Community Experience Distilled

Spring Batch Essentials

Design, develop, and deliver robust batch applications with the power of the Spring Batch framework

P. Raja Malleswara Rao

[PACKT] PUBLISHING open source* community experience distilled

Spring Batch Essentials

Design, develop, and deliver robust batch applications with the power of the Spring Batch framework

P. Raja Malleswara Rao

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Spring Batch Essentials

First published: January 2015

Production reference: 1210115

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78355-337-2

`www.packtpub.com`

Credits

Author
P. Raja Malleswara Rao

Reviewers
Alexey Grigorev
Mykola Kolisnyk
Luca Masini
Anirudh Prabhu

Commissioning Editor
Pramila Balan

Acquisition Editor
Nikhil Karkal

Content Development Editor
Sharvari Tawde

Technical Editors
Manal Pednekar
Faisal Siddiqui

Copy Editors
Shambavi Pai
Adithi Shetty

Project Coordinator
Aboli Ambardekar

Proofreaders
Simran Bhogal
Cathy Cumberlidge

Indexer
Monica Ajmera Mehta

Graphics
Sheetal Aute
Valentina D'silva
Abhinash Sahu

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

P. Raja Malleswara Rao is a senior consultant, focusing on enterprise architecture and development of Java-related technologies. He is a certified Java and web components developer with deep expertise in building enterprise applications using diverse frameworks and methodologies. He is an active participant in technical forums, groups, and conferences. He has worked with several Fortune 500 organizations and is passionate about learning new technologies and their developments.

I would like to thank my family and friends for their love and support, especially my wife, Bhargavi, for sacrificing the quality time I could have spent with her and encouraging me through this stint.

I would also like to thank my reviewers for their valuable suggestions in improving the quality of this book, my colleagues, and the Spring community for sharing great thoughts that helped me a lot in keeping myself updated.

About the Reviewers

Alexey Grigorev is an experienced software engineer with a background in Java and Spring, as well as in data transformation and integration. He is interested in machine learning and Big Data analysis, and he likes to build smart, robust, and scalable applications. He currently works as a freelancer and can be reached at `alexey.s.grigoriev@gmail.com`.

I would like to thank my wife, Larisa Kosareva, for her support.

Mykola Kolisnyk has been part of test automation since 2004, being involved in various activities, including creating test automation solutions from the scratch, leading the test automation team, and performing consultancy regarding test automation processes. During his working career, he has had experience of using different test automation tools, such as Mercury WinRunner, Micro Focus SilkTest, SmartBear TestComplete, Selenium RC, WebDriver, SoapUI, BDD frameworks, and many other different engines and solutions. Mykola has experience of working with multiple programming technologies based on Java, C#, Ruby, and so on. He also has experience of different domain areas such as healthcare, mobile, telecom, social networking, business process modeling, performance and talent management, multimedia, e-commerce, and investment banking.

He was a permanent employee at ISD, GlobalLogic, and Luxoft, and he also has experience in freelancing activities. He was invited as an independent consultant to introduce test automation approaches and practices to external companies.

He currently works in DevOps. He's one of the authors (along with Gennadiy Alpaev) of the online SilkTest Manual (`http://silktutorial.ru/`) and has participated in the creation of the TestComplete tutorial (`http://tctutorial.ru/`), which is one of the biggest related documentation available in ru-Net.

Also, he was the reviewer of *TestComplete Cookbook*, *Packt Publishing*.

Luca Masini is a senior software engineer and architect. Born as a game developer for Commodore 64 (Football Manager) and Commodore Amiga (Ken il guerriero), he soon converted to object-oriented programming and he was attracted by the Java language from its beginning in 1995.

He worked on this passion as a consultant for major Italian banks, developing and integrating the main software projects for which he has often taken the technical leadership. He adopted Java Enterprise in environments where COBOL was the flagship platform, converting the environments from mainframe centric to distributed types.

He then shifted his focus towards open source, starting with Linux to enterprise frameworks with which he was able to introduce concepts such as IoC, ORM, and MVC with low impact. He was an early adopter of Spring, Hibernate, Struts, and a whole host of other technologies that in the long run have given his customers a technological advantage and therefore development costs cuts.

After introducing this new technology, he decided that it was time for simplification and standardization of development with Java EE and, hence, he's now working at the ICT of a large Italian company where he introduced build tools (Maven and Continuous Integration), archetypes of project, and *Agile Development* with plain standards.

Finally, he focused his attention towards mobilizing the enterprise and is now working on a whole set of standard and development processes to introduce mobile concepts and applications for sales force and management.

He has worked on the following books by Packt Publishing:

- *Securing WebLogic Server 12c*
- *Google Web Toolkit GWT Java AJAX Programming*
- *Spring Web Flow 2 Web Development*
- *Spring Persistence with Hibernate*

Anirudh Prabhu is a software engineer with over 5 years of industry experience. He specializes in technologies such as HTML5, CSS3, PHP, jQuery, Twitter Bootstrap, and SASS, and also has knowledge of CoffeeScript and AngularJS.

In addition to web development, he has been involved in building training material and writing tutorials for twenty19 (`http://www.twenty19.com/`) for the technologies mentioned.

Besides Packt Publishing, he has been associated with Apress and Manning Publication as a technical reviewer for several titles.

Table of Contents

Preface

Welcome to the world of Spring Batch! We're pleased that you have chosen our book, fully dedicated to the essentials of Spring Batch 3.0.2 release. Before we get started with this book, I would like to give an overview of how this book is organized and how you can get the most out of it. Once you have completed reading this book, you should be familiar with key batch processing concepts and understand how to solve the majority of the real-world problems that you will need to solve with Spring Batch. This book gives an insight of the essential concepts and applications of the Spring Batch framework, which will allow you to handle any unexpected use cases the book does not cover.

Spring Batch is an open source, lightweight, and comprehensive solution, designed to enable the development of robust batch applications, vital for enterprise operations.

This tutorial starts with an insight into batch applications and Spring Batch offerings. Understanding the architecture and key components, you will learn how to develop and execute a batch application. You can then walk through the essential configurations and execution techniques, key technical implementations of the read and write, and processing features of different forms of data. Next, you will move onto the key features, such as transaction management, job flows, job monitoring, and data sharing across the steps of the executing jobs. Finally, you will learn how Spring Batch can integrate with diverse enterprise technologies and facilitate optimization and performance improvement with scaling and partitioning techniques.

This book uses a simple application based on Spring Batch to illustrate how to solve real-world problems. The application is intended to be very simple and straightforward, and purposely contains very little functionality—the goal of this application is to encourage you to focus on the Spring Batch concepts, and not get tied up in the complexities of application development. You will have a much easier time following the book if you take the time to review the sample application source code. Some tips on getting started are found in the appendix.

What this book covers

Chapter 1, *Spring Batch Fundamentals*, introduces you to batch applications and details about Spring Batch and its offerings. It discusses Spring Batch architecture, and also demonstrates how to design and execute a Spring Batch job.

Chapter 2, *Getting Started with Spring Batch Jobs*, covers the Spring Batch XML features, how to configure jobs, transactions and repository, working with EL and listeners, and executing jobs from command line and web application schedulers.

Chapter 3, *Working with Data*, demonstrates the essential data handling mechanisms, including reading the data from different sources such as flat files, XML and databases, processing the data, and writing the data to different destinations. It also explains about transforming and validating the data in the processing data phase.

Chapter 4, *Handling Job Transactions*, covers transactions and Spring Batch transaction management, and explains how to customize the transaction along with transaction patterns.

Chapter 5, *Step Execution*, explains the techniques to control the job flow, sharing the data between steps in execution and process reuse with externalization. It also demonstrates terminating the batch job in different states and their importance.

Chapter 6, *Integrating Spring Batch*, covers the integration techniques for enterprise applications based on Spring Batch, and some of the job launching and process techniques.

Chapter 7, *Inspecting Spring Batch Jobs*, demonstrates the monitoring techniques of Spring Batch jobs, access methods to execute job data, monitoring and reporting with listeners and web monitoring techniques.

Chapter 8, *Scaling with Spring Batch*, discusses the performance and scaling concerns with Spring Batch support for configuration-based tuning techniques such as parallel processing, remote chunking, and partitioning.

Chapter 9, *Testing the Spring Batch*, covers details of testing techniques for Spring Batch applications with the help of open source frameworks and Spring support for unit testing, integration testing, and functional testing.

Appendix covers the setup references for Java, Eclipse IDE, a project with dependencies, and an administration project.

What you need for this book

The following list provides the required software in order to run the sample applications included with this book. Some chapters have additional requirements that are outlined within the chapter itself:

- Java Development Kit 1.6+ can be downloaded from Oracle's website at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- Eclipse Java EE IDE for Web Developers can be downloaded from Eclipse's website at `https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplersr2`

Who this book is for

This book is for a Java developer with some experience in development of enterprise applications, who wants to learn about batch application development with Spring Batch, to build simple yet powerful batch applications on a Java-based platform.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The ChunkProvider is the interface returning chunks from `ItemReader`."

A block of code is set as follows:

```
public interface ChunkProvider<T> {
void postProcess(StepContribution contribution, Chunk<T> chunk);
Chunk<T> provide(StepContribution contribution) throws Exception;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<step id="initialStep">
<partition step="stepPartition" handler="handler"/>
</step>
<bean class="org.springframework.batch.core.partition.support.
TaskExecutorPartitionHandler">
<property name="taskExecutor" ref="taskExecutor"/>
<property name="step" ref="stepPartition"/>
<property name="gridSize" value="10"/>
</bean>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The Maven software needs to be integrated into the Eclipse IDE as mentioned in the instructions at `https://www.eclipse.org/m2e/`".

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Spring Batch Fundamentals

Organizations need to process huge volumes of data through a series of transactions in their day-to-day operations. These business operations should be automated to process the information efficiently without human intervention. Batch processing can execute such a series of operations through programs, with a predefined set of data groups as input, process the data, and generate a set of output data groups and/or update the database.

In this chapter, we will cover the following topics:

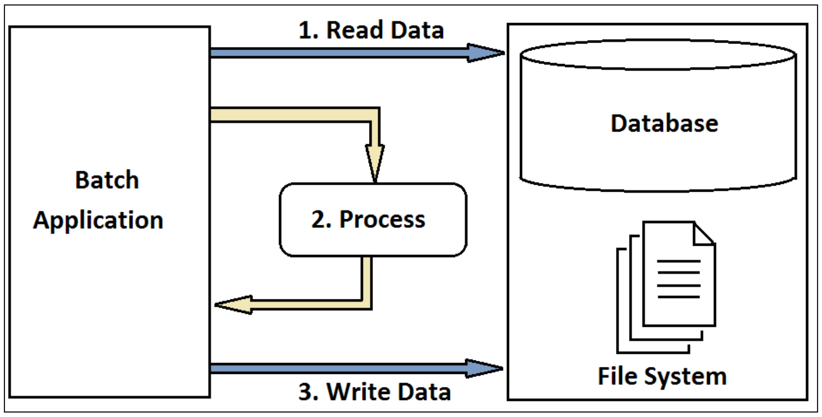
- Introduction to batch applications
- Spring Batch and its offerings
- Spring Batch infrastructure
- Job design and executions

Introduction to batch applications

Organizations need to accomplish diverse business operations that include a large amount of data processing. Following are some examples of such operations:

- Generation of salary slips and tax calculations in a large enterprise
- Credit card bill generation by banks
- Fresh stock updated by retail stores in their catalog

All such operations are executed with a predefined set of configurations and schedules, to run at a particular offload system time. Batch applications should be able to process large volumes of data without human intervention. The following figure represents a typical batch application:



A standard batch application is expected to have the following capabilities:

- **Scalable**: It should be able to process billions of records and be reliable without crashing the application
- **Robust**: It should be intelligent enough to identify the invalid data and keep track of such mishaps to rerun with corrected data
- **Dynamic**: It should interact with different systems to access the data using the credentials provided and process the operations
- **Concurrent**: It must process multiple jobs in parallel with the shared resources
- **Systematic**: It should process the workflow-driven batches in a sequence of dependent steps
- **High performance**: It must complete the processing in a specified batch window

Spring Batch and its offerings

Spring Batch is a lightweight, comprehensive batch framework designed to enable the development of robust batch applications that are vital for the daily operations of enterprise systems developed by SpringSource and Accenture in collaboration.

Spring Batch follows POJO-based development to let developers easily implement batch processing and integrate with other enterprise systems when needed.

Plain Old Java Object (**POJO**) represents an ordinary Java object that can be used to store a data item and exchange information between services easily.

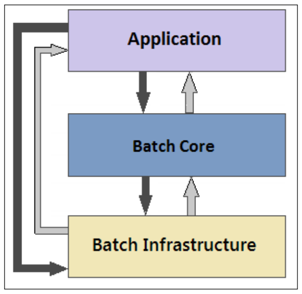
While Spring Batch provides many reusable functions adopted from the Spring framework and customized for batch applications to perform common batch (such as split processing of huge volumes of data, logging, transaction management, job process-skip-restart, and effective resource management), it is not a scheduling framework. Spring Batch can work in conjunction with a scheduler (such as Quartz/Control-M), but cannot replace a scheduler.

We discussed the capabilities expected from a standard batch application in the previous section. Spring Batch is designed to fulfill the expected features, along with its high capability, to integrate with different applications developed in other frameworks. Let's observe some of the important features offered by Spring Batch:

- Support for multiple file formats, including fixed length, delimited files, XML and common database access using JDBC, and other prominent frameworks
- Automatic retry after failure
- Job control language to monitor and perform common operations such as job start, stop, suspend, and cancel
- Tracking status and statistics during the batch execution and after completing the batch processing
- Support for multiple ways of launching the batch job, including script, HTTP, and message
- Support to run concurrent jobs
- Support for services such as logging, resource management, skip, and restarting the processing

Spring Batch infrastructure

Spring Batch is designed with a layered architecture, including three major components, namely, Application, Core, and Infrastructure, as shown in the following figure:



The Application layer contains the developer-written code to run the batch jobs using Spring Batch.

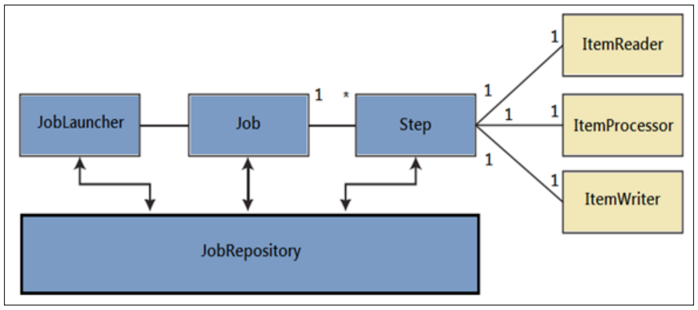
The Batch Core layer contains the core runtime classes such as `JobLauncher`, `Job`, and `Step`, necessary to launch and control the batch job. This layer interacts with the Application layer and Batch Infrastructure layer to run the batch jobs.

The Batch Infrastructure layer contains the common readers, writers, and services. Both Application and Batch Core are built on top of Infrastructure. They refer to Infrastructure for the information required to run the batch jobs.

Multiple components are involved in Spring Batch job execution. The components and their relationship are discussed in the next section.

Spring Batch components

The following figure represents the Spring Batch job components and the relationship between these components:



`JobLauncher` is the interface responsible for beginning a job. When a job is first launched, `JobLauncher` verifies in the `JobRepository`, if the job is already executed and the validity of the `Job` parameter before executing the job.

A job is the actual batch process to be executed. A `Job` parameter can be configured in an XML or a Java program.

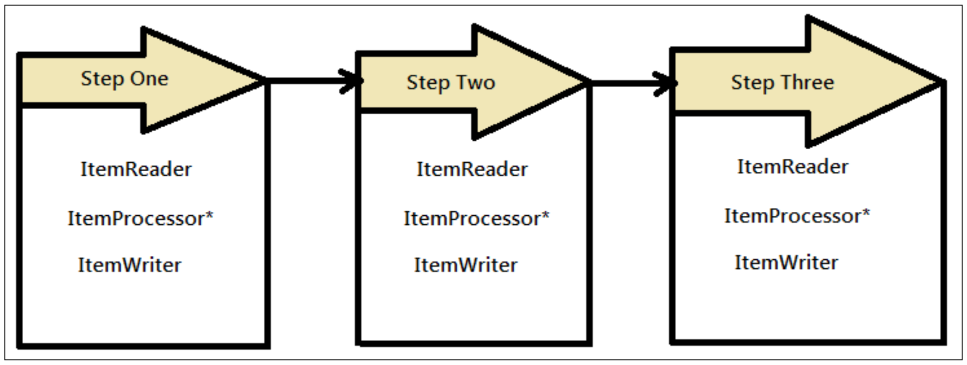
`JobInstance` is the logical instance of the job per cycle. If a `JobInstance` execution fails, the same `JobInstance` can be executed again. Hence, each `JobInstance` can have multiple job executions.

`JobExecution` is the representation of single run of a job. `JobExecution` contains the run information of the job in execution, such as `status`, `startTime`, `endTime`, `failureExceptions`, and so on.

`JobParameters` are the set of parameters used for a batch job.

A `Step` is a sequential phase of a batch job. `Step` contains the definition and control information of a batch job. The following figure represents multiple steps in a batch job. Each `Step` constitutes three activities, namely, data reading, processing, and writing, which are taken care of by `ItemReader`, `ItemProcessor`, and `ItemWriter` respectively. Each record is read, processed (optional), and written to the system.

`StepExecution` is the representation of a single run of a `Step`. `StepExecution` contains the run information of the step, such as `status`, `startTime`, `endTime`, `readCount`, `writeCount`, `commitCount`, and so on.



`JobRepository` provides **create, retrieve, update, and delete** (**CRUD**) operations for the `JobLauncher`, `Job`, and `Step` implementations.

`ItemReader` is the abstract representation of the retrieval operation of `Step`. `ItemReader` reads one item at a time.

`ItemProcessor` is the abstract representation of the business processing of the item read by `ItemReader`. `ItemProcessor` processes valid items only and returns `null` if the item is invalid.

`ItemWriter` is the abstract representation of the output operation of `Step`. `ItemWriter` writes one batch or chunk of items at a time.

In the next section, we will use our understanding of these components and develop a simple batch application using the essential Spring Batch job components. Also included are the code snippets of this application in steps.

Job design and executions

Spring Batch can be configured in your project in multiple ways, by including downloaded ZIP distribution and checking out from Git or configure using Maven. In our case, we will use the Maven configuration. You should have Maven installed in your system directly or using an IDE-based plugin (we are using Eclipse in this example). Refer to `https://www.eclipse.org/m2e/` to integrate Maven in your Eclipse IDE. The latest versions of Eclipse come with this plugin installed; verify this before installing.

A Spring Batch job can be launched in multiple ways, including the following:

- Launching the job from the command line
- Launching the job using job schedulers
- Launching the job from a Java program
- Launching the job from a web application

For this sample program, we are launching the batch job from a simple Java program.

The following are the steps, with code snippets, to run the first batch job using Spring Batch:

1. Create a Maven-enabled Java project (let's call it `SpringBatch`). Maven is the software to manage the projects effectively. The `pom.xml` file is the configuration file for Maven to include any API dependencies. There are dedicated Maven archetypes that can create sample projects. The location for Maven is `http://mvnrepository.com/artifact/org.springframework.batch/spring-batch-archetypes`.
2. Configure `pom.xml` in the `root` directory of your project to have the required Maven dependencies that include the following:
    - Spring framework with batch

- `log4j` for logging
- JUnit to test the application
- Commons Lang helper utilities for the `java.lang` API
- **HyperSQL Database** (**HSQLDB**) to be able to run using HSQLDB, which is a relational database management system written in Java

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>batch</groupId>
    <artifactId>SpringBatch</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  <properties>
    <spring.framework.version>3.2.1.RELEASE
    </spring.framework.version>
    <spring.batch.version>3.0.2.RELEASE
    </spring.batch.version>
  </properties>


<dependencies>
    <dependency>
      <groupId>commons-lang</groupId>
      <artifactId>commons-lang</artifactId>
      <version>2.6</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.batch</groupId>
      <artifactId>spring-batch-core</artifactId>
      <version>${spring.batch.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.batch</groupId>
      <artifactId>spring-batch-infrastructure</artifactId>
      <version>${spring.batch.version}</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>
    <dependency>
```

```
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.8.2</version>
            <scope>test</scope>
          </dependency>
          <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>${spring.framework.version}</version>
          </dependency>
          <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring.framework.version}</version>
          </dependency>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.7</version>
          </dependency>
      </dependencies>
      </project>
```

3. Create `log4j.xml` under the `src\main\resources` directory to log with the following content, which will produce a formatted console output:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/
log4j/">

  <appender name="CONSOLE"
   class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out"/>
    <param name="Threshold" value="INFO" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d %-5p %c -
      %m%n"/>
    </layout>
  </appender>
  <logger name="org.springframework" additivity="false">
    <level value="INFO"/>
    <appender-ref ref="CONSOLE"/>
  </logger>
  <root>
```

```
        <level value="DEBUG"/>
        <appender-ref ref="CONSOLE"/>
    </root>
</log4j:configuration>
```

4. Include the configuration file (`context.xml`) under the `src\main\resources\batch` directory with the following content. Context configuration includes the `jobRepository`, `jobLauncher`, and `transactionManager` configuration. We configured the batch as the default schema in this configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/batch"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch-3.0.xsd">
    <beans:bean id="jobRepository"
    class="org.springframework.batch.core.repository.
    support.MapJobRepositoryFactoryBean">
    <beans:property name="transactionManager"
     ref="transactionManager"/>
    </beans:bean>
    <beans:bean id="jobLauncher"
      class="org.springframework.batch.core.launch.support.
      SimpleJobLauncher">
      <beans:property name="jobRepository"
      ref="jobRepository" />
    </beans:bean>

    <beans:bean id="transactionManager"
      class="org.springframework.batch.support.transaction.
      ResourcelessTransactionManager"/>
    </beans:beans>
```

5. Include the job config (`firstBatch.xml`) under the `src\main\resources\batch` directory with the following content. Batch job configuration includes configuring the batch job with step and tasklet, using a Java program.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns ="http://www.springframework.org/schema/batch"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
```

```
http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch-3.0.xsd">
  <beans:import resource="context.xml" />
  <beans:bean id="firstBatch" class=" batch.FirstBatch"/>
    <step id="firstBatchStepOne">
      <tasklet ref="firstBatch"/>
    </step>
    <job id="firstBatchJob">
    <step id="stepOne" parent="firstBatchStepOne"/>
  </job>
</beans:beans>
```

6. Write the tasklet (the strategy for processing in a step) for the first job (`FirstBatch.java`) under the `src\main\java\batch` directory with the following content. This tasklet program is referred to in the `firstBatch.xml` configuration for tasklet reference under `Job`.

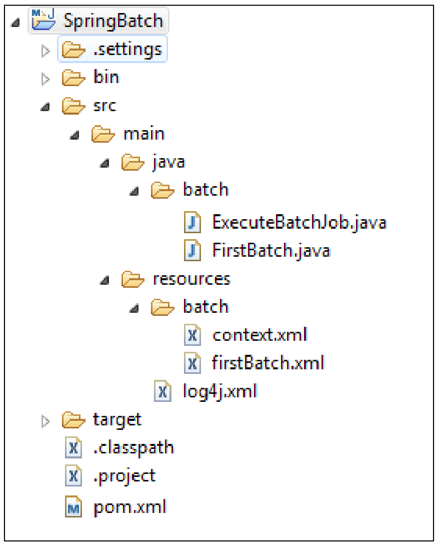
```
package batch;

import org.apache.log4j.Logger;
import org.springframework.batch.core.StepContribution;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;

public class FirstBatch implements Tasklet {
  static Logger logger = Logger.getLogger("FirstBatch");

  public RepeatStatus execute(StepContribution arg0,
  ChunkContext arg1)
      throws Exception {
    logger.info("** First Batch Job is Executing! **");
    return RepeatStatus.FINISHED;
  }
}
```

7. Write the Java program to execute the batch job (`ExecuteBatchJob.java`) under the `src\main\java\batch` directory with the following content. Through this program, we access the job configuration file and identify the `JobLauncher` and `Job` beans from the configuration files. `JobExecution` is invoked from the `run` method of `JobLauncher` by passing the job and `jobParameters`.

As mentioned earlier, we can run a batch job from either of the options, including command line, job schedulers, web application, or a simple Java program. We are using a simple Java program here to run our first job.

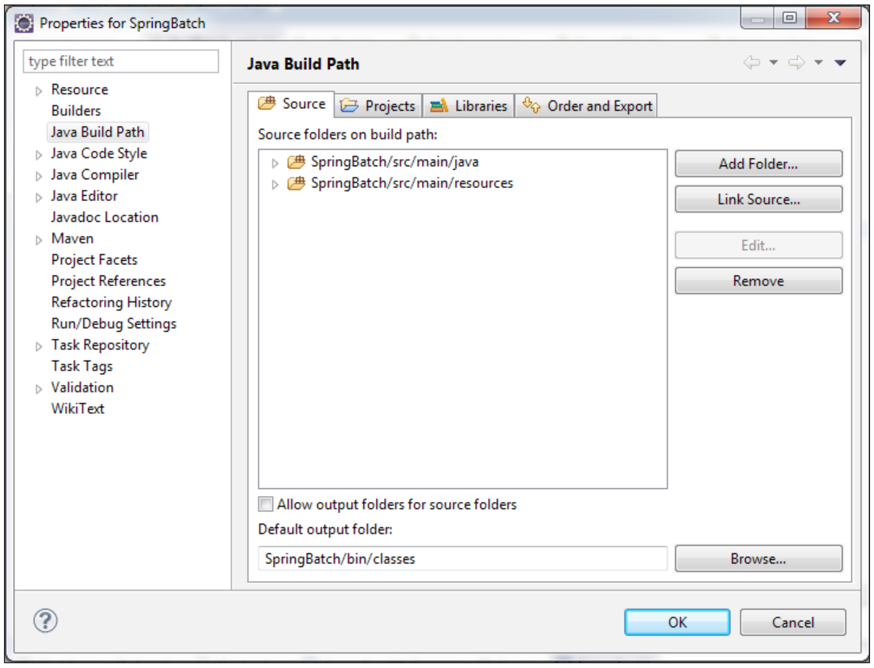
```
package batch;

import org.apache.log4j.Logger;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

public class ExecuteBatchJob {

  static Logger logger =
  Logger.getLogger("ExecuteBatchJob");
  public static void main(String[] args) {

    String[] springConfig  = {"batch/firstBatch.xml"};
    ApplicationContext context = new
    ClassPathXmlApplicationContext(springConfig);

    JobLauncher jobLauncher = (JobLauncher)
    context.getBean("jobLauncher");
    Job job = (Job) context.getBean("firstBatchJob");
    try {
      JobExecution execution = jobLauncher.run(job, new
      JobParameters());
      logger.info("Exit Status : " +
      execution.getStatus());
      } catch (Exception e) {
          e.printStackTrace();
      } finally {
        if (context != null) {
          context = null;
        }
      }
    logger.info("Done");
  }
}
```

8. Following is the folder structure to be generated in the `SpringBatch` project, after including the resources mentioned earlier:



```
SpringBatch
  .settings
  bin
  src
    main
      java
        batch
          ExecuteBatchJob.java
          FirstBatch.java
      resources
        batch
          context.xml
          firstBatch.xml
        log4j.xml
  target
  .classpath
  .project
  pom.xml
```

Add `src/main/java and src/main/resources` to the project source through build path properties, as shown in the following screenshot:



Properties for SpringBatch

Java Build Path

Source folders on build path:

SpringBatch/src/main/java
SpringBatch/src/main/resources

Allow output folders for source folders

Default output folder:

SpringBatch/bin/classes

9. Build the project with the Maven installation and run the `ExecuteBatchJob` Java program to get the batch job execution status printed on the console:

```
2014-06-01 17:02:29,548 INFO  org.springframework.batch.
core.launch.support.SimpleJobLauncher - Job: [FlowJob:
[name=firstBatchJob]] launched with the following parameters: [{}]
2014-06-01 17:02:29,594 INFO  org.springframework.batch.core.job.
SimpleStepHandler - Executing step: [stepOne]
2014-06-01 17:02:29,599 INFO ** First Batch Job is Executing! **
2014-06-01 17:02:29,633 INFO  org.springframework.batch.
core.launch.support.SimpleJobLauncher - Job: [FlowJob:
[name=firstBatchJob]] completed with the following parameters:
[{}] and the following status: [COMPLETED]
2014-06-01 17:02:29,637 INFO Exit Status :COMPLETED
2014-06-01 17:02:29,639 INFO Done
```

Following the previously mentioned steps, we configured our first batch job using Spring Batch and executed it successfully from a Java program.

Summary

Throughout this chapter, we learned about batch applications, real-time batch applications, and the capabilities expected from a standard batch application. We also learned about Spring Batch applications and the features offered by the Spring Batch technology, high-level Spring Batch architecture, and components involved in Spring Batch job execution, along with the relationships among those components. We completed this chapter with the development of a simple batch application and ran the program successfully.

In the next chapter, we will learn about the configuration of batch jobs using XML and EL, and the execution of batch jobs from the command line and application. We will also discuss the scheduling of batch jobs.

2
Getting Started with Spring Batch Jobs

In the previous chapter, we learned about batch applications, the offerings and architecture of Spring Batch, and how to build a Spring Batch application to run a batch job. It is important to understand the details of a framework and its components to be able to effectively configure them for business needs. XML- and annotation-based configurations have made programming more efficient and flexible with Spring Batch.

Some applications expect the configuration to be flexible to the style of programming they follow. Different programs need the ability to trigger a batch job in different ways, including command line and schedulers, or a part of the program itself. It is also important to stop executing batch jobs elegantly if needed.

In this chapter, we will cover the following topics:

- Spring Batch XML features
- Configuring jobs, transactions, and repositories
- EL and listeners
- Executing jobs from command line and web applications
- Schedulers

Spring Batch XML features

Spring Batch XML configuration is the most important aspect of Spring Batch programming. Spring Batch has a unique XML terminology and namespace. Understanding these terminologies and using the right set of entities helps to build an efficient batch application.

Spring Batch XML namespace

Spring Batch has dedicated XML namespace support to provide comfortable configurations. The Spring XML application context file needs to have the following declaration to activate the namespace:

```
<beans xmlns:batch="http://www.springframework.org/schema/batch"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch-3.0.xsd">

<batch:job id="readDetailsJob">
...
</batch:job>

</beans>
```

The namespace configuration provides the prefix that can be used to configure the details in the context file. In the preceding example, we have `batch` as a prefix to configure a job. The prefixes are the identifiers specific to this document only. One can use any valid names as a prefix for the namespace configuration. If any namespace is configured without a prefix, it is considered as the default namespace, and one should configure the elements without a prefix to configure using the default prefix. In the previous chapter, we configured `batch` as the default prefix and, hence, we directly configured the job and the step.

Spring Batch XML tags

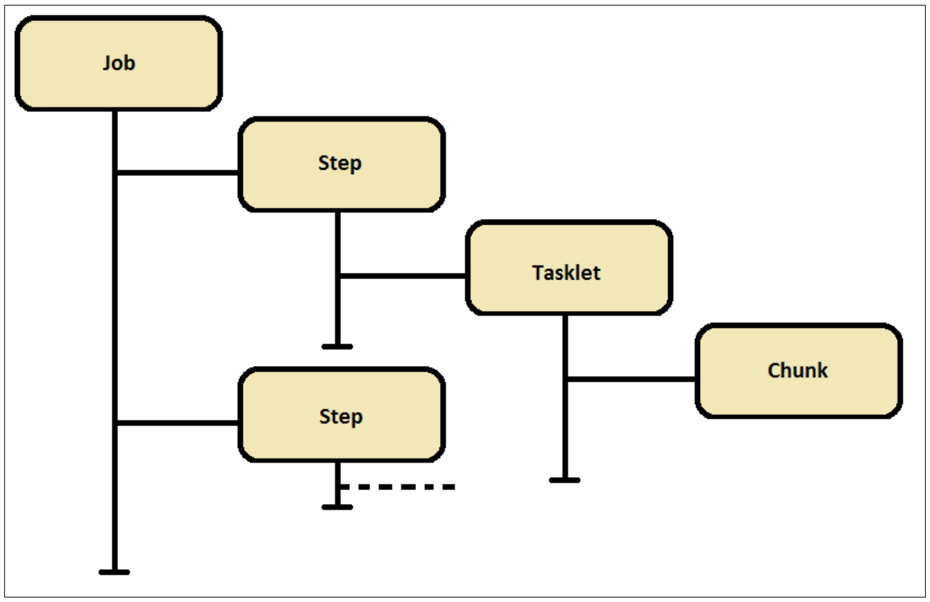
Spring Batch XML configuration defines the flow of the batch job. Following are the important Spring Batch tags and their descriptions:

- `job`: This defines a job composed of a set of steps and transitions between the steps. The job will be exposed in the enclosing bean factory as a component of type `Job` that can be launched using `JobLauncher`.
- `step`: This defines a stage in job processing backed by a step.
- `tasklet`: This declares the implementation of the tasklet strategy (the strategy for processing in a step). It can be done by configuring a chunk or by configuring a reference to the `Tasklet` interface.

- `chunk`: This declares that the owning step will perform chunk-oriented processing (reading data one at a time and creating groups to be written), delegate what defines a chunk, and configure the chunk-oriented components.
- `job-repository`: This configures `JobRepository` (the repository responsible for persistence of batch metadata entities) using a relational data store. It is needed by other components such as the `job` and `step` implementations.
- `flow`: This defines a flow composed of a set of steps and transitions between steps.

Configuring jobs, transactions, and repositories

As mentioned in the previous section, we can configure Spring Batch jobs conveniently through XML configuration itself. **Job** is the primary element and the following figure shows the hierarchy of the components in the configuration:



Each job can contain multiple steps, each step contains tasklets, and each tasklet contains chunks. Each component has individual elements that are defined as subelements of the other. The following is the syntax for one such batch job:

```
<beans xmlns:batch ... >
<batch:job id="jobId">
```

```
    <batch:step id="stepId">
      <batch:tasklet ref="beanReference">
        <batch:chunk reader="dataReader"
          processor="dataProcessor"
          writer="dataWriter" commit-interval="500" />
      </batch:tasklet>
    </batch:step>
    <batch:step>
    ...
    </batch:step>
  </batch:job>
  </beans>
```

Job configuration

Job is the root element in batch application configuration. A job defines the batch job to be executed with the configurations of job repository, and properties such as restartable or not. The following are the attributes of the `job` element:

- `id`: This is a unique identifier for the `job` element.
- `abstract`: This is used to configure if the job is abstract, that is, it is not meant to be instantiated by itself, but rather it is just serving as a parent for concrete child job definitions. By default it is `false`.
- `increment`: This is a reference to a `JobParametersIncrementer` bean definition. This will be used to provide a new set of parameters by altering the previous set of parameters to be eligible for a fresh run as the `next` instance.
- `job-repository`: This is the bean name of the `JobRepository` that is to be used. This attribute is not mandatory, and it defaults to the `jobRepository` bean.
- `parent`: This is the name of the parent job from which a job should inherit.
- `restartable`: This defines whether the job should be retartable or not in the case of failure. Set this to `false` if the job should not be restarted. By default it is `true`.

A validator of type `DefaultJobParametersValidator` can be configured as a part of the job configuration to validate simple and optional parameters. The following is a snippet of such a configuration:

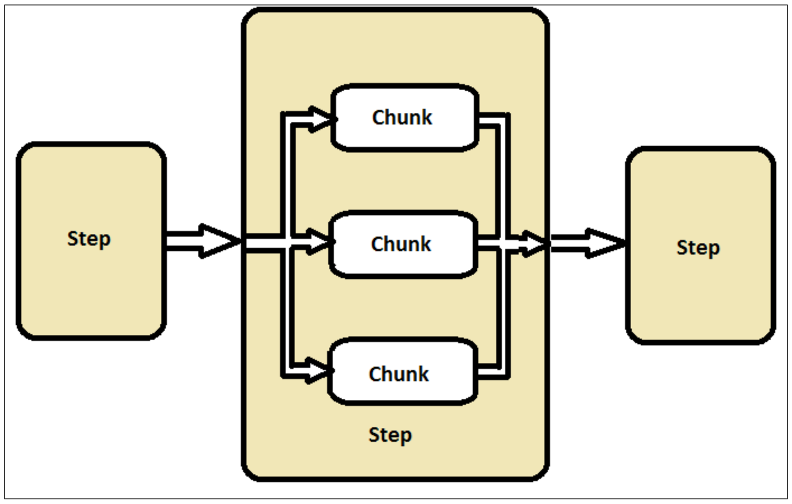
```
<beans xmlns:batch ... >
<batch:job id="jobId">
  <batch:step id="stepId">
    ...
  </batch:step>
  <batch:validator ref="validatorId"/>
</batch:job>

<bean id="validatorId" class="beans.JobParametersValidator">
  <property name="Keys">
    <set>
      <value>keyValues</value>
    </set>
  </property>
</bean>
</beans>
```

For complex constraints, the `validator` interface can also be implemented.

Step configuration

Step is the first subelement of a job. A job can contain multiple steps. The following are different approaches in which multiple steps can be configured:

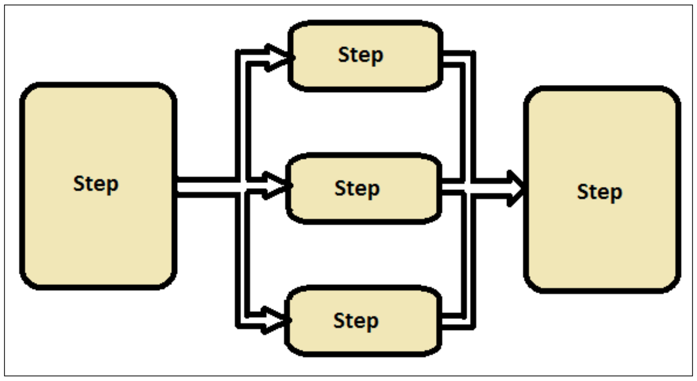
- **Multithreaded step (single process):** Spring Batch allows you to execute chunks of work in parallel as a part of a single process. Each chunk processes a set of records when there is a large amount of data to process in threads.



The simplest way to start parallel processing is by adding `taskExecutor` to your step configuration as an attribute of the tasklet.

```
<step id="loading">
  <tasklet task-executor="taskExecutor">...</tasklet>
</step>
```

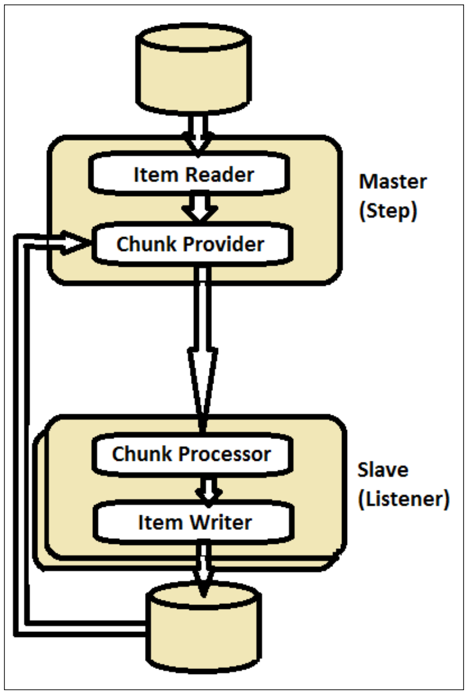
- **Parallel step (single process)**: This is the mechanism of processing multiple steps in a single process.



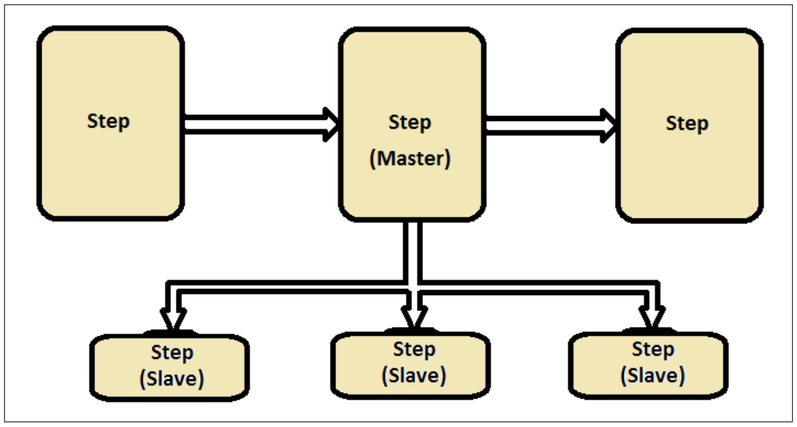
The following is the snippet to configure parallel steps:

```
<job id="jobId">
  <split id="splitId" task-executor="taskExecutor"
    next="step3">
    <flow>
      <step id="step1" next="step2"/>
      <step id="step2"/>
    </flow>
  </split>
  <step id="step3"/>
</job>
<beans:bean id="taskExecutor" class="TaskExecutor"/>
```

- **Remote chunking of step (multiprocess)**: This splits the step processing across multiple processes, communicating with each other through a middleware. A step of Spring Batch acts as the master and the listeners of corresponding middleware act as the slaves. While the master component runs as a single process, slaves are the multiple remote processes.



- **Partitioning a step (single process or multiprocess)**: Partitioning is the process in which one step is configured to have sub steps. The super step is the master and the sub steps are the slaves. Slave steps have to complete the execution to consider the master step as completed.



The following are the attributes of the step element:

- `id`: This is the unique identifier for the `step` element
- `next`: This is a shortcut to specify the next step to execute after the current step
- `parent`: This is the name of the parent step from which a job should inherit
- `allow-start-if-complete`: This is set to `true` to allow a step to be started even if it is already complete

The following is a sample step configuration:

```
<step id="firstBatchStepOne">
  <tasklet ref="firstBatch"/>
</step>
<job id="firstBatchJob">
  <step id="stepOne" parent="firstBatchStepOne"/>
</job>
<bean id="firstBatch" class="FirstBatch"/>
```

Tasklet configuration

Tasklet is the subelement of the step element that can be used to specify the step process that is repeatable and transactional as part of a step.

The following are the attributes of the `tasklet` element:

- `ref`: This is the reference to a bean definition that implements the `Tasklet` interface.
- `allow-start-if-complete`: This is set to `true` to allow a step to be started even if it is already complete.
- `method`: This is the method specification for the tasklet execution.
- `start-limit`: This is the maximum number of times a step may be started.
- `task-executor`: The task executor is responsible for the execution of the task.
- `throttle-limit`: This is the maximum number of tasks that can be queued for concurrent processing to prevent thread pools from being overwhelmed. The default value is `4`.
- `transaction-manager`: This is the bean name of the transaction manager that is to be used. Default is `transactionManager`, if not specified.

The following is the sample job configuration with tasklet:

```
<step id="firstBatchStepOne">
  <tasklet ref="firstBatch" start-limit="6">
  ...
  </tasklet>
</step>
<job id="firstBatchJob">
  <step id="stepOne" parent="firstBatchStepOne"/>
</job>
<bean id="firstBatch" class="FirstBatch"/>
```

Chunk configuration

Chunk is the child element of tasklet that can be used to perform read-write processing. Chunk configuration involves more data beans compared to other element's configuration.

The following are the attributes of the chunk element:

- `reader`: This is the bean name of the item reader that is to be used for the process and implements the `ItemReader` interface.
- `processor`: This is the bean name of the item processor that is to be used for the process and implements the `ItemProcessor` interface.
- `writer`: This is the bean name of the item writer that is to be used for the process and implements the `ItemWriter` interface.
- `cache-capacity`: This is the capacity of the cache in the retry policy.
- `chunk-completion-policy`: A transaction will be committed when this policy decides to complete. Defaults to `SimpleCompletionPolicy` with the chunk size equal to the `commit-interval` attribute.
- `commit-interval`: The number of items that will be processed before `commit` is called for the transaction. Set either this or the `chunk-completion-policy` attribute, but not both.
- `processor-transactional`: This determines whether the processor is transaction aware or not.
- `reader-transactional-queue`: This determines whether the reader is a transactional queue.
- `retry-limit`: This is the maximum number of times the processing of an item will be retried.
- `retry-policy`: This is the bean specification of the retry policy. If specified, then the `retry-limit` and `retryable` exceptions are ignored.

- `skip-limit`: This is the maximum number of items that will be allowed to be skipped.
- `skip-policy`: This is the bean specification of skip policy. If specified, then the `skip-limit` and `skippable` exceptions are ignored.

The following is the sample job configuration with the tasklet chunk:

```
<step id="firstBatchStepOne">
  <tasklet ref="firstBatch">
    <chunk reader="itemReader" processor="itemProcessor"
writer="itermWriter" commit-interval="150"/>
  </tasklet>
</step>
<job id="firstBatchJob">
  <step id="stepOne" parent="firstBatchStepOne"/>
</job>
<bean id="firstBatch" class="FirstBatch"/>
<bean id="itemReader" class="ItemReader"/>
<bean id="itemProcessor" class="ItemProcessor"/>
<bean id="itemWriter" class="ItemProcessor"/>
```

The chunk configuration can have the exception skip and retry elements added as its child components. The `skippable-exception-classes` and `retryable-exception-classes` elements for the skip and retry configurations. The bean configuration can as well be annotated to keep the Spring Batch configuration simpler.

Transaction configuration

Transaction configuration is one of the key aspects of Spring Batch. The Spring transaction manager is the configuration for transactions. Spring provides diverse transaction managers for diverse specifications; for JDBC it is `DataSourceTransactionManager` and for JPA it is `JpaTransactionManager`.

Spring Batch lets us configure the `transaction-attributes` element as a child element of the chunk, to set the isolation and propagation levels of the transaction.

The exceptions for which the rollback operation need not be performed can be chosen. These exceptions can be configured using the `include` element as a child of the `no-rollback-exception-classes` element, which is a child element of the tasklet.

The following is a sample job configuration with the transaction manager:

```
<step id="firstBatchStepOne">
  <tasklet ref="firstBatch" transaction-manager="transactionManager">
  ...
```

```
    </tasklet>
  </step>
  <job id="firstBatchJob">
    <step id="stepOne" parent="firstBatchStepOne"/>
  </job>
  <bean id="firstBatch" class="FirstBatch"/>
  <bean id="transactionManager" class="org.springframework.jdbc.
  datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="DataSource"/>
  </bean>
```

Job repository configuration

The job repository maintains the information related to the job execution. It also maintains the state of the batch job. Job repositories are available in two types from Spring Batch: in-memory repository and persistent repository.

In-memory repository lets the job run again for the same job configuration and parameters multiple times. In-memory repository is volatile and so, it does not allow restart between JVM instances. It also cannot guarantee that any two job instances with the same parameters will be launched concurrently, hence, it is not suitable in a multithreaded job or in a locally partitioned step. It can be configured using `MapJobRepositoryFactoryBean`.

It requires the transaction manager for rollback semantics within the repository and to handle the business logic defined in the transactional database.

The following is a sample in-memory repository configuration:

```
<bean id="jobRepository" class="org.springframework.batch.core.
repository.support.MapJobRepositoryFactoryBean">
  <property name="transactionManager-ref"
    ref="transactionManager"/>
</bean>
<bean id="transactionManager"
class="org.springframework.batch.support.transaction.
ResourcelessTransactionManager"/>
<job id="deductionsJob" job-repository="jobRepository">
...
</job>
```

Persistent repository can be configured using the `job-repository` element to perform persistent database operations on a database. The datasource can be configured using any API, for example, we have used Apache commons `BasicDataSource` in the following configurations.

The following is a sample persistent repository configuration:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${batch.jdbc.driver}" />
  <property name="url" value="${batch.jdbc.url}" />
  <property name="username" value="${batch.jdbc.user}" />
  <property name="password" value="${batch.jdbc.password}" />
</bean>
<bean id="transactionManager" class="org.springframework.jdbc.
datasource.DataSourceTransactionManager" lazy-init="true">
  <property name="dataSource" ref="dataSource" />
</bean>
<job-repository id="jobRepository" data-source="dataSource"
transaction-manager="transactionManager"/>
```

EL and listeners

Spring Batch provides an interesting feature starting from version 3: **Expression Language** (**EL**). **Spring Expression Language** (**SpEL**) lets us make XML configuration dynamic by capturing the values at runtime from execution context. SpEL can resolve the expressions from both properties and beans. This runtime capturing behavior lets the job access late binding configurations as well.

The following is a sample SpEL configuration:

```
<bean id="processBean" class="JobProcessBean" scope="step">
  <property name="name" value="#{jobParameters[name]}"/>
</bean>
```

Listeners

Spring Batch can be configured to have a set of additional events identified with the help of listeners. Listeners can be used in different combinations to identify the events at different levels. The following are the various listener types provided by Spring Batch for batch processing.

- Job listeners: They identify the job level events
- Step listeners: They identify the step level events
- Item listeners: They identify the item repetition and retry events

Job listeners

Job listeners identify the events occurring at the job level. Job listeners can be configured by the following:

- **Implementing JobExecutionListener**: The following is the sample listener configuration using `JobExecutionListener` implementation:

```
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobExecutionListener;
public class JobStatusListener implements JobExecutionListener {
public void beforeJob(JobExecution jobExecution) {
  System.out.println("Job: " +
  jobExecution.getJobInstance().getJobName() + " is beginning");
}
public void afterJob(JobExecution jobExecution) {
  System.out.println("Job: " + jobExecution.getJobInstance().
  getJobName() + " has completed");
  System.out.println("The Job Execution status is: " +
  jobExecution.getStatus());
}
}
```

  The XML configuration for the preceding defined listener is as follows:

```
<job id="employeeDeductionsJob">
  <listeners>
    <listener ref="jobStatusListener"/>
  </listeners>
</job>
```

- **Using annotations**: The following is the sample listener configuration using annotations:

```
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobExecutionListener;
import org.springframework.batch.core.annotation.AfterJob;
import org.springframework.batch.core.annotation.BeforeJob;
public class JobStatusListener {
@BeforeJob
public void beforeJob(JobExecution jobExecution) {
  System.out.println("Job: " +
  jobExecution.getJobInstance().getJobName() + " is
  beginning");
}
@AfterJob
public void afterJob(JobExecution jobExecution) {
  System.out.println("Job: " + jobExecution.getJobInstance().
getJobName() + " has completed");
  System.out.println("The Job Execution status is: " +
  jobExecution.getStatus());
}
}
```

The way to configure the annotated listener is same as the `JobExecutionListener` configuration.

Step listeners

Just like job listeners capture the execution status of jobs, steps have certain listeners to capture different events. The way of implementing this set of listeners is the same as the job listeners (by implementing the corresponding interface or by using annotations), except that the listener element has to be configured as the child element of the `step` element.

The following is a list of step listeners with the methods to override:

- `StepExecutionListener`: This identifies the before and after of step execution events using the `beforeStep` and `afterStep` methods respectively
- `ChunkListener`: This identifies the before and after of chunk execution events using the `beforeChunk` and `afterChunk` methods respectively
- `ItemReadListener`: This identifies if the before and after item is read and when an exception occurs, it reads an item event using the `beforeRead`, `afterRead`, and `onReadError` methods respectively
- `ItemProcessListener`: This identifies the state before and after `ItemProcessor` gets an item and when an exception is thrown by the processor using the `beforeProcess`, `afterProcess`, and `onProcessError` methods respectively
- `ItemWriteListener`: This identifies the before and after of when an item is written and when an exception occurs, writing an item event using the `beforeWrite`, `afterWrite`, and `onWriteError` methods respectively
- `SkipListener`: This identifies the skip event of reading, processing, or writing an item using the `onSkipInRead`, `onSkipInProcess`, and `onSkipInWrite` methods respectively

Item listeners

Item listeners identify the retry and repeat events. These listeners can be configured in the same way as job or step listeners.

The following are the item listeners with the methods to override:

- `RepeatListener`: This identifies the before and after of each repeat event using the `before` and `after` methods respectively. It identifies the first and last repeat event using the `open` and `close` methods respectively. It also identifies every failure event using the `onError` method.
- `RetryListener`: This identifies the first and last try event, irrespective of whether the retry is a success or a failure, using the `open` and `close` methods respectively. It also identifies the every failure event using the `onError` method.

Executing jobs from the command line and web applications

In the first chapter, we learned how to configure and run a simple batch job application using Spring Batch, by launching the job from a Java program. The Java-based API of Spring Batch makes the job launching very convenient through different ways of invoking the batch job. In this section, let's examine the concepts of launching a batch job in different ways and stopping the batch job elegantly.

JobLauncher

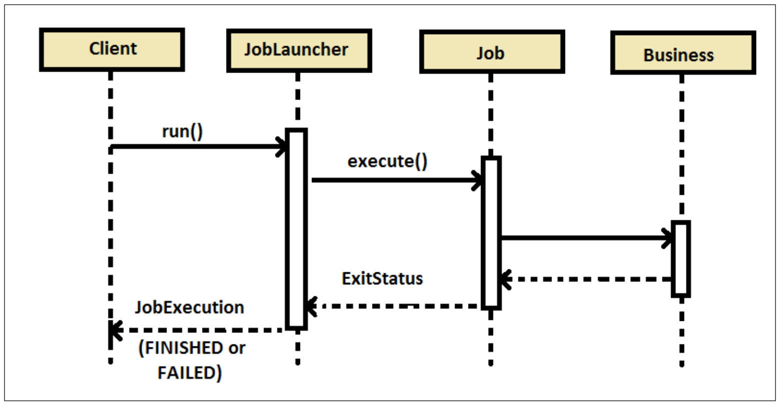
Spring Batch makes it easier to launch a batch job with the help of the `JobLauncher`. `JobLauncher` represents a simple interface to launch a job with a given set of job parameters. The run method of `JobLauncher` takes `Job` and `JobParameters` of type Spring beans as parameters and invokes the batch job execution.

The following is the code snippet we have used in the previous chapter to launch a job using `JobLauncher`:

```
String[] springConfig  = {"batch/firstBatch.xml"};
context = new ClassPathXmlApplicationContext(springConfig);
JobLauncher jobLauncher = (JobLauncher) context.
getBean("jobLauncher");
Job job = (Job) context.getBean("firstBatchJob");
JobExecution execution = jobLauncher.run(job, new JobParameters());
System.out.println("Exit Status : " + execution.getStatus());
```

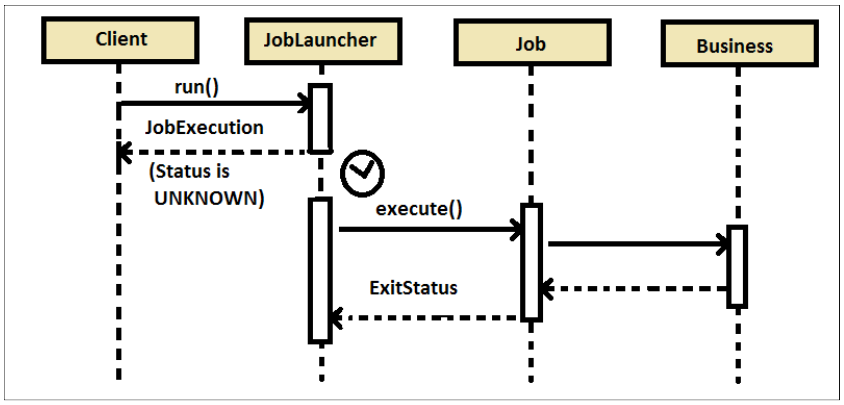
We can use `JobParametersBuilder` to build different types of `JobParameter`. A `JobLauncher` can be configured with a persistent job repository, using the following syntax:

```
<bean id="jobLauncher"
class="org.springframework.batch.core.launch.support.
SimpleJobLauncher">
  <property name="jobRepository" ref="jobRepository"/>
</bean>
```



Launching a job from the `run` method of `JobLauncher` calls `execute` on `job` and confirms the job execution status (`FINISHED` or `FAILED`) after execution, which is a synchronous process.

However, in certain business scenarios, we want the `JobLauncher` to invoke and handover the process to another controller to make it asynchronous so that multiple processes can be triggered. `TaskExecutor` helps in this scenario, if configured along with `JobLauncher`.



The following is the syntax to configure `SimpleJobLauncher` with `taskExecutor` to make the process asynchronous:

```
<bean id="jobLauncher"
class="org.springframework.batch.core.launch.support.
SimpleJobLauncher">
<property name="jobRepository" ref="jobRepository"/>
<property name="taskExecutor">
<bean class="org.springframework.core.task.SimpleAsyncTaskExecutor"/>
</property>
</bean>
```

Launching a job from a command line

A `CommandLineJobRunner` makes it simple to launch a Spring Batch job from a command line. The following are the steps in which `CommandLineJobRunner` works:

- Loading the appropriate `ApplicationContext`
- Parsing command-line arguments into `JobParameters`
- Locating the appropriate job based on arguments
- Using the `JobLauncher` provided in the application context to launch the job

The following is the command to launch a job using `CommandLineJobRunner`:

```
> java -classpath "/lib/*" org.springframework.batch.core.launch.
support.CommandLineJobRunner firstBatch.xml firstBatchJob schedule.
date(date)=2007/05/05
```

The exit code of the job execution represents the status of the batch job after run, where `0` represents `COMPLETED`, `1` represents `FAILED`, and `2` represents an error from the command-line job runner, such as not being able to find a job in the provided context.

Launching a job from within a web application

So far, we have learned how to launch a batch job from a Java program and command line. There are scenarios when a job needs to be launched from within a web application. Applications that generate reports from within an application and trigger asynchronous processes from the applications with thread-based configuration are such business scenarios.

The following is the program to launch the job using Spring MVC framework with Spring dependency:

```
@Controller
public class JobLauncherController {
@Autowired
JobLauncher jobLauncher;
@Autowired
Job job;
@RequestMapping("/jobLauncher.html")
public void handle() throws Exception {
  jobLauncher.run(job, new JobParameters());
}
}
```

The controller launches the jobs using `JobController` that is auto-wired in the `JobLauncherController` through configurations. The controller can be called from a request URL with `RequestMapping` configured with the `handle` method.

Stopping batch jobs gracefully

Jobs can be gracefully stopped when necessary from within a program with the help of the `JobOperator` interface. `JobOperator` provides the CRUD operations of the job.

The following is the syntax to stop a job using `JobOperator`:

```
Set<Long> executions =jobOperator.getRunningExecutions("jobName");
If( executions.iterator().hasNext()) {
jobOperator.stop(executions.iterator().next());
}
```

`JobOperator` identifies the running job with the given `jobName` and calls the `stop` method by attaining the job `id` from executions.

The `JobOperator` needs to be configured to be available for the program. The following is the sample configuration of the `jobOperator` with the resources, job explorer (the entry point to browse executions of running, or historical jobs and steps), registry, and repository properties.

```
<bean id="jobOperator" class="org.springframework.batch.core.launch.
support. SimpleJobOperator">
<property name="jobExplorer">
<bean class=" org.springframework.batch.core.explore.support.
JobExplorerFactoryBean">
<property name="dataSource" ref="dataSource"/>
</bean>
```

```
</property>
<property name="jobRepository" ref="jobRepository"/>
<property name="jobRegistry" ref="jobRegistry"/>
<property name="jobLauncher" ref="jobLauncher"/>
</bean>
```

The job configuration supports the stop setup at tasklet and the chunk-oriented step level as well.

Schedulers

Schedulers are the programs that can periodically launch other programs. As mentioned earlier, Spring Batch is not a scheduling framework. Spring Batch can work in conjunction with a scheduler (such as Quartz/Control-M), but cannot replace a scheduler.

The following are popular schedulers:

- **Cron**: This is an expression-based job scheduler available on Unix-like systems to launch other programs
- **Control-M**: This is a batch scheduling software available for distributed computing platforms including Unix, Windows, Linux, and OpenVMS environments
- **Spring scheduler**: This scheduler from Spring supports XML, or annotation-based or cron expressions to launch a batch job
- **Quartz**: Quartz is a richly featured, open source job scheduling library that can be integrated within virtually any Java application

While Cron and Control-M can use `CommandLineJobRunner` to launch a batch job, Quartz and Spring scheduler can launch the batch job from within the application programmatically. One can choose between these options based on the frequency of job execution and the way it is to be invoked.

Summary

In this chapter, we learned the configuration details of the Spring Batch jobs and their components to be able to effectively use them for business needs. We learned how to make the batch programming more efficient and flexible using XML- and annotation-based configurations. We also learned different ways of launching the batch job, such as from a command-line, a Java program, and within a web application, and also how to stop a batch job gracefully from within a program. We completed this chapter with an understanding of the different job schedulers available in the market and which launching solution can be used in combination with these schedulers.

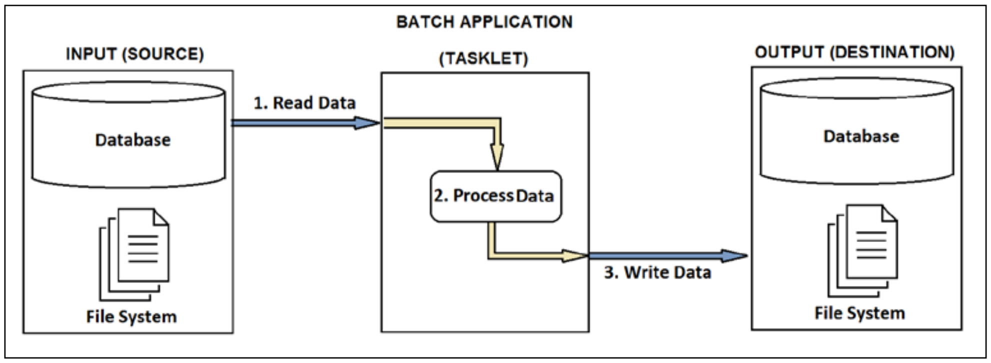
In the next chapter, we will learn in detail about reading, processing, and writing different forms of data using Spring Batch.

3
Working with Data

In the previous chapter, we learned about batch configurations, components, and execution modes to match with diverse business needs. Handling the data, which includes reading, processing, and writing, is an essential part of any kind of application, and batch applications are no exception. Spring Batch provides the ability to read different forms of data, process the data in the way business expects, and write it back to different systems, which can be easily integrated with different frameworks.

In this chapter, we will cover three major operations involved in data handling:

- Data reading
- Data processing
- Data writing



The preceding figure shows the three stages of handling data in a batch application. The input (source) can be a database, a filesystem (flat file or XML), or data from a web service as well. Applications need to read the data from an input (source), process it, and write to the output (destination) system. The output (destination) can be a database or a filesystem (a flat file or an XML file). In the processing stage, data read in format can be verified and transformed into the desired format before writing to the output. Let's examine each of these stages now.

Data reading

Spring Batch provides the configuration to read different forms of data from different sources, including flat files, XML, and relational databases. It also supports the custom reader configurations for the formats that are not available with the specification.

ItemReader

Spring Batch provides an interface in the form of `ItemReader` to read bulk data from different forms, which include the following:

- **Flat files**: These are typically of two types: fixed width and delimited character-based files
- **XML**: This format is used for different forms of application data
- **Database**: This maintains a set of records of similar or different groups of information

The following is the definition of the `ItemReader` interface:

```
public interface ItemReader<T> {

T read() throws Exception, UnexpectedInputException, ParseException;

}
```

Let's examine how `ItemReader` can help us with reading different formats.

Reading data from flat files

Flat files are configured in two formats, namely, **fixed width** and **delimited**. Fixed width files have each field detail configured with a predefined fixed width, whereas the delimited files have fields with a specific character (or tab in general) used to delimit them from the other fields.

Fixed width file

A fixed width file generally has a predefined specification of its fields, how much length each field should occupy on the file, and from which position to which position on a line.

The following is one such specification of the fixed width file we want to read:

```
Field                 Length                   Position
ID                    2 characters             1 to 2
Last name             10 characters            3 to 12
First name            10 characters            13 to 22
Designation           10 characters            23 to 32
Department            15 characters            33 to 47
Year of joining       4 characters             48 to 51
```

Using the preceding specification, let's fill a sample file with employee information in a fixed width file, as follows (`employees.txt`):

```
11Alden     Richie    associate   sales          1996
12Casey     Stanley   manager     sales          1999
13Rex       An        architect   development    2001
14Royce     Dan       writer      development    2006
15Merlin    Sams      accountant  finance        1995
16Olin      Covey     manager     finance        1989
```

If a Java object is to be generated corresponding to this specification, we can create a **Plain Old Java Object** (**POJO**) with the following representation:

```
package batch;

import java.io.Serializable;

public class Employee implements Serializable {
  int id;
  String lastName;
  String firstName;
  String designation;
  String department;
  int yearOfJoining;

  public int getID() {
  return id;
  }
  public void setID(int id) {
    this.id = id;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  public String getDesignation() {
    return designation;
  }
  public void setDesignation(String designation) {
    this.designation = designation;
  }
  public String getDepartment() {
    return department;
  }
  public void setDepartment(String department) {
    this.department = department;
  }
  public int getYearOfJoining() {
    return yearOfJoining;
  }
  public void setYearOfJoining(int yearOfJoining) {
    this.yearOfJoining = yearOfJoining;
  }

  public String toString() {
    return "Employee: ID="+ id + ", Last Name="+ lastName +
    ", First Name="+ firstName + ", Designation="+ designation +
    ", Department="+ department + ",Year of joining="+
    yearOfJoining;
    }
}
```

FlatFileItemReader

`FlatFileItemReader` provides a means of reading different types of flat files and parsing them by the following:

- `resource`: This represents the file from which the data has to be read.
- `lineMapper`: This represents the mapper that converts the String that is read by `ItemReader` to the Java object.
- `linesToSkip`: This is used with files having header content before records. It is the number of lines we want to ignore at the top of the file.

LineMapper

The `LineMapper` interface lets us read each line from the file with the line number passed in every iteration. It is the Spring Batch standard implementation of `LineMapper`. The following is the `LineMapper` interface:

```
public interface LineMapper<T> {

T mapLine(String line, int lineNumber) throws Exception;

}
```

The `LineTokenizer` interface converts the line read from `LineMapper` to the set of fields (`FieldSet`). `DelimitedLineTokenizer`, `FixedLengthTokenizer`, and `PatternMatchingCompositeLineTokenizer` are Spring Batch's supporting implementations of `LineTokenizer`. The following is the `LineTokenizer` interface:

```
public interface LineTokenizer {

FieldSet tokenize(String line);

}
```

The `FieldSetMapper` interface lets us map each field from the String read to the `Employee` object. The following is the `FieldSetMapper` interface:

```
public interface FieldSetMapper<T> {

  T mapFieldSet(FieldSet fieldSet);

}
```

We can implement `FieldSetMapper` for our `Employee` data, as follows:

```
package batch;

import org.springframework.batch.item.file.mapping.FieldSetMapper;
import org.springframework.batch.item.file.transform.FieldSet;

class EmployeeFieldSetMapper implements FieldSetMapper<Employee> {
  public Employee mapFieldSet(FieldSet fieldSet) {
    Employee emp = new Employee();
    emp.setID(fieldSet.readInt("ID"));
    emp.setLastName(fieldSet.readString("lastName"));
    emp.setFirstName(fieldSet.readString("firstName"));
    emp.setDesignation(fieldSet.readString("designation"));
    emp.setDepartment(fieldSet.readString("department"));
    emp.setYearOfJoining(fieldSet.readInt("yearOfJoining"));
    return emp;
  }
}
```

The data can be read from the file as part of the batch job, as shown in the following code snippet:

```
FlatFileItemReader<Employee> itemReader = new
FlatFileItemReader<Employee>();
itemReader.setResource(new FileSystemResource("employees.txt"));
// FixedLengthTokenizer reads each field of length specified
DefaultLineMapper<Employee> lineMapper = new
DefaultLineMapper<Employee>();
FixedLengthTokenizer lineTokenizer = new FixedLengthTokenizer();

lineMapper.setLineTokenizer(lineTokenizer);
lineMapper.setFieldSetMapper(new EmployeeFieldSetMapper());
itemReader.setLineMapper(lineMapper);
itemReader.open);
Employee ;
while (employee != null) {
  employee = itemReader.read();
  if (employee == null) {
    return RepeatStatus.FINISHED;
  }
  System.out.println(employee.toString());
}
```

The `setResource()` method sends the flat file resource to `FlatFileItemReader`. The `LineTokenizer` interface can be used with field names and ranges with the start and end positions on the file set as an array using the `setNames()` and `setColumns()` methods respectively. Every time the `read()` method is invoked on `itemReader`, it reads a line and moves on to the next.

The following is the output of the batch program after reading data from a fixed width flat file and capturing them into Java objects:

```
** Executing the fixed width file read batch job! **
Employee: ID=11, Last Name=Alden, First Name=Richie,
Designation=associate, Department=sales,Year of joining=1996
Employee: ID=12, Last Name=Casey, First Name=Stanley,
Designation=manager, Department=sales,Year of joining=1999
Employee: ID=13, Last Name=Rex, First Name=An, Designation=architect,
Department=development,Year of joining=2001
Employee: ID=14, Last Name=Royce, First Name=Dan, Designation=writer,
Department=development,Year of joining=2006
Employee: ID=15, Last Name=Merlin, First Name=Sams,
Designation=accountant, Department=finance,Year of joining=1995
Employee: ID=16, Last Name=Olin, First Name=Covey,
Designation=manager, Department=finance,Year of joining=1989
Exit Status : COMPLETED
```

The `reader`, `linetokenizer`, and `fieldsetmapper` are used in the batch as beans, as follows:

```
<beans:bean id="employeeFile"
class="org.springframework.core.io.FileSystemResource" >
<beans:constructor-arg value=""/>
</beans:bean>
<beans:bean id="employeeReader"
class="org.springframework.batch.item.file.FlatFileItemReader">
<beans:property name="resource" ref="employeeFile" />
<beans:property name="lineMapper">
<beans:bean class="org.springframework.batch.item.file.mapping.
DefaultLineMapper">
<beans:property name="lineTokenizer">
<beans:bean class="org.springframework.batch.item.file.transform.
FixedLengthTokenizer">
<beans:property name="names" value="ID, lastName, firstName,
designation, department, yearOfJoining"/>
<beans:property name="columns"
value="1-2,3-12,13-22,23-32,33-47,48-51"/>
</beans:bean>
</beans:property>
```

```
<beans:property name="fieldSetMapper">


</beans:bean>
</beans:property>
</beans:bean>
</beans:property>
</beans:bean>
<beans:bean id="employee" class="" />batchstep
```

Delimited file

A delimited flat file contains fields separated by a specific character in each line. The following is an example of a delimited file, with each field delimited by the character comma. Let's take an example of reading employee details from a delimited flat file.

The following are the specifications of the file:

- ID
- Last name
- First name
- Designation
- Department
- Year of joining

Each field should be separated from the other with a comma. The following is the sample file content (`employees.csv`):

```
1,Alden,Richie,associate,sales,1996
2,Casey,Stanley,manager,sales,1999
3,Rex,An,architect,development,2001
4,Royce,Dan,writer,development,2006
5,Merlin,Sams,accountant,finance,1995
6,Olin,Covey,manager,finance,1989
```

Delimited files need to be handled the same way as the fixed width flat files, except that `LineTokenizer` used in this case should be `DelimitedLineTokenizer`. The following is the Java code realized to read a delimited flat file to be handled as part of the batch job:

```
// Delimited File Read
FlatFileItemReader<Employee> itemReader = new
FlatFileItemReader<Employee>();
itemReader.setResource(new FileSystemResource("employees.csv"));
```

```
// DelimitedLineTokenizer defaults to comma as its delimiter
DefaultLineMapper<Employee> lineMapper = new
DefaultLineMapper<Employee>();
DelimitedLineTokenizer lineTokenizer = new DelimitedLineTokenizer();
lineTokenizer.setNames(new String[] { "ID", "lastName", "firstName",
"designation", "department", "yearOfJoining" });
lineMapper.setLineTokenizer(lineTokenizer);
lineMapper.setFieldSetMapper(new EmployeeFieldSetMapper());
itemReader.setLineMapper(lineMapper);
itemReader.open(new ExecutionContext());
  employee = itemReader.read();
  if (employee == null) {
    return RepeatStatus.FINISHED;
  }
  System.out.println(employee.toString());
}
```

With the delimited files, we don't have to set the column's property. The delimiter used has to be set unless the delimiter is a comma. Executing this program should read the delimited flat files into the Java objects and the output is as follows:

```
** Executing the delimited file read batch job! **
Employee: ID=1, Last Name=Alden, First Name=Richie,
Designation=associate, Department=sales,Year of joining=1996
Employee: ID=2, Last Name=Casey, First Name=Stanley,
Designation=manager, Department=sales,Year of joining=1999
Employee: ID=3, Last Name=Rex, First Name=An, Designation=architect,
Department=development,Year of joining=2001
Employee: ID=4, Last Name=Royce, First Name=Dan, Designation=writer,
Department=development,Year of joining=2006
Employee: ID=5, Last Name=Merlin, First Name=Sams,
Designation=accountant, Department=finance,Year of joining=1995
Employee: ID=6, Last Name=Olin, First Name=Covey, Designation=manager,
Department=finance,Year of joining=1989
Exit Status : COMPLETED
```

The `ItemReader`, `LineTokenizer`, and `FieldSetMapper` in the case of delimited files can be configured in the batch as beans and used in the program as follows:

```
<beans:bean id="employeeFile"
  class="org.springframework.core.io.FileSystemResource"
  scope="step">
<beans:constructor-arg value="#{jobParameters[employeeFile]}"/>
</beans:bean>
<beans:bean id="employeeFileReader" class="org.springframework.batch.
item.file.FlatFileItemReader">
```

```
<beans:property name="resource" ref="employeeFile" />
<beans:property name="lineMapper">
<beans:bean class="org.springframework.batch.item.file.mapping.
DefaultLineMapper">
<beans:property name="lineTokenizer">
<beans:bean class="org.springframework.batch.item.file.transform.
DelimitedLineTokenizer">
<beans:property name="names" value="ID, lastName, firstName,
designation, department, yearOfJoining"/>
<beans:property name="delimiter" value=","/>
</beans:bean>
</beans:property>
<beans:property name="fieldSetMapper">
<beans:bean
class="batch.EmployeeFieldSetMapper"/>
</beans:property>
</beans:bean>
</beans:property>
</beans:bean>
```

If the lines of file are defined in a different format specific to a business, `LineTokenizer` is open for custom implementation and configuration. The `PatternMatchingCompositeLineMapper` can be used to read files with complex patterns. For example, if we have multiple record types within a single flat file, we can use `PatternMatchingCompositeLineMapper` to have tokenizers for each record type, as follows.

A sample flat file with multiple record types:

```
EMPLOYEE,Steve,Jacob,21,manager,2009
BANKINFO,524851478569,STEVEJACOB,REDROSECITY
ADDRESSINFO,No24,SUNFLOWERWAY,CASAYA
```

The following is the bean configuration for this multiple record type:

```
<bean id="employeeFileLineMapper"
class=" org.springframework.batch.item.file.mapping.
PatternMatchingCompositeLineMapper">
<property name="tokenizers">
<map>
<entry key="EMPLOYEE*" value-ref="employeeTokenizer"/>
<entry key="BANKINFO*" value-ref="bankInfoTokenizer"/>
<entry key="ADDRESSINFO*" value-ref="addressInfoTokenizer"/>
</map>
</property>
<property name="fieldSetMappers">
```

```
<map>
<entry key="EMPLOYEE*" value-ref="employeeFieldSetMapper"/>
<entry key="BANKINFO*" value-ref="bankInfoFieldSetMapper"/>
<entry key="ADDRESSINFO*" value-ref="addressInfoFieldSetMapper"/>
</map>
</property>
</bean>
```

The `PatternMatchingCompositeLineMapper` identifies each line by its pattern, with matching keys to let the corresponding `Tokenizer` and `FieldSetMapper` read and match the records.

Exceptions from flat file reading

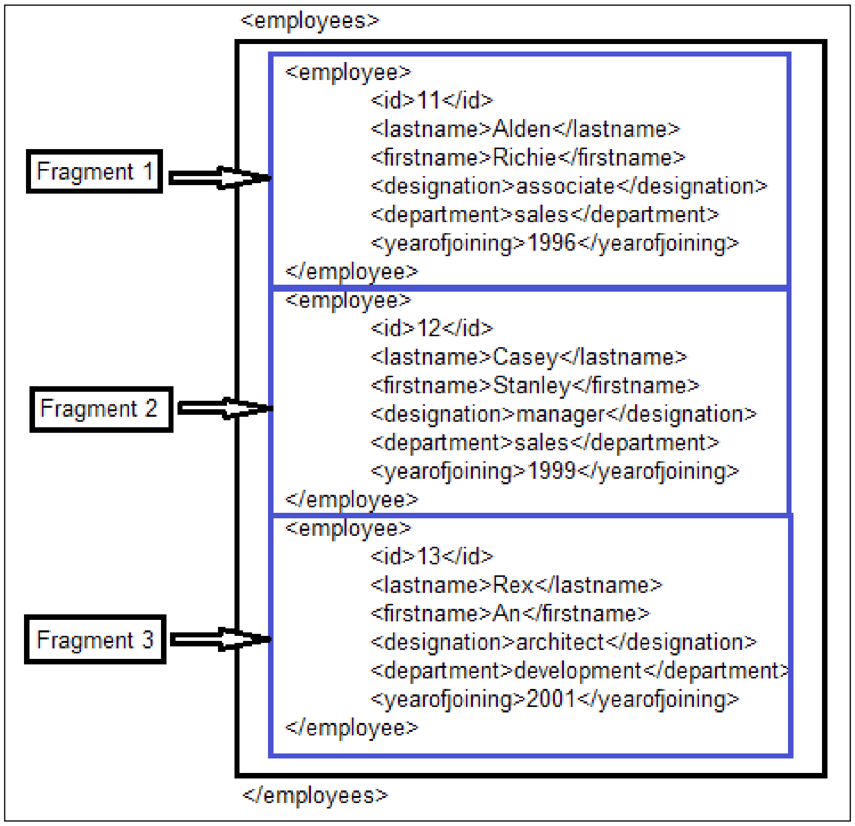
The following are the possible exceptions from flat files, for example when in case the file has an incorrect format, a problem in reading the data from file, or inconsistent data in the flat file:

- `FlatFileParseException`: This is the exception thrown by `FlatFileItemReader` for the errors that occur during file reading
- `FlatFileFormatException`: This is the exception thrown by `LineTokenizer` for the errors that occur during data tokenizing
- `IncorrectTokenCountException`: This is thrown if the number of columns specified do not match with the number of columns tokenized
- `IncorrectLineLengthException`: This is thrown during the fixed width flat file reading if the line/field lengths do not match with the ones specified

Reading data from XML

Extensible Markup Language (**XML**) is a markup language to define documents with data that can be readable by both humans and machines. XML is mainly used when multiple systems interact with each other.

Spring Batch uses the **Streaming API for XML** (**StAX**) parser. In the StAX metaphor, the programmatic entry point is a cursor that represents a point within the document. The application moves the cursor forward, 'pulling' the information from the parser as required. Hence, the reading happens in fragments of XML content from a file that is represented in the following figure:



```
<employees>
<employee>
        <id>11</id>
        <lastname>Alden</lastname>
        <firstname>Richie</firstname>
        <designation>associate</designation>
        <department>sales</department>
        <yearofjoining>1996</yearofjoining>
</employee>
<employee>
        <id>12</id>
        <lastname>Casey</lastname>
        <firstname>Stanley</firstname>
        <designation>manager</designation>
        <department>sales</department>
        <yearofjoining>1999</yearofjoining>
</employee>
<employee>
        <id>13</id>
        <lastname>Rex</lastname>
        <firstname>An</firstname>
        <designation>architect</designation>
        <department>development</department>
        <yearofjoining>2001</yearofjoining>
</employee>
</employees>
```

The `StaxItemReader` lets us parse the XML file, considering that the root element of each fragment is common (`employee` in the above example). `unmarshaller` converts the data into Java objects.

The following are the `employeeFile` and `employeeFileReader` configuration as beans:

```
<beans:bean id="employeeFile"
class="org.springframework.core.io.FileSystemResource" scope="step">
<beans:constructor-arg value="#{jobParameters[employeeFile]}"/>
</beans:bean>
<beans:bean id="employeeFileReader"
class="org.springframework.batch.item.xml.StaxEventItemReader">
```

```
<beans:property name="fragmentRootElementName" value="employee" />
<beans:property name="resource" ref="employeeFile" />
<beans:property name="unmarshaller" ref="employeeMarshaller" />
</beans:bean>
```

We can use different unmarshalling technologies, including JAXB, XStream binding, JiBX, and XML Beans. We used StAX as an engine for marshalling. Let's consider the XStream binding and the following configuration with it:

```
<bean id="employeeMarshaller"
class="org.springframework.oxm.xstream.XStreamMarshaller">
<property name="aliases">
<util:map id="aliases">
<entry key="employee"
value="batch.Employee" />
<entry key="ID" value="java.lang.Integer" />
</util:map>
</property>
</bean>
```

```
StaxEventItemReader xmlStaxEventItemReader = ;
Resource resource = ( .getBytes());
Map aliases = new HashMap();
aliases.put("employee","batch.Employee");
aliases.put("ID","java.lang.Integer");
Marshaller marshaller = newXStreamMarshaller();
marshaller.setAliases(aliases);
xmlStaxEventItemReader.setUnmarshaller(marshaller);
xmlStaxEventItemReader.setResource(resource);
xmlStaxEventItemReader.setFragmentRootElementName("employee");
xmlStaxEventItemReader.open(newExecutionContext());
boolean hasNext = true;
Employee employee = null;
while(hasNext) {
  employee = xmlStaxEventItemReader.read();
  if(employee == null) {
    hasNext = false;
  }
  else{
    System.out.println(employee.getName());
  }
}
```

If more than one file has the XML details to be read, we can use `MuliResourceItemReader` to configure multiple resources to be read in a read operation.

Reading data from a database

A database contains information in the form of tables with multiple columns to hold each field in it. If a batch job has to read the data from a database, it can be performed using the following two types of item reading concepts:

- **Cursor-based item reading**: This reads each fragment having a cursor pointing to one after the other
- **Page-based item reading**: This reads multiple records together, considering them as a page

In comparison, cursor-based item reading works well as it reads little data and processes unless their memory leaks with the system.

JdbcCursorItemReader

To read the data in a cursor-based technique, we can use `JdbcCursorItemReader`. It configures with `RowMapper` (of Spring framework) to match each attribute in the database to the Java object attributes.

The `RowMapper` for the employee example can be implemented as follows:

```
public class EmployeeRowMapper implements RowMapper {
public static final String ID_COLUMN = "id";
public static final String LAST_NAME_COLUMN = "lastname";
public static final String FIRST_NAME_COLUMN = "firstname";
public static final String DESIGNATIoN_COLUMN = "designation";
public static final String DEPARTMENT_COLUMN = "department";
public static final String YEAR_OF_JOINING_COLUMN = "yearOfJoining";

public Object mapRow(ResultSet rs, int rowNum) throws SQLException
  {
  Employee employee = new Employee();
  employee.setId(rs.getInt(ID_COLUMN));
  employee.setLastName(rs.getString(LAST_NAME_COLUMN));
  employee.setFirstName(rs.getString(FIRST_NAME_COLUMN));
  employee.setDesignation(rs.getString(DESIGNATION_COLUMN));
  employee.setDepartment(rs.getString(DEPARTMENT_COLUMN));
  employee.setYearOfJoining(rs.getString(YEAR_OF_JOINING_COLUMN));
  return employee;
  }
}
```

The Java program to read the data from the database with `EmployeeRowMapper` can be realized as follows:

```
JdbcCursorItemReader itemReader = new JdbcCursorItemReader();
itemReader.setDataSource(dataSource);
itemReader.setSql("SELECT ID, LASTNAME, FIRSTNAME,DESIGNATION,DEPARTME
NT,YEAROFJOINING from EMPLOYEE");
itemReader.setRowMapper(new EmployeeRowMapper());
int counter = 0;
ExecutionContext executionContext = new ExecutionContext();
itemReader.open(executionContext);
Object employee = newObject();
while(employee != null){
employee = itemReader.read();
counter++;
}
itemReader.close(executionContext);
```

The `JdbcCursorItemReader` and `EmployeeRowMapper` can be configured in the batch XML as follows:

```
<bean id="itemReader" class=" org.springframework.batch.item.database.
JdbcCursorItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="sql" value=" SELECT ID, LASTNAME, FIRSTNAME,DESIGNAT
ION,DEPARTMENT,YEAROFJOINING from EMPLOYEE "/>
  <property name="rowMapper">
    <bean class="batch.EmployeeRowMapper"/>
  </property>
</bean>
```

`JdbcCursorItemReader` can be customized by setting ignore warnings, fetch size, max rows, query timeout, verify cursor position, and such properties with the corresponding options for respective items.

If we want to configure the database read activity using the Hibernate framework, we can use `HibernateCursorItemReader`. The stored procedure-based read operation can be performed using the `StoredProcedureItemReader`.

JdbcPagingItemReader

The paging mode read operation on a database can be performed using the `JdbcPagingItemReader`. The configuration with the `JdbcPagingItemReader` with the properties of `dataSource`, `queryProvider`, and query with different clauses can be done as follows:

```
<bean id="itemReader" class=" JdbcPagingItemReader.
JdbcPagingItemReader">
<property name="dataSource" ref="dataSource"/>
<property name="queryProvider">
<bean class=" org.springframework.batch.item.database.support.
SqlPagingQueryProviderFactoryBean">
<property name="selectClause" value=" SELECT ID, LASTNAME, FIRSTNAME,D
ESIGNATION,DEPARTMENT,YEAROFJOINING "/>
<property name="fromClause" value="from EMPLOYEE"/>
<property name="whereClause" value="where designation=:designation"/>
<property name="sortKey" value="id"/>
</bean>
</property>
<property name="parameterValues">
<map>
<entry key="designation" value="manager"/>
</map>
</property>
<property name="pageSize" value="100"/>
<property name="rowMapper" ref="employeeMapper"/>
</bean>
```

Using the `SqlPagingQueryProviderFactoryBean`, we can set the `select`, `from`, and `where` clauses separately, along with a sortkey and parameter to be passed.

Spring Batch supports different object relational frameworks as well the corresponding item readers such as `JpaPagingItemReader` for JPA and `IbatisPagingItemReader` for IBatis.

Data processing

Spring Batch provides the means to read input data in one form, process it, and return it in a desired form of output data. The `ItemProcessor` interface is the interface that supports the processing activity.

ItemProcessor

Spring Batch provides the simple interface `ItemProcessor` to take the object, process it, and transform it to the desired form and return as another object.

The following is the definition of the `ItemProcessor` interface:

```
public interface ItemProcessor<I, O> {

O process(I item) throws Exception;

}
```

`ValidatingItemProcessor` is an implementation of the `ItemProcessor` that lets us validate the input data before processing. If the data fails to pass the validation, it gives `org.springframework.batch.item.validator.ValidationException`. Frameworks such as Hibernate have the validation framework (`hibernate-validator`) that lets us configure annotation-based validators for the beans.

The `ItemProcessor` can be implemented for the `Employee` example as follows:

```
public class Employee {}
public class Associate {
public Associate (Employee employee) {}
}
public class EmployeeProcessor implements ItemProcessor<Employee,Asso
ciate>{
public Associate process(Employee employee) throws Exception {
  return new Associate(employee);
}
}
```

The preceding program takes the `employee` data object, transforms the object, and returns an `Associate` data object.

The `itemProcessor` can be configured as a job chunk in the following format:

```
<job id="jobId">
<step name="stepName">
<tasklet>
<chunk reader="itemReaderName" processor="itemProcessorName"
writer="itemWriterName" commit-interval="2"/>
</tasklet>
</step>
</job>
```

Chaining the process

The processing activity can be chained by defining more than one item processor and calling from one another to make `compositeItemProcessor` as follows:

```
<job id="jobId">
<step name="stepName">
<tasklet>
<chunk reader="itemReaderName" processor="compositeItemProcessorName"
writer="itemWriterName"
commit-interval="2"/>
</tasklet>
</step>
</job>
      <bean id="compositeItemProcessorName"
class="org.springframework.batch.item.support.CompositeItemProcessor">
<property name="delegates">
<list>
<bean class="batch.EmployeeProcessor"/>
<bean class="batch.AssociateProcessor"/>
</list>
</property>
</bean>
```

Data writing

Spring Batch provides the configuration to write the read and processed data to a different output (destination). The writer can integrate easily with different relational frameworks. It can also be customized for the different formats.

ItemWriter

Spring Batch provides an interface in the form of `ItemWriter` to write bulk data. The following is the definition of the `ItemWriter` interface:

```
public interface ItemWriter<T> {

void write(List<? extends T> items) throws Exception;

}
```

Based on the destination platform onto which we have to write the data, we have the following item writers:

- **Flat file item writers**: These write the content onto a flat file (fixed width and delimited)
- **XML item writers**: These write the data onto an XML file
- **Database item writers**: These write the data onto a database

Flat file item writers

The data read from any of the existing formats can be processed to the desired format and then be written onto multiple formats, including flat files. The following are the APIs that help in flat file item writing.

LineAggregator

The `LineAggregator` API concatenates multiple fields into a String to write onto the flat file. This works exactly the opposite way of `LineTokenizer` in the read operation.

```
public interface LineAggregator<T> {

  public String aggregate(T item);

}
```

PassThroughLineAggregator

`PassThroughLineAggregator` is an implementation of `LineAggreagator` that considers the object in use is already aggregated and simply returns the String from the object using the `toString()` method.

```
public class PassThroughLineAggregator<T> implements LineAggregator<T>
{

  public String aggregate(T item) {

    return item.toString();

  }

}
```

The `FlatFileItemWriter` can be configured with the `PassThroughLineAggregator`, as follows:

```
<bean id="itemWriter" class=" org.springframework.batch.item.file.
FlatFileItemWriter">
<property name="resource" value="file:target/outputfiles/employee_
output.txt"/>
<property name="lineAggregator">
<bean class=" org.springframework.batch.item.file.transform.
PassThroughLineAggregator"/>
</property>
</bean>
```

FieldExtractor

If the object writing is more than just writing its String form onto the file, `FieldExtractor` needs to be used, wherein each object gets converted to the array of fields, aggregated together to form a String to write onto the file.

```
public interface FieldExtractor<T> {

  Object[] extract(T item);

}
```

Field extractors are primarily of two types:

- `PassThroughFieldExtractor`: For the scenario where the object collection has to just be converted to the array and passed to write
- `BeanWrapperFieldExtractor`: With a field-level configuration of how each field of the object gets placed in the String to write onto the file, this works exactly the opposite way of `BeanWrapperFieldSetMapper`

The `BeanWrapperFieldSetExtractor` works as follows:

```
BeanWrapperFieldExtractor<Employee> extractor = new BeanWrapperFieldEx
tractor<Employee>();
extractor.setEmployees(new String[] { "id", "lastname", "firstname","d
esignation","department","yearofjoining"});
int id = 11;
String lastname = "Alden";
String firstname = "Richie";
String desination = "associate";
String department = "sales";
int yearofjoining = 1996;
```

```
Employee employee = new Employee(id, lastname, firstname,designation,
department, yearofjoining);
Object[] values = extractor.extract(n);
assertEquals(id, values[0]);
assertEquals(lastname, values[1]);
assertEquals(firstname, values[2]);
assertEquals(designation, values[3]);
assertEquals(department, values[4]);
assertEquals(yearofjoining, values[5]);
```

Writing delimited files

If the Java object can be written onto the flat files in delimited file format, we can perform it as shown in the following example. Let's consider the `Employee` object defined already.

This object can be configured with the `FlatFileItemWriter`, the `DelimitedLineAggregator`, and the `BeanWrapperFieldExtractor` to perform the delimited flat file, as follows:

```
<bean id="itemWriter" class="org.springframework.batch.item.file.
FlatFileItemWriter">
<property name="resource" ref="outputResource"/>
<property name="lineAggregator">
<bean class=" org.springframework.batch.item.file.transform.
DelimitedLineAggregator">
<property name="delimiter" value=","/>
<property name="fieldExtractor">
<bean class=" org.springframework.batch.item.file.transform.
BeanWrapperFieldExtractor">
<property name="employees"
value="id,lastname,firstname,designation,department,yearofjoining"/>
</bean>
</property>
</bean>
</property>
</bean>
```

Writing a fixed width file

Spring Batch supports fixed width file writing with the help of `FormatterLineAggregator`. Considering the same example data as delimited flat file writing, we can perform the fixed width file writing using the following configuration:

```
<bean id="itemWriter" class="org.springframework.batch.item.file.
FlatFileItemWriter">
<property name="resource" ref="outputResource"/>
<property name="lineAggregator">
<bean class=" org.springframework.batch.item.file.transform.
FormatterLineAggregator">
<property name="fieldExtractor">
<bean class=" org.springframework.batch.item.file.transform.
BeanWrapperFieldExtractor">
<property name="employees" value=" id,lastname,firstname,designation,d
epartment,yearofjoining"/>
</bean>
</property>
<property name="format" value="%-2d%-10s%-10s%-10s%-15s%-4d"/>
</bean>
</property>
</bean>
```

The format value is formed based on the following listed formatter conversions, where `arg` represents the argument for conversion:

Conversion	Category	Description
`b`, `B`	general	This converts Boolean to the String format. The value is `false` for `null`
`h`, `H`	general	This is the `Integer.toHexString(arg.hashCode())`
`s`, `S`	general	If `arg` implements `Formattable`, then `arg.formatTo()` Otherwise, `arg.toString()`
`c`, `C`	character	This is a Unicode character
`d`	integral	This is a decimal integer
`o`	integral	This is an octal integer
`x`, `X`	integral	This is a hexadecimal integer
`e`, `E`	floating point	This is a decimal number in computerized scientific notation
`f`	floating point	This is a decimal number

Conversion	Category	Description
`g, G`	floating point	This is a computerized scientific notation or decimal format, depending on the precision and value after rounding
`a, A`	floating point	This is a hexadecimal floating point number with a significand and an exponent
`t, T`	date/time	This is the prefix for date and time conversion characters
`%`	percent	This is a literal `%` (`\u0025`)
`n`	line separator	This is the platform-specific line separator

`FlatFileItemWriter` can be configured with the `shouldDeleteIfExists` option, to delete a file if it already exists in the specified location. The header and footer can be added to the flat file by implementing `FlatFileHeaderCallBack` and `FlatFileFooterCallBack` and including these beans with the `headerCallback` and `footerCallback` properties respectively.

XML item writers

The data can be written to the **Extensible Markup Language** (**XML**) format using `StaxEventItemWriter`. The Spring Batch configuration for this activity, for the employee example can be the following:

```
<bean id="itemWriter" class="org.springframework.batch.item.xml.
StaxEventItemWriter">
  <property name="resource" ref="outputResource"/>
  <property name="marshaller" ref="employeeMarshaller"/>
  <property name="rootTagName" value="employees"/>
  <property name="overwriteOutput" value="true"/>
</bean>
```

Using the XStream to do the marshalling activity, the following is the configuration:

```
<bean id="employeeMarshaller"
class="org.springframework.oxm.xstream.XStreamMarshaller">
<property name="aliases">
<util:map id="aliases">
<entry key="employee"
value="batch.Employee"/>
<entry key="ID" value="java.lang.Integer"/>
</util:map>
</property>
</bean>
```

The Java code for the preceding configuration can be realized as follows:

```
StaxEventItemWriter staxItemWriter = newStaxEventItemWriter();
FileSystemResource resource = new FileSystemResource("export/employee_
output.xml")
Map aliases = newHashMap();
aliases.put("employee","batch.Employee");
aliases.put("ID","java.lang.Integer");
Marshaller marshaller = newXStreamMarshaller();
marshaller.setAliases(aliases);
staxItemWriter.setResource(resource);
staxItemWriter.setMarshaller(marshaller);
staxItemWriter.setRootTagName("employees");
staxItemWriter.setOverwriteOutput(true);
ExecutionContext executionContext = newExecutionContext();
staxItemWriter.open(executionContext);
Employee employee = new Employee();
employee.setID(11);
employee.setLastName("Alden");
employee.setFirstName("Richie");
employee.setDesignation("associate");
employee.setDepartment("sales");
employee.setYearOfJoining("1996");
staxItemWriter.write(employee);
```

Database item writers

Spring Batch supports database item writing with two possible access types: JDBC and ORM.

JDBC-based database writing

Spring Batch supports JDBC-based database writing with the help of `JdbcBatchItemWriter`, which is an implementation of `ItemWriter`, which executes multiple SQL statements in the batch mode. The following is the sample configuration for the employee example with the JDBC-based database writing:

```
<bean id="employeeWriter" class="org.springframework.batch.item.
database.JdbcBatchItemWriter">
<property name="assertUpdates" value="true" />
<property name="itemPreparedStatementSetter">
<bean class="batch.EmployeePreparedStatementSetter" />
</property>
<property name="sql"
```

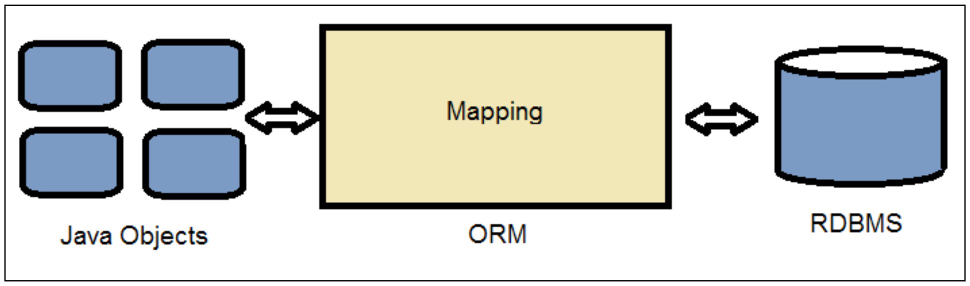
```
value="INSERT INTO EMPLOYEE (ID, LASTNAME, FIRSTNAME, DESIGNATION,
DEPARTMENT, YEAROFJOINING) VALUES(?, ?, ?, ?, ?, ?)" />
<property name="dataSource" ref="dataSource" />
</bean>
```

The `ItemPreparedStatementSetter` can be implemented for our example of `Employee` data as follows:

```
public class EmployeePreparedStatementSetter
implements ItemPreparedStatementSetter<Employee> {

@Override
public void setValues(Employee item, PreparedStatement ps) throws
SQLException {
  ps.setInt(1, item.getId());
  ps.setString(2, item.getLastName());
  ps.setString(3, item.getFirstName());
  ps.setString(4, item.getDesignation());
  ps.setString(5, item.getDepartment());
  ps.setInt(6, item.getYearOfJoining());
}
}
```

ORM-based database writing

Object relational mapping (**ORM**) is defined as a programming technique to convert data between incompatible type systems in object-oriented programming languages. ORM takes care of the data persistence from the object oriented program to the database. Spring Batch supports multiple ORMs including Hibernate, JPA, and iBatis.



In our example, the `Employee` class should be annotated to be used with ORM (Hibernate/JPA) for persistence as follows:

```
@Entity("employee")
public class Employee {
@Id("id")
private int id;
@Column("lastName")
private String lastName;
@Column("firstName")
private String firstName;
@Column("designation")
private String designation;
@Column("department")
private String department;
@Column("yearOfJoining")
private int yearOfJoining;

public int getID() {
  return id;
}
public void setID(int id) {
  this.id = id;
}
public String getLastName() {
  return lastName;
}
public void setLastName(String lastName) {
  this.lastName = lastName;
}
public String getFirstName() {
  return firstName;
}
public void setFirstName(String firstName) {
  this.firstName = firstName;
}
public String getDesignation() {
  return designation;
}
public void setDesignation(String designation) {
  this.designation = designation;
}
public String getDepartment() {
  return department;
}
public void setDepartment(String department) {
  this.department = department;
}
public int getYearOfJoining() {
  return yearOfJoining;
}
public void setYearOfJoining(int yearOfJoining) {
  this.yearOfJoining = yearOfJoining;
}
}
```

The annotations specify that the class `Employee` is representing a corresponding table in the database with a name as shown with `@Entity`, and each field corresponds to a column in the database as shown with the `@ID` and `@Column` annotations.

The following is the configuration to be made with Hibernate for the employee example:

```
<bean id="employeeWriter"
class="org.springframework.batch.item.database.HibernateItemWriter">
<property name="hibernateTemplate" ref="hibernateTemplate" />
</bean>
```

Similarly, for JPA and iBatis, the configurations can be made with `JpaItemWriter` and `IbatisBatchItemWriter` respectively.

Custom item readers and writers

Spring Batch supports custom item readers' and writers' configurations. This can be done easily by implementing the `ItemReader` and `ItemWriter` interfaces for the respective read and write operations with the business logic we want, and configuring the `ItemReader` and `ItemWriter` in the XML batch configuration.

Summary

Through this chapter we learned the essential data handling mechanisms, including reading the data from different sources (such as flat files, XML, and databases), processing the data, and writing the data to different destinations including flat files, XML, and databases. We also learned about transforming and validating the data in the processing data section. We finished this chapter with an understanding of the Spring Batch support to custom formats by implementing the interface to match the business needs that are different from the default formats. In the next chapter, we will learn about managing the transactions with diverse configurations and patterns in detail.

4
Handling Job Transactions

In the previous chapter, we learned about essential data handling mechanisms, including reading, processing, and writing data from/to different sources, such as flat files, XML, and databases. From the previous chapters, we learned that Spring Batch jobs handle bulk data reading, manipulating, and writing activities. Through these activities, it is important to make the activity consistent through a transaction while interacting with files/databases. Spring Batch provides strong transaction support through job processing.

In this chapter, we will cover the following topics:

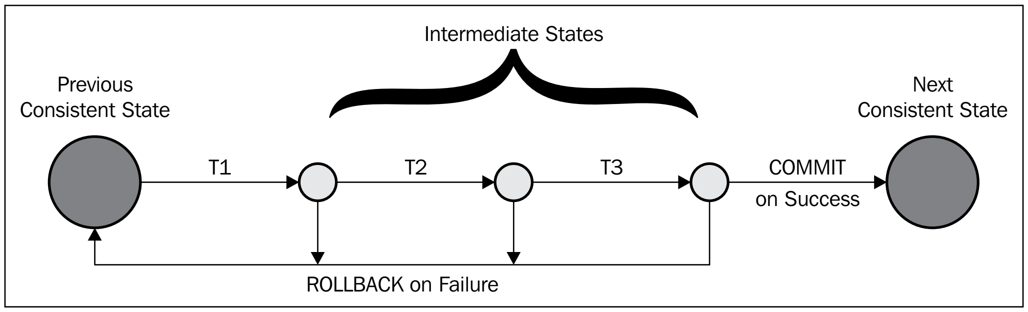
- Transactions
- Spring Batch transaction management
- Customizing the transaction
- Transaction patterns

Transactions

As part of the job processing, activities involve reading data from different sources, processing the data, and writing it to different sources, including files and databases. Data, as a complete set of records or in chunks, has to either be completely processed and written to the end system, or be tracked as failed records in the case of any error. Transaction management should take care of this operation to make it consistent, by committing the correct information and rolling back in case of any error. The following are the activities involved in a database transaction:

- Beginning the transaction
- Processing a set of records
- Committing the transaction if no errors occur during processing

- Rolling back the transaction if any errors occur during processing



Hence, a transaction is defined as a series of operations that obey the **atomic, consistent, isolated, and durable** (**ACID**) characteristics described as follows:

- **Atomic**: This ensures success in either all or none of the operations in the transaction
- **Consistent**: This ensures that the transaction brings the resource from one valid state to the other valid state
- **Isolated**: One transaction's state and effect are hidden from all other transactions during concurrent execution
- **Durable**: The result of a transaction should be persistent and survive a system crash once the transaction is completed

If a transaction follows these ACID characteristics, it can handle any unexpected errors, by aborting the error that occurs during the transaction, to gracefully recover the consistent state of the system.

Spring Batch transaction management

Spring Batch provides transaction management through step execution, where each transaction is committed after successful data processing and is rolled back if any error is found through processing.

Spring Batch manages the transactions in either of the following cases:

- Tasklet steps
- Chunk-oriented steps
- Listeners

Tasklet steps

Tasklets are used in Spring Batch to process business-specific activities, such as archiving, remote interactions, and invoking services. By default, the `execute` method of the tasklet itself is transactional. Hence, each call to the `execute` method calls for a new transaction. The following is a sample tasklet configuration:

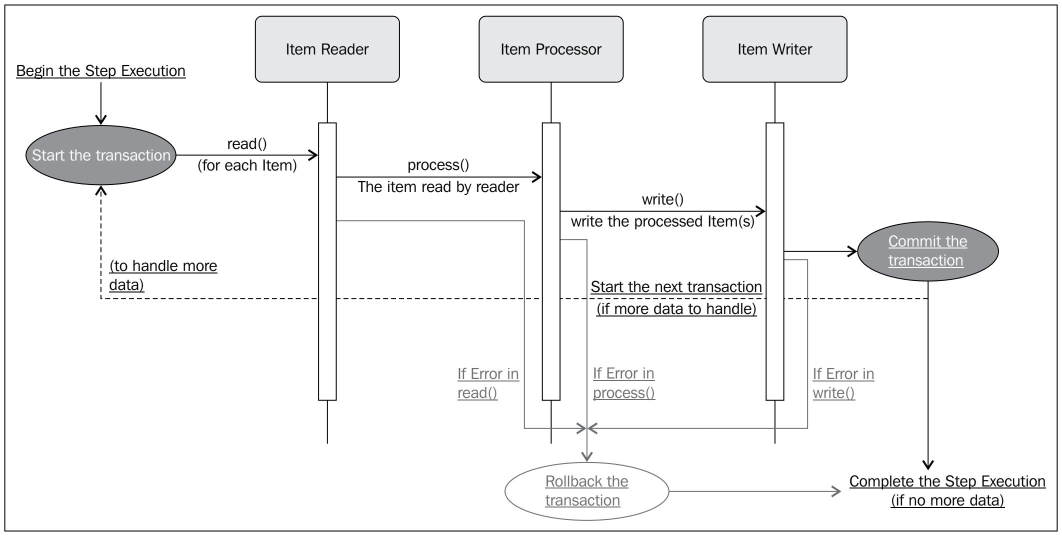
```
<step id="stepOne">
<tasklet ref="myTasklet"/>
</step>
<bean id="myTasklet" class="batch.MyTasklet">
<property name="targetObject">
<bean class="batch.EmployeeData"/>
</property>
<property name="targetMethod" value="updateEmployeeDetails"/>
</bean>
```

The implementation of the tasklet can be as follows:

```
public class MyTasklet implements Tasklet {
@Override
publicRepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throwsException {
...
returnRepeatStatus.FINISHED;
}
}
```

Chunk-oriented steps

Chunk-oriented steps processing undertakes chunks of records in a read-process-write mechanism, as shown in the following diagram. Each step, once the transaction is started, handles the data to be read, processed, and written, and after the successful completion of these stages, the step commits the transaction. Then, it follows the next transaction to handle the next set of records. If any error occurs in either of these steps, it rolls back the transaction and completes the step execution.



Hence, chunk-oriented steps are preferred for bulk data processing so that the entire data is divided into chunks and processed in individual transactions. If any exception occurs in any phase, it rolls back that transaction, hence, the data handling will be more efficient and complete. Failed steps, which can be logged and re-run with corrected information, are gracefully rolled back. The following is a sample chunk-oriented step configuration:

```
<step id="stepOne">
<tasklet allow-start-if-complete="true">
<chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>
</tasklet>
</step>

<step id="stepTwo" parent="stepOne">
<tasklet start-limit="5">
<chunk processor="itemProcessor" commit-interval="5"/>
</tasklet>
</step>
```

In the preceding configuration, we have the first step (`stepOne`) with tasklet and chunk within it. To ensure the order of execution, `stepOne` is the parent of `stepTwo`. While `stepOne` configures `itemReader` and `itemWriter`, `stepTwo` configures `itemProcessor`.

Listeners

Spring Batch supports listeners to execute certain operations after/before an event occurs. Spring Batch handles each of these events, and the way transactions are handled in each listener is specific to that listener and how they handle the data. Hence, it is important to observe if the listener methods are handled as part of the step transaction. If not, then the application should handle such transactions programmatically. The following is the sample listener configuration:

```
<bean id="myStepExecutionListener"
      class="org.java.MyStepExecutionListener" />
  <job id="readEmployeeDetails">
    <step id="step1">
      <tasklet>
        <chunk reader="itemReader" writer="itemWriter"
            commit-interval="1" />
          <listeners>
            <listener ref="myStepExecutionListener" />
          </listeners>
      </tasklet>
    </step>
  </job>
```

The implementation of the listener can be as follows:

```
public class MyStepExecutionListener implements StepExecutionListener
{

  @Override
  public void beforeStep(StepExecution stepExecution) {
    System.out.println("StepExecutionListener : beforeStep");
  }

  @Override
  public ExitStatus afterStep(StepExecution stepExecution) {
    System.out.println("StepExecutionListener : afterStep");
    return null;
  }

}
```

Customizing the transaction

Spring Batch allows the configurations to customize the way transactions are handled. The data exchange between different transactions, if processed and read gracefully, make the transaction clean. However, we have different ways to configure the visibility of transaction integrity to other interactions, called isolation levels. The following are the isolation levels to customize the Spring Batch transactions:

- **Serializable**: This is the highest isolation level. Based on lock-based or non-lock-based concurrency control, it ensures clean data reading.
- **Repeatable reads**: This lock-based implementation maintains read and write locks, hence clean data is guaranteed; however, with no support for range locks, phantom reads may occur.
- **Read committed**: This lock-based implementation maintains the write lock, hence it promises any data read is committed the moment it is read, and restricts any intermediate, uncommitted, and dirty read.
- **Read uncommitted**: This is the lowest isolation level. One transaction can see the uncommitted changes of other transactions as well. Hence, dirty reads are allowed in this level.

There are predefined constants for each isolation level. By default, the configuration is `READ_COMMITTED` for the Spring Batch isolation level. Based on the criticality and importance of the data to be read across transactions, one has to set the isolation level for that transaction. For example, a bank account transaction might want to read only clean, committed data from other transactions and make the transaction with persistent data. In such cases, one has to choose between the isolation level and performance of the application. The following is a sample isolation level configuration:

```
<job-repository id="jobRepository" isolation-level-for-
create="SERIALIZABLE"/>
```

If the batch jobs are integrated with applications having other frameworks using similar nomenclature for configurations, then one should be careful with configurations. The intent of control in one technology can control others; it's better to disable either one of the configurations on need basis in such scenarios.

The attributes on each of the batch components, as discussed in *Chapter 2*, *Getting Started with Spring Batch Jobs*, can help us better customize the batch job transaction configurations. For example, one can control the rollback transactions, in specific exception scenarios, by configuring `no-rollback-exception-classes`.

Transaction patterns

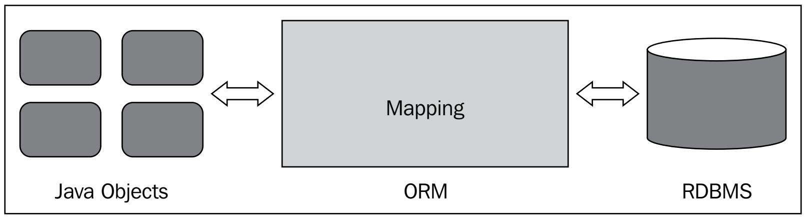
Spring Batch job processing involves handling data across multiple sources. Such scenarios, which usually occur time and again, can be identified as transaction patterns.

The following are the patterns identified:

- **Simple transaction**: This is a transaction with a single application and data source (source and target)
- **Global transaction**: This is a transaction involving more than one data source to be handled through the same transaction

Simple transaction

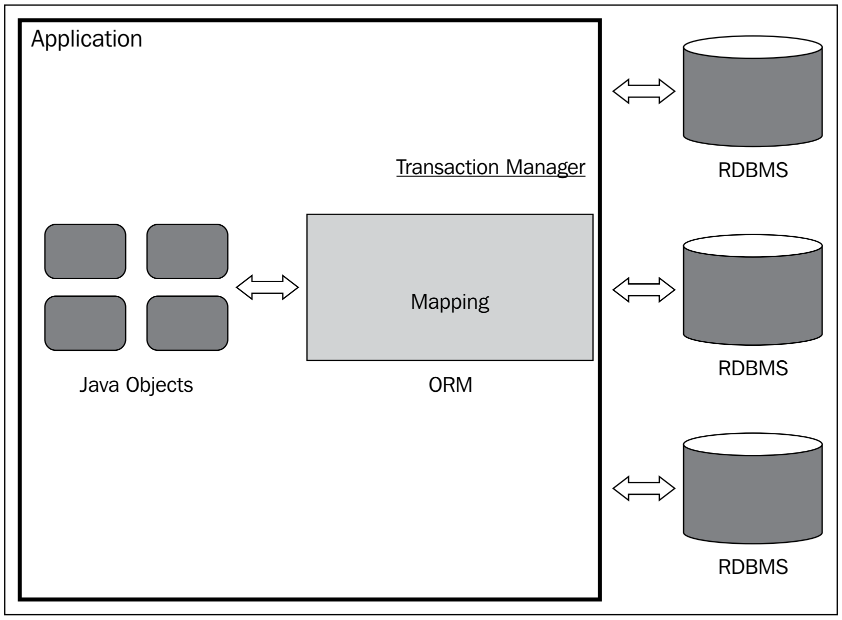
Simple transactions with a single batch application and a data source can be easily implemented with the support of Spring Batch integrated with diverse database interaction techniques, such as JDBC, JPA, or Hibernate supports from Spring Batch. The interaction would be as shown in the following figure:



Global transaction

If more than one data source has to be persisted through a single transaction, such transactions are termed as global transactions, which can managed by the transaction manager. It is the responsibility of the transaction manager to make sure the transaction obeys the ACID characteristics through its multiple data sources and the data is persisted consistently.

However, if an application is deployed in an integrated enterprise server, which supports a transaction manager, that might as well be considered against the **Java Transaction API** (**JTA**) based transaction manager. The following is a representation of a managed transaction.



These transactions can also be configured to maintain the references of the one database schema as a synonym in another database schema to refer it virtually as a local transaction. However, the effort of creating such synonyms must be considered.

Summary

Through this chapter we learned about transactions and key characteristics of transactions. We also learned how Spring Batch performs transaction management in different scenarios, including tasklet steps, chunk-oriented steps, and listeners. We also learned about customizing the transaction with isolation levels and attribute configurations. We finished this chapter with an understanding of commonly used transaction patterns with single and multiple data sources in a batch application.

In the next chapter, we will learn in detail about the flow of jobs and sharing data between steps of the executing jobs.

5
Step Execution

In the previous chapter, we learned about transactions and managing the transactions in different scenarios, along with customizing the transactions with isolation levels, and attribute configurations for single and multiple data sources with patterns. So far, we have discussed the simple jobs, where the flow is linear and contains jobs with steps executing one after the other. In real world applications, we need to configure jobs with a combination of steps, sharing data between them and deciding which step to execute at runtime.

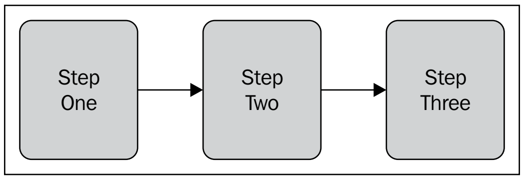
In this chapter, we will cover the following topics:

- Controlling the job flow
- Data sharing
- Externalization and termination

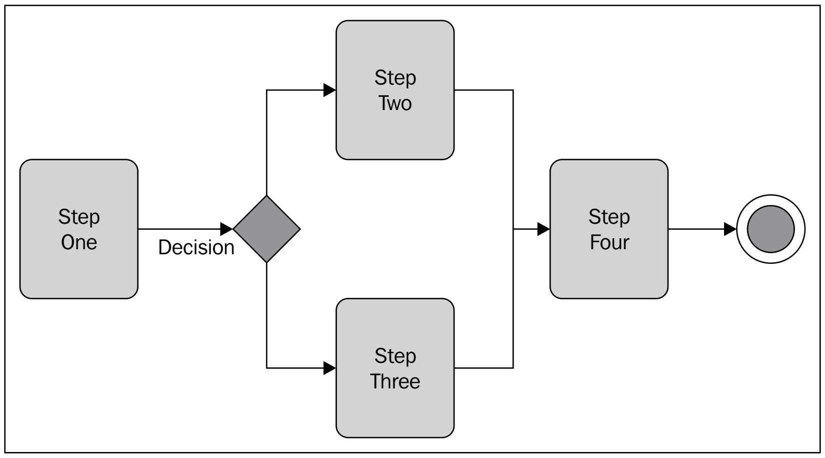
Controlling the job flow

So far we have seen batch jobs configured with steps executing consecutively in a linear fashion. There could be scenarios to decide which step to execute based on the outcome of the previous step during the execution of a batch job, which is a nonlinear execution.

The following figure shows how the linear step execution happens in a batch job:



The following figure shows how the nonlinear step execution happens in a batch job:



Let's understand how to handle such a job flow. There are primarily two ways to handle it:

- Using an exit code
- Using a decision logic

Using an exit code

Job flow can be handled based on the exit status of a step along with the configuration of the `next` tag with the `on` and `to` properties. The following is a sample configuration using an exit code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns ="http://www.springframework.org/schema/batch"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch-3.0.xsd">

<beans:import resource="context.xml" />

<beans:bean id="testTasklet" class=" batch.TestTasklet">
<beans:property name="success" value="true"/>
</beans:bean>
<beans:bean id="successTasklet" class=" batch.StatusTasklet">
```

```
<beans:property name="status" value="Success"/>
</beans:bean>
<beans:bean id="failTasklet" class=" batch.StatusTasklet">
<beans:property name="status" value="Failure"/>
</beans:bean>

<job id="nonLinearJob">
<step id="stepOne">
  <tasklet ref="testTasklet"/>
  <next on="*" to="stepTwo"/>
  <next on="FAILED" to="stepThree"/>
</step>
<step id="stepTwo">
  <tasklet ref="successTasklet"/>
</step>
<step id="stepThree">
  <tasklet ref="failTasklet"/>
</step>
</job>
</beans:beans>
```

In the preceding configuration, `stepOne` is the first step to be executed in the batch job. Based on the output (`ExitStatus`) of this step, which includes `testTasklet`, the next tag decides which step to execute. If `testTasklet` returns a `FAILED` status, it executes `stepThree`, otherwise `stepTwo`. The status can be returned either by the attribute of job execution or step execution. The following are the different statuses:

- String: The exit status should match with a String, for example, `COMPLETED`/`FAILED`, which can be verified from `FlowExecutionStatus`.
- `*`: This matches with zero or more characters. It matches any value.
- `?`: This matches only one character.

Using a decision logic

The nonlinear job execution can also be handled with the decision logic using the implementation of `JobExecutionDecider` and decision tag configuration.

The following is the `JobExecutionDecider` implementation to check the exit status and return `FlowExecutionStatus` accordingly:

```
package batch;
import org.springframework.batch.core.ExitStatus;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.job.flow.FlowExecutionStatus;
import org.springframework.batch.core.job.flow.JobExecutionDecider;

public class JobFlowDecider implements JobExecutionDecider {
@Override
public FlowExecutionStatus decide(JobExecution jobExecution,
StepExecution stepExecution) {
  if(!ExitStatus.FAILED.equals(stepExecution.getExitStatus())) {
    return new FlowExecutionStatus(FlowExecutionStatus.FAILED.
getName());
  } else {
    return new FlowExecutionStatus(FlowExecutionStatus.COMPLETED.
getName());
  }
  }
}
```

The following is the job configuration with the `JobExecutionDecider` implementation and decision tag configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns ="http://www.springframework.org/schema/batch"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch-3.0.xsd">

<beans:bean id="decider" class="batch.JobFlowDecider"/>
<beans:bean id="successTasklet" class=" batch.StatusTasklet">
<beans:property name="status" value="Success"/>
</beans:bean>
<beans:bean id="failTasklet" class=" batch.StatusTasklet">
<beans:property name="status" value="Failure"/>
</beans:bean>
<job id="nonLinearJob">
  <step id="stepOne" next="decision">
  <tasklet>
    <chunk reader="itemReader" writer="itemWriter"
    commit-interval="20"/>
  </tasklet>
  </step>
  <decision id="decision"> decider="decider"
    <next on="*" to="stepTwo"/>
```

```
      <next on="FAILED" to="stepThree"/>
    </decision>
    <step id="stepTwo">
      <tasklet ref="successTasklet"/>
    </step>
    <step id="stepThree">
      <tasklet ref="failTasklet"/>
    </step>
  </job>
```

One can choose between these two options (exit codes and decision logic) based on the monitoring status needs; the job execution decider makes the configuration more readable.

Data sharing

While each step should be configured to execute on its own in an ideal scenario, the steps need to share the data in real-world scenarios. The data can be shared between steps in different ways. The following are the options:

- Using execution context
- Using Spring holder beans

Using execution context

We learned from the previous chapters that the Spring Batch jobs maintain information about the job execution in a context called batch job metadata. We can use this context to share data between steps. The key-value-based data is maintained by `org.springframework.batch.item.ExecutionContext` in its usage. The following is the way to put/get the data from it:

```
String importId = jobExecutionContext.getString("importId");
executionContext.putString("importId", importId);
```

Both job and step have their own execution context in the form of `JobExecutionContext` and `StepExecutionContext`. While jobs have a unique execution context, each step in a job maintains its own step execution context. Step context can be accessed from the chunk context (`org.springframework.batch.core.scope.context.ChunkContext`) and the job context can be accessed from the step context.

Using Spring holder beans

The data between steps can be shared using the concept of Spring holder beans as well. Metadata configuration is represented by `ImportMetadata`, through which we can set and get the data. We can write a bean that can hold the reference of `ImportMetadata` and configure the same as `ImportMetadataHolder` in job configuration. The following is the sample configuration for `ImportMetadataHolder`:

```
package batch;
public class ImportMetadataHolder {
  private ImportMetadata importMetadata;
  public ImportMetadata get() {
    return importMetadata;
  }
  public void set(ImportMetadata importMetadata) {
    this.importMetadata = importMetadata;
  }
}
```

The data can be set and got from the holder by using following syntax:

```
importMetadataHolder.set(
batchService.extractMetadata(outputDirectory));
importMetadataHolder.get().getImportId();
```

The `ImportMetadataHolder` can be configured just like any other bean and injected into the tasklets with the property specification.

Externalization and termination

Externalization and termination are the concepts that help to make reusable components of Spring Batch and handle job termination graciously.

Externalization

Spring Batch allows code reuse using externalization, the concept of separating the reusable steps of operation and including them in desired jobs. Along with the configuration of the individual steps as beans and including them in each job, externalization can be achieved in the following ways:

- External flow definition and including it in desired jobs
- Inherited jobs mechanism

External flow definition and including it in desired jobs

The following is a sample configuration for external flow definition and including it in the desired job:

```
<flow id="externalFlow">
  <step id="stepOne" next="stepTwo">
    <tasklet ref="taskletOne"/>
  </step>
  <step id="stepTwo">
    <tasklet ref="taskletTwo"/>
  </step>
</flow>
<job id="mainJob">
  <flow parent="externalFlow" id="mainStep" next="stepThree"/>
  <step id="stepThree">
    <tasklet ref="taskletThree"/>
  </step>
</job>
```

Inherited jobs mechanism

The other way of externalizing the process is by inheriting one job into the other, which means defining an independent job and referring to it in another job as a part of it. The following is a sample configuration for it:

```
<job id="mainJob">
  <step id="stepOne" next="stepTwo">
    <tasklet ref="taskletOne"/>
  </step>
  <step id="stepTwo">
    <tasklet ref="taskletTwo"/>
  </step>
</job>
<job id="subJob">
  <step id="stepThree" next="stepFour">
<job ref="mainJob" job-parameters-extractor="jobParametersExtractor"
/>
</step>
  <step id="stepFour" parent="runBatch"/>
</job>
```

The main job has a couple of steps and the sub job is defined to refer to the main job as a part of its first step.

Termination

Ending the execution programmatically is an important aspect of the batch job execution. To be able to effectively program this, one should be aware of the different states in which the job can be terminated. The different states are as follows:

- `COMPLETED`: This end state can be used to tell Spring Batch that the processing has ended successfully. When a job instance is terminated with this end state, it isn't allowed to rerun with the same set of parameters.
- `FAILED`: This end state can be used to tell Spring Batch that the processing has failed. Spring Batch lets the failed jobs rerun with the same set of parameters.
- `STOPPED`: This end state is like pausing an executing job. If ended with this state, Spring Batch not only lets us restart the job, it also lets us restart from where it left off, even though there are no errors in execution.

Terminating in the COMPLETED state

The following is the configuration to terminate a job in the `COMPLETED` state, based on the `ExitStatus` with the end tag configuration:

```
<job id="nonLinearJob">
  <step id="stepOne">
    <tasklet ref="successTasklet"/>
    <end on="*"/>
    <next on="FAILED" to="stepTwo"/>
  </step>
  <step id="stepTwo">
    <tasklet ref="failureTasklet"/>
  </step>
</job>
```

This configuration ends the job after successful execution, and we can't rerun the job with the same set of parameters. The first step is configured to invoke the second step if it has failed.

Terminating in the FAILED state

The following is the configuration to terminate a job in the `FAILED` state, based on the `ExitStatus` with the fail tag configuration:

```
<job id="nonLinearJob">
  <step id="stepOne">
    <tasklet ref="successTasklet"/>
```

```
        <next on="*" to="stepTwo"/>
        <fail on="FAILED" exit-code="STEP-ONE-FAILED"/>
      </step>
      <step id="stepTwo">
        <tasklet ref="failureTasklet"/>
      </step>
    </job>
```

This configuration ends the job with the `FAILED` state if the exit status is `FAILED`, and we can rerun the job with the same set of parameters.

Terminating in the STOPPED state

The following is the configuration to terminate a job in the `STOPPED` state, based on the `ExitStatus` with the stop tag configuration:

```
    <job id="nonLinearJob">
      <step id="stepOne">
        <tasklet ref="successTasklet"/>
        <next on="*" to="stepTwo"/>
        <stop on="FAILED" restart="stepTwo"/>
      </step>
      <step id="stepTwo">
        <tasklet ref="failureTasklet"/>
      </step>
    </job>
```

This configuration ends the job with the `STOPPED` state if the exit status is `FAILED`, and we can rerun the job with the same set of parameters.

Summary

Through this chapter, we learned about controlling the flow of a batch job using exit codes and decision logic. We also learned how to share data between the steps in execution with the help of execution context and holder beans. We also learned about reusing the process by externalizing the flow and inherited job mechanisms. We finished this chapter with an understanding of terminating the batch job in different states and their importance. In the next chapter, we will learn in detail about the enterprise integration using Spring integration and RESTful job processing.

6
Integrating Spring Batch

In the previous chapter, we learned about controlling the flow of a batch job using exit code and decision logic, sharing the data between the steps in execution, and reusing the process by externalizing the flow and inherited job mechanisms. We also learned how to terminate the batch job in different states and their importance. An organization performs its operations with the help of a number of tools and maintains its data and applications across locations. It is important to integrate the data across these applications with a decent mechanism to synchronize the systems.

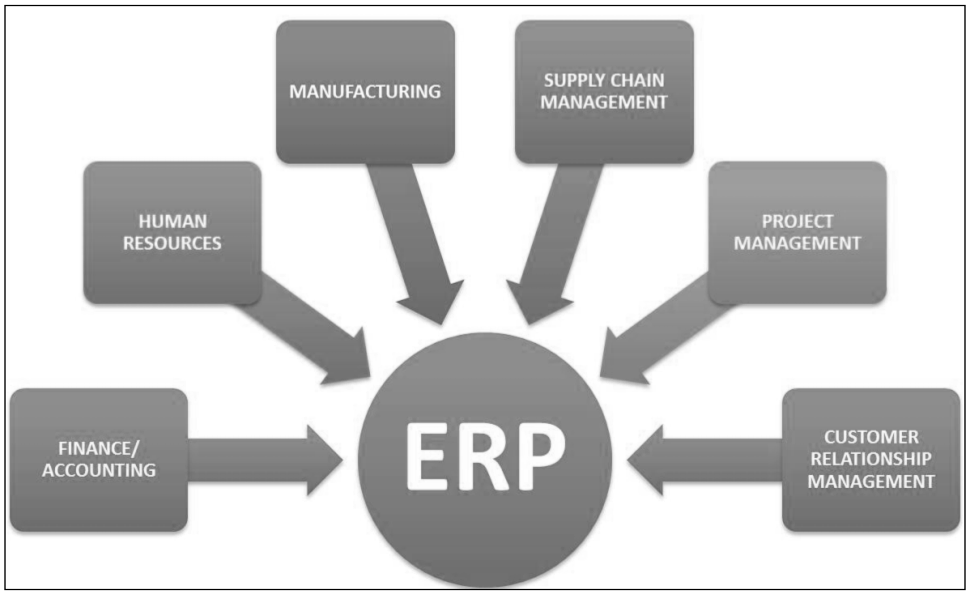
In this chapter, we will cover the following topics:

- Enterprise Integration
- Spring Integration
- RESTful job processing

Enterprise Integration

So far we have seen batch jobs configured with different steps; reading data from different sources, performing operations, and writing data to different destinations. In real time, organizations use different applications to perform their operations. The application used to maintain the employee information and process their payroll might not be the same as the one that takes care of logistics and sales. In such scenarios, it is important to integrate these applications seamlessly to process the whole data together at any particular point and perform an operation on the system.

The following figure shows how an **Enterprise Resource Planning** (**ERP**) system integrates different modules, accesses the information from its systems, and maintains it as an entity.

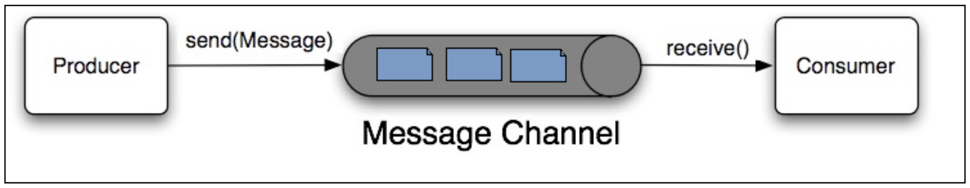


The following are the different ways to integrate enterprise applications:

- **File-based data transfer**: Applications exchange data based on flat files; the source system writes data onto a flat file and exports the file to the destination system. The destination system reads data from a flat file and imports into its destination database.
- **Resource sharing**: Applications share common resources, such as a filesystem or a database, to perform their operations. Virtually, they act as individual systems; however, they populate/write data onto a common system.
- **Service invocation**: Applications expose their operations as services (web services in recent days) to let other applications call them. One can transfer/receive data from such services, depending on the way they are designed.
- **Messaging services**: Applications use a common messaging server; one application can send a message and the other receives it.

Spring Integration

Spring project defines Spring Integration as an extension of the Spring programming model for Enterprise Integration. Spring Integration is developed to support lightweight messaging within Spring-based applications and supports system integration with external systems through declarative adapters. These adapters provide an abstraction of Spring's support for remoting, messaging, and scheduling.



While Spring Batch operates on a file- or database-based integration system, Spring Integration provides the application's message-based integration. Adding this messaging feature to the Spring Batch application automates its operations and also separates the key operational concerns. Let's understand how we can make Spring Integration configuration be a part of the integrated enterprise application. The following are some of the key operations that can be performed with message integration:

- Triggering a batch job to execute
- Triggering a message with the job completion/fail status
- Asynchronous processor's operation
- Externalization

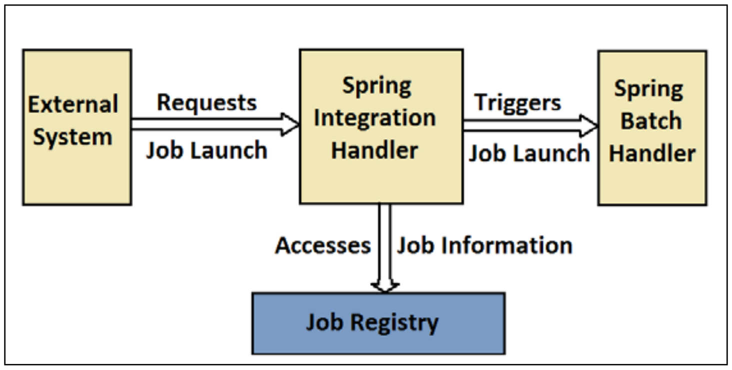
The following is the Spring XML application context file with Spring Batch integration enabled:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:integrate="http://www.springframework.org/schema/integration"
xmlns:batch="http://www.springframework.org/schema/batch"
xmlns:batch-integrate="http://www.springframework.org/schema/batch-
integration"
xsi:schemaLocation="http://www.springframework.org/schema/batch-
integration
http://www.springframework.org/schema/batch-integration/spring-batch-
integration.xsd
http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch.xsd
http://www.springframework.org/schema/beans
```

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

Triggering a batch job to execute

So far we have been triggering jobs through command line, programmatically from applications. However, certain applications have to transfer data using remote file transfer (FTP/SFTP) and launch jobs to import data into the application. Spring Integration provides different adapters to easily make launch configurations. `JobLaunchingMessageHandler` of Spring Integration is an easy-to-implement, event-driven execution on `JobLauncher`. Spring Integration provides `JobLaunchRequest` as the input for the `JobLaunchingMessageHandler`.



The following is the listing for `JobLaunchRequest` transformation from a file:

```
package com.java.batchJob;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.integration.launch.JobLaunchRequest;
import org.springframework.integration.annotation.Transformer;
import org.springframework.messaging.Message;
import java.io.File;

public class FileMessageToJobRequest {
private Job job;
private String fileParameterName;
public void setFileParameterName(String fileParameterName) {
this.fileParameterName = fileParameterName;
}
public void setJob(Job job) {
this.job = job;
}
@Transformer
Public JobLaunchRequest toRequest(Message<File> message) {
JobParametersBuilder jobParametersBuilder =
new JobParametersBuilder();
jobParametersBuilder.addString(fileParameterName, message.
getPayload().getAbsolutePath());
return new JobLaunchRequest(job, jobParametersBuilder.
toJobParameters());
}
}
```

The job execution status is returned with the instance of `JobExecution`. The `JobExecution` ID helps the user track the status of the job execution through `JobRepository`.

The following configuration is for taking the file input (CSV file) through an adapter, transforming it to `JobRequest` through the transformer `FileMessageToJobRequest`, launching the job through `JobLaunchingGateway`, and logging the output of `JobExecution`.

```
<integrate:channel id="inputFileRepository"/>
<integrate:channel id="jobRequestChannel"/>
<integrate:channel id="jobTriggeringStatusChannel"/>
<integrate-file:inbound-channel-adapter id="inputFile"
channel="inputFileRepository"
directory="file:/tmp/batchfiles/"
filename-pattern="*.csv">
<integrate:poller fixed-rate="1000"/>
</integrate-file:inbound-channel-adapter>
<integrate:transformer input-channel="inputFileRepository"
output-channel="jobRequestChannel">
<bean class="batchJob.FileMessageToJobRequest">
<property name="job" ref="employeeJob"/>
<property name="fileParameterName" value="input.file.name"/>
</bean>
</integrate:transformer>
<batch-integrate:job-launching-gateway request-
channel="jobRequestChannel"
reply-channel="jobTriggeringStatusChannel"/>
<integrate:logging-channel-adapter channel="jobTriggeringStatusChann
el"/>
```

The item reader can be configured to pick the input filename as job parameter from the following configuration:

```
<bean id="itemReader" class="org.springframework.batch.item.file.
FlatFileItemReader"
scope="step">
<property name="resource" value="file://#{jobParameters['input.file.
name']}"/>
...
</bean>
```

Spring Integration has message access from the Spring application context. Hence, the batch job can as well be triggered with the request accessed from the application context, as shown in the following code:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-
integration-job.xml");
EmployeeJobLaunchRequest employeeJobLaunchRequest = new EmployeeJobLau
nchRequest("employeeJob", Collections.singletonMap("key", "value"));
Message<EmployeeJobLaunchRequest> msg = MessageBuilder.withPayload(
employeeJobLaunchRequest).build();
MessageChannel jobRequestsChannel = ctx.getBean("inputFileRepository",
MessageChannel.class);
jobRequestsChannel.send(msg);
```

In the preceding code, `EmployeeJobLaunchRequest` is the user-defined `JobLaunchRequest` that is wrapped with the Spring Integration Message. The Spring Integration class to generate a message is `MessageBuilder`. With this request, we can pass the input request details, such as file repository, and launch the job. Spring Integration can be learned in detail from *Spring Integration Essentials*, *Chandan Pandey*, *Packt Publishing*.

RESTful job processing

A web service is a method of communication between two electronic devices over a network. It is a software function provided at a network address over the Web, with the service always on as in the concept of utility computing.

REST is an architectural style consisting of a coordinated set of architectural constraints, applied to components, connectors, and data elements, within a distributed hypermedia system. The REST architectural style is also applied to the development of web services. With REST-compliant web services, the primary purpose of the service is to manipulate XML representations of web resources using a uniform set of stateless operations.

Spring Batch supports the job launching and processing using REST web services with the methods `Put/Post`. The following is a sample listing with Spring CXF (an open source services framework):

```
@Autowired
private JobLauncher jobLauncher;

@Autowired
private Job job;

public boolean startJob() throws Exception {
try {
final JobParameters jobParameters = new JobParametersBuilder().
addLong("time", System.nanoTime()).toJobParameters();
final JobExecution execution = jobLauncher.run(job, jobParameters);
final ExitStatus status = execution.getExitStatus();
if (ExitStatus.COMPLETED.getExitCode().equals(status.getExitCode()))
  {
    result = true;
  }
}
} catch (JobExecutionAlreadyRunningException ex) {
  System.out.println("Exception" + ex);
} catch (JobRestartException ex) {
  System.out.println("Exception" + ex);
} catch (JobInstanceAlreadyCompleteException ex) {
  System.out.println("Exception" + ex);
} catch (JobParametersInvalidException ex) {
  System.out.println("Exception" + ex);
} catch (IOException ex) {
  System.out.println("Exception" + ex);
}
 return false;
}
```

Autowired `JobLauncher` and `Job` objects get injected into the application. The `startJob()` method creates `JobParameters` using `JobParametersBuilder`, and the job gets triggered by `jobLauncher.run()`. This invocation of batch job from the web service calls `JobLauncher.run()` to trigger a batch job in a synchronous thread. The `ExitStatus` can be accessed from the `JobExecution` object. Any exception during job launch can be caught with proper exception handling, as mentioned in the preceding list.

Summary

Through this chapter, we learned about enterprise integration and different methods available for enterprise application integration. We also learned how the Spring Integration project can integrate Spring Batch applications with its message-driven approach. We also learned about launching the batch jobs by accessing the Spring Integration components from the application context. We finished this chapter with an understanding of the RESTful job processing technique.

In the next chapter, we will learn about inspecting the Spring Batch jobs, including accessing execution data, listeners, and web monitoring.

7
Inspecting Spring Batch Jobs

In the previous chapter, we learned about enterprise integration, varieties of enterprise application integration, and the Spring Integration project to integrate Spring Batch applications with its message-driven approach. We also learned about launching batch jobs with Spring Integration and RESTful job processing techniques. Spring Batch job execution deals with huge data that changes time to time. This changing data might get corrupted at times and lead to failed job executions. It is important to keep a close eye on such failures, and failure reasons should be saved in a constructive manner for future tracking and fixing.

In this chapter, we will cover the following topics:

- Batch job monitoring
- Accessing execution data
- Listeners
- Web monitoring

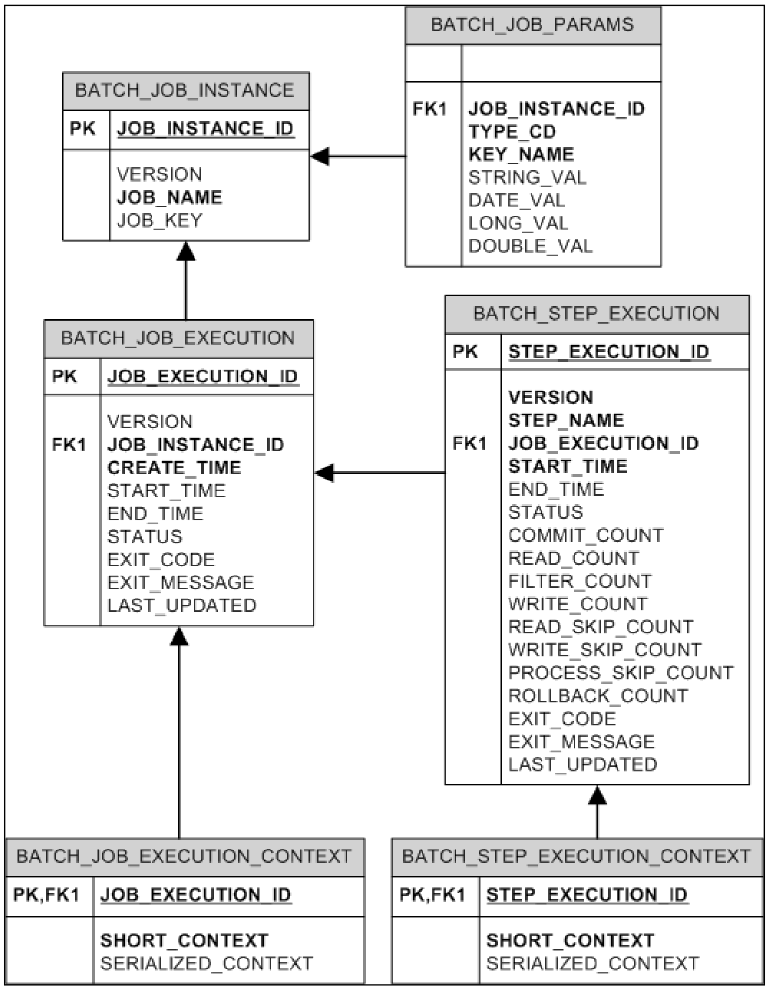
Batch job monitoring

So far we have seen varieties of batch job configurations and executions handling data from diverse sources, processing it, and pushing the outcomes into another data store. Everything looks good as long as the jobs keep executing the way we make the configurations. The stability of an application can be figured by how strong and detailed the response of the application is to any problems with its surroundings, that is, the environment in which the application is running, the availability and accessibility of external systems, and the correctness of the data supplied to the application.

Applications should be able to generate clear tracking information on what is happening in and out of the application in terms of functionality, who is using it, how the performance is, and a detailed stack of issues/errors the application faces. Spring Batch addresses these parameters and generates a greater infrastructure to monitor the batch job processing and store this monitored information.

The application infrastructure should take care of identifying any such problems and also reporting to the respective departments through the preconfigured channels of communication. Spring Batch has a strong infrastructure to maintain the monitored job information in a database. Let us understand the database infrastructure and how each entity is related to each other.

The following is the schema diagram defined by Spring Batch:

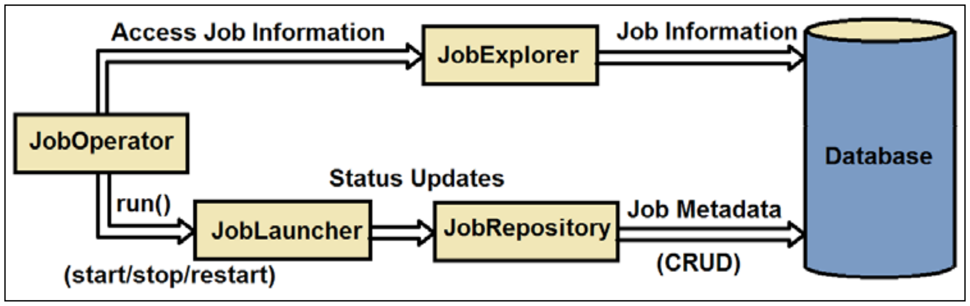


The preceding figure depicts the schema of the batch job that takes care of the job execution information. The significance of each of these entities is as follows:

- `BATCH_JOB_INSTANCE`: This maintains high-level information on the batch jobs, along with the instance of each job. It contains a unique identifier for different job instances created for the same job, with a different set of job parameters (`JOB_KEY`), along with the job name and version of each record.
- `BATCH_JOB_PARAMS`: This maintains information related to each set of job parameters instance. It maintains the key/value pairs of job parameters to be passed to a job.
- `BATCH_JOB_EXECUTION`: This maintains the job execution information for each instance of the job. It maintains individual records for each execution of the batch job by connecting with `BATCH_JOB_INSTANCE`.
- `BATCH_STEP_EXECUTION`: This maintains the step execution information for each step of a job instance. It connects with `BATCH_JOB_EXECUTION` to maintain the step execution information for each job execution instance.
- `BATCH_JOB_EXECUTION_CONTEXT`: This is the information needed for each job execution instance. This is unique for each execution, so the same information as that of the previous run is considered for retry jobs. Hence, it connects with `BATCH_JOB_EXECUTION` to maintain an instance per execution.
- `BATCH_STEP_EXECUTION_CONTEXT`: This is similar to `BATCH_JOB_EXECUTION_CONTEXT`, except that it maintains the context information for each of the step execution. Hence, it connects with `BATCH_STEP_EXECUTION` to maintain the unique instance with each step execution instance.

Accessing execution data

While Spring Batch saves all the monitoring and job information to the database, let's understand each of the administration components of Spring Batch, how they interact with each other, and their configurations.



Database

The database saves job-related information and acts as a source to monitor the job execution information.

A database can be configured using the following syntax:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName" value="${batch.jdbc.driver}"/>
<property name="url" value="${batch.jdbc.url}"/>
<property name="username" value="${batch.jdbc.user}"/>
<property name="password" value="${batch.jdbc.password}"/>
</bean>
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.
DataSourceTransactionManager" lazy-init="true">
<property name="dataSource" ref="dataSource"/>
</bean>
```

The values of `driverClassName`, `url`, `username`, and `password` can be specific to the database and a particular user connection to that database. `DataSourceTransactionManager` is the transaction manager here and it refers to the database with the `datasource` property.

JobRepository

The `org.springframework.batch.core.repository.JobRepository` interface is the central point of access for job-related information. It accesses the state and metadata of batch jobs from the database and supplies it to other resources.

The configuration for the `JobRepository` follows the ensuing syntax:

```
<job-repository id="jobRepository"
data-source="dataSource" transaction-manager="transactionManager" />
```

JobLauncher

The `org.springframework.batch.core.launch.JobLauncher` interface is responsible for job execution and also updates the changing job status in `JobRepository`.

The configuration for `JobLauncher` follows the following syntax:

```
<job-repository id="jobRepository"
data-source="dataSource" transaction-manager="transactionManager" />
<bean id="taskExecutor"
class="org.springframework.core.task.SimpleAsyncTaskExecutor"/>
<bean id="jobLauncher"
class="org.springframework.batch.core.launch.support.
SimpleJobLauncher">
<property name="jobRepository" ref="jobRepository"/>
< property name="taskExecutor" ref="taskExecutor"/>
</bean>
```

JobOperator

The `org.springframework.batch.core.launch.JobOperator` interface acts as a controlling point for batch job processing. It sends start, stop, and restart signals to another administrator, namely, `JobLauncher` by accessing the job information from `JobExplorer`.

The configuration for `JobOperator` follows the following syntax:

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

JobExplorer

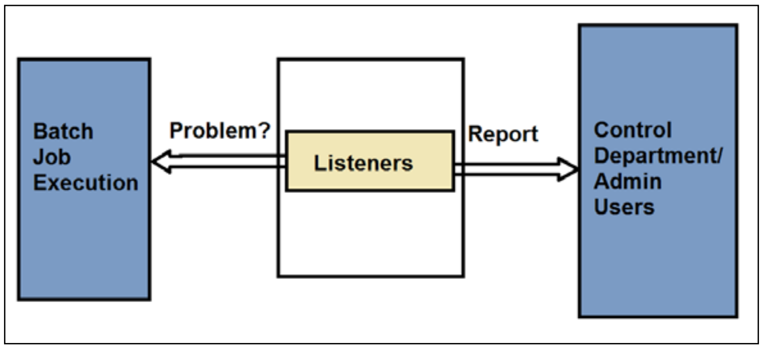
The `org.springframework.batch.core.explore.JobExplorer` interface reads the job-related information from the database and provides the information to other administrators, such as `JobOperator` in job execution, with read-only access.

The configuration for `JobExplorer` follows the ensuing syntax:

```
<bean id="jobExplorer"
class="org.springframework.batch.core.explore.support.
JobExplorerFactoryBean"
prop:dataSource-ref="dataSource" />
```

Listeners

In the previous chapters, we discussed that listeners are the components that get triggered by the preconfigured events in an execution. We can use such listeners to trigger a particular event on the batch job and act as a monitoring tool for the corresponding problems. A listener can also be configured to report the corresponding department for a particular problem in execution.



The following is a sample listener definition and configuration to monitor a batch job problem.

A listener class can be defined to execute before and after the execution of a job, as shown in the following code:

```
public class JobMonitoringListener {
  @BeforeJob
  public void executeBeforeJob(JobExecution jobExecution) {
    //pre-run executions
  }
  @AfterJob
  public void executeAfterJob(JobExecution jobExecution) {
  if(jobExecution.getStatus() == BatchStatus.FAILED) {
  /* Report the departments for a failed job execution status.
     The reporter can be a preconfigured mail-sender or an SMS
     sender or any other channel of communication.*/
      reporter.report(jobExecution);
    }
  }
}
```

The configuration of a batch job with the monitoring listener can be as follows:

```
<batch:job id="firstBatchJob">
  <batch:step id="importEmployees">
  ..
  </batch:step>
  <batch:listeners>
    <batch:listener ref="jobMonitoringListener"/>
  </batch:listeners>
</batch:job>
<bean id=" jobMonitoringListener" class="org.java.
JobMonitoringListener"/>
```

Web monitoring

The Spring Batch job execution can be monitored and examined with the web interface provided by the open-source project of Spring, that is, **Spring Batch Admin**. This is a simple web application built with the Spring MVC user interface to act as an admin console for the Spring Batch applications and systems. The main use cases developed in this project are inspecting jobs, launching jobs, inspecting the executions, and stopping the executions.

Refer to the Spring Batch Admin reference guide at `http://docs.spring.io/spring-batch-admin/reference.html` for detailed information on installation and usage.

Summary

Through this chapter, we learned the importance of job execution monitoring and Spring Batch job monitoring infrastructure. We also learned how to access job execution information with the help of the administrators' configurations. In addition, we learned about monitoring and reporting batch job problems with the help of listeners. We finished this chapter with an understanding of the Spring Batch Administration project features and how it can help with batch job monitoring.

In the next chapter, we will learn in detail about the batch scaling model, parallel processing, and partitioning concepts.

8
Scaling with Spring Batch

In the previous chapter, we learned about monitoring, accessing the execution information and administering the configurations, using listeners, reporting the batch job problems, and understanding the Spring Batch Administration features. The Spring Batch job execution deals with huge data that changes time-to-time. This detailed processing consumes huge infrastructure. It is obvious to expect these jobs to perform efficiently and meet the scaling needs with the growing size of the organization's data.

In this chapter, we will cover the following topics:

- The batch scaling model
- The thread model
- Parallel processing
- Remote chunking
- Partitioning

The batch scaling model

So far we have seen how to handle different types of batch jobs, configurations, and executions. As the organization size is growing day-by-day, the data to be processed per batch job also gets increased accordingly. It is important to design and configure our batch jobs to meet these performance and scaling expectations.

The batch jobs we write with certain business logic, keeping different resources interacting in between, cannot be changed every time we see change in data load or performance issues. Spring Batch offers rich configuration infrastructure to be able to scale jobs without altering them, by just tuning the configuration information.

Scaling the infrastructure can be done in either of the following two ways:

- **By increasing the capacity of the system hardware**: In this way of scaling, we can replace the existing slow infrastructure with more powerful infrastructure.
- **Adding more servers**: In this way of scaling, we can add more processing systems of the same capacity in parallel to the existing infrastructure. These additional nodes share the work and increase the scaling of the total system.

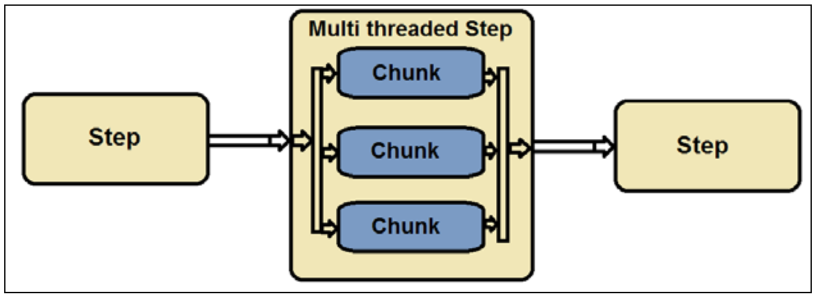
Spring Batch offers the following ways to scale the batch applications:

- **Thread model**: This is a multithreaded step with a single process
- **Parallel processing**: This is a parallel step execution with a single process
- **Remote chunking**: This is the remote chunking of a step with multi process
- **Partitioning**: This is the partitioning of a step; it can be a single or multi process

The thread model

By default, step execution is a single-thread model. Spring Batch lets us configure the step to execute in multiple chunks to let the single step execute in a multithread model with the help of `org.springframework.core.task.TaskExecutor`.

The following diagram depicts the multithread model of a step execution:



The following is the sample configuration for the multithread step with `TaskExecutor`:

```
<step id="employeePayProcessing">
<tasklet task-executor="taskExecutor">
<chunk reader="employeeWorkingDaysReader" processor="employeePayProce
ssor"
writer="employeeSalariesWriter"/>
</tasklet>
</step>
<beans:bean id="taskExecutor"
```

```
class="org.springframework.core.task.SimpleAsyncTaskExecutor">
<beans:property name="concurrencyLimit" value="20"/>
</beans:bean>
```

With the preceding configuration, the `employeePayProcessing` step considers the configured reader, processor, and writers for the tasklets and task execution, with the help of `org.springframework.core.task.SimpleAsyncTaskExecutor` having a thread pool of 20 threads, each executing in parallel with chunks of data being processed in each thread.

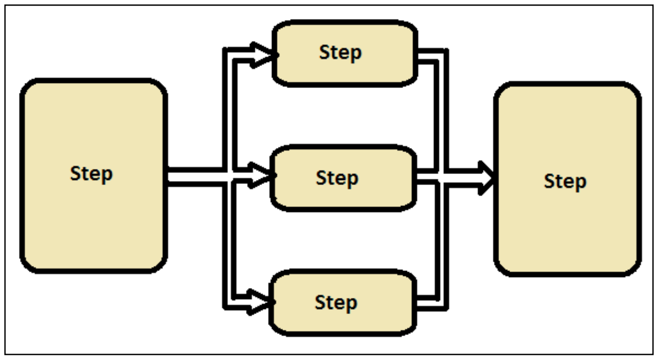
Just like any other multithread model, the Spring Batch multithread models also take into account the resources used by the multiple threads, and whether they are thread safe. `ItemReader` is one such process that is not thread safe.

To configure a thread-safe operation, the recommendation is to synchronize the `ItemReader` process by synchronizing the read method.

Parallel processing

While multithreading allows a single step to be processed in multiple threads of chunks, Spring Batch allows us to process multiple steps and flows simultaneously with the help of parallel processing. This feature enables the independent steps to execute in parallel and ensures a faster processing.

The following figure shows the multiple steps under execution in parallel:



With parallel processing, the independent steps need not wait for the other steps to complete before execution.

The following is the sample configuration for the parallel steps in processing:

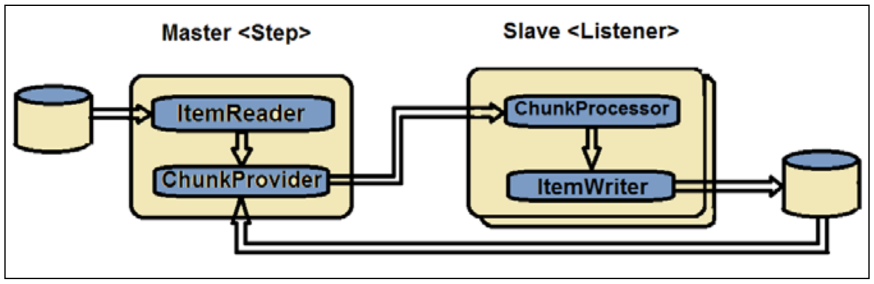
```
<job id="employeePayProcessing">
<split id="splitProcess" task-executor="taskExecutor"
next="payCalculations">
<flow>
<step id="readEmployeeData" parent="stepOne"
next="processEmployeeData"/>
<step id="processEmployeeData" parent="stepTwo"/>
</flow>
<flow>
<step id="organizationDataSetup" parent="stepThree"/>
</flow>
</split>
<step id="payCalculations" parent="stepFour"/>
</job>
<beans:bean id="taskExecutor" class=" org.springframework.core.task.
SimpleAsyncTaskExecutor"/>
```

In the preceding configuration, the `readEmployeeData` and `processEmployeeData` steps get executed in parallel with `organizationDataSetup`. By default, the `taskExecutor` is `SyncTaskExecutor`; with the preceding configuration we changed it to `SimpleAsyncTaskExecutor` to support parallel step processing.

Remote chunking

Remote chunking is the process in which the original step reads the data calls from remote process to process and writes or receives the processed data back to write on to the system. As the remote chunking deals with the data transmission to another system that is remotely located, we should also consider the cost in building this infrastructure versus the advantage we are getting in remote processing. The actual step (master) executes the read process, and the remote slaves (listeners) could be the JMS listeners that execute the process and write steps, or return the processed information to the master.

The following figure depicts the steps in remote chunking:



The `ChunkProvider` interface returns chunks from `ItemReader`:

```
public interface ChunkProvider<T> {
void postProcess(StepContribution contribution, Chunk<T> chunk);
Chunk<T> provide(StepContribution contribution) throws Exception;
}
```

The `ChunkProcessor` interface processes the chunks:

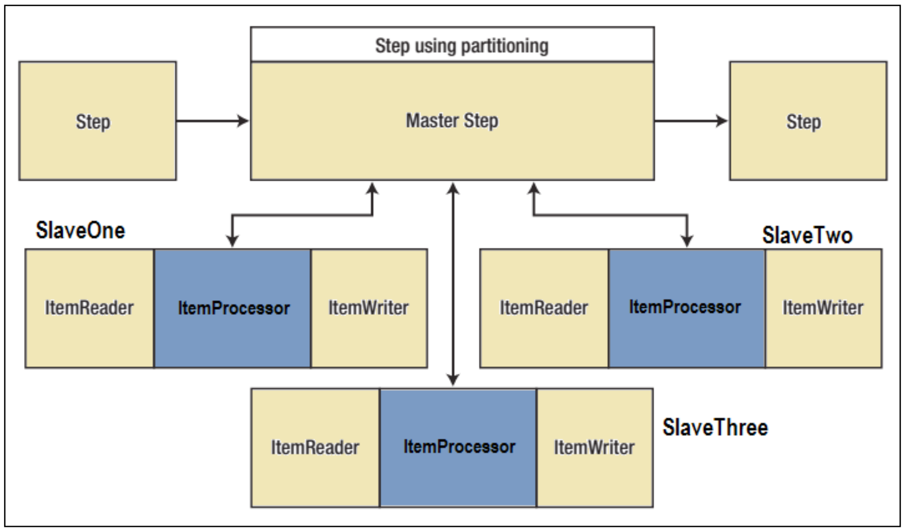
```
public interface ChunkProcessor<I> {
void process(StepContribution contribution, Chunk<I> chunk) throws
Exception;
}
```

To be able to effectively perform the remote interactions, the remote chunking process can have the Spring Integration project included to deal with the integration of resources.

Partitioning

While the remote chunking reads the data at master node and handles the processing to another remote system (slave), partitioning executes the entire process (reading, processing, and writing) in parallel, by having the multiple systems having the entire processing ability. Here, the master step takes care of understanding the job and handing over the task to multiple slaves, and slaves have to take care of the rest of the tasks (reading, processing, and writing). Essentially, the slaves constitute the steps that take care of the read, process, and write in their own world.

The advantages of partitioning over remote chunking include the data transmission not being there, as the slave system takes care of the read step as well.



Even though the communication sent by the master to the slaves in this pattern fails to deliver, batch metadata in the `JobRepository` ensures that each slave gets executed only once per job execution.

The Spring Batch partitioning **Service Provider Interface** (**SPI**) has the following infrastructure for effective partitioning:

- `PartitionHandler`: This sends `StepExecution` a request to the remote steps. It doesn't have to know how to split or integrate the data, and `TaskExecutorPartitionHandler` is the default implementation of `PartitionHandler`.
- `Partitioner`: This generates the step executions for the partitioned steps (only for new step executions). `SimplePartitioner` is the default implementation of `Partitioner`.
- `StepExecutionSplitter`: This generates the input execution contexts for the partitioned step execution, and `SimpleStepExecutionSplitter` is the default implementation.

The following is the sample partitioned step execution configuration:

```
<step id="initialStep">
<partition step="stepPartition" handler="handler"/>
</step>
<beans:bean class="org.springframework.batch.core.partition.support.
TaskExecutorPartitionHandler">
<beans:property name="taskExecutor" ref="taskExecutor"/>
<beans:property name="step" ref="stepPartition"/>
<beans:property name="gridSize" value="10"/>
</beans:bean>
```

The preceding configuration starts its execution with `initialStep` and hands over the execution to the partitioned step. The grid size indicates the number of different steps to be created.

While the multithread model fits for the basic tuning of chunk processing, parallel processing lets us configure independent steps to execute in parallel. Remote chunking needs comparatively larger infrastructure and configuration but fits for distributed nodes processing. Partitioning helps quickly replicate the batch infrastructure and configure the entire process to execute in parallel nodes, with a single point of the repository acting as master.

Based on the system requirement and feasibility of the available infrastructure, one can choose either of the earlier mentioned scaling strategies for batch job execution.

Summary

Through this chapter we learned the importance of performance and scaling of the batch. We also learned Spring Batch offerings to scale the batch applications. In addition, we learned about the details and configurations of a thread model, parallel processing, remote chunking, and partitioning techniques. We finished this chapter with an understanding of choosing the right strategy to scale the batch application with the available infrastructure.

In the next chapter, we will learn in detail about performing different types of testing on Spring Batch applications.

9
Testing the Spring Batch

In the previous chapter, we learned the importance of performance and scaling batch applications through different configurations (namely, thread model, parallel processing, remote chunking, and partitioning techniques) and how to choose the right strategy to scale the batch application with the available infrastructure. Spring Batch applications are developed and configured with individual components and different integrations, and hence it is important to test the individual features as well as the integrated project for its expected behavior.

In this chapter, we will cover the following topics:

- Types of testing for Spring Batch
- Unit testing
- Integration testing
- Functional testing

Types of testing for Spring Batch

The primary purpose of any software testing is to detect software failures and correct them. The scope of software testing can be established from validating the software components to verifying the software functionality and the software functioning in various environments and conditions.

The following are the types of software testing that we might want to perform on Spring Batch applications:

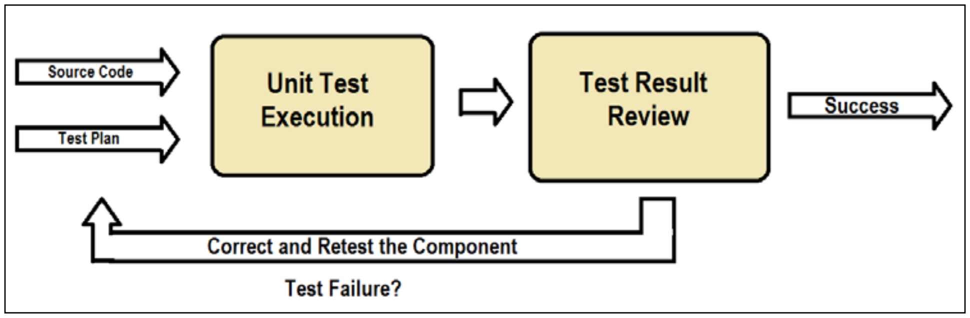
- **Unit testing**: Also known as component testing, this refers to verifying the functionality of a specific piece of code. Unit testing is generally written by the developers.

- **Integration testing**: This identifies the defects in the interfaces and the interaction between integrated components. As the software components are integrated in an iteratively incremental fashion, integration testing is an important testing aspect in larger projects.
- **Functional testing**: This verifies the functionality of a specific code component or group of code components, as defined in the functional specification of a particular application.
- **Performance testing**: This verifies if the entire system is meeting the performance standards expected from the specified environment or run conditions.

Functional testing and performance testing are usually covered together in system testing.

Unit testing

Unit testing is the component-level testing performed by developers, with the source code and test plan prepared by developers. If the unit test fails, developers can fix the issues of the component and perform the unit test again. The following figure depicts the unit test scenario.



JUnit

JUnit is the standard Java framework to perform unit testing. Most IDEs have in-built support for JUnit. TestNG can also be used as a JUnit analog.

JUnit test cases can be written as simple Java classes to be executed with the `@Test` annotation on a method that performs the test operation.

The following is an example of JUnit on a Java String concatenation operation:

```
public class MyClass {
    public String concatenate(String former, String later){
        return former + latter;
    }
}
```

The `Junit` class to test this Java class can be as follows:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MyClassTest {

    @Test
    public void testConcatenate() {
        MyClass myclass = new MyClass();

        String output = myClass.concatenate("Spring", "Batch");

        assertEquals("SpringBatch", output);

    }
}
```

In the preceding class, the `testConcatenate` method with the `@Test` annotation verified the `MyClass java component`. The `assertEquals()` method does the actual testing by comparing the `MyClass.concatenate()` method output with the expected output. If the comparison fails, the `assertEquals()` method throws an exception. We can also write methods to set up operations before the unit test method execution with `@Before` annotation and clean up tasks after the unit test operation with the `@After` annotation. A detailed list of JUnit classes and their usage can be referred to from JUnit API (`http://junit.sourceforge.net/javadoc/`).

Mockito

As we need to perform batch application testing, each component can access the other with some dependency. Replicating all those classes needs to create the instances of such objects and provide to the component under test. Mockito is an open source Java framework that lets us create test double objects (mock objects) easily for the purpose of testing.

Mockito can be added to the application with an easy Maven dependency, such as the following:

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-all</artifactId>
<version>1.10.8</version>
<scope>test</scope>
</dependency>
```

In a batch application, we have the `EmployeeReader` class that we need to create an object during the test execution. This can be performed with the help of `Mockito` within the `JUnit` test class, as follows:

```
import static org.mockito.Matchers.any;
import static org.mockito.Matchers.eq;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

import org.junit.Before;
import org.junit.Test;

public class EmployeeReaderUnitTest {
  EmployeeReader empReaderMock;

  @Before
  public void setUp() {
  empReaderMock = mock(EmployeeReader.class);
  }

  @Test
  public void testEmpReader()throws Exception {
  ...
  }
}
```

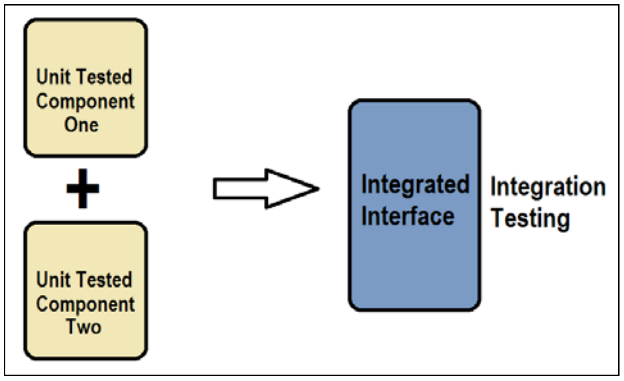
In the preceding code snippet, we created the `EmployeeReader` object with the help of the `mock()` method from `Mockito` within the JUnit `@Before` (setup) method. We shall utilize this object in the `@Test` method to validate the functionality of the component.

Likewise, any Spring Batch components, such as listeners and validator must be unit tested to verify functionality.

Integration testing

Integration testing identifies the defects in the software components as they are integrated in an iteratively incremental fashion. Integration testing is an important testing aspect in larger projects.

The modules that are unit tested are grouped to the larger aggregation, tested according to the integration test plan, and then the tested application is ready for its next level of testing as a system for functional testing.



The following are the two ways to perform component testing with Spring Batch.

Listener-based approach

The following class-level annotations help in testing the Spring Batch components:

- `@RunWith(SpringJUnit4ClassRunner.class)`: This annotation indicates that the class should use the Spring support for JUnit facilities.
- `@ContextConfiguration`: This lets the program know about the configuration properties that contain the application context.
- `@TestExecutionListeners`: This helps in configuring listeners to help the test execution to set up the abilities such as dependency injection and step scope test execution.

The following is a sample configuration using these annotations:

```
@ContextConfiguration(locations = { "/app-context.xml" })
@TestExecutionListeners({ DependencyInjectionTestExecutionListener.
class,
StepScopeTestExecutionListener.class })
```

```
@RunWith(SpringJUnit4ClassRunner.class)
public class CarFileReaderIntegrationTest {
...
}
```

The StepScopeTestUtils approach

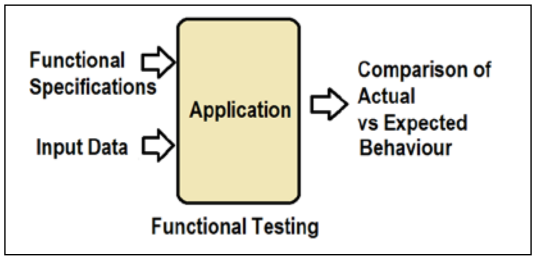
The `StepScopeTestUtils` utility class is used to create and manipulate `StepScope` in unit tests. This helps in using the Spring test support and injecting dependencies into the test case being the step scoped in the application context.

```
@Test
public void testEmployeeReader() throws Exception {
  StepExecution execution = getStepExecution();
  int empReads =
  StepScopeTestUtils.doInStepScope(stepExecution, new
  Callable<Integer>() {
  @Override
  public int call() throws Exception {
  ((ItemStream) employeeReader).open(new ExecutionContext());
  int count = 0;
  while(employeeReader.read() != null) {
    count++;
    }
  return count;
  }
  });
  assertEquals(empReads, 10);
}
```

The `doInStepScope()` method of `StepScopeTestUtils` takes in `stepExecution` and the callable implementation; it automatically takes the runtime dependency injection and returns the result. The rest of the test is about validating the number of `empReads` with the expected number, with the `assertEquals()` method of `JUnit`.

Functional testing

Functional testing verifies the functionality of a specific code component or a group of code components, as defined in the functional specification of the particular application with input data for the components and the output behavior is compared with the expected behaviour. Functional testing is the "black box" testing, as we deal with only the external system behavior for the specific input with the expected output.


Functional Specifications
Input Data
Application
Comparison of Actual vs Expected Behaviour
Functional Testing

In a Spring Batch application, the entire job is considered as a unit of execution and it can be tested for its functionality with the help of `JobLauncherTestUtils`, a utility class to test Spring Batch jobs. `JobLauncherTestUtils` provides methods to launch an entire `AbstractJob`, allowing for end-to-end testing of individual steps without having to run every step in the job. `JobLauncherTestUtils` also provides the ability to run steps individually from `FlowJob` or `SimpleJob`. By launching steps within a job on their own, end-to-end testing of individual steps can be performed without having to run every step in the job.

The following code snippet is an example of using `JobLauncherTestUtils` to perform job and step launching:

```
@Before
public void setup() {
jobLaunchParameters = new JobParametersBuilder().
addString("employeeData", EMPFILE_LOCATION)
.addString("resultsData", "file:/" + RESULTFILE_LOCATION)
.toJobParameters();
}

@Test
public void testEmployeeJob() throws Exception {
JobExecution execution = jobLauncherTestUtils.
launchJob(jobLaunchParameters);
assertEquals(ExitStatus.COMPLETED, execution.getExitStatus());
StepExecution stepExecution =
execution.getStepExecutions().iterator().next();
assertEquals(ExitStatus.COMPLETED, stepExecution.getExitStatus());
}
```

In the preceding code, with the help of `JobLauncherTestUtils`, we are able to launch the batch job, a particular step, with the help of a simple API as part of the JUnit `@Test` method. The `@Before` (setup) method prepares the `JobLaunchParameters` with the details of the input `employeeData` file to be processed and output result file location to be stored.

Summary

Through this chapter, we learned the importance of software testing and the types of software testing we might want to perform on a Spring Batch application. We also learned about different open source frameworks, such as JUnit and Mockito, to perform unit testing on Spring Batch components. We finished this chapter with an understanding of Spring support, APIs to perform unit testing, integration testing, and functional testing on Spring Batch applications.

In the *Appendix* section, we discuss in detail about setting up the development environment, project configurations, and Spring Batch administration.

Appendix

In the previous chapter, we learned the importance of software testing, types of software testing to perform on a Spring Batch application with the help of frameworks such as JUnit, Mockito on Spring Batch components and Spring support, APIs to perform unit testing, integration testing, and functional testing on Spring Batch applications. The Spring Batch project development setup needs a system to contain Java and IDE (Eclipse), and a project needs to be set up with the dependencies configuration. Also, Spring Batch administration is another important aspect to understand.

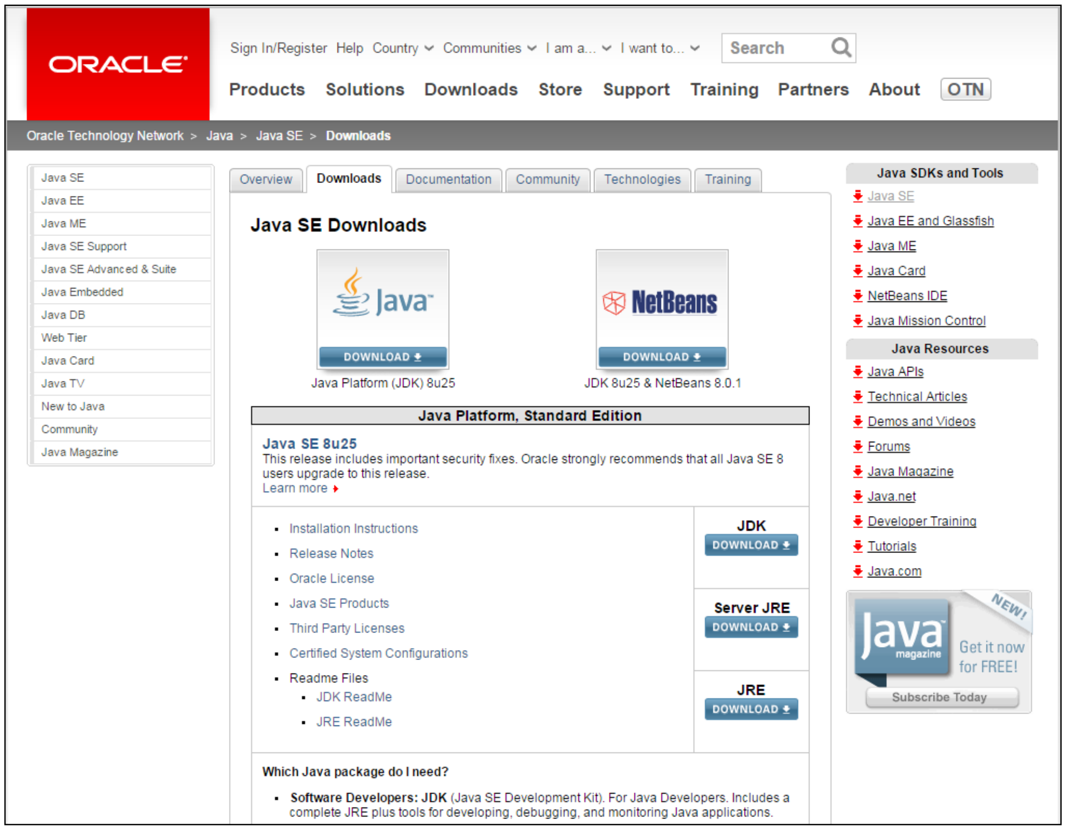
In this section, we will cover the following topics:

- Setting up Java
- Setting up Eclipse IDE
- Setting up the project and its dependencies
- Spring Batch Administration

Setting up Java

The **Java Software Development Kit** (**JDK**) is an application platform released by Oracle and aimed at Java developers using Solaris, Linux, Mac OS X, or Windows. The JDK can be downloaded from the Oracle web downloads (`http://www.oracle.com/technetwork/java/javase/downloads/index.html`).

It can be installed with installation instructions provided on the same page.



Java SE Downloads

Java
DOWNLOAD
Java Platform (JDK) 8u25

NetBeans
DOWNLOAD
JDK 8u25 & NetBeans 8.0.1

Java Platform, Standard Edition

Java SE 8u25
This release includes important security fixes. Oracle strongly recommends that all Java SE 8 users upgrade to this release.
Learn more

- Installation Instructions
- Release Notes
- Oracle License
- Java SE Products
- Third Party Licenses
- Certified System Configurations
- Readme Files
  - JDK ReadMe
  - JRE ReadMe

JDK
DOWNLOAD

Server JRE
DOWNLOAD

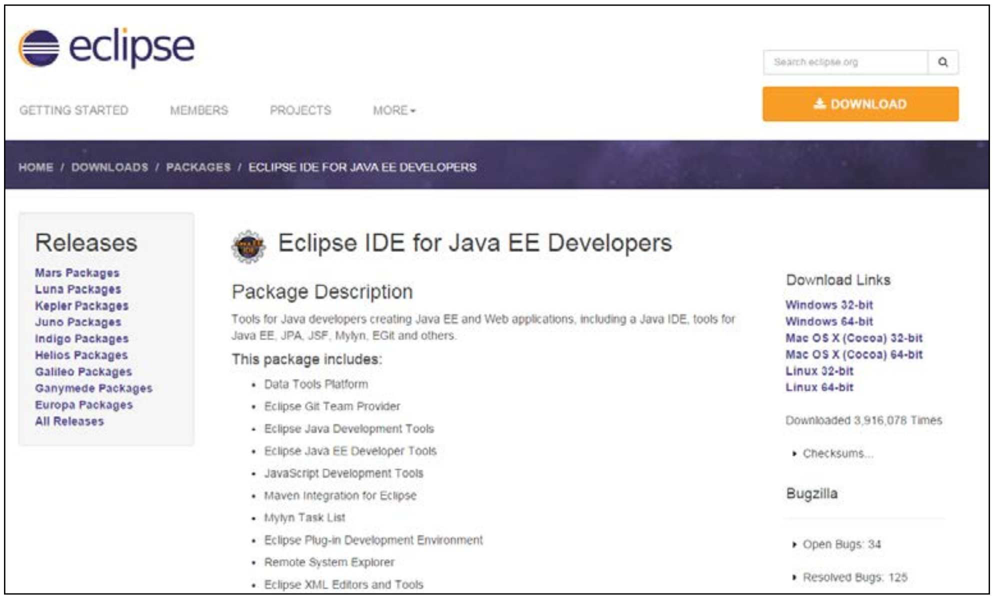
JRE
DOWNLOAD

Which Java package do I need?

- **Software Developers: JDK** (Java SE Development Kit). For Java Developers. Includes a complete JRE plus tools for developing, debugging, and monitoring Java applications.

Setting up Eclipse IDE

Eclipse is one of the most prominent **Integrated Development Environment** (**IDEs**) with base workspace to work on projects, along with extensible plugins for customizing it. Intelij IDEA and NetBeans are some of the other prominent IDEs. We pick the Eclipse IDE for Java EE developers through `https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplersr2`.



eclipse

GETTING STARTED MEMBERS PROJECTS MORE

DOWNLOAD

HOME / DOWNLOADS / PACKAGES / ECLIPSE IDE FOR JAVA EE DEVELOPERS

Releases

Mars Packages
Luna Packages
Kepler Packages
Juno Packages
Indigo Packages
Helios Packages
Galileo Packages
Ganymede Packages
Europa Packages
All Releases

Eclipse IDE for Java EE Developers

Package Description

Tools for Java developers creating Java EE and Web applications, including a Java IDE, tools for Java EE, JPA, JSF, Mylyn, EGit and others.

This package includes:

- Data Tools Platform
- Eclipse Git Team Provider
- Eclipse Java Development Tools
- Eclipse Java EE Developer Tools
- JavaScript Development Tools
- Maven Integration for Eclipse
- Mylyn Task List
- Eclipse Plug-in Development Environment
- Remote System Explorer
- Eclipse XML Editors and Tools

Download Links

Windows 32-bit
Windows 64-bit
Mac OS X (Cocoa) 32-bit
Mac OS X (Cocoa) 64-bit
Linux 32-bit
Linux 64-bit

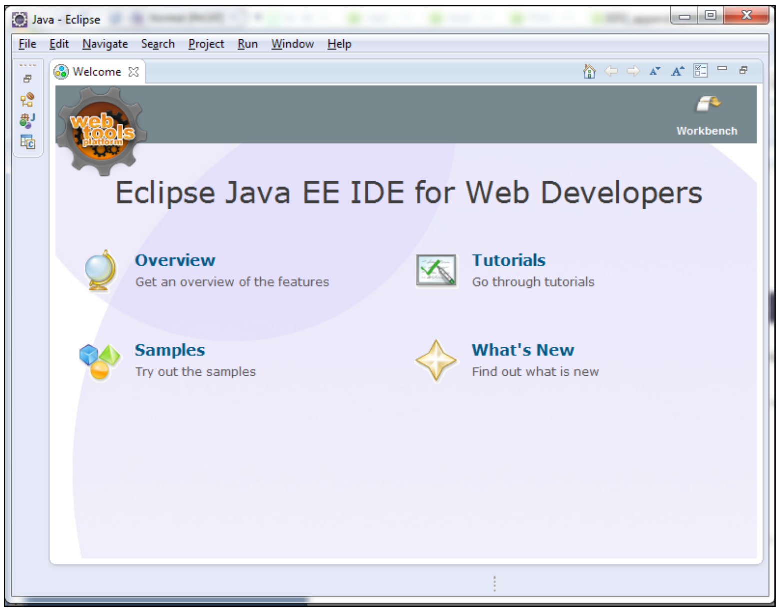
Downloaded 3,916,078 Times

- Checksums...

Bugzilla

- Open Bugs: 34
- Resolved Bugs: 125

Usually, the Eclipse IDE downloads the compressed version and can be unzipped to get the `eclipse` folder. Under the `eclipse` directory, we can observe the executable eclipse program (`eclipse.exe`), which opens the Eclipse IDE as shown in the following screenshot, making sure the Java Development Kit is installed on the machine and is available on system path.

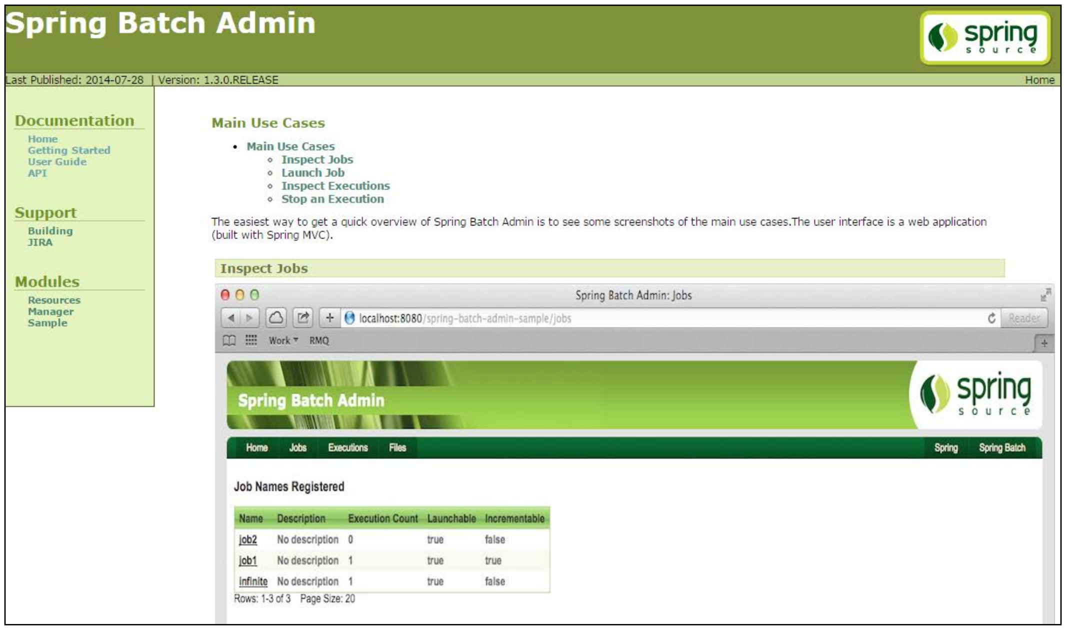


Setting up the project and its dependencies

The Maven software needs to be integrated into the Eclipse IDE, as mentioned in the instructions at `https://www.eclipse.org/m2e/`, and a Java project with Maven dependencies can be managed with the required dependencies for the Spring Batch job, as mentioned in the *Job design and executions* section of the *Chapter 1*, *Spring Batch Fundamentals*.

Spring Batch Administration

The administration tasks include starting and stopping the process jobs, and monitoring the job performance statistics of job executions. Spring Batch is available with an administration project from Spring to have a web-based control tool, namely, Spring Batch Admin project. Spring Batch Admin is an easily deployable project with very good documentation. The documentation of the Spring Batch Admin project is available at `http://docs.spring.io/spring-batch-admin/getting-started.html`.



Main Use Cases

- Main Use Cases
  - Inspect Jobs
  - Launch Job
  - Inspect Executions
  - Stop an Execution

The easiest way to get a quick overview of Spring Batch Admin is to see some screenshots of the main use cases.The user interface is a web application (built with Spring MVC).

Inspect Jobs

Job Names Registered

Name	Description	Execution Count	Launchable	Incrementable
job2	No description	0	true	false
job1	No description	1	true	true
infinite	No description	1	true	false

Rows: 1-3 of 3 Page Size: 20

Through this appendix, we covered how to set up Java, how to set up an Eclipse IDE, how to set up a project with dependencies, and how to administer the batch application with the help of Spring Batch Administration.

This concludes the essential concepts to be learned for Spring Batch job application development.

Index