

Jakarta Batch Guide

JEUS 9.1

TMAXSOFT

Copyright

Copyright 2025. TmaxSoft Co., Ltd. All Rights Reserved.

Company Information

TmaxSoft Co., Ltd.

TmaxSoft Tower, 45, Jeongjail-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, South Korea

Website: <https://www.tmaxsoft.com/en/>

Restricted Rights Legend

All TmaxSoft Software (JEUS®) and documents are protected by copyright laws and international convention. TmaxSoft software and documents are made available under the terms of the TmaxSoft License Agreement and this document may only be distributed or copied in accordance with the terms of this agreement. No part of this document may be transmitted, copied, deployed, or reproduced in any form or by any means, electronic, mechanical, or optical, without the prior written consent of TmaxSoft Co., Ltd. Nothing in this software document and agreement constitutes a transfer of intellectual property rights regardless of whether or not such rights are registered) or any rights to TmaxSoft trademarks, logos, or any other brand features.

This document is for information purposes only. The company assumes no direct or indirect responsibilities for the contents of this document, and does not guarantee that the information contained in this document satisfies certain legal or commercial conditions. The information contained in this document is subject to change without prior notice due to product upgrades or updates. The company assumes no liability for any errors in this document.

Trademarks

JEUS® is registered trademark of TmaxSoft Co., Ltd.

Java, Solaris are registered trademarks of Oracle Corporation and its subsidiaries and affiliates.

Microsoft, Windows, Windows NT are registered trademarks or trademarks of Microsoft Corporation.

HP-UX is a registered trademark of Hewlett Packard Enterprise Company.

AIX is a registered trademark of International Business Machines Corporation.

UNIX is a registered trademark of X/Open Company, Ltd.

Linux is a registered trademark of Linus Torvalds.

Noto is a trademark of Google Inc. Noto fonts are open source. All Noto fonts are published under the SIL Open Font License, Version 1.1. (<https://www.google.com/get/noto/>)

Other products and company names are trademarks or registered trademarks of their respective

owners.

The names of companies, systems, and products mentioned in this manual may not necessarily be indicated with a trademark symbol (TM, ®).

Open Source Software Notice

Some modules or files of this product are subject to the terms of the following licenses: APACHE2.0, CDDL1.0, EDL1.0, OPEN SYMPHONY SOFTWARE1.1, TRILEAD-SSH2, Bouncy Castle, BSD, MIT, SIL OPEN FONT1.1

Detailed Information related to the license can be found in the following directory:
\${INSTALL_PATH}/license/oss_licenses

Document History

| Product Version | Guide Version | Date | Remarks |
|-----------------|---------------|------------|---------|
| JEUS 9.1 | 3.3.1 | 2025-12-10 | - |
| JEUS 9 | 3.1.2 | 2025-03-24 | - |
| JEUS 9 | 3.1.1 | 2024-12-24 | - |

Table of Contents

| | |
|---|----|
| 1. Introduction | 1 |
| 1.1. Basic Concepts of Jakarta Batch | 1 |
| 1.2. Jakarta Batch Components | 1 |
| 1.2.1. JobOperator | 2 |
| 1.2.2. Job | 2 |
| 1.2.3. Step | 3 |
| 1.2.4. ItemReader | 4 |
| 1.2.5. ItemWriter | 5 |
| 1.2.6. ItemProcessor | 5 |
| 1.2.7. Checkpoints | 5 |
| 2. Configuring a Batch Thread Pool | 6 |
| 2.1. Overview | 6 |
| 2.2. Configuring Batch Thread Pool DD | 6 |
| 2.2.1. Configuring Thread Pools in Component DD | 6 |
| 2.2.2. Configuring Thread Pools in Application DD | 7 |
| 2.3. Logging | 7 |
| 3. Examples of Jakarta Batch | 8 |
| 3.1. Overview | 8 |
| 3.2. Configuring Data Source | 8 |
| 3.3. Configuring jeus-web-dd.xml | 9 |
| 3.4. Configuring a Job | 9 |
| 3.5. JBatchServlet | 10 |
| 3.6. ItemReader | 11 |
| 3.7. ItemWriter | 12 |
| 3.8. ItemProcessor | 13 |

1. Introduction

This chapter outlines the concepts and components of Jakarta Batch.

1.1. Basic Concepts of Jakarta Batch

Batch processing is a processing mode: it processes a bulk of workload in the background without requiring real-time user involvement. It is used for processing jobs that take long hours, requires heavy computation, and are suitable for both parallel and sequential processes. Batch process types include ad-hoc processes, scheduled processes, and on-demand processes. It is also provided with features such as logging, checkpoint algorithms, parallel process, and supports monitoring of batch workloads so that administrators are allowed to publish, stop, or resume jobs.

JEUS supports the referential integrity checking included in the specifications and the option to specify a thread pool for a certain job.

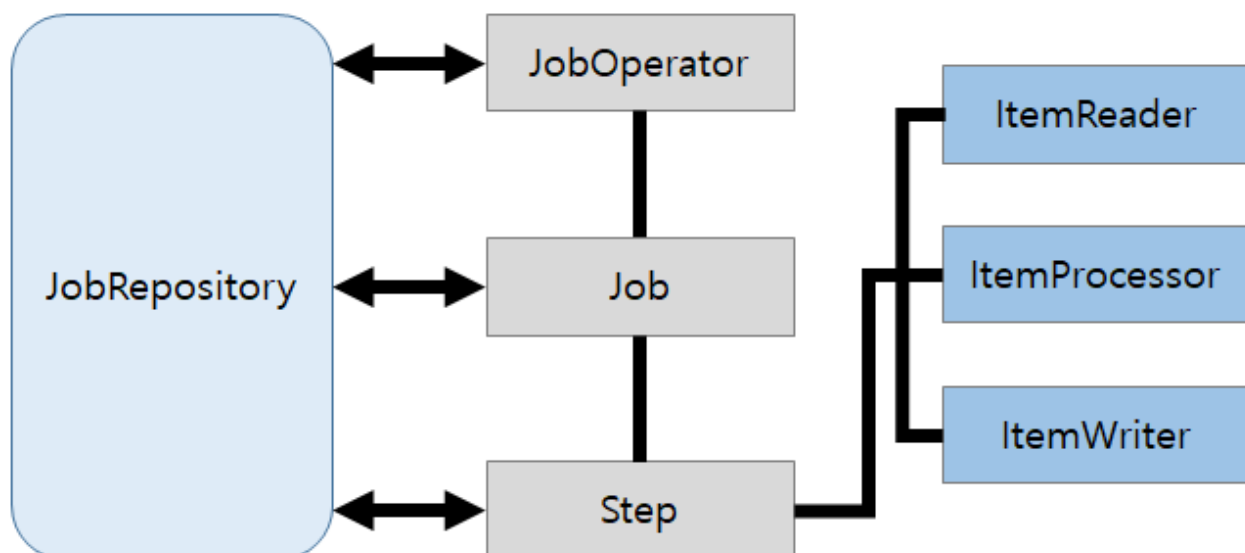


For details about Jakarta Batch, refer to the specifications.

1.2. Jakarta Batch Components

This chapter outlines the concepts and components of Jakarta Batch.

The following illustrates the structure of the core Jakarta Batch components.



Organization of the Core Jakarta Batch Components

Client applications have access to JobOperator through which they can execute and stop jobs that contain specific tasks to perform. A job contains one or more steps, which has three different processing modules of ItemReader, ItemProcessor, and ItemWriter. All of this information is stored as meta-data in JobRepository in the Jakarta Batch implementation.

1.2.1. JobOperator

JobOperator serves as the interface for client applications to execute batch jobs by providing control over all job-processing statuses. With JobOperator, you can control the start, restart, and termination of a job and to call StepExecution.

1.2.2. Job

A job is an object that encapsulates the entire process of a batch job, containing one or more steps and allowing for global configuration of properties for the steps. Configuring a job includes the name, ordering of steps, and an indication of whether the job can be restarted or not.

- JobInstance

A JobInstance is a conceptual operation of a job.

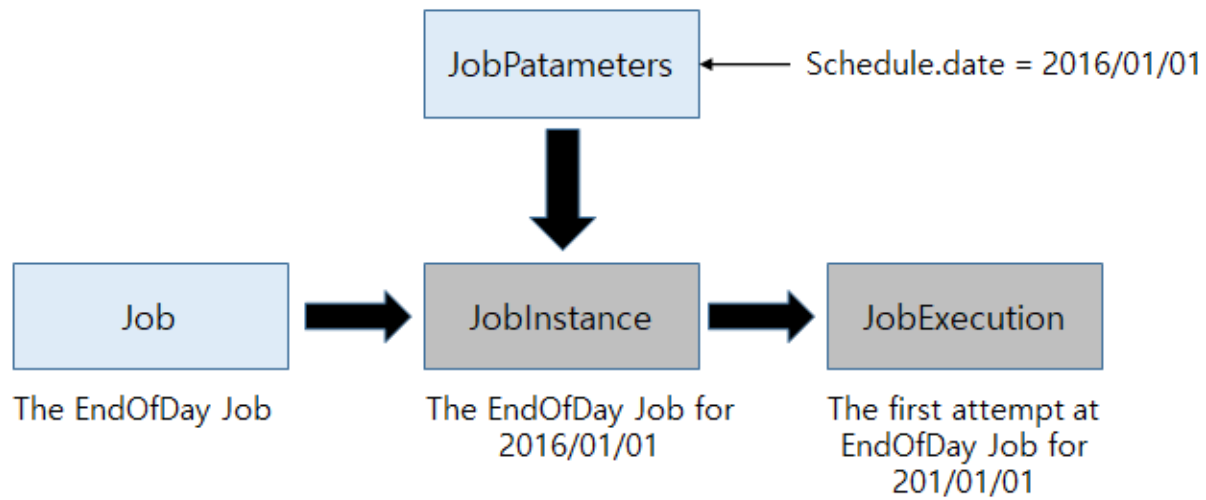
Suppose that a batch job should be run once at a certain hour every day. For this job, a JobInstance is created for one given day, and another for another given day. For example, there will be a January 1st run and a January 2nd run, which are discrete JobInstances of the given Job that runs everyday. If the January 1st run fails and is run again the next day, it is still the January 1st run. In addition, the January 2nd run is run on its own. Therefore, one job run may be run more than once, and each technical attempt to run a job is defined as a JobExecution. Using a new JobInstance indicates that a batch job will be run from scratch, and using an existing JobInstance indicates that a batch job will be run from the checkpoint.

In concept, a JobInstance has nothing to do with the data it processes. It is up to the ItemReader implementation used to load data in units of an item.

- JobParameters

A JobInstance is distinguished from another by the JobParameters it has. JobParameters are a set of `java.util.Properties` used to start a batch job. The same JobParameters may be used for two distinct JobInstances.

For example, the January 1st Jobinstance above has "`schedule.date = 2016/01/01`" as JobParameters, and the January 2nd Jobinstance has "`schedule.date = 2016/01/02`" as JobParameters. In short, a job contains one or more JobInstances, and each JobInstance is defined by the JobParameters it has and distinguished from one another. A JobInstance can have one or more job attempts called JobExecution.



Relationship among Job, JobInstance, JobParameters, and JobExecution

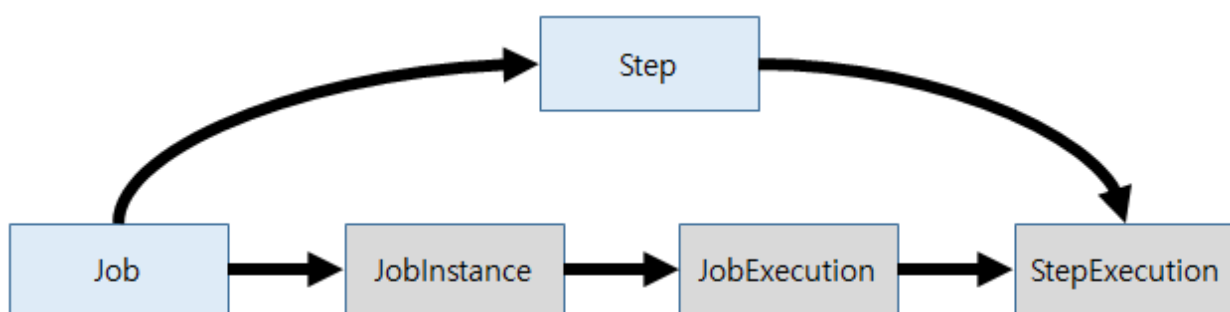
- JobExecution

A JobExecution is a technical attempt at a job. An execution could result in either success or failure, but the JobInstance of a given execution will not be considered complete until the execution is completed successfully. Let's consider the previous example of a job that runs at a certain hour everyday. The job itself defines what to do and how to do it, but it is a JobExecution that is responsible for what happens during a job run. For example, if a January 1st JobExecution fails, the January 1st JobInstance creates another JobExecution the next day to complete the job.

1.2.3. Step

A step is an object that encapsulates an independent process that is part of a series of ordered processes of a batch job. A job consists of one or more steps, and a step contains all information necessary for defining and controlling batch processes. A simple example of a step is a task of reading files into a database. Just like a job has JobExecutions, a step has one or more StepExecutions.

The following figure illustrates the one-to-many relationship between a job and a step. A job can have multiple steps, and each step can create one or more StepExecutions. That is, a JobExecution can also have one-to-many relationship with StepExecutions, meaning that a job may create many StepExecutions for job completion.



Relationship between a Job and a Step in Jakarta Batch

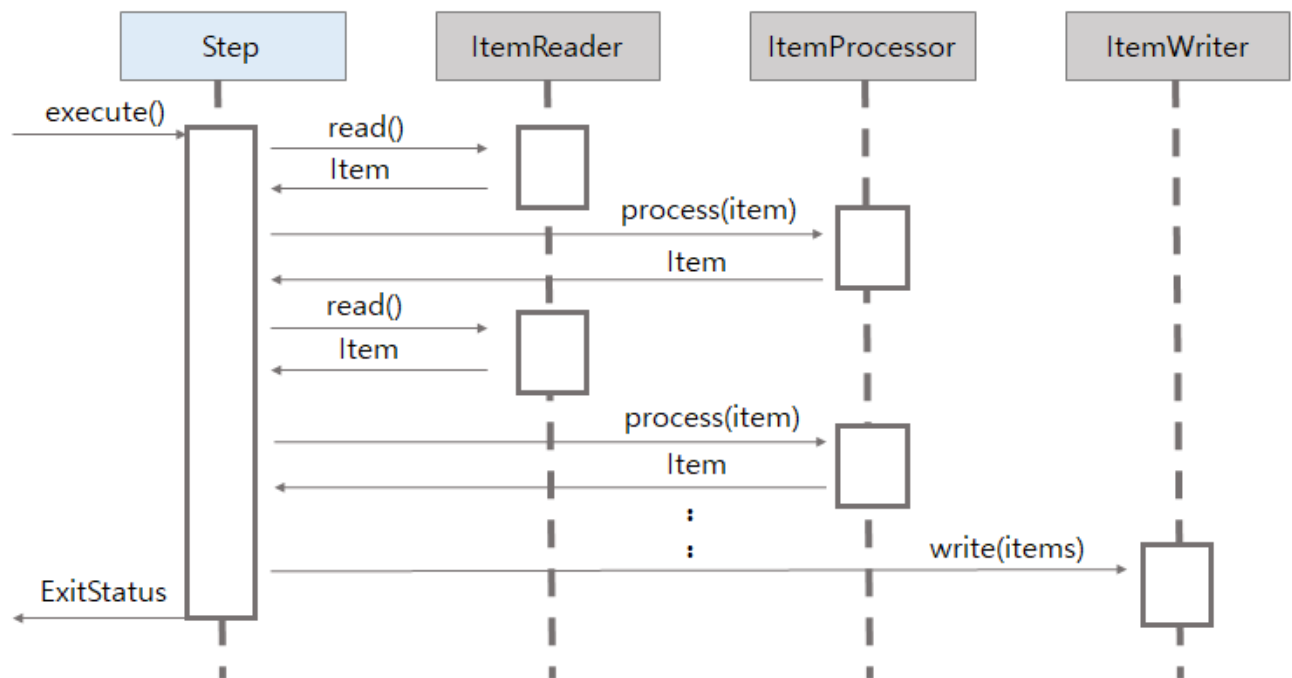
- StepExecution

A `StepExecution` is a technical attempt at a step so that running a step creates a `StepExecution`. Step types include chunk steps and batchlet steps, which are mutually exclusive and thus cannot be used together at once.

- **Chunk**

Jakarta Batch uses a chunk step to perform item-oriented processing. It can read, process, and write items it contains in chunks, and a chunk is defined as the items processed within the scope of a transaction. During a chunk step, checkpoints are taken regularly before starting a new transaction.

The following figure shows the reader-processor-writer modules of a chunk step.



Reader-Processor-Writer Modules of a Step in Jakarta Batch

When a step starts, one item is read from an `ItemReader` and handed to an `ItemProcessor`, and processed. This process is repeated until the entire of chunk of items are processed and handed to an `ItemWriter`.

- **Batchlet**

Jakarta Batch uses a batchlet step to perform task-oriented processing that an item-oriented processing may not be able to handle. For example, a batchlet step can be used to send files or to execute commands.

1.2.4. `ItemReader`

An `ItemReader` reads an item that is in the scope of a step and hands it to an `ItemProcessor`. Let's suppose that a file contains lines of data where each line serves as an independent record. In this case, a line serves as an item, which is read by an `ItemReader`.

Checkpoints are applicable to an `ItemReader`. Checkpointing allows an `ItemReader` to begin reading

an item from a checkpoint at which an item has been read successfully in case an error occurs or a job is restarted. In the case of the file containing lines of data as independent record, the number of the line that has been read last is checkpointed.

1.2.5. ItemWriter

An ItemWriter writes out the processing result of a chunk, a list of items handed from an ItemProcessor. Only the information about the items that have been handed can be written out, not the information about the other items that have not been handed yet.

1.2.6. ItemProcessor

An ItemProcessor is responsible for processing an item handed from an ItemReader and handing it to an ItemWriter. The processing logic can be implemented on an ItemProcessor.

1.2.7. Checkpoints

As batch processing applications take long hours to process massive amounts of data, checkpointing and restartability are required.

Checkpointing allows the progress status of a StepExecution to be bookmarked so that a job may begin from the last checkpoint in case it is restarted. Because checkpointing entails a lock on the checkpointed item, frequent checkpoints may have adverse effect on the system performance. Therefore, it is recommended that the checkpoints are spaced discreetly.

2. Configuring a Batch Thread Pool

This chapter describes configuration for a thread pool that execute a job defined in Jakarta Batch.

2.1. Overview

A thread pool can be configured in the Web (or EJB), or in the application whose batch-thread-pool option contains thread pool specifications and allows threads to be used regardless of priority.



If thread pools are configured in both Web DD and application DD, or in both EJB DD and application DD, they follow the **configuration in either Web DD or EJB DD**.

2.2. Configuring Batch Thread Pool DD

If batch-thread-pool is not specified in DD, the server uses the default thread pool to execute a job.

The following describes the thread pool settings.

| Item | Description |
|-----------------|--|
| min | Minimum number of threads for a thread pool to execute a job. (Default value: 0) |
| max | Maximum number of threads for a thread pool to execute a job. (Default value: 0) |
| keep-alive-time | Amount of time after which idle threads are discarded when the number of threads are between the values set for min and max. No thread is discarded if there is none in the thread pool. (Default value: 60000, Unit: ms) |
| queue-size | Size of the queue of pending requests defined in the thread pool. (Default value: 4096) |

Tuning the thread pool depends on the priority of each DD statement. This section describes each DD configuration.

2.2.1. Configuring Thread Pools in Component DD

A component refers to any one of Jakarta Batch components such as a servlet or an EJB, and it has the highest priority in case there are many thread pool configurations in the server. The thread pool

can be configured in jeus-web-dd.xml or jeus-ejb-dd.xml.

The following is an example of configuring batch-thread-pool in /WEB-INF/jeus-web-dd.xml.

Configuring Thread Pools in Component DD: <jeus-web-dd.xml>

```
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="8.0">
  <batch-thread-pool>
    <min>10</min>
    <max>20</max>
    <keep-alive-time>60000</keep-alive-time>
    <queue-size>4096</queue-size>
  </batch-thread-pool>
</jeus-web-dd>
```

The following is an example of configuring batch-thread-pool in /META-INF/jeus-ejb-dd.xml.

Configuring Thread Pools in Component DD: <jeus-ejb-dd.xml>

```
<jeus-ejb-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="8.0">
  <batch-thread-pool>
    <min>10</min>
    <max>20</max>
    <keep-alive-time>60000</keep-alive-time>
    <queue-size>4096</queue-size>
  </batch-thread-pool>
</jeus-ejb-dd>
```

2.2.2. Configuring Thread Pools in Application DD

The following is an example of configuring batch-thread-pool in /META-INF/jeus-application-dd.xml.

Configuring Thread Pools in Application DD: <jeus-application-dd.xml>

```
<application xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="8.0">
  <batch-thread-pool>
    <min>10</min>
    <max>20</max>
    <keep-alive-time>60000</keep-alive-time>
    <queue-size>4096</queue-size>
  </batch-thread-pool>
</application>
```

2.3. Logging

Jakarta Batch does not use the JEUS logger but a Java logger, which needs configuration. For information about configuring a Java logger, refer to "Property Configuration" in *JEUS Server Guide*.



The level of the logger must be configured according to the com.ibm.jbatch.

3. Examples of Jakarta Batch

This section provides an example of a Jakarta Batch application developed in JEUS.

3.1. Overview

The example provided in this section describes a simple application that executes a batch job requested from a JEUS servlet and returns the results to the client.

The following figure shows the structure of the WAR file of a Jakarta batch application.

```
Example.war
|--META-INF
|   |--MANIFEST.MF
|--WEB-INF
|   |--[X]jeus-web-dd.xml
|   |--classes
|       |--META-INF
|           |--batch-jobs
|               |--[X]simple_file_batch.xml
|           |--task
|               |--[T]personList.txt
|           |--tmaxsoft
|               |--jbatch
|                   |--test
|                       |--[C]jBatchServlet.class
|                       |--handler
|                           |--[C]FileItemReader.class
|                           |--[C]LogWriter.class
|                           |--[C]Person.class
|                           |--[C]StringToPersonProcessor.class

* Legend
- [01]: binary or executable file
- [X] : XML document
- [J] : JAR file
- [T] : Text file
- [C] : Class file
- [V] : java source file
- [DD] : deployment descriptor
```

3.2. Configuring Data Source

Before running a Jakarta Batch application on JEUS, you should define a data source used to execute Jakarta Batch jobs.

In Linux, you can run derby by executing JEUS_HOME/bin/startderby. Specify the export name of the data source as "jdbc/batch".



Only Apache Derby is supported now. Other databases have not been verified yet, and thus are not recommended for use.

The following is an example of a data source defined in the domain.xml file.

Configuring a DataSource: <domain.xml>

```
<domain>
  <data-source>
    ...
    <database>
      <data-source-id>jdbc/batch</data-source-id>
      <data-source-class-name>
        org.apache.derby.jdbc.ClientConnectionPoolDataSource
      </data-source-class-name>
      <data-source-type>ConnectionPoolDataSource</data-source-type>
      <server-name>localhost</server-name>
      <port-number>1527</port-number>
      <database-name>derbyDB</database-name>
      <user>app</user>
      <password>app</password>
      <property>
        <name>ConnectionAttributes</name>
        <type>java.lang.String</type>
        <value>;create=true</value>
      </property>
    </database>
  </data-source>
</domain>
```

3.3. Configuring jeus-web-dd.xml

A thread pool can be configured with the batch-thread-pool element in jeus-web-dd to execute a job. If it is not specified, it is set to the default thread pool. For details, refer to [Component DD Configuration](#).

3.4. Configuring a Job

Job specifications are defined in an XML file including the name, steps, ItemReader, ItemWriter, and ItemProcessor. The XML file should be located in the META-INF/batch-jobs/ directory.



For a WAR file, a META-INF directory should be placed in the /WEB-INF/classes directory for successful operation.

The following is an example of configuring the /WEB-INF/classes/META-INF/batch-jobs/simple_file_batch.xml file.

Configuring a Job: <simple_file_batch.xml>

```
<job id="demo" xmlns="http://xmlns.jcp.org/xml/ns/jakartaee" version="1.0">
  <step id="step1">
    <chunk>
      <reader ref="tmaxsoft.bhkim.test.filebatch.FileItemReader">
        <properties>
          <property name="filePath" value="#{jobParameters['filePath']}" />
        </properties>
      </reader>
      <processor ref="tmaxsoft.bhkim.test.filebatch.StringToPersonProcessor" />
      <writer ref="tmaxsoft.bhkim.test.filebatch.LogWriter" />
    </chunk>
  </step>
</job>
```

3.5. JBatchServlet

The JBatchServlet hands the job configuration file (.xml) as well as the JobParameters to the JobOperator.

The following is an example of passing the path to the personList.txt file that contains data used for processing a filePath into a key.

JBatchServlet Example

```
@WebServlet("/JBatchServlet")
public class JBatchServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private final String FILE_PATH = "META-INF/task/personList.txt";

    public JBatchServlet() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        Properties jobParams = new Properties();
        URL personListURL = Thread.currentThread().getContextClassLoader().getResource(FILE_PATH);
        jobParams.setProperty("filePath", personListURL.getPath());

        JobOperator operator = BatchRuntime.getJobOperator();
        // xml file name without format
        long id = operator.start("simple_file_batch", jobParams);

        waitForEnd(operator, id);

        response.getWriter().println("File Batch is successfully precessed.");
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```



```

public static void waitForEnd(final JobOperator jobOperator, final long id) {
    final Collection<BatchStatus> endStatuses = Arrays.asList(BatchStatus.COMPLETED,
BatchStatus.FAILED);
    do {
        try {
            Thread.sleep(100);
        } catch (final InterruptedException e) {
            return;
        }
    } while (!endStatuses.contains(jobOperator.getJobExecution(id).getBatchStatus()));
}
}

```

FILE_PATH contains the following definitions:

```

Name1,27,Computer Science
Name2,22,English Education
Name3,23,Electronic Engineering

```

3.6. ItemReader

An ItemReader reads an item in. Below is an example of FileItemReader, an ItemReader with four functions—open, close, readItem, and checkpointInfo. In the following example, the ItemReader reads a line of data as an item.

Serializable, a parameter variable of the open function, verifies a checkpoint so that when a job is abnormally terminated, an ItemReader can begin at the last checkpoint.



Injectors may not work for pure batch logic except when made into a servlet or an EJB.

The following is an example of a FileItemReader with the open function implemented to bookmark a recordNumber after every readItem so that the FileItemReader can begin from the last recordNumber in case the job has been abnormally terminated.

FileItemReader Example

```

public class FileItemReader implements ItemReader {

    @Inject
    @BatchProperty
    private String filePath; // JobProperties in xml setting is injected to the field

    private BufferedReader reader = null;

    private int recordNumber;

    @Override
    public void open(Serializable checkpoint) throws Exception {
        if (filePath == null)

```



```

        throw new RuntimeException("Can't find any input");

        final File file = new File(filePath);
        if (!file.exists())
            throw new RuntimeException("A file '" + filePath + "' doesn't exist");

        reader = new BufferedReader(new FileReader(file));

        if (checkpoint != null) {
            assert (checkpoint instanceof Integer);
            recordNumber = (Integer) checkpoint;

            // Pass over latest checkpoint record
            for (int i = 1; i < recordNumber; i++)
                reader.readLine();
        }

        @Override
        public void close() throws Exception {
            if (reader != null)
                reader.close();
        }

        @Override
        public Object readItem() throws Exception {
            Object item = reader.readLine();

            // checkpoint line update
            recordNumber++;
            return item;
        }

        @Override
        public Serializable checkpointInfo() throws Exception {
            return recordNumber;
        }
    }
}

```

3.7. ItemWriter

The following is an example of a `LogWriter`, which is an `ItemWriter` with four functions—open, close, writeItems, and checkpointInfo. The list of items to be processed is received by writeItems.



Injectors may not work for pure batch logic except when made into a servlet or an EJB.

The following is an example of using an internally defined logger to write out a list of items received.

ItemWriter Example

```

public class LogWriter implements ItemWriter {

```



```

private static final Logger logger = Logger.getLogger(LogWriter.class.getSimpleName());
int writeRecordNumber;

@Override
public void open(Serializable checkpoint) throws Exception {
}

@Override
public void close() throws Exception {
}

@Override
public void writeItems(List<Object> items) throws Exception {
    // simply print passed item to logger
    for (Object o: items) {
        logger.info("writeItems > " + o.toString());
    }
}

@Override
public Serializable checkpointInfo() throws Exception {
    return null;
}
}

```

3.8. ItemProcessor

An ItemProcessor processes items that have been read in by an ItemReader.



Injectors may not work for pure batch logic except when made into a servlet or an EJB.

The following is an example of an ItemProcessor that receives as item parameter each line of a file and processes the character strings into the form of a Person object. The objects are passed to an ItemWriter after a phase.

ItemProcessor Example

```

public class StringToPersonProcessor implements ItemProcessor {

    @Override
    public Object processItem(Object item) throws Exception {
        // Items passed as parameters are arrayed like "name, age, department.
        final String[] line = String.class.cast(item).split(",");

        if (line == null || line.length != 3)
            return null;

        return new Person(line[0], Integer.parseInt(line[1]), line[2]);
    }
}

```


The following is the definition of the Person object to use in the ItemProcessor above.

Person Object

```
public class Person {
    private String name;
    private int age;
    private String department;

    public Person(String name, int age, String department) {
        this.name = name;
        this.age = age;
        this.department = department;
    }

    @Override
    public String toString() {
        return name + "," + age + "," + department;
    }
}
```