  <!-- (3) -->
            </map>
        </property>
   </bean>
```

Extension points

Sr. No.	Description
(1)	Set extended interface of `BatchJobRequestMapper` in `mapperInterface` property by FQCN..
(2)	Set Map defined in (3), in `optionalPollingQueryParams` property of `JobRequestPollTask`.
(3)	Set group ID assigned by environment variable (GROUP_ID) in group ID (groupId) of query parameter.

5. Set group ID in environment variable and start `AsyncBatchDaemon`.

Starting AsyncBatchDaemon

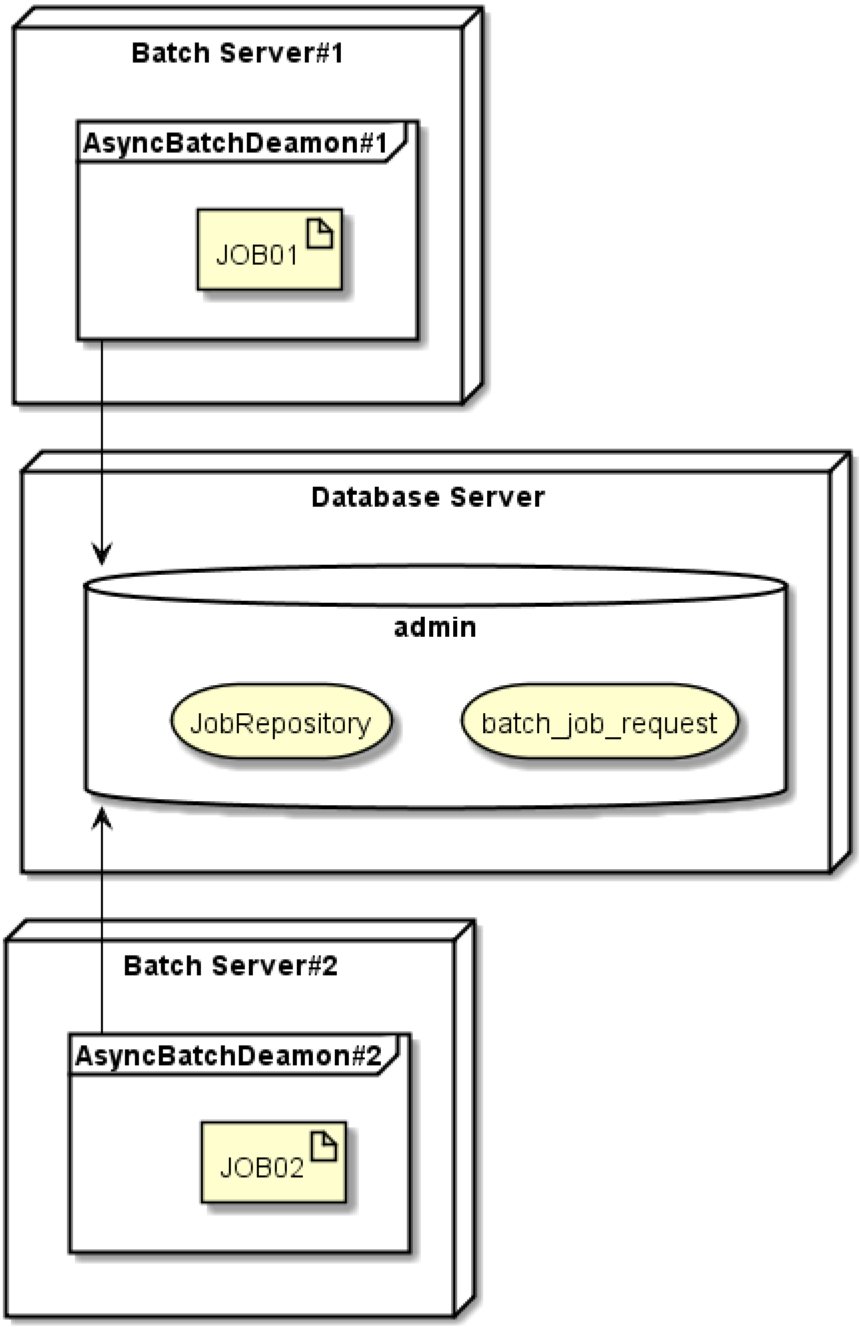
```
# Set environment variables
$ export GROUP_ID=G1

# Start AsyncBatchDaemon
$ java -cp dependency/* org.terasoluna.batch.async.db.AsyncBatchDaemon
```

4.3.4.2. Multiple runnings

Asynchronous batch daemon is run on multiple servers for the following purposes.

- Enhanced availability
  - It suffices if an asynchronous batch job can be executed on one of the servers, and it is to eliminate the situation that the job can not be run.
- Enhanced performance
  - When batch processing load is to be distributed across multiple servers
- Effective use of resources
  - When a specific job is to be distributed on a server with optimal resources when a variation is observed in the server performance
    - Equivalent to dividing a job node based on group ID shown in Customising Job-request-table

An operational design must be adopted considering whether it can be used based on the viewpoints given above.



Schematic diagram for multiple starts

> *When multiple asynchronous batch daemons fetch identical job request records*
>
> Since `JobRequestPollTask` performs exclusive control using optimistic locking, it can execute the job of the record fetched by asynchronous batch daemon which can update the polling status from INIT to POLLED. Other exclusive asynchronous batch daemons fetch next job request record.

4.3.5. Appendix

4.3.5.1. About modularization of job definition

Although it is briefly explained in ApplicationContext configuration, following events can be avoided by using `AutomaticJobRegistrar`.

* When same BeanID (BeanName) is used, Bean is overwritten and the job shows unintended behaviour.
  * Accordingly, there is a high risk of occurrence of unintended errors.
* Naming should be performed to make all Bean IDs in the job unique, to avoid these errors.
  * As the number of job increases, it becomes difficult to manage the same resulting in unnecessary troubles.

An event when `AutomaticJobRegistrar` is not used is explained. Since the contents explained here pose the issues given above, it is not used in asynchronous execution.

Job1.xml

```
<!-- Reader -->
<!-- (1) -->
<bean id="reader" class="org.mybatis.spring.batch.MyBatisCursorItemReader"
      p:queryId="jp.terasoluna.batch.job.repository.EmployeeRepositoy.findAll"
      p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

<!-- Writer -->
<!-- (2) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:resource="file:#{jobParameters[basedir]}/input/employee.csv">
  <property name="lineAggregator">
    <bean
class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
      <property name="fieldExtractor">
        <bean
class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
              p:names="invoiceNo,salesDate,productId,customerId,quant,price"/>
      </property>
    </bean>
  </property>
</bean>


<!-- Job -->
<batch:job id="job1" job-repository="jobRepository">
  <batch:step id="job1.step">
    <batch:tasklet transaction-manager="transactionManager">
      <batch:chunk reader="reader" writer="writer" commit-interval="100" />
    </batch:tasklet>
  </batch:step>
</batch:job>
```

Job2.xml

```xml
<!-- Reader -->
<!-- (3) -->
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[basedir]}/input/invoice.csv">
  <property name="lineMapper">
    <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">
        <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
              p:names="invoiceNo,salesDate,productId,customerId,quant,price"/>
      </property>
      <property name="fieldSetMapper" ref="invoiceFieldSetMapper"/>
    </bean>
  </property>
</bean>

<!-- Writer -->
<!-- (4) -->
<bean id="writer" class="org.mybatis.spring.batch.MyBatisBatchItemWriter"
      p:statementId="jp.terasoluna.batch.job.repository.InvoiceRepository.create"
      p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

<!-- Job -->
<batch:job id="job2" job-repository="jobRepository">
  <batch:step id="job2.step">
    <batch:tasklet transaction-manager="transactionManager">
      <batch:chunk reader="reader" writer="writer" commit-interval="100" />
    </batch:tasklet>
  </batch:step>
</batch:job>
```

Definition wherein BeanId is overwritten

```
<bean id="automaticJobRegistrar"

class="org.springframework.batch.core.configuration.support.AutomaticJobRegistrar">
    <property name="applicationContextFactories">
        <bean
class="org.springframework.batch.core.configuration.support.ClasspathXmlApplicationCon
textsFactoryBean">
            <property name="resources">
                <list>
                    <value>classpath:/META-INF/jobs/other/async/*.xml</value>  <!--
(5) -->
                </list>
            </property>
        </bean>
    </property>
    <property name="jobLoader">
        <bean
class="org.springframework.batch.core.configuration.support.DefaultJobLoader"
              p:jobRegistry-ref="jobRegistry"/>
    </property>
</bean>

<bean
class="org.springframework.batch.core.configuration.support.JobRegistryBeanPostProcess
or"
    p:jobRegistry-ref="jobRegistry" />

<import resource="classpath:/META-INF/jobs/async/*.xml" />   <!-- (6) -->
```

List of setup points

Sr. No.	Description
(1)	In Job1, ItemReader which reads from the database is defined by a Bean ID - `reader`.
(2)	In Job1, ItemWriter which writes in a file is defined by a Bean ID - `writer`.
(3)	In Job2, ItemReader which reads from the file is defined by a Bean ID - `reader`.
(4)	In Job2, ItemWriter which writes to a database is defined by a Bean ID - `writer`.
(5)	`AutomaticJobRegistrar` is set so as to read job definitions other than target jobs.
(6)	Use import of Spring and enable reading of target job definition.

In this case, if Job1.xml and Job2.xml are read in the sequence, reader and writer to be defined by Job1.xml will be overwritten by Job2.xml definition.
As a result, when Job1 is executed, reader and writer of Job2 are used and intended processing cannot be performed.

4.4. Asynchronous execution (Web container)

4.4.1. Overview

A method to execute the job asynchronously in Web container is explained.

The usage method of this function is same in the chunk model as well as tasklet model.

What is Asynchronous execution of jobs by Web container

Web application that contains a job is deployed in a Web container and the job is executed based on information of sent request.
Since one thread is allocated for each job execution and operation is run in parallel, it can be executed independent of processes for other jobs and requests.

Function offered

TERASOLUNA Batch 5.x does not offer implementation for asynchronous execution (Web container).
Only methods of implementation will be provided in this guideline.
This is because the start timing of the Web application is various such as HTTP / SOAP / MQ, and hence it is determined that the implementation should be appropriately done by the user.

Usage premise

- A Web container is required besides the application.
- Besides implementation of job, required Web application and client are separately implemented according to the operation requirements.
- Execution status and results of the job is entrusted to `JobRepository`. Further, a permanently residing database is used instead of in-memory database to enable execution status and results of job to be referred from `JobRepository` even after stopping Web container.

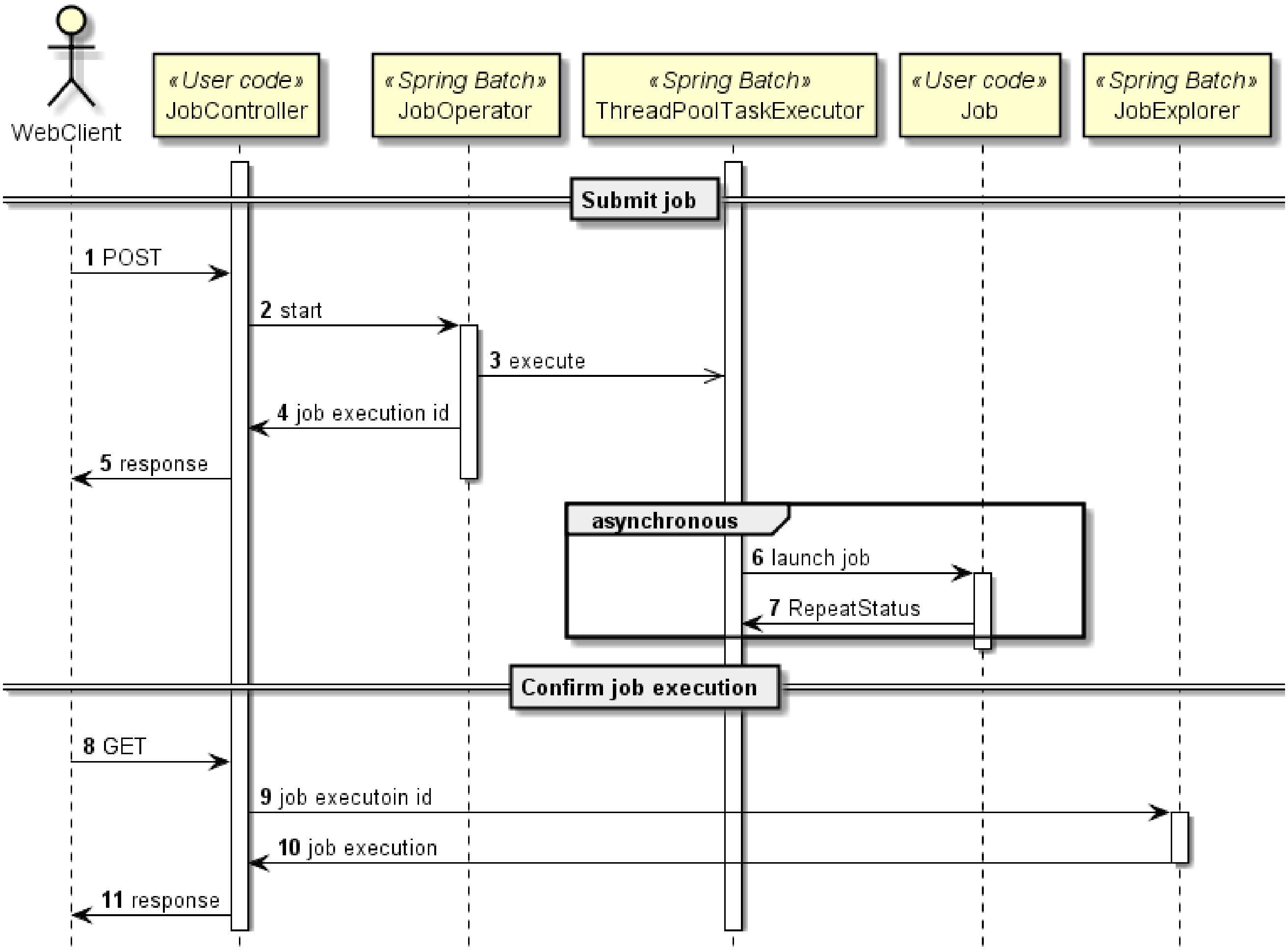
Usage scene

It is same as Asynchronous execution (DB polling) - Overview.

> *Difference with asynchronous execution (DB polling)*
>
> On the architecture front, immediacy at the time of asynchronous execution and presence or absence of request management table are different.
> Asynchronous execution (DB polling) performs asynchronous execution of multiple jobs registered in the request management table.
> On the other hand, this function does not require request management table and accepts asynchronous execution on the Web container instead.
> It is suitable for a short batch which requires immediacy till the start of the operation in order to execute the operation immediately by sending a Web request.

4.4.2. Architecture

Asynchronous jobs by using this method are operated as applications (war) deployed on the Web container, however, the job itself runs asynchronously (another thread) from the request processing of Web container.



Process sequence diagram of asynchronous execution (Web container)

Running a job

1. Web client requests Web container to execute the job.
2. `JobController` asks `JobOperator` of Spring Batch to start the execution of the job.
3. Execute the job asynchronously by using `ThreadPoolTaskExecutor`.
4. Return a job execution ID (`job execution id`) for uniquely identifying an executed target job.
5. `JobController` returns a response including job execution ID for the Web client.
6. Execute target job.
    - Job results are reflected in `JobRepository`.
7. `Job` returns execution results. It cannot be notified directly to the client.

Confirm job execution results

8. Web client sends job execution ID and `JobController` to Web container.
9. `JobController` asks `JobExplorer` for execution results of job by using a job execution ID.
10. `JobExplorer` returns job execution results.
11. `JobController` returns a response for Web client.

* Set Job execution ID in the response.

After receiving a request using Web container, operation is synchronised with the request processing till job execution ID payout, however subsequent job execution is performed asynchronously in a thread pool different from that of Web container.+ As long as the query is not sent again by sending a request, it signifies that execution status of asynchronous job cannot be detected on Web client side.

Hence, the request should be sent once at the time of "running a job" on the Web client side during one job execution. When "confirmation of results" is necessary, request must be sent once again to the Web container.
Abnormality detection which looks different from first "running a job" will be explained later in About detection of abnormality occurrence at the time of running a job.

> Job execution status can be checked by referring direct RDBMS, by using `JobRepository` and `JobExplorer`. For details of the function which refer to job execution status and results, refer Job management.

> *About handling job execution ID (job execution id)*
>
> Job execution ID generates a different sequence value for each job even though job and job parameters are identical.
> Job execution ID accepted by sending a request is persisted in external RDBMS by `JobRepository`.
> However, when this ID is lost due to failure of Web client, specifying or tracking job execution status becomes difficult.
> Hence, adequate preparations must be made on Web client side to cope with loss of job execution ID like logging the job execution ID returned as a response.

4.4.2.1. About detection of abnormality occurrence at the time of running a job

After sending a job run request from Web client, abnormality detection appearance varies along with job execution ID payout.

* Abnormality can be detected immediately by the response at the time of running a job
  * Job to be activated does not exist.
  * Invalid job parameter format.
* After running a job, queries regarding job execution status and results for Web container are necessary
  * Job execution status
  * Job start failure due to depletion of thread pool used in asynchronous job execution

"Job running error" can be detected as an exception occurring in Spring MVC controller. Since the explanation is omitted here, refer Implementation of exception handling of TERASOLUNA Server 5.x Development Guideline described separately.

Further, input check of the request used as a job parameter is performed in the Spring MVC controller as required.
For basic implementation methods, refer Input check of TERASOLUNA Server 5.x Development Guideline.

Job start failure occurring due to depletion of thread pool cannot be caught at the time of running a job.

Job start failure due to depletion of thread pool is not generated from `JobOperator`, hence it must be checked separately. One of the methods of confirmation include using `JobExplorer` while checking execution status of job and checking whether the following conditions are satisfied.

- Status is `FAILED`
- Exception stack trace of `org.springframework.core.task.TaskRejectedException` is recorded in `jobExecution.getExitStatus().getExitDescription()`.

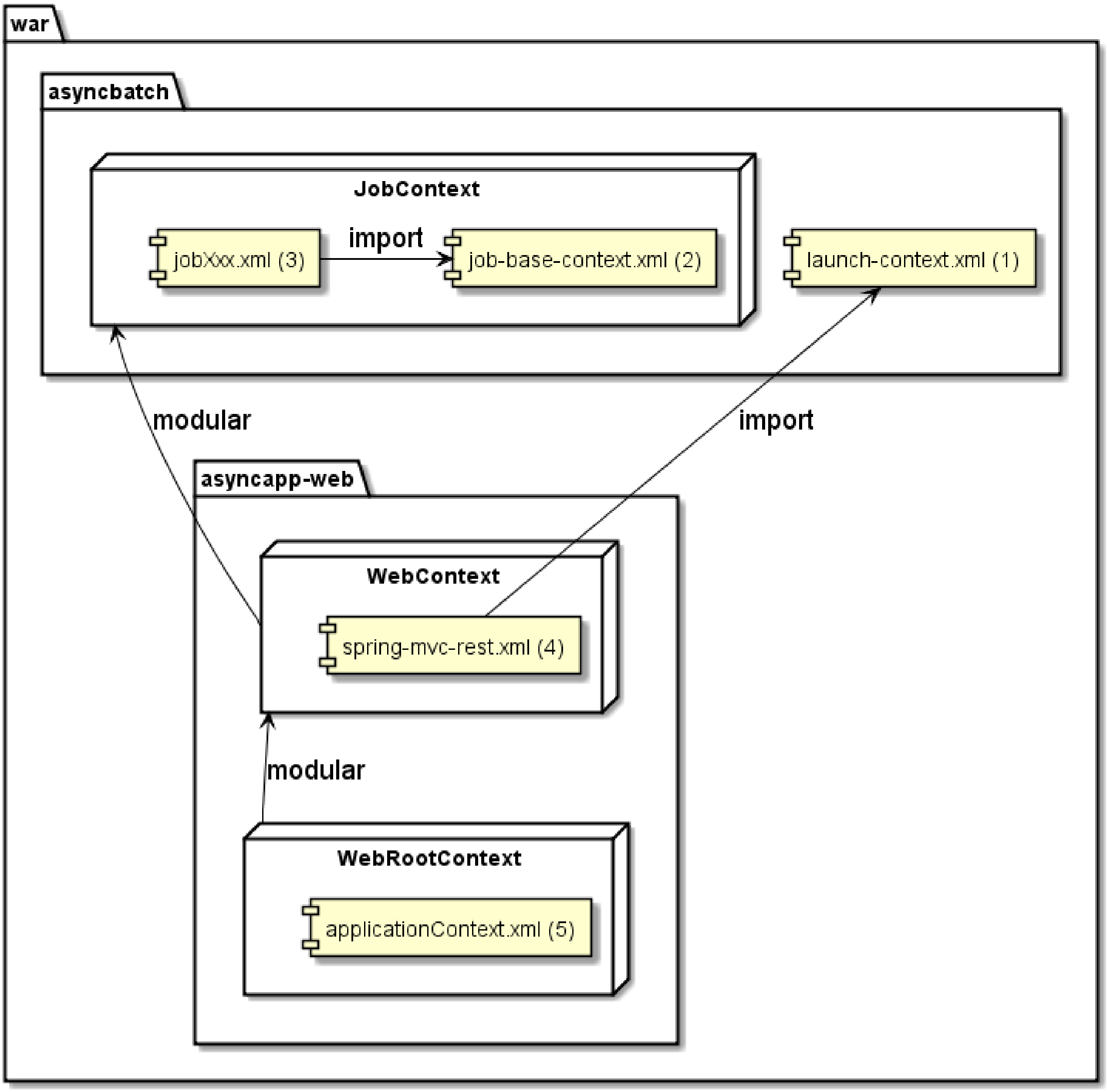
4.4.2.2. Application configuration of asynchronous execution (Web container)

The function is same as Asynchronous execution (DB polling) and use `async` and `AutomaticJobRegistrar` of Spring profile as a configuration specific to asynchronous execution.

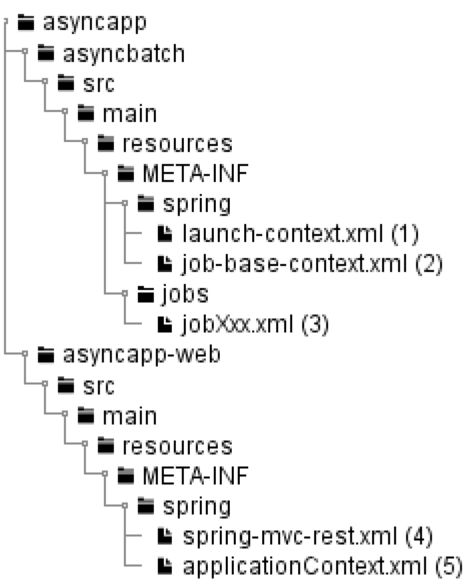
On the other hand, prior knowledge and some specific settings are required in order to use these functions asynchronously (Web container). Refer ApplicationContext configuration.
For configuration methods of basic `async` profile and `AutomaticJobRegistrar`, How to implement applications using asynchronous execution (Web container) will be described later.

4.4.2.2.1. ApplicationContext configuration

As described above, multiple application modules are included as application configuration of asynchronous execution (Web container).
It is necessary to understand respective application contexts, types of Bean definitions and their relationships.



ApplicationContext configuration



Bean definition file configuration

`ApplicationContext` of batch application is incorporated in the context, in `ApplicationContext` during asynchronous execution (Web container).
Individual job contexts are modularised from Web context using `AutomaticJobRegistrar` and it acts as a sub-context of Web context.

Bean definition file which constitute respective contexts are explained.

List of Bean definition files

Sr. No.	Description
(1)	Common Bean definition file. It acts as a parent context in the application and is uniquely shared among jobs acting as sub-contexts.
(2)	Bean definition file which is always imported from job Bean definitions. If Spring profile is `async` specified at the time of asynchronous execution, `launch-context.xml` of (1) is not read.
(3)	Bean definition file created for each job. It is modularized by `AutomaticJobRegistrar` and are used as respective independent sub-contexts in the application.
(4)	It is read from `DispatcherServlet`. Define the Beans unique to asynchronous execution such as `AutomaticJobRegistrar` which performs modularization of job Bean definition and `taskExecutor` which is a thread pool used in asynchronous and parallel execution of jobs. Further, in asynchronous execution, `launch-context.xml` of (1) is imported directly and uniquely shared as parent contexts.
(5)	It acts as a parent context shared within the Web application by using `ContextLoaderListener`.

4.4.3. How to use

Here, explanation is given using TERASOLUNA Server Framework for Java (5.x), as an implementation example of Web application.
Kindly remember that only explanation is offered and TERASOLUNA Server 5.x is not a necessary requirement of asynchronous execution (Web container).

4.4.3.1. Overview of implementation of application by asynchronous execution (Web container)

Explanation is given based on following configuration.

- Web application project and batch application project are independent and a batch application is referred from a web application.
  - war file generated from Web application project contains jar file generated from batch application project

Implementation of asynchronous execution is performed in accordance with Architecture wherein

Spring MVC controller in the Web application starts the job by using `JobOperator`.

> *About isolation of Web/batch application project*
>
> Final deliverable of application build is a war file of Web application, however, a development project should be implemented by separating Web/batch applications.
> Since it is a library which can be operated by a batch application alone, it helps in identifying work boundary and library dependency besides making the development project testing easier to implement.

Web/batch development is explained now assuming the use of 2 components below.

- Batch application project by TERASOLUNA Batch 5.x
- Web application project by TERASOLUNA Server 5.x

For how to create a batch application project and how to implement a basic job, refer How to create a project and Asynchronous execution (DB polling).

Here, we will focus on starting a batch application from a Web application.

Here, explanation is given by creating a batch application project, by using Maven archetype:generate.

How to create a job project

Name	Value
groupId	org.terasoluna.batch.sample
archetypeId	asyncbatch
version	1.0-SNAPSHOT
package	org.terasoluna.batch.sample

A job registered from the beginning for a blank project is used for convenience of explanation.

Job used for explanation

Name	Description
Job name	job01
Job parameter	param1=value1

> *Precautions for asynchronous execution (Web container) job design*
>
> Individual jobs are completed in a short period of time as a characteristic of asynchronous execution (Web container) and are operated in a stateless manner on the Web container.
> Further, it is necessary to build a job definition with only a single step to avoid complexity and it is desirable not to define flow branching by using exit codes of step and parallel/multiple processing.

Create a Web application as a state wherein a jar file including a job implementation can be created.

Implementation of Web application

How to implement a Web application is explained by using a blank project offered by TERASOLUNA Server 5.x. For details, refer TERASOLUNA Server 5.x Development Guideline [Creating a development project for Web application](#).

Here, similar to asynchronous execution application project, explanation is given below creating with the following names.

How to create a Web container project

Name	Value
groupId	org.terasoluna.batch.sample
archetypeId	asyncapp
version	1.0-SNAPSHOT
package	org.terasoluna.batch.sample

> *About naming of groupId*
>
> Although naming a project is optional, when a batch application as a Maven multiproject is considered as a sub-module, it is easy to manage if `groupId is integrated.`
> `Here, groupId of both is considered as org.terasoluna.batch.sample`\`.

4.4.3.2. Various settings

Include batch application as a part of Web application

Edit pom.xml and include batch application as a part of Web application.

> Batch application is registered in NEXUS or Maven local repository as `jar` This process is not required while setting a separate project from that of Web application.
> However, target to be built by Maven is a separate project and it will not be reflected while building the web application even if the batch application is modified.
> It should be registered in the same repository in order to reflect the modification of batch application in the Web application.

```
asyncapp
├─ asyncbatch
├─ asyncapp-domain
├─ asyncapp-env
├─ asyncapp-initdb
├─ asyncapp-selenium
└─ asyncapp-web
```

Directory structure

asyncapp/pom.xml

```
<project>
  <!-- omitted -->
  <modules>
    <module>asyncapp-domain</module>
    <module>asyncapp-env</module>
    <module>asyncapp-initdb</module>
    <module>asyncapp-web</module>
    <module>asyncapp-selenium</module>
    <module>asyncbatch</module> <!-- (1) -->
  </modules>
</project>
```

asyncapp/asyncbatch/pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.terasoluna.batch.sample</groupId> <!-- (2) -->
  <artifactId>asyncbatch</artifactId>
  <version>1.0-SNAPSHOT</version> <!-- (2) -->
  <!-- (1) -->
  <parent>
    <groupId>org.terasoluna.batch.sample</groupId>
    <artifactId>asyncapp</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
  </parent>
  <!-- omitted -->
</project>
```

Deleted / added contents

Sr. No.	Description
(1)	Add settings for considering the Web application as a parent and batch application as a child.
(2)	Delete unnecessary description with deletion of child or sub-module.

Addition of dependent library

Add a batch application as a dependent library of Web application.

asyncapp/async-web/pom.xml

```xml
<project>
  <!-- omitted -->
  <dependencies>
    <!-- (1) -->
    <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>asyncbatch</artifactId>
        <version>${project.version}</version>
    </dependency>
    <!-- omitted -->
  </dependencies>
  <!-- omitted -->
</project>
```

Details added

Sr. No.	Description
(1)	Add a batch application as a dependent library of Web application.

4.4.3.3. Implementation of Web application

Here, a RESTful Web service is created as a Web application using TERASOLUNA Server 5.x Development Guideline as a reference below.

Setting for enabling Spring MVC component which is necessary for RESTful Web Service

4.4.3.3.1. Web application settings

At first, add, delete and edit various configuration files from the blank project of Web application.

> For the explanation, an implementation which use RESTful Web Service as an implementation status of batch application is given.
> Procedure will be same even when conventional Web application (Servlet/JSP) or SOAP is used. Read accordingly.

```
asyncapp
└ asyncapp-web
  └ src
    └ main
      └ resources
        └ META-INF
          └ spring
            ├ spring-mvc.xml (1)
            └ spring-mvc-rest.xml (2)
```

Bean definition file to be added/deleted from a blank project

Description example of asyncapp/asyncapp-web/src/main/resources/META-INF/spring/spring-mvc-rest.xml

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

<bean id="jsonMessageConverter"

class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"
      p:objectMapper-ref="objectMapper"/>

<bean id="objectMapper"
      class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean">
  <property name="dateFormat">
      <bean class="com.fasterxml.jackson.databind.util.StdDateFormat"/>
  </property>
</bean>

<mvc:annotation-driven>
  <mvc:message-converters register-defaults="false">
    <ref bean="jsonMessageConverter"/>
  </mvc:message-converters>
</mvc:annotation-driven>

<mvc:default-servlet-handler/>

<!-- (2) -->
<context:component-scan base-package="org.terasoluna.batch.sample.app.api"/>

<!-- (3) -->
<bean
class="org.springframework.batch.core.configuration.support.AutomaticJobRegistrar">
    <property name="applicationContextFactories">
        <bean
class="org.springframework.batch.core.configuration.support.ClasspathXmlApplicationCon
textsFactoryBean">
            <property name="resources">
                <list>
                  <value>classpath:/META-INF/jobs/**/*.xml</value>
                </list>
            </property>
        </bean>
    </property>
    <property name="jobLoader">
        <bean
class="org.springframework.batch.core.configuration.support.DefaultJobLoader"
              p:jobRegistry-ref="jobRegistry"/>
    </property>
</bean>

<!-- (4) -->
<task:executor id="taskExecutor" pool-size="3" queue-capacity="10"/>
```

```xml
<!-- (5) -->
<bean id="jobLauncher"
class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
      p:jobRepository-ref="jobRepository"
      p:taskExecutor-ref="taskExecutor"/>
<!-- omitted -->
```

Description example of asyncapp/asyncapp-web/src/main/webapp/WEB-INF/web.xml

```xml
<!-- omitted -->
<servlet>
    <servlet-name>restApiServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <!-- (6) -->
        <param-value>classpath*:META-INF/spring/spring-mvc-rest.xml</param-value>
    </init-param>
    <!-- (7) -->
    <init-param>
        <param-name>spring.profiles.active</param-name>
        <param-value>async</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>restApiServlet</servlet-name>
    <url-pattern>/api/v1/*</url-pattern>
</servlet-mapping>
<!-- omitted -->
```

RESTful Web Service validation example

Sr. No.	Description
(1)	Import `launch-context.xml` which is in the batch application and incorporate required Bean definition.
(2)	Describe package for dynamically scanning the controller.
(3)	Describe a Bean definition of `AutomaticJobRegistrar` which dynamically loads as a child or sub context by modularizing each Bean definition file.

Sr. No.	Description
(4)	Define `TaskExecutor` which executes the job asynchronously. Asynchronous execution can be performed by setting `AsyncTaskExecutor` implementation class in `TaskExecutor` of JobLauncher. Use `ThreadPoolTaskExecutor` which is one of the components of `AsyncTaskExecutor` implementation class. Further, multiplicity of threads which can be operated in parallel can be specified. In this example, 3 threads are assigned to the job execution and requests exceeding this number are queued upto 10. Queued job is in "not started" state, however REST request is considered to be successful.+ In addition, job requests that exceed the queuing limit generate `` `org.springframework.core.task.TaskRejectedException `` and job run request is rejected.
(5)	Override `jobLauncher` defined in `launch-context.xml` to enable `taskExecutor` of (4).
(6)	Specify `spring-mvc-rest.xml` described above as a Bean definition read by `DispatcherServlet`.
(7)	Specify `async` which shows an asynchronous batch, as a profile of Spring Framework.

When async profile is not specified

In this case, a Bean defined in `launch-context.xml` which should be shared across Web applications is duplicated for each job.
Even in case of duplication, since the operation takes place at the functional level, it is difficult to notice an error and it may result in unexpected resource exhaustion and performance degradation. Must be specified.

Thread pool sizing

When the upper limit of thread pool is in excess, an enormous amount of jobs run in parallel resulting in deterioration of entire thread pool. Sizing should be done and appropriate upper value must be determined.
Besides thread pool of asynchronous execution, request thread of Web container and other applications working in the same enclosure must also be considered.

Further, a separate request must be sent from Web client for checking occurrence of `TaskRejectException` due to thread pool exhaustion and its re-execution. Hence, `queue-capacity` which waits for job to start must be set at the time of thread pool exhaustion.

Implementation of RESTful Web Service API

Here, "Running a job" and "Job status check" are defined as 2 examples of requests used in REST API.

REST API Definition example

Sr. No.	API	Path	HTTP method	Request / Response	Message format	Message details
(1)	Running a job	/api/v1/job/*Job name*	POST	Request	JSON	Job parameter
				Response	JSON	Job execution ID Job name Message
(2)	Job execution status check	/api/v1/job/*Job execution ID*	GET	Request	N/A	N/A
				Response	JSON	Job execution ID Job name Job execution status Job exit code Step execution ID Step name Step exit code

4.4.3.3.2. Implementation of JavaBeans used in Controller

Create following 3 classes that are returned to REST client as JSON message.

- Job run operation `JobOperationResource`
- Job execution status `JobExecutionResource`
- Step execution status `StepExecutionResource`

These classes are implementations for reference except for job execution ID (`job execution id`) of `JobOperationResource` and implementation of field is optional.

Implementation example of job run operation information

```java
// asyncapp/asyncapp-
web/src/main/java/org/terasoluna/batch/sample/app/api/jobinfo/JobOperationResource.jav
a
package org.terasoluna.batch.sample.app.api.jobinfo;

public class JobOperationResource {

    private String jobName = null;

    private String jobParams = null;

    private Long jobExecutionId = null;

    private String errorMessage = null;

    private Exception error = null;

    // Getter and setter are omitted.
}
```

Implementation example of job execution information

```
// asyncapp/asyncapp-
web/src/main/java/org/terasoluna/batch/sample/app/api/jobinfo/JobExecutionResource.jav
a
package org.terasoluna.batch.sample.app.api.jobinfo;

// omitted.

public class JobExecutionResource {

    private Long jobExecutionId = null;

    private String jobName = null;

    private Long stepExecutionId = null;

    private String stepName = null;

    private List<StepExecutionResource> stepExecutions = new ArrayList<>();

    private String status = null;

    private String exitStatus = null;

    private String errorMessage;

    private List<String> failureExceptions = new ArrayList<>();

    // Getter and setter are omitted.
}
```

Implementation example of step execution information

```
// asyncapp/asyncapp-
web/src/main/java/org/terasoluna/batch/sample/app/api/jobinfo/StepExecutionResource.ja
va
package org.terasoluna.batch.sample.app.api.jobinfo;

public class StepExecutionResource {

  private Long stepExecutionId = null;

  private String stepName = null;

  private String status = null;

  private List<String> failureExceptions = new ArrayList<>();

    // Getter and setter are omitted.
}
```

4.4.3.3.3. Implementation of controller

A controller of RESTful Web Service is implemented by using `@RestController`.
in order to simplify, `JobOperator` is injected in the controller and running a job and execution status are fetched. Of course, `JobOperator` can also be started by using Service from the controller in accordance with TERASOLUNA Server 5.x.

> *About job parameters that are passed at the time of running a job*
>
> The job parameter passed in the second argument of `JobOperator#start()` at running a job is `String`. When there are multiple job parameters, they should be separated by using a comma unlike `CommandLineJobRunner` of synchronous execution. Basically the format is as below.
> `{Job parameter 1}={Value 1},{Job parameter 2}={Value 2},…`
>
> This is same as the method of specifying job parameters in Asynchronous execution (DB polling).

Example of implementing a controller

```
// asyncapp/asyncapp-
web/src/main/java/org/terasoluna/batch/sample/app/api/JobController.java
package org.terasoluna.batch.sample.app.api;

// omitted

// (1)
@RequestMapping("job")
@RestController
public class JobController {
```

```
    // (2)
    @Inject
    JobOperator jobOperator;

    // (2)
    @Inject
    JobExplorer jobExplorer;

    @RequestMapping(value = "{jobName}", method = RequestMethod.POST)
    public ResponseEntity<JobOperationResource> launch(@PathVariable("jobName") String
jobName,
            @RequestBody JobOperationResource requestResource) {

        JobOperationResource responseResource = new JobOperationResource();
        responseResource.setJobName(jobName);
        try {
            // (3)
            Long jobExecutionId = jobOperator.start(jobName, requestResource
.getJobParams());
            responseResource.setJobExecutionId(jobExecutionId);
            return ResponseEntity.ok().body(responseResource);
        } catch (NoSuchJobException | JobInstanceAlreadyExistsException |
JobParametersInvalidException e) {
            responseResource.setError(e);
            return ResponseEntity.badRequest().body(responseResource);
        }
    }

    @RequestMapping(value = "{jobExecutionId}", method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    public JobExecutionResource getJob(@PathVariable("jobExecutionId") Long
jobExecutionId) {

        JobExecutionResource responseResource = new JobExecutionResource();
        responseResource.setJobExecutionId(jobExecutionId);

        // (4)
        JobExecution jobExecution = jobExplorer.getJobExecution(jobExecutionId);

        if (jobExecution == null) {
            responseResource.setErrorMessage("Job execution not found.");
        } else {
            mappingExecutionInfo(jobExecution, responseResource);
        }

        return responseResource;
    }

    private void mappingExecutionInfo(JobExecution src, JobExecutionResource dest) {
      dest.setJobName(src.getJobInstance().getJobName());
```

```
        for (StepExecution se : src.getStepExecutions()) {
            StepExecutionResource ser = new StepExecutionResource();
            ser.setStepExecutionId(se.getId());
            ser.setStepName(se.getStepName());
            ser.setStatus(se.getStatus().toString());
            for (Throwable th : se.getFailureExceptions()) {
                ser.getFailureExceptions().add(th.toString());
            }
            dest.getStepExecutions().add(ser);
        }
        dest.setStatus(src.getStatus().toString());
        dest.setExitStatus(src.getExitStatus().toString());
      }
}
```

Implementation of controller

Sr. No.	Description
(1)	Specify `@RestController`. Further, when servlet mapping of `web.xml` is done by using `@RequestMapping("job")`, base path of REST API is `contextName/api/v1/job/`.
(2)	Describe field injections of `JobOperator` and `JobExplorer`.
(3)	Use `JobOperator` and start a new asynchronous job. Receive job execution ID as a return value and return to REST client.
(4)	Use `JobExplorer` and fetch job execution status (`JobExecution`) based on job execution ID. Return it to REST client after converting it to a pre-designed message.

4.4.3.3.4. Integration of Web/batch application module setting

Batch application module (`asyncbatch`) operates as a stand-alone application. Hence, batch application module (`asyncbatch`) consists of settings which are in conflict and overlapping with settings of Web application module (`asyncapp-web`). These settings must be integrated as required.

1. Integration of log configuration file `logback.xml`
   When multiple Logback definition files are defined in Web/batch, they do not work appropriately.+ The contents of `asyncbatch/src/main/resources/logback.xml` are integrated into same file of `asyncapp-env/src/main/resources/` and then the file is deleted.
2. Data source and MyBatis configuration file are not integrated
   Definitions of data source and MyBatis configuration file are not integrated between Web/batch since the definition of application context is independent due to following relation.
   - `asyncbatch` module of the batch is defined in the servlet as a closed context.
   - `asyncapp-domain` and `asyncapp-env` modules of Web are defined as contexts used by entire application.

Cross-reference of data source and MyBatis settings by Web and batch modules

Since the scope of context for Web and batch modules is different, data source, MyBatis settings and Mapper interface cannot be referred especially from Web module.
Since initialization of RDBMS schema is also carried out independently based on the different settings of respective modules, adequate care must be taken not to perform unintended initialization due to mutual interference.

CSRF countermeasures specific to REST controller

When a request is sent for REST controller in the initialization settings of Web blank project, it results in a CSRF error and execution of job is rejected. Hence, explanation is given here assuming that CSRF countermeasures are disabled by the following method.

CSRF countermeasures

Web application created here is not published on the internet and CSRF countermeasures are disabled on the premise that REST request is not sent from a third party who can exploit CSRF as a means of attack. Please note that necessity may differ in the actual Web application depending on the operating environment.

4.4.3.3.5. Build

Build Maven command and create a war file.

```
$ cd asyncapp
$ ls
asyncbatch/  asyncapp-web/  pom.xml
$ mvn clean package
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] TERASOLUNA Server Framework for Java (5.x) Web Blank Multi Project (MyBatis3)
[INFO] TERASOLUNA Batch Framework for Java (5.x) Blank Project
[INFO] asyncapp-web
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building TERASOLUNA Server Framework for Java (5.x) Web Blank Multi Project
(MyBatis3) 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------

(omitted)

[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] TERASOLUNA Server Framework for Java (5.x) Web Blank Multi Project (MyBatis3)
SUCCESS [  0.226 s]
[INFO] TERASOLUNA Batch Framework for Java (5.x) Blank Project SUCCESS [  6.481s]
[INFO] asyncapp-web ....................................... SUCCESS [  5.400 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12.597 s
[INFO] Finished at: 2017-02-10T22:32:43+09:00
[INFO] Final Memory: 38M/250M
[INFO] ------------------------------------------------------------------------
$
```

4.4.3.3.6. Deploy

Start a Web container like Tomcat and deploy `war` file generated in the build. Detailed process is omitted.

4.4.3.4. Job start and confirmation of execution results using REST Client

Here, curl command is used as a REST client and an asynchronous job is started.

```
$ curl -v \
  -H "Accept: application/json" -H "Content-type: application/json" \
  -d '{"jobParams": "param1=value1"}' \
  http://localhost:8080/asyncapp-web/api/v1/job/job01
* timeout on name lookup is not supported
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8088 (#0)
> POST /asyncapp-web/api/v1/job/job01 HTTP/1.1
> Host: localhost:8088
> User-Agent: curl/7.51.0
> Accept: application/json
> Content-type: application/json
> Content-Length: 30
>
* upload completely sent off: 30 out of 30 bytes
< HTTP/1.1 200
< X-Track: 0267db93977b4552880a4704cf3e4565
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Fri, 10 Feb 2017 13:55:46 GMT
<
{"jobName":"job01","jobParams":null,"jobExecutionId":3,"error":null,"errorMessag
e":null}* Curl_http_done: called premature == 0
* Connection #0 to host localhost left intact
$
```

From the above, it can be confirmed that job is executed with a job execution ID `jobExecutionId = 3`.
Subsequently, job execution results are fetched by using job execution ID.

```
$ curl -v http://localhost:8080/asyncapp-web/api/v1/job/3
* timeout on name lookup is not supported
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8088 (#0)
> GET /asyncapp-web/api/v1/job/3 HTTP/1.1
> Host: localhost:8088
> User-Agent: curl/7.51.0
> Accept: */*
>
< HTTP/1.1 200
< X-Track: 7d94bf4d383745efb20cbf37cb6a8e13
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Fri, 10 Feb 2017 14:07:44 GMT
<
{"jobExecutionId":3,"jobName":"job01","stepExecutions":[{"stepExecutionId":5,"st
epName":"job01.step01","status":"COMPLETED","failureExceptions":[]}],"status":"C
OMPLETED","exitStatus":"exitCode=COMPLETED;exitDescription=","errorMessage":null
}* Curl_http_done: called premature == 0
* Connection #0 to host localhost left intact
$
```

Since `exitCode=COMPLETED`, it can be confirmed that the job is completed successfully.

> *When execution results of curl are to be determined by a shell script etc*
>
> In the example above, it is displayed upto the response message using REST API.
> When only HTTP status is to be confirmed by curl command, HTTP status can be displayed in standard output by considering `curl -s URL -o /dev/null -w "%{http_code}\n"`.
> However, since job execution ID need to analyse JSON of response body part, REST client application must be created as required.

4.4.4. How to extend

4.4.4.1. Stopping and restarting jobs

It is necessary to stop and restart asynchronous jobs from the multiple jobs that are being executed. Further, when jobs of identical names are running in parallel, it is necessary to target only those jobs with the issues. Hence, job execution to be targeted must be identified and the status of the job must be confirmed.
When this premise is met, an implementation for stopping and restarting asynchronous executions is explained here.

Further, a method to add job stopping (stop) and restarting (restart) is explained for `JobController` of Implementation of controller.

Job stopping and restarting can also be implemented without using `JobOperator`. For details, refer Job management and identify a method suitable for this objective.

Implementation example of stop and restart

```
// asyncapp/asyncapp-
web/src/main/java/org/terasoluna/batch/sample/app/api/JobController.java
package org.terasoluna.batch.sample.app.api;

// omitted

@RequestMapping("job")
@RestController
public class JobController {

    // omitted.

    @RequestMapping(value = "stop/{jobExecutionId}", method = RequestMethod.PUT)
    @Deprecated
    public ResponseEntity<JobOperationResource> stop(
            @PathVariable("jobExecutionId") Long jobExecutionId) {

      JobOperationResource responseResource = new JobOperationResource();
      responseResource.setJobExecutionId(jobExecutionId);
      boolean result = false;
      try {
          // (1)
          result = jobOperator.stop(jobExecutionId);
          if (!result) {
              responseResource.setErrorMessage("stop failed.");
              return ResponseEntity.badRequest().body(responseResource);
          }
          return ResponseEntity.ok().body(responseResource);
      } catch (NoSuchJobExecutionException | JobExecutionNotRunningException e) {
          responseResource.setError(e);
          return ResponseEntity.badRequest().body(responseResource);
      }
    }

    @RequestMapping(value = "restart/{jobExecutionId}",
                    method = RequestMethod.PUT)
    @Deprecated
    public ResponseEntity<JobOperationResource> restart(
            @PathVariable("jobExecutionId") Long jobExecutionId) {

        JobOperationResource responseResource = new JobOperationResource();
        responseResource.setJobExecutionId(jobExecutionId);
        try {
            // (2)
```

```
            Long id = jobOperator.restart(jobExecutionId);
            responseResource.setJobExecutionId(id);
            return ResponseEntity.ok().body(responseResource);
        } catch (JobInstanceAlreadyCompleteException |
                NoSuchJobExecutionException | NoSuchJobException |
                JobRestartException | JobParametersInvalidException e) {
            responseResource.setErrorMessage(e.getMessage());
            return ResponseEntity.badRequest().body(responseResource);
        }
    }

    // omitted.
}
```

Implementation example of stop / restart using controller

Sr. No.	Description
(1)	Specify "stop" for job being executed by calling `JobOperator#stop()`.
(2)	Re-execute from the step where the job has terminated abnormally or stopped by calling `JobOperator#restart()`.

4.4.4.2. Multiple running

Multiple running signify that a Web container is started for multiple times and waits for respective job requests.

Execution of asynchronous jobs is controlled by external RDBMS so as to connect to each application. By sharing an external RDBMS, it is possible to wait for an asynchronous job to be started across the same enclosure or another enclosure.

Applications include load balancing and redundancy for specific jobs. However, as described in Web application implementation, these effects cannot be obtained easily just by starting multiple Web containers or enhancing parallel operations. Sometimes measures similar to a general Web application need to be taken in order to obtain the effect. An example is given below.

- 1 request processing operates in a stateless manner according to the characteristics of Web application, however, asynchronous execution of batch is likely to have a reduced failure tolerance unless it is designed in combination with job start results and confirmation.
  For example, even when Web container for starting a job is made redundant, it is difficult to confirm the progress and results of the job when the job execution ID is lost after starting a job due to failure on the client side.
- A function to distribute request destinations on the client side must be implemented and a load balancer must be introduced in order to distribute the load on multiple Web containers.

In this way, adequacy of multiple starts cannot be necessarily determined. Hence, using load balancer and reviewing a control method to send requests by Web client should be considered based on the purpose and use. A design which does not degrade the performance and fault tolerance of the asynchronous execution application is required.

4.5. Listener

4.5.1. Overview

A listener is an interface for inserting processing before and after executing a job or a step.

Since this function works differently for chunk model and tasket model, respective explanations are given.

A listener consists of multiple interfaces, respective roles are explained here. Subsequently, how to set and implement a listener is explained.

4.5.1.1. Types of listener

A lot of listener interfaces are defined in Spring Batch. All will not be explained here, however we will focus on the interface with highest usage frequency.

A listener is roughly divided into 2 types.

JobListener
: An interface to insert the processing for execution of the job

StepListener
: An interface to insert the processing for execution of the step

> ⚠ *About JobListener*
>
> An interface called `JobListener` does not exist in Spring Batch. It is conveniently described in this guideline for the comparison with `StepListener`.
> Java Batch(jBatch) consists of an interface called `javax.batch.api.listener.JobListener`, hence care should be taken at the time of implementation to avoid mistakes. Further, `StepListener` also consists of interface with same name but different signature (`javax.batch.api.listener.StepListener`), so it is necessary to take adequate precautions.

4.5.1.1.1. JobListener

`JobListener` interface consists of only one `JobExecutionListener`.

JobExecutionListener
: Process is inserted prior to starting a job and after terminating a job.

JobExecutionListener interface

```
public interface JobExecutionListener {
  void beforeJob(JobExecution jobExecution);
  void afterJob(JobExecution jobExecution);
}
```

4.5.1.1.2. StepListener

`StepListener` interface is of multiple types as below.

StepListener

Marker interfaces of various listeners will be introduced later.

StepExecutionListener

Process is inserted prior to starting a step and after terminating a job.

StepExecutionListener interface

```
public interface StepExecutionListener extends StepListener {
  void beforeStep(StepExecution stepExecution);
  ExitStatus afterStep(StepExecution stepExecution);
}
```

ChunkListener

A process is inserted between before and after processing of one chunk and when an error occurs.

ChunkListener interface

```
public interface ChunkListener extends StepListener {
  static final String ROLLBACK_EXCEPTION_KEY = "sb_rollback_exception";
  void beforeChunk(ChunkContext context);
  void afterChunk(ChunkContext context);
  void afterChunkError(ChunkContext context);
}
```

> *Uses of ROLLBACK_EXCEPTION_KEY*
>
> It is used when the exception occurred is to be fetched by `afterChunkError` method. If an error occurs during chunk process, Spring Batch uses `sb_rollback_exception` key in `ChunkContext` to call `ChunkListener` after storing the exception which can be accessed as below.
>
> *Usage example*
>
> ```
> public void afterChunkError(ChunkContext context) {
>     logger.error("Exception occurred while chunk. [context:{}]",
> context,
>             context.getAttribute(ChunkListener.
> ROLLBACK_EXCEPTION_KEY));
> }
> ```

ItemReadListener

Insert a process before and after fetching 1 data record by ItemReader and when an error occurs.

ItemReadListener interface

```
public interface ItemReadListener<T> extends StepListener {
  void beforeRead();
  void afterRead(T item);
  void onReadError(Exception ex);
}
```

ItemProcessListener

Insert a process before and after processing 1 data record by ItemProcessor and when an error occurs.

ItemProcessListener interface

```
public interface ItemProcessListener<T, S> extends StepListener {
  void beforeProcess(T item);
  void afterProcess(T item, S result);
  void onProcessError(T item, Exception e);
}
```

ItemWriteListener

Insert a process before and after output of 1 chunk by ItemWriter and when an error occurs.

ItemWriteListener interface

```
public interface ItemWriteListener<S> extends StepListener {
  void beforeWrite(List<? extends S> items);
  void afterWrite(List<? extends S> items);
  void onWriteError(Exception exception, List<? extends S> items);
}
```

This guideline does not explain following listeners.

- Retry type listener
- Skip type listener

These listeners are intended to be used for exception handling, however, the policy of these guidelines is not to perform exception handling using these listeners. For details, refer Exception handling.

4.5.2. How to use

Explanation is given about how to implement and set a listener.

4.5.2.1. Implementation of a listener

Explanation is given about how to implement and set a listener.

1. Implement the listener interface with `implements`.
2. Implement components with method-based annotation.

The type of implementation to use will be choosed on the role of the listener. Criteria will be described later.

4.5.2.1.1. When an interface is to be implemented

Various listener interfaces are implemented by using `implements`. Multiple interfaces can be implemented at the same time based on requirement. Implementation example is shown below.

Implementation example for JobExecutionListener

```
@Component
public class JobExecutionLoggingListener implements JobExecutionListener { // (1)

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

    @Override
    public void beforeJob(JobExecution jobExecution) { // (2)
        // do nothing.
    }

    @Override
    public void afterJob(JobExecution jobExecution) { // (3)

        logger.info("job finished.[JobName:{}][ExitStatus:{}]",
                jobExecution.getJobInstance().getJobName(),
                jobExecution.getExitStatus().getExitCode());  // (4)

        // per step execution
        // (5)
        jobExecution.getStepExecutions().forEach(stepExecution -> {
            Object errorItem = stepExecution.getExecutionContext().get("ERROR_ITEM");
            if (errorItem != null) {
                logger.error("detected error on this item processing. " +
                        "[step:{}] [item:{}]", stepExecution.getStepName(),
                        errorItem);
            }
        });
    }
}
```

Configuration example of listener

```
<batch:job id="chunkJobWithListener" job-repository="jobRepository">
     <batch:step id="chunkJobWithListener.step01">
         <batch:tasklet transaction-manager="jobTransactionManager">
             <batch:chunk reader="reader" processor="processor"
                          writer="writer" commit-interval="10"/>
             <batch:listeners>
                 <batch:listener ref="loggingEachProcessInStepListener"/>
             </batch:listeners>
         </batch:tasklet>
     </batch:step>
     <batch:listeners>
         <batch:listener ref="jobExecutionLoggingListener"/> <!-- (6) -->
     </batch:listeners>
 </batch:job>
```

Description

Sr. No.	Description
(1)	Implement `JobExecutionListener` using `implements`.
(2)	Implement `beforeJob` method defined by `JobExecutionListener`. In this example, no operation is performed before starting a job.
(3)	Implement `afteJob` method defined by `JobExecutionListener`. In this example, final status of job at the time of job execution and exception information are output in a log.
(4)	Output job name and exit code in INFO log. Fetch necessary information from `JobExecution` of argument.
(5)	Output exception occurred for each step in a log. Fetch and implement `StepExecution` linked from `JobExecution` of argument. Here, input data which caused the exception is fetched from `ExecutionContext` using a key called `ERROR_ITEM`. For example set in `ExecutionContext`, refer Exception handling of job units.
(6)	Set the listener implemented in (1), in `<listeners>` tag of Bean definition. Details of setup method are explained in Listener settings.

Listener support class

When multiple listener interfaces are set to `implements`, blank implementation is required to be done for the components which are not necessary for the process. Support classes wherein blank implementation is performed are provided in Spring Batch in order to simplify this operation. Please note that support classes may be used instead of interfaces, and `extends` is used instead of `implements`.

Support class

- `org.springframework.batch.core.listener.ItemListenerSupport`
- `org.springframework.batch.core.listener.StepListenerSupport`

4.5.2.1.2. When annotations are assigned

Annotations corresponding to various listener interfaces are assigned. Multiple annotations can also be implemented as required.

Correspondence table with listener interface

Listener interface	Annotation
JobExecutionListener	`@beforeJob` `@afterJob`
StepExecutionListener	`@BeforeStep` `@AfterStep`

ChunkListener	`@BeforeChunk`
	`@AfterChunk`
	`@afterChunkError`

ItemReadListener	`@BeforeRead`
	`@AfterRead`
	`@OnReadError`

ItemProcessListener	`@beforeProcess`
	`@afterProcess`
	`@onProcessError`

ItemWriteListener	`@BeforeWrite`
	`@AfterWrite`
	`@OnWriteError`

These annotations work for the target scope by assigning them to the implementation method which is divided into components. Implementation example is given below.

Implementation example for ItemProcessor wherein the annotation is assigned

```
@Component
public class AnnotationAmountCheckProcessor implements
        ItemProcessor<SalesPlanDetail, SalesPlanDetail> {

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

    @Override
    public SalesPlanDetail process(SalesPlanDetail item) throws Exception {
        if (item.getAmount().signum() == -1) {
            throw new IllegalArgumentException("amount is negative.");
        }
        return item;
    }

    // (1)
    /*
    @BeforeProcess
    public void beforeProcess(Object item) {
        logger.info("before process. [Item :{}]", item);
    }
    */

    // (2)
    @AfterProcess
    public void afterProcess(Object item, Object result) {
        logger.info("after process. [Result :{}]", result);
    }

    // (3)
    @OnProcessError
    public void onProcessError(Object item, Exception e) {
        logger.error("on process error.", e);
    }
}
```

Configuration example of listener

```
<batch:job id="chunkJobWithListenerAnnotation" job-repository="jobRepository">
    <batch:step id="chunkJobWithListenerAnnotation.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader"
                         processor="annotationAmountCheckProcessor"
                         writer="writer" commit-interval="10"/>  <! -- (4) -->
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Description

Sr. No.	Description
(1)	When the annotation is to be used for implementation, only the annotations of the timing required for the processing should be assigned. In this example, since no operation is required prior to processing of ItemProcess, the implementation wherein `@beforeProcess` is assigned, becomes unnecessary.
(2)	Implement the process to be performed after the processing of ItemProcess. In this example, process results are output in a log.
(3)	Implement processing when an error occurs in ItemProcess. Exception generated in this example is output in a log.
(4)	Set ItemProcess wherein the listener is implemented by using annotation in `<chunk>` tag. Unlike listener interface, the listener is automatically registered even when it is not set in `<listener>` tag.

Constraints for the method which assigns the annotations

Any method cannot be used as a method to assign the annotation. The signature must match with the method of corresponding listener interface. This point is clearly mentioned in javadoc of respective annotations.

Precautions while implementing JobExecutionListener by an annotation

Since JobExecutionListener has a different scope than the other listeners, listener is not automatically registered in the configuration above. Hence, it is necessary to explicitly set in the `<listener>` tag. For details ,refer Listener settings.

Implementation of a listener to Tasklet implementation by using annotation

When a listener is implemented in Tasklet implementation by using an annotation, Note that listener does not start with the following settings.

In case of Tasklet

```
<batch:job id="taskletJobWithListenerAnnotation" job-
repository="jobRepository">
    <batch:step id="taskletJobWithListenerAnnotation.step01">
        <batch:tasklet transaction-manager="jobTransactionManager"
                       ref=
"annotationSalesPlanDetailRegisterTasklet"/>
    </batch:step>
</batch:job>
```

In case of Tasket model, the listener interface should be used in accordance with How to choose an interface or an annotation.

4.5.2.2. Listener settings

Listeners are set by `<listeners>.<listener>` tag of Bean definition. Although it can be described at various locations by XML schema definition, some operations do not work as intended based on the type of interface. Set it to the following position.

Position where listener is set

```
<!-- for chunk mode -->
<batch:job id="chunkJob" job-repository="jobRepository">
    <batch:step id="chunkJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="(1)"
                         processor="(1)"
                         writer="(1)" commit-interval="10"/>
            <batch:listeners>
                <batch:listener ref="(2)"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="(3)"/>
    </batch:listeners>
</batch:job>

<!-- for tasklet mode -->
<batch:job id="taskletJob" job-repository="jobRepository">
    <batch:step id="taskletJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager" ref="tasklet">
            <batch:listeners>
                <batch:listener ref="(2)"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="(3)"/>
    </batch:listeners>
</batch:job>
```

Description of configuration value

Sr. No.	Description
(1)	Set the component which includes the implementation attributing to StepListener, performed by using an annotation. In case of an annotation, it will be inevitably set to this location.
(2)	Set listener interface implementation attributing to StepListener.
(3)	Set listener attributing to JobListener. Either of interface or annotations must be implemented here.

4.5.2.2.1. Setting multiple listeners

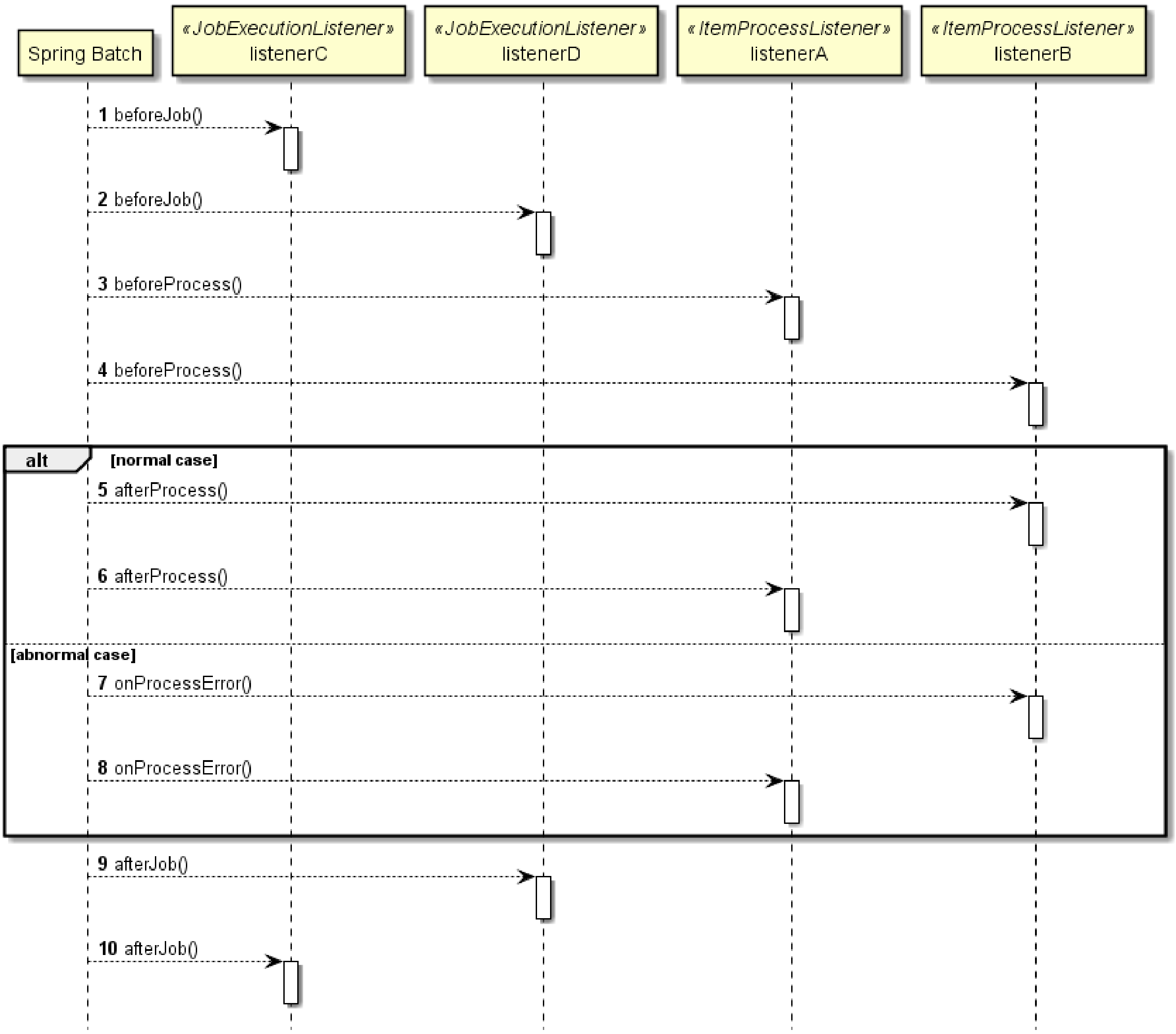
Multiple listeners can be set in `<batch:listeners>` tag.

The sequence in which the listeners are started while registering multiple listeners is shown below.

- ItemProcessListener implementation
  - listenerA, listenerB
- JobExecutionListener implementation
  - listenerC, listenerD

Configuration example of multiple listeners

```
<batch:job id="chunkJob" job-repository="jobRepository">
    <batch:step id="chunkJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader"
                         processor="pocessor"
                         writer="writer" commit-interval="10"/>
            <batch:listeners>
                <batch:listener ref="listenerA"/>
                <batch:listener ref="listenerB"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="listenerC"/>
        <batch:listener ref="listenerD"/>
    </batch:listeners>
</batch:job>
```



Listener startup sequence

- Processing corresponding to pre-processing is started in the sequence of listener registration.
- Processing corresponding to post-processing or error processing is started in the reverse sequence of listener registration.

4.5.2.3. How to choose an interface or an annotation

How to choose listener used a interface or listener used an annotation is explained.

Listener interface

It is used in case of cross-sectional processes which are shared across job, step and chunk.

Annotation

It is used when business logic specific process is to be performed.
As a rule, it is implemented only for ItemProcessor.

Chapter 5. Input/Output of Data

5.1. Transaction control

5.1.1. Overview

In this section, transaction control in jobs will be described in the following order.

1. About the pattern of transaction control in general batch processing
2. Transaction control in Spring Batch
3. How to process resources like database and file transactionally

Since this function is different in usage between chunk model and tasklet model, each will be explained.

5.1.1.1. About the pattern of transaction control in general batch processing

Generally, since batch processing is processing a large number of cases, if any errors are thrown at the end of the processing and all processing need to be done again, the batch system schedule will be adversely affected.
In order to avoid this, the influence at the time of error occurrence is often localized by advancing the process while confirming the transaction for each fixed number of data within the processing of one job.
(Hereafter, we call the "intermediate commit method" as the method of commiting the transaction for every fixed number of data, and the "chunk" as the one grouping the data in the commit unit.)

The points of the intermediate commit method are summarized below.

1. Localize the effects at the time of error occurrence.
   - Even if an error occurs, the processing to the chunk just before the error part is confirmed.
2. Only use a certain amount of resources.
   - Regardless of whether the data to be processed is large or small, only resources for chunks are used, so they are stable.

However, the intermediate commit method is not a valid method in every situation.
Processed data and unprocessed data are mixed in the system even though it is temporary. As a result, since it is necessary to identify unprocessed data at the time of recovery processing, there is a possibility that the recovery becomes complicated. In order to avoid this, all of the cases must be confirmed with one transaction, and not use the intermediate commit method.
(Hereinafter, the method of determining all transactions in one transaction is called "single commit method".)

Nevertheless, if you process a large number of such as tens of thousands of items in a single commit method, you will get a heavy load trying to reflect all the databases when committing. Therefore, although the single commit method is suitable for small-scale batch processing, care must be taken when adopting it in a large-scale batch. So this method is not a versatile method too.

In other words, there is a trade-off between "localization of impact" and "ease of recovery". Which one of "intermediate commit method" and "single commit method" is used depends on the nature of the job and decides which one should be prioritized.
Of course, it is not necessary to implement all the jobs in the batch system on either side. It is natural to basically use "intermediate commit method" and use "single commit method" for special jobs (or the other way).

Below is the summary of advantages, disadvantages and adoption points of "intermediate commit method" and "single commit method".

Features list by method

Commit method	Advantage	Disadvantage	Adoption point
intermediate commit method	Localize the effect at the time of error occurrence	Recovery processing may be complicated	When you want to process large amounts of data with certain machine resources
single commit method	Ensure data integrity	There is a possibility of high work-load when processing a large number of cases	When you want to set the processing result for the persistent resource to All or Nothing Suitable for small batch processing

Notes on inputting and outputting to the same table in the database

In terms of the structure of the database, care is required when handling large amounts of data in processing to input and output to the same table regardless of the commit method.

- As information that guarantees reading consistency is lost due to output (issuance of UPDATE), errors may occur at the input (SELECT).

In order to avoid this, the following measures are taken.

- Increase the area to secure information.
  - When expanding, please consider it carefully in resource design.
  - Since the extension method depends on the database to be used, refer to the manual.
- Divides input data and performs multiplexing processing.
  - Refer to Partitioning Step (Multiple processing) for multiple processing.

5.1.2. Architecture

5.1.2.1. Transaction control in Spring Batch

Job transaction control leverages the mechanism of Spring Batch.

Two kinds of transactions are defined below.

Framework transaction
: Transaction controlled by Spring Batch

User transaction
: Transactions controlled by the user

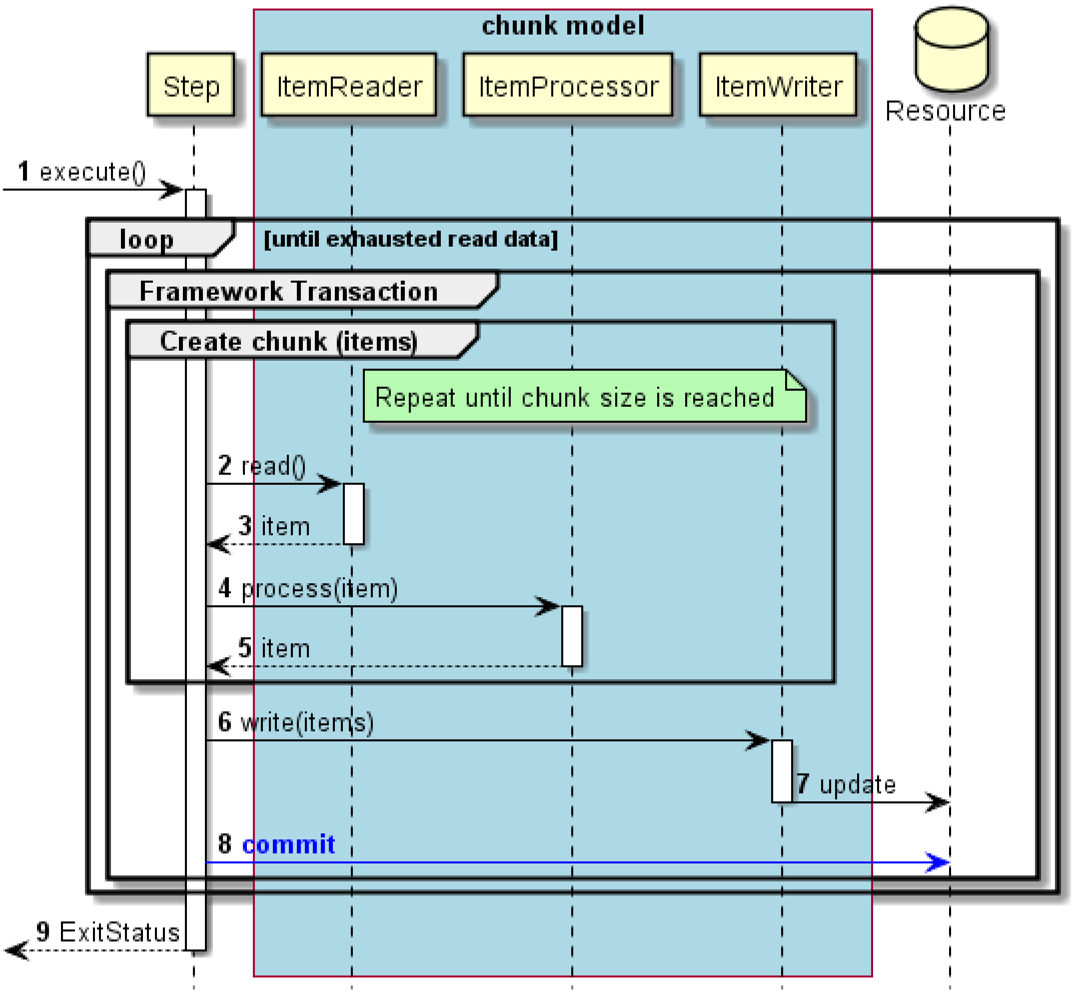
5.1.2.1.1. Transaction control mechanism in chunk model

Transaction control in the chunk model is only the intermediate commit method. A single commit method can not be done.

> The single commit method in the chunk model is reported in JIRA.
> https://jira.spring.io/browse/BATCH-647
> As a result, it is solved by customizing `chunk completion policy` and dynamically changing the chunk size. However, with this method, since all data is stored in one chunk and memory is compressed, it can not be adopted as a method.

A feature of this method is that transactions are repeatedly performed for each chunk.

Transaction control in normal process
: Transaction control in normal process will be explained.



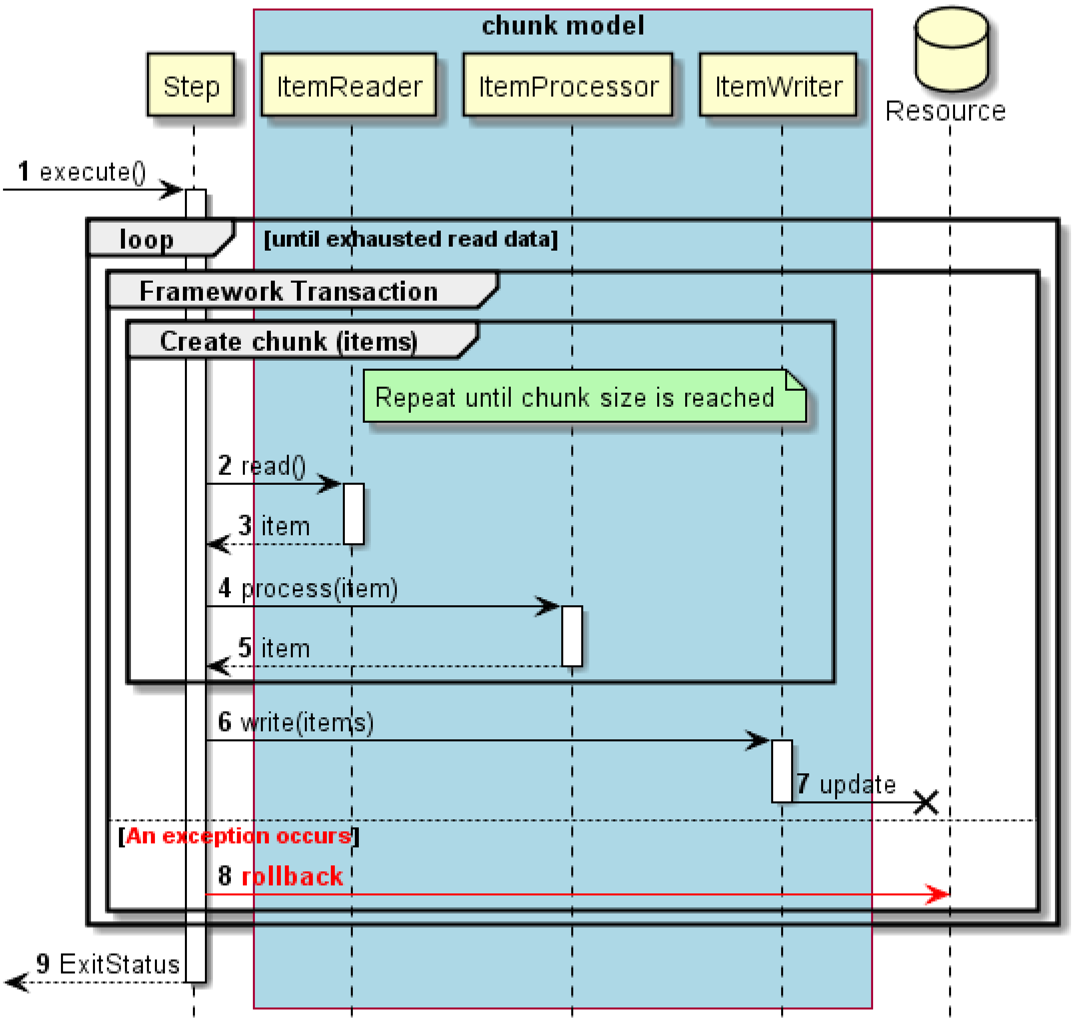
Sequence diagram of normal process

Description of the Sequence Diagram

1. Steps are executed from the job.
   - The subsequent processing is repeated until there is no input data.
   - Start a framework transaction on a per chunk basis.
   - Repeat steps 2 to 5 until the chunk size is reached.
2. The step obtains input data from `ItemReader`.
3. `ItemReader` returns the input data to the step.
4. In the step, `ItemProcessor` processes input data.
5. `ItemProcessor` returns the processing result to the step.
6. The step outputs data for chunk size with `ItemWriter`.
7. `ItemWriter` will output to the target resource.
8. The step commits the framework transaction.

Transaction control in abnormal process

Transaction control in abnormal process will be explained.



Sequence diagram of abnormal process

Description of the Sequence Diagram

1. Steps are executed from the job.
   - The subsequent processing is repeated until there is no input data.
   - Start a framework transaction on a per chunk basis.
   - Repeat steps 2 to 5 until the chunk size is reached.
2. The step obtains input data from `ItemReader`.
3. `ItemReader` returns the input data to the step.
4. In the step, `ItemProcessor` processes input data.
5. `ItemProcessor` returns the processing result to the step.
6. The step outputs data for chunk size with `ItemWriter`.
7. `ItemWriter` will output to the target resource.

If any **exception occurs** between the process from 2 to 7,

8. The step rolls back the framework transaction.

5.1.2.1.2. Mechanism of transaction control in tasklet model

For transaction control in the tasklet model, either the single commit method or the intermediate commit method can be used.

single commit method

Use the transaction control mechanism of Spring Batch

Intermediate commit method

Manipulate the transaction directly with the user

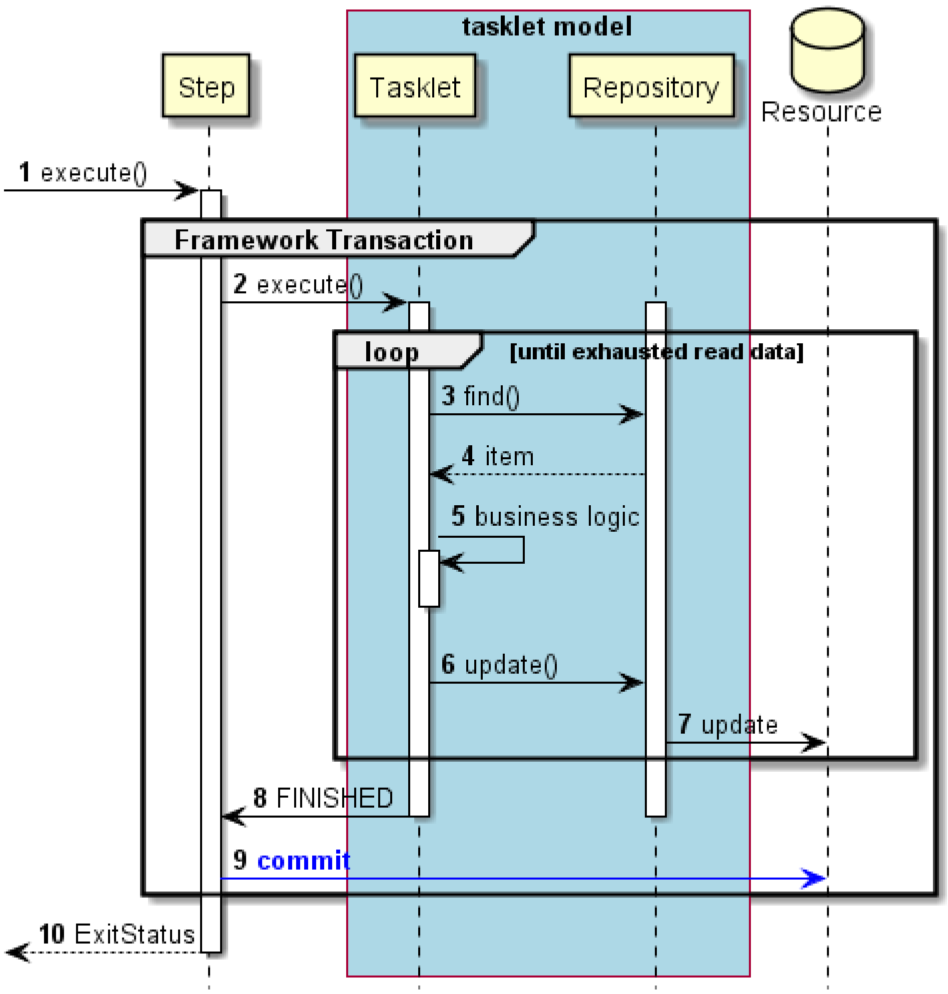
single commit method in tasklet model

Explain the mechanism of transaction control by Spring Batch.

A feature of this method is to process data repeatedly within one transaction.

Transaction control in normal process

Transaction control in normal process will be explained.



Sequence diagram of normal process

Description of the Sequence Diagram

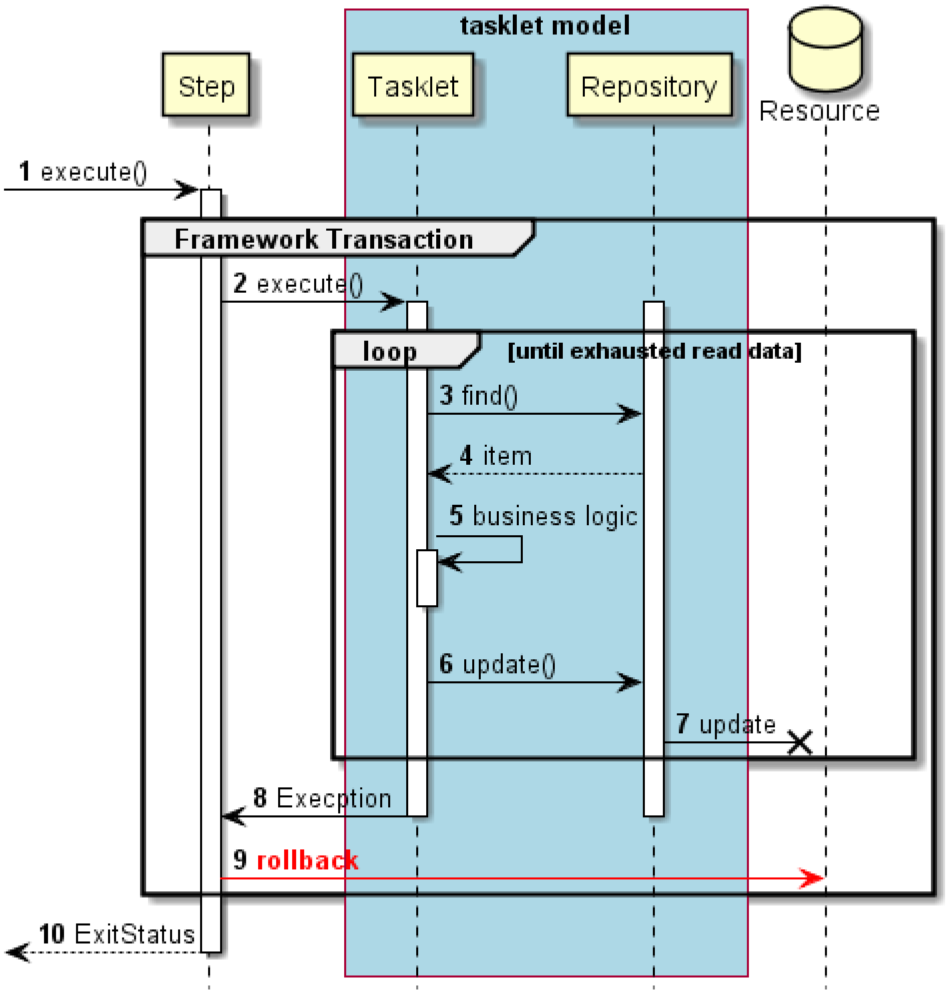
1. Steps are executed from the job.

The step starts a framework transaction.

2. The step executes the tasklet.
   - Repeat steps 3 to 7 until there is no more input data.
3. Tasklet gets input data from `Repository`.
4. `Repository` will return input data to tasklet.
5. Tasklets process input data.
6. Tasklets pass output data to `Repository`.
7. `Repository` will output to the target resource.
8. The tasklet returns the process end to the step.
9. The step commits the framework transaction.

Transaction control in abnormal process

Transaction control in abnormal process will be explained.



Sequence diagram of abnormal process

Description of the Sequence Diagram

1. Steps are executed from the job.
   - The step starts a framework transaction.
   -

2. The step executes the tasklet.
    - Repeat steps 3 to 7 until there is no more input data.
3. Tasklet gets input data from `Repository`.
4. `Repository` will return input data to tasklet.
5. Tasklets process input data.
6. Tasklets pass output data to `Repository`.
7. `Repository` will output to the target resource.

If any **exception occurs** between the process from 2 to 7,

8. The tasklet throws an exception to the step.
9. The step rolls back the framework transaction.

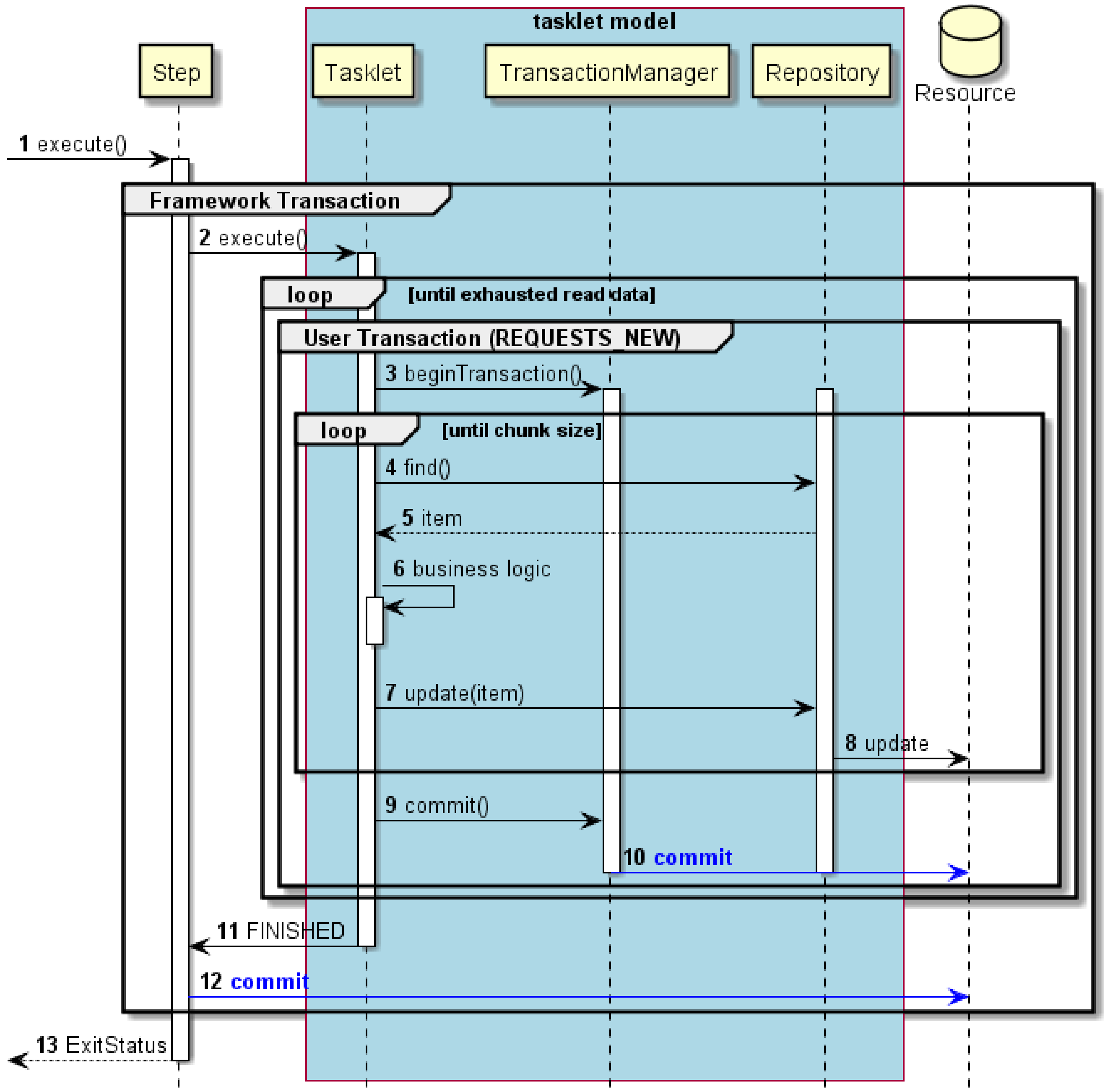
Intermediate commit method in tasklet model

A mechanism for directly operating a transaction by a user will be described.

A feature of this scheme is to start and manipulate a new user transaction within a framework transaction.

Transaction control in normal process

Transaction control in normal process will be explained.



Sequence diagram of normal process

Description of the Sequence Diagram

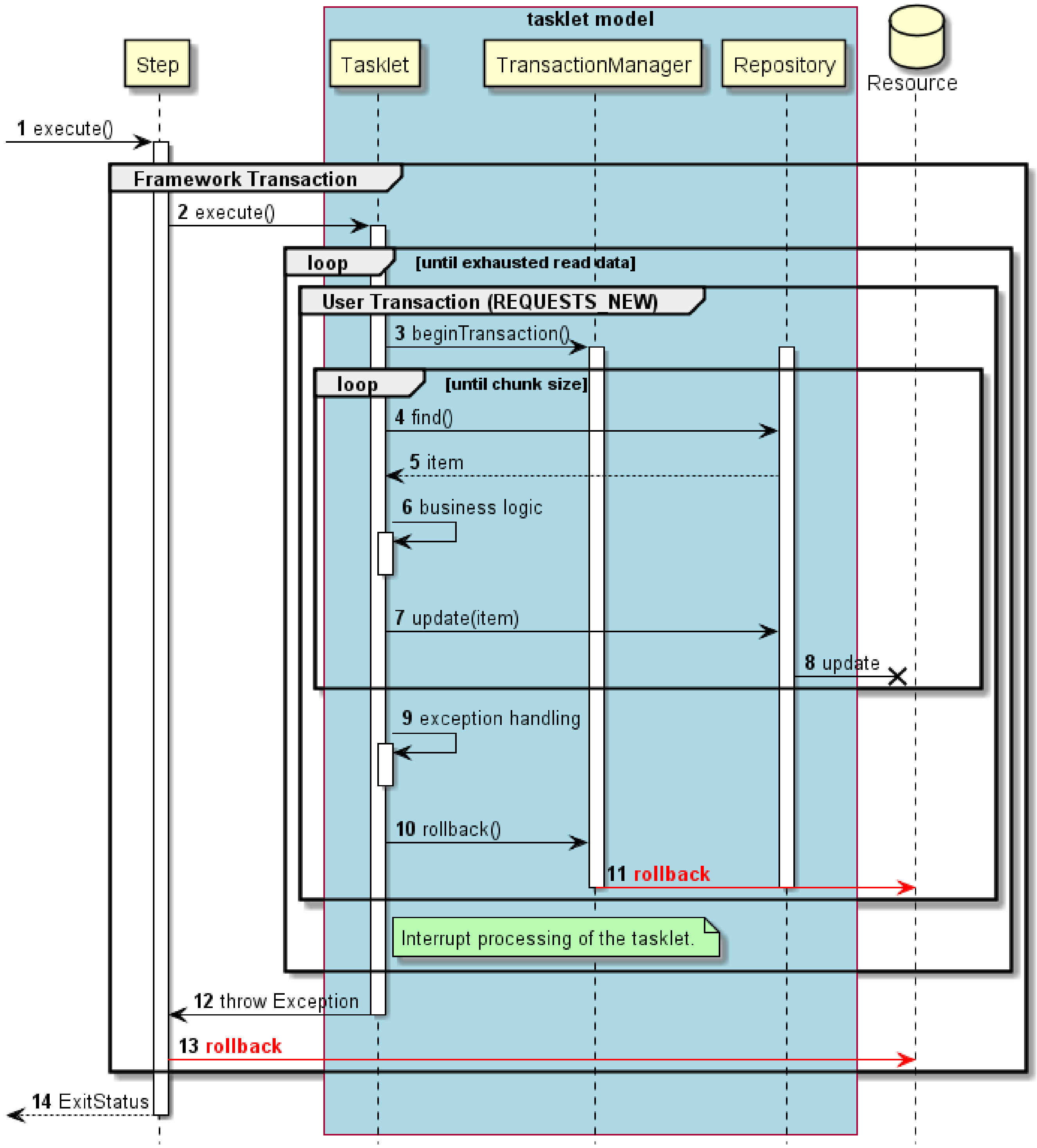
1. Steps are executed from the job.
   - The step starts **framework transaction**.
2. The step executes the tasklet.
   - Repeat steps 3 to 10 until there is no more input data.
3. The tasklet starts **user transaction** via `TransacitonManager`.
   - Execute the user transaction with **REQUESTS_NEW** to separate it from the framework transaction.
   - Repeat steps 4 to 6 until the chunk size is reached.
4. Tasklet gets input data from `Repository`.

5. `Repository` will return input data to tasklet.
6. Tasklets process input data.
7. Tasklets pass output data to `Repository`.
8. `Repository` will output to the target resource.
9. The tasklet commits the **user transaction** via `TransacitonManager`.
10. `TransacitonManager` issues a commit to the target resource.
11. The tasklet returns the process end to the step.
12. The step commits the **framework transaction**.

> In this case, each item is output to a resource, but like the chunk model, it is also possible to update the processing throughput collectively by chunk unit and improve the processing throughput. At that time, you can also use BatchUpdate by setting `executorType` of `SqlSessionTemplate` to `BATCH`. This is the same behavior as using MyBatis' ItemWriter, so you can update it using MyBatis' ItemWriter. For details of MyBatis' ItemWriter, refer to Database access with ItemWriter.

Transaction control in abnormal process

Transaction control in abnormal process will be explained.



Sequence diagram of abnormal process

Description of the Sequence Diagram

1. Steps are executed from the job.
   - The step starts **framework transaction**.
2. The step executes the tasklet.
   - Repeat steps 3 to 11 until there is no more input data.
3. The tasklet starts **user transaction** from `TransacitonManager`.
   - Execute the user transaction with **REQUESTS_NEW** to separate it from the framework

transaction.

   - Repeat steps 4 to 6 until the chunk size is reached.

4. Tasklet gets input data from `Repository`.
5. `Repository` will return input data to tasklet.
6. Tasklets process input data.
7. Tasklets pass output data to `Repository`.
8. `Repository` will output to the target resource.

If any **exception occurs** between the process from 3 to 8,

9. The tasklet processes the exception that occurred.
10. The tasklet performs a rollback of **user transaction** via `TransacitonManager`.
11. `TransacitonManager` issues a rollback to the target resource.
12. The tasklet throws an exception to the step.
13. The step rolls back **framework transaction**.

> *About processing continuation*
>
> Here, although processing is abnormally terminated after handling exceptions and rolling back the processing, it is possible to continue processing the next chunk. In either case, it is necessary to notify the subsequent processing by changing the status / end code of the step that an error has occurred during that process.

> *About framework transactions*
>
> In this case, although the job is abnormally terminated by throwing an exception after rolling back the user transaction, it is also possible to return the processing end to the step and terminate the job normally. In this case, the framework transaction is **committed**.

5.1.2.1.3. Selection policy for model-specific transaction control

In Spring Batch that is the basis of TERASOLUNA Batch 5.x, only the intermediate commit method can be implemented in the chunk model. However, in the tasklet model, either the intermediate commit method or the single commit method can be implemented.

Therefore, in TERASOLUNA Batch 5.x, when the single commit method is necessary, it is to be implemented in the tasklet model.

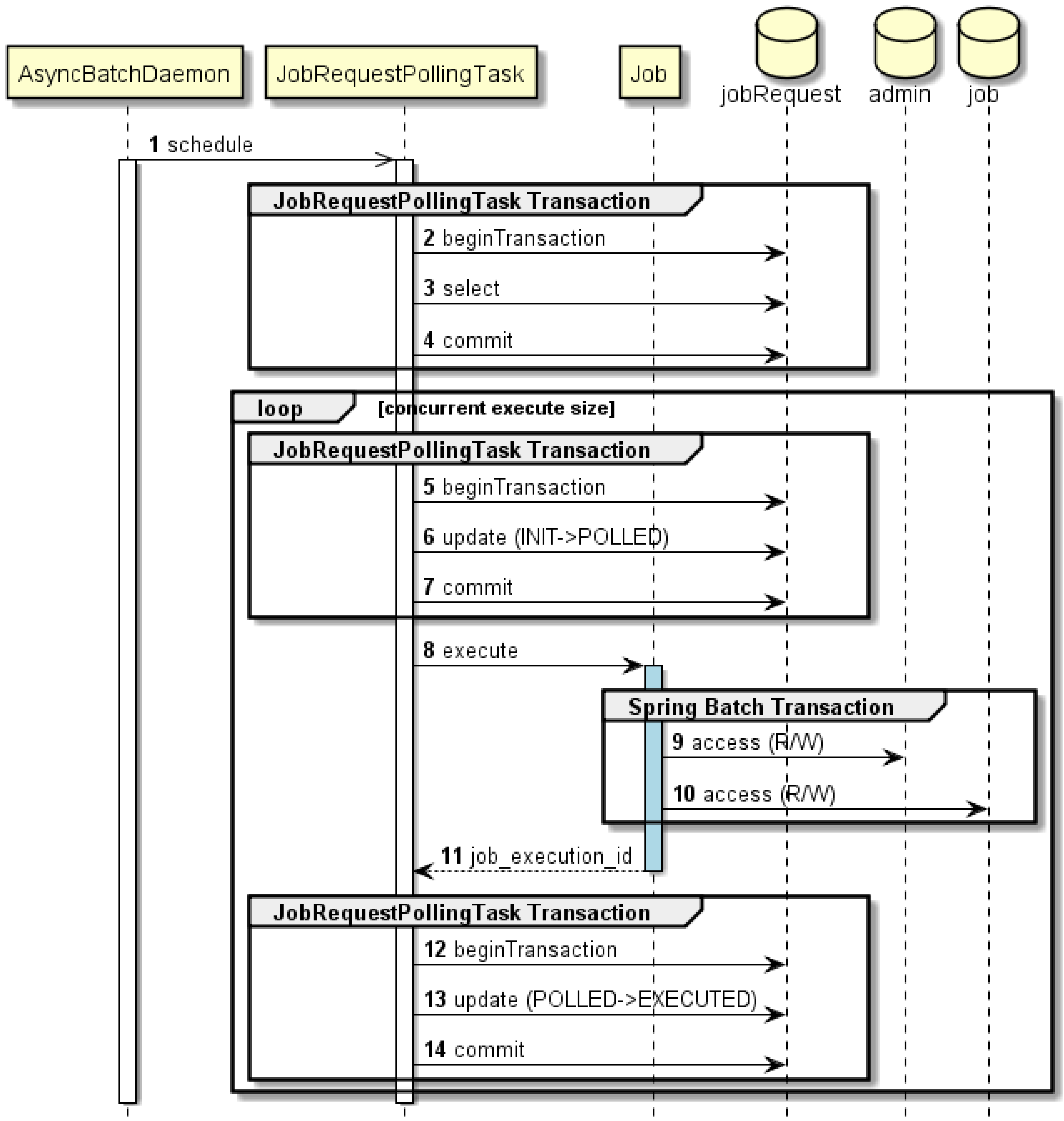
5.1.2.2. Difference in transaction control for each execution method

Depending on the execution method, a transaction that is not managed by Spring Batch occurs before and after the job is executed. This section explains transactions in two asynchronous execution processing schemes.

5.1.2.2.1. About transaction of DB polling

Regarding processing to the Job-request-table performed by the DB polling, transaction processing other than Spring Batch managed will be performed. Also, regarding exceptions that occurred in the job, since correspondence is completed within the job, it does not affect transactions performed by `JobRequestPollTask`.

A simple sequence diagram focusing on transactions is shown in the figure below.



Transaction of DB polling

Description of the Sequence Diagram

1. `JobRequestPollTask` is executed periodically from asynchronous batch daemon.
2. `JobRequestPollTask` will start a transaction other than Spring Batch managed.
3. `JobRequestPollTask` will retreive an asynchronous batch to execute from Job-request-table.

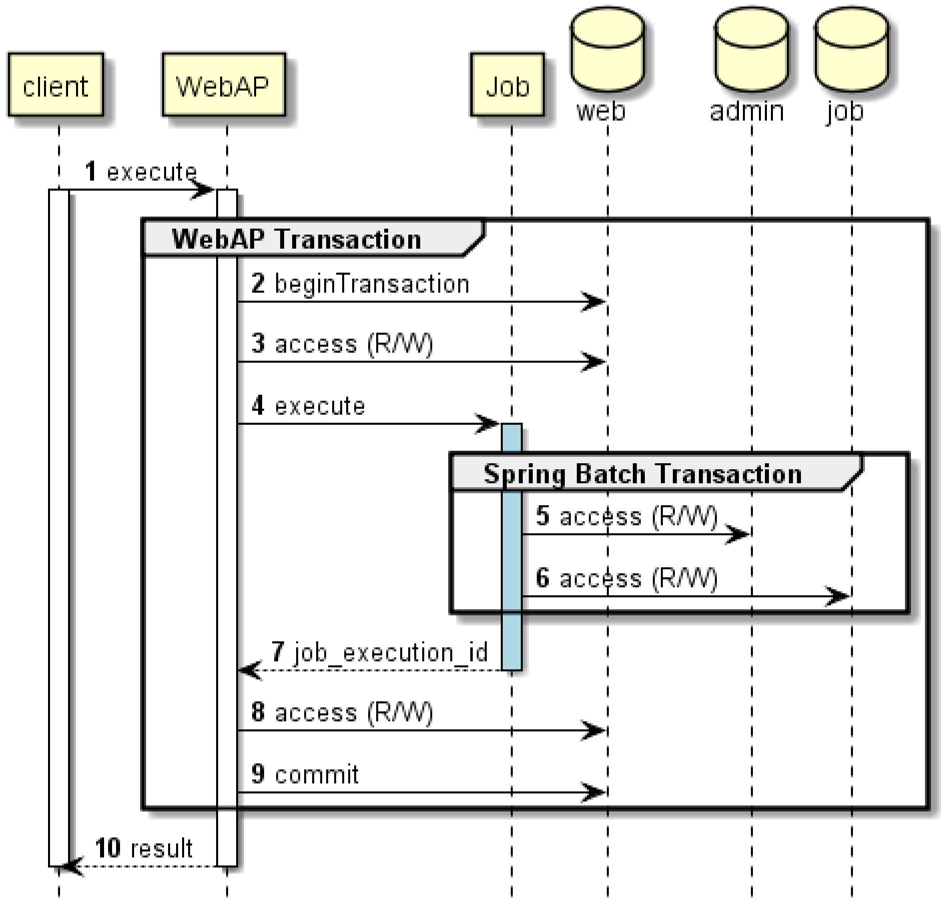
4. `JobRequestPollTask` will commit the transaction other than Spring Batch managed.
5. `JobRequestPollTask` will start a transaction other than Spring Batch managed.
6. `JobRequestPollTask` will update the status of Job-request-table's polling status from INIT to POLLED.
7. `JobRequestPollTask` will commit the transaction other than Spring Batch managed.
8. `JobRequestPollTask` will execute the job.
9. Inside the job, transaction control for DB for Management(`JobRepository`) will be managed by Spring Batch.
10. Inside the job, transaction control for DB for Job will be managed by Spring Batch.
11. job_execution_id is returned to `JobRequestPollTask`
12. `JobRequestPollTask` will start a transaction other than Spring Batch managed.
13. `JobRequestPollTask` will update the status of Job-request-table's polling status from INIT to EXECUTE.
14. `JobRequestPollTask` will commit the transaction other than Spring Batch managed.

> *About Commit at SELECT Issuance*
>
> Some databases may implicitly start transactions when SELECT is issued. Therefore, by explicitly issuing a commit, the transaction is confirmed so that the transaction is clearly distinguished from other transactions and is not influenced.

5.1.2.2.2. About the transaction of WebAP server process

As for processing to resources targeted by WebAP, transaction processing outside Spring Batch managed is performed. Also, regarding exceptions that occurred in the job, since correspondence is completed within the job, it does not affect transactions performed by WebAP.

A simple sequence diagram focusing on transactions is shown in the figure below.



Transaction of WebAP server process

Description of the Sequence Diagram

1. WebAP processing is executed by the request from the client
2. WebAP will start the transaction managed outside of Spring Batch.
3. WebAP reads from and writes to resources in WebAP before job execution.
4. WebAP executes the job.
5. Within a job, Spring Batch carries out transaction management to the Management DB (`JobRepository`).
6. Within a job, Spring Batch carries out transaction management to the Job DB.
7. job_execution_id is returned to WebAP
8. WebAP reads from and writes to resources in WebAP after job execution.
9. WebAP will commit the transaction managed outside of Spring Batch.
10. WebAP returns a response to the client.

5.1.3. How to use

Here, transaction control in one job will be explained separately in the following cases.

- For a single data source
- For multiple data sources

The data source refers to the data storage location (database, file, etc.). A single data source refers to one data source, and multiple data sources refers to two or more data sources.

In the case of processing a single data source, the case of processing database data is representative.
There are some variations in the case of processing multiple data sources as follows.

- multiple databases
- databases and files

5.1.3.1. For a single data source

Transaction control of jobs input / output to one data source will be described.

Below is a sample setting with TERASOLUNA Batch 5.x.

DataSource setting(META-INF/spring/launch-context.xml)

```
<!-- Job-common definitions -->
<bean id="jobDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${jdbc.driver}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}"
      p:maxTotal="10"
      p:minIdle="1"
      p:maxWaitMillis="5000"
      p:defaultAutoCommit="false" />
```

TransactionManager setting(META-INF/spring/launch-context.xml)

```
<!-- (1) -->
<bean id="jobTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
      p:dataSource-ref="jobDataSource"
      p:rollbackOnCommitFailure="true" />
```

No	Description
(1)	Bean definition of TransactionManager. Set `jobDataSource` defined above for the data source. It has been set to roll back if commit fails.

5.1.3.1.1. Implement transaction control

The control method differs depending on the job model and the commit method.

In case of chunk model

In the case of the chunk model, it is an intermediate commit method, leaving transaction control to Spring Batch. Do not control the transaction at all by the user.

Setting sample(job definition)

```
<batch:job id="jobSalesPlan01" job-repository="jobRepository">
    <batch:step id="jobSalesPlan01.step01">
        <batch:tasklet transaction-manager="jobTransactionManager"> <!-- (1) -->
            <batch:chunk reader="detailCSVReader"
                         writer="detailWriter"
                         commit-interval="10" /> <!-- (2) -->
        </batch:tasklet>
    </batch:step>
</batch:job>
```

No	Description
(1)	Set `jobTransactionManager` which is already defined in `transaction-manager` attribute of `<batch:tasklet>` tag. The intermediate commit method transaction is controlled by the transaction manager set here.
(2)	Set chunk size to `commit-interval` attribute. In this sample, commit once for every 10 processing.

For the tasklet model

In the case of the tasklet model, the method of transaction control differs depending on whether the method is single commit method or the intermediate commit method.

single commit method

Spring Batch control transaction.

Setting sample(job definition)

```
<batch:job id="jobSalesPlan01" job-repository="jobRepository">
    <batch:step id="jobSalesPlan01.step01">
        <!-- (1) -->
        <batch:tasklet transaction-manager="jobTransactionManager"
                       ref="salesPlanSingleTranTask" />
    </batch:step>
</batch:job>
```

No	Description
(1)	Set `jobTransactionManager` which is already defined in `transaction-manager` attribute of `<batch:tasklet>` tag. The single commit method transaction is controlled by the transaction manager set here.

intermediate commit method

Control transaction by user.

- If you want to commit in the middle of processing, inject the `TransacitonManager` and operate manually.

Setting sample(job definition)

```
<batch:job id="jobSalesPlan01" job-repository="jobRepository">
    <batch:step id="jobSalesPlan01.step01">
        <!-- (1) -->
        <batch:tasklet transaction-manager="jobTransactionManager"
                       ref="salesPlanChunkTranTask" />
    </batch:step>
</batch:job>
```

Implementation sample

```
@Component()
public class SalesPlanChunkTranTask implements Tasklet {

    @Inject
    ItemStreamReader<SalesPlanDetail> itemReader;

     // (2)
    @Inject
    @Named("jobTransactionManager")
    PlatformTransactionManager transactionManager;

    @Inject
    SalesPlanDetailRepository repository;

    private static final int CHUNK_SIZE = 10;

    @Override
    public RepeatStatus execute(StepContribution contribution,
                                ChunkContext chunkContext) throws Exception {

        DefaultTransactionDefinition definition = new DefaultTransactionDefinition();
        definition.setPropagationBehavior(TransactionDefinition
                .PROPAGATION_REQUIRES_NEW);  // (3)
        TransactionStatus status = null;

        try {
            // omitted

            itemReader.open(executionContext);

            while ((item = itemReader.read()) != null) {
```

```
                if (count % CHUNK_SIZE == 0) {
                    status = transactionManager.getTransaction(definition); // (4)
                }
                count++;

                // omitted

                repository.create(item);
                if (count % CHUNK_SIZE == 0) {
                    transactionManager.commit(status);  // (5)
                }
            }
        } catch (Exception e) {
            logger.error("Exception occurred while reading.", e);
            transactionManager.rollback(status);    // (6)
            throw e;
        } finally {
            if (!status.isCompleted()) {
                transactionManager.commit(status);   // (7)
            }
            itemReader.close();
        }

        return RepeatStatus.FINISHED;
    }
}
```

No	Description
(1)	Set `jobTransactionManager` which is already defined in `transaction-manager` attribute of `<batch:tasklet>` tag. One transaction manager is used by both the framework and the user, but it can be handled independently by the following procedure.
(2)	Inject the transaction manager. In the @Named annotation, specify `jobTransactionManager` to identify the bean to use.
(3)	Specify `PROPAGATION_REQUIRES_NEW` to separate from the framework transaction.
(4)	Start transaction at the beginning of chunk.
(5)	Commit the transaction at the end of the chunk.
(6)	When an exception occurs, roll back the transaction.
(7)	For the last chunk, commit the transaction.

About the Propagation of TransacitonManager

In the tasklet model, transaction control is newly performed within the transaction controlled by Spring Batch. Therefore, it is necessary to set Propagation of Transaciton Manager to **REQUESTS_NEW**.

> *Updating by ItemWriter*
>
> In the above example, although Repository is used, it is possible to update data using ItemWriter. Using ItemWriter has the effect of simplifying implementation, especially FlatFileItemWriter should be used when updating files.

5.1.3.1.2. Note for non-transactional data sources

In the case of files, no transaction setting or operation is necessary.

When using `FlatFileItemWriter`, pseudo transaction control can be performed. This is implemented by delaying the writing to the resource and actually writing out at the commit timing. Normally, when it reaches the chunk size, it outputs chunk data to the actual file, and if an exception occurs, data output of the chunk is not performed.

`FlatFileItemWriter` can switch transaction control on and off with `transactional` property. The default is true and transaction control is enabled. If the `transactional` property is false, `FlatFileItemWriter` will output the data regardless of the transaction.

When adopting the single commit method, it is recommend to set the `transactional` property to false. As described above, since data is written to the resource at the commit timing, until then, all the output data is held in the memory. Therefore, when the amount of data is large, there is a high possibility that the memory becomes insufficient and an error will occur.

On TransacitonManager settings in jobs that only handle files

As in the following job definition, the `transaction-manager` attribute of `batch:tasklet` is mandatory in the xsd schema and can not be omitted.

Excerpt of TransacitonManager setting part

```
<batch:tasklet transaction-manager="jobTransactionManager">
<batch:chunk reader="reader" writer="writer" commit-interval="100" />
</batch:tasklet>
```

Therefore, always specify `jobTransactionManager`. At this time, the following behaviors are obtained.

- If `transactional` is true
  - Synchronize with specified TransacitonManager and output to resource.
- If `transactional` is false
  - Transaction processing of the specified TransacitonManager is idle and it outputs to the resource regardless of the transaction.

At this time, transactions are issued to the resource (eg, database) referred to by `jobTransactionManager`, but since there is no table access, there is no actual damage.

If you do not want to issue transactions to refer to even if it is idle or in case of actual damage, you can use `ResourcelessTransactionManager` which does not require resources.

Sample usage of ResourcelessTransactionManager

```
<batch:tasklet transaction-manager="resourcelessTransactionManager">
<batch:chunk reader="reader" writer="writer" commit-interval="100" />
</batch:tasklet>

<bean id="resourcelessTransactionManager"

class="org.springframework.batch.support.transaction.ResourcelessTrans
actionManager"/>
```

5.1.3.2. For multiple data sources

Transaction control of jobs input / output to multiple data sources will be described. Since consideration points are different between input and output, they will be explained separately.

5.1.3.2.1. Input from multiple data source

When retrieving data from multiple data sources, the data that is the axis of the process and it's additional data should be retrieved separately. Hereinafter, the data as the axis of processing is referred to as the process target record, and the additional data accompanying it is referred to as accompanying data.

Because of the structure of Spring Batch, ItemReader is based on the premise that it retriedves a process target record from one resource. This is the same way of thinking regardless of the type of resource.

1. Retriving process target record
   - Get it by ItemReader.
2. Retriving accompanying data
   - In the accompanying data, it is necessary to select the following retreiving method according to the presence or absence of change to the data and the number of cases. This is not an option, and it may be used in combination.
     - Batch retrieval before step execution
     - Retrieve each time according to the record to be processed

When retrieving all at once before step execution

Implement Listener to do the following and refer to data from the following Step.

- Retrieve data collectively
- Store the information in the bean whose scope is `Job` or `Step`
  - `ExecutionContext` of Spring Batch can be used, but a diffferent class can be created to store data considering the readability and maintainability. For the sake of simplicity, the sample will be explained using `ExecutionContext`.

This method is adopted when reading data that does not depend on data to be processed such as master data. However, even if it is a master data, if there is a large number of items which may give an impact to the memory, retrieving each time should be considered.

Implementation of Listener for collective retrieve

```
@Component
// (1)
public class BranchMasterReadStepListener extends StepExecutionListenerSupport {

    @Inject
    BranchRepository branchRepository;

    @Override
    public void beforeStep(StepExecution stepExecution) {   // (2)

        List<Branch> branches = branchRepository.findAll(); //(3)

        Map<String, Branch> map = branches.stream()
                .collect(Collectors.toMap(Branch::getBranchId,
                        UnaryOperator.identity()));  // (4)

        stepExecution.getExecutionContext().put("branches", map); // (5)
    }
}
```

Definition of Listener for collective retrieve

```
<batch:job id="outputAllCustomerList01" job-repository="jobRepository">
    <batch:step id="outputAllCustomerList01.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader"
                         processor="retrieveBranchFromContextItemProcessor"
                         writer="writer" commit-interval="10"/>
            <batch:listeners>
                <batch:listener ref="branchMasterReadStepListener"/> <!-- (6) -->
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

An example of referring data collectively retrieved by the ItemProcessor of the subsequent step

```
@Component
public class RetrieveBranchFromContextItemProcessor implements
        ItemProcessor<Customer, CustomerWithBranch> {

    private Map<String, Branch> branches;

    @BeforeStep       // (7)
    @SuppressWarnings("unchecked")
    public void beforeStep(StepExecution stepExecution) {
        branches = (Map<String, Branch>) stepExecution.getExecutionContext()
                .get("branches"); // (8)
    }

    @Override
    public CustomerWithBranch process(Customer item) throws Exception {
        CustomerWithBranch newItem = new CustomerWithBranch(item);
        newItem.setBranch(branches.get(item.getChargeBranchId()));    // (9)
        return newItem;
    }
}
```

No	Description
(1)	Implement `StepExecutionListener` interface. In order to simplify the implementation here, it is an extension from `StepExecutionListenerSupport` which implements the `StepExecutionListener` interface.
(2)	Implement the `beforeStep` method to get data before step execution.
(3)	Implement processing to retrieve master data.
(4)	Convert from List type to Map type so that it can be used easily in subsequent processing.
(5)	Set the acquired master data in the context of the step as `branches`.
(6)	Register the created Listener to the target job.
(7)	In order to acquire master data before step execution of ItemProcessor, set up Listener with @BeforeStep annotation.
(8)	In the method given the @BeforeStep annotation, obtain the master data set in (5) from the context of the step.
(9)	In the process method of ItemProcessor, data is retrieved from the master data.

Object to store in context

The object to be stored in the context(`ExecutionContext`) must be a class that implements `java.io.Serializable`. This is because `ExecutionContext` is stored in `JobRepository`.

Retrieving each time according to the record to be processed

Apart from ItemProcessor of business processing, it retrieves by ItemProcessor designated just for retrieving every time. This simplifies processing of each ItemProcessor.

1. Define ItemProcessor designated just for retrieving every time, and separate it from business process.
   - At this time, use MyBatis as it is when accessing the table.
2. Concatenate multiple ItemProcessors using CompositeItemProcessor.
   - Note that ItemProcessor is processed in the order specified in the delegates attribute.

Sample implementation of ItemProcessor designated just for retrieving every time

```
@Component
public class RetrieveBranchFromRepositoryItemProcessor implements
        ItemProcessor<Customer, CustomerWithBranch> {

    @Inject
    BranchRepository branchRepository;  // (1)

    @Override
    public CustomerWithBranch process(Customer item) throws Exception {
        CustomerWithBranch newItem = new CustomerWithBranch(item);
        newItem.setBranch(branchRepository.findOne(
                item.getChargeBranchId())); // (2)
        return newItem; // (3)
    }
}
```

Definition sample of ItemProcessor designated just for retrieving every time and ItemProcessor for business process

```
<bean id="compositeItemProcessor"
      class="org.springframework.batch.item.support.CompositeItemProcessor">
    <property name="delegates">
        <list>
            <ref bean="retrieveBranchFromRepositoryItemProcessor"/> <!-- (4) -->
            <ref bean="businessLogicItemProcessor"/>  <!-- (5) -->
        </list>
    </property>
</bean>
```

No	Description
(1)	Inject Repository for retrieving every time using MyBatis.
(2)	Accompaniment data is retrieved from the Repository for input data(process target record).

No	Description
(3)	Return data with processing target record and accompanying data together. Notice that this data will be the input data to the next ItemProcessor.
(4)	Set ItemProcessor for retrieving every time.
(5)	Set ItemProcessor for business logic.

5.1.3.2.2. Output to multiple data sources(multiple steps)

Process multiple data sources throughout the job by dividing the steps for each data source and processing a single data source at each step.

- Data processed at the first step is stored in a table, and at the second step, it is outputted to a file.
- Although each step is simple and easy to recover, there is a possibility that it may be troublesome twice.
  - As a result, in the case of causing the following harmful effects, consider processing multiple data sources in one step.
    - Processing time increases
    - Business logic becomes redundant

5.1.3.2.3. Output to multiple data sources(single step)

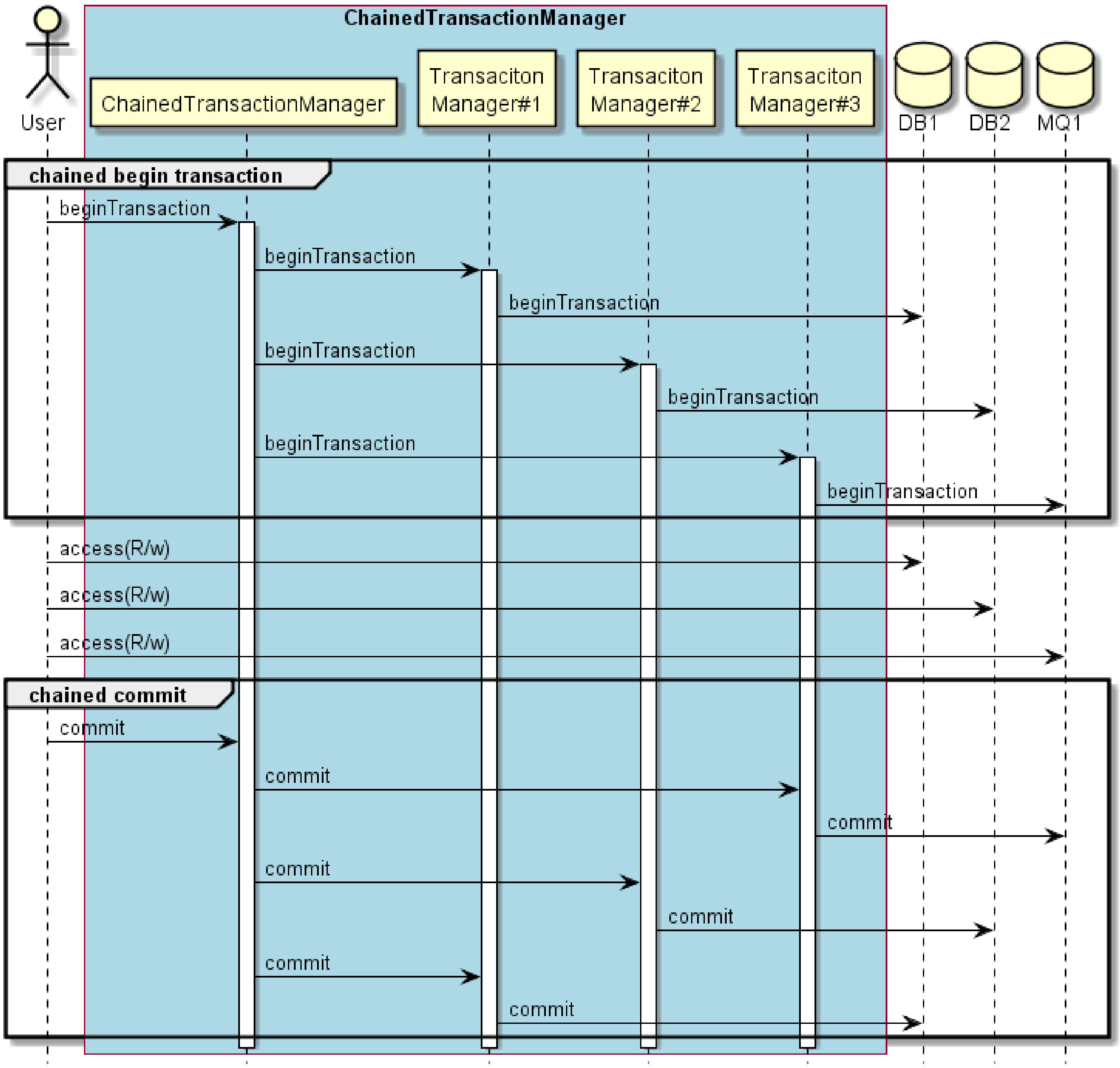
Generally, when transactions for a plurality of data sources are combined into one, a distributed transaction based on 2 phase-commit is used. However, it is also known that there are the following disadvantages.

- Middleware must be compatible with distributed transaction API such as XAResource, and special setting based on it is required
- In standalone Java like a batch program, you need to add a JTA implementation library for distributed transactions
- Recovery in case of failure is difficult

Although it is possible to utilize distributed transactions also in Spring Batch, the method using global transaction by JTA requires performance overhead due to the characteristics of the protocol. As a method to process multiple data sources collectively more easily, **Best Efforts 1PC pattern** is recommended.

What is Best Efforts 1PC pattern

Briefly, it refers to the technique of handling multiple data sources as local transactions and issuing sequential commits at the same timing. The conceptual diagram is shown in the figure below.



Conceptual diagram of Best Efforts 1PC pattern

Description of figure

1. The user instructs `ChainedTransactionManager` to start the transaction.
2. `ChainedTransactionManager` starts a transaction sequentially with registered transaction managers.
3. The user performs transactional operations on each resource.
4. The user instructs `ChainedTransactionManager` to commit.
5. `ChainedTransactionManager` issues sequential commits on registered transaction managers.
   - Commit(or roll back) in reverse order of transaction start

Since this method is not a distributed transaction, there is a possibility that data consistency may not be maintained if a failure(exception) occurs at commit / rollback in the second and subsequent transaction managers. Therefore, although it is necessary to design a recovery method when a failure occurs at a transaction boundary, there is an effect that the recovery frequency can be reduced and the recovery procedure can be simplified.

When processing multiple transactional resources at the same time

Use it on cases such as when processing multiple databases simultaneously, when processing databases and MQ, and so on.

Process as 1 phase-commit by defining multiple transaction managers as one using `ChainedTransactionManager` as follows. Note that `ChainedTransactionManager` is a class provided by Spring Data.

pom.xml

```
<dependencies>
    <!-- omitted -->
    <!-- (1) -->
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-commons</artifactId>
    </dependency>
<dependencies>
```

Sample usage of chainedTransactionManager

```
<!-- Chained Transaction Manager -->
<!-- (2) -->
<bean id="chainedTransactionManager"
      class="org.springframework.data.transaction.ChainedTransactionManager">
   <constructor-arg>
        <!-- (3) -->
        <list>
            <ref bean="transactionManager1"/>
            <ref bean="transactionManager2"/>
        </list>
    </constructor-arg>
</bean>

<batch:job id="jobSalesPlan01" job-repository="jobRepository">
    <batch:step id="jobSalesPlan01.step01">
        <!-- (4) -->
        <batch:tasklet transaction-manager="chainedTransactionManager">
            <!-- omitted -->
        </batch:tasklet>
    </batch:step>
</batch:job>
```

No	Description
(1)	Add a dependency to use `ChainedTransactionManager`.

No	Description
(2)	Define the bean of `ChainedTransactionManager`.
(3)	Define multiple transaction managers that you want to summarize in a list.
(4)	Specify the bean ID defined in (1) for the transaction manager used by the job.

When processing transactional and nontransactional resources simultaneously

This method is used when processing databases and files at the same time.

For database it is the same as For a single data source.

For files, setting FlatFileItemWriter’s `transactional` property to true provides the same effect as the "Best Efforts 1PC pattern" described above.
For details, refer to Note for non-transactional data sources.

This setting delays writing to the file until just before committing the transaction of the database, so it is easy to synchronize with the two data sources. However, even in this case, if an error occurs during file output processing after committing to the database, there is a possibility that data consistency can not be maintained, It is necessary to design a recovery method.

5.1.3.3. Notes on intermediate method commit

Although it is deprecated, when processing data is skipped in ItemWriter, the chunk size setting value is forcibly changed. Note that this has a very big impact on transactions. Refer to Skip for details.

5.2. Database Access

5.2.1. Overview

MyBatis3 (hereafter, called [MyBatis]) is used for database access in TERASOLUNA Batch 5.x. Please refer below TERASOLUNA Server 5.x Development Guideline for basic usage of database access using MyBatis.

- Database Access (Common)
- Database Access (MyBatis3)

This chapter mainly explain how to use database access as TERASOLUNA Batch 5.x specifically.

Since this function works differently for chunk model and tasket model, respective explanations are given.

5.2.2. How to use

Explain how to use database access as TERASOLUNA Batch 5.x.

There are following 2 ways to use database access in TERASOLUNA Batch 5.x.
Please choose them based on the components accessing the database.

1. Use ItemReader and ItemWriter for MyBatis.
   - For Input/Output by using database access as chunk model.
     - org.mybatis.spring.batch.MyBatisCursorItemReader
     - org.mybatis.spring.batch.MyBatisBatchItemWriter
2. Use Mapper interface
   - For bussiness logic processing as chunk model.
     - With ItemProcessor implementation.
   - For whole database access as tasklet model.
     - With Tasklet implementation.

5.2.2.1. Common Settings

Explain common settings required for database access.

1. DataSource Setting
2. MyBatis Setting
3. Mapper XML definition
4. MyBatis-Spring setting

5.2.2.1.1. DataSource Setting

It assumes two data sources in TERASOLUNA Batch 5.x. Show 2 default data sources in `launch-context.xml`.

Data source list

Data source name	Description
`adminDataSource`	Data source used by Spring Batch and TERASOLUNA Batch 5.x It is used in JobRepository and Asynchronous execution(DB polling)
`jobDataSource`	Data source used by job

Show the property of connection information and launch-context.xml below.

Set these settings according to the user's environment.

resources\META-INF\spring\launch-context.xml

```
<!-- (1) -->
<bean id="adminDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${admin.jdbc.driver}"
      p:url="${admin.jdbc.url}"
      p:username="${admin.jdbc.username}"
      p:password="${admin.jdbc.password}"
      p:maxTotal="10"
      p:minIdle="1"
      p:maxWaitMillis="5000"
      p:defaultAutoCommit="false"/>

<!-- (2) -->
<bean id="jobDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${jdbc.driver}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}"
      p:maxTotal="10"
      p:minIdle="1"
      p:maxWaitMillis="5000"
      p:defaultAutoCommit="false" />
```

batch-application.properties

```
# (3)
# Admin DataSource settings.
admin.jdbc.driver=org.h2.Driver
admin.jdbc.url=jdbc:h2:mem:batch;DB_CLOSE_DELAY=-1
admin.jdbc.username=sa
admin.jdbc.password=

# (4)
# Job DataSource settings.
jdbc.driver=org.postgresql.Driver
jdbc.url=jdbc:postgresql://localhost:5432/postgres
jdbc.username=postgres
jdbc.password=postgres
```

Description

Sr. No.	Description
(1)	`adminDataSource` definition. Connection information of (3) is set.
(2)	`jobDataSource` definition. Connection information of (4) is set.
(3)	Connection information to database used by `adminDataSource` H2 is used in this example.
(4)	Connection information to database used by `jobDataSource` PostgreSQL is used in this example.

5.2.2.1.2. MyBatis Setting

Important points for setting MyBatis on TERASOLUNA Batch 5.x.

One of the important points in implementing batch processing is "to efficiently process large amounts of data with certain resources"
Explain the setting.

- `fetchSize`
  - In general batch processing, it is mandatory to specify the appropriate `fetchSize` for the JDBC driver to reduce the communication cost of processing large amounts of data. `fetchSize` is a parameter that sets the number of data to be acquired by one communication between the JDBC driver and the database. It is desirable to set this value as large as possible. However, if it is too large, it presses memory. So please be careful. user has to tune the parameter.
  - In MyBatis, user can set `defaultFetchSize` as a common setting for all queries, and can override it with `fetchSize` setting for each query.
- `executorType`
  - In general batch processing, the same SQL is executed within the same transaction for the number of `total data count/fetchSize`. At this time, it is possible to process efficiently by

reusing a statement instead of creating it each time.

- In the MyBatis setting, it can reuse statements by setting `REUSE` in `defaultExecutorType` and contributes to improved processing throughput.
- When updating a large amount of data at once, performance improvement can be expected by using batch update of JDBC.
  Therefore, `SqlSessionTemplate` used in `MyBatisBatchItemWriter`
  is set to `BATCH` (not `REUSE`) in `executorType`.

In TERASOLUNA Batch 5.x, two different `ExecutorType` exists at the same time. It is assumed that it is often implemented by one `ExecutorType`, but special attention is required when using them together. The detail will be explained in Database Access other than ItemReader・ItemWriter.

> *Other parameter of MyBatis*
>
> For other parameters, refer to the following links and make settings that match the application characteristics.
> http://www.mybatis.org/mybatis-3/configuration.html

Show the default setting below.

META-INF/spring/launch-context.xml

```xml
<bean id="jobSqlSessionFactory"
      class="org.mybatis.spring.SqlSessionFactoryBean"
      p:dataSource-ref="jobDataSource">
    <!-- (1) -->
    <property name="configuration">
        <bean class="org.apache.ibatis.session.Configuration"
              p:localCacheScope="STATEMENT"
              p:lazyLoadingEnabled="true"
              p:aggressiveLazyLoading="false"
              p:defaultFetchSize="1000"
              p:defaultExecutorType="REUSE"/>
    </property>
</bean>

<!-- (2) -->
<bean id="batchModeSqlSessionTemplate"
      class="org.mybatis.spring.SqlSessionTemplate"
      c:sqlSessionFactory-ref="jobSqlSessionFactory"
      c:executorType="BATCH"/>
```

Description

Sr. No.	Description
(1)	Various setting of MyBatis fetchSize is set to 1000 by default.
(2)	For `MyBatisBatchItemWriter`, `executorType` defines `SqlSessionTemplate` of `BATCH`.

For the definition of SqlSessionFactory using adminDataSource

When performing synchronous execution, `SqlSessionFactory` using adminDataSource is unnecessary and is not defined. When performing Asynchronous execution(DB polling), it is defined in `META-INF/spring/async-batch-daemon.xml` to access the Job-request-table.

META-INF/spring/async-batch-daemon.xml

```
<bean id="adminSqlSessionFactory"
      class="org.mybatis.spring.SqlSessionFactoryBean"
      p:dataSource-ref="adminDataSource" >
    <property name="configuration">
        <bean class="org.apache.ibatis.session.Configuration"
              p:localCacheScope="STATEMENT"
              p:lazyLoadingEnabled="true"
              p:aggressiveLazyLoading="false"
              p:defaultFetchSize="1000"
              p:defaultExecutorType="REUSE"/>
    </property>
</bean>
```

5.2.2.1.3. Mapper XML definition

Please refer to Implementation of database access process in TERASOLUNA Server 5.x Development Guideline, because there are no specific description about TERASOLUNA Batch 5.x.

5.2.2.1.4. MyBatis-Spring setting

When using ItemReader and ItemWriter provided by MyBatis-Spring, it is necessary to set Mapper XML used in Mapper's Config.

As the setting method, there are following 2 methods.

1. Register Mapper XML to be used for all jobs as a common setting.
   - All Mapper XML has to be described in `META-INF/spring/launch-context.xml`.
2. Register Mapper XML to be used for each job as individual setting.
   - Mapper XML required by each job has to be described in bean definition under `META-INF/jobs/`

If common settings are made, the following adverse effects arise because not only Mapper XML of jobs executed, but also Mapper XML used by other jobs are also read when executing synchronous execution.

- It takes time to start the job
- Consumption of memory resources increases

To avoid it, TERASOLUNA Batch 5.x adopts a setting method that specifies only Mapper XML that the job requires for each job definition as individual setting.

For the basic setting method, please refer to MyBatis-Spring settings in TERASOLUNA Server 5.x Development Guideline.

In TERASOLUNA Batch 5.x, since multiple `SqlSessionFactory` and `SqlSessionTemplate` are defined, it is necessary to explicitly specify which one to use.
Basically, specify `jobSqlSessionFactory`

Show setting example below.

META-INF/jobs/common/jobCustomerList01.xml

```
<!-- (1) -->
<mybatis:scan
    base-package="org.terasoluna.batch.functionaltest.app.repository.mst"
    factory-ref="jobSqlSessionFactory"/>
```

Description

Sr. No.	Description
(1)	Set `jobSqlSessionFactory` in `factory-ref` attribute of `<mybatis:scan>`

5.2.2.2. Database access with ItemReader

Explain Database access with ItemReader here.

5.2.2.2.1. ItemReader of MyBatis

MyBatis-Spring provides the following two ItemReader.

- `org.mybatis.spring.batch.MyBatisCursorItemReader`
- `org.mybatis.spring.batch.MyBatisPagingItemReader`

`MyBatisPagingItemReader` is an ItemReader that uses the mechanism described in Pagination search for Entity (SQL refinement method) of TERASOLUNA Server 5.x Development Guideline
Since SQL is issued again after acquiring a certain number of cases, there is a possibility that data consistency may not be maintained. Therefore, it is dangerous to use it in batch processing, so TERASOLUNA Batch 5.x does not use it in principle.
TERASOLUNA Batch 5.x uses only `MyBatisCursorItemReader`.

In TERASOLUNA Batch 5.x, as explained in MyBatis-Spring setting, It adopts a method of dynamically registering Mapper XML with `mybatis:scan`. Therefore, it is necessary to prepare an interface corresponding to Mapper XML. For details, please refer to Implementation of database access process in TERASOLUNA Server 5.x Development Guideline.

Show an example of usage of `MyBatisCursorItemReader` below.

META-INF/jobs/common/jobCustomerList01.xml

```xml
<!-- (1) -->
<mybatis:scan
    base-package="org.terasoluna.batch.functionaltest.app.repository.mst"
    factory-ref="jobSqlSessionFactory"/>

<!-- (2) (3) (4) -->
<bean id="reader"
      class="org.mybatis.spring.batch.MyBatisCursorItemReader" scope="step"

p:queryId="org.terasoluna.batch.functionaltest.app.repository.mst.CustomerRepository.f
indAll"
      p:sqlSessionFactory-ref="jobSqlSessionFactory"/>
```

org/terasoluna/batch/functionaltest/app/repository/mst/CustomerRepository.xml

```xml
<!-- (5) -->
<mapper
namespace="org.terasoluna.batch.functionaltest.app.repository.mst.CustomerRepository">

    <!-- (6) -->
    <select id="findAll"
            resultType="org.terasoluna.batch.functionaltest.app.model.mst.Customer">
        <![CDATA[
        SELECT
            customer_id AS customerId,
            customer_name AS customerName,
            customer_address AS customerAddress,
            customer_tel AS customerTel,
            charge_branch_id AS chargeBranchId,
            create_date AS createDate,
            update_date AS updateDate
        FROM
            customer_mst
        ORDER by
            charge_branch_id ASC, customer_id ASC
        ]]>
    </select>

    <!-- omitted -->
</mapper>
```

org.terasoluna.batch.functionaltest.app.repository.mst.CustomerRepository

```
public interface CustomerRepository {
    // (7)
    List<Customer> findAll();

    // omitted
}
```

Description

Sr. No.	Description
(1)	Register Mapper XML.
(2)	Define `MyBatisCursorItemReader`.
(3)	Specify the SQL ID defined in (6) with `namespace` + `<method name>` of (5) to the property of `queryId`.
(4)	Specify `SqlSessionFactory` of the database to be accessed in `sqlSessionFactory` property.
(5)	Define Mapper XML. Match the value of namespace with the FQCN of the interface.
(6)	Define SQL.
(7)	Define the method corresponding to the SQL ID defined in (6) for the interface.

5.2.2.3. Database Access with ItemWriter

Explain database access with ItemWriter in here.

5.2.2.3.1. ItemWriter of MyBatis

MyBatis-Spring provides only one following ItemWriter.

- `org.mybatis.spring.batch.MyBatisBatchItemWriter`

The basic setting is the same as ItemReader of MyBatis. `MyBatisBatchItemWriter` needs to specify `batchModeSqlSessionTemplate` described in MyBatis Setting.

Show an example definition of `MyBatisBatchItemWriter` below.

META-INF/jobs/common/jobSalesPlan01.xml

```xml
<!-- (1) -->
<mybatis:scan
    base-package="org.terasoluna.batch.functionaltest.app.repository.plan"
    factory-ref="jobSqlSessionFactory"/>

<!-- (2) (3) (4) -->
<bean id="detailWriter" class="org.mybatis.spring.batch.MyBatisBatchItemWriter"

p:statementId="org.terasoluna.batch.functionaltest.app.repository.plan.SalesPlanDetail
Repository.create"
      p:sqlSessionTemplate="batchModeSqlSessionTemplate"/>

<!-- omitted -->
```

org/terasoluna/batch/functionaltest/app/repository/plan/SalesPlanDetailRepository.xml

```xml
<!-- (5) -->
<mapper
namespace="org.terasoluna.batch.functionaltest.app.repository.plan.SalesPlanDetailRepo
sitory">

    <!-- (6) -->
    <insert id="create"

parameterType="org.terasoluna.batch.functionaltest.app.model.plan.SalesPlanDetail">
        <![CDATA[
        INSERT INTO
            sales_plan_detail(branch_id, year, month, customer_id, amount)
        VALUES (
            #{branchId}, #{year}, #{month}, #{customerId}, #{amount}
        )
        ]]>
    </insert>

    <!-- omitted -->
</mapper>
```

org.terasoluna.batch.functionaltest.app.repository.plan.SalesPlanDetailRepository

```java
public interface SalesPlanDetailRepository {

    // (7)
    void create(SalesPlanDetail salesPlanDetail);

    // omitted
}
```

Description

Sr. No.	Description
(1)	Register Mapper XML.
(2)	Define `MyBatisBatchItemWriter`.
(3)	Specify the SQL ID defined in (6) with `namespace` + `<method name>` of (5) to the property of `statementId`.
(4)	Specify `SessionTemplate` of the database to be accessed in `sqlSessionTemplate` property. The specified `SessionTemplate` is mandatory that `executorType` is set to `BATCH`.
(5)	Define Mapper XML. Match the value of namespace with the FQCN of the interface.
(6)	Define SQL.
(7)	Define the method corresponding to the SQL ID defined in (6) for the interface.

5.2.2.4. Database Access other than ItemReader · ItemWriter

Explain database access except for ItemReader · ItemWriter.

To access the database except for ItemReader · ItemWriter, use the Mapper interface. In using the Mapper interface, TERASOLUNA Batch 5.x has the following restrictions.

The available points of Mapper interface.

Process	ItemProcessor	Tasklet	Listner
Reference	Available	Available	Available
Update	Conditionally available	Available	Unavailable

Restrictions in ItemProcessor

There is a restriction that it should not be executed with two or more `ExecutorType` within the same transaction in MyBatis.
If "use `MyBatisBatchItemWriter` for ItemWriter" and "use ItemProcessor to update and reference the Mapper interface" are satisfied at the same time, it conflicts with this restriction.
To avoid this restriction, database is accessed by using Mapper interface that `ExecutorType` is `BATCH` in ItemProcessor.
In addition, `MyBatisBatchItemWriter` checks whether it is SQL issued by itself with the status check after executing SQL but naturally it can not manage SQL execution by ItemProcessor and an error will occur.
Therefore, if `MyBatisBatchItemWriter` is used, updating with the Mapper interface will not be possible and only reference.

> It can set to invalidate the error check of `MyBatisBatchItemWriter`, but the setting is prohibited because there is a possibility that unexpected behavior may occur.

Restrictions in Tasklet

In Tasklet, since it is basic to use the Mapper interface, there is no influence like ItemProcessor.

It is possible to use `MyBatisBatchItemWriter` by Inject, but in that case Mapper interface itself can be processed with `BATCH` setting. In other words, there is basically no need to use `MyBatisBatchItemWriter` by Inject.

Restrictions in Listener

Even at the listener, the same restriction as that of ItemProcessor is established. In addition, for listeners, use cases requiring updates are difficult to think. Therefore, update processing is prohibited at the listner.

> *Replacement of update processing assumed by the listner*
>
> *Job state management*
>
> It is done by JobRepository of Spring Batch
>
> *Log output to database*
>
> It should be done in the log Appender. It is necessary to manage it separately from the transaction of the job.

5.2.2.4.1. Database access with ItemProcessor

Show an example of database access with ItemProcessor.

Implementation example with ItemProcessor

```java
@Component
public class UpdateItemFromDBProcessor implements
        ItemProcessor<SalesPerformanceDetail, SalesPlanDetail> {

    // (1)
    @Inject
    CustomerRepository customerRepository;

    @Override
    public SalesPlanDetail process(SalesPerformanceDetail readItem) throws Exception {

        // (2)
        Customer customer = customerRepository.findOne(readItem.getCustomerId());

        // (3)
        SalesPlanDetail writeItem = new SalesPlanDetail();
        writeItem.setBranchId(customer.getChargeBranchId());
        writeItem.setYear(readItem.getYear());
        writeItem.setMonth(readItem.getMonth());
        writeItem.setCustomerId(readItem.getCustomerId());
        writeItem.setAmount(readItem.getAmount());
        return writeItem;
    }
}
```

Bean definition

```
<!-- (2) -->
<mybatis:scan
        base-package="org.terasoluna.batch.functionaltest.app.repository"
        template-ref="batchModeSqlSessionTemplate"/>

<bean id="reader" class="org.mybatis.spring.batch.MyBatisCursorItemReader"

p:queryId"org.terasoluna.batch.functionaltest.app.repository.performance.SalesPerforma
nceDetailRepository.findAll"
      p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

<!-- (3) -->
<bean id="writer" class="org.mybatis.spring.batch.MyBatisBatchItemWriter"

p:statementId="org.terasoluna.batch.functionaltest.app.repository.plan.SalesPlanDetail
Repository.create"
      p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"/>

<batch:job id="DBAccessByItemProcessor" job-repository="jobRepository">
    <batch:step id="DBAccessByItemProcessor.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <!-- (4) -->
            <batch:chunk reader="reader"
                         processor="updateItemFromDBProcessor"
                         writer="writer" commit-interval="10"/>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Mapper interface and Mapper XML are omitted.

Description

Sr. No.	Description
(1)	Inject Mapper interface.
(2)	Register Mapper XML. By specifying `batchModeSqlSessionTemplate` set as `BATCH` in `template-ref` attribute, database access with ItemProcessor is `BATCH`. if you set `factory-ref="jobSqlSessionFactory"`, it conflicts with the above restriction and an exception is thrown when `MyBatisBatchItemWriter` is executed.
(3)	Define `MyBatisBatchItemWriter` Specify `batchModeSqlSessionTemplate` set as `BATCH` in `sqlSessionTemplate` property.
(4)	Set ItemProcessor that injected Mapper interface.

Supplement of MyBatisCursorItemReader setting

Different `ExecutorType` can be used for MyBatisCursorItemReader and MyBatisBatchItemWriter like the definition example below. This is because the opening of the resource by MyBatisCursorItemReader is done before the start of the transaction.

```
<bean id="reader"
class="org.mybatis.spring.batch.MyBatisCursorItemReader"
      p:queryId="xxx"
      p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

<bean id="writer"
class="org.mybatis.spring.batch.MyBatisBatchItemWriter"
      p:statementId="yyy"
      p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"/>
```

5.2.2.4.2. Database Access with Tasklet

Show an example of database access in Tasklet.

Implementation example with Tasklet

```java
@Component
public class OptimisticLockTasklet implements Tasklet {

    // (1)
    @Inject
    ExclusiveControlRepository repository;

    // omitted

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception {

        Branch branch = repository.branchFindOne(branchId); // (2)
        ExclusiveBranch exclusiveBranch = new ExclusiveBranch();

        exclusiveBranch.setBranchId(branch.getBranchId());
        exclusiveBranch.setBranchName(branch.getBranchName() + " - " + identifier);
        exclusiveBranch.setBranchAddress(branch.getBranchAddress() + " - " +
identifier);
        exclusiveBranch.setBranchTel(branch.getBranchTel());
        exclusiveBranch.setCreateDate(branch.getUpdateDate());
        exclusiveBranch.setUpdateDate(new Timestamp(System.currentTimeMillis()));
        exclusiveBranch.setOldBranchName(branch.getBranchName());

        int result = repository.branchExclusiveUpdate(exclusiveBranch); // (3)

        return RepeatStatus.FINISHED;
    }
}
```

Bean definition

```xml
<!-- (4) -->
<mybatis:scan
        base-
package="org.terasoluna.batch.functionaltest.ch05.exclusivecontrol.repository"
        factory-ref="jobSqlSessionFactory"/>

<batch:job id="taskletOptimisticLockCheckJob" job-repository="jobRepository">
    <batch:step id="taskletOptimisticLockCheckJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager"
                       ref="optimisticLockTasklet"> <!-- (5) -->
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Mapper interface and Mapper XML are omitted.

Description

Sr. No.	Description
(1)	Inject Mapper interface.
(2)	Execute the search process with the Mapper interface.
(3)	Execute the update process with the Mapper interface.
(4)	Register Mapper XML Specify `jobSqlSessionFactory` set as `REUSE` in `factory-ref` attribute.
(5)	Inject Mapper interface and set Tasklet.

> *Use batchModeSqlSessionTemplate*
>
> If there are many updating processes with the tasklet model, set `batchModeSqlSessionTemplate` in `factory-ref` attribute. As a result, batch update processing is performed, so performance improvement can be expected. However, be aware that executing batch updates requires `flush` explicitly. For details, please refer to Precautions when using batch mode Repository.

5.2.2.4.3. Database Access with Listener

Database access with listener is often linked with other components. Depending on the listener to be used and the implementation method, It is necessary to prepare additional mechanism to hand over to other components.

Show an example in which StepExecutionListener acquires data before step execution and uses the data acquired by ItemProcessor.

Implementation example with Listener

```
public class CacheSetListener extends StepExecutionListenerSupport {

    // (1)
    @Inject
    CustomerRepository customerRepository;

    // (2)
    @Inject
    CustomerCache cache;

    @Override
    public void beforeStep(StepExecution stepExecution) {
        // (3)
        customerRepository.findAll().forEach(customer ->
                cache.addCustomer(customer.getCustomerId(), customer));
    }
}
```

Application example with ItemProcessor

```
@Component
public class UpdateItemFromCacheProcessor implements
        ItemProcessor<SalesPerformanceDetail, SalesPlanDetail> {

    // (4)
    @Inject
    CustomerCache cache;

    @Override
    public SalesPlanDetail process(SalesPerformanceDetail readItem) throws Exception {
        Customer customer = cache.getCustomer(readItem.getCustomerId());  // (5)

        SalesPlanDetail writeItem = new SalesPlanDetail();

        // omitted
        writerItem.setCustomerName(customer.getCustomerName); // (6)

        return writeItem;
    }
}
```

Cache class

```
// (7)
@Component
public class CustomerCache {

    Map<String, Customer> customerMap = new HashMap<>();

    public Customer getCustomer(String customerId) {
        return customerMap.get(customerId);
    }

    public void addCustomer(String id, Customer customer) {
        customerMap.put(id, customer);
    }
}
```

Bean definition

```
<!-- omitted -->

<!-- (8) -->
<mybatis:scan
        base-package="org.terasoluna.batch.functionaltest.app.repository"
        template-ref="batchModeSqlSessionTemplate"/>
<!-- (9) -->
<bean id="cacheSetListener"
      class="org.terasoluna.batch.functionaltest.ch05.dbaccess.CacheSetListener"/>

<!-- omitted -->

<batch:job id="DBAccessByItemListener" job-repository="jobRepository">
    <batch:step id="DBAccessByItemListener.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader"
                         processor="updateItemFromCacheProcessor"
                         writer="writer" commit-interval="10"/> <!-- (10) -->
            <!-- (11) -->
            <batch:listeners>
                <batch:listener ref="cacheSetListener"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Description

Sr. No.	Description
(1)	Inject Mapper interface.
(2)	Inject a bean for caching data acquired from the Mapper interface.
(3)	Get data from the Mapper interface and cache it at the listener. In this case, I/O is reduced and processing efficiency is improved by creating a cache before step execution with `StepExecutionListener#beforeStep` and referring to the cache in the subsequent processing.
(4)	Inject the same bean as the cache set in (2).
(5)	Get corresponding data from the cache.
(6)	Reflect the data from the cache in the update data.
(7)	Implement the cache class as a component. The Bean scope is `singleton` in here. Please set according to job.
(8)	Register Mapper XML Specify `batchModeSqlSessionTemplate` set as `BATCH` in `template-ref` attribute.
(9)	Define the listener that uses the Mapper interface.

(10)	Specify ItemProcessor that uses cache.
(11)	Register the listener defined in (9).

Using SqlSessionFactory with the Listener

In the above example, `batchModeSqlSessionTemplate` is set, but `jobSqlSessionFactory` also can be set.

For listeners that run outside the scope of chunks, since it is processed outside the transaction, setting `jobSqlSessionFactory` does not matter.

5.3. File Access

5.3.1. Overview

This chapter describes how to input and output files.

The usage method of this function is same in the chunk model as well as tasklet model.

5.3.1.1. Type of File which can be handled

Type of File which can be handled

The type of files that can be handled with TERASOLUNA Batch 5.x are ones decribed as below.
This is the same for which Spring Batch can handle.

- Flat File
- XML

Here it will explain how to handle flat file first, and then explain about XML in How To Extend.

First, show the types of Flat File which can be used with TERASOLUNA Batch 5.x. Each row inside the flat file will be called `record`, and type of file is determined by the record's format.

Record Format

Format	Overview
Variable-length Record	A record format which each items are separated by a delimiter, such as CSV and TSF. Each item's length can be variable.
Fixed-length Record	A record format which each items are separeted by the items length(bytes). Each item's length are fixed.
Single String Record	1 Record will be handled as 1 String item.

File Structure which can be handled

The basic structure for Flat File is constructed by these 2 points.

- Record Division
- Record Format

Elements to construct format of Flat File

Element	Overview
Record Division	A division will indicate the type of record, such as Header Record, Data Record, and Trailer Record. Details will be described later.

Element	Overview
Record Format	The format will have informations of the record such as how many rows there is for Header, Data, and Trailer, how many times eace record will repeat, and so on. There is also Single Format and Multi Format.Details will be described later.

With TERASOLUNA Batch 5.x, Flat File with Single Format of Multi Format which includes each record division can be handles.

Here it willl explain about the record division and the record formats.

The overview of each record devision is explained as below.

Characteristic of each Record Division

Record Division	Overview
Header Record	A record that is mentioned at the beginning of the file(data part). It has items such as field names, common matters of the file, and summary of the data part.
Data Record	It is a record having data to be processed as a main object of the file.
Trailer/Footer Record	A record that is mentioned at the end of the file if the file(data part). It has items such as common matters of the file and summary of the data part. For Single Format file, it is called a Fotter Record.
Footer/End Record	A record that is mentioned at the end of the file if the file is a Multi Format. It has items such as common matters of the file and summary of the data part.

About the field that indicates the record division

A flat file having a header record or a trailer record may have a field indicating a record division.
In TERASOLUNA Batch 5.x, especially in the processing of multi-format files, the record division field is utilized, for example when different processing is performed for each record division.
Refer to Multi format for the implementation when selecting the processing to be executed by record classification.

About the name of file format

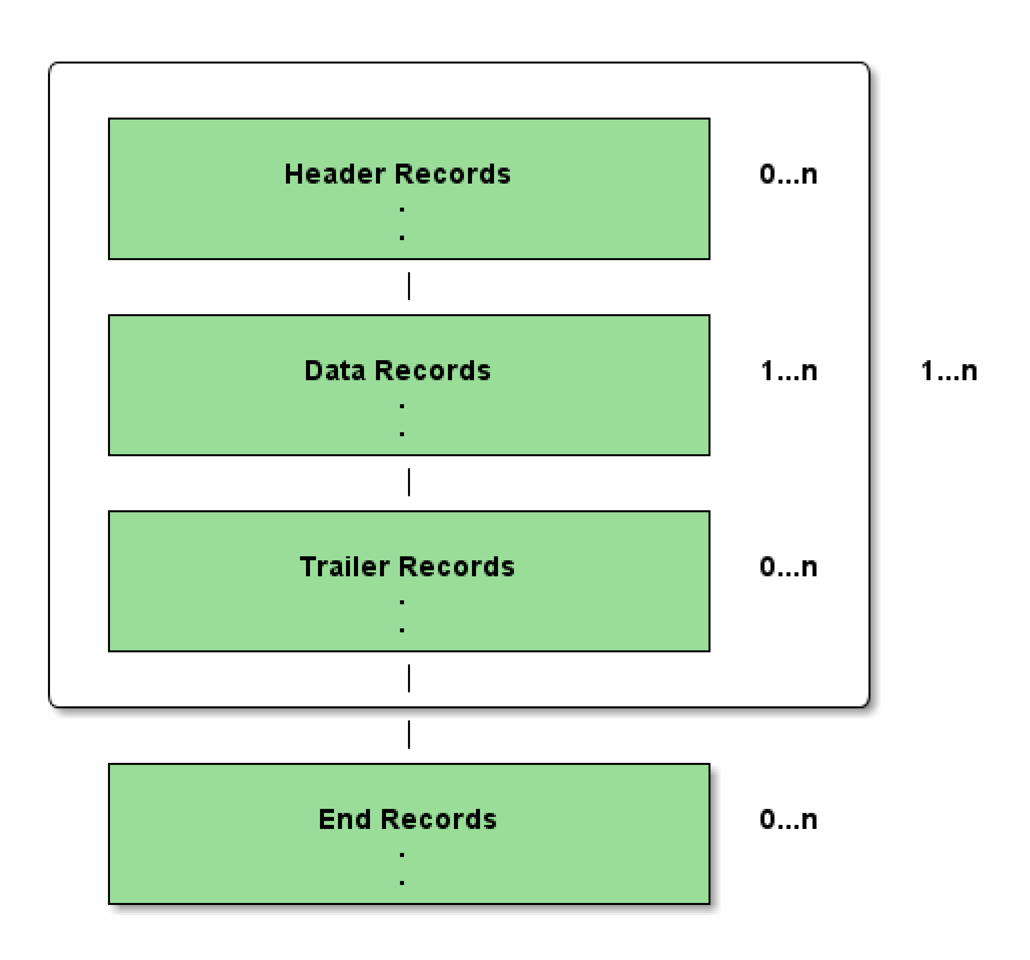
Depending on the definition of the file format in each system, There are cases where names are different from this guideline such as calling Footer Record as End Record.
Must be read as appropriate.

A summary of Single Format and Multi Format is shown below.

Overview of Single Format and Multi Format

Format	Overview
Single Format	A format with Header N Rows + Data N Rows + Trailer N Rows.
Multi Format	A format with (Header N Rows + Data N Rows + Trailer N Rows) * N + Footer N Rows. A format in which a Footer Record is added after repeating a Single Format a plurality of times.

The Multi Format record structure is shown in the figure as follows.



Multi Format Rrecord Structure Diagram

An example of a Single Format and Multi Format flat file is shown below.
`//` is used as a comment-out character for the description of the file.

Example of Single Format, flat file(CSV format) without record division

```
branchId,year,month,customerId,amount  // (1)
000001,2016,1,0000000001,100000000  // (2)
000001,2016,1,0000000002,200000000  // (2)
000001,2016,1,0000000003,300000000  // (2)
000001,3,600000000  // (3)
```

Item list of file contents

No	Descriptions
(1)	A header record Field name of the data part is described.
(2)	A data record.
(3)	A trailer record. It holds summary information of the data part.

Example of Single Format, flat file(CSV format) with record division

```
// (1)
H,branchId,year,month,customerId,amount  // (2)
D,000001,2016,1,0000000001,100000000
D,000001,2016,1,0000000002,200000000
D,000001,2016,1,0000000003,300000000
T,000001,3,600000000
H,branchId,year,month,customerId,amount  // (2)
D,00002,2016,1,0000000004,400000000
D,00002,2016,1,0000000005,500000000
D,00002,2016,1,0000000006,600000000
T,00002,3,1500000000
H,branchId,year,month,customerId,amount  // (2)
D,00003,2016,1,0000000007,700000000
D,00003,2016,1,0000000008,800000000
D,00003,2016,1,0000000009,900000000
T,00003,3,2400000000
F,3,9,4500000000  // (3)
```

Item list of file contents

No	Descriptions
(1)	It has a field indicating the record division at the beginning of the record. Each record division is defined as below. `H` : Header Record `D` : Data Record `T` : Trailer Record `F` : Footer Record
(2)	Every time branchId changes, it repeats header, data, trailer.
(3)	A footer record. It holds summary information for the whole file.

> *Assumptions on format of data part*
>
> In How To Use, it will explain on the premise that the layout of the data part is the same format. This means that all the records of the data part are mapped to the same conversion target class

About explanation of Multi Format file

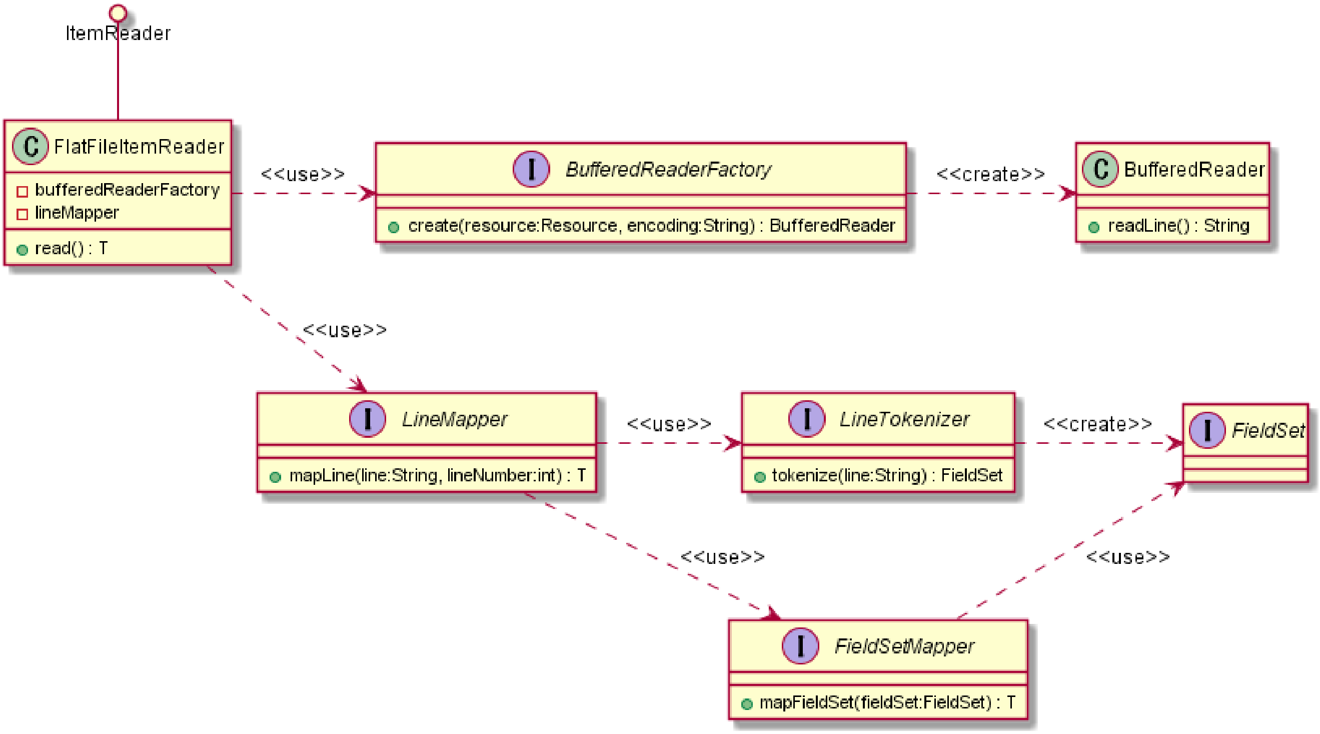
- In How To Use, it will describe about the Single Format file.
- For flat files having Multi Format or a structure including a footer part in the above structure, refer to How To Extend

5.3.1.2. A component that inputs and outputs a flat file

Describe a class for handling flat file.

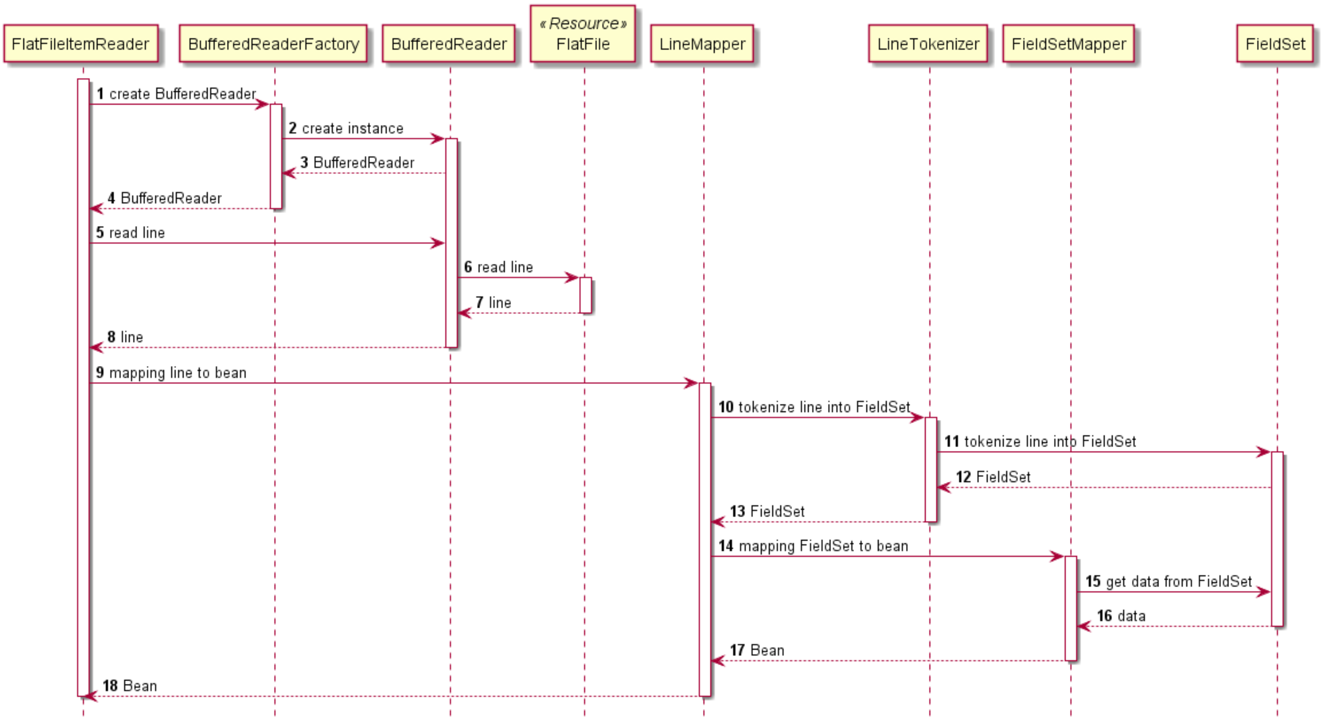
Input

The relationships of classes used for inputting flat files are as follows.



Relationship of classes used for inputting flat files

The calling relationship of each component is as follows.



Calling relationship of each component

Details of each component are shown below.

org.springframework.batch.item.file.FlatFileItemReader

Implementation class of `ItemReader` to use for loading flat files. Use the following components.
The flow of simple processing is as follows.
1.Use `BufferedReaderFactory` to get `BufferedReader`.
2.Read one record from the flat file using the acquired `BufferedReader`.
3.Use `LineMapper` to map one record to the target bean.

org.springframework.batch.item.file.BufferedReaderFactory

Generate `BufferedReader` to read the file.

org.springframework.batch.item.file.LineMapper

One record is mapped to the target bean. Use the following components.
The flow of simple processing is as follows.
1.Use `LineTokenizer` to split one record into each item.
2.Mapping items split by `FieldSetMapper` to bean properties.

org.springframework.batch.item.file.transform.LineTokenizer

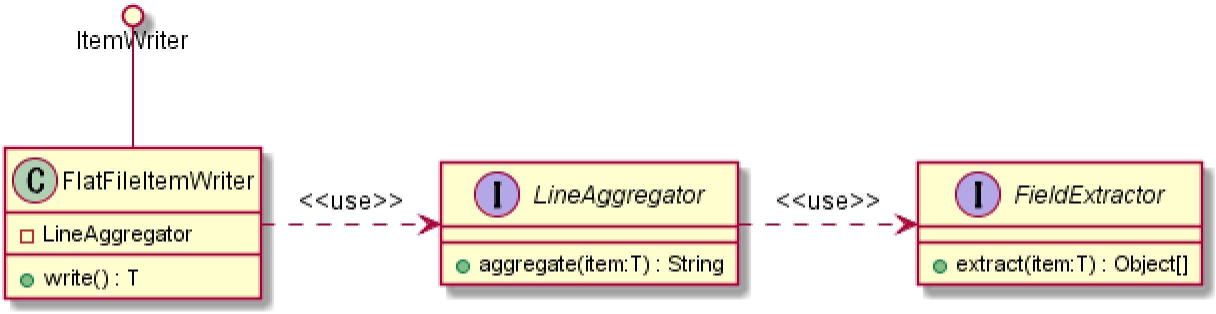
Divide one record acquired from the file into each item.
Each partitioned item is stored in `FieldSet` class.

org.springframework.batch.item.file.mapping.FieldSetMapper

Map each item in one divided record to the property of the target bean.

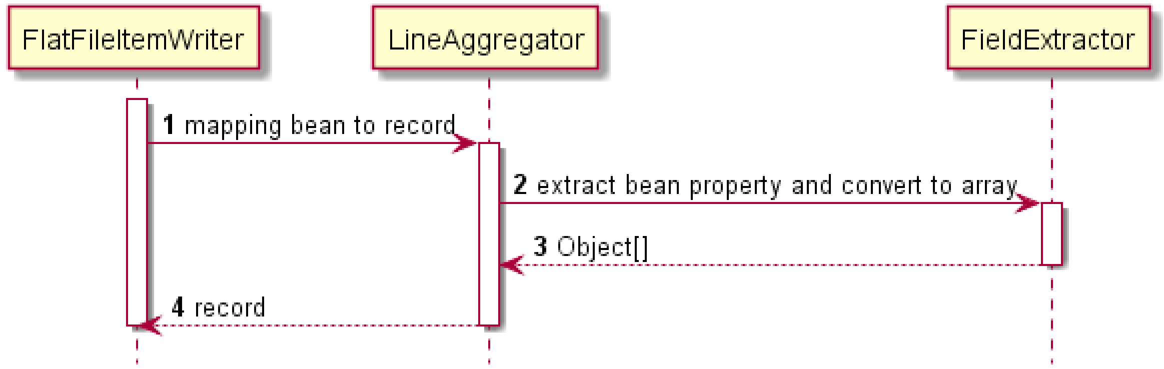
Output

Relationships of classes used for outputting flat files are as follows.



Relationship of classes used for outputting flat files

The calling relationship of each component is as follows.



Calling relationship of each component

org.springframework.batch.item.file.FlatFileItemWriter

Implementation class of `ItemWriter` for exporting to a flat file. Use the following components. `LineAggregator` Mapping the target bean to one record.

org.springframework.batch.item.file.transform.LineAggregator

It is used to map the target bean to one record. The mapping between the properties of the bean and each item in the record is done in `FieldExtractor`.

org.springframework.batch.item.file.transform.FieldExtractor

Map the property of the target bean to each item in one record.

5.3.2. How To Use

Descriptoins for how to use flat file according to the record format.

- Variable-length record
- Fixed-length record
- Single String record

Then, the following items are explained.

- Header and Footer
- Multiple Files
- Control Break

5.3.2.1. Variable-length record

Describe the definition method when dealing with variable-length record file.

5.3.2.1.1. Input

An example of setting for reading the following input file is shown.

Input File Sample

```
000001,2016,1,0000000001,1000000000
000002,2017,2,0000000002,2000000000
000003,2018,3,0000000003,3000000000
```

Class to be converted

```java
public class SalesPlanDetail {

    private String branchId;
    private int year;
    private int month;
    private String customerId;
    private BigDecimal amount;

    // omitted getter/setter
}
```

The setting for reading the above file is as follows.

Bean definition example

```
<!-- (1) (2) (3) -->
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[inputFile]}"
      p:encoding="MS932"
      p:strict="true">
  <property name="lineMapper">  <!-- (4) -->
    <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">  <!-- (5) -->
        <!-- (6) (7) (8) -->
        <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
              p:names="branchId,year,month,customerId,amount"
              p:delimiter=","
              p:quoteCharacter='"'/>
      </property>
      <property name="fieldSetMapper">  <!-- (9) -->
        <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

p:targetType="org.terasoluna.batch.functionaltest.app.model.plan.SalesPlanDetail"/>
      </property>
    </bean>
  </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	resource	Set the input file.	✔	Nothing
(2)	encoding	Sets the character code of the input file.		JavaVM's default character set
(3)	strict	If true is set, an exception occurs if the input file does not exist(can not be opened).		true
(4)	lineMapper	Set `org.springframework.batch.item.file.mapping.DefaultLineMapper`. `DefaultLineMapper` is `LineMapper` which provides the basic operation of converting records to the class to be converted using the defined `LineTokenizer` and `FieldSetMapper`.	✔	Nothing

No	Property Name	Setting contents	Required	Default Value
(5)	lineTokenizer	Set `org.springframework.batch.item.file.transform.DelimitedLineTokenizer`. `DelimitedLineTokenizer` is an implementation class of `LineTokenizer` that separates records by specifying delimiters. It corresponds to the reading of escaped line feeds, delimiters, and enclosed characters defined in the specification of RFC-4180, which is a general format of CSV format.	✔	Nothing
(6)	names	Give a name to each item of one record. Each item can be retrieved using the name set in `FieldSet` used in `FieldSetMapper`. Set each name from the beginning of the record with a comma separator. When using `BeanWrapperFieldSetMapper` it is mandatory setting.		Nothing
(7)	delimiter	Set delimiter		comma
(8)	quoteCharacter	Set enclosing character		Nothing
(9)	fieldSetMapper	If special conversion processing such as character strings and numbers is unnecessary, use `org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper`, and specify the class to be converted to property `targetType`. By doing this, an instance that automatically sets the value in the field that matches the name of each item set in (5) will be created. If conversion processing is necessary, set the implementation class of `org.springframework.batch.item.file.mapping.FieldSetMapper`.	✔	Nothing

See How To Extend for the case of implementing FieldSetMapper yourself.

How to enter TSV format file

When reading the TSV file, it can be realized by setting a tab as a delimiter.

TSV file loading: Example of delimiter setting (setting by constant)

```
<property name="delimiter">
    <util:constant
            static-
field="org.springframework.batch.item.file.transform.DelimitedLineToke
nizer.DELIMITER_TAB"/>
</property>
```

Or, it may be as follows.

TSV file reading: Example of delimiter setting (setting by character reference)

```
<property name="delimiter" value="&#09;"/>
```

5.3.2.1.2. Output

An example of setting for writing the following output file is shown.

Output file example

```
001,CustomerName001,CustomerAddress001,11111111111,001
002,CustomerName002,CustomerAddress002,11111111111,002
003,CustomerName003,CustomerAddress003,11111111111,003
```

Class to be converted

```
public class Customer {

    private String customerId;
    private String customerName;
    private String customerAddress;
    private String customerTel;
    private String chargeBranchId;
    private Timestamp createDate;
    private Timestamp updateDate;

    // omitted getter/setter
}
```

The settings for writing the above file are as follows.

Bean definition example

```
<!-- Writer -->
<!-- (1) (2) (3) (4) (5) (6) (7) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:resource="file:#{jobParameters[outputFile]}"
      p:encoding="MS932"
      p:lineSeparator="&#x0A;"
      p:appendAllowed="true"
      p:shouldDeleteIfEmpty="true"
      p:shouldDeleteIfExists="false"
      p:transactional="true">
  <property name="lineAggregator">  <!-- (8) -->
    <bean
class="org.springframework.batch.item.file.transform.DelimitedLineAggregator"
          p:delimiter=",">  <!-- (9) -->
      <property name="fieldExtractor">  <!-- (10) -->
        <!-- (11) -->
        <bean
class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"

p:names="customerId,customerName,customerAddress,customerTel,chargeBranchId">
      </property>
    </bean>
  </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	resource	Set the output file.	✔	Nothing
(2)	encoding	Sets the character code of the output file.		JavaVM default character set
(3)	lineSeparator	Set record break (line feed code).		`line.separator` of system's property
(4)	appendAllowed	If true, add to the existing file.		false
(5)	shouldDeleteIfEmpty	If true, delete the output if it is an empty file.		false
(6)	shouldDeleteIfExists	If true, delete the file if it already exists. If false, throw an exception if the file already exists.		true
(7)	transactional	Set whether to perform transaction control. For details, see Transaction Control.		true

No	Property Name	Setting contents	Required	Default Value
(8)	lineAggregator	Set `org.springframework.batch.item.file.transform.DelimitedLineAggregator`. To enclose a field around it, set `org.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregator`. Usage of `EnclosableDelimitedLineAggregator` will be described later.	✔	Nothing
(9)	delimiter	Sets the delimiter.		comma
(10)	fieldExtractor	If special conversion processing for strings and numbers is unnecessary, you can use `org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor`. If conversion processing is necessary, set implementation class of `org.springframework.batch.item.file.transform.FieldExtractor`. An example for implementatiion of `FieldExtractor`, refer to Output of Fixed-length file where a sample is described using full-width character.	✔	Nothing
(11)	names	Give a name to each item of one record. Set each name from the beginning of the record with a comma separator.	✔	Nothing

How to use EnclosableDelimitedLineAggregator

To enclose a field around it, use `org.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregator` provided by TERASOLUNA Batch 5.x.
The specification of `EnclosableDelimitedLineAggregator` is as follows.

- Optional specification of enclosure character and delimiter character
  - Default is the following value commonly used in CSV format
    - Enclosed character: `"`(double quote)
    - Separator: `,` (comma)
- If the field contains a carriage return, line feed, enclosure character, or delimiter, enclose the field with an enclosing character
  - When enclosing characters are included, the enclosing character will be escaped by adding an enclosing character right before this enclosing characters.
  - All fields can be surrounded by characters by setting

The usage of `EnclosableDelimitedLineAggregator` is shown below.

Output file example

```
"001","CustomerName""001""","CustomerAddress,001","11111111111","001"
"002","CustomerName""002""","CustomerAddress,002","11111111111","002"
"003","CustomerName""003""","CustomerAddress,003","11111111111","003"
```

Class to be converted

```
// Same as above example
```

Bean definition example(only settings for lineAggregator)

```
<property name="lineAggregator">  <!-- (1) -->
  <!-- (2) (3) (4) -->
  <bean
class="org.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregator"
        p:delimiter=","
        p:enclosure='"'
        p:allEnclosing="true">
      <property name="fieldExtractor">
        <!-- omitted settings -->
      </property>
  </bean>
</property>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	lineAggregator	Set `org.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregator`.	✔	Nothing
(2)	delimiter	Sets the delimiter.		comma
(3)	enclosure	Set the enclosing character. If the enclosing character is included in the field, it is replaced with a concatenated character as an escape process.		double quote
(4)	allEnclosing	If true, all fields are enclosed in an enclosing character. If false, only fields containing carriage return (CR), line-leading (LF), delimiter, and enclosing characters will be enclosed.		false

TERASOLUNA Batch 5.x provides the extension class `org.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregator` to satisfy the specification of RFC-4180.

The `org.springframework.batch.item.file.transform.DelimitedLineAggregator` provided by Spring Batch does not correspond to the enclosing process of the field, therefore it can not satisfy the specification of RFC-4180. Refer to Spring Batch/BATCH-2463 .

The format of the CSV format is defined as follows in RFC-4180 which is a general format of CSV format.

- If the field does not contain line breaks, enclosing characters, or delimiters, each field can be enclosed in double quotes (enclosing characters) or not enclosed
- Fields that contain line feed (CRLF), double quote (enclosing character), comma (delimiter) should be enclosed in double quotes
- If the field is enclosed in double quotes (enclosing characters), the double quotes contained in the value of the field must be escaped with a single double quote immediately before it

How to output TSV format file

When outputting a TSV file, it can be realized by setting a tab as a delimiter.

Setting example of delimiter when outputting TSV file (setting by constant)

```
<property name="delimiter">
    <util:constant
            static-
field="org.springframework.batch.item.file.transform.DelimitedLineToke
nizer.DELIMITER_TAB"/>
</property>
```

Or, it may be as follows.

Example of delimiter setting when TSV file is output (setting by character reference)

```
<property name="delimiter" value="&#09;"/>
```

5.3.2.2. Fixed-length record

Describe how to define fixed length record files.

5.3.2.2.1. Input

An example of setting for reading the following input file is shown.

TERASOLUNA Batch 5.x corresponds to a format in which record delimitation is judged by line feed and format judged by the number of bytes.

Input file example 1 (record breaks are line feeds)

```
Sale012016 1   00000011000000000
Sale022017 2   00000022000000000
Sale032018 3   00000033000000000
```

Input file example 2 (record delimiter is byte number, 32 bytes is 1 record)

```
Sale012016 1   00000011000000000Sale022017 2   00000022000000000Sale032018 3
00000033000000000
```

Input file specification

No	Field Name	Data Type	Number of bytes
(1)	branchId	String	6
(2)	year	int	4
(3)	month	int	2
(4)	customerId	String	10
(5)	amount	BigDecimal	10

Class to be converted

```
public class SalesPlanDetail {

    private String branchId;
    private int year;
    private int month;
    private String customerId;
    private BigDecimal amount;

    // omitted getter/setter
}
```

The setting for reading the above file is as follows.

Bean definition example

```
<!-- (1) (2) (3) -->
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[inputFile]}"
      p:encoding="MS932"
      p:strict="true">
    <property name="bufferedReaderFactory">  <!-- (4) -->
        <bean class=
"org.springframework.batch.item.file.DefaultBufferedReaderFactory"/>
    </property>
    <property name="lineMapper">  <!-- (5) -->
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">  <!-- (6) -->
                <!-- (7) -->
                <!-- (8) -->
                <!-- (9) -->
                <bean
class="org.terasoluna.batch.item.file.transform.FixedByteLengthLineTokenizer"
                      p:names="branchId,year,month,customerId,amount"
                      c:ranges="1-6, 7-10, 11-12, 13-22, 23-32"
                      c:charset="MS932" />
            </property>
            <property name="fieldSetMapper">  <!-- (10) -->
              <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

p:targetType="org.terasoluna.batch.functionaltest.app.model.plan.SalesPlanDetail"/>
            </property>
        </bean>
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Require d	Default Value
(1)	resource	Set the input file.	✔	Nothing
(2)	encoding	Sets the character code of the input file.		JavaVM default character set
(3)	strict	If true is set, an exception occurs if the input file does not exist(can not be opened).		true

No	Property Name	Setting contents	Required	Default Value
(4)	bufferedReaderFactory	To decide record breaks by line breaks, use the default value `org.springframework.batch.item.file.DefaultBufferedReaderFactory`. `BufferedReader` generated by `DefaultBufferedReaderFactory` acquires up to a newline as one record. To judge the delimiter of a record by the number of bytes, set `org.terasoluna.batch.item.file.FixedByteLengthBufferedReaderFactory` provided by TERASOLUNA Batch 5.x. `BufferedReader` generated by `FixedByteLengthBufferedReaderFactory` acquires up to the specified number of bytes as one record. Detailed specifications and usage of `FixedByteLengthBufferedReaderFactory` will be described later.		`DefaultBufferedReaderFactory`
(5)	lineMapper	Set `org.springframework.batch.item.file.mapping.DefaultLineMapper`.	✔	Nothing
(6)	lineTokenizer	Set `org.terasoluna.batch.item.file.transform.FixedByteLengthLineTokenizer` provided by TERASOLUNA Batch 5.x.	✔	Nothing
(7)	names	Give a name to each item of one record. Each item can be retrieved using the name set in `FieldSet` used in `FieldSetMapper`. Set each name from the beginning of the record with a comma separator. When using `BeanWrapperFieldSetMapper` it is mandatory setting.		Nothing
(8)	ranges (Constructor argument)	Sets the break position. Set the delimiter position from the beginning of the record, separated by commas. The unit of each delimiter position is byte, and it is specified in `start position - end position` format. The range specified from the record is acquired in the order in which the delimiter positions are set, and stored in `FieldSet`. When names of (6) are specified, the delimiter positions are stored in `FieldSet` in correspondence with names in the order in which they are set.	✔	Nothing

No	Property Name	Setting contents	Required	Default Value
(9)	charset (Constructor argument)	Set the same character code as (2).	✔	Nothing
(10)	fieldSetMapper	If special conversion processing for character strings and numbers is unnecessary, use `org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper`, and specify the conversion target class as property `targetType`. By doing this, we create an instance that automatically sets the value in the field that matches the name of each item set in (6). If conversion processing is necessary, set the implementation class of `org.springframework.batch.item.file.mapping.FieldSetMapper`.	✔	Nothing

> See How To Extend for the case of implementing FieldSetMapper yourself.

How to use FixedByteLengthBufferedReaderFactory

To read a file that judges record delimiter by byte count, use `org.terasoluna.batch.item.file.FixedByteLengthBufferedReaderFactory` provided by TERASOLUNA Batch 5.x.

By using `FixedByteLengthBufferedReaderFactory`, it is possible to acquire up to the number of bytes specified as one record.
The specification of `FixedByteLengthBufferedReaderFactory` is as follows.

- Specify byte count of record as constructor argument
- Generate `FixedByteLengthBufferedReader` which reads the file with the specified number of bytes as one record

Use of `FixedByteLengthBufferedReader` is as follows.

- Reads a file with one byte length specified at instance creation
- If there is a line feed code, do not discard it and read it by including it in the byte length of one record
- The file encoding to be used for reading is the value set for `FlatFileItemWriter`, and it will be used when `BufferedReader` is generated.

The method of defining `FixedByteLengthBufferedReaderFactory` is shown below.

```xml
<property name="bufferedReaderFactory">
    <bean class="org.terasoluna.batch.item.file.FixedByteLengthBufferedReaderFactory"
        c:byteLength="32"/>  <!-- (1) -->

</property>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	byteLength (Constructor argument)	Set the number of bytes per record.	✔	Nothing

Components to use when handling Fixed-length files

When dealing with Fixed-length files, it is based on using the component provided by TERASOLUNA Batch 5.x.

FixedByteLengthBufferedReaderFactory

`BufferedReader` generation class that reads one record from the fixed-length file without line break by the number of bytes of the specified character code

FixedByteLengthLineTokenizer

The `FixedLengthTokenizer` extension class, separated by the number of bytes corresponding to the multibyte character string

Processing records containing multibyte character strings

When processing records containing multibyte character strings, be sure to use `FixedByteLengthLineTokenizer`.
The `FixedLengthTokenizer` provided by Spring Batch separates the record by the number of characters instead of the number of bytes, so there is a possibility that the item will not be extracted as expected.

Since this issue is already reported to JIRA Spring Batch/BATCH-2540, it might be unnecessary in the future.

For the implementation of FieldSetMapper, refer to How To Extend.

5.3.2.2.2. Output

An example of setting for writing the following output file is shown.

In order to write a fixed-length file, it is necessary to format the value obtained from the bean according to the number of bytes of the field.
The format execution method differs as follows depending on whether double-byte characters are included or not.

- If double-byte characters is not included(single-byte characters only and the number of bytes of characters is constant)
  - Format using `FormatterLineAggregator`.
  - The format is set by the format used in the `String.format` method.
- If double-byte characters is included(The number of bytes of characters is not constant depending on the character code)
  - Format with implementation class of `FieldExtractor`.

First, a setting example in the case where double-byte characters are not included in the output file is shown, followed by a setting example in the case where double-byte characters are included.

The setting when double-byte characters are not included in the output file is shown below.

Output file example

```
0012016 10000000001  10000000
0022017 20000000002  20000000
0032018 30000000003  30000000
```

Output file specification

No	Field Name	Data Type	Number of bytes
(1)	branchId	String	6
(2)	year	int	4
(3)	month	int	2
(4)	customerId	String	10
(5)	amount	BigDecimal	10

If the field's value is less than the number of bytes specified, the rest of the field will be filled with halfwidth space.

Class to be converted

```
public class SalesPlanDetail {

    private String branchId;
    private int year;
    private int month;
    private String customerId;
    private BigDecimal amount;

    // omitted getter/setter
}
```

The settings for writing the above file are as follows.

Bean definition

```
<!-- Writer -->
<!-- (1) (2) (3) (4) (5) (6) (7) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:resource="file:#{jobParameters[outputFile]}"
      p:encoding="MS932"
      p:lineSeparator="&#x0A;"
      p:appendAllowed="true"
      p:shouldDeleteIfEmpty="true"
      p:shouldDeleteIfExists="false"
      p:transactional="true">
    <property name="lineAggregator">  <!-- (8) -->
        <bean
class="org.springframework.batch.item.file.transform.FormatterLineAggregator"
              p:format="%6s%4s%2s%10s%10s"/>  <!-- (9) -->
            <property name="fieldExtractor">  <!-- (10) -->
              <bean
class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
                    p:names="branchId,year,month,customerId,amount"/>  <!-- (11) -->
            </property>
        </bean>
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	resource	Set the output file.	✔	Nothing
(2)	encoding	Sets the character code of the output file.		JavaVM default character set
(3)	lineSeparator	Set the record break(line feed code) To make it without line breaks, set `(empty string)`.		`line.separator` of system's property
(4)	appendAllowed	If true, append to the existing file.		false
(5)	shouldDeleteIfEmpty	If true, delete the output if it is an empty file.		false
(6)	shouldDeleteIfExists	If true, delete the file if it already exists. If false, throw an exception if the file already exists.		true
(7)	transactional	Set whether to perform transaction control. For details, see Transaction Control.		true

No	Property Name	Setting contents	Required	Default Value
(8)	lineAggregator	Set `org.springframework.batch.item.file.transform.FormatterLineAggregator`.	✔	Nothing
(9)	format	Set the output format with the format used in the `String.format` method.	✔	Nothing
(10)	fieldExtractor	If special conversion processing for strings and numbers is unnecessary, you can use `org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor`. If conversion processing is necessary, set implementation class of `org.springframework.batch.item.file.transform.FieldExtractor`. An example for implementatiion of `FieldExtractor` to format double-byte characters is written later on.		`PassThroughFieldExtractor`
(11)	names	Give a name to each item of one record. Set the names of each field from the beginning of the record with a comma.	✔	Nothing

> *About PassThroughFieldExtractor*
>
> Deafult value for property `fieldExtractor` of `FormatterLineAggregator` is `org.springframework.batch.item.file.transform.PassThroughFieldExtractor`.
>
> `PassThroughFieldExtractor` is a class to return the original item without processing anything, and is used when `FieldExtractor` will not process anything.
>
> If the item is an array or a collection, it is returned as is, otherwise it is wrapped in an array of single elements.

Example of how to format a field with double-byte character

When formatting for double-byte characters, since the number of bytes per character differs depending on the character code, use the implementation class of `FieldExtractor` instead of `FormatterLineAggregator`.

Implementation class of `FieldExtractor` is to be done as follows.

- Implement `FieldExtractor` and override extract method.
- extract method is to be implemented as below
  - get the value from the item(target bean), and perform the conversion as needed
  - set the value to an array of object and return it.

The format of a field that includes double-byte characters is to be done in the implementation class of `FieldExtractor` by the following way.

- Get the number of bytes for the character code
- Format the value by trimming or padding it according to be number of bytes

Below is a setting example for formatting a field including double-byte characters.

Output file example

```
   0012016 10000000001  10000000
  番号2017 2 売上高002  20000000
 番号32018 3   売上003  30000000
```

Use of the output file is the same as the example above.

Bean definition(settings of lineAggregator only)

```
<property name="lineAggregator">  <!-- (1) -->
    <bean
class="org.springframework.batch.item.file.transform.FormatterLineAggregator"
          p:format="%s%4s%2s%s%10s"/>  <!-- (2) -->
        <property name="fieldExtractor">  <!-- (3) -->
            <bean
class="org.terasoluna.batch.functionaltest.ch05.fileaccess.plan.SalesPlanFixedLengthFi
eldExtractor"/>
        </property>
    </bean>
</property>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	lineAggregator	Set `org.springframework.batch.item.file.transform.FormatterLineAggregator`.	✔	Nothing
(2)	format	Set the output format with the format used in the `String.format` method. The number of digits is specified only for fields that do not contain double-byte characters.	✔	Nothing
(3)	fieldExtractor	Set implementation class of `FieldExtractor`. An implementation example will be described later.		`PassThroughFieldExtractor`

Class to be converted

```java
public class SalesPlanDetail {

    private String branchId;
    private int year;
    private int month;
    private String customerId;
    private BigDecimal amount;

    // omitted getter/setter
}
```

Sample implementation of FieldExtractor to format double-byte characters

```
public class SalesPlanFixedLengthFieldExtractor implements FieldExtractor
<SalesPlanDetail> {
    // (1)
    @Override
    public Object[] extract(SalesPlanDetail item) {
        Object[] values = new Object[5];  // (2)

        // (3)
        values[0] = fillUpSpace(item.getBranchId(), 6);  // (4)
        values[1] = item.getYear();
        values[2] = item.getMonth();
        values[3] = fillUpSpace(item.getCustomerId(), 10);  // (4)
        values[4] = item.getAmount();

        return values; // (7)
    }

    // It is a simple impl for example
    private String fillUpSpace(String val, int num) {
        String charsetName = "MS932";
        int len;
        try {
            len = val.getBytes(charsetName).length;  // (5)
        } catch (UnsupportedEncodingException e) {
            // omitted exception handling
        }

        String fillStr = "";
        for (int i = 0; i < (num - len); i++) { // (6)
            fillStr += " ";
        }

        return fillStr + val;
    }
}
```

Item list of setting contents

No	Description
(1)	Implement `FieldExtractor` class and override extract method. Set the conversion target class as the type argument of `FieldExtractor`.
(2)	Define a Object type array to store data after the conversion.
(3)	Get the value from the item(target bean), and perform the conversion as needed, set the value to an array of object.
(4)	Format the field that includes double-byte character. Refer to (5) and (6) for the details of format process.

No	Description
(5)	Get the number of bytes for the character code.
(6)	Format the value by trimming or padding it according to be number of bytes. In the implementation example, white space characters are added before the character string up to the specified number of bytes.
(7)	Returns an array of Object type holding the processing result.

5.3.2.3. Single String record

Describe the definition method when dealing with a single character string record file.

5.3.2.3.1. Input

An example of setting for reading the following input file is shown below.

Input file sample

```
Summary1:4,000,000,000
Summary2:5,000,000,000
Summary3:6,000,000,000
```

The setting for reading the above file is as follows.

Bean definition

```
<!-- (1) (2) (3) -->
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[inputFile]}"
      p:encoding="MS932"
      p:strict="true">
    <property name="lineMapper">  <!-- (4) -->
        <bean
class="org.springframework.batch.item.file.mapping.PassThroughLineMapper"/>
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	resource	Set the input file.	✔	Nothing
(2)	encoding	Sets the character code of the input file.		JavaVM default character set
(3)	strict	If true is set, an exception occurs if the input file does not exist(can not be opened).		true

No	Property Name	Setting contents	Required	Default Value
(4)	lineMapper	Set `org.springframework.batch.item.file.mapping.PassThroughLineMapper`. `PassThroughLineMapper` is a implementation class of `LineMapper`, and it will return the String value of passed record as it is.	✔	Nothing

5.3.2.3.2. Output

The setting for writing the above file is as follows.

Output file example

```
Summary1:4,000,000,000
Summary2:5,000,000,000
Summary3:6,000,000,000
```

Bean definition

```
<!-- Writer -->
<!-- (1) (2) (3) (4) (5) (6) (7) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:resource="file:#{jobParameters[outputFile]}"
      p:encoding="MS932"
      p:lineSeparator="&#x0A;"
      p:appendAllowed="true"
      p:shouldDeleteIfEmpty="true"
      p:shouldDeleteIfExists="false"
      p:transactional="true">
    <property name="lineAggregator">  <!-- (8) -->
        <bean
class="org.springframework.batch.item.file.transform.PassThroughLineAggregator"/>
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	resource	Set the output file.	✔	Nothing
(2)	encoding	Sets the character code of the output file.		JavaVM default character set
(3)	lineSeparator	Set the record break(line feed code)		`line.separator` of system's property

No	Property Name	Setting contents	Required	Default Value
(4)	appendAllowed	If true, append to the existing file.		false
(5)	shouldDeleteIfEmpty	If true, delete the output if it is an empty file.		false
(6)	shouldDeleteIfExists	If true, delete the file if it already exists. If false, throw an exception if the file already exists.		true
(7)	transactional	Set whether to perform transaction control. For details, see Transaction Control.		true
(8)	lineAggregator	Set `org.springframework.batch.item.file.transform.PassThroughLineAggregator`. `PassThroughLineAggregator` is the implementation class of `LineAggregator` that will return the converted String value of the item(target Bean) as it is by processing `item.toString()`.	✔	Nothing

5.3.2.4. Header and Footer

Explain the input / output method when there is a header / footer.

Here, a method of skipping the header footer by specifying the number of lines will be explained. When the number of records of header / footer is variable and it is not possible to specify the number of lines, use `PatternMatchingCompositeLineMapper` with reference to Multi format input

5.3.2.4.1. Input

Skipping Header

There are 2 ways to skip the header record.

- Set the number of lines to skip to property `linesToSkip` of `FlatFileItemReader`
- Remove header record in preprocessing by OS command

Input file sample

```
sales_plan_detail_11
branchId,year,month,customerId,amount
000001,2016,1,0000000001,1000000000
000002,2017,2,0000000002,2000000000
000003,2018,3,0000000003,3000000000
```

The first 2 lines is the header record.

```
The setting for reading the above file is as follows.
```

Skip by using linesToSkip

```
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[inputFile]}"
      p:linesToSkip=value="2">  <!-- (1) -->
    <property name="lineMapper">
      <!-- omitted settings -->
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	linesToSkip	Set the number of header record lines to skipl.		0

Skip by using OS command

```
# Remove number of lines in header from the top of input file
tail -n +`expr 2 + 1` input.txt > output.txt
```

Use the tail command and get the 3rd line and after from input.txt, and then write it out to output.txt. Please note that the value specified for option `-n + K` of tail command is the number of header records + 1.

OS command to skip header record and footer record

By using the head and tail commands, it is possible to skip the header record and footer record by specifying the number of lines.

How to skip the header record

Execute the tail command with option `-n +K`, and get the lines after `K` from the target file.

How to skip the footer record

Execute the head command with option `-n -K`, and get the lines befor `K` from the target file.

A sample of shell script to skip header record and footer record can be written as follows.

An example of a shell script that removes a specified number of lines from a header / footer

```
#!/bin/bash

if [ $# -ne 4 ]; then
  echo "The number of arguments must be 4, given is $#." 1>&2
  exit 1
fi

# Input file.
input=$1

# Output file.
output=$2

# Number of lines in header.
header=$3

# Number of lines in footer.
footer=$4

# Remove number of lines in header from the top of input file
# and number of lines in footer from the end,
# and save to output file.
tail -n +`expr ${header} + 1` ${input} | head -n -${footer} >
${output}
```

Arguments

No	Description
(1)	Input file
(2)	Output file
(3)	Number of lines to skip for header
(4)	Number of lines to skip for footer

Retrieving header information

Here shows how to recognize and retrive the header record.

The extraction of header information is implemented as follows.

Settings

- Write the process for header record in implementation class of `org.springframework.batch.item.file.LineCallbackHandler`
  - Set the information retrieved in `LineCallbackHandler#handleLine()` to `stepExecutionContext`

- Set implementation class of `LineCallbackHandler` to property skippedLinesCallback `of` FlatFileItemReader``
- Set the number of lines to skip to property `linesToSkip` of `FlatFileItemReader`

Reading files and retrieving header information

- For each line which is skipped by the setting of `linesToSkip`, `LineCallbackHandler#handleLine()` is executed
    - Header information is set to `stepExecutionContext`

Use retrieved header information

- Get header information from `stepExecutionContext` and use it in the processing of the data part

An example of implementation for retrieving header record information is shown below.

Bean definition

```
<bean id="lineCallbackHandler"

class="org.terasoluna.batch.functionaltest.ch05.fileaccess.module.HoldHeaderLineCallba
ckHandler"/>

<!-- (1) (2) -->
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:linesToSkip="2"
      p:skippedLinesCallback-ref="lineCallbackHandler"
      p:resource="file:#{jobParameters[inputFile]}">
    <property name="lineMapper">
        <!-- omitted settings -->
    </property>
</bean>

<batch:job id="jobReadCsvSkipAndReferHeader" job-repository="jobRepository">
    <batch:step id="jobReadCsvSkipAndReferHeader.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader"
                         processor="loggingHeaderRecordItemProcessor"
                         writer="writer" commit-interval="10"/>
            <batch:listeners>
                <batch:listener ref="lineCallbackHandler"/>  <!-- (3) -->
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	linesToSkip	Set the number of lines to skip.		0
(2)	skippedLinesCallback	Set implementation class of `LineCallbackHandler`. An implementation sample will be described later.		Nothing
(2)	listener	Set implementation class of `StepExecutionListener`. This setting is needed sinche the `LineCallbackHandler` set to property `skippedLinesCallback` of `FlatFileItemReader` will not be automatically registered as the `Listener`. The detailed reason will be described later.		Nothing

About the listener

Since the following two cases are not automatically registered as `Listener`, it is necessary to add a definition to `Listeners` at the time of job definition.
(If listener definitions are not added, `StepExecutionListener # beforeStep ()` will not be executed)

- `StepExecutionListener` of `LineCallbackHandler` which is set to `skippedLinesCallback` of `FlatFileItemReader`
- `StepExecutionListener` implemented to implementation class of `Tasklet`

```
    <batch:job id="jobReadCsvSkipAndReferHeader" job-
repository="jobRepository">
        <batch:step id="jobReadCsvSkipAndReferHeader.step01">
            <batch:tasklet transaction-manager=
"jobTransactionManager">
                <batch:chunk reader="reader"

processor="loggingHeaderRecordItemProcessor"
                             writer="writer" commit-interval="10"/>
                <batch:listeners>
                    <batch:listener ref="loggingItemReaderListener"/>
                    <!-- mandatory -->
                    <batch:listener ref="lineCallbackHandler"/>
                </batch:listeners>
            </batch:tasklet>
        </batch:step>
    </batch:job>
```

`LineCallbackHandler` should be implemented as follows.

Implement `StepExecutionListener#beforeStep()`

- Implement `StepExecutionListener#beforeStep()` by either ways shown below
  - Implement `StepExecutionListener` class and override beforeStep method
  - Implement beforeStep method and annotate with `@BeforeStep`
- Get `StepExecution` in the beforeStep method and save it in the class field

- Implement `LineCallbackHandler#handleLine()`
  - Implement `LineCallbackHandler` class and override `handleLine`
    - Note that `handleLine` method will be executed each time skip is proceeded
  - Get `stepExecutionContext` from `StepExecution` and set header information to `stepExecutionContext`

Sample implementation of LineCallbackHandler

```
@Component
public class HoldHeaderLineCallbackHandler implements LineCallbackHandler {  // (!)
    private StepExecution stepExecution;  // (2)

    @BeforeStep  // (3)
    public void beforeStep(StepExecution stepExecution) {
        this.stepExecution = stepExecution;  // (4)
    }

    @Override  // (5)
    public void handleLine(String line) {
        this.stepExecution.getExecutionContext().putString("header", line);  // (6)
    }
}
```

Item list of setting contents

No	Description
(1)	Implement `LineCallbackHandler` class and override `handleLine`.
(2)	Define a field to save `StepExecution`.
(3)	Implement `beforeStep` method and annotate it with `@BeforeStep`. The signature will be `void beforeStep(StepExecution stepExecution)`. It is also possible to implement the `StepExecutionListener` class and override `beforeStep` method.
(4)	Get the `StepExecution` and save it to the class field.
(5)	Implement `LineCallbackHandler` class and override `handleLine` method.
(6)	Get `stepExecutionContext` from `StepExecution`, set header information to `stepExecutionContext` by using key `header`. Here, for simplicity, only the last one line of two lines to be skipped is stored.

Here is a sample of getting the header information from stepExecutionContext and using it for processing of data part.
A sample of using header information in ItemProcessor will be described as an example.
The same can be done when using header information in other components.

The implementation of using header information is done as follows.

- As like the sample of implementing LineCallbackHandler, implement StepExecutionListener#beforeStep()
- Get StepExecution in beforeStep method and save it to the class field
- Get stepExecutionContext and the header information from StepExecution and use it

Sample of how to use header information

```
@Component
public class LoggingHeaderRecordItemProcessor implements
        ItemProcessor<SalesPlanDetail, SalesPlanDetail> {
    private StepExecution stepExecution;  // (1)

    @BeforeStep  // (2)
    public void beforeStep(StepExecution stepExecution) {
        this.stepExecution = stepExecution;  // (3)
    }

    @Override
    public SalesPlanDetail process(SalesPlanDetail item) throws Exception {
        String headerData = this.stepExecution.getExecutionContext()
                .getString("header");  // (4)
        // omitted business logic
        return item;
    }
}
```

Item list of setting contents

No	Description
(1)	Define a field to save StepExecution.
(2)	Implement beforeStep method and annotate it with @BeforeStep. The signature will be void beforeStep(StepExecution stepExecution). It is also possible to implement the StepExecutionListener class and override beforeStep method.
(3)	Get the StepExecution and save it to the class field.
(4)	Get stepExecutionContext from StepExecution, set header information to stepExecutionContext by using key header.

About the use of ExecutionContext of Job/Step

In retrieving header (footer) information, the method is to store the read header information in `ExecutionContext` of `StepExecution`, and retrieves it from `ExecutionContext` when using it.

In the example below, header information is stored in `ExecutionContext` of `StepExecution` in order to obtain and use header information within one step. If step is divided by retreiving and using the header information, use `ExecutionContext` of `JobExecution`.

For details about `ExecutionContext` of Job/Step, refer to Architecture of Spring Batch

Skipping Footer

Since Spring Batch nor TERASOLUNA Batch 5.x does not support skipping footer record, it needs to be done by OS command.

Input File Sample

```
000001,2016,1,0000000001,1000000000
000002,2017,2,0000000002,2000000000
000003,2018,3,0000000003,3000000000
number of items,3
total of amounts,6000000000
```

The last 2 lines is the footer record.

The setting for reading the above file is as follows.

Skipping by OS command

```
# Remove number of lines in footer from the end of input file
head -n -2 input.txt > output.txt
```

Use head command, get the lines above the second line from the last from input.txt, and write it out to output.txt.

It is reported to JIRA Spring Batch/BATCH-2539 that Spring Batch does not have a functions to skip the footer record.
Hence, there is a possibility that not only by OS command, but Spring Batch will be able to skip the footer record in the future.

Retrieving footer information

In Spring Batch and TERASOLUNA Batch 5.x, functions for skipping footer record retreiving footer information is not provided.

Therefore, it needs to be divided into preprocessing OS command and 2 steps as described below.

- Divide footer record by OS command
- In 1st step, read the footer record and set footer information to `ExecutionContext`
- In 2nd step, retrive footer information from `ExecutionContext` and use it

Retreiving footer information will be implemented as follows.

Divide footer record by OS command

- Use OS command to divide the input file to footer part and others

1st step, read the footer record and get footer information

- Read the footer record and set it to `jobExecutionContext`
  - Since the steps are different in storing and using footer information, store it in `jobExecutionContext`.
  - The use of `jobExecutionContext` is same as the `stepExecutionContext` explained in Retrieving header information, execpt for the scope of Job and Step.

2nd step, use the retrieved footer information

- Get the footer information from `jobExecutionContext` and use it for processing of data part.

An example will be described in which footer information of the following file is taken out and used.

Input File Sample

```
000001,2016,1,0000000001,1000000000
000002,2017,2,0000000002,2000000000
000003,2018,3,0000000003,3000000000
number of items,3
total of amounts,6000000000
```

The last 2 lines is footer record.

Divide footer record by OS command

The setting to divide the above file into footer part and others by OS command is as follows.

Skipping by OS command

```
# Extract non-footer record from input file and save to output file.
head -n -2 input.txt > input_data.txt

# Extract footer record from input file and save to output file.
tail -n 2 input.txt > input_footer.txt
```

Use head command, write footer part of input.txt to input_footer.txt, and others to input_data.txt.

Output file sample is as follows.

Output file example(input_data.txt)

```
000001,2016,1,0000000001,1000000000
000002,2017,2,0000000002,2000000000
000003,2018,3,0000000003,3000000000
```

Output file example(input_footer.txt)

```
number of items,3
total of amounts,6000000000
```

Get/Use footer information

Explain how to get and use footer information from a footer record divided by OS command.

The step of reading the footer record is divided into the preprocessing and main processing.
Refer to Flow Controll for details of step dividing.

In the example below, a sample is shown in which footer information is retreived and stored in `jobExecutionContext`.
Footer information can be used by retreiving it from `jobExecutionContext` like the same way described in Retrieving header information.

Class to set information of data record

```
public class SalesPlanDetail {

    private String branchId;
    private int year;
    private int month;
    private String customerId;
    private BigDecimal amount;

    // omitted getter/setter
}
```

Class to set information of footer record

```
public class SalesPlanDetailFooter implements Serializable {

    // omitted serialVersionUID

    private String name;
    private String value;

    // omitted getter/setter
}
```

Define the Bean like below.

- Define `ItemReader` to read footer record
- Define `ItemReader` to read data record
- Define business logic to retreive footer record
  - In the sample below, it is done by implementing `Tasklet`
- Define a job
  - Define a step with a preprocess to get footer information and a main process to read data records.

Bean definition

```
<!-- ItemReader for reading footer records -->
<!-- (1) -->
<bean id="footerReader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[footerInputFile]}">
    <property name="lineMapper">
        <!-- omitted other settings -->
    </property>
</bean>

<!-- ItemReader for reading data records -->
<!-- (2) -->
<bean id="dataReader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[dataInputFile]}">
    <property name="lineMapper">
        <!-- omitted other settings -->
    </property>
</bean>

<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step">
  <!-- omitted settings -->
</bean>

<!-- Tasklet for reading footer records -->
<bean id="readFooterTasklet"

class="org.terasoluna.batch.functionaltest.ch05.fileaccess.module.ReadFooterTasklet"/>

<batch:job id="jobReadAndWriteCsvWithFooter" job-repository="jobRepository">
    <!-- (3) -->
    <batch:step id="jobReadAndWriteCsvWithFooter.step01"
            next="jobReadAndWriteCsvWithFooter.step02">
        <batch:tasklet ref="readFooterTasklet"
                       transaction-manager="jobTransactionManager"/>
    </batch:step>
    <!-- (4) -->
    <batch:step id="jobReadAndWriteCsvWithFooter.step02">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="dataReader"
                         writer="writer" commit-interval="10"/>
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="readFooterTasklet"/> <!-- (5) -->
    </batch:listeners>
</batch:job>
```

Item list of setting contents

No	Item	Setting contents	Required	Default Value
(1)	footerReader	Define `ItemReader` to read a file with footer record. Used by injecting it to `readFooterTasklet` which is executed when retreiving footer information.		
(2)	dataReader	Define `ItemReader` to read a file with data record.		
(3)	preprocess step	Define a step to get the footer information. Implemented at `readFooterTasklet`. Implementation sample is written later on.		
(4)	main process step	A step of retreiving data information and using footer information is defined. Use `dataReader` for `reader`. In the sample, method to get footer information from `jobExecutionContext` such as `ItemProcessor` is not implemented. Footer information can be retreived and used the same way described in Retrieving header information.		
(5)	listeners	Set `readFooterTasklet`. Without this setting, `JobExecutionListener#beforeJob()` implemented in `readFooterTasklet` will not be executed. For details, refer to Retrieving header information.		Nothing

An example for reading a file with footer record and storing it to `jobExecutionContext`is shown below.

The way to make it as the implementation class of `Tasklet` is as follows.

- Inject the bean defined `footerReader` by name using `@Inject@` and `@Named`
- Set the footer information to `jobExecutionContext`
  - The realization method is the same as Retrieving header information

Getting footer information

```
public class ReadFooterTasklet implements Tasklet {
    // (1)
    @Inject
    @Named("footerReader")
    ItemStreamReader<SalesPlanDetailFooter> itemReader;

    private JobExecution jobExecution;

    @BeforeJob
    public void beforeJob(JobExecution jobExecution) {
        this.jobExecution = jobExecution;
    }

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception {
        ArrayList<SalesPlanDetailFooter> footers = new ArrayList<>();

        // (2)
        itemReader.open(chunkContext.getStepContext().getStepExecution()
                .getExecutionContext());

        SalesPlanDetailFooter footer;
        while ((footer = itemReader.read()) != null) {
            footers.add(footer);
        }

        // (3)
        jobExecution.getExecutionContext().put("footers", footers);

        return RepeatStatus.FINISHED;
    }
}
```

Item list of setting contents

No	Description
(1)	Inject the bean defined `footerReader` by name using `@Inject@` and `@Named`.
(2)	Use `footerReader` to read the file with footer record and get the footer information. To use `ItemReader` bean defined in implementation class of `Tasklet`, refer to Creating a tasklet-oriented job
(3)	Get `jobExecutionContext` from `JobExecution`, set the footer information to `jobExecutionContext` by key `footers`.

5.3.2.4.2. Output

Output header information

To output header information to a flat file, implement as follows.

- Implement `org.springframework.batch.item.file.FlatFileHeaderCallback`
- Set the implemented `FlatFileHeaderCallback` to property `headerCallback` of `FlatFileItemWriter`
  - By setting `headerCallback`, `FlatFileHeaderCallback#writeHeader()` will be executed at first when processing `FlatFileItemWriter`

Implement `FlatFileHeaderCallback` as follows.

- Implement `FlatFileHeaderCallback` class and override `writeHeader`.
- Write the header information using `Writer` from the argument.

Sample implementation of `FlatFileHeaderCallback` is shown below.

Sample implementation of FlatFileHeaderCallback

```
@Component
// (1)
public class WriteHeaderFlatFileFooterCallback implements FlatFileHeaderCallback {
    @Override
    public void writeHeader(Writer writer) throws IOException {
        // (2)
        writer.write("omitted");
    }
}
```

Item list of setting contents

No	Description
(1)	Implement `FlatFileHeaderCallback` class and override writeHeader method.
(2)	Write the header information using `Writer` from the argument. Write method of `FlatFileItemWriter` will be executed right after the execution of `FlatFileHeaderCallback#writeHeader()`. Therefore, printing line break at the end of header information is not needed. The line feed that is printed is the one set when `FlatFileItemWriter` bean was defined.

Bean definition

```
<!-- (1) (2) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:headerCallback-ref="writeHeaderFlatFileFooterCallback"
      p:lineSeparator="&#x0A;"
      p:resource="file:#{jobParameters[outputFile]}">
    <property name="lineAggregator">
        <!-- omitted settings -->
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	headerCallback	Set implementation class of `FlatFileHeaderCallback`.		
(2)	lineSeparator	Set the record break(line feed code)		`line.separator` of system's property

> *When implementing FlatFileHeaderCallback, printing line feed at the end of header information is not necessary*
>
> Right after executing `FlatFileHeaderCallback#writeHeader()` in `FlatFileItemWriter`, line feed is printed according to the bean definition, so the line feed at the end of header information does not need to be printed.

Output footer information

To output footer information to a flat file, implement as follows.

- Implement `org.springframework.batch.item.file.FlatFileFooterCallback`
- Set the implemented `FlatFileFooterCallback` to property `footerCallback` of `FlatFileItemWriter`
  - By setting `footerCallback`, `FlatFileHeaderCallback#writeFooter()` will be executed at first when processing `FlatFileItemWriter`

A method of outputting footer information with a flat file will be described.

Implement `FlatFileFooterCallback` as follows.

- Output footer information using `Writer` from the argument.
- Implement `FlatFileFooterCallback` class and override `writeFooter`.

Below is an implementation sample of `FlatFileFooterCallback` class for a Job to get footer information from `ExecutionContext` and write it out to a file.

Class to set information of footer record

```
public class SalesPlanDetailFooter implements Serializable {

    // omitted serialVersionUID

    private String name;
    private String value;

    // omitted getter/setter
}
```

Implementation Sample of FlatFileFooterCallback

```
@Component
public class WriteFooterFlatFileFooterCallback implements FlatFileFooterCallback {  //
(1)
    private JobExecution jobExecution;

    @BeforeJob
    public void beforeJob(JobExecution jobExecution) {
        this.jobExecution = jobExecution;
    }

    @Override
    public void writeFooter(Writer writer) throws IOException {
        @SuppressWarnings("unchecked")
        ArrayList<SalesPlanDetailFooter> footers = (ArrayList<SalesPlanDetailFooter>)
this.jobExecution.getExecutionContext().get("footers");  // (2)

        BufferedWriter bufferedWriter = new BufferedWriter(writer);  // (3)
        // (4)
        for (SalesPlanDetailFooter footer : footers) {
            bufferedWriter.write(footer.getName() +" is " + footer.getValue());
            bufferedWriter.newLine();
            bufferedWriter.flush();
        }
    }
}
```

Item list of setting contents

No	Description
(1)	Implement `FlatFileFooterCallback` class and override writeFooter method.
(2)	Get footer information form `ExecutionContext` of the Job using key `footers`. In the sample, it uses ArrayList to get several footer informations.
(3)	In the sample, in order to use `BufferedWriter.newLine()` for printing line feed, it is using `Writer` from the argument as a parameter to generate `BufferedWriter`.

No	Description
(4)	Use the `Writer` of argument to print footer information.

Bean definition

```
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:resource="file:#{jobParameters[outputFile]}"
      p:footerCallback-ref="writeFooterFlatFileFooterCallback">  <!-- (1) -->
    <property name="lineAggregator">
        <!-- omitted settings -->
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	footerCallback	Set implementation class of `FlatFileFooterCallback`.		

5.3.2.5. Multiple Files

Describe how to handle multiple files.

5.3.2.5.1. Input

To read multiple files of the same record format, use `org.springframework.batch.item.file.MultiResourceItemReader`. `MultiResourceItemReader` can use the specified `ItemReader` to read multiple files specified by regular expressions.

Implement `MultiResourceItemReader` as follows.

- Define bean of `MultiResourceItemReader`
  - Set file to read to property `resources`
    - user regular expression to read multiple files
  - Set `ItemReader` to read files to property `delegate`

Below is a definition example of `MultiResourceItemReader` to read multiple files with the following file names.

File to be read (file name)

```
sales_plan_detail_01.csv
sales_plan_detail_02.csv
sales_plan_detail_03.csv
```

Bean definition

```
<!-- (1) (2) -->
<bean id="multiResourceReader"
      class="org.springframework.batch.item.file.MultiResourceItemReader"
      scope="step"
      p:resources="file:input/sales_plan_detail_*.csv"
      p:delegate-ref="reader"/>
</bean>

<!-- (3) -->
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="lineMapper">
      <!-- omitted settings -->
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	resource	Set multiple input files with regular expressions.	✔	Nothing
(2)	delegate	Set `ItemReader` where it has the actual file read implementation.	✔	Nothing
(3)	`ItemReader` with the actual file read implementation	Bean definition for property `resource` is not needed since it will be automatically set from `MultiResourceItemReader`.	✔	

It is unnecessary to specify resource for ItemReader used by MultiResourceItemReader

The `resource` of `ItemReader` delegated from `MultiResourceItemReader` is automatically set from `MultiResourceItemReader`, so setting in the bean definition is unnecessary.

5.3.2.5.2. Output

Explain how to define multiple files.

To output to a different file for a certain number of cases, use `org.springframework.batch.item.file.MultiResourceItemWriter`.

`MultiResourceItemWriter` can output to multiple files for each number specified using the specified `ItemWriter`.
It is necessary to make the output file name unique so as not to overlap, but `ResourceSuffixCreator` is provided as a mechanism for doing it.
`ResourceSuffixCreator` is a class that generates a suffix that makes the file name unique.

For example, if you want to make the output target file a file name `outputDir / customer_list_01.csv` (`01` part is serial number), set it as follows.

- Set `outputDir/customer_list_` to `MultiResourceItemWriter`
- Implement a code to generate suffix `01.csv`(`01` part is serial number) at `ResourceSuffixCreator`
  - Serial numbers can use the value automatically incremented and passed from `MultiResourceItemWriter`
- `outputDir/customer_list_01.csv` is set to the `ItemWriter` that is actually used

`MultiResourceItemWriter` is defined as follows. How to implement `ResourceSuffixCreator` is described later.

- Define implementation class of `ResourceSuffixCreator`
- Define bean for `MultiResourceItemWriter`
  - Set output file to property `resources`
    - Set the file name up to the suffix given to implementation class of `ResourceSuffixCreator`
  - Set implementation class of `ResourceSuffixCreator` that generates suffix to property `resourceSuffixCreator`
  - Set `ItemWrite` which is to be used to read file to property `delegate`
  - Set the number of output per file to property `itemCountLimitPerResource`

Bean definition

```
<!-- (1) (2) (3) (4) -->
<bean id="multiResourceItemWriter"
      class="org.springframework.batch.item.file.MultiResourceItemWriter"
      scope="step"
      p:resource="file:#{jobParameters[outputDir]}"
      p:resourceSuffixCreator-ref="customerListResourceSuffixCreator"
      p:delegate-ref="writer"
      p:itemCountLimitPerResource="4"/>
</bean>

<!-- (5) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="lineAggregator">
        <!-- omitted settings -->
    </property>
</bean>

<bean id="customerListResourceSuffixCreator"

class="org.terasoluna.batch.functionaltest.ch05.fileaccess.module.CustomerListResource
SuffixCreator"/>  <!-- (6) -->
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	resource	Sets the state before adding the suffix of the output target file. A file name with suffix given automatically by `MultiResourceItemWriter` is set to `ItemWriter`.	✔	Nothing
(2)	resourceSuffixCreator	Set implementation class of `ResourceSuffixCreator`. Default is `org.springframework.batch.item.file.SimpleResourceSuffixCreator` whice generates suffix `"." + index`.		`SimpleResourceSuffixCreator`
(3)	delegate	Set a `ItemWriter` which actually reads the file.	✔	Nothing
(4)	itemCountLimitPerResource	Set the number of output per file.		`Integer.MAX_VALUE`
(5)	`ItemWriter` which actually reads the file.	Property `resource` is not needed since it will be automatically set from `MultiResourceItemWriter`.	✔	

Setting of resource of ItemWrite used by MultiResourceItemWriter is not necessary

Since `Resource` of `ItemWriter` delegated from `MultiResourceItemWriter` is automatically set from `MultiResourceItemWriter`, it is unnecessary to set it in the bean definition.

Implement `ResourceSuffixCreator` as follows.

- Implement `ResourceSuffixCreator` and override getSuffix method
- Use argument's `index` and generate suffix to return
  - `index` is an `int` type value with initial value `1`, and will be incremented for each output file

Sample implementation of ResourceSuffixCreator

```
// (1)
public class CustomerListResourceSuffixCreator implements ResourceSuffixCreator {
    @Override
    public String getSuffix(int index) {
        return String.format("%02d", index) + ".csv";  // (2)
    }
}
```

Item list of setting contents

No	Description
(1)	Implement `ResourceSuffixCreator` class and override getSuffix method.
(2)	Use argument's `index` to generate suffix to return. `index` is an `int` type value with initial value `1`, and will be incremented for each output file.

5.3.2.6. Control Break

How to actually do the Control Break will be described here.

What is Control Break

> Control Break process(or Key Break process) is a process method to read sorted records one by one, and handle records with a certain item(key item) as one group.
> It is an algorithm that is used mainly for aggregating data, continues counting while key items are the same value, and outputs aggregate values when key items become different values.

In order to perform the control break processing, it is necessary to pre-read the record in order to judge the change of the group. Pre-reading records can be done by using `org.springframework.batch.item.support.SingleItemPeekableItemReader`.
Also, control break can be processed only in tasklet model. This is because of the premise that the chunk model is based on, which are "processing N data rows defined by one line" and "transaction boundaries every fixed number of lines", does not fit with the control break's basic algorithm, "proceed at the turn of group".

The execution timing of control break processing and comparison conditions are shown below.

- Execute control break before processing the target record
  - Keep the previously read record, compare previous record with current record
- Execute control break after processing the target record
  - Pre-read the next record by `SingleItemPeekableItemReader` and compare the current record with the next record

A sample for outputting process result from input data using control break is shown below.

Input Data

```
01,2016,10,1000
01,2016,11,1500
01,2016,12,1300
02,2016,12,900
02,2016,12,1200
```

Process Result

```
Header Branch Id : 01,,,
01,2016,10,1000
01,2016,11,1500
01,2016,12,1300
Summary Branch Id : 01,,,3800
Header Branch Id : 02,,,
02,2016,12,900
02,2016,12,1200
Summary Branch Id : 02,,,2100
```

Implementation Sample of Control Break

```
@Component
public class ControlBreakTasklet implements Tasklet {

    @Inject
    SingleItemPeekableItemReader<SalesPerformanceDetail> reader; // (1)

    @Inject
    ItemStreamWriter<SalesPerformanceDetail> writer;

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception {

        // omitted.

        SalesPerformanceDetail previousData = null;   // (2)
        BigDecimal summary = new BigDecimal(0);  //(3)
```

```
        List<SalesPerformanceDetail> items = new ArrayList<>();   // (4)

        try {
            reader.open(executionContext);
            writer.open(executionContext);

            while (reader.peek() != null) {   // (5)
                SalesPerformanceDetail data = reader.read(); // (6)

                // (7)
                if (isBreakByBranchId(previousData, data)) {
                    SalesPerformanceDetail beforeBreakData =
                            new SalesPerformanceDetail();
                    beforeBreakData.setBranchId("Header Branch Id : "
                              + currentData.getBranchId());
                    items.add(beforeBreakData);
                }

                // omitted.
                items.add(data);  // (8)

                SalesPerformanceDetail nextData = reader.peek();  // (9)
                summary = summary.add(data.getAmount());

                // (10)
                SalesPerformanceDetail afterBreakData = null;
                if (isBreakByBranchId(nextData, data)) {
                    afterBreakData = new SalesPerformanceDetail();
                    afterBreakData.setBranchId("Summary Branch Id : "
                            + currentData.getBranchId());
                    afterBreakData.setAmount(summary);
                    items.add(afterBreakData);
                    summary = new BigDecimal(0);
                    writer.write(items);  // (11)
                    items.clear();
                }
                previousData = data;  // (12)
            }
        } finally {
            try {
                reader.close();
            } catch (ItemStreamException e) {
            }
            try {
                writer.close();
            } catch (ItemStreamException e) {
            }
        }
        return RepeatStatus.FINISHED;
    }
    // (13)
```

```
    private boolean isBreakByBranchId(SalesPerformanceDetail o1,
            SalesPerformanceDetail o2) {
        return (o1 == null || !o1.getBranchId().equals(o2.getBranchId()));
    }
}
```

Item list of setting contents

No	Description
(1)	Inject `SingleItemPeekableItemReader`.
(2)	Define a variable to set the previously read record.
(3)	Define a variable to set aggregated values for each group.
(4)	Define a variable to set records for each group including the control break's process result
(5)	Repeat the process until there is no input data.
(6)	Read the record to be processed.
(7)	Execute a control break before target record processing. In the sample, if it is at the beginning of the group, heading is set stored it the variable defined in (4).
(8)	Set the process result to the variable defined in (4).
(9)	Pre-read the next record.
(10)	Execute a control break after target record processing. In this case, if it is at the end of the group, the aggregated data is set in the trailer and stored in the variable defined in (4).
(11)	Output processing results for each group.
(12)	Store the processing record in the variable defined in (2).
(13)	Judge whether the key item has switched or not.

Bean definition

```
<!-- (1) -->
<bean id="reader"
      class="org.springframework.batch.item.support.SingleItemPeekableItemReader"
      p:delegate-ref="delegateReader" />  <!-- (2) -->

<!-- (3) -->
<bean id="delegateReader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[inputFile]}">
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                      p:names="branchId,year,month,customerId,amount"/>
            </property>
            <property name="fieldSetMapper">
                <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

p:targetType="org.terasoluna.batch.functionaltest.app.model.performance.SalesPerforman
ceDetail"/>
            </property>
        </bean>
    </property>
</bean>
```

Item list of setting contents

No	Description
(1)	Define bean for `SingleItemPeekableItemReader`. It will be injected to the Tasklet.
(2)	Set the bean of ItemReader that actually reads the file to `delegate` property.
(3)	Define a bean for ItemReader that actually read the file.

5.3.3. How To Extend

Here, an explanation will be written based on the below case.

- Implmementation of FieldSetMapper
- Input/Output of XML File
- Input/Output of Multi format

5.3.3.1. Implmementation of FieldSetMapper

Explain how to implement `FieldSetMapper` yourself.

Implement `FieldSetMapper` class as follows.

- Implement `FieldSetMapper` class and override mapFieldSet method.
- Get the value from argument's `FieldSet`, do any process needed, and then set it to the conversion target bean as a return value
  - The `FieldSet` class is a class that holds data in association with an index or name, as in the JDBC `ResultSet` class
  - The `FieldSet` class holds the value of each field of a record divided by `LineTokenizer`
  - You can store and retrieve values by specifying an index or name

Here is sample implementation for reading a file that includes data that needs to be converted, such as BigDecimal type with comma and Date type of Japanese calendar format.

Input File Sample

```
"000001","平成28年1月1日","000000001","1,000,000,000"
"000002","平成29年2月2日","000000002","2,000,000,000"
"000003","平成30年3月3日","000000003","3,000,000,000"
```

Input file specification

No	Field Name	Data Type	Note
(1)	branchId	String	
(2)	Date	Date	Japanese calendar format
(3)	customerId	String	
(4)	amount	BigDecimal	include comma

Class to be converted

```
public class UseDateSalesPlanDetail {

    private String branchId;
    private Date date;
    private String customerId;
    private BigDecimal amount;

    // omitted getter/setter
}
```

Implementation Sample of FieldSetMapper

```
@Component
public class UseDateSalesPlanDetailFieldSetMapper implements FieldSetMapper
<UseDateSalesPlanDetail> {  // (1)
    /**
     * {@inheritDoc}
     *
     * @param fieldSet {@inheritDoc}
     * @return Sales performance detail.
     * @throws BindException {@inheritDoc}
     */
    @Override
    public UseDateSalesPlanDetail mapFieldSet(FieldSet fieldSet) throws BindException
{
        UseDateSalesPlanDetail item = new UseDateSalesPlanDetail();  // (2)

        item.setBranchId(fieldSet.readString("branchId"));  // (3)

        // (4)
        DateFormat japaneseFormat = new SimpleDateFormat("GGGGy年M月d日", new Locale
("ja", "JP", "JP"));
        try {
            item.setDate(japaneseFormat.parse(fieldSet.readString("date")));
        } catch (ParseException e) {
            // omitted exception handling
        }

        // (5)
        item.setCustomerId(fieldSet.readString("customerId"));

        // (6)
        DecimalFormat decimalFormat = new DecimalFormat();
        decimalFormat.setParseBigDecimal(true);
        try {
            item.setAmount((BigDecimal) decimalFormat.parse(fieldSet.readString
("amount")));
        } catch (ParseException e) {
            // omitted exception handling
        }

        return item;  // (7)
    }
}
```

Item list of setting contents

No	Description
(1)	Implement `FieldSetMapper` class and override mapFieldSet method. Set conversion target class for type argument of `FieldSetMapper`.

No	Description
(2)	Define a variable of conversion target class to store converted data.
(3)	Get `branchId` from argument's `FieldSet`, and store it to conversion target class variable. Conversion for `branchId` is not done in the sample since it is not necessary.
(4)	Get `date` from argument's `FieldSet`, and store it to conversion target class variable. Use `SimpleDateFormat` to convert Japanese calendar format date to Date type value.
(5)	Get `customerId` from argument's `FieldSet`, and store it to conversion target class variable. Conversion for `customerId` is not done in the sample since it is not necessary.
(4)	Get `amount` from argument's `FieldSet`, and store it to conversion target class variable. Use `DecimalFormat` to convert value with comma to BigDecimal type value.
(7)	Return the conversion target class holding the processing result.

Getting value from FieldSet class

The `FieldSet` class has methods corresponding to various data types for obtaining stored values such as listed below.
When generating `FieldSet` if data is stored in association with the field name, it is possible to get data by specifying that name or by specifying the index.

- `readString()`
- `readInt()`
- `readBigDecimal()`

etc

5.3.3.2. XML File

Describe the definition method when dealing with XML files.

For the conversion process between Bean and XML (O / X (Object / XML) mapping), use the library provided by Spring Framework.
Implementation classes are provided as `Marshaller` and `Unmarshaller` using XStream, JAXB, etc. as libraries for converting between XML files and objects.
Use one suitable for your situation.

Below are features and points for adopting JAXB and XStream.

JAXB

- Specify the bean to be converted in the bean definition file
- Validation using a schema file can be performed
- It is useful when the schema is defined externally and the specification of the input file is strictly determined

XStream

- You can map XML elements and bean fields flexibly in the bean definition file

- It is useful when you need to flexibly map beans

Here is a sample using JAXB.

5.3.3.2.1. Input

For inputting XML file, use `org.springframework.batch.item.xml.StaxEventItemReader` provided by Spring Batch.
`StaxEventItemReader` can read the XML file by mapping the XML file to the bean using the specified `Unmarshaller`.

Implement `StaxEventItemReader` as follows.

- Add `@XmlRootElement` to the conversion target class of XML root element
- Set below property to `StaxEventItemReader`
  - Set the file to read to property `resource`
  - Set the name of the root element to property `fragmentRootElementName`
  - Set `org.springframework.oxm.jaxb.Jaxb2Marshaller` to property `unmarshaller`
- Set below property to `Jaxb2Marshaller`
  - Set conversion target classs in list format to property `classesToBeBound`
  - To validate using schema file, set the 2 properties as below
    - Set the schema file for validation to property `schema`
    - Set implementation class of `ValidationEventHandler` to property `validationEventHandler` to handle events occured during the validation

Here is the sample setting to read the input file below.

Input File Sample

```
<?xml version="1.0" encoding="UTF-8"?>
<records>
    <SalesPlanDetail>
        <branchId>000001</branchId>
        <year>2016</year>
        <month>1</month>
        <customerId>0000000001</customerId>
        <amount>1000000000</amount>
    </SalesPlanDetail>
    <SalesPlanDetail>
        <branchId>000002</branchId>
        <year>2017</year>
        <month>2</month>
        <customerId>0000000002</customerId>
        <amount>2000000000</amount>
    </SalesPlanDetail>
    <SalesPlanDetail>
        <branchId>000003</branchId>
        <year>2018</year>
        <month>3</month>
        <customerId>0000000003</customerId>
        <amount>3000000000</amount>
    </SalesPlanDetail>
</records>
```

Class to be converted

```
@XmlRootElement(name = "SalesPlanDetail")  // (1)
public class SalesPlanDetailToJaxb {

    private String branchId;
    private int year;
    private int month;
    private String customerId;
    private BigDecimal amount;

    // omitted getter/setter
}
```

Item list of setting contents

No	Description
(1)	Add `@XmlRootElement` for the XML root element. Set `SalesPlanDetail` for the tag name.

The setting for reading the above file is as follows.

Bean definition

```
<!-- (1) (2) (3) -->
<bean id="reader"
      class="org.springframework.batch.item.xml.StaxEventItemReader" scope="step"
      p:resource="file:#{jobParameters[inputFile]}"
      p:fragmentRootElementName="SalesPlanDetail"
      p:strict="true">
    <property name="unmarshaller">  <!-- (4) -->
        <!-- (5) (6) -->
        <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller"
              p:schema="file:files/test/input/ch05/fileaccess/SalesPlanDetail.xsd"
              p:validationEventHandler-ref="salesPlanDetailValidationEventHandler">
            <property name="classesToBeBound">  <!-- (7) -->
                <list>

<value>org.terasoluna.batch.functionaltest.ch05.fileaccess.model.plan.SalesPlanDetailT
oJaxb</value>
                </list>
            </property>
        </bean>
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	resource	Set the input file.	✔	Nothing
(2)	fragmentRootElementName	Set the name of the root element. If there are several target objects, use `fragmentRootElementNames`.		Nothing
(3)	strict	If true is set, an exception occurs if the input file does not exist(can not be opened).		true
(4)	unmarshaller	Set the unmarshaller. Set Bean of `org.springframework.oxm.jaxb.Jaxb2Marshaller` when using JAXB.	✔	Nothing
(5)	schema	Set shema file for validation.		
(6)	validationEventHandler	Set implementation class of `ValidationEventHandler` to handle events occured during the validation. Sample implementation of `ValidationEventHandler` is described later on.		
(7)	classesToBeBound	Set conversion target classes in list format.	✔	Nothing

Sample implementation of ValidationEventHandler

```
@Component
// (1)
public class SalesPlanDetailValidationEventHandler implements ValidationEventHandler {
    /**
     * Logger.
     */
    private static final Logger logger =
            LoggerFactory.getLogger(SalesPlanDetailValidationEventHandler.class);

    @Override
    public boolean handleEvent(ValidationEvent event) {
        // (2)
        logger.error("[EVENT [SEVERITY:{}] [MESSAGE:{}] [LINKED EXCEPTION:{}]" +
                " [LOCATOR: [LINE NUMBER:{}] [COLUMN NUMBER:{}] [OFFSET:{}]" +
                " [OBJECT:{}] [NODE:{}] [URL:{}] ] ]",
                event.getSeverity(),
                event.getMessage(),
                event.getLinkedException(),
                event.getLocator().getLineNumber(),
                event.getLocator().getColumnNumber(),
                event.getLocator().getOffset(),
                event.getLocator().getObject(),
                event.getLocator().getNode(),
                event.getLocator().getURL());
        return false;  // (3)
    }
}
```

Item list of setting contents

No	Description
(1)	Implement `ValidationEventHandler` class and override handleEvent method.
(2)	Get event information from argument's event(`ValidationEvent`), and do any process needed. In the sample, logging is proceeded.
(3)	Return false to end the search process. Return true to continue the search process. Return false to end this operation by generating appropriate `UnmarshalException`, `ValidationException` or `MarshalException`.

Adding dependency library

Library dependency needs to be added as below when using Spring Object/Xml Marshalling provided by Spring Framework such as `org.springframework.oxm.jaxb.Jaxb2Marshaller`.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
</dependency>
```

5.3.3.2.2. Output

Use `org.springframework.batch.item.xml.StaxEventItemWriter` provided by Spring Batch for outputting XML file. `StaxEventItemWriter` can output an XML file by mapping the bean to XML using the specified `Marshaller`.

Implement `StaxEventItemWriter` as follows.

- Do the below setting to conversion target class
  - Add `@XmlRootElement` to the class as it is to be the root element of the XML
  - Use `@XmlType` annotation to set orders for outputting fields
  - If there is a field to be excluded from conversion to XML, add `@XmlTransient` to the getter method of it's field
- Set below properties to `StaxEventItemWriter`
  - Set output target file to property `resource`
  - Set `org.springframework.oxm.jaxb.Jaxb2Marshaller` to property `marshaller`
- Set below property to `Jaxb2Marshaller`
  - Set conversion target classes in list format to property `classesToBeBound`

Here is a sample for outputting below file.

Output file example

```
<?xml version="1.0" encoding="UTF-8"?>
<records>
  <Customer>
    <customerId>001</customerId>
    <customerName>CustomerName001</customerName>
    <customerAddress>CustomerAddress001</customerAddress>
    <customerTel>11111111111</customerTel>
    <chargeBranchId>001</chargeBranchId></Customer>
  <Customer>
    <customerId>002</customerId>
    <customerName>CustomerName002</customerName>
    <customerAddress>CustomerAddress002</customerAddress>
    <customerTel>11111111111</customerTel>
    <chargeBranchId>002</chargeBranchId></Customer>
  <Customer>
    <customerId>003</customerId>
    <customerName>CustomerName003</customerName>
    <customerAddress>CustomerAddress003</customerAddress>
    <customerTel>11111111111</customerTel>
    <chargeBranchId>003</chargeBranchId>
  </Customer>
</records>
```

About XML file fomatX(line break and indents)

In the sample above, the output XML file has been formatted(has line break and indents), but the actual XML will not be formatted.

`Jaxb2Marshaller` has a function to format it when outputting the XML file, but it does not work as is it expected.
This issue is being discussed in the Spring Forum, and might be fixed in the future.

To avoid this and output the formatted XML, set `marshallerProperties` as below.

```
<property name="marshaller">
    <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <!-- omitted settings -->
        </property>
        <property name="marshallerProperties">
            <map>
                <entry>
                    <key>
                        <util:constant
                            static-
field="javax.xml.bind.Marshaller.JAXB_FORMATTED_OUTPUT"/>
                    </key>
                    <value type="java.lang.Boolean">true</value>
                </entry>
            </map>
        </property>
    </bean>
</property>
```

Class to be converted

```
@XmlRootElement(name = "Customer")  // (2)
@XmlType(propOrder={"customerId", "customerName", "customerAddress",
        "customerTel", "chargeBranchId"})  // (2)
public class CustomerToJaxb {

    private String customerId;
    private String customerName;
    private String customerAddress;
    private String customerTel;
    private String chargeBranchId;
    private Timestamp createDate;
    private Timestamp updateDate;

    // omitted getter/setter

    @XmlTransient  // (3)
    public Timestamp getCreateDate() { return createDate; }

    @XmlTransient  // (3)
    public Timestamp getUpdateDate() { return updateDate; }
}
```

Item list of setting contents

No	Description
(1)	Add `@XmlRootElement` annotation to make this as the root element of XML. Set `Customer` for the tag name.
(2)	Use `@XmlType` annotation to set field output order.
(3)	Add `@XmlTransient` to getter method of fileds which is to be excluded from XML conversion.

The settings for writing the above file are as follows.

Bean definition

```
<!-- (1) (2) (3) (4) (5) (6) -->
<bean id="writer"
      class="org.springframework.batch.item.xml.StaxEventItemWriter" scope="step"
      p:resource="file:#{jobParameters[outputFile]}"
      p:encoding="MS932"
      p:rootTagName="records"
      p:overwriteOutput="true"
      p:shouldDeleteIfEmpty="false"
      p:transactional="true">
    <property name="marshaller">  <!-- (7) -->
        <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
            <property name="classesToBeBound">  <!-- (8) -->
                <list>

<value>org.terasoluna.batch.functionaltest.ch05.fileaccess.model.mst.CustomerToJaxb</v
alue>
                </list>
            </property>
        </bean>
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	resource	Set output file	✔	Nothing
(2)	encoding	Set character encoding for output file.		JavaVM default character set
(3)	rootTagName	Set XML root tag name.		
(4)	overwriteOutput	If true, delete the file if it already exists. If false, throw an exception if the file already exists.		true
(5)	shouldDeleteIfEmpty	If true, delete the output if it is an empty file.		false
(6)	transactional	Set whether to perform transaction control. For details, see Transaction Control.		true
(7)	marshaller	Set the marshaller. Set `org.springframework.oxm.jaxb.Jaxb2Marshaller` when using JAXB.	✔	Nothing
(8)	classesToBeBound	Set conversion target classes in list format.	✔	Nothing

Adding dependency library

Library dependency needs to be added as below when using Spring Object/Xml Marshalling provided by Spring Framework such as `org.springframework.oxm.jaxb.Jaxb2Marshaller`.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
</dependency>
```

Output Header / Footer

For output of header and footer, use the implementation class of `org.springframework.batch.item.xml.StaxWriterCallback`.

Set implementation of `headerCallback` for header output, and `footerCallback` for footer output.

Below is a sample of output file.
Header is printed right after the root tag's openning element, and footer is printed right before the root element's closing tag.

Output file example

```
<?xml version="1.0" encoding="UTF-8"?>
<records>
<!-- Customer list header -->
  <Customer>
    <customerId>001</customerId>
    <customerName>CustomerName001</customerName>
    <customerAddress>CustomerAddress001</customerAddress>
    <customerTel>11111111111</customerTel>
    <chargeBranchId>001</chargeBranchId></Customer>
  <Customer>
    <customerId>002</customerId>
    <customerName>CustomerName002</customerName>
    <customerAddress>CustomerAddress002</customerAddress>
    <customerTel>11111111111</customerTel>
    <chargeBranchId>002</chargeBranchId></Customer>
  <Customer>
    <customerId>003</customerId>
    <customerName>CustomerName003</customerName>
    <customerAddress>CustomerAddress003</customerAddress>
    <customerTel>11111111111</customerTel>
    <chargeBranchId>003</chargeBranchId>
  </Customer>
<!-- Customer list footer -->
</records>
```

About XML file fomatX(line break and indents)

In the sample above, the output XML file has been formatted(has line break and indents), but the actual XML will not be formatted.

Refer to About XML file fomatX(line break and indents) for details.

To output the above file, do the setting as below.

Bean definition

```
<!-- (1) (2) -->
<bean id="writer"
      class="org.springframework.batch.item.xml.StaxEventItemWriter" scope="step"
      p:resource="file:#{jobParameters[outputFile]}"
      p:headerCallback-ref="writeHeaderStaxWriterCallback"
      p:footerCallback-ref="writeFooterStaxWriterCallback">
    <property name="marshaller">
        <!-- omitted settings -->
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	headerCallback	Set implementation class of `StaxWriterCallback`.		
(2)	footerCallback	Set implementation class of `StaxWriterCallback`.		

Implement `StaxWriterCallback` as follows.

- Implement `StaxWriterCallback` class and override write method
- Print header/footer by using the argument's `XMLEventWriter`

Implementation Sample of StaxWriterCallback

```
@Component
public class WriteHeaderStaxWriterCallback implements StaxWriterCallback { // (1)
    @Override
    public void write(XMLEventWriter writer) throws IOException {
        XMLEventFactory factory = XMLEventFactory.newInstance();
        try {
            writer.add(factory.createComment(" Customer list header ")); // (2)
        } catch (XMLStreamException e) {
            // omitted exception handling
        }
    }
}
```

Item list of setting contents

No	Description
(1)	Implement `StaxWriterCallback` class and override write method.
(2)	Print header/footer by using the argument's `XMLEventWriter`

> *XML output using XMLEventFactory*
>
> In the output of the XML file using the `XMLEventWriter` class, you can efficiently generate `XMLEvent` by using the `XMLEventFactory` class.
>
> The `XMLEventWriter` class has an add method defined, which takes an XMLEvent object as an argument and outputs an XML file.
> Since it is very time consuming to generate an `XMLEvent` object each time, use the `XMLEventFactory` class which can easily generate `XMLEvent`.
> In the `XMLEventFactory` class, methods corresponding to the event to be created are defined, such as `createStartDocument` method and `createStartElement` method.

5.3.3.3. Multi format

Describe the definition method when dealing with multi format file.

As described in Overview, multi format is basically (Header N Rows + Data N Rows + Trailer N Rows) * N + Footer N Rows format, but there are other format patterns like below.

- When there is a footer record or not
- When there are records with different formats in the same record classification
  - eg) there is a data record that has 5 items and a data record with 6 items in data part

Although there are several patterns to multi format file, implementation method will be the same.

5.3.3.3.1. Input

Use `org.springframework.batch.item.file.mapping.PatternMatchingCompositeLineMapper` provided by Spring Batch for readeing multi format file.
In multi format file, each record needs to be mapped to defferent bean for each format.
`PatternMatchingCompositeLineMapper` will select the `LineTokenizer` and `FieldSetMapper` to use for the record by regular expression.

For example, `LineTokenizers` to use can be selected like below.

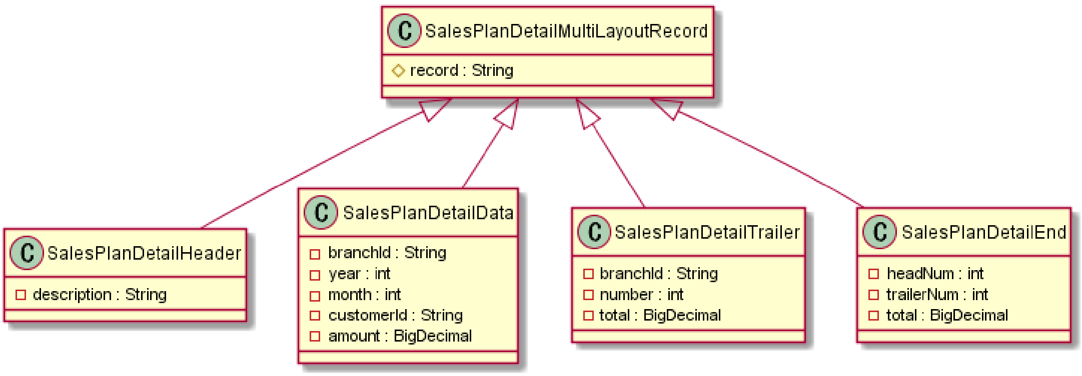
- Use `userTokenizer` if the beginning of the record matches to regular expression `USER*`
- Use `lineATokenizer` if the beginning of the record matches to regular expression `LINEA*`

> *Restrictions on the format of records when reading multi-format files*
>
> In order to read a multi-format file, it must be in a format that can distinguish record classification by regular expression.

Implement `PatternMatchingCompositeLineMapper` as follows.

- Define a class with record division for conversino target class, and inherit this class to each classes of each record division
- Define `LineTokenizer` and `FieldSetMapper` to map each record to bean
- Define `PatternMatchingCompositeLineMapper`
  - Set `LineTokenizer` that correspond to each record division to property `tokenizers`
  - Set `FieldSetMapper` that correspond to each record division to property `fieldSetMappers`

> *Define a class with record division for conversino target class, and inherit this class to each classes of each record division*
>
> `ItemProcessor` has a specification that takes one type as an argument.
>
> However, if you simply map `PatternMatchingCompositeLineMapper` to a multi-format file to a different bean for each record division, `ItemProcessor` can not handle multiple types as it takes one type as an argument.
>
> Therefore, it is possible to solve this by giving an inheritance relation to the class to be converted and specifying a superclass as the type of the argument of `ItemProcessor`.
>
> The class diagram of the conversion target class and the definition sample of `ItemProcessor` are shown below.



Class diagram of conversion target class

Implementation Sample of ItemProcessor

```
public class MultiLayoutItemProcessor implements
        ItemProcessor<SalesPlanDetailMultiLayoutRecord, String> {
    @Override
    // (1)
    public String process(SalesPlanDetailMultiLayoutRecord item)
throws Exception {
        String record = item.getRecord();  // (2)

        switch (record) {  // (3)
        case "H":
            // omitted business logic
        case "D":
            // omitted business logic
        case "T":
            // omitted business logic
        case "E":
            // omitted business logic
        default:
            // omitted exception handling
        }
    }
}
```

Item list of setting contents

No	Description
(1)	Set the superclass of the class to be converted whose inheritance relation is given as the argument of `ItemProcessor`.
(2)	Get the record division from item. Actual classes are different depending on each record division, but record divison can be retrieved by polymorphism.
(3)	Judge the record division and process things needed for each record division. Perform class conversions as needed.

Here is a setting sample and implementation sample for reading below input file.

Input File Sample

```
H,Sales_plan_detail header No.1
D,000001,2016,1,0000000001,100000000
D,000001,2016,1,0000000002,200000000
D,000001,2016,1,0000000003,300000000
T,000001,3,600000000
H,Sales_plan_detail header No.2
D,00002,2016,1,0000000004,400000000
D,00002,2016,1,0000000005,500000000
D,00002,2016,1,0000000006,600000000
T,00002,3,1500000000
H,Sales_plan_detail header No.3
D,00003,2016,1,0000000007,700000000
D,00003,2016,1,0000000008,800000000
D,00003,2016,1,0000000009,900000000
T,00003,3,2400000000
E,3,9,4500000000
```

Below is the bean definition sample of conversion target class.

Class to be converted

```java
/**
 * Model of record indicator of sales plan detail.
 */
public class SalesPlanDetailMultiLayoutRecord {

    protected String record;

    // omitted getter/setter
}

/**
 * Model of sales plan detail header.
 */
public class SalesPlanDetailHeader extends SalesPlanDetailMultiLayoutRecord {

    private String description;

    // omitted getter/setter
}

/**
 * Model of Sales plan Detail.
 */
public class SalesPlanDetailData extends SalesPlanDetailMultiLayoutRecord {

    private String branchId;
```

```java
    private int year;
    private int month;
    private String customerId;
    private BigDecimal amount;

    // omitted getter/setter
}

/**
 * Model of Sales plan Detail.
 */
public class SalesPlanDetailTrailer extends SalesPlanDetailMultiLayoutRecord {

    private String branchId;
    private int number;
    private BigDecimal total;

    // omitted getter/setter
}

/**
 * Model of Sales plan Detail.
 */
public class SalesPlanDetailEnd extends SalesPlanDetailMultiLayoutRecord {
    // omitted getter/setter

    private int headNum;
    private int trailerNum;
    private BigDecimal total;

    // omitted getter/setter
}
```

The setting for reading the above file is as follows.

Bean definition example

```xml
<!-- (1) -->
<bean id="headerDelimitedLineTokenizer"
      class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
      p:names="record,description"/>

<bean id="dataDelimitedLineTokenizer"
      class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
      p:names="record,branchId,year,month,customerId,amount"/>

<bean id="trailerDelimitedLineTokenizer"
      class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
      p:names="record,branchId,number,total"/>

<bean id="endDelimitedLineTokenizer"
```

```
      class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
      p:names="record,headNum,trailerNum,total"/>

<!-- (2) -->
<bean id="headerBeanWrapperFieldSetMapper"
      class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

p:targetType="org.terasoluna.batch.functionaltest.ch05.fileaccess.model.plan.SalesPlan
DetailHeader"/>

<bean id="dataBeanWrapperFieldSetMapper"
      class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

p:targetType="org.terasoluna.batch.functionaltest.ch05.fileaccess.model.plan.SalesPlan
DetailData"/>

<bean id="trailerBeanWrapperFieldSetMapper"
      class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

p:targetType="org.terasoluna.batch.functionaltest.ch05.fileaccess.model.plan.SalesPlan
DetailTrailer"/>

<bean id="endBeanWrapperFieldSetMapper"
      class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

p:targetType="org.terasoluna.batch.functionaltest.ch05.fileaccess.model.plan.SalesPlan
DetailEnd"/>

<bean id="reader"
    class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
    p:resource="file:#{jobParameters[inputFile]}">
    <property name="lineMapper">  <!-- (3) -->
        <bean
class="org.springframework.batch.item.file.mapping.PatternMatchingCompositeLineMapper"
>
            <property name="tokenizers">  <!-- (4) -->
                <map>
                    <entry key="H*" value-ref="headerDelimitedLineTokenizer"/>
                    <entry key="D*" value-ref="dataDelimitedLineTokenizer"/>
                    <entry key="T*" value-ref="trailerDelimitedLineTokenizer"/>
                    <entry key="E*" value-ref="endDelimitedLineTokenizer"/>
                </map>
            </property>
            <property name="fieldSetMappers">  <!-- (5) -->
                <map>
                    <entry key="H*" value-ref="headerBeanWrapperFieldSetMapper"/>
                    <entry key="D*" value-ref="dataBeanWrapperFieldSetMapper"/>
                    <entry key="T*" value-ref="trailerBeanWrapperFieldSetMapper"/>
                    <entry key="E*" value-ref="endBeanWrapperFieldSetMapper"/>
                </map>
            </property>
```

```
        </bean>
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	The `LineTokenizer` corresponding to each record	Define `LineTokenizer` that corresponds to each record.		
(2)	The `FieldSetMapper` corresponding to each record	Define `FieldSetMapper` that corresponds to each record.		
(3)	lineMapper	Set `org.springframework.batch.item.file.mapping.PatternMatchingCompositeLineMapper`.	✔	Nothing
(3)	tokenizers	Set `LineTokenizer` corresponding to each record in map format. Set regular expression to determine record for `key`, and set the `LineTokenizer` to use for `value-ref`.	✔	Nothing
(4)	tokenizers	Set `FieldSetMapper` corresponding to each record in map format. Set regular expression to determine record for `key`, and set the `FieldSetMapper` to use for `value-ref`.	✔	Nothing

5.3.3.3.2. Output

Describe the definition method when dealing with multi format file.

For reading multi format file `PatternMatchingCompositeLineMapper` was provided to determine which `LineTokenizer` and `FieldSetMapper` to use for each record division. However for writing, no similar components are provided.

Therefore, processing up to conversion target class to record (character string) within `ItemProcessor` is carried out, and `ItemWriter` writes the received character string as it is to achieve writing of multi format file .

Implement multi format output as follows.

- `ItemProcessor` converts the conversion target class to a record (character string) and passes it to `ItemWriter`
  - In the sample, define `LineAggregator` and `FieldExtractor` for each record division and use it by injecting it with `ItemProcessor`
- `ItemWriter` writes the received character string as it is to the file

- Set `PassThroughLineAggregator` to property `lineAggregator` of `ItemWriter`
- `PassThroughLineAggregator` is `LineAggregator` which returns `item.toString ()` result of received item

Here is a setting sample and implementation sample for writing below output file.

Output file example

```
H,Sales_plan_detail header No.1
D,000001,2016,1,0000000001,100000000
D,000001,2016,1,0000000002,200000000
D,000001,2016,1,0000000003,300000000
T,000001,3,600000000
H,Sales_plan_detail header No.2
D,00002,2016,1,0000000004,400000000
D,00002,2016,1,0000000005,500000000
D,00002,2016,1,0000000006,600000000
T,00002,3,1500000000
H,Sales_plan_detail header No.3
D,00003,2016,1,0000000007,700000000
D,00003,2016,1,0000000008,800000000
D,00003,2016,1,0000000009,900000000
T,00003,3,2400000000
E,3,9,4500000000
```

Definition of conversion target class and `ItemProcessor` sample, notes are the same as Multi format Input.

Settings to output above file is as below. Bean definition sample for `ItemProcessor` is written later.

Bean definition example

```
<!-- (1) -->
<bean id="headerDelimitedLineAggregator"
      class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
    <property name="fieldExtractor">
        <bean
class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
              p:names="record,description"/>
    </property>
</bean>

<bean id="dataDelimitedLineAggregator"
      class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
    <property name="fieldExtractor">
        <bean
class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
              p:names="record,branchId,year,month,customerId,amount"/>
    </property>
</bean>

<bean id="trailerDelimitedLineAggregator"
      class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
    <property name="fieldExtractor">
        <bean
class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
              p:names="record,branchId,number,total"/>
    </property>
</bean>

<bean id="endDelimitedLineAggregator"
      class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
    <property name="fieldExtractor">
        <bean
class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
              p:names="record,headNum,trailerNum,total"/>
    </property>
</bean>


<bean id="writer" class="org.springframework.batch.item.file.FlatFileItemWriter"
scope="step"
      p:resource="file:#{jobParameters[outputFile]}"/>
    <property name="lineAggregator">  <!-- (2) -->
        <bean
class="org.springframework.batch.item.file.transform.PassThroughLineAggregator"/>
    </property>
</bean>
```

Item list of setting contents

No	Property Name	Setting contents	Required	Default Value
(1)	The `LineAggregator` and `FieldExtractor` corresponding to each record division	Define `LineAggregator` and `FieldExtractor`. Use `LineAggregator` by injecting it to `ItemProcessor`.		
(2)	lineAggregator	Set `org.springframework.batch.item.file.transform.PassThroughLineAggregator`.	✔	Nothing

Implementation sample of `ItemProcessor` is shown below.
In this sample, only the process of converting the received item to a string and passing it to `ItemWriter` is performed.

Sample Implementation of ItemProcessor

```
public class MultiLayoutItemProcessor implements
        ItemProcessor<SalesPlanDetailMultiLayoutRecord, String> {

    // (1)
    @Inject
    @Named("headerDelimitedLineAggregator")
    DelimitedLineAggregator<SalesPlanDetailMultiLayoutRecord>
headerDelimitedLineAggregator;

    @Inject
    @Named("dataDelimitedLineAggregator")
    DelimitedLineAggregator<SalesPlanDetailMultiLayoutRecord>
dataDelimitedLineAggregator;

    @Inject
    @Named("trailerDelimitedLineAggregator")
    DelimitedLineAggregator<SalesPlanDetailMultiLayoutRecord>
trailerDelimitedLineAggregator;

    @Inject
    @Named("endDelimitedLineAggregator")
    DelimitedLineAggregator<SalesPlanDetailMultiLayoutRecord>
endDelimitedLineAggregator;

    @Override
    // (2)
    public String process(SalesPlanDetailMultiLayoutRecord item) throws Exception {
        String record = item.getRecord();  // (3)

        switch (record) {  // (4)
        case "H":
            return headerDelimitedLineAggregator.aggregate(item);  // (5)
        case "D":
            return dataDelimitedLineAggregator.aggregate(item);  // (5)
        case "T":
            return trailerDelimitedLineAggregator.aggregate(item);  // (5)
        case "E":
            return endDelimitedLineAggregator.aggregate(item);  // (5)
        default:
            throw new IncorrectRecordClassificationException(
                    "Record classification is incorrect.[value:" + record + "]");
        }
    }
}
```

Item list of setting contents

No	Description
(1)	Inject `LineAggregator` corresponding to each record division.
(2)	Set the superclass of the class to be converted whose inheritance relation is given as the argument of `ItemProcessor`.
(3)	Get the record division from item.
(4)	Judge record division and do any process for each record division.
(5)	Use `LineAggregator` corresponding to each record division to convert the conversion target class to a record (character string) and pass it to `ItemWriter`.

5.4. Exclusive Control

5.4.1. Overview

Exclusive control is a process performed to maintain consistency of data when update processing is performed simultaneously for the same resource from multiple transactions. In the case where there is a possibility that updating processing is performed simultaneously for the same resource from multiple transactions, it is basically necessary to perform exclusive control.

Multiple transaction means following in this chapter.

- Transaction at the time of simultaneous execution of multiple jobs
- Transaction at the time of simultaneous execution with online processing

> *Exclusive control of multiple jobs*
>
> When multiple jobs are executed at the same time, it is fundamental to design jobs so that exclusive control is not required. This means that it is basic to divide the resources to be accessed and the processing target for each job.

Since the concept of exclusive control is the same as online processing, please refer to Exclusive Control in TERASOLUNA Server 5.x Development Guideline

Here, focus on the part not described in TERASOLUNA Server 5.x.

The usage method of this function is same in the chunk model as well as tasklet model.

5.4.1.1. Necessity of Exclusive Control

For the necessity of exclusive control, please refer to Necessity of Exclusive Control in TERASOLUNA Server 5.x Development Guideline.

5.4.1.2. Exclusive Control for File

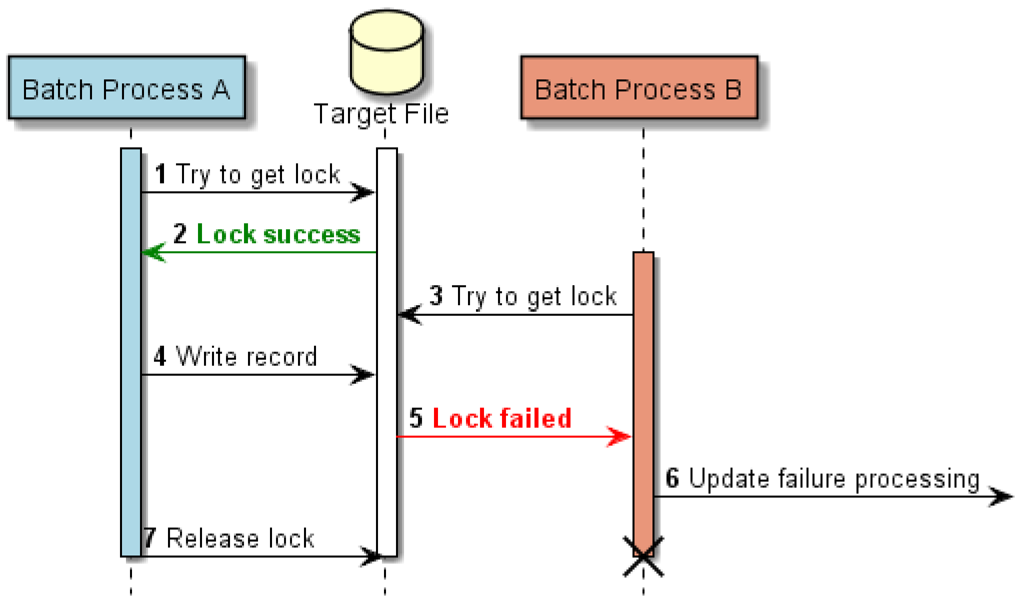
Exclusive control for file is generally implemented by file locking.

File Locking

File locking is a mechanism for restricting reading and writing from other programs while using files with a certain program. The outline of file lock processing is as follows.

Scenario

- The batch process A acquires the lock of the file and starts the file updating process.
- Batch process B attempts to update the same file and attempts to acquire the file lock fails.
- The batch process A ends the processing and unlocks the file



Overview of File Lock Processing

1. The Batch Process A tries to acquire the lock of the Target File.
2. The Batch Process A succeeds in acquiring the lock of the Target File.
3. The Batch Process B tries to acquire the lock of the Target File.
4. The Batch Process A writes the Target File.
5. Since the Batch Process A is locked the Target File, the Batch Process B fails to acquire the lock of the Target File.
6. The Batch Process B performs processing of file update failure.
7. The Batch Process A releases the lock of the Target File.

Prevention of Deadlock

Even in a file,in the same as a database, when acquiring a lock on multiple files, deadlock may occur in some cases. Therefore, it is important to make a rule the update order of files.
The prevention of deadlock is similar to prevention of deadlock between tables in the database. For details, refer to Prevention of deadlock in TERASOLUNA Server 5.x Development Guideline.

5.4.1.3. Exclusive Control of Database

For details About Exclusive Control of Database, refer to Exclusive control using database locking in TERASOLUNA Server 5.x Development Guideline.

5.4.1.4. Choose Exclusive Control Scheme

Explain the locking scheme and suitable situation for TERASOLUNA Batch 5.x.

Choose exclusive control scheme

Lock scheme	Suitable situation
Optimistic locking	The case where process can be continued with the update result of another transaction being out of processing targets in a transaction at the time of concurrent execution.
Pessimistic locking	Process in which the processing time is long and it is difficult to redo because the situation of the target data has changed during processing Process requiring exclusive control for files

5.4.1.5. Relationship between Exclusive Control and Components

The relationship between each component provided by TERASOLUNA Batch 5.x and exclusive control is as follows.

Optimistic lock

Relationship between exclusive control and components

Processing model	Component	File	Database
Chunk	ItemReader	-	Acquires data including a column that can confirm that the same data is obtained at the time of acquiring and updating such as Version column.
	ItemProcessor	-	Exclusive control is unnecessary.
	ItemWriter	-	Check the difference between acquisition and update, confirm that it is not updated by other processing, then update.
Tasklet	Tasklet	-	When acquiring data, execute the processing described in the ItemReader section, and when updating the data, the processing described in ItemWriter section. The concept is the same when using the Mapper interface directly.

Optimistic lock on files

Because of the characteristic of the file, do not apply optimistic lock on files.

Pessimistic lock

Relationship between exclusive control and components

Processing model	Component	File	Database
Chunk	ItemReader	-	Use FOR UPDATE of SQL statement.
	ItemProcessor		Since it is fundamental to handle locked data, in principle, exclusive control is not performed here.
	ItemWriter	-	Update data without conscious of exclusion.
Tasklet	Tasklet	Get a file lock right after opening a file with ItemStreamReader. Release the file lock just before closing ItemStreamWriter.	When acquiring data, execute the processing described in the ItemReader section, and when updating the data, the processing described in ItemWriter section. The concept is the same when using the Mapper interface directly.

Pessimistic lock on file

Pessimistic lock on files should be implemented in the tasklet model. In the chunk model, due to its structure, there is a period that can not be excluded in the gap of chunk processing. Also, it is assumed that file access is done by Injecting ItemStreamReader / ItemStreamWriter.

Waiting time due to Pessimistic lock in database

When pessimistic locking is performed, the wait time for processing due to contention may be prolonged. In that case, it is reasonable to use the pessimistic lock by specifying the NO WAIT option and the timeout time.

5.4.2. How to use

Explain how to use exclusive control by resource.

- Exclusive control of file
- Exclusive Control of Database

5.4.2.1. Exclusive control of file

Exclusive control of file with TERASOLUNA Batch 5.x is realized by implementing Tasklet. As a means of achieving exclusion, exclusive control is performed by file lock acquisition using the `java.nio.channels.FileChannel` class.

Details of the FileChannel class

For details and how to use `FileChannel` class, refer to Javadoc.

Show an example of using `FileChannel` class to get a file lock.

Tasklet implementation

```
@Component
@Scope("step")
public class FileExcusiveTasklet implements Tasklet {

    private String targetPath = null; // (1)

    @Inject
    ItemStreamReader<SalesPlanDetail> reader;

    @Inject
    ItemStreamWriter<SalesPlanDetailWithProcessName> writer;

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception {

        // omitted.

        FileChannel fc = null;
        FileLock fileLock = null;

        try {
            reader.open(executionConetxt);
            writer.open(executionConetxt);  // (2)

            try {
                File file = new File(targetPath);
                fc = FileChannel.open(file.toPath(), StandardOpenOption.WRITE,
                        StandardOpenOption.CREATE,
                        StandardOpenOption.APPEND); // (3)
                fileLock = fc.tryLock();  // (4)
            } catch (IOException e) {
                logger.error("Failure other than lock acquisition", e);
                throw new FailedOtherAcquireLockException(
                        "Failure other than lock acquisition", e);
            }
            if (fileLock == null) {
                logger.error("Failed to acquire lock. [processName={}]", processName);
                throw new FailedAcquireLockException("Failed to acquire lock");
            }

            // (5)
            SalesPlanDetail item;
            List<SalesPlanDetailWithProcessName> items = new ArrayList<>();
            while ((item = reader.read()) != null) {

                // omitted.
```

```
                items.add(item);
                if (items.size() >= 10) {
                    writer.write(items);
                    items.clear();
                }
            }
            if (items.size() > 0) {
                writer.write(items);
            }

        } finally {
            if (fileLock != null) {
                try {
                    fileLock.release();  // (6)
                } catch (IOException e) {
                    logger.warn("Lock release failed.", e);
                }
            }
            if (fc != null) {
                try {
                    fc.close();
                } catch (IOException e) {
                    // ignore
                }
            }
            writer.close(); // (7)
            reader.close();
        }
        return RepeatStatus.FINISHED;
    }

    // (8)
    @Value("#{jobParameters[outputFile]}")
    public void setTargetPath(String targetPath) {
        this.targetPath = targetPath;
    }
}
```

Description

Sr. No.	Description
(1)	File path to be exclusively controlled.
(2)	Open file to be exclusively controlled.
(3)	Get file channel. In this example, channels for new creation, addition and writing of files are obtained.
(4)	Get file lock.
(5)	Execute business logic with file output.

Sr. No.	Description
(6)	Release file lock.
(7)	Close file to be exclusively controlled.
(8)	Set file path. In this example, it receives from the job parameter.

About the method of FileChannel used for lock acquisition

It is recommended to use the `tryLock()` method which is not waiting because the `lock()` method waits until the lock is released if the target file is locked. Note that trylock() can select shared lock and exclusive lock, but in batch processing, exclusive lock is normally used.

Exclusive control between threads in the same VM

Attention must be paid to exclusive control between threads in the same VM. When processing files between threads in the same VM, the lock function using the `FileChannel` class can not determine whether a file is locked by another thread's processing.
Therefore, exclusive control between threads does not function. In order to avoid this, exclusive control between threads can be performed by performing synchronization processing in the part where writing to the file is performed. However, synchronizing reduces the merit of parallel processing, and it is not different from processing with a single thread. As a result, since it is not suitable to perform exclusive control with different threads on the same file and process it, do not design and implement such processing.

About appendAllowed property of FlatFileItemWriter

When creating (overwriting) a file, exclusive control can be realized by setting the `appendAllowed` property to `false` (default). This is because `FileChannel` is controlled inside `FlatFileItemWriter`. However, if the file is appended (`appendAllowed` property is `true`), developers need to implement exclusive control with `FileChannel`.

5.4.2.2. Exclusive Control of Database

Explain exclusive control of database in TERASOLUNA Batch 5.x.

The exclusive control implementation of the database is basically How to implement while using MyBatis3 in TERASOLUNA Server 5.x Development Guideline. In this guideline, Explain it on the premise that How to implement while using MyBatis3 is done.

As shown in Relationship between Exclusive Control and Components, there are variations due to combination of processing model and component.

Variation of exclusive control of database

Exclusive control scheme	Processing model	Component
Optimistic lock	Chunk model	ItemReader/ItemWriter
	Tasklet model	ItemReader/ItemWriter
		Mapper interface
Pessimistic lock	Chunk model	ItemReader/ItemWriter
	Tasklet model	ItemReader/ItemWriter
		Mapper interface

When using the Mapper interface in tasklet model, the explanation is omitted. Refer to How to implement while using MyBatis3.

When using ItemReader/ItemWriter in tasklet model, the calling part in the Mapper interface is replaced by ItemReader/ItemWriter, so the explanation is also omitted.

Therefore, exclusive control of chunk model will be explained here.

5.4.2.2.1. Optimistic Lock

Explain Optimistic lock in chunk model.

Since the behavior of the job changes according to the setting of the `assertUpdates` property of MyBatisBatchItemWriter, it is necessary to set it appropriately according to the business requirements.

Show the job definition for optimistic lock.

job definition

```
<!-- (1) -->
<bean id="reader"
      class="org.mybatis.spring.batch.MyBatisCursorItemReader" scope="step"

p:queryId="org.terasoluna.batch.functionaltest.ch05.exclusivecontrol.repository.Exclus
iveControlRepository.branchFindOne"
      p:sqlSessionFactory-ref="jobSqlSessionFactory"/>
    <property name="parameterValues">
        <map>
            <entry key="branchId" value="#{jobParameters[branchId]}"/>
        </map>
    </property>
</bean>

<!-- (2) --->
<bean id="writer"
      class="org.mybatis.spring.batch.MyBatisBatchItemWriter" scope="step"

p:statementId="org.terasoluna.batch.functionaltest.ch05.exclusivecontrol.repository.Ex
clusiveControlRepository.branchExclusiveUpdate"
      p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"
      p:assertUpdates="true" />  <!-- (3) -->

<batch:job id="chunkOptimisticLockCheckJob" job-repository="jobRepository">
    <batch:step id="chunkOptimisticLockCheckJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader" processor="branchEditItemProcessor"
                         writer="writer" commit-interval="10" />
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Description

Sr. No.	Description
(1)	Set SQLID of data acquisition by optimistic lock.
(2)	Set SQLID of data update by optimistic lock.
(3)	Set whether to check the number of batch updates. If set to `true` (default), throw an exception if the number of updates is 0. if set to `false`, perform normal processing even if the number of updates is 0.

5.4.2.2.2. Pessimistic Lock

Explain pessimistic lock in chunk model.

Show the job definition for pessimistic lock.

job definition

```xml
<!-- (1) -->
<bean id="reader"
      class="org.mybatis.spring.batch.MyBatisCursorItemReader" scope="step"

p:queryId="org.terasoluna.batch.functionaltest.ch05.exclusivecontrol.repository.Exclus
iveControlRepository.branchFindOneWithNowWaitLock"
      p:sqlSessionFactory-ref="jobSqlSessionFactory">
    <property name="parameterValues">
        <map>
            <entry key="branchId" value="#{jobParameters[branchId]}"/>
        </map>
    </property>
</bean>

<!-- (2) -->
<bean id="writer"
      class="org.mybatis.spring.batch.MyBatisBatchItemWriter" scope="step"

p:statementId="org.terasoluna.batch.functionaltest.ch05.exclusivecontrol.repository.Ex
clusiveControlRepository.branchUpdate"
      p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"
      p:assertUpdates="#{new Boolean(jobParameters[assertUpdates])}" />

<batch:job id="chunkPessimisticLockCheckJob" job-repository="jobRepository">
    <batch:step id="chunkPessimisticLockCheckJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader" processor="branchEditItemProcessor"
                         writer="writer" commit-interval="10" />
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="jobExecutionLoggingListener"/>
    </batch:listeners>
</batch:job>
```

Description

Sr. No.	Description
(1)	Set SQLID of data acquisition by pessimistic lock.
(2)	Set the same SQLID as SQL of data update without exclusive control.

Behavior during exclusive control

If performing pessimistic lock by setting NO WAIT or timeout, when excluded by another transaction, an exception is thrown in the `doOpen()` method of MyBatisCursorItemReader.

Chapter 6. Support to abnormal system

6.1. Input Validation

6.1.1. Overview

In this section, Explain the validation check of job input data (hereinafter referred to as input validation).

This function is the same usage for chunk model and tasklet model.

In general, input validation in batch processing is often carried out to confirm that data received from other systems etc. is valid in its own system.
Conversely, it can be said that it is unnecessary to perform input validation on reliable data in its own system (for example, data stored in the database).

Please refer to input Validation in TERASOLUNA Server 5.x Development Guideline because the input validation duplicates the contents of TERASOLUNA Server 5.x. Explain the main comparisons below.

Main comparison list

Comparison target	TERASOLUNA Server 5.x	TERASOLUNA Batch 5.x
Available input validation rules	Same as TERASOLUNA Server 5.x	
The target to which the rule is attached	`form class`	`DTO`
Validation execute method	`Give @Validated annotation to the Controller`	`Call the API of Validator class`
Setting error messages	Same as Definition of error messages in TERASOLUNA Server 5.x Development Guideline.	
Error message output destination	View	Log etc.

The input validation to be explained in this section mainly covers data obtained from `ItemReader`.
For checking job parameters, refer to Validation check of parameters.

6.1.1.1. Classification of input validation

The input validation is classified into single item check and correlation item check.

List of setting contents

Type	Description	Example	Implementation method
Single item check	Check to be completed with a single field	Required input check Digit check Type check	Bean Validation (using Hibernate Validator as implementation library)

Type	Description	Example	Implementation method
Correlation item check	Check to compare multiple fields	Comparison of numerical values Comparison of dates	`Validation` class that implements `org.springframework.validation.Validator` interface or Bean Validation

Spring supports Bean Validation which is a Java standard. For this single item check, this Bean Validation is used. For correlation item check, use Bean Validation of the `org.springframework.validation.Validator` interface provided by Spring.

In this respect, same as Classification of input validation in TERASOLUNA Server 5.x Development Guideline.

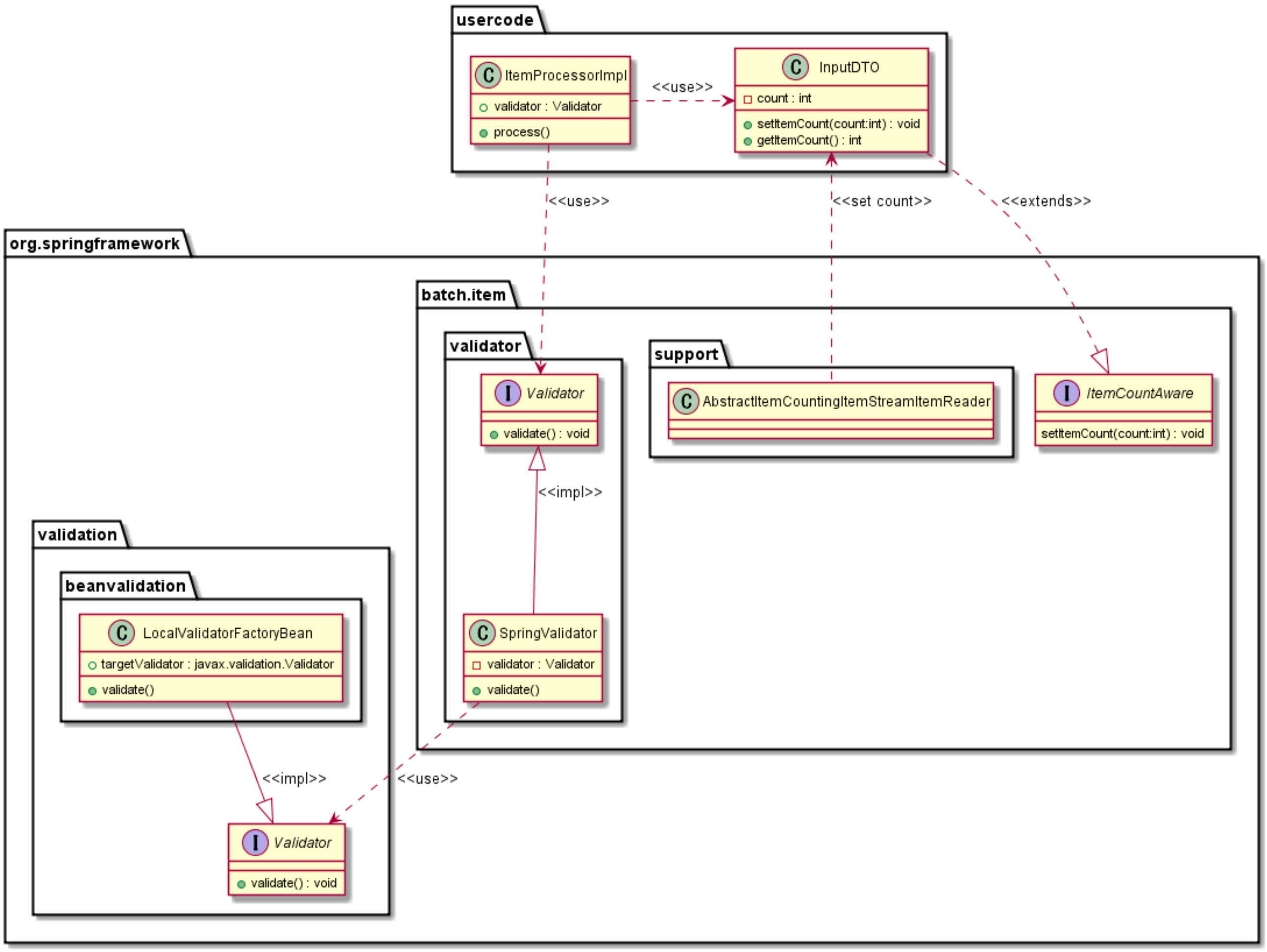
6.1.1.2. Overview of Input Validation

The timing of input validation in the chunk model and tasklet model is as follows.

- For chunk model, use `ItemProcessor`
- For tasklet model, use `Tasklet#execute()` at an arbitrary timing.

In the chunk model and tasklet model, the implementation method of input validation is the same, so here, explain the case where input validation is done in `ItemProcessor` of the chunk model.

First, explain an overview of input validation. The relationships of classes related to input validation are as follows.



Related class of input validation

- Inject `org.springframework.batch.item.validator.SpringValidator` which is the implementation of `org.springframework.batch.item.validator.Validator` in `ItemProcessor` and execute the validate method.
  - `SpringValidator` internally holds `org.springframework.validation.Validator` and execute the validate method.
    It can be said that it is a wrapper for `org.springframework.validation.Validator`.
    The implementation of `org.springframework.validation.Validator` is `org.springframework.validation.beanvalidation.LocalValidatorFactoryBean`. Use Hibernate Validator through this class.
- Implement `org.springframework.batch.item.ItemCountAware` in the input DTO to determine where the input validation error occured.

> *Setting the number of data*
>
> `ItemCountAware#setItemCount` is set by `AbstractItemCountingItemStreamItemReader`. Therefore, if you do not use `ItemReader` in the tasklet model, it will not be updated. In this case, it is necessary for the user to set what error occurred in the data.

Validators such as javax.validation.Validator or org.springframework.validation.Validator should not be used directly.

Validators such as `javax.validation.Validator` or `org.springframework.validation.Validator` should not be used directly, use `org.springframework.batch.item.validator.SpringValidator`.

`SpringValidator` is wrapper of `org.springframework.validation.Validator`. `SpringValidator` wraps the raised exception in `BindException` and throws it as `ValidationException`.
Therefore, `BindException` can be accessed via `ValidationException` which makes flexible handling easier.

On the other hand, if validators such as `javax.validation.Validator` and `org.springframework.validation.Validator` are used directly, it will be cumbersome logic to process the information that caused the validation error.

Do not use org.springframework.batch.item.validator.ValidatingItemProcessor

The input validation by `org.springframework.validation.Validator` can also be realized by using `ValidatingItemProcessor` provided by Spring Batch.

However, depending on the circumstances, it is necessary to extend it because of the following reasons, so do not use it from the viewpoint of unifying the implementation method.

- input validation error can not be handled and processing can not be continued.
- It is not possible to flexibly deal with data that has become an input validation error.
  - It is assumed that the processing for the data that becomes the input validation error becomes various kinds by the user (only log output, save error data to another file, etc.).

6.1.2. How to use

As mentioned earlier, the implementation method of input validation is the same as TERASOLUNA Server 5.x as follows.

- single item check uses Bean Validation.
- correlation item check uses Bean Validation or the `org.springframework.validation.Validator` interface provided by Spring.

Explain The method of input validation in the following order.

- Various settings
- Input validation rule definition
- Input validation execution

- Input validation error handling

6.1.2.1. Various settings

Use Hibernate Validator for input validation. Confirm that the definition of Hibernate Validator is in the library dependency and that the required bean definition exists. These have already been set in the blank project provided by TERASOLUNA Batch 5.x.

Setting example of dependent library

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
</dependency>
```

launch-context.xml

```
<bean id="validator" class="org.springframework.batch.item.validator.SpringValidator"
      p:validator-ref="beanValidator"/>

<bean id="beanValidator"
      class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"
/>
```

Error message setting

As mentioned earlier, for setting of error messages, refer to Definition of error messages in TERASOLUNA Server 5.x Development Guideline.

6.1.2.2. Input validation rule definition

The target of implementing the rule of input validation is the DTO obtained through `ItemReader`. Implement the DTO obtained through `ItemReader` as follows.

- Implement `org.springframework.batch.item.ItemCountAware` in the input DTO to determine where the input validation error occured.
    - In the `setItemCount` method, hold a numerical value in the class field indicating the number of items read in the currently processed item received as an argument.
- Define the input validation rule.
    - refer to Input Validation in TERASOLUNA Server 5.x Development Guideline.

Show an example of a DTO defining an input validation rule below.

An example of a DTO defining an input validation rule

```
public class VerificationSalesPlanDetail implements ItemCountAware {  // (1)

    private int count;

    @NotEmpty
    @Size(min = 1, max = 6)
    private String branchId;

    @NotNull
    @Min(1)
    @Max(9999)
    private int year;

    @NotNull
    @Min(1)
    @Max(12)
    private int month;

    @NotEmpty
    @Size(min = 1, max = 10)
    private String customerId;

    @NotNull
    @DecimalMin("0")
    @DecimalMax("9999999999")
    private BigDecimal amount;

    @Override
    public void setItemCount(int count) {
        this.count = count;  // (2)
    }

    // omitted getter/setter
}
```

List of setting contents

Sr. No.	Description
(1)	Implement the `ItemCountAware` class and override the `setItemCount` method. `ItemCountAware#setItemCount()` is passed to the argument as to what the data read by `ItemReader` is.
(2)	Holds the `count` received in the argument in the class field. This value is used to determine the number of items of data that caused an input validation error.

6.1.2.3. Input validation execution

Explain how to implement input validation. Implement input validation execution as follows.

- Execute `org.springframework.batch.item.validator.Validator#validate()` in the implementation of `ItemProcessor`.
  - Use an instance of `SpringValidator` by injecting it as `Validator` field.
- Handle input validation error. For details, refer to Input validation error handling.

Show an implementation example of input validation below.

An implementation example of input validation

```
@Component
public class ValidateAndContinueItemProcessor implements ItemProcessor
<VerificationSalesPlanDetail, SalesPlanDetail> {
    @Inject  // (1)
    Validator<VerificationSalesPlanDetail> validator;

    @Override
    public SalesPlanDetail process(VerificationSalesPlanDetail item) throws Exception
{
        try {  // (2)
            validator.validate(item);  // (3)
        } catch (ValidationException e) {
          // omitted exception handling
        }

        SalesPlanDetail salesPlanDetail = new SalesPlanDetail();
        // omitted business logic

        return salesPlanDetail;
    }
}
```

List of setting contents

Sr. No.	Description
(1)	Inject `SpringValidator` instance. For the type argument of `org.springframework.batch.item.validator.Validator`, set the DTO to be acquired via `ItemReader`.
(2)	Handle input validation error. In the example, exception is handled by catching with try/catch. For details, refer to Input validation error handling.
(3)	Execute `Validator#validate()` with the DTO obtained through `ItemReader` as an argument.

6.1.2.4. Input validation error handling

There are following 2 ways to handle input validation error.

1. Processing is aborted at the time when input validation error occurs, abnormally end the job.
2. Leave the occurrence of input validation error in the log etc. and continue processing the subsequent data. Thereafter, at the end of the job, the job is ended by specifying a warning.

6.1.2.4.1. Abnormal Termination of Processing

In order to abnormally terminate processing when an exception occurs, it throws `java.lang.RuntimeException` or its subclass.

There are two ways to perform processing such as log output when an exception occurs.

1. Catch exceptions with try/catch and do it before throwing an exception.
2. Do not catch exceptions with try/catch, implement `ItemProcessListener` and do it with the onProcessError method.
   - `ItemProcessListener#onProcessError()` can be implemented using the `@OnProcessError` annotation. For details, refer to Listener.

Following is an example of logging exception information and abnormally terminating processing when an exception occurs.

An error handling example with try/catch

```
@Component
public class ValidateAndAbortItemProcessor implements ItemProcessor
<VerificationSalesPlanDetail, SalesPlanDetail> {
    /**
     * Logger.
     */
    private static final Logger logger = LoggerFactory.getLogger
(ValidateAndAbortItemProcessor.class);

    @Inject
    Validator<VerificationSalesPlanDetail> validator;

    @Override
    public SalesPlanDetail process(VerificationSalesPlanDetail item) throws Exception
{
        try {  // (1)
            validator.validate(item);  // (2)
        } catch (ValidationException e) {
            // (3)
            logger.error("Exception occurred in input validation at the {} th item.
[message:{}]",
                    item.getCount(), e.getMessage());
            throw e;  // (4)
        }

        SalesPlanDetail salesPlanDetail = new SalesPlanDetail();
        // omitted business logic

        return salesPlanDetail;
    }
}
```

List of setting contents

Sr. No.	Description
(1)	Catch exceptions with try/catch.
(2)	Execute input validation.
(3)	Perform log output processing before throwing an exception.
(4)	Throw exceptions Since `org.springframework.batch.item.validator.ValidationException` is a subclass of `RuntimeException`, it can be thrown as it is.

An error handling example with ItemProcessListener#OnProcessError

```
@Component
public class ValidateAndAbortItemProcessor implements ItemProcessor
<VerificationSalesPlanDetail, SalesPlanDetail> {

    /**
     * Logger.
     */
    private static final Logger logger = LoggerFactory.getLogger
(ValidateAndAbortItemProcessor.class);

    @Inject
    Validator<VerificationSalesPlanDetail> validator;

    @Override
    public SalesPlanDetail process(VerificationSalesPlanDetail item) throws Exception
{
        validator.validate(item);  // (1)

        SalesPlanDetail salesPlanDetail = new SalesPlanDetail();
        // omitted business logic

        return salesPlanDetail;
    }

    @OnProcessError  // (2)
    void onProcessError(VerificationSalesPlanDetail item, Exception e) {
        // (3)
        logger.error("Exception occurred in input validation at the {} th item.
[message:{}]", item.getCount() ,e.getMessage());
    }
}
```

List of setting contents

Sr. No.	Description
(1)	Execute input validation.
(2)	Implement `ItemProcessListener#onProcessError()` using `@OnProcessError` annotation.
(3)	Perform log output processing before throwing an exception.

Note on using ItemProcessListener#onProcessError()

Using of the onProcessError method is useful for improving the readability of source code, maintainability, etc. since business process and exception handling can be separated.
However, when an exception other than `ValidationException` performing handling processing in the above example occurs, the same method is executed, so it is necessary to be careful.

When outputting log output in `ItemProcessor#process()` by exception, it is necessary to judge the kind of exception caused by the onProcessError method and handle exception. If this is cumbersome, it is good to share responsibility so that only input validation errors are handled by handling with try / catch and others are handed over to listeners.

6.1.2.4.2. Skipping Error Records

After logging the information of the record where input validation error occurred, skip the record where the error occurred and continue the processing of the subsequent data as follows.

- Catch exceptions with try/catch.
- Perform log output etc. when an exceptions occurs.
- Return `null` as the return value of `ItemProcessor#process()`.
  - By returning `null`, records in which an input validation error occurs are no longer included in subsequent processing targets (output with `ItemWriter`).

A skipping example with ItemProcessor

```
@Component
public class ValidateAndContinueItemProcessor implements ItemProcessor
<VerificationSalesPlanDetail, SalesPlanDetail> {
    /**
     * Logger.
     */
    private static final Logger logger = LoggerFactory.getLogger
(ValidateAndContinueItemProcessor.class);

    @Inject
    Validator<VerificationSalesPlanDetail> validator;

    @Override
    public SalesPlanDetail process(VerificationSalesPlanDetail item) throws Exception
{
        try {  // (1)
            validator.validate(item);  // (2)
        } catch (ValidationException e) {
            // (3)
            logger.warn("Skipping item because exception occurred in input validation
at the {} th item. [message:{}]",
                    item.getCount(), e.getMessage());
            // (4)
            return null;  // skipping item
        }

        SalesPlanDetail salesPlanDetail = new SalesPlanDetail();
        // omitted business logic

        return salesPlanDetail;
    }
}
```

List of setting contents

Sr. No.	Description
(1)	Catch exceptions with try/catch
(2)	Execute the input validation.
(3)	Perform log output processing before returning `null`.
(4)	Return `null` to skip this data and move on to the next data processing.

6.1.2.4.3. Setting the exit code

When an input validation error occurs, in order to distinguish between the case where input validation error did not occur and the state of the job, be sure to set an exit code that is not a normal termination.

If data with input validation error is skipped, setting of exit code is required even when abnormal termination occurs.

For details on how to set the exit code, refer to Job Management.

6.2. Exception handling

6.2.1. Overview

How to handle exception generated at the time of job execution is explained.

Since this function has different usage for chunk model and tasklet model, each will be explained.

First, classification of exceptions is explained, and handling method according to the type of exception is explained.

6.2.1.1. Classification of exception

The exception generated at the time of job execution are classified into 3 types as below.

Classification list of exceptions

Sr. No.	Classification	Description	Exception type
(1)	Exception that can be resolved the cause by job re-execution (parameters, change / modification of input data, etc.).	An exception that can resolve the cause by re-execution of a job that handles an exception with application code and performs exception handling.	Business exception Library exception occurring during normal operation
(2)	Exception that can not be resolved by job re-execution.	Exceptions that can be resolved by job re-execution are handled with the following pattern. 1. If exception can be caught in StepListener, handling exception with application code. 2. If exception cannot be caught in StepListener, handling exception in the framework.	System exception Unexpected system exception Fatal error

(3)	(During asynchronous execution)Exception caused by illegal request for job request	Exception caused by illegal request of job request is handled in the framework and performs exception handling. ⚠ In case of Asynchronous execution(DB polling) in the polling process, the validity of the job request is not verified.Therefore, it is desirable that the input check for the request is made in advance by the application that registers the job request. ⚠ In case of Asynchronous execution(Web container) It is assumed that the input check for the request is made in advance by the Web application. Therefore, exception handling is performed in an application that accepts requests or job requests.	Invalid job request error

> ⚠ *Avoid transaction processing within exception processing*
>
> When performing transactional processing such as writing to a database in exception processing, there is a possibility of causing a secondary exception. Exception processing should be based on output log analysis and end code setting.

6.2.1.2. Exception type

Types of exceptions are explained.

6.2.1.2.1. Business exception

A business exception is **an exception notifying that a violation of a business rule has been detected**.
This exception is generated within the logic of the step.
Since it is assumed to be an application state, it is not necessary to deal with the system operator.

Business exception example

- In case of stock out at inventory reserve
- When the number of days exceeds the scheduled date
- etc ...

> ℹ *Applicable exception class*
>
> - `java.lang.RuntimeException`and its subclass
>   - It is recommended to create business exception classes by the user.

6.2.1.2.2. Library exception occurring during normal operation

A library exception that occurs during normal operation refers to an exception **that may occur when the system is operating normally**, among exceptions generated in the framework and library.
Exceptions raised in the framework and library are exception classes that occur in the Spring Framework and other libraries.
Since it is assumed to be an application state, it is not necessary to deal with the system operator.

Example of library exception that occurs during normal operation

- Optimistic lock exception which occurs in exclusive control with online processing.
- Unique constraint exception that occurs when registering the same data at same time from multiple jobs or online processing.
- etc ...

> *Applicable exception class*
>
> - `org.springframework.dao.EmptyResultDataAccessException` (Exception that occurs when optimistic locking is done, when data update count is 0)
> - `org.springframework.dao.DuplicateKeyException` (Exception that occurs when a unique constraint violation occurs)
> - etc ...

6.2.1.2.3. System exception

A system exception is **an exception to notify that a state that should not occur is detected when the system is operating normally**.
This exception is generated within the logic of the step.
Action by the system operator is required.

Example of system exception

- Master data, directory, file, etc. that should exist in advance do not exist.
- When an exception classified as system abnormality is caught(IOException at file operation, etc.) among inspection exception occurring in the framework or library.
- etc...

> *Applicable exception class*
>
> - `java.lang.RuntimeException` or its subclass
>   - Creating a system exception class is recommended.

6.2.1.2.4. Unexpected system exception

Unexpected system exceptions are **non-inspection exceptions that do not occur when the system is operating normally**.
It is necessary for the system operator to deal with it or to analyze it by the system developer.

Unexpected system exceptions will not be handled except by doing the following processing. If handled, throw the exception again.

- Log capture exception for analysis and set the corresponding exit code.

Unexpected system exception example

- Bugs are hidden in applications, frameworks, and libraries.
- When the DB server is down.
- etc...

> *Applicable exception class*
>
> - `java.lang.NullPointerException` (Exception caused by a bug)
> - `org.springframework.dao.DataAccessResourceFailureException`(Exception raised when the DB server is down)
> - etc ...

6.2.1.2.5. Fatal error

A fatal error is an error **that notifies that a fatal problem has occurred that affects the entire system (application)**.
It is necessary for system operator or system developer to cope with it and recover.

Fatal errors are not handled except for the following processing.If handled, throw the exception again.

- Log capture exception for analysis and set the corresponding exit code.

Fatal error example

- When memory available for Java virtual machine is insufficient.
- etc...

> *Applicable exception class*
>
> - Classes that inherit`java.lang.Error`.
>   - java.lang.OutOfMemoryError (Error occurred when memory is insufficient)etc
> - etc ...

6.2.1.2.6. Invalid job request error

Job request invalid error is an error **to notify that a problem has occurred in the job request at asynchronous execution**.
It is necessary for the system operator to cope with and recover from it.

Job request incorrect error is based on exception handling in the application processing the request of job request, It is not explained in this guideline.

6.2.1.3. How to handle exceptions

How to handle exceptions is explained.

The exception handling pattern is as follows.

1. Decide whether to continue the job when an exception occurs (3 types)
2. Decide how to re-execute the suspended job (2 types)

How to decide whether to continue the job

Sr.No.	How to handle exceptions	Description
(1)	Skip	Skip error record and continue processing.
(2)	Retry	Reprocess the error record until the specified condition (number of times, time etc.) is reached.
(3)	Process interruption	Processing is interrupted.

> Even if an exception has not occurred, the job may stop while processing because the job has exceeded the expected processing time.
> In this case, please refer Stopping a job.

How to re-execute the suspended job

Sr.No.	How to handle exceptions	Description
(1)	Job rerun	Re-execute the suspended job from the beginning.
(2)	Job restart	Re-execute the interrupted job from the point where it was interrupted.

For details, please refer how to re-execute the suspended job Rerun processing.

6.2.1.3.1. Skip

Skipping is a method of skipping error data without stopping batch processing and continuing processing.

Skipping example

- Invalid record exists in input data
- When a business exception occurs
- etc ...

> *Reprocess skip record*
>
> When skipping, design how to deal with skipped invalid records. In the case of extracting and reprocessing an invalid record, a case like processing it is included at the time of the next execution,can be considered.

6.2.1.3.2. Retry

Retrying is a method of repeatedly attempting until a specified number of times or time is reached for a record that failed a specific process.
It is used only when the cause of processing failure depends on the execution environment and it is expected to be resolved over time.

Example of retrying

- When the record to be processed is locked by exclusive control
- When message transmission fails due to instantaneous interruption of network
- etc ...

> *Application of retry*
>
> If the retry is applied in every scene, the processing time unnecessarily increases at the time of occurrence of an abnormality, there is a risk that the detection of the abnormality will be delayed
> Therefore, it is desirable to apply the retry to only a part of the process, It is advisable to limit the target to those that are less reliable such as external system cooperation.

6.2.1.3.3. Process interruption

Process interruption is literally a method of interrupting processing midway.
It is used when processing cannot be continued on detecting an erroneous content or when there is requirement which does not allow skipping of records.

Examples of processing interruption

- Invalid record exists in input data
- When a business exception occurs
- etc ...

6.2.2. How to use

How to implement exception handling is explained.

A log is the main user interface for batch application operation.Therefore, monitoring of exception occurred will also be done through the log.

In Spring Batch, if an exception occurs during step execution, the log is output and abnormally terminated, so there is a possibility that the requirement can be satisfied without additional implementation by the user. The following explanation should be implemented pinpoint only when it is necessary for the user to output logs according to the system. Basically there are no case where all processing must be implemented.

For common log setting of exception handling, please refer Logging.

6.2.2.1. Step unit exception handling

Exception handling method in step units is explained.

Exception handling with ChunkListener interface

If you want to handle exceptions uniquely regardless of the processing model, use ChunkListener interface.
Although it can be implemented by using a step or job listener which is wider in scope than chunk Adopt `ChunkListener` emphasizing in handling it immediately as soon as possible after occurrence.

The exception handling method for each processing model is as follows.

Exception handling in chunk model

Implement the function using various Listener interfaces provided by Spring Batch.

Exception handling in tasklet model

Implement exception handling independently within tasklet implementation.

> *Why unified handling possible with ChunkListener.*
>
> A sense of incompatibility might be felt with `ChunkListener` being able to handle exceptions occurring within tasklet implementation. This is because in Spring Batch, execution of business logic is considered based on chunk, since one tasklet execution is handled as one chunk processing.
>
> This point also appears in `org.springframework.batch.core.step.tasklet.Tasklet` interface.
>
> ```
> public interface Tasklet {
>   RepeatStatus execute(StepContribution contribution,
>           ChunkContext chunkContext) throws Exception;
> }
> ```

6.2.2.1.1. Exception handling with ChunkListener interface

Implement `afterChunkError` method of ChunkListener interface.
Get error information from `ChunkContext` argument of `afterChunkError` method using `ChunkListener.ROLLBACK_EXCEPTION_KEY`as a key.

For details on how to set the listener,please refer Listerner setting.

Implementation example of ChunkListener

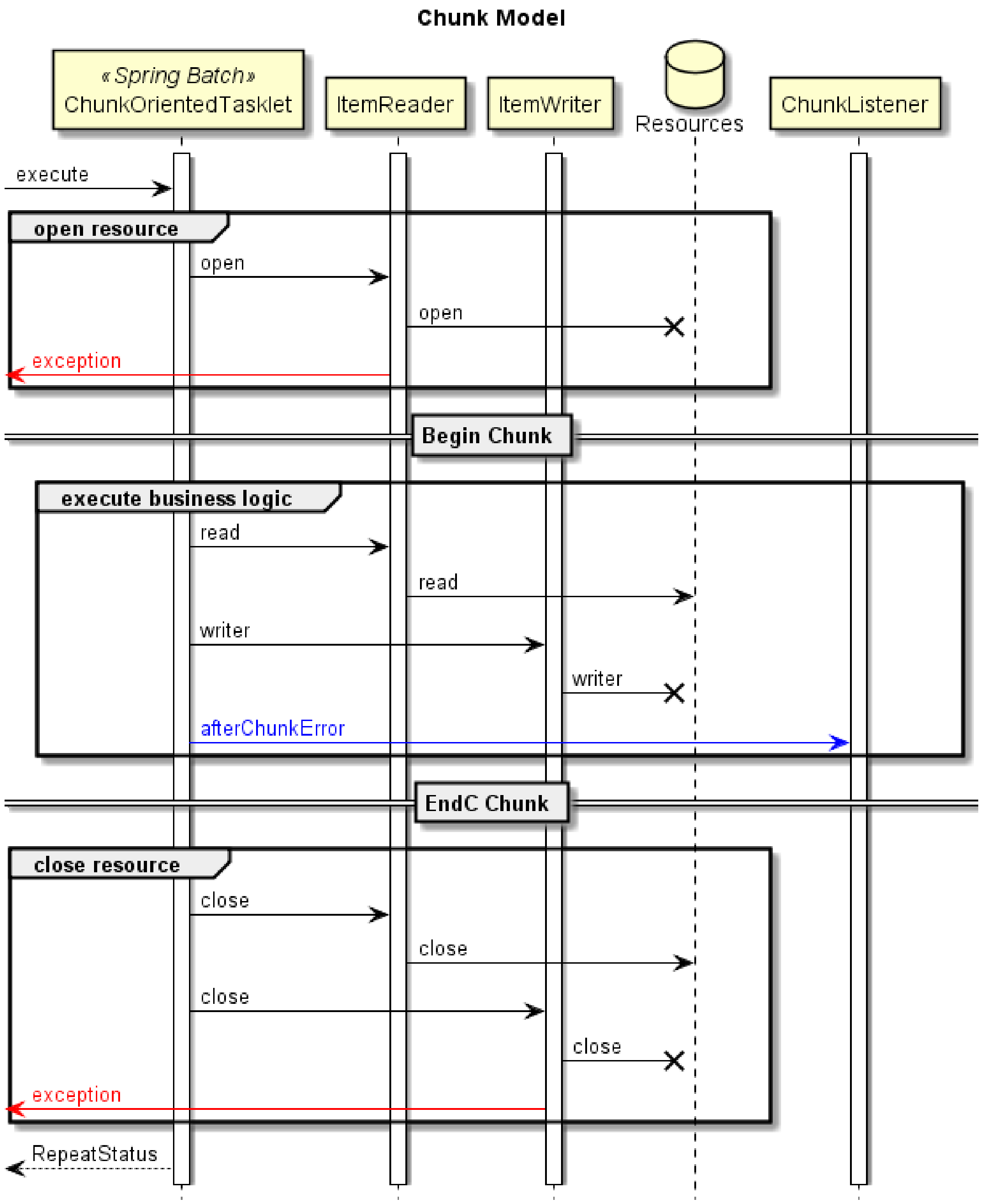
```
@Component
public class ChunkAroundListener implements ChunkListener {

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

    @Override
    public void beforeChunk(ChunkContext context) {
        logger.info("before chunk. [context:{}]", context);
    }

    @Override
    public void afterChunk(ChunkContext context) {
        logger.info("after chunk. [context:{}]", context);
    }

    // (1)
    @Override
    public void afterChunkError(ChunkContext context) {
        logger.error("Exception occurred while chunk. [context:{}]", context,
                context.getAttribute(ChunkListener.ROLLBACK_EXCEPTION_KEY)); // (2)
    }
}
```

Description

Sr.No.	Description
(1)	Implement `afterChunkError` method.
(2)	Get error information from `ChunkContext` using `ChunkListener.ROLLBACK_EXCEPTION_KEY` as a key. In this example, the stack trace of the acquired exception is logged.

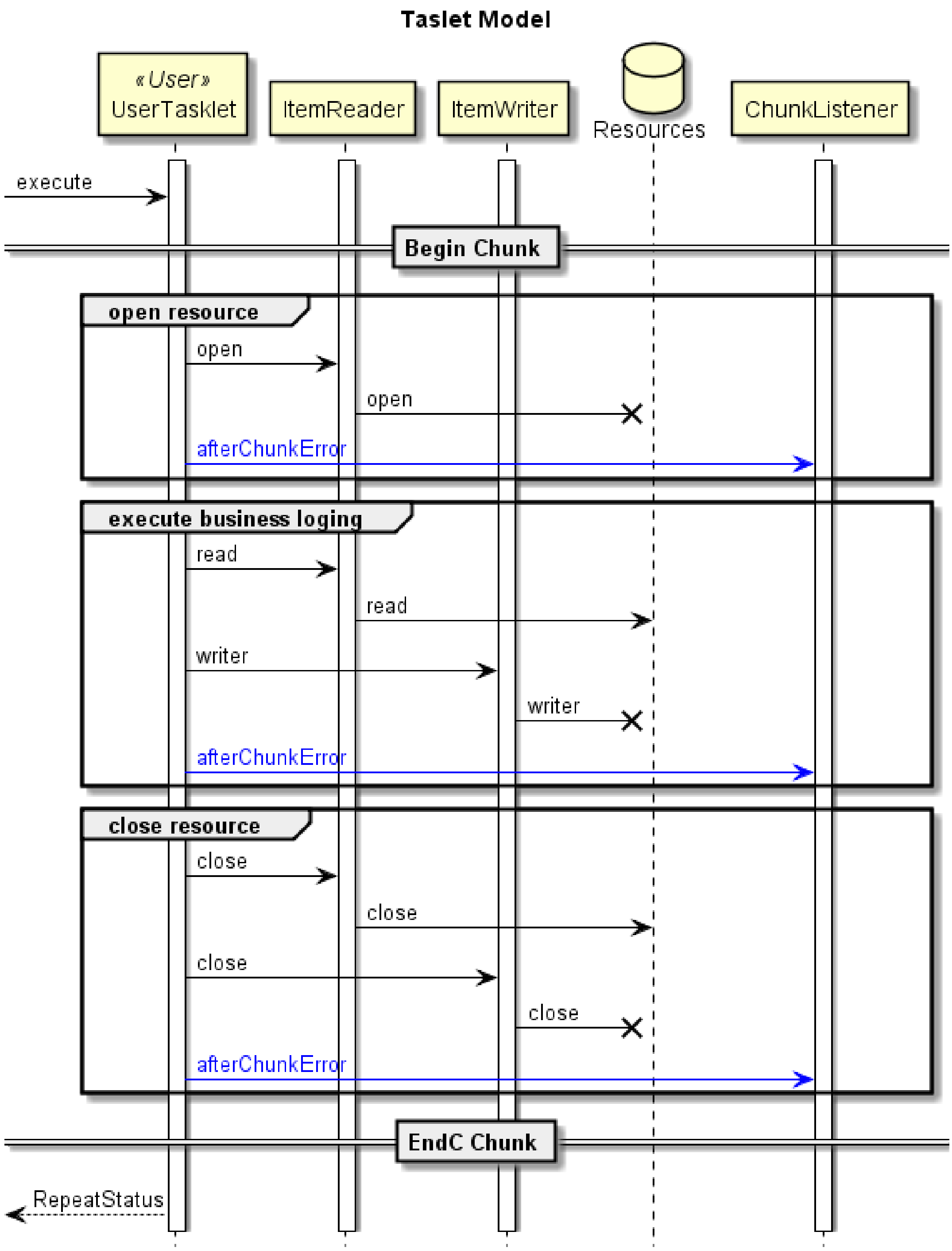
Difference in behavior of ChunkListener due to difference in processing model

In the chunk model, handling is not performed with the afterChunkError method because exceptions caused by opening / closing resources are outside the scope captured by the ChunkListener interface. A schematic diagram is shown below.



Schematic diagram of exception handling in chunk model

In the tasklet model, exceptions caused by opening and closing resources are handled by the afterChunkError method because they are within the scope captured by the ChunkListener interface. A schematic diagram is shown below.



Schematic diagram of exception handling in the tasklet model

If you wish to handle exceptions unifiedly by absorbing this behavior difference, it can be implemented by checking the occurrence of an exception in the StepExecutionListener interface. However, the implementation is slightly more complicated than `ChunkListener`.

Example of StepExecutionListener implementation.

```
@Component
public class StepErrorLoggingListener implements StepExecutionListener
{

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

    @Override
    public void beforeStep(StepExecution stepExecution) {
        // do nothing.
    }

    // (1)
    @Override
    public ExitStatus afterStep(StepExecution stepExecution) {
        // (2)
        List<Throwable> exceptions = stepExecution
.getFailureExceptions();
        // (3)
        if (exceptions.isEmpty()) {
            return ExitStatus.COMPLETED;
        }

        // (4)
        logger.info("This step has occurred some exceptions as follow.
" +
                "[step-name:{}] [size:{}]",
                stepExecution.getStepName(), exceptions.size());
        exceptions.forEach(th -> logger.error(
                "exception has occurred in job.", th));
        return ExitStatus.FAILED;
    }
```

Description

Sr.No.	Description
(1)	Implement `afterStep` method.
(2)	Get error information from the `stepExecution` argument.Be aware that you need to handle multiple exceptions together.
(3)	there is no error information, it is determined as normal termination.
(4)	If there is error information, exception handling is performed. In this example, log output with stack trace is done for all exceptions that has occurred.

6.2.2.1.2. Exception handling in chunk model

In the chunk model, exception handling with an inherited Listener StepListener.

For details on how to set listener, please refer Listner setting.

Coding point(ItemReader)

By implementing `onReadError` method of ItemReadListener interface, exceptions raised within ItemReader are handled.

Implementation example of ItemReadListener#onReadError

```
@Component
public class CommonItemReadListener implements ItemReadListener<Object> {

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

    // omitted.

    // (1)
    @Override
    public void onReadError(Exception ex) {
        logger.error("Exception occurred while reading.", ex);  // (2)
    }

    // omitted.
}
```

Description

Sr.No.	Description
(1)	Implement `onReadError` method.
(2)	Implement exception handling. In this example, the stack trace of the exception acquired from the argument is logged.

Coding point (ItemProcessor)

There are two ways to handle exception in ItemProcessor, and use it according to requirements.

1. How to try ~ catch in ItemProcessor
2. Using ItemProcessListener interface.

Why they are used properly is explained.
The argument of the `onProcessError` method executed when an exception occurs in ItemProcessor processing has two items, processing item and exception.
Depending on the requirements of the system, when handling exceptions such as log output in the `ItemProcessListener` interface, these two arguments may not satisfy the requirement. In that case, it

is recommended to catch the exception with try ~ catch in ItemProcessor and perform exception handling processing.
Note that implementing try ~ catch in ItemProcessor and implementing the `ItemProcessListener` interface may result in double processing, so care must be taken
If fine-grained exception handling is to be done, then adopt a method to try ~ catch in ItemProcessor.

Each method is explained.

How to try ~ catch in ItemProcessor

This is used to do fine-grained exception handling.
As explained in the skip section below,it will be used when doing error record of Skip.

Implementation example of try ~ catch in ItemProcessor

```
@Component
public class AmountCheckProcessor implements
        ItemProcessor<SalesPerformanceDetail, SalesPerformanceDetail> {

    // omitted.

    @Override
    public SalesPerformanceDetail process(SalesPerformanceDetail item)
            throws Exception {
        // (1)
        try {
            checkAmount(item.getAmount(), amountLimit);
        } catch (ArithmeticException ae) {
            // (2)
            logger.error(
                "Exception occurred while processing. [item:{}]", item, ae);
            // (3)
            throw new IllegalStateException("check error at processor.", ae);
        }
        return item;
    }
}
```

Description

Sr.No.	Description
(1)	Implement `try~catch`.Here special handling is only for certain exceptions (`ArithmeticException`).
(2)	Implement exception handling. In this example, the stack trace of the exception acquired from the argument is logged.
(3)	Throw a transaction rollback exception. This exception throw also allows common exception handling with `ItemProcessListener`.

How to use the `ItemProcessListener` interface

Use this,if business exceptions can be handled in the same way.

Implemenation example of ItemProcessListener#onProcessError

```
@Component
public class CommonItemProcessListener implements ItemProcessListener<Object, Object>
{

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

    // omitted.

    // (1)
    @Override
    public void onProcessError(Object item, Exception e) {
        // (2)
        logger.error("Exception occurred while processing. [item:{}]", item, e);
    }

    // omitted.
}
```

Description

Sr.No.	Description
(1)	Implement `onProcessError` method.
(2)	Implement exception handling. In this example, the processing target data acquired from the arguments and the stack trace of the exception are logged.

Coding point(ItemWriter)

By implementing the `onWriteError` method of ItemWriteListener interface exceptions raised within ItemWriter are handled.

Implementation example of ItemWriteListener#onWriteError

```
@Component
public class CommonItemWriteListener implements ItemWriteListener<Object> {

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

    // omitted.

    // (1)
    @Override
    public void onWriteError(Exception ex, List item) {
        // (2)
        logger.error("Exception occurred while processing. [items:{}]", item, ex);
    }

    // omitted.
}
```

Description

Sr.No.	Description
(1)	Implement `onWriteError` method.
(2)	Implement exception handling. In this example, the chunk of the output target obtained from the argument and the stack trace of the exception are logged.

6.2.2.1.3. Exception handling in tasklet model

Implement exception handling of tasklet model on its own in tasklet.

When performing transaction processing, be sure to throw the exception again in order to roll back.

Exception handling implementation example in tasklet model

```
@Component
public class SalesPerformanceTasklet implements Tasklet {

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

    // omitted

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception {

        // (1)
        try {
            reader.open(chunkContext.getStepContext().getStepExecution()
                    .getExecutionContext());

            List<SalesPerformanceDetail> items = new ArrayList<>(10);
            SalesPerformanceDetail item = null;
            do {
                // Pseudo operation of ItemReader
                // omitted

                // Pseudo operation of ItemProcessor
                checkAmount(item.getAmount(), amountLimit);


                // Pseudo operation of ItemWriter
                // omitted

            } while (item != null);
        } catch (Exception e) {
            logger.error("exception in tasklet.", e);   // (2)
            throw e;    // (3)
        } finally {
            try {
                reader.close();
            } catch (Exception e) {
                // do nothing.
            }
        }

        return RepeatStatus.FINISHED;
    }
}
```

Description

Sr.No.	Description
(1)	Implement `try-catch`
(2)	Implement exception handling. In this example, the stack trace of the exception that occurred is logged.
(3)	Throw the exception again to roll back the transaction.

6.2.2.2. Job-level exception handling

Exception handling method on a job level is explained.
It is a common handling method for chunk model and tasklet model.

Implement errors such as system exception and fatal error etc. in job level JobExecutionListener interface.

In order to collectively define exception handling processing, handling is performed on a job level without defining handling processing for each step.
In the exception handling here, do output log and setting ExitCode, do not implement transaction processing.

> *Prohibition of transaction processing*
>
> The processing performed by `JobExecutionListener` is out of the scope of business transaction management. Therefore, it is prohibited to execute transaction processing in exception handling on a job level.

Here, an example of handling an exception when occurs in ItemProcessor is shown. For details on how to set the listener,please refer Listner setting.

Implementation example of ItemProcessor

```
@Component
public class AmountCheckProcessor implements
        ItemProcessor<SalesPerformanceDetail, SalesPerformanceDetail> {

    // omitted.

    private StepExecution stepExecution;

    // (1)
    @BeforeStep
    public void beforeStep(StepExecution stepExecution) {
        this.stepExecution = stepExecution;
    }

    @Override
    public SalesPerformanceDetail process(SalesPerformanceDetail item)
            throws Exception {
        // (2)
        try {
            checkAmount(item.getAmount(), amountLimit);
        } catch (ArithmeticException ae) {
            // (3)
            stepExecution.getExecutionContext().put("ERROR_ITEM", item);
            // (4)
            throw new IllegalStateException("check error at processor.", ae);
        }
        return item;
    }
}
```

Exception handling implementation in JobExecutionListener

```
@Component
public class JobErrorLoggingListener implements JobExecutionListener {

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

    @Override
    public void beforeJob(JobExecution jobExecution) {
        // do nothing.
    }

    // (5)
    @Override
    public void afterJob(JobExecution jobExecution) {

        // whole job execution
        List<Throwable> exceptions = jobExecution.getAllFailureExceptions(); // (6)
        // (7)
        if (exceptions.isEmpty()) {
            return;
        }
        // (8)
        logger.info("This job has occurred some exceptions as follow. " +
                "[job-name:{}] [size:{}]",
                jobExecution.getJobInstance().getJobName(), exceptions.size());
        exceptions.forEach(th -> logger.error("exception has occurred in job.", th));

        // (9)
        jobExecution.getStepExecutions().forEach(stepExecution -> {
            Object errorItem = stepExecution.getExecutionContext()
                    .get("ERROR_ITEM"); // (10)
            if (errorItem != null) {
                logger.error("detected error on this item processing. " +
                        "[step:{}] [item:{}]", stepExecution.getStepName(),
                        errorItem);
            }
        });

    }
}
```

Description

Sr.No.	Description
(1)	In order to output error data with `JobExecutionListener`, get the `StepExecution` instance before step execution.
(2)	Implement `try-catch`.

Sr.No.	Description
(3)	Implement exception handling. In this example, error data is stored in the context of the StepExecution instance with the key ERROR_ITEM.
(4)	Throw an exception to do exception handling with JobExecutionListener.
(5)	Implement exception handling in afterJob method.
(6)	From the argument jobExecution it gets error information which occurred in the entire job.
(7)	If there is no error information, it is determined as normal termination.
(8)	If there is error information, exception handling is performed. In this example, log output with stack trace is done for all exceptions that occurred.
(9)	In this example, when there is error data, log output is performed. Get the StepExecution instance from all the steps defined in the job and check whether the error data is stored with the key ERROR_ITEM. If it is stored, it is logged as error data.

Object to be stored in ExecutionContext

The object to be stored in ExecutionContext must be a class that implements java.io.Serializable. This is because ExecutionContext is stored in JobRepository.

6.2.2.3. Determination as to whether processing can be continued

How to decide whether or not to continue processing jobs when an exception occurs is explained.

Process continuation propriety method list

- Skip
- Retry
- Process interruption

6.2.2.3.1. Skip

A method of skipping an erroneous record and continuing processing is described.

Chunk model

In the chunk model, the implementation method differs for components of each processing

before applying the contents described here, make sure to read About reason why <skippable-exception-classes> is not used.

- Skip with ItemReader
- Skip with ItemProcessor
- Skip with ItemWriter

Skip with ItemReader

Specify the skip method in `skip-policy` attribute of `<batch:chunk>`. In `<batch:skippable-exception-classes>`, specify the exception class that occurs in the ItemReader to be skipped
For the `skip-policy` attribute, use one of the following classes provided by Spring Batch.

skip-policy list

Class name	Description
AlwaysSkipItemSkipPolicy	Always skip.
NeverSkipItemSkipPolicy	Do not skip.
LimitCheckingItemSkipPolicy	Skip until the upper limit of the specified number of skips is reached. When the upper limit value is reached, the following exception occurs. `org.springframework.batch.core.step.skip.SkipLimitExceededException` This is the skipping method used by default when `skip-policy` is omitted.
ExceptionClassifierSkipPolicy	Use this when you want to change `skip-policy` which applies to each exception.

Implementation example of skipping is explained.

Handle case where an incorrect record exists when reading a CSV file with `FlatFileItemReader`.
The following exceptions occur at this time.

- `org.springframework.batch.item.ItemReaderException`(Base exception class)
  - `org.springframework.batch.item.file.FlatFileParseException` (Exception occured class)

`skip-policy` shows how to define it separately.

Definition of ItemReader as a prerequisite

```
<bean id="detailCSVReader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step">
    <property name="resource" value="file:#{jobParameters[inputFile]}"/>
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                      p:names="branchId,year,month,customerId,amount"/>
            </property>
            <property name="fieldSetMapper">
                <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

p:targetType="org.terasoluna.batch.functionaltest.app.model.performance.SalesPerforman
ceDetail"/>
            </property>
        </bean>
    </property>
</bean>
```

AlwaysSkipItemSkipPolicy

Specification example of AlwaysSkipItemSkipPolicy

```
<!-- (1) -->
<bean id="skipPolicy"
      class="org.springframework.batch.core.step.skip.AlwaysSkipItemSkipPolicy"/>

<batch:job id="jobSalesPerfAtSkipAllReadError" job-repository="jobRepository">
    <batch:step id="jobSalesPerfAtSkipAllReadError.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                         processor="amountCheckProcessor"
                         writer="detailWriter" commit-interval="10"
                         skip-policy="skipPolicy"> <!-- (2) -->
            </batch:chunk>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Description

Sr.No.	Description
(1)	Define `AlwaysSkipItemSkipPolicy` as a bean.
(2)	Set the bean defined in (1) to the `skip-policy` attribute of `<batch:chunk>`

NeverSkipItemSkipPolicy

Specification example of NeverSkipItemSkipPolicy

```
<!-- (1) -->
<bean id="skipPolicy"
      class="org.springframework.batch.core.step.skip.NeverSkipItemSkipPolicy"/>

<batch:job id="jobSalesPerfAtSkipNeverReadError" job-repository="jobRepository">
    <batch:step id="jobSalesPerfAtSkipNeverReadError.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                         processor="amountCheckProcessor"
                         writer="detailWriter" commit-interval="10"
                         skip-policy="skipPolicy"> <!-- (2) -->
            </batch:chunk>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Description

Sr.No.	Description
(1)	Define `NeverSkipItemSkipPolicy`as a bean.
(2)	Set the bean defined in (1) to the `skip-policy` attribute of `<batch:chunk>`.

LimitCheckingItemSkipPolicy

Specification example of LimitCheckingItemSkipPolicy

```
(1)
<!--
<bean id="skipPolicy"
      class="org.springframework.batch.core.step.skip.LimitCheckingItemSkipPolicy"/>
-->

<batch:job id="jobSalesPerfAtValidSkipReadError" job-repository="jobRepository">
    <batch:step id="jobSalesPerfAtValidSkipReadError.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                         processor="amountCheckProcessor"
                         writer="detailWriter" commit-interval="10"
                         skip-limit="2">  <!-- (2) -->
                <!-- (3) -->
                <batch:skippable-exception-classes>
                    <!-- (4) -->
                    <batch:include
                        class="org.springframework.batch.item.ItemReaderException"/>
                </batch:skippable-exception-classes>
            </batch:chunk>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Description

Sr.No.	Description
(1)	Define `LimitCheckingItemSkipPolicy` as a bean. The `skip-policy` attribute is omitted by default, so it does not have to be defined.
(2)	Set the upper limit value of skip number in the `skip-limit` attribute of `<batch:chunk>` The `skip-policy` attribute is omitted by default.
(3)	Define `<batch:skippable-exception-classes>` and set the targetted exception in the tag.
(4)	Set `ItemReaderException` as a skip target class.

ExceptionClassifierSkipPolicy

ExceptionClassifierSkipPolicy specification example

```
<!-- (1) -->
<bean id="skipPolicy"
      class="org.springframework.batch.core.step.skip.ExceptionClassifierSkipPolicy">
    <property name="policyMap">
        <map>
            <!-- (2) -->
            <entry key="org.springframework.batch.item.ItemReaderException"
                   value-ref="alwaysSkip"/>
        </map>
    </property>
</bean>
<!-- (3) -->
<bean id="alwaysSkip"
      class="org.springframework.batch.core.step.skip.AlwaysSkipItemSkipPolicy"/>

<batch:job id="jobSalesPerfAtValidNolimitSkipReadError"
           job-repository="jobRepository">
    <batch:step id="jobSalesPerfAtValidNolimitSkipReadError.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <!-- skip-limit value is dummy. -->
            <batch:chunk reader="detailCSVReader"
                         processor="amountCheckProcessor"
                         writer="detailWriter" commit-interval="10"
                         skip-policy="skipPolicy"> <!-- (4) -->
            </batch:chunk>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Description

Sr.No.	Description
(1)	Define `ExceptionClassifierSkipPolicy` as a bean.
(2)	Set the `policyMap` property to a map whose key is an exception class and whose value is skipped. In this example, when `ItemReaderException` occurs, it is set to be the skip method defined in (3).
(3)	Define the skipping method you want to execute by exception. In this example, `AlwaysSkipItemSkipPolicy` is defined.
(4)	Set the bean defined in (1) to the `skip-policy` attribute of `<batch:chunk>`.

Skip on ItemProcessor

Try ~ catch in ItemProcessor and return null.
Skip with `skip-policy` is not used because reprocessing occurs in ItemProcessor. For details, please refer About reason why <skippable-exception-classes> is not used.

Restrictions on exception handling in ItemProcessor

As in About reason why <skippable-exception-classes> is not used,、 In ItemProcessor, skipping using `<batch:skippable-exception-classes>` is forbidden. Therefore,cannot skip applying "How to use the ItemProcessListener interface." explained in Coding point (ItemProcessor).

Implementation example of skip.

Implementation example of try~catch in ItemProcessor of Coding point (ItemProcessor) correspond to skip.

try~catch example in ItemProcessor

```
@Component
public class AmountCheckProcessor implements
        ItemProcessor<SalesPerformanceDetail, SalesPerformanceDetail> {

    // omitted.

    @Override
    public SalesPerformanceDetail process(SalesPerformanceDetail item) throws
Exception {
        // (1)
        try {
            checkAmount(item.getAmount(), amountLimit);
        } catch (ArithmeticException ae) {
            logger.warn("Exception occurred while processing. Skipped. [item:{}]",
                    item, ae); // (2)
            return null; // (3)
        }
        return item;
    }
}
```

Description

Sr.No.	Description
(1)	Implement `try～catch`
(2)	Implement exception handling In this example, the stack trace of the exception acquired from the argument is logged.
(3)	Skip error data by returning null.

Skip with ItemWriter

In ItemWriter skip processing is not done generally.
Even when skipping is necessary, skipping by `skip-policy` will not be used as the chunk size will change. For details, please refer About reason why <skippable-exception-classes> is not used.

Tasket model

Handle exceptions in business logic and implement processing to skip error records independently.

Implementation example of Exception handling in tasklet model corresponds to skip.

Implementation example with tasklet model

```java
@Component
public class SalesPerformanceTasklet implements Tasklet {

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

    // omitted

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception {

        // (1)
        try {
            reader.open(chunkContext.getStepContext().getStepExecution()
                    .getExecutionContext());

            List<SalesPerformanceDetail> items = new ArrayList<>(10);
            SalesPerformanceDetail item = null;
            do {
                // Pseudo operation of ItemReader
                // omitted

                // Pseudo operation of ItemProcessor
                checkAmount(item.getAmount(), amountLimit);


                // Pseudo operation of ItemWriter
                // omitted

            } while (item != null);
        } catch (Exception e) {
            logger.warn("exception in tasklet. Skipped.", e);   // (2)
            continue;    // (3)
        } finally {
            try {
                reader.close();
            } catch (Exception e) {
                // do nothing.
            }
        }

        return RepeatStatus.FINISHED;
    }
}
```

Description

Sr.No.	Description
(1)	Implement `try-catch`
(2)	Implement exception handling. In this example, the stack trace of the exception that occurred is logged.
(3)	Processing of error data is skipped by continue.

6.2.2.3.2. Retry

When an exception is detected, a method of reprocessing until the specified number of times is reached is described.

For retry, it is necessary to consider various factors such as the presence or absence of state management and the situation where retry occurs, there is no reliable method, and retrying it unnecessarily deteriorates the situation.

Therefore, this guideline explains how to use `org.springframework.retry.support.RetryTemplate` which implements a local retry.

> As with skip, there is also a way to specify the target exception class with `<retryable-exception-classes>`. However, as with About reason why <skippable-exception-classes> is not used There is a side effect that causes performance degradation, so TERASOLUNA Batch 5.x does not use it.

RetryTemplate Implementation code

```java
public class RetryableAmountCheckProcessor implements
        ItemProcessor<SalesPerformanceDetail, SalesPerformanceDetail> {

    // omitted

    // (1)
    private RetryPolicy retryPolicy;

    @Override
    public SalesPerformanceDetail process(SalesPerformanceDetail item)
            throws Exception {

        // (2)
        RetryTemplate rt = new RetryTemplate();
        if (retryPolicy != null) {
            rt.setRetryPolicy(retryPolicy);
        }

        try {
            // (3)
            rt.execute(context -> {
                item.setAmount(item.getAmount().divide(new BigDecimal(10)));
                checkAmount(item.getAmount(), amountLimit);
                return null;
            });
        } catch (ArithmeticException ae) {
            // (4)
            throw new IllegalStateException("check error at processor.", ae);
        }
        return item;
    }

    public void setRetryPolicy(RetryPolicy retryPolicy) {
        this.retryPolicy = retryPolicy;
    }
}
```

Bean definition

```xml
<!-- omitted -->

<bean id="amountCheckProcessor"

class="org.terasoluna.batch.functionaltest.ch06.exceptionhandling.RetryableAmountCheck
Processor"
      scope="step"
      p:retryPolicy-ref="retryPolicy"/> <!-- (5) -->

<!-- (6) (7) (8)-->
<bean id="retryPolicy" class="org.springframework.retry.policy.SimpleRetryPolicy"
      c:maxAttempts="3"
      c:retryableExceptions-ref="exceptionMap"/>

<!-- (9) -->
<util:map id="exceptionMap">
    <entry key="java.lang.ArithmeticException" value="true"/>
</util:map>

<batch:job id="jobSalesPerfWithRetryPolicy" job-repository="jobRepository">
    <batch:step id="jobSalesPerfWithRetryPolicy.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                         processor="amountCheckProcessor"
                         writer="detailWriter" commit-interval="10"/>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Description

Sr.No.	Description
(1)	Store the retry condition.
(2)	Create an instance of RetryTemplate. The default retry count = 3, all exceptions are subject to retry.
(3)	Use the `RetryTemplate#execute` method to execute the business logic you wish to retry Execute only the part that you want to retry with the `RetryTemplate#execute` method, not the entire business logic.
(4)	Exception handling when the number of retries exceeds the specified number of times The exception that occurs in the business logic is thrown as it is.
(5)	Specify the retry condition defined in (6).
(6)	Define the retry condition in the class that implements `org.springframework.retry.RetryPolicy`. In this example, we use `SimpleRetryPolicy` which is provided by Spring Batch.

Sr.No.	Description
(7)	Specify the number of retries in the `maxAttempts` constructor argument.
(8)	Specify the map that defines the target exception to be retried defined in (9) in `retryableExceptions` of the constructor argument.
(9)	Define a map in which the target exception class to be retried and the boolean value are set in the key If the boolean value is `true`, target exception is retried.

6.2.2.3.3. Process interruption

If you want to abort step execution, throw RuntimeException or its subclass other than skip / retry object.

Implementation example of skip is shown based on LimitCheckingItemSkipPolicy

Bean definition

```
<batch:job id="jobSalesPerfAtValidSkipReadError" job-repository="jobRepository">
    <batch:step id="jobSalesPerfAtValidSkipReadError.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                         processor="amountCheckProcessor"
                         writer="detailWriter" commit-interval="10"
                         skip-limit="2">
                <batch:skippable-exception-classes>
                    <!-- (1) -->
                    <batch:include
class="org.springframework.batch.item.validator.ValidationException"/>
                </batch:skippable-exception-classes>
            </batch:chunk>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Description

Sr.No.	Description
(1)	If an exception other than `ValidationException` occurs, processing is interrupted.

An implementation example of retry is shown based on Retry.

Bean definition

```
<!-- omitted -->

<bean id="retryPolicy" class="org.springframework.retry.policy.SimpleRetryPolicy"
      c:maxAttempts="3"
      c:retryableExceptions-ref="exceptionMap"/>

<util:map id="exceptionMap">
    <!-- (1) -->
    <entry key="java.lang.UnsupportedOperationException" value="true"/>
</util:map>

<batch:job id="jobSalesPerfWithRetryPolicy" job-repository="jobRepository">
    <batch:step id="jobSalesPerfWithRetryPolicy.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                         processor="amountCheckProcessor"
                         writer="detailWriter" commit-interval="10"/>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Description

Sr.No.	Description
(1)	If an exception other than `UnsupportedOperationException` occurs, processing is interrupted.

6.2.3. Appendix

6.2.3.1. About reason why <skippable-exception-classes> is not used

Spring Batch provides a function to specify an exception to be skipped for the entire job, skip processing the item where the exception occurred, and continue the processing.

It implements the function by setting the `<skippable-exception-classes>` tag under the `<chunk>` tag and specifying the exception to be skipped as follows.

Usage example of <skippable-exception-classes>

```
<job id="flowJob">
    <step id="retryStep">
        <tasklet>
            <chunk reader="itemReader" writer="itemWriter"
                   processor="itemProcessor" commit-interval="20"
                   skip-limit="10">
                <skippable-exception-classes>
                    <!-- specify exceptions to the skipped -->
                    <include class="java.lang.Exception"/>
                    <exclude class="java.lang.NullPointerException"/>
                </skippable-exception-classes>
            </chunk>
        </tasklet>
    </step>
</job>
```

By using this function, it is possible to skip the record where the input check error has occurred and continue processing the subsequent data, For TERASOLUNA Batch 5.x it is not used for the following reasons.

- If an exception is skipped using the `<skippable-exception-classes>` tag, since the number of data items included in one chunk varies, performance deterioration may occur.
  - This depends on where the exception occurred (`ItemReader` / `ItemProcessor` / `ItemWriter`). Details are described later.

> *Avoid using SkipPolicy without defining <skippable-exception-classes>*
>
> All exceptions are implicitly registered, and the possibility of performance degradation increases dramatically.

The behavior of each exception occurrence (`ItemReader` / `ItemProcessor` / `ItemWriter`) is explained respectively.
The transaction operation is not processed regardless of where the exception occurred, but if an exception occurs, it is always rolled back and then processed again.

When an exception occurs in ItemReader

- When an exception occurs in the process of `ItemReader`, the processing object moves to the next item.
- There is no side effect by this.

When an exception occurs in ItemProcessor

- If an exception occurs within the processing of `ItemProcessor`, return to the beginning of the chunk and reprocess from the first.
- Items to be skipped for reprocessing are not included.
- The chunk size at the first processing and reprocessing does not change.

When an exception occurs in ItemWriter

- If an exception occurs within the processing of `ItemWriter`, return to the beginning of the chunk and reprocess from the first.
- Reprocessing is fixed to `ChunkSize=1` and executed one by one.
- Items to be skipped for reprocessing are also included.

If an exception occurs in `ItemProcessor`, considering the case of `ChunkSize=1000` as an example, when an exception occurs on the 1000th case, reprocessing is done from the 1st and total of 1999 processes are executed.

If an exception occurs in `ItemWriter`, `ChunkSize=1` is fixed and reprocessed. Considering the case of `ChunkSize = 1000` as an example, it is divided into 1000 transactions regardless of originally 1 transaction and processed.

These means that the processing time of the entire job is prolonged, and there is a high possibility of deteriorating the situation at the time of abnormality. In addition, there is a possibility that the double treatment itself becomes a problem, and it gives rise to additional consideration to the design manufacturing.

Therefore, we do not recommend using `<skippable-exception-classes>`. Skipping data that failed in `ItemReader` does not cause these problems, in order to prevent accidents, basically avoid it and apply it only when it is absolutely necessary.

6.3. Restart processing

6.3.1. Overview

After the job gets terminated abnormally due to occurence of some failure,means to recover to restart the job is explained.

Since this function has different usage for chunk model and tasklet model, each will be explained.

There are the following methods to restart a job.

1. Job rerun
2. Job restart
    - Stateless restart
        - Number based restart
    - Stateful restart
        - Determine processing status, restart process to extract unprocessed data
            - It is necessary to separately implement a process for identifying the processing state

Below is terminology definition:

Rerun
: Redoing the job from the beginning.
As a preliminary work, it is necessary to recover to the state before failure occurred such as initializing data,at the time of starting the job.

Restart
: Resume the processing from where the job was interrupted.
It is necessary to design/implement retention of restart position processing, acquisition method, data skip method till restart position etc in advance.
There are two types of restart, stateless and stateful.

Stateless restart
: A restart method not considering the state (unprocessed / processed) for each input data.

Number based restart
: One of stateless restart.
A method of retaining the processed input data count and skipping that input data at the time of restart.
If the output is a non-transactional resource, it is also necessary to hold the output position and move the write position to that position at the time of restart.

Stateful restart
: A restart method in which the state (unprocessed / processed) for each input data is judged, and only unprocessed data is acquired as an acquisition condition.

If the output is a non-transactional resource, make the resource additional, and at the time of restart, add it to the previous result.

Generally rerun is the easiest way to re-execute. With Rerun < Stateless restart < Stateful restart order, it becomes difficult to design and implement. Of course, it is always preferable to use rerun if possible, For each job that the user implements, please consider which method to apply depending on the allowable batch window and processing characteristics.

6.3.2. How to use

Implementation method of Rerun and restart is explained.

6.3.2.1. Job rerun

How to implement job rerun is explained.

1. Preliminary work of data recovery such as initialization of data before re-run is carried out.
2. Execute the failed job again with the same condition (same parameter).
    - In Spring Batch, if you execute a job with the same parameters, it will be treated as double execution, but TERASOLUNA Batch 5.x treats it as a separate job
      For details,please referAbout parameter conversion class.

6.3.2.2. Job restart

How to restart a job is explained.

When restarting a job, it is basically done on a job executed synchronously.

It is recommended that asynchronously executed jobs should be designed with a corresponding job design with a rerun instead of a restart. This is difficult to judge whether it is **"intended restart execution"** or **"unintended duplicate execution"**, this is because there is a possibility of confusion in operation.

If restart requirements can not be excluded for asynchronous execution job, The following methods can be used to clarify **"intended restart execution"**.

- Restart by `-restart` of `CommandLineJobRunner`
    - Restart asynchronously executed job separately from synchronous execution. It becomes effective when progressing the recovery process sequentially.
- Restart by `JobOperator#restart(JobExecutionId)`
    - Restart the asynchronously executed job on the mechanism of asynchronous execution again.It is effective when progressing with recovery processing collectively.
        - Asynchronous execution(DB polling) does not support restart. Therefore, it is necessary to implement it separately by the user.
        - Asynchronous execution(Web container) guides how to implement restart.User implements it according to this description.

About restart when there is input check

The input check error is not recoverable unless the input resource causing the check error is corrected. For reference, an input resource correction example when an input check error occurs is shown below.

1. When an input check error occurs, log output is performed so that the target data can be specified.
2. Based on the output log information, correct the input data.
    - Make sure that the order of input data does not change.
    - Correction method differs according to generation method of input resource.
        - Correct manually
        - Recreate with job etc
        - Retransmission from collaboration source
3. Deploy corrected input data and execute restart.

In the case of multiple processing (Partition Step)

When restarting in multiple processing(Partition Step), processing is carried out again from **split processing**. When all of the data are processed as the result of dividing the data、 unnecessary splitting is performed and recorded on `JobRepository`, there is no problem such as data inconsistency caused by this.

6.3.2.3. Stateless restart

How to implement stateless restart is explained.

Stateless restart with TERASOLUNA Batch 5.x refers to a number based restart.This is implemented by using the mechanism of Spring Batch as it is.
The number based restart can be used in job execution of chunk model. In addition, the number based restart uses context information about inputs and outputs registered in `JobRepository`.
Therefore, in a number based restart, it is assumed that `JobRepository` does not use the in-memory database, but uses persistence guaranteed.

About failure occurence of JobRepository

Updating to `JobRepository` is done in transactions that are independent of transactions of the database used by the business process.
In other words, only the failure to the business process is subject to recovery.
This means that if a failure occurs in `JobRepository`, there is a possibility that it will deviate from the actual count of processes, it means that there is a danger of double processing at restart.
Therefore, it is necessary to consider how to deal with failure. For example, design availability of `JobRepository` higher, then review rerun's method in advance, and so on.

Input at restart

Since most of the ItemReaders provided by Spring Batch are compatible with the number-based restart, special support is not necessary.
If you want to create a number based restartable ItemReader yourself, the following abstract classes can be extended that have restart processing implemented.

- `org.springframework.batch.item.support.AbstractItemCountingItemStreamItemReader`

Since the number based restart determines the restart starting point based only on the number of items to the last, it can not detect change / addition / deletion of input data. Often the input data is corrected after the job ends abnormally, When data change like the following, be careful as there will be a difference in the output between, the result after the job ends normally and the result after the job ends abnormally is restarted and recovered.

- Change the data acquisition order
  - At the time of restart, duplicate or unprocessed data will get generated, so never go as it results in a different recovery result from the result of rerun.
- Update processed data
  - Since the data updated at the time of restarting is skipped, it is not preferred as there are cases where rerun result and the recovered result by restart result changes.
- Update or add unprocessed data
  - As rerun result and recovered result becomes same, its allowed. However, it is different from the result of the normal termination in the first execution. This should be used when patching abnormal data in an emergency coping manner or when processing as much as possible data received at the time of execution.

Output at Restart

Care must be taken in output to non-transactional resources. For example in a file, it is necessary to grasp the position to which the output was made and output from that position.
The `FlatFileItemWriter` provided by Spring Batch gets the previous output position from the context and outputs from that position at the time of restart, so no special countermeasure is necessary.
For transactional resources, since rollback is performed at the time of failure, it is possible to perform processing without taking any special action at restart.

If the above conditions are satisfied, add the option `-restart` to the failed job and execute it again. Below an example of job restart is shown.

Restart example of synchronous job

```
# (1)
java -cp dependency/*
org.springframework.batch.core.launch.support.CommandLineJobRunner <jobPath> <jobName>
-restart
```

Description

Sr.No.	Description
(1)	At `CommandLineJobRunner`specify the failed job's job bean path and job name, add `-restart`and execute it Since job parameters are restored from `JobRepository`, it is not necessary to specify them.

An example of restarting a job executed in asynchronous execution (DB polling) is shown below.

Restart example of job executed in asynchronous execution (DB polling)

```
# (1)
java -cp dependency/*
org.springframework.batch.core.launch.support.CommandLineJobRunner <JobExecutionId>
-restart
```

Description

Sr.No.	Description
(1)	Run `CommandLineJobRunner`by specifying the same job execution ID (JobExecutionId) as the failed job and adding `-restart`. Since job parameters are restored from `JobRepository`, it is not necessary to specify them. The job execution ID can be acquired from the job-request-table. About the job-request-table, please refer About polling table.

Output log of job execution ID

In order to promptly specify the job execution ID of the abnormally terminated job, It is recommended to implement a listener or exception handling class that logs the job execution ID when the job ends or when an exception occurs.

An example of restart in asynchronous execution (Web container) is shown below.

Examples of restarting jobs executed in asynchronous execution (web container)

```
public long restart(long JobExecutionId) throws Execption {
  return jobOperator.restart(JobExecutionId); // (1)
}
```

Description

Sr.No.	Description
(1)	Specify the same job execution ID (JobExecutionId) as the failed job to `JobOperator` and execute it with `restart` method. Job parameters are restored from `JobRepository`. The job execution ID can be obtained from the ID acquired when executing the job with the web application or from `JobRepository`. For acquisition method, please refer Job state management.

6.3.2.4. Stateful restart

How to achieve stateful restart is explained.

Stateful restart is a method of reprocessing by acquiring only unprocessed data together with input/output results at the time of execution. Although this method is difficult to design such as state retaining / determination unprocessed etc, it is sometimes used because it has a strong characteristic in data change.

In stateful restart, since restart conditions are determined from input / output resources, persistence of `JobRepository` becomes unnecessary.

Input at restart

Prepare an ItemReader that implements logic that acquires only unprocessed data with input / output results.

Output at restart

Similar to Stateless restart caution is required for output to non-transactional resource.
In the case of a file, assuming that the context is not used, it is necessary to design such that file addition is permitted.

Stateful restart,similar to Job rerun reruns the job with the same condition as with the failed job. Unlike stateless restart, `-restart` option is not used.

An example of implementing an easy stateful restart is shown below.

Processing specification

1. Define a processed column in the input target table, and update it with a value other than NULL if the processing succeeds.
   - For the extraction condition of unprocessed data, the value of the processed column is NULL.
2. Output the processing result to a file.

RestartOnConditionRepository.xml

```
<!-- (1) -->
<select id="findByProcessedIsNull"

resultType="org.terasoluna.batch.functionaltest.app.model.plan.SalesPlanDetail">
    <![CDATA[
    SELECT
        branch_id AS branchId, year, month, customer_id AS customerId, amount
    FROM
        sales_plan_detail
    WHERE
        processed IS NULL
    ORDER BY
        branch_id ASC, year ASC, month ASC, customer_id ASC
    ]]>
</select>

<!-- (2) -->
<update id="update"
parameterType="org.terasoluna.batch.functionaltest.app.model.plan.SalesPlanDetail">
    <![CDATA[
    UPDATE
        sales_plan_detail
    SET
        processed = '1'
    WHERE
        branch_id = #{branchId}
    AND
        year = #{year}
    AND
        month = #{month}
    AND
        customer_id = #{customerId}
    ]]>
</update>
```

restartOnConditionBasisJob.xml

```
<!-- (3) -->
<bean id="reader" class="org.mybatis.spring.batch.MyBatisCursorItemReader"

p:queryId="org.terasoluna.batch.functionaltest.ch06.reprocessing.repository.RestartOnC
onditionRepository.findByZeroOrLessAmount"
      p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

<!-- (4) -->
<bean id="dbWriter" class="org.mybatis.spring.batch.MyBatisBatchItemWriter"

p:statementId="org.terasoluna.batch.functionaltest.ch06.reprocessing.repository.Restar
```

```xml
tOnConditionRepository.update"
      p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"/>

<bean id="fileWriter"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:resource="file:#{jobParameters[outputFile]}"
      p:appendAllowed="true"> <!-- (5) -->
    <property name="lineAggregator">
        <bean
class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
            <property name="fieldExtractor">
                <bean
class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
                      p:names="branchId,year,month,customerId,amount"/>
            </property>
        </bean>
    </property>
</bean>
<!-- (6) -->
<bean id="compositeWriter"
class="org.springframework.batch.item.support.CompositeItemWriter">
    <property name="delegates">
        <list>
            <ref bean="fileWriter"/>
            <ref bean="dbWriter"/>
        </list>
    </property>
</bean>

<batch:job id="restartOnConditionBasisJob"
           job-repository="jobRepository" restartable="false"> <!-- (7) -->

    <batch:step id="restartOnConditionBasisJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader" processor="amountUpdateItemProcessor"
                         writer="compositeWriter" commit-interval="10" />
        </batch:tasklet>
    </batch:step>

</batch:job>
```

Example of restart command execution

```
# (8)
java -cp dependency/*
org.springframework.batch.core.launch.support.CommandLineJobRunner <jobPath> <jobName>
<jobParameters> ...
```

Description

Sr.No.	Description
(1)	Define SQL so that the processed column has only NULL data.
(2)	Define SQL to update processed columns with non-NULL
(3)	For ItemReader, set the SQLID defined in (1).
(4)	For updating to the database, set the SQLID defined in (2).
(5)	At restart,allow addition of files in order to make it possible to write from the last interruption point.
(6)	Set `CompositeItemWriter` to be processed in the order of file output → database update, and set it to chunk writer.
(7)	It is not mandatory, but set the `restartable` attribute to false so that it will get an error if it is started accidentally with the `-restart` option.
(8)	Execute again according to the execution condition of the failed job.

About the job's restartable attribute

If `restartable`is true, as explained in Stateless restart, use the context information to skip input / output data. If you are using ItemReader or ItemWriter provided by Spring Batch in stateful restart, there is a possibility that this process may stop processing as expected. Therefore, by setting `restartable` to false, activation with the `-restart` option will result in an error, preventing malfunction.

Chapter 7. Job Management

7.1. Overview

Explain how to manage job execution.

This function is the same usage for chunk model and tasklet model.

7.1.1. What is Job Execution Management?

It means to record the activation state and execution result of the job and maintain the batch system. In particular, it is important to secure necessary information in order to detect when an abnormality has occurred and determine what action should be taken next (such as rerun / restart after abnormal termination). Due to the characteristics of the batch application, it is rare that the result can be confirmed on the user interface immediately after startup. Therefore, it is necessary to have a mechanism to record execution status and results separately from job execution, such as job scheduler / RDBMS / application log.

7.1.1.1. Functions Offered by Spring Batch

Spring Batch provides the following interface for job execution management.

List of job management functions

Function	Corresponding interface
Record job execution status/result	`org.springframework.batch.core.repository.JobRepository`
Convert job exit code and process exit code	`org.springframework.batch.core.launch.support.ExitCodeMapper`

Spring Batch uses `JobRepository` for recording the job's activation status and execution result. For TERASOLUNA Batch 5.x, if all of the following are true, persistence is optional:

- Using TERASOLUNA Batch 5.x only for synchronous job execution.
- Managing all job execution with the job scheduler including job stop/restart.
  - Especially not using restart assuming `JobRepository` of Spring Batch.

When these are applicable, use `H2` which is an in-memory/built-in database as an option of RDBMS used by `JobRepository`.
On the other hand, when using asynchronous execution or when using stop/restart of Spring Batch, an RDBMS that can persist the job execution status/result is required.

Default transaction isolation level

In xsd provided by Spring Batch, the transaction isolation level of `JobRepository` has `SERIALIZABLE` as the default value. However, in this case, when multiple jobs are executed concurrently regardless of whether it is synchronous or asynchronous, an exception occurs in updating `JobRepository`. Therefore, TERASOLUNA Batch 5.x sets the transaction isolation level of `JobRepository` to `READ_COMMITTED` in advance.

In-memory JobRepository Options

Spring Batch has `org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean` which performs job execution management in-memory, but it is not used in this guideline.
As shown in the Javadoc of this class, It is explained that it is for testing as `This repository is only really intended for use in testing and rapid prototyping.` and that it is inappropriate for parallel processing as `Not suited for use in multi-threaded jobs with splits`

For the job execution management using job scheduler, refer to the manual of each product.
In this guideline, explain the following items related to managing the job status using `JobRepository` within TERASOLUNA Batch 5.x.

Items related to state management within TERASOLUNA Batch

- Job Status Management
  - How to persist state
  - How to check the status
  - How to stop the job manually
- Customizing Exit Codes
- Double Activation Prevention
- Logging
- Message Management

7.2. How to use

`JobRepository` provided by Spring Batch registers/updates the job status/execution result in RDBMS automatically.
When confirming them, select one of the following methods so that unintended change processing is not performed from inside or outside the batch application.

- Query the table relating to Job Status Management
- Use `org.springframework.batch.core.explore.JobExplorer`

7.2.1. Job Status Management

Explain job status management method using `JobRepository`.
By Spring Batch, the following Entities are registered in the RDBMS table.

Entity class and table name managed by JobRepository

Sr. No.	Entity class	Table name	Generation unit	Desctiption
(1)	`JobExecution`	`BATCH_JOB_EXECUTION`	Execution of one job	Maintain job status/execution result.
(2)	`JobExecutionContext`	`BATCH_JOB_EXECUTION_CONTEXT`	Execution of one job	Maintain the context inside the job.
(3)	`JobExecutionParams`	`BATCH_JOB_EXECUTION_PARAMS`	Execution of one job	Hold job parameters given at startup.
(4)	`StepExecution`	`BATCH_STEP_EXECUTION`	Execution of one step	Maintain the state/execution result of the step, commit/rollback number.
(5)	`StepExecutionContext`	`BATCH_STEP_EXECUTION_CONTEXT`	Execution of one step	Maintain the context inside the step.
(6)	`JobInstance`	`BATCH_JOB_INSTANCE`	Combination of job name and job parameter	Hold job name and string serialized job parameter.

For example, when three steps are executed with one job execution, the following difference occurs.

- `JobExecution`, `JobExecutionContext` and `JobExecutionParams` register 1 record.
- `StepExecution` and `StepExecutionContext` register 3 record.

Also, `JobInstance` is used to suppress double execution by the same name job and same parameter started in the past, TERASOLUNA Batch 5.x does not check this. For details, refer to Double Activation Prevention.

The structure of each table by `JobRepository` is described in Architecture of Spring Batch.

About the item count of StepExecution in the chunk method

As shown below, it seems that inconsistency is occurring, but there are cases where it is reasonable from the specification.

- The transaction issue count of `StepExecution(BATCH_STEP_EXECUTION table)` sometimes does not match the number of input data counts.
  - The number of transaction issues refers to the sum of `COMMIT_COUNT` and `ROLLBACK_COUNT` of `BATCH_STEP_EXECUTION`.
    However, `COMMIT_COUNT` becomes +1 if the number of input data is divisible by the chunk size.
    This is because null representing the end is also counted as input data and is processed empty, after reading the number of input data counts.
- The number of processing of `BATCH_STEP_EXECUTION` and `BATCH_STEP_EXECUTION_CONTEXT` may be different.
  - In `READ_COUNT` and `WRITE_COUNT` of `BATCH_STEP_EXECUTION`, the number of items read and written by `ItemReader` and `ItemWriter` are recorded.
  - The `SHORT_CONTEXT` column in the `BATCH_STEP_EXECUTION_CONTEXT` table records the number of read operations by `ItemReader` in JSON format. However, it does not necessarily match the number of processing by `BATCH_STEP_EXECUTION`.
  - This means that the `BATCH_STEP_EXECUTION` table based on the chunk method records the number of reads and writes irrespective of success or failure, whereas the `BATCH_STEP_EXECUTION_CONTEXT` table records the position to be resumed by a restart in case of a failure in the course of processing.

7.2.1.1. Status Persistence

By using external RDBMS, job execution management information by `JobRepository` can be made persistent. To enable this, change the following items in batch-application.properties to be data sources, schema settings for external RDBMS.

batch-application.properties

```
# (1)
# Admin DataSource settings.
admin.jdbc.driver=org.postgresql.Driver
admin.jdbc.url=jdbc:postgresql://serverhost:5432/admin
admin.jdbc.username=postgres
admin.jdbc.password=postgres

# (2)
spring-batch.schema.script=classpath:org/springframework/batch/core/schema-
postgresql.sql
```

List of setting contents (PostgreSQL)

Sr. No.	Description
(1)	Describe the setting of the external RDBMS to be connected as the value of the property to which the prefix `admin` is attached.
(2)	Specify a script file to automatically generate the schema as `JobRepository` at application startup.

Supplementary to administrative/business data sources

- Connection settings to DB are separately defined as management and business data sources.
  TERASOLUNA Batch 5.x separately defined, `JobRepository` has been set up to use an administrative data source prefixed with `admin` as its property prefix.
- When using asynchronous execution (DB polling), specify the same management data source and schema generation script in the job request table.
  For details, refer to Asynchronous Execution (DB polling).

7.2.1.2. Confirmation of job status/execution result

Explain how to check the job execution status from `JobRepository`
In either method, the job execution ID to be checked is known in advance.

7.2.1.2.1. Query directly

Using the RDBMS console, query directly on the table persisted by `JobRepository`.

SQL sample

```
admin=# select JOB_EXECUTION_ID, START_TIME, END_TIME, STATUS, EXIT_CODE from
BATCH_JOB_EXECUTION where JOB_EXECUTION_ID = 1;
 job_execution_id |       start_time        |        end_time         |  status   |
exit_code
------------------+-------------------------+-------------------------+-----------+
-----------
                1 | 2017-02-14 17:57:38.486 | 2017-02-14 18:19:45.421 | COMPLETED |
COMPLETED
(1 row)
admin=# select JOB_EXECUTION_ID, STEP_EXECUTION_ID, START_TIME, END_TIME, STATUS,
EXIT_CODE from BATCH_STEP_EXECUTION where JOB_EXECUTION_ID = 1;
 job_execution_id | step_execution_id |       start_time        |        end_time
|  status   | exit_code
------------------+-------------------+-------------------------+
-------------------------+-----------+-----------
                1 |                 1 | 2017-02-14 17:57:38.524 | 2017-02-14 18:19:45
.41 | COMPLETED | COMPLETED
(1 row)
```

7.2.1.2.2. Use JobExplorer

Under sharing the application context of the batch application, JobExplorer enables to confirm job execution status by injecting it.

API code sample

```
// omitted.

@Inject
private JobExplorer jobExplorer;

private void monitor(long jobExecutionId) {

  // (1)
  JobExecution jobExecution = jobExplorer.getJobExecution(jobExecutionId);

  // (2)
  String jobName = jobExecution.getJobInstance().getJobName();
  Date jobStartTime = jobExecution.getStartTime();
  Date jobEndTime = jobExecution.getEndTime();
  BatchStatus jobBatchStatus = jobExecution.getStatus();
  String jobExitCode = jobExecution.getExitStatus().getExitCode();

  // omitted.

  // (3)
  jobExecution.getStepExecutions().forEach( s -> {
      String stepName = s.getStepName();
      Date stepStartTime = s.getStartTime();
      Date stepEndTime = s.getEndTime();
      BatchStatus stepStatus = s.getStatus();
      String stepExitCode = s.getExitStatus().getExitCode();

      // omitted.
  });
}
```

List of setting contents (PostgreSQL)

Sr. No.	Description
(1)	Specify the job execution ID from the injected JobExplorer and get JobExecution.
(2)	Get the job execution result by JobExecution.
(3)	Get a collection of steps executed within the job from JobExecution and get individual execution result.

7.2.1.3. Stopping a Job

"Stopping a job" is a function that updates the running status of `JobRepository` to a stopping status and stops jobs at the boundary of steps or at chunk commit by chunk method.
Combined with restart, processing from the stopped position can be restarted.

> For details of the restart, refer to Job Restart.

> "Stopping a job" is not a function to immediately stop a job in progress but to update the running status of `JobRepository` to the stopping status.
> It does not perform any kind of stop processing such as interrupting the in-process thread immediately for the job.
>
> Therefore, it can be said that stopping the job is "to reserve to stop when a processing that becomes a milestone is completed, such as a chunk break". For example, even if you stop the job under the following circumstances, it will not be the expected behavior.
>
> - Job execution consisting of `Tasklet` in a single step.
> - When number of data input < `commit-interval` in chunk method.
> - When an infinite loop has occurred within processing.

Explain how to stop the job below.

- Stop from command line
  - Available for both synchronous and asynchronous jobs
  - Use `-stop` option of `CommandLineJobRunner`

The method of specifying the job name at startup

```
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
    classpath:/META-INF/jobs/job01/job01.xml job01 -stop
```

- Stopping a job by its name specification is suitable for synchronous batch execution when jobs with the same name rarely start in parallel.

The method of specifying the job execution ID (JobExecutionId)

```
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
    classpath:/META-INF/jobs/job01/job01.xml 3 -stop
```

- Stopping a job by JobExecutionId specification is suitable for asynchronous batch execution when jobs with tha same name often start in parallel.

- For the confirmation method of JobExecutionId, refer to Confirmation of job status/execution result.
- `JobOpertion#stop()` can be used to stop the job based on the JobExecutionId. For stopping jobs using `JobOperation#stop()`, refer to Stop and Restart Async Job.

7.2.2. Customizing Exit Codes

When the job is terminated by synchronous execution, the exit code of the java process can be customized according to the end status of the job. To customize the exit code, the following two operations are required.

1. Change the exit code of the job, which indicates the end status of the job.
2. Map the exit code (character string) of the job/step and the process exit code (numerical value).

Explain the following in order.

7.2.2.1. Change the exit code of the job.

The exit code of the job returned as a string can be changed.

- Implement the `afterStep` method to return a specific exit status at the end of the step.
  - Implement `StepExecutionListener`

An implementation example of StepExecutionListener

```
@Component
public class ExitStatusChangeListener implements StepExecutionListener {

    @Override
    public ExitStatus afterStep(StepExecution stepExecution) {

        ExitStatus exitStatus = stepExecution.getExitStatus();
        if (conditionalCheck(stepExecution)) {
            // (1)
            exitStatus = new ExitStatus("CUSTOM STEP FAILED");
        }
        return exitStatus;
    }

    private boolean conditionalCheck(StepExecution stepExecution) {
        // omitted.
    }
}
```

Job definition

```
<batch:step id="exitstatusjob.step">
    <batch:tasklet transaction-manager="transactionManager">
        <batch:chunk reader="reader" writer="writer" commit-interval="10" />
    </batch:tasklet>
    <batch:listeners>
        <batch:listener ref="exitStatusChangeListener"/>
    </batch:listeners>
</batch:step>
```

List of implementation contents

Sr. No.	Description
(1)	Set custom exit code according to the execution result of step.

- Reflect the exit code returned by the step at the end of the job as the final job exit code.
    - Implement `afterJob` method in implementation class of `JobExecutionListener`.

An Implementation example of JobExecutionListener

```
@Component
public class JobExitCodeChangeListener extends JobExecutionListenerSupport {

    @Override
    public void afterJob(JobExecution jobExecution) {
        // (1)
        if (jobExecution.getStepExecutions().stream()
                .anyMatch(s -> "CUSTOM STEP FAILED".equals(s.getExitStatus
().getExitCode()))) {
            jobExecution.setExitStatus(new ExitStatus("CUSTOM FAILED"));
        }
    }
}
```

Job definition

```
<batch:job id="exitstatusjob" job-repository="jobRepository">
    <batch:step id="exitstatusjob.step">
        <!-- omitted -->
    </batch:step>
    <batch:listeners>
        <batch:listener ref="jobExitCodeChangeListener"/>
    </batch:listeners>
</batch:job>
```

List of implementation contents

Sr. No.	Description
(1)	Set the final job exit code to `JobExecution` according to the execution result of the job. In this case, if `CUSTOM STEP FAILED` is included in one of the exit codes returned from the step, It has an exit code `CUSTOM FAILED`.

7.2.2.2. Define mapping of exit codes additionally.

- Define the mapping between the exit code of the job and the process exit code.

launch-context.xml

```xml
<!-- exitCodeMapper -->
<bean id="exitCodeMapper"
      class="org.springframework.batch.core.launch.support.SimpleJvmExitCodeMapper">
    <property name="mapping">
        <util:map id="exitCodeMapper" key-type="java.lang.String"
                  value-type="java.lang.Integer">
            <!-- ExitStatus -->
            <entry key="NOOP" value="0" />
            <entry key="COMPLETED" value="0" />
            <entry key="STOPPED" value="255" />
            <entry key="FAILED" value="255" />
            <entry key="UNKNOWN" value="255" />
            <entry key="CUSTOM FAILED" value="100" /> <!-- Custom Exit Status -->
        </util:map>
    </property>
</bean>
```

Process exit code 1 is prohibited

Generally, when a Java process is forcibly terminated due to a VM crash or SIGKILL signal reception, the process may return 1 as the exit status. Since it should be clearly distinguished from the end state of a batch application regardless of whether it is normal or abnormal, do not define 1 as a process exit code within an application.

About the difference between status and exit code

There are "status (`STATUS`)" and "exit code (`EXIT_CODE`)" as the states of jobs and steps managed by `JobRepository`, but they differ in the following points.

- The status can not be customized because it is used in internal control of Spring Batch and specific value by enum type `BatchStatus` is defined.
- The exit code can be used for job flow control and process exit code change, and can be customized.

7.2.3. Double Activation Prevention

In Spring Batch, when running a job, confirm whether the following combination exists from `JobRepository` to `JobInstance(BATCH_JOB_INSTANCE table)`.

- Job name to be activated
- Job parameters

TERASOLUNA Batch 5.x makes it possible to activate multiple times even if the combinations of job and job parameters match.
That is, it allows double activation. For details, refer to Job Activation Parameter

In order to prevent double activation, it is necessary to execute in the job scheduler or application.
Detailed means are strongly dependent on job scheduler products and business requirements, so omitted here.
Consider whether it is necessary to suppress double start for each job.

7.2.4. Logging

Explain log setting method.

Log output, settings and considerations are in common with TERASOLUNA Server 5.x. At first, refer to Logging.

Explain specific considerations of TERASOLUNA Batch 5.x here.

7.2.4.1. Clarification of log output source

It is necessary to be able to clearly specify the output source job and job execution at the time of batch execution. Therefore, it is good to output the thread name, the job name and the job execution ID. Especially at asynchronous execution, since jobs with the same name will operate in parallel with different threads, recording only the job name may make it difficult to specify the log output source.

Each element can be realized in the following way.

Thread name
  Specify `%thread` which is the output pattern of `logback.xml`

Job name / Job Execution ID
  Create a component implementing `JobExecutionListener` and record it at the start and end of the job

An implementation example of JobExecutionListener

```java
// package and import omitted.

@Component
public class JobExecutionLoggingListener implements JobExecutionListener {
    private static final Logger logger =
            LoggerFactory.getLogger(JobExecutionLoggingListener.class);

    @Override
    public void beforeJob(JobExecution jobExecution) {
        // (1)
        logger.info("job started. [JobName:{}][jobExecutionId:{}]",
            jobExecution.getJobInstance().getJobName(), jobExecution.getId());
    }

    @Override
    public void afterJob(JobExecution jobExecution) {
        // (2)
        logger.info("job finished.[JobName:{}][jobExecutionId:{}][ExitStatus:{}]"
                , jobExecution.getJobInstance().getJobName(),
                , jobExecution.getId(), jobExecution.getExitStatus().getExitCode());
    }

}
```

Job Bean definition file

```xml
<!-- omitted. -->
<batch:job id="loggingJob" job-repository="jobRepository">
    <batch:step id="loggingJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <!-- omitted. -->
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <!-- (3) -->
        <batch:listener ref="jobExecutionLoggingListener"/>
    </batch:listeners>
</batch:job>
<!-- omitted. -->
```

An example of log output of job name and job execution ID

Sr. No.	Description
(1)	Before starting the job, the job name and job execution ID are output to the INFO log.
(2)	When the job ends, an exit code is also output in addition to (1).

Sr. No.	Description
(3)	Associate `JobExecutionLoggingListener` registered as a component with the bean definition of a specific job.

7.2.4.2. Log Monitoring

In the batch application, the log is the main user interface of the operation. Unless the monitoring target and the actions at the time of occurrence are clearly designed, filtering becomes difficult and there is a danger that logs necessary for action are buried. For this reason, it is advisable to determine in advance a message or code system to be a keyword to be monitored for logs. For message management to be output to the log, refer to Message Management below.

7.2.4.3. Log Output Destination

For log output destinations in batch applications, it is good to design in which units logs are distributed / aggregated. For example, even when logs are output to a flat file, multiple patterns are considered as follows.

- Output to 1 file per 1 job
- Output to 1 file in units of multiple jobs grouped together
- 1 Output to 1 file per server
- Output multiple servers in 1 file

In each case, depending on the total number of jobs / the total amount of logs / I/O rate to be generated in the target system, it is decided which unit is best to be grouped. It also depends on how to check logs. It is assumed that options will change depending on the utilization method such as whether to refer frequently from the job scheduler or from the console frequently.

The important thing is to carefully examine the log output in operational design and to verify the usefulness of the log in the test.

7.2.5. Message Management

Explain message management.

In order to prevent variations in the code system and to facilitate designing extraction as a keyword to be monitored, it is desirable to give messages according to certain rules.

As with logging, message management is basically the same as TERASOLUNA Server 5.x.

About utilization of MessageSource

`MessageSource` can be used to use messages from property files.

- For specific settings and implementation examples, refer to Uniform management of log messages
  - As a sample of log output here, it is exemplified along the case of the Spring MVC controller, but please replace it to any component of Spring Batch.
  - Instance of `MessageSource` is generated independently here, but it is not necessary for TERASOLUNA Batch 5.x. This is because each component is accessed only after `ApplicationContext` is created. In TERASOLUNA Batch 5.x, it is set as follows.

launch-context.xml

```
<bean id="messageSource"

class="org.springframework.context.support.ResourceBundleMessageSource
"
      p:basenames="classpath:META-INF/i18n/application-messages" />
```

7.3. Appendix. Spring Batch Admin

Spring Batch Admin is a subproject of Spring Batch, and it is possible to check the job execution status through the Web interface. Introduce it because it can refer easily regardless of test/commercial environment.

Spring Batch Admin is distributed as a sample application. Use Apache Tomcat as the web container and deploy the war file here.

Spring Batch Admin can not only check the execution status and results of jobs but also start and stop jobs. In that case, It is needed to include the job in the same war file, which creates a strong restriction that it is essential to execute the job with the web container. Since there is no necessity as long as it is just confirming the execution state/result, it is introduced as a reference method here.

The installation procedure is as follows.

- Download 1.3.1.RELEASE zip file from Release Distribution Site, and unzip it anywhere.
- Create an external RDBMS-defined property file `batch-RDBMSNAME.properties`.
  - Place it in `spring-batch-admin-1.3.1.RELEASE/spring-batch-admin-sample/src/main/resources`.

batch-postgresql.properties (for PostgreSQL)

```
# Placeholders batch.*
#    for PostgreSQL:
# (1)
batch.jdbc.driver=org.postgresql.Driver
batch.jdbc.url=jdbc:postgresql://localhost:5432/admin
batch.jdbc.user=postgres
batch.jdbc.password=postgres
batch.jdbc.testWhileIdle=true
batch.jdbc.validationQuery=SELECT 1
# (2)
batch.schema.script=classpath:/org/springframework/batch/core/schema-postgresql.sql
batch.drop.script=classpath*:/org/springframework/batch/core/schema-drop-
postgresql.sql
batch.business.schema.script=classpath:/business-schema-postgresql.sql
batch.database.incrementer.class=org.springframework.jdbc.support.incrementer.PostgreS
QLSequenceMaxValueIncrementer

# Non-platform dependent settings that you might like to change
# (3)
batch.data.source.init=false
```

List of setting contents (PostgreSQL)

Sr. No.	Description
(1)	Describe the JDBC driver setting of the connection destination RDBMS.
(2)	Describe `JobRepository` and a script for initializing the business database. It is not used for the reason of (3), but if `batch.xxxx` runs short, it will cause an error at startup, so it may be a dummy, so describe it.
(3)	Be sure to specify `false` so that `JobRepository` and the business database schema are not initialized when Spring Batch Admin is started. If this setting is not described, all the management/business data source on the RDBMS will be cleared.

- Add the JDBC driver dependency library to pom.xml under `spring-batch-admin-1.3.1.RELEASE/spring-batch-admin-sample/`

Add the JDBC driver dependency library to pom.xml (PostgreSQL)

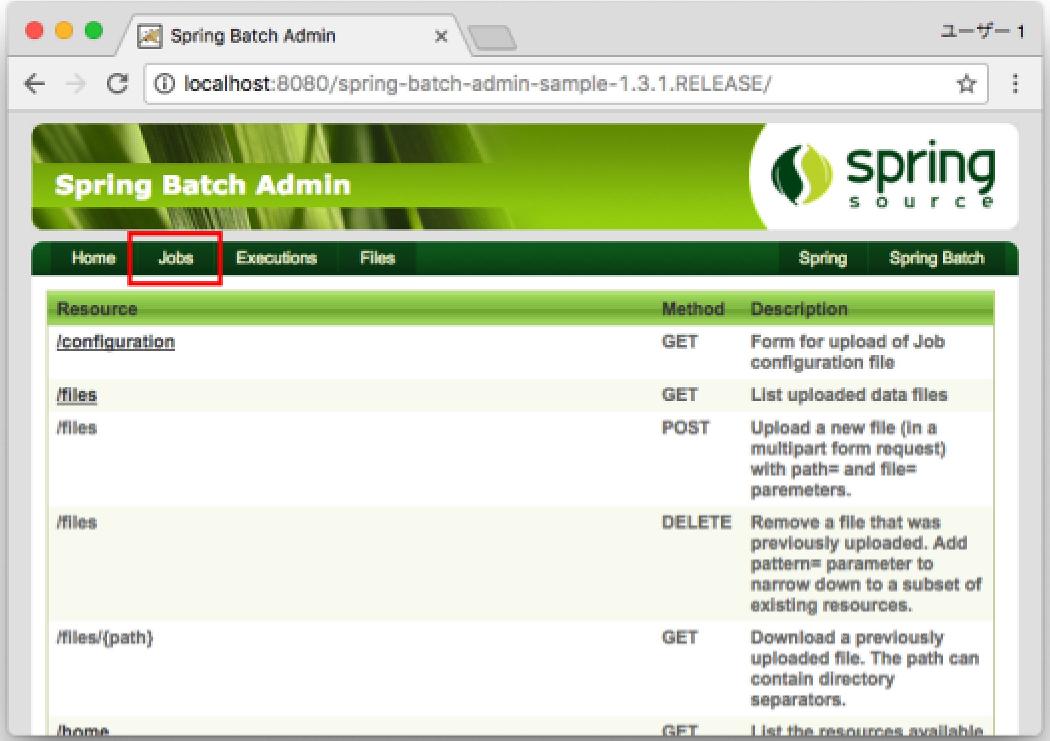
```
<project>
  <!-- omitted. -->
  <dependencies>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>9.4.1212.jre7</version>
        <scope>runtime</scope>
    </dependency>
  </dependencies>
  <!-- omitted. -->
</project>
```

- Create a war file with `mvn clean package` command.
- Set the external RDBMS name `-DENVIRONMENT=postgresql` to the environment variable `JAVA_OPTS` and start Tomcat.

Setting environment variables and starting Tomcat

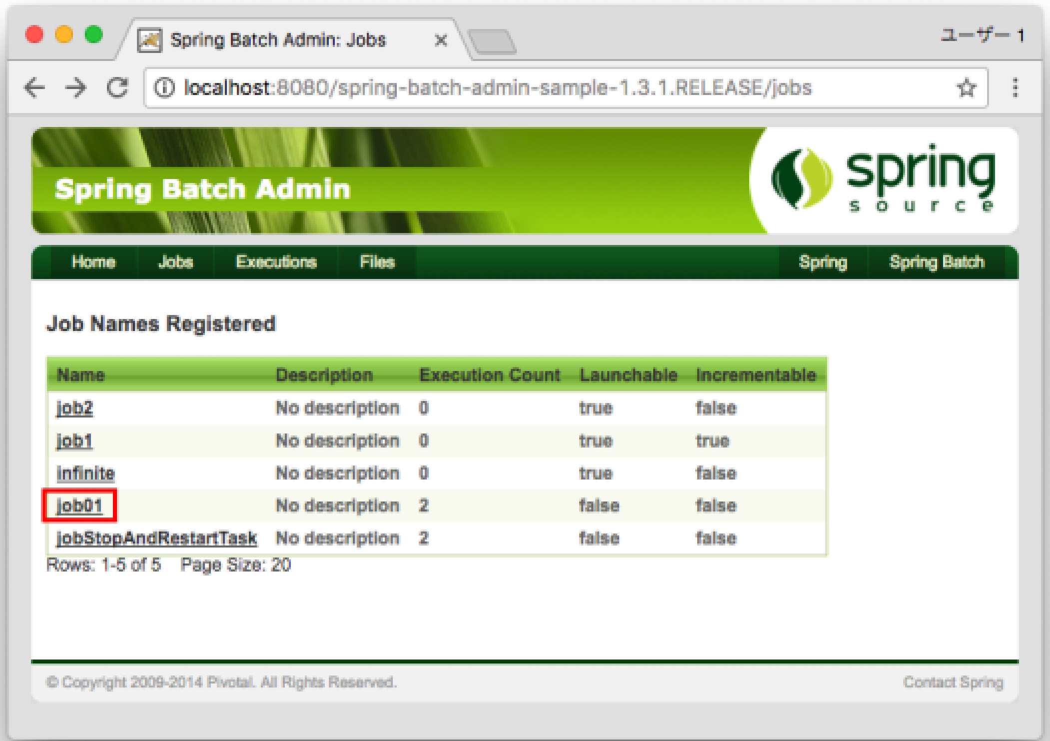
```
$ export JAVA_OPTS="$JAVA_OPTS -DENVIRONMENT=postgresql"
$ echo $JAVA_OPTS
-DENVIRONMENT=postgresql
$ TOMCAT_HOME/bin/catalina.sh run
```

- Deploy `target/spring-batch-admin-1.3.1.war` to Tomcat.
- In the browser, open http://tomcathost:port/spring-batch-admin-sample-1.3.1.RELEASE/ and select Jobs.



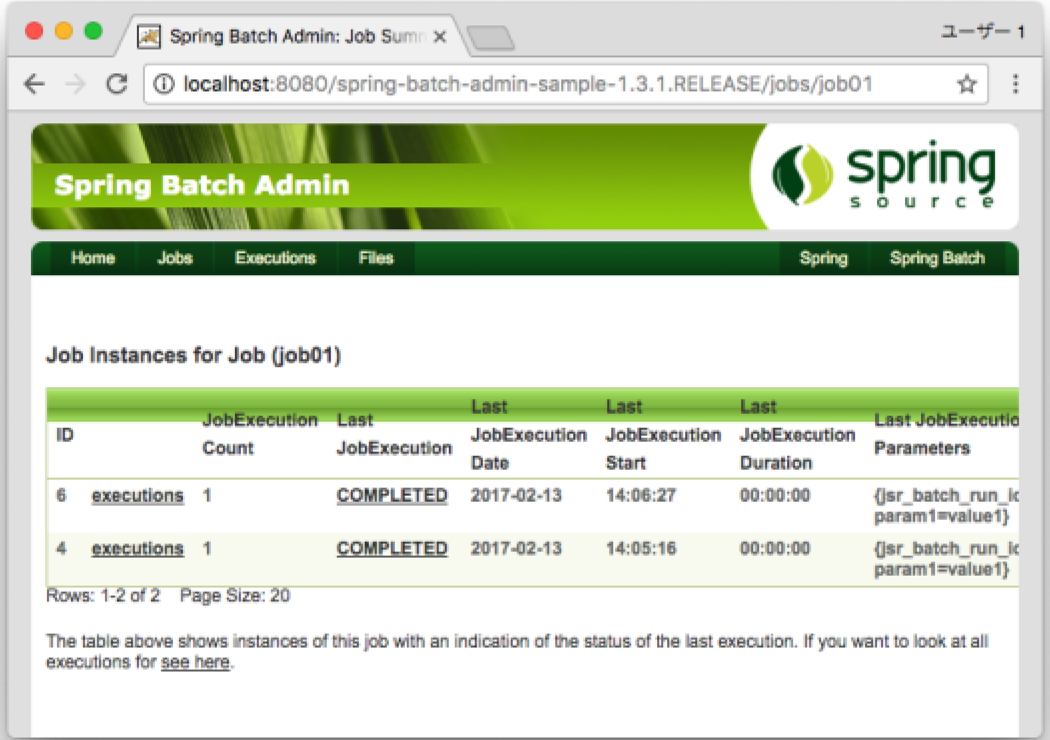
Spring Batch Admin root view

- Select the job name of the execution status/result acquisition target.



Spring Batch Admin job select view

• Display the execution status and result of the target job.



Spring Batch Admin job status/result view

Chapter 8. Flow control and parallel, multiple processing

8.1. Flow control

8.1.1. Overview

It is a method of implementing a single business process by splitting one job to multiple jobs and combining them instead of implementing by integrating them in one job. The item wherein dependency relationship between jobs is defined, is called as job net.

The advantages of defining a job net are enumerated below.

- It is easier to visualize progress status of a process
- It is possible to do partial re-execution, pending execution and stop execution of jobs
- It is easier to do parallel execution of jobs

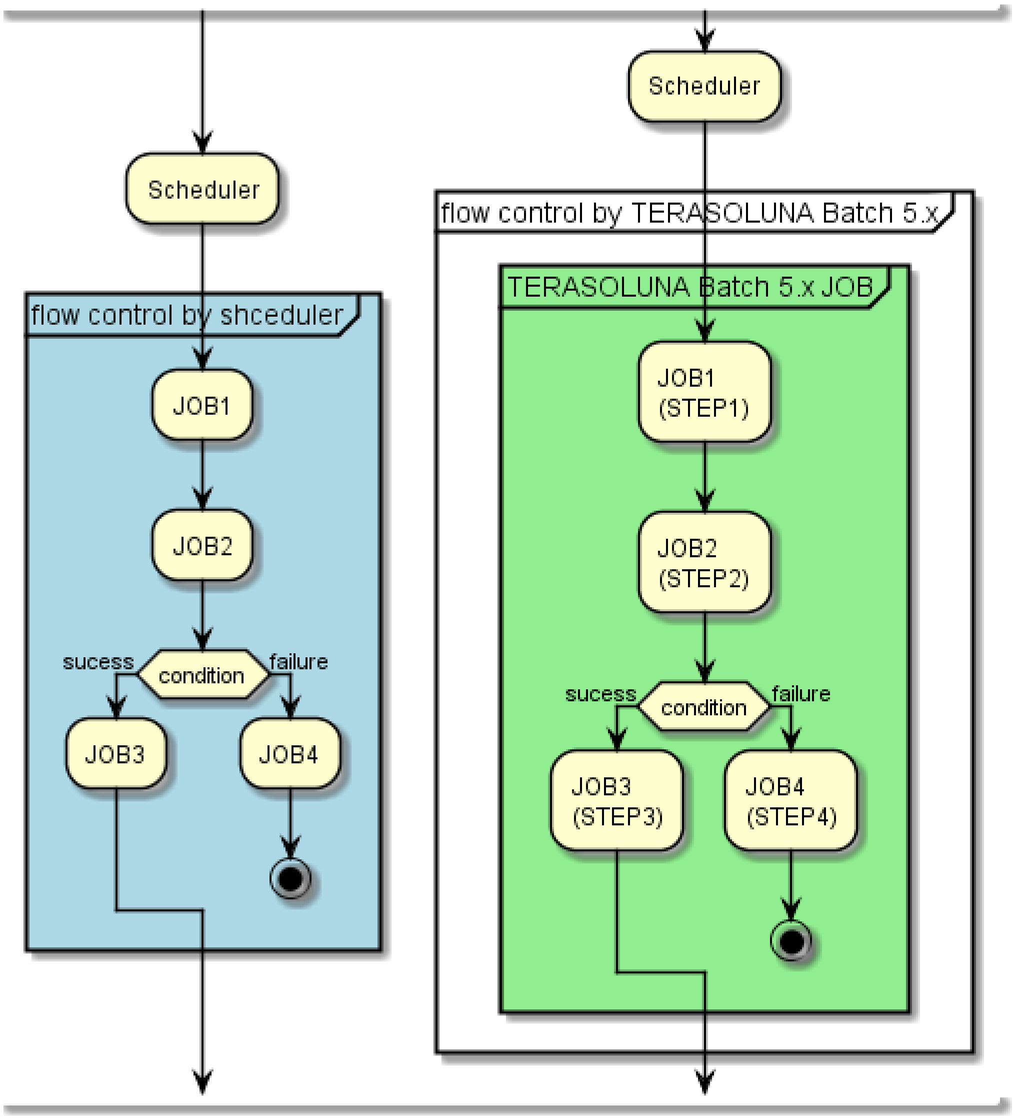
When designing a batch process, it is common to design the job net and the jobs together.

Suitability of processing contents and job net

Job nets are often not suitable for simple business process that need not be split and the process for integrating with online process.

In this guideline, control of the flow of jobs between job nets is called flow control. In the processing flow, the previous job is called as preceding job and the next job is called as subsequent job. The dependency relationship between the preceding job and the subsequent job is called preceding and succeeding relationship.

The conceptual diagram of flow control is shown below.



Overview of flow control

As shown above, flow control can be implemented by both the job scheduler and TERASOLUNA Batch 5.x. However, it is desirable to use the job scheduler as much as possible due to the following reasons.

When implemented in TERASOLUNA Batch 5.x

- There is a strong tendency to have diverse processes and status for one job making it easier to form a black box.
- The boundary between the job scheduler and the job becomes ambiguous
- It becomes difficult to see the situation at the time of error from the job scheduler

However, it is generally known that there are following disadvantages when the number of jobs defined in the job scheduler increases.

- The cost mentioned below increases and the processing time of the entire system increases due to the job scheduler
  - Job scheduler product specific communication, control of execution node, etc.
  - Overhead cost associated with Java process start for each job

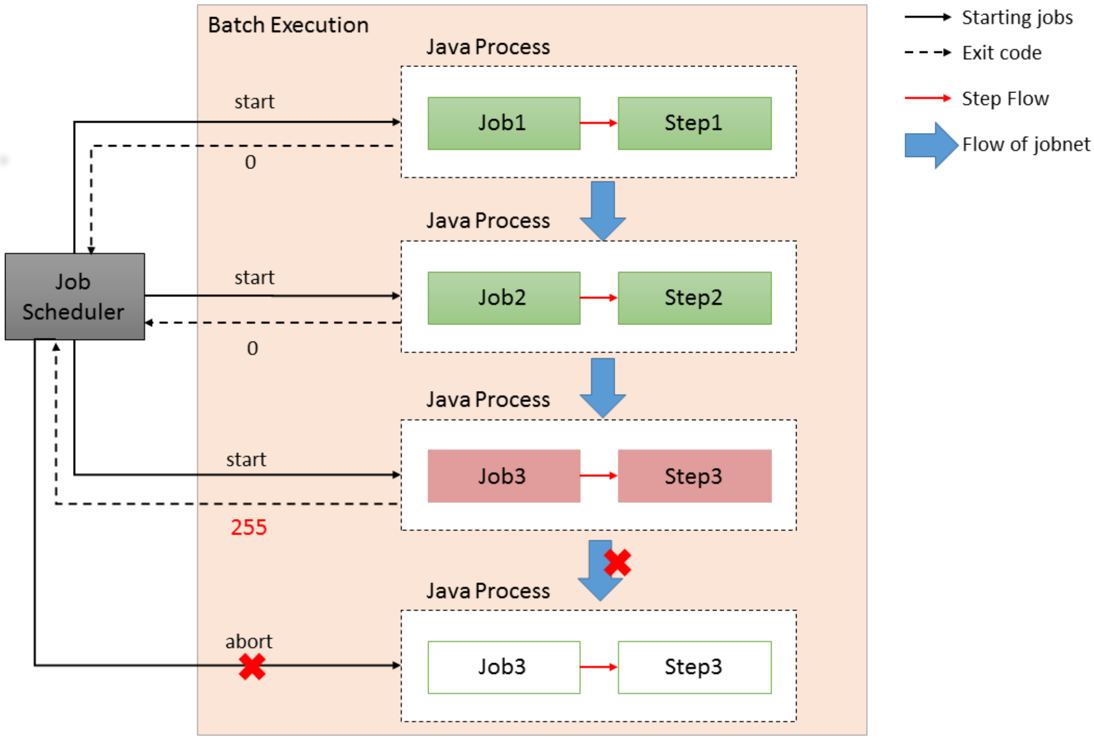
- Number of job registrations limit

The policy is as follows.

- Basically, flow control is performed by the job scheduler.
- Following measures are taken only when any damage is caused due to the large number of jobs.
  - Multiple sequential processes are consolidated in one job in TERASOLUNA Batch 5.x.
    - Simple preceding and succeeding relationships are only consolidated in one job.
    - Changing the step exit code and conditional branch of executing subsequent step by it can be used functionally. However, because job execution management becomes complicated, it is used in principle only for the process exit code determination at the end of the job.

Refer to Customization of exit code for the details of deciding job exit code.

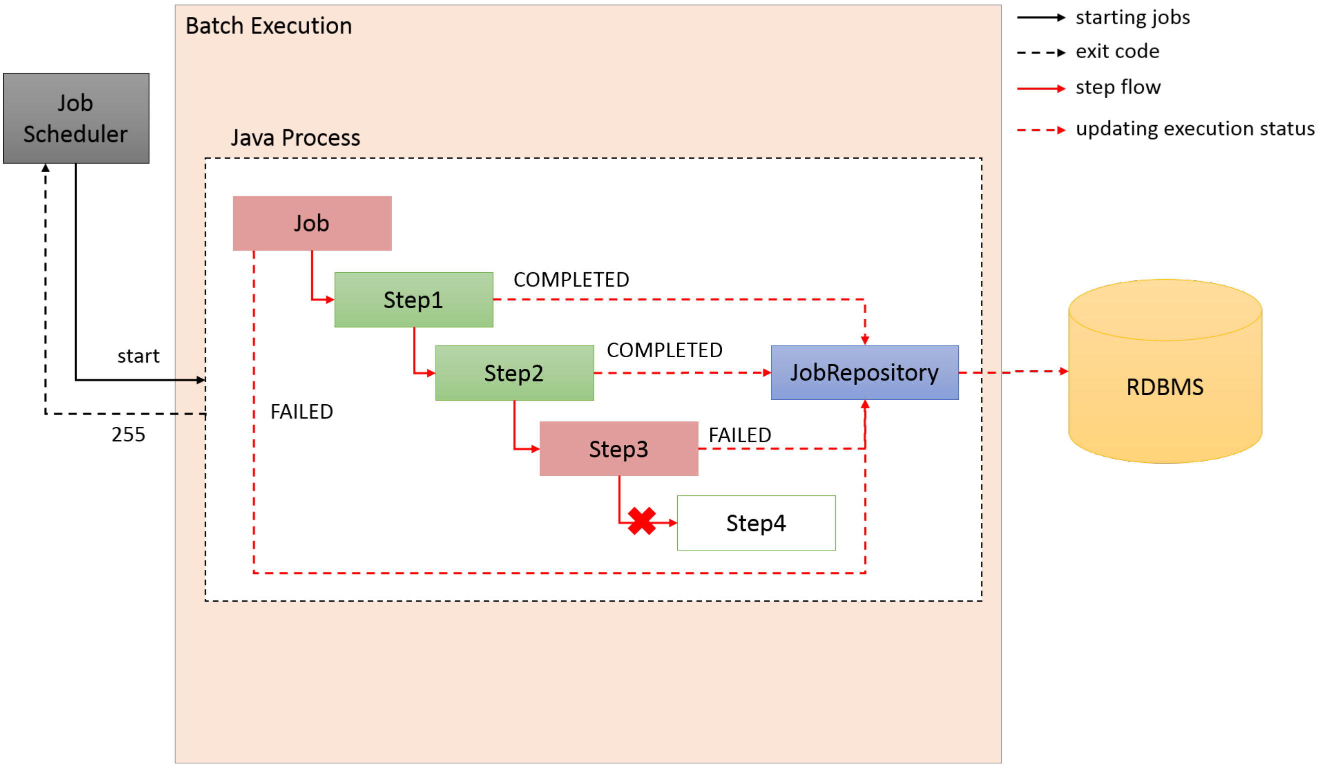
The points to be considered for implementing preceding and succeeding process are shown below.



Flow control by job scheduler

Points to be considered

- Job scheduler starts java process through shell.
- One job should correspond to one java process.
  - In the entire process, 4 java processes start.
- The job scheduler controls the start order of each process. Each java process is independent.
- The process exit code of the preceding job is used for deciding the start of the succeeding job.
- External resources such as files, DB etc. should be used to pass the data between jobs.



Flow control by TERASOLUNA Batch 5.x

Points to be considered

- Job scheduler starts java process through shell.
- One job should be one java process.
    - In the entire process, only one java process is used.
- Start order of each step is controlled by one java process. Each step is independent.
- The exit code of the preceding job is used for deciding the start of the succeeding job.
- The data can be passed between steps by in-memory.

How to implement flow control by TERASOLUNA Batch 5.x is explained below.
The flow control of job scheduler is strongly dependent on the product specifications so it is not explained here.

> *Application example of flow control*
>
> In general, parallel/multiple processes of multiple jobs is often implemented by job scheduler and job net.
> However, in TERASOLUNA Batch 5.x, how to implement parallel and multiple processes of multiple jobs by applying the flow control function, is explained.
> Refer to Parallel and multiple process for details.

The usage method of this function is same in the chunk model as well as tasklet model.

8.1.2. How to use

How to use flow control in TERASOLUNA Batch 5.x is explained.

8.1.2.1. Sequential flow

Sequential flow is a flow that links the before step and after step in series.
If any business process ends abnormally in a step of the sequential flow, the succeeding step is not executed and the job is interrupted. In this case, the step and job status and exit code associated with the job execution ID are recorded as `FAILED` by `JobRepository`.
By restarting after recovering the cause of failure, it is possible to resume the process from the abnormally ended step.

> Refer to Restart job for how to restart a job.

Set sequential flow of the job consisting of 3 steps.

Bean definition

```
<bean id="sequentialFlowTasklet"
    class="org.terasoluna.batch.functionaltest.ch08.flowcontrol.SequentialFlowTasklet"
        p:failExecutionStep="#{jobParameters[failExecutionStep]}" scope="step"/>

<batch:step id="parentStep">
    <batch:tasklet ref="sequentialFlowTasklet"
                   transaction-manager="jobTransactionManager"/>
</batch:step>

<batch:job id="jobSequentialFlow" job-repository="jobRepository">
    <batch:step id="jobSequentialFlow.step1"
                next="jobSequentialFlow.step2" parent="parentStep"/> <!-- (1) -->
    <batch:step id="jobSequentialFlow.step2"
                next="jobSequentialFlow.step3" parent="parentStep"/> <!-- (1) -->
    <batch:step id="jobSequentialFlow.step3" parent="parentStep"/>   <!-- (2) -->
</batch:job>
```

Sr. No.	Explanation
(1)	Specify the next step to be started after this step ends normally in `<batch:step>`. Set `id` of the next step to `next` attribute.
(2)	`next` attribute is not required in the step at the end of the flow.

As a result, steps are started in series in the following order.
`jobSequentialFlow.step1` → `jobSequentialFlow.step2` → `jobSequentialFlow.step3`

How to define using <batch:flow>

In the above example, the flow is directly defined in `<batch: job>` . Flow definition can be defined outside using `<batch:flow>`. An example of using `<batch:flow>` is shown below.

```
<batch:job id="jobSequentialOuterFlow" job-repository="jobRepository">
    <batch:flow id="innerFlow" parent="outerFlow"/> <!-- (1) -->
</batch:job>

<!-- (2) -->
<batch:flow id="outerFlow">
    <batch:step id="jobSequentialOuterFlow.step1"
                next="jobSequentialOuterFlow.step2"
                parent="parentStep"/>
    <batch:step id="jobSequentialOuterFlow.step2"
                next="jobSequentialOuterFlow.step3"
                parent="parentStep"/>
    <batch:step id="jobSequentialOuterFlow.step3"
                parent="parentStep"/>
</batch:flow>
```

Sr. No.	Explanation
(1)	Set flow id is defined in (2) in the parent attribute.
(2)	Define sequential flow.

8.1.2.2. Passing data between steps

In Spring Batch, `ExecutionContext` of execution context that can be used in the scope of each step and job is provided. By using the execution context, data can be shared between the components in the step. At this time, since the execution context of the step cannot be shared between steps, the execution context of the preceding step cannot be referred from the succeeding step. It can be implemented if the execution context of the job is used, but since it can be referred from all steps, it needsto be handled carefully. When the information between the steps needs to be inherited, it can be done by the following procedure.

1. In the post-processing of the preceding step, the information stored in the execution scope of the step scope is passed to the execution context of the job scope.
2. The succeeding step gets information from the execution context of the job scope.

By using `ExecutionContextPromotionListener` provided by Spring Batch, the first procedure can be realized only by specifying the inherited information to the listener even without implementing it.

Notes on using ExecutionContext

`ExecutionContext` used for passing data is saved in serialized state in `BATCH_JOB_EXECUTION_CONTEXT` and `BATCH_JOB_STEP_EXECUTION_CONTEXT` of RDBMS so note the following 3 points.

1. The passed data should be an object in a serializable format.
   - `java.io.Serializable` should be implemented.
2. Minimum required passed data should be retained.
   `ExecutionContext` is also used for storing execution control information by Spring Batch, so larger the passed data, the more the serialization cost.
3. The above `ExecutionContextPromotionListener` should be used for passed data without saving it directly in the job execution context.
   The job execution context has a wider scope than the step, unnecessary serialized data gets easily accumulated.

Also, it is possible to exchange information by sharing Bean of Singleton or Job scope rather than going through the execution context. Note that the larger the size of this method more the pressure on memory resources.

The data passed between steps is explained for the tasklet model and the chunk model respectively below.

8.1.2.2.1. Data passing between steps using tasklet model

In order to save and fetch passing data, get `ExecutionContext` from `ChunkContext` and pass the data between the steps.

Implementation example of data passing source Tasklet

```
// package, imports are omitted.

@Component
public class SavePromotionalTasklet implements Tasklet {

    // omitted.

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception {

        // (1)
        chunkContext.getStepContext().getStepExecution().getExecutionContext()
                .put("promotion", "value1");

        // omitted.

        return RepeatStatus.FINISHED;
    }
}
```

Implementation example of data passing destination Tasklet

```
// package and imports are omitted.

@Component
public class ConfirmPromotionalTasklet implements Tasklet {

    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) {
        // (2)
        Object promotion = chunkContext.getStepContext().getJobExecutionContext()
                .get("promotion");

        // omitted.

        return RepeatStatus.FINISHED;
    }
}
```

Description example of job Bean definition

```
<!-- import,annotation,component-scan definitions are omitted -->

<batch:job id="jobPromotionalFlow" job-repository="jobRepository">
    <batch:step id="jobPromotionalFlow.step1" next="jobPromotionalFlow.step2">
        <batch:tasklet ref="savePromotionalTasklet"
                       transaction-manager="jobTransactionManager"/>
        <batch:listeners>
            <batch:listener>
                <!-- (3) -->
                <bean
class="org.springframework.batch.core.listener.ExecutionContextPromotionListener"
                      p:keys="promotion"
                      p:strict="true"/>
            </batch:listener>
        </batch:listeners>
    </batch:step>
    <batch:step id="jobPromotionalFlow.step2">
        <batch:tasklet ref="confirmPromotionalTasklet"
                       transaction-manager="jobTransactionManager"/>
    </batch:step>
</batch:job>
<!-- omitted -->
```

Explanation of implementation contents

Sr. No.	Explanation
(1)	Set the value to be passed to the after step in the `ExecutionContext` of the step execution context. Here, `promotion` is specified as the key required for a series of data passing.
(2)	Get the passing data set in (1) of the preceding step using `promotion` key specified in the passing source from `ExecutionContext`. Note that the `ExecutionContext` used here is not the step execution context of (1) but is the job execution context.
(3)	Using `ExecutionContextPromotionListener`, pass the data from the step execution context to the job execution context. Specify the passing key specified in (1) to `keys` property. `IllegalArgumentException` is thrown by `strict=true` property when it does not exist in the step execution context. In case of `false`, processing continues even if there is no passing data.

Regarding ExecutionContextPromotionListener and step exit code

`ExecutionContextPromotionListener` passes the data from step execution context to job execution context only when the step exit code of data passing source ends normally (`COMPLETED`).
To customize the exit code in which the succeeding step is executed continuously, exit code should be specified in `status` property in the array format.

8.1.2.2.2. Data passing between steps using the chunk model

Use the method assigned with `@AfterStep`、`@BeforeStep` annotation in `ItemProcessor`. The listener to be used for data passing and how to use `ExecutionContext` is the same as the tasklet model.

Implementation example of data passing source ItemProcessor

```
// package and imports are omitted.

@Component
@Scope("step")
public class PromotionSourceItemProcessor implements ItemProcessor<String, String> {

    @Override
    public String process(String item) {
        // omitted.
    }

    @AfterStep
    public ExitStatus afterStep(StepExecution stepExecution) {
        // (1)
        ExecutionContext jobContext = stepExecution.getExecutionContext();
        // (2)
        jobContext.put("promotion", "value2");
        return null;
    }
}
```

Implementation example of data passing target ItemProcessor

```
// package and imports are omitted.

@Component
@Scope("step")
public class PromotionTargetItemProcessor implements ItemProcessor<String, String> {

    @Override
    public String process(String item) {
        // omitted.
    }

    @BeforeStep
    public void beforeStep(StepExecution stepExecution) {
        // (3)
        ExecutionContext jobContext = stepExecution.getJobExecution()
                .getExecutionContext();
        // omitted.
    }
}
```

Description example of job Bean definition

```
<!-- import,annotation,component-scan definitions are omitted -->
<batch:job id="jobChunkPromotionalFlow" job-repository="jobRepository">
    <batch:step id="jobChunkPromotionalFlow.step1" parent="sourceStep"
                next="jobChunkPromotionalFlow.step2">
        <batch:listeners>
            <batch:listener>
                <!-- (4) -->
                <bean
class="org.springframework.batch.core.listener.ExecutionContextPromotionListener"
                      p:keys="promotion"
                      p:strict="true" />
            </batch:listener>
        </batch:listeners>
    </batch:step>
    <batch:step id="jobChunkPromotionalFlow.step2" parent="targetStep"/>
</batch:job>

<!-- step definitions are omitted. -->
```

Explanation of implementation contents

Sr. No.	Explanation
(1)	Set the value to be passed to the succeeding step to `ExecutionContext` of step execution context. Here, `promotion` is specified as the key required for a series of data passing.
(2)	Get the passing data set in (1) of the preceding step using `promotion` key specified in the passing source from `ExecutionContext`. Note that the `ExecutionContext` used here is not the step execution context of (1) but is the job execution context.
(3)	Using `ExecutionContextPromotionListener`, pass the data from the step execution context to the job execution context. Specification of property is same as the tasklet.

8.1.3. How to extend

Here, conditional branching of succeeding step and the condition to stop the job before executing the after step isexplained.

> *Difference between exit code and status of job and step.*
>
> In the following explanation, the terms "Status" and "Exit code" frequently appear.
> When distinction cannot be made, it may cause confusion, so refer to "Customization of exit code.

8.1.3.1. Conditional branching

The conditional branch means receiving the exit code which is the execution result of the preceding

step, selecting one from multiple after steps and continuing execution.
To stop the job without executing any succeeding step, refer to Stop condition.

Description example Job Bean definition

```
<batch:job id="jobConditionalFlow" job-repository="jobRepository">
    <batch:step id="jobConditionalFlow.stepA" parent="conditionalFlow.parentStep">
        <!-- (1) -->
        <batch:next on="COMPLETED" to="jobConditionalFlow.stepB" />
        <batch:next on="FAILED" to="jobConditionalFlow.stepC"/>
    </batch:step>
    <!-- (2) -->
    <batch:step id="jobConditionalFlow.stepB" parent="conditionalFlow.parentStep"/>
    <!-- (3) -->
    <batch:step id="jobConditionalFlow.stepC" parent="conditionalFlow.parentStep"/>
</batch:job>
```

Explanation of implementation contents

Sr. No.	Explanation
(1)	Do not specify `next` attribute in the `<batch:step>` element as in the sequential flow. By setting multiple `<batch:next>`, it can be assigned to the succeeding step specified by `to` attribute. Specify exit code of the step that is the transition condition, in `on`.
(2)	It is the succeeding step executed only when the step exit code of (1) is `COMPLETED`.
(3)	It is the succeeding step executed only when the step exit code of (1) is `FAILED`. By specifying this, succeeding step such as recovery process etc. is executed without stopping the job when the preceding stepprocess fails.

Notes on recovery process by after steps

When recovery process of the succeeding step is performed due to failure of preceding step process (Exit code is `FAILED`), the status of before step changes to `ABANDONED` and it cannot restart regardless of success or failure of the recovery process.

When the recovery process of the succeeding step fails, only the recovery process is re-executed on restarting the job.
For this reason, when processing is to be performed again including the preceding step, it is necessary to rerun as anotherjob execution.

8.1.3.2. Stop condition

How to stop the job depending on the exit code of the preceding step, is explained.
There are methods to specify the following 3 elements as means to stop.

1. `end`
2. `fail`
3. `stop`

If these exit codes correspond to the preceding step, the succeeding step is not executed.
Multiple exit codes can be specified within the same step.

Description example of job Bean definition

```
<batch:job id="jobStopFlow" job-repository="jobRepository">
    <batch:step id="jobStopFlow.step1" parent="stopFlow.parentStep">
        <!-- (1) -->
        <batch:end on="END_WITH_NO_EXIT_CODE"/>
        <batch:end on="END_WITH_EXIT_CODE" exit-code="COMPLETED_CUSTOM"/>
        <!-- (2) -->
        <batch:next on="*" to="jobStopFlow.step2"/>
    </batch:step>
    <batch:step id="jobStopFlow.step2" parent="stopFlow.parentStep">
        <!-- (3) -->
        <batch:fail on="FORCE_FAIL_WITH_NO_EXIT_CODE"/>
        <batch:fail on="FORCE_FAIL_WITH_EXIT_CODE" exit-code="FAILED_CUSTOM"/>
        <!-- (2) -->
        <batch:next on="*" to="jobStopFlow.step3"/>
    </batch:step>
    <batch:step id="jobStopFlow.step3" parent="stopFlow.parentStep">
        <!-- (4) -->
        <batch:stop on="FORCE_STOP" restart="jobStopFlow.step4" exit-code=""/>
        <!-- (2) -->
        <batch:next on="*" to="jobStopFlow.step4"/>
    </batch:step>
    <batch:step id="jobStopFlow.step4" parent="stopFlow.parentStep"/>
</batch:job>
```

Explanation of setting contents of job stop

Sr. No.	Explanation
(1)	When `on` attribute of `<batch:end>` and the step exit code match, the job is recorded as normal end (Status : `COMPLETED`) in `JobRepository`. When `exit-code` attribute is assigned, exit code of job can be customized from `COMPLETED` by default.
(2)	By specifying wildcard (`*`) in `on` attribute of `<batch:next>`, when it does not correspond to any of `end`, `fail`, `stop`, subsequent job can be continued. It is described at the end of step elements however, since matching condition of exist code is evaluated before, the arrangement order of elements is optional if it is within step elements.
(3)	When `<batch:fail>` is used, the job is recorded as abnormal end (Status: `FAILED`) in `JobRepository`. Like `<batch:end>`, by assigning `exit-code` attribute, exit code of job can be customized from `FAILED` by default.

Sr. No.	Explanation
(4)	When `<batch:stop>` is used, the job is recorded as stopped (Status: `STOPPED`) in `JobRepository` at the time ofnormal end of step. For `restart` attribute, specify the stop where the job is resumed from stop at the time of restart. Like `<batch:end>`, `exit-code` attribute can be assigned, however, null string should be specified.(refer to column later)

When customizing the exit code by the exit-code attribute, it should be mapped to the process exit code without omission.

Refer to Customization of exit code for details.

Empty charactor string should be specified to exit-code in <batch:stop>.

```
<step id="step1" parent="s1">
    <stop on="COMPLETED" restart="step2"/>
</step>

<step id="step2" parent="s2"/>
```

The expected flow control should be that when step1 ends normally, the job stops and step2 is executed when restart is executed again.
However, due to some failure in Spring Batch, the operation does not take place as expected.
step2 is not executed after restart, the exit code of job becomes `NOOP` and status `COMPLETED`.

To avoid this, "" (empty charactor string) should be assigned in `exit-code` as shown above.

Refer to Spring Batch/BATCH-2315 for the details of failure.

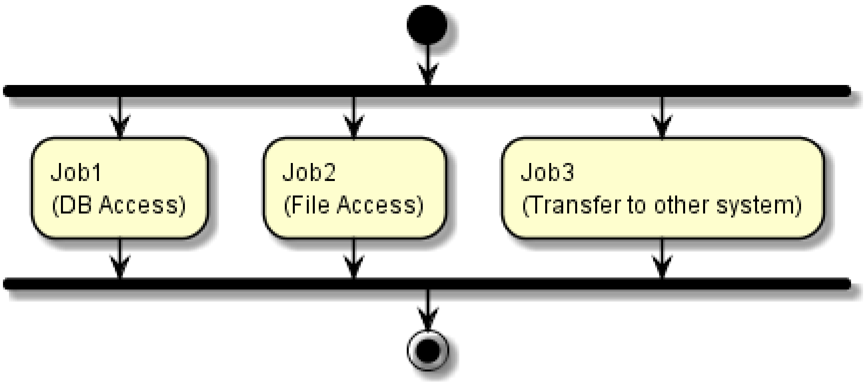
8.2. Parallel processing and multiple processing

8.2.1. Overview

Generally, the batch system where the batch window is severe (time available for batch processing) is designed to reduce overall processing time as much as possible by operating multiple jobs in parallel (hereafter referred to as parallel processing).
However, it may happen that processing time does not fit in the batch window due to large size of 1 processing job.
In this case, a method to reduce processing time by dividing the processing data of a job and performing multiple processing (hereafter referred to as multiple processing) can be used.
Although parallel processing and multiple processing can be handled with the same significance, the definitions are given here as below.

Parallel processing

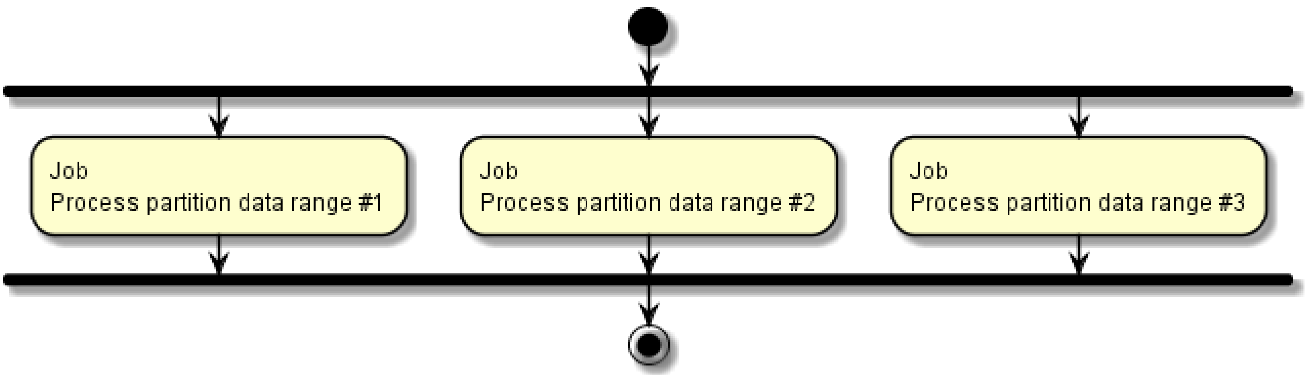
Execute multiple different jobs at the same time.



Schematic diagram of parallel processing

Multiple processing

Divide the processing target of 1 job and execute simultaneously.



Schematic diagram of multiple processing

A method to use job scheduler and a method to use TERASOLUNA Batch 5.x are used for both parallel processing and multiple processing.
Note that, parallel processing and multiple processing in TERASOLUNA Batch 5.x is established on Flow control.

How to implement parallel processing and multiple processing

Implementation method	Parallel processing	Multiple processing
Job scheduler	It is defined to enable execution of multiple different jobs without dependencies to run at the same time.	It is defined to execute multiple identical jobs in different data scopes. Pass information to narrow down data to be processed by each job argument, to each job. For example, divide data of 1 year for each month, divide by units such as area, branch etc.
TERASOLUNA Batch 5.x	Parallel Step (Parallel processing) Perform parallel processing in steps. Each step need not have identical processing and parallel processing can be performed for resources of different types such as DB and file.	Partitioning Step (Multiple processing) In the master step, a key to distribute target data is fetched and in the slave step, the distributed data is processed based on this key. Unlike parallel step, the processing of the slave step is identical.

When job scheduler is used

Since 1 process is allocated to 1 job, it is activated by multiple processes. Hence, designing and implementing one job is not very difficult.
However, since multiple processes are started, the load on machine resources increase when number of synchronous executions increase.
Hence, when the number of synchronous executions is 3 or 4, a job scheduler may be used.
Of course, this number is not absolute. It would like you to used as a guide as it depends on execution environment or job implementation.

When TERASOLUNA Batch 5.x is used

Since each step is assigned to a thread, it is operated as 1 process with multiple threads. Hence, the difficulty level for design and implementation of 1 job is higher than while using a job scheduler.
However, since the process is implemented by multiple threads, the load on machine resources will not be as high as the time when job scheduler is used even when the number of synchronous executions show an increase. Hence, when number of synchronous executions is large (5 or more than 5), TERASOLUNA Batch 5.x may be used.
Of course, this number is not absolute. It would like you to used as a guide as it depends on execution environment and system characteristics.

One of the parallel processing methods that can be executed in Spring Batch is `Multi Thread Step`, however, its use in TERASOLUNA Batch 5.x is deprecated due to following reasons.

A Multi Thread Step method
: performs parallel processing by multiple threads in chunk units.

Reason for deprecation
: A majority of Readers and Writers offered by Spring Batch are not designed for multi-thread processing. Hence, issues like loss of data or duplicate processing are likely to occur resulting in low process reliability. Further, since the process is performed in multiple threads, a definite processing order is not established.
Even when ItemReader/ItemProcessor/ItemWriter is created on its own, various points must be taken into consideration in order to use `Multi Thread Step` like thread safe wherein difficulty of implementation and operation is high. `Multi Thread Step` is deprecated for these reasons.
It is recommended to use Partitioning Step (Multiple processing) as an alternative.

> Existing ItemReader can be made thread safe by using `org.springframework.batch.item.support.SynchronizedItemStreamReader`. Even then issue of processing sequence is still to be considered, `Multi Thread Step` is not used in TERASOLUNA Batch 5.x.

When data is to be updated to 1 database by parallel processing and multiple processing, resource conflict and deadlock are likely to occur. Potential conflicts should be eliminated from the job design stage.

Distributed processing for multiple processes and multiple housings is included in Spring Batch as a function. However, since the failure design becomes difficult for TERASOLUNA Batch 5.x, it should not be used.

The usage method of this function is same in the chunk model as well as tasklet model.

8.2.1.1. Parallel processing and multiple processing by job scheduler

Parallel processing and multiple processing using a job scheduler is explained here.

For job registration and schedule setting, refer the manual of the job scheduler to be used.

8.2.1.1.1. Parallel processing of jobs using job scheduler

The processes to be executed in parallel are registered as jobs and schedules are set so that each job starts on the time. Each job can be registered as a different process.

8.2.1.1.2. Multiple processing of jobs using job scheduler

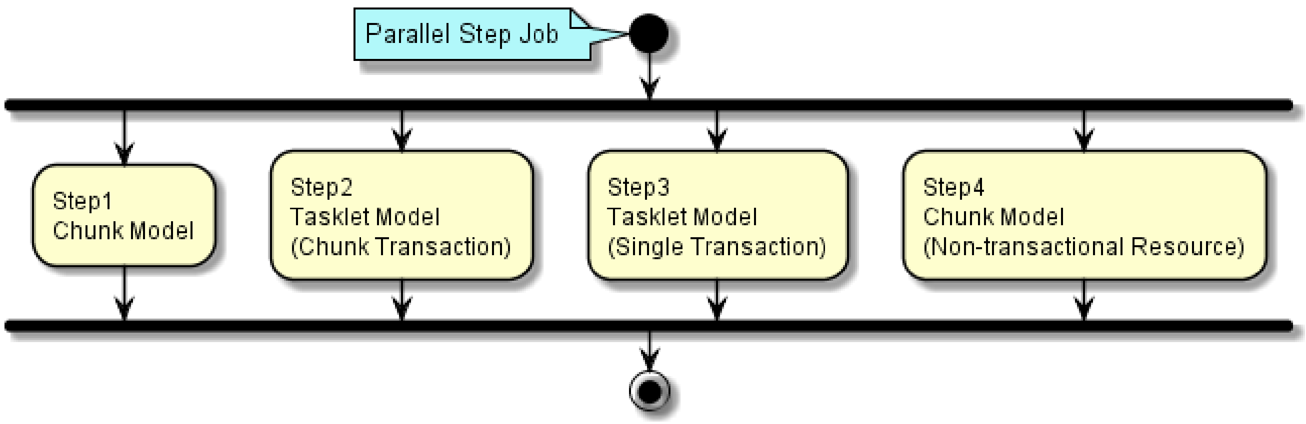
Processes to be subjected to multiple processing are registered multiple times and extraction scope of target data is specified by parameters. Further, the schedule is set to enable the respective jobs at the same time. Although each job is in the same process, data range to be processed must be independent.

8.2.2. How to use

A method to perform parallel processing and multiple processing in TERASOLUNA Batch 5.x is explained.

8.2.2.1. Parallel Step (Parallel processing)

A method of Parallel Step (parallel processing) is explained.



Schematic diagram for Parallel Step

Description of schematic diagram

Separate processes can be defined for each step and can be executed in parallel. A thread is allocated for each step.

How to define Parallel Step is shown below using schematic diagram of Parallel Step.

Job definition of Parallel Step

```xml
<!-- Task Executor -->
<!-- (1) -->
<task:executor id="parallelTaskExecutor" pool-size="10" queue-capacity="200"/>

<!-- Job Definition -->
<!-- (2) -->
<batch:job id="parallelStepJob" job-repository="jobRepository">
  <batch:split id="parallelStepJob.split" task-executor="parallelTaskExecutor">
      <batch:flow>  <!-- (3)  -->
          <batch:step id="parallelStepJob.step.chunk.db">
               <!-- (4) -->
              <batch:tasklet transaction-manager="jobTransactionManager">
                  <batch:chunk reader="fileReader" writer="databaseWriter"
                          commit-interval="100"/>
              </batch:tasklet>
          </batch:step>
      </batch:flow>

      <batch:flow>  <!-- (3)  -->
          <batch:step id="parallelStepJob.step.tasklet.chunk">
               <!-- (5) -->
              <batch:tasklet transaction-manager="jobTransactionManager"
                             ref="chunkTransactionTasklet"/>
          </batch:step>
      </batch:flow>

      <batch:flow>  <!-- (3)  -->
          <batch:step id="parallelStepJob.step.tasklet.single">
               <!-- (6) -->
              <batch:tasklet transaction-manager="jobTransactionManager"
                             ref="singleTransactionTasklet"/>
          </batch:step>
      </batch:flow>

      <batch:flow>  <!-- (3)  -->
          <batch:step id="parallelStepJob.step.chunk.file">
              <batch:tasklet transaction-manager="jobTransactionManager">
                   <!-- (7) -->
                  <batch:chunk reader="databaseReader" writer="fileWriter"
                          commit-interval="200"/>
              </batch:tasklet>
          </batch:step>
      </batch:flow>

  </batch:split>
</batch:job>
```

Description

Description	Sr. No.
(1)	Define a thread pool to assign to each thread for parallel processing.
(2)	Define steps to be executed in parallel in `<batch:split>` tag in a format which uses `<batch:flow>` tag. Set the Bean of thread pool defined in (1), in `task-executor` attribute.
(3)	Define `<batch:step>` wherein parallel processing is to be performed for each `<batch:flow>`.
(4)	Step 1 of schematic diagram ：Define intermediate commit method processing of chunk model.
(5)	Step 2 of schematic diagram ：Define intermediate commit method processing of tasket model.
(6)	Step 3 of schematic diagram ：Define batch commit method processing of tasket model.
(7)	Step 4 of schematic diagram ：Define intermediate commit method processing for non-transactional resources of chunk model.

Cases wherein processing performance deteriorates due to parallel processing

In the parallel processing, same process can be run in parallel by changing the data range, similar to multiple processing. In this case, data range is assigned by the parameters.+ At this time, when the amount of data to be processed is not very significant for each process, footprints like resource amount and processing time which are occupied at the time of execution proves to be a disadvantage in parallel processing, and on the contrary, processing performance may get deteriorated.

Examples of footprints

- Processing from opening for input resources to fetching initial data range
  - Resource open requires more processing time than fetching data
  - Similarly, a process which initializes memory area of data range requires time

Further, steps of common processing can be defined as well before and after Parallel Step process.

Example of Parallel Step which includes common processing steps

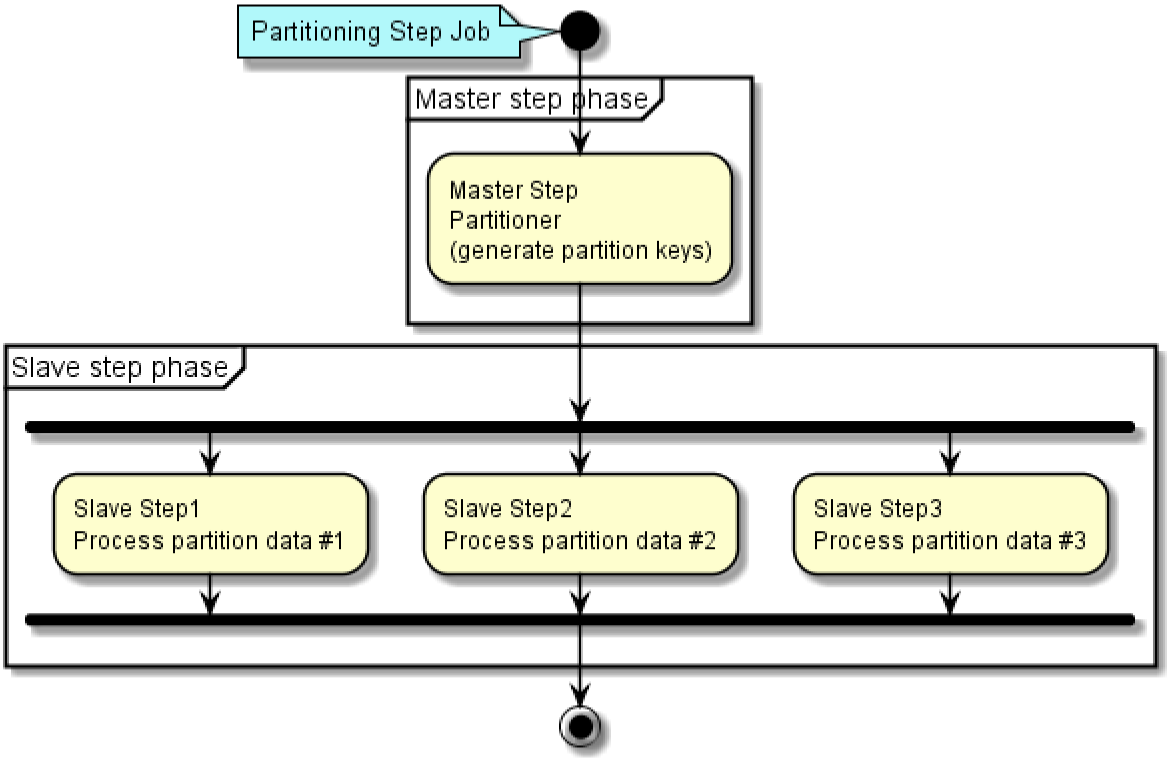
```
<batch:job id="parallelRegisterJob" job-repository="jobRepository">
    <!-- (1) -->
    <batch:step id="parallelRegisterJob.step.preprocess"
                next="parallelRegisterJob.split">
        <batch:tasklet transaction-manager="jobTransactionManager"
                       ref="deleteDetailTasklet" />
    </batch:step>

    <!--(2) -->
    <batch:split id="parallelRegisterJob.split" task-executor="parallelTaskExecutor">
        <batch:flow>
            <batch:step id="parallelRegisterJob.step.plan">
                <batch:tasklet transaction-manager="jobTransactionManager">
                    <batch:chunk reader="planReader" writer="planWriter"
                            commit-interval="1000" />
                </batch:tasklet>
            </batch:step>
        </batch:flow>
        <batch:flow>
            <batch:step id="parallelRegisterJob.step.performance">
                <batch:tasklet transaction-manager="jobTransactionManager">
                    <batch:chunk reader="performanceReader" writer="performanceWriter"
                            commit-interval="1000" />
                </batch:tasklet>
            </batch:step>
        </batch:flow>
    </batch:split>
</batch:job>
```

Description

Sr. No.	Description
(1)	Define steps to be processed as preprocessing. Specify id set in `<batch:split>`, in `next` attribute. For details of subsequent step specification using `next` attribute, refer Sequential flow.
(2)	Define Parallel Step. Define `<batch:step>` wherein parallel processing is to be performed for each `<batch:flow>`.

8.2.2.2. Partitioning Step (Multiple processing)

A method of Partitioning Step (multiple processing) is explained.



Schematic diagram of Partitioning Step

Description of schematic diagram

Partitioning Step is divided into processing phases of Master step and Slave step.

1. In Master step, `Partitioner` generates a `Parition Key` to specify data range wherein each Slave step is processed. `Parition Key` is stored in the step context.
2. In Slave step, `Parition Key` assigned on its own from step context is fetched and data for processing is specified using the same. Step defined for specified data for processing are executed.

In the Partitioning Step, although it is necessary to divide the processing data, either of the variable number and fixed number are handled for the number of partitionings.

Number of partitionings

In case of a variable number

Divide by department or process for each file existing in specific directory

In case of a fixed number

Process data by dividing overall data in fixed numbers

In Spring Batch, fixed number is called `grid-size` and data partitioning range is determined so that `grid-size` becomes `Partitioner`.

In Partitioning Step, number of partitionings can be significantly higher than the thread size. In this case, multiple executions are performed using number of threads and a step is generated wherein the process is not executed until the thread becomes empty.

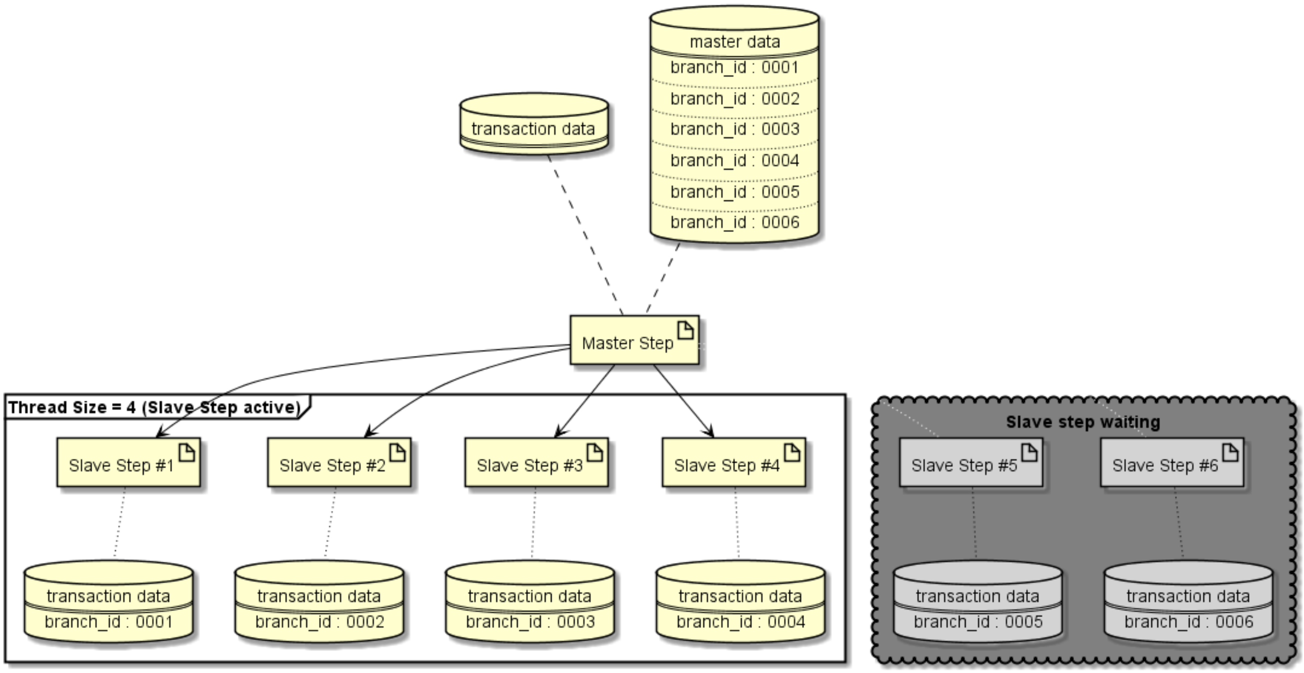
Use case of Partitioning Step is shown below.

Partitioning Step use case

Use case	Master(Patitioner)	Slave	Number of partitionings
A case wherein transaction information is divided or multiple processing is performed from master information Aggregate processing for each department and for each month	DB (Master information)	DB (Transaction information)	Variable
A case wherein multiple processing is performed for 1 file from a list of files Branch-wise multiple processing of transfer data from each branch (Aggregate processing for each branch)	Multiple files	Single file	Variable
A case wherein a large amount of data is divided by a fixed number or multiple processing is performed A case wherein since recovery design other than re-run becomes difficult in case of a failure occurrence, it is not used the actual operation. In case of a re-run, since all the records are processed again, merits of partitioning are eliminated.	Specify data range from `grid-size` and transaction information count	DB (Transaction information)	Fixed

8.2.2.2.1. When number of partitionings are variable

A method wherein number of partitionings are made variable by Partitioning Step is explained. Processing image is shown below.



Processing image diagram

How to implement is shown below using the processing image as an example.

Defining Repository(SQLMapper) (PostgreSQL)

```xml
<!-- (1) -->
<select id="findAll"
resultType="org.terasoluna.batch.functionaltest.app.model.mst.Branch">
    <![CDATA[
    SELECT
        branch_id AS branchId,
        branch_name AS branchName,
        branch_address AS branchAddrss,
        branch_tel AS branchTel,
        create_date AS createDate,
        update_date AS updateDate
    FROM
        branch_mst
    ]]>
</select>

<!-- (2) -->
<select id="summarizeInvoice"

resultType="org.terasoluna.batch.functionaltest.app.model.performance.SalesPerformance
Detail">
    <![CDATA[
    SELECT
        branchId, year, month, customerId, SUM(amount) AS amount
    FROM (
        SELECT
            t2.charge_branch_id AS branchId,
            date_part('year', t1.invoice_date) AS year,
            date_part('month', t1.invoice_date) AS month,
            t1.customer_id AS customerId,
            t1.invoice_amount AS amount
        FROM invoice t1
        INNER JOIN customer_mst t2 ON t1.customer_id = t2.customer_id
        WHERE
            t2.charge_branch_id = #{branchId}
        ) t3
    GROUP BY branchId, year, month, customerId
    ORDER BY branchId ASC, year ASC, month ASC, customerId ASC
    ]]>
</select>

<!-- omitted -->
```

Implementation example of Partitioner

```
@Component
public class BranchPartitioner implements Partitioner {

    @Inject
    BranchRepository branchRepository; // (3)

    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {

        Map<String, ExecutionContext> map = new HashMap<>();
        List<Branch> branches = branchRepository.findAll();

        int index = 0;
        for (Branch branch : branches) {
            ExecutionContext context = new ExecutionContext();
            context.putString("branchId", branch.getBranchId()); // (4)
            map.put("partition" + index, context);  // (5)
            index++;
        }

        return map;
    }
}
```

Bean definition

```
<!-- (6) -->
<task:executor id="parallelTaskExecutor"
               pool-size="${thread.size}" queue-capacity="10"/>

<!-- (7) -->
<bean id="reader" class="org.mybatis.spring.batch.MyBatisCursorItemReader"
scope="step"

p:queryId="org.terasoluna.batch.functionaltest.app.repository.performance.InvoiceRepos
itory.summarizeInvoice"
      p:sqlSessionFactory-ref="jobSqlSessionFactory">
    <property name="parameterValues">
        <map>
            <!-- (8) -->
            <entry key="branchId" value="#{stepExecutionContext[branchId]}" />
        </map>
    </property>
</bean>

<!-- omitted -->

<batch:job id="multipleInvoiceSummarizeJob" job-repository="jobRepository">
    <!-- (9) -->
    <batch:step id="multipleInvoiceSummarizeJob.master">
        <!-- (10) -->
        <batch:partition partitioner="branchPartitioner"
                         step="multipleInvoiceSummarizeJob.slave">
            <!-- (11) -->
            <batch:handler grid-size="0" task-executor="parallelTaskExecutor" />
        </batch:partition>
    </batch:step>
</batch:job>

<!-- (12) -->
<batch:step id="multipleInvoiceSummarizeJob.slave">
    <batch:tasklet transaction-manager="jobTransactionManager">
        <batch:chunk reader="reader" writer="writer" commit-interval="10"/>
    </batch:tasklet>
</batch:step>
```

Description

Sr. No.	Description
(1)	Define a SQL wherein processing target is fetched from master data.
(2)	Define a SQL wherein fetched values from master data are considered as search conditions.
(3)	Inject defined Repository(SQLMapper).

Sr. No.	Description
(4)	Store master value processed by 1 Slave step in the step context.
(5)	Store each Slave in Map so that it can fetch corresponding context.
(6)	Define a thread pool to assign to each thread of Slave step in multiple processing. Master step is processed by the main thread.
(7)	Define ItemReader for fetching data using master value.
(8)	Fetch master value set in (4) from step context and add to search conditions.
(9)	Define Master step.
(10)	Define processing to generate partitioning conditions of data. Set `Partitioner` interface implementation, in `partitioner` attribute. Set Bean ID of Slave Step defined in (12), in `step` attribute.
(11)	Since `grid-size` is not used in `partitioner`, set any arbitrary value in `grid-size` attribute. Set Bean ID of thread pool defined in (6), in `task-executor` attribute.
(12)	Define Slave step. Set ItemReader defined in (7), in `reader` attribute.

When multiple processing is performed for each file from the list of files, `Partitioner` given below offered by Spring Batch can be used.

- `org.springframework.batch.core.partition.support.MultiResourcePartitioner`

How to use `MultiResourcePartitioner` is shown below.

An example wherein multiple processing is performed for files

```
<!-- (1) -->
<task:executor id="parallelTaskExecutor" pool-size="10" queue-capacity="200"/>

<!-- (2) -->
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="#{stepExecutionContext[fileName]}"> <!-- (3) -->
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper"
            p:fieldSetMapper-ref="invoiceFieldSetMapper">
            <property name="lineTokenizer">
                <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                      p:names="invoiceNo,salesDate,productId,customerId,quant,price"/>
            </property>
        </bean>
    </property>
</bean>

<!-- (4) -->
<bean id="patitioner"

class="org.springframework.batch.core.partition.support.MultiResourcePartitioner"
      scope="step"
      p:resources="file:#{jobParameters[basedir]}/input/invoice-*.csv"/> <!-- (5) -->

<!--(6) -->
<batch:job id="inspectPartitioninglStepFileJob" job-repository="jobRepository">
    <batch:step id="inspectPartitioninglStepFileJob.step.master">
        <batch:partition partitioner="patitioner"
                         step="inspectPartitioninglStepFileJob.step.slave">
            <batch:handler grid-size="0" task-executor="parallelTaskExecutor"/>
        </batch:partition>
    </batch:step>
</batch:job>

<!-- (7) -->
<batch:step id="inspectPartitioninglStepFileJob.step.slave">
    <batch:tasklet>
        <batch:chunk reader="reader" writer="writer" commit-interval="20"/>
    </batch:tasklet>
</batch:step>
```

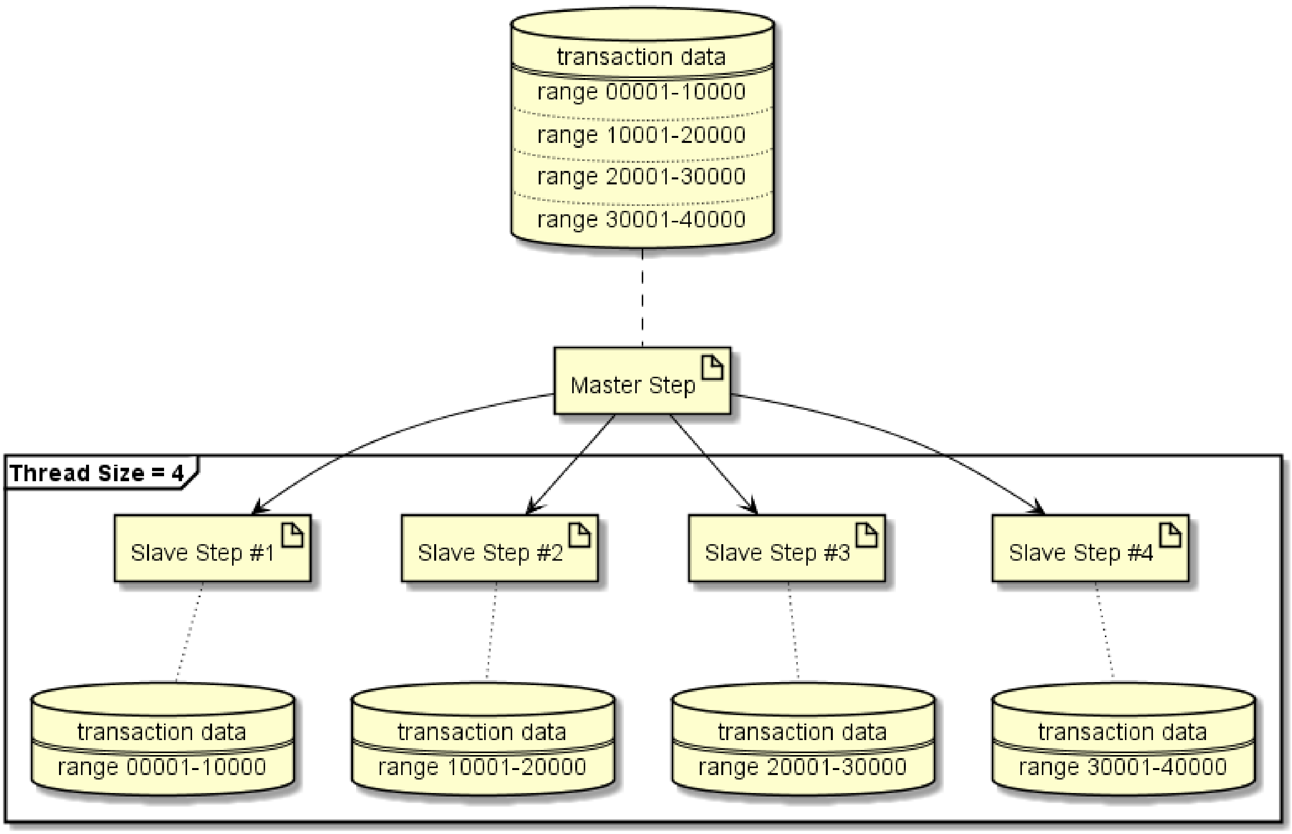
Description

Sr. No.	Description
(1)	Define a thread pool to be assigned to each thread of Slave step in multiple processing. Master step is processed in the main thread.

Sr. No.	Description
(2)	Define ItemReader to read a single file.
(3)	In `resouce` property, specify the file split by `MultiResourcePartitioner` in input file. `MultiResourcePartitioner` stores the file path in the step context using a key called "fileName".
(4)	Define `MultiResourcePartitioner` as `Partitioner`.
(5)	Multiple files can be handled by using a pattern wherein * is used.
(6)	Define Master step. Definition contents are the same as contents of Partitioning Step described above.
(7)	Define Slave step. Set ItemReader defined in (2), in `reader` attribute.

8.2.2.2.2. When number of partitionings are fixed

How to fix number of partitionings in Partitioning Step is explained.
Processing image diagram is shown below.



Processing image diagram

How to implement is shown below using the processing image as an example.

Definition of Repository(SQLMapper) (PostgreSQL)

```
<!-- (1) -->
<select id="findByYearAndMonth"

resultType="org.terasoluna.batch.functionaltest.app.model.performance.SalesPerformance
Summary">
    <![CDATA[
    SELECT
        branch_id AS branchId, year, month, amount
    FROM
        sales_performance_summary
    WHERE
        year = #{year} AND month = #{month}
    ORDER BY
        branch_id ASC
    LIMIT
        #{dataSize}
    OFFSET
        #{offset}
    ]]>
</select>

<!-- (2) -->
<select id="countByYearAndMonth" resultType="_int">
    <![CDATA[
    SELECT
        count(*)
    FROM
        sales_performance_summary
    WHERE
        year = #{year} AND month = #{month}
    ]]>
</select>

<!-- omitted -->
```

Implementation example of Partitioner

```java
@Component
public class SalesDataPartitioner implements Partitioner {

    @Inject
    SalesSummaryRepository repository;  // (3)

    // omitted

    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {

        Map<String, ExecutionContext> map = new HashMap<>();
        int count = repository.countByYearAndMonth(year, month);
        int dataSize = (count / gridSize) + 1;        // (4)
        int offset = 0;

        for (int i = 0; i < gridSize; i++) {
            ExecutionContext context = new ExecutionContext();
            context.putInt("dataSize", dataSize);     // (5)
            context.putInt("offset", offset);         // (6)
            offset += dataSize;
            map.put("partition:" + i, context);       // (7)
        }

        return map;
    }
}
```

Bean definition

```
<!-- (8) -->
<task:executor id="parallelTaskExecutor"
               pool-size="${thread.size}" queue-capacity="10"/>

<!-- (9) -->
<bean id="reader"
      class="org.mybatis.spring.batch.MyBatisCursorItemReader" scope="step"

p:queryId="org.terasoluna.batch.functionaltest.ch08.parallelandmultiple.repository.Sal
esSummaryRepository.findByYearAndMonth"
      p:sqlSessionFactory-ref="jobSqlSessionFactory">
    <property name="parameterValues">
        <map>
            <entry key="year" value="#{new Integer(jobParameters[year])}" />
            <entry key="month" value="#{new Integer(jobParameters[month])}" />
            <!-- (10) -->
            <entry key="dataSize" value="#{stepExecutionContext[dataSize]}" />
            <!-- (11) -->
            <entry key="offset" value="#{stepExecutionContext[offset]}" />
        </map>
    </property>
</bean>

<!-- omitted -->

<batch:job id="multipleCreateSalesPlanSummaryJob" job-repository="jobRepository">
    <!-- (12) -->
    <batch:step id="multipleCreateSalesPlanSummaryJob.master">
        <!-- (13) -->
        <batch:partition partitioner="salesDataPartitioner"
              step="multipleCreateSalesPlanSummaryJob.slave">
            <!-- (14) -->
            <batch:handler grid-size="4" task-executor="parallelTaskExecutor" />
        </batch:partition>
    </batch:step>
</batch:job>

<!-- (15) -->
<batch:step id="multipleCreateSalesPlanSummaryJob.slave">
    <batch:tasklet transaction-manager="jobTransactionManager">
        <batch:chunk reader="reader" processor="addProfitsItemProcessor"
              writer="writer" commit-interval="10"/>
    </batch:tasklet>
</batch:step>
```

Description

Sr. No.	Description
(1)	Define a pagination search (SQL narrowing down method) to fetch a specific data range. For details of pagination search (SQL narrowing down method),refer Pagination search of Entity (SQL narrow down method) of TERASOLUNA Server 5.x Development Guideline.
(2)	Define SQL to fetch total number of records for processing.
(3)	Inject defined Repository(SQLMapper).
(4)	Calculate data records processed by one Slave step.
(5)	Store data records of (4) in step context.
(6)	Store search start position of each Slave step in step context.
(7)	Each Slave is stored in the Map to enable fetching of corresponding context.
(8)	Define a thread poolto be assigned to each thread of Slave step in multiple processing. Master step is processed by main thread.
(9)	Define ItemReader for fetching data by using pagination search (SQL narrow down method).
(10)	Fetch data records set in (5) from step context and add to search conditions.
(11)	Fetch search start position set in (6) from step context and add to search conditions.
(12)	Define Master step.
(13)	Define a process which generates partitioning conditions for data. Set `Partitioner` interface implementation in `partitioner` attribute. Set Bean ID of Slave step defined in (15), in `step` attribute.
(14)	Set number of partitionings (fixed number) in `grid-size` attribute. Set Bean ID of thread pool defined in (8), in `task-executor` attribute.
(15)	Define Slave step. Set ItemReader defined in (9), in `reader` attribute.

Chapter 9. Summary of points

9.1. Notes on TERASOLUNA Batch 5.x

This is a summarized list of the rules and notes about using TERASOLUNA Batch 5.x that are explained in each section. Users should keep in mind the following points and proceed when developing a batch application.

> Only the salient points are mentioned here instead of covering the entire consider material. Users should read the guideline of function to be used.

Rules and notes to be considered for batch process

- Single batch process should be simplified and complex logical structures should be avoided.
- Same operation should not be performed in multiple jobs over and over again.
- Usage of system resources should be minimized, unnecessary physical I/O should be avoided and on-memory operations should be utilized.

Guidelines for TERASOLUNA Batch 5.x

- Development of batch application
  - Create as 1 job=1 Bean definition(1 job definition)
  - Create as 1 step=1 batch process=1 business logic
- Chunk model
  - Use it for efficiently processing large amount of data.
- Tasklet model
  - Use for simple processing, processing that is hard to standardize, and to process data by single commit.
- Synchronous job
  - Use for starting a job as per the schedule and for batch processing by combining multiple jobs.
- Asynchronous job(DB polling)
  - Use for delayed process, continuous execution of job with short processing time and consolidation of large jobs.
- Asynchronous job (Web container)
  - Similar to DB polling. However, use when instantaneous start is required.
- Management of JobRepository

- In Spring Batch, use `JobRepository` for recording start status and execution result of job.
- In TERASOLUNA Batch 5.x, persistence is optional if it corresponds to all the following.
  - Using TERASOLUNA Batch 5.x for executing synchronous job only.
  - All job execution management including stop, restart of job is assigned to the job scheduler.
    - Do not use restart where the `JobRepository` possessed by Spring Batch is a prerequisite.
- When these are applicable, use `H2` which is an in-memory and built-in database as an option of RDBMS used by `JobRepository`. On the other hand, when using asynchronous job or stop and restart by Spring Batch, RDBMS that can make the job execution status and result permanent, is required.
  For this point, Job management should also be read.

How to choose chunk model or tasklet model

- Chunk model
  - When you want to steadily process large amount of data
  - When you want to restart based on the record count
- Tasklet model
  - When you want to make recovery as simple as possible
  - When you want to consolidate the process contents

How to choose chunk model or tasklet model should also be read.

Unification of bean scope

- In the Tasklet implementation, match with the scope of component to be Injected.
- Composite type component matches with the scope of component to be delegated.
- When using JobParameter, set to the scope of `step`.
- If you want to save instance variables in Step unit, set to the scope of `step`.

Performance tuning points

- Adjust chunk size
  - When using Chunk, set the number of commits to an appropriate size. Do not increase the size too much.
- Adjust fetch size
  - In database access, set fetch size to an appropriate size. Do not increase the size too much.

- Make file reading more efficient
  - Provide a dedicated FieldSetMapper interface implementation.
- Parallel process and multiple processes
  - Implement by job scheduler.
- Distributed processing
  - Implement by job scheduler.

Asynchronous job (DB polling)

- Usage of in-memory database
  - It is not suitable for long-term continuous operation so it is desirable to restart it periodically.
  - When it is to be used for long-term continuous operation, maintenance work such as periodically deleting data from `JobRepository` is required.
- Narrow-down of registered job
  - Specify the designed and implemented job based on asynchronous execution.
- Mass processing of very short batch is not suitable since performance deterioration is possible.
- Since parallel execution of the same job is possible, it is necessary to prevent the same job from affecting in parallel execution

---

Asynchronous job (Web container)

- The basic consideration is same as Asynchronous job (DB polling).
- Adjust thread pool.
  - Apart from the thread pool of asynchronous execution, it is necessary to consider the request thread of the Web container and other applications operating within the same unit.
- In Web and batch, you cannot cross-reference data source, MyBatis setting and Mapper interface.
- Failure to start a job due to thread pool exhaustion cannot be catched at job start, so provide a means to confirm it separately.

Database access and transaction

- "Use `MyBatisBatchItemWriter` in ItemWriter" and "Update reference using Mapper interface in ItemProcessor" cannot be done at the same time.
  - There is a restriction in MyBatis of not executing with two or more `ExecutorType` in the same transaction. Refer to Database access of other than ItemReader/ItemWriter.

- Notes on input/output of database to the same table
  - As the result of losing the information that guarantees reading consistency due to output (issue of UPDATE), error may occur in the input (SELECT). Consider the following measures.
    - It depends on the database so, increase the area to secure the information.
    - Split the input data and perform multiple processing.

File access

- When dealing with the following fixed-length file, be sure to use the component provided by TERASOLUNA Batch 5.x.
  - Fixed-length file containing multibyte characters
  - Fixed length file without line break
- When skipping footer records, it is necessary to process with OS command.

Exclusive control

- When multiple jobs are concurrently executed, design a job so that the exclusive control is not required.
  - Resources to be accessed and processing targets should be split for each job.
- Design in such a way that deadlocks are prevented from occurring.
- File exclusive control should be implemented in the tasklet model.

Handling abnormal system

- Do not perform transaction processing in exception handling.
- Note that ChunkListener behaves differently by the process model.
  - The exceptions generated by opening and closing the resources are
    - Chunk model: Not in the scope of catching by ChunkListener interface.
    - Tasklet model: In the scope of catching by ChunkListener interface.
- Input check error cannot be recovered even by restarting unless the input resource that is the cause of check error is modified
- How to cope when a failure occurs in JobRepository should be considered.

About ExecutionContext

- Since `ExecutionContext` is stored in the `JobRepository`, there are following restrictions.
  - The object to be stored in `ExecutionContext` should be the class that implements

`java.io.Serializable`.
    - There should be a limit in the size that can be stored.

Exit code

- The forced end of Java process and the end status of batch application is clearly distinguished.
    - It is strictly prohibited to set the exit code of process to 1 by batch application.

Parallel processing and multiple processing

- Do not use `Multi Thread Step`.
- Depending on the processing content, be careful to possibility of resource contention and deadlock