Batch Application Style

Spring Batch Developer Guide

Table of Contents

1 Introduction 2
1.1 Overview 2
1.2 Audience 2
2 The Domain of Batch 2
2.1 Job 2
2.2 Step 3
2.3 Execution Context 3
2.4 Job Repository 4
3 Batch Namespace 4
4 Configuring a Job 5
4.1 Controlling Step Flow 5
4.1.1 Sequential Flow 5
4.1.2 Complex Flow 5
5 Configuring a Step 6
5.1 Chunk-Oriented Processing 6
5.1.1 Configuring Skip Logic 7
5.2 Tasklet Step 8
6 Reading and Writing 9
6.1 Working with the Database 9
6.1.1 Reading from the database 9
6.1.2 Writing to the database 10
6.2 Working with Flat Files 11
6.2.1 Reading from a Flat File 11
6.2.2 Writing to a Flat File 12
7 Listeners 13
7.1 Job Listeners 14
7.2 Step Listeners 14
8 Running a Job 14
9 Testing 15
9.1 Creating a Test Class 15
9.2 Testing a Job 15
9.3 Testing a Step 16
10 References 16

INTRODUCTION

Overview

Spring Batch is an open source framework created by Accenture and Spring Source that provides the platform for batch processing. Spring Batch provides most of the functionality required to process large

volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management. It is written in Java and is dependent on the Spring Framework.

This guide is meant to introduce the core concepts of Spring Batch. It is not meant to be a comprehensive guide. For a comprehensive look at Spring Batch, please refer to the official Spring Batch Reference Documentation.

Audience

Application Developer

THE DOMAIN OF BATCH

Job

A **Batch Job** is an encapsulation of an entire batch process. It is the entity with which the user (or scheduler) interacts; when processing needs to be done, a batch job is launched. A batch job is defined in a "job configuration" XML document and each job has a unique name. For example, assume that there is a batch job called "EndOfDay" which runs at the end of each day to clean out old records from the database. The "EndOfDay" job is defined in an job configuration.

The "EndOfDay" job must know the current date when it runs because it uses this date as part of its logic. The current date would be given to the job as a **Job Parameter**, that is, a value given to the job when it is launched. For example, on May 1st, 2009, the "EndOfDay" job would be launched with the job parameter "schedule.date=2009/05/01". When the job is launched the next day, the parameter "schedule.date=2009/05/02" would be given. Together, the job and a set of job parameters make a Job Instance.

A Job Instance is a batch job with a particular set of job parameters. So, when "EndOfDay" is launched on May 1st with parameter "schedule.date=2009/05/01" it is a considered to be a separate Job Instance from when "EndOfDay" is launched on May 2nd with parameter "schedule.date=2009/05/02", even though it is the same Job both times. Therefore, there may be many Job Instances for a single Job.

A **Job Execution** is a particular execution (or run) of a Job Instance. Because a batch job may fail during execution, it is possible to restart the job. However, when a job is restarted it would be given the same Job Parameters. For example, if the "EndOfDay" job is launched on May 1st with job parameter "schedule.date=2009/05/01" and fails, then it will have to be restarted, again with job parameter "schedule.date=2009/05/01" to make sure that all of the May 1st processing is completed. Each of these launches will be considered a separate Job Execution even though they are both part of the same Job Instance (and Job) since the Job Parameters are the same. Consequently, there can be multiple Job Executions for a single Job Instance.

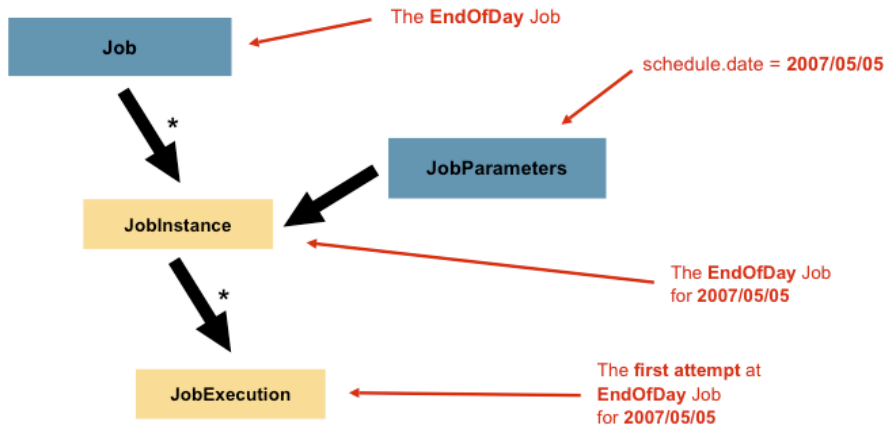

Figure 1. Job Stereotypes

Step

A **Step** is an encapsulation of a single phase of processing in a batch job. A batch job contains one or more steps which it will execute when it is launched.

A **Step Execution**, similar to a Job Execution, is a single execution (or run) of a particular step within a particular Job Execution. For example, if the "EndOfDay" job contains step "StepA" and "StepB", then when the "EndOfDay" job is launched, then "StepA" will be executed followed by "StepB". Each of these steps will produce a Step Execution when it is run. If the job fails and is restarted, then any steps that are executed this time will produce new Step Executions.

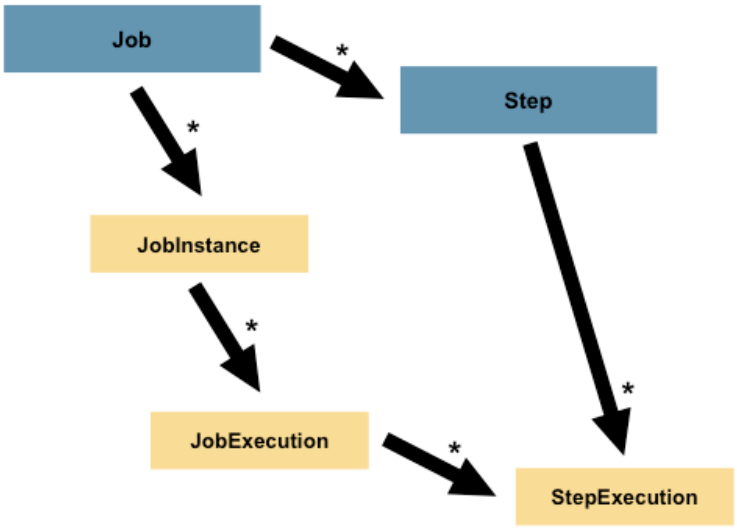

Figure 2. Job and Step Stereotypes

Execution Context

The **Execution Context** is a Map containing data that is used to store data partitioning to the job or the step. It can be used to store data used in processing, such as a "total" amount that must be updated until the end of the step before it can be written. It can also be used to store "restart data" such as the current line number from an input flat file so that if the job fails, it can be restarted where it left off.

During execution, there will always be two Execution Contexts, one for the Job and one for the Step. They are scoped to the JobExecution and StepExecution, respectively. In other words, the job's

Execution Context exists as long as the job is running but the step's Execution Context exists only as long the particular step is running.

In order to have the Execution Context data available on restart, the map is persisted to the database at regular intervals. The job's Execution Context is written to the database at the end of each step. The step's Execution Context is written to the database at the end of each commit interval, after the ItemWriter writes its items.

Job Repository

The **Job Repository** is the persistence mechanism used by Spring Batch to store the meta-data stereotypes mentioned above. It is configured in the job configuration XML:

Example 1. Job Repository Configuration

```
<job-repository id="jobRepository"
                dataSource="dataSource"
                transactionManager="transactionManager"
                isolation-level-for-create="serializable"
                table-prefix="BATCH_"/>
```

The only required attribute is "id". All of the other attributes, if not specified, will default to the values shown above.

BATCH NAMESPACE

Spring Batch provides its own XML namespace to simplify the writing of Spring ApplicationContext files. The namespace is declared with the following header:

Example 2. Batch Namespace Configuration Header

```
<?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-2.0.xsd
      http://www.springframework.org/schema/batch
      http://www.springframework.org/schema/batch/spring-batch-2.0.xsd">
```

CONFIGURING A JOB

A batch job should be defined using the namespace. All that is required is a name, a JobRepository, and one or more steps:

Example 3. Job Configuration

```
<job id="job1" job-repository="jobRepository">
    <step ... />
</job>
```

The job-repository attribute defaults to "jobRepository", so it is only required if the JobRepository defined has a different id.

Controlling Step Flow

The flow from one step to the next is controlled through various XML elements and attributes. A step can be configured to go to any other step in the job based on its status; or, it can be instructed to simply end the job.

Sequential Flow

The simplest case of a batch job is one where steps are executed sequentially until either a step fails or the last step has completed. This behavior can be achieved using the "next" attribute of the <step/> element:

Example 4. Sequential Flow

```
<job ... >
    <step id="step1" ... next="step2"/>
    <step id="step2" ... next="step3"/>
    <step id="step3" ... />
</job>
```

According to this configuration, once "step1" completes, "step2" will run, followed by "step3". However, if any step fails, then the job will end with status FAILED and subsequent steps will not execute.

Complex Flow

For more complex flow scenarios, Spring Batch provides transition elements that can be used to dictate flow behavior. These transitions, along with their required attributes, are:

Table 1. Flow Transitions

Transition Element	Description
<next on="..." to="..."/>	Transition to the specified step.
<end on="..."/>	End the job with status COMPLETED. Restart is not allowed.
<fail on="..."/>	End the job with status FAILED. Restarting would begin on this step.
<stop on="..." restart="..."/>	End the job with status STOPPED. Restarting would begin on the specified step.

All of these transitions contain an "on" attribute. This specifies the exit code to which the transition applies. For example <end on="FAILED"/> would mean that the job should end if the step results in an exit code of FAILED. The "on" attribute also allows wildcards, so <next on="*" to="step2"/> would mean that the step should transition to "step2" no matter what the exit code. It should be noted that the framework will also use the most specific transition available, so if the exit code is COMPLETED and there are two transitions, one "COMPLETED" and the other "*", the transition with "COMPLETED" will be used. For example:

Example 5. Complex Flow

```
<job ... >
    <step id="step1" ... >
        <next on="*" to="step2"/>
        <end on="FAILED"/>
    </step>
    <step id="step2" ... />
</job>
```

CONFIGURING A STEP

A step represents a single stage of processing within a batch job. There is no limit to the number of steps a job may have, but it must have at least one step.

Chunk-Oriented Processing

The most common type of step is one that processes a collection of records according to a uniform set of rules. For instance, a step may read a list of transaction from a file and insert them into the database or select a list of accounts from the database and update the status of each one. To read from an input source (usually a database or file) an ItemReader class is used. To write to an output source (again, usually a database or file) an ItemWriter is used. A third class, the ItemProcessor, may optionally be called for each item read before the item is given to the ItemWriter.

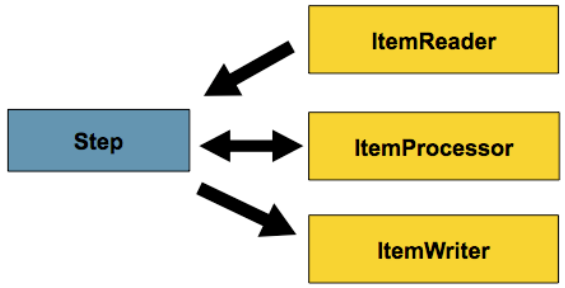

Figure 3. Chunk-Oriented Processing Stereotypes

This type of processing is called "chunk-oriented" because the framework doesn't read or write all of the records at once; it reads only a small portion of them (a chunk) and then writes that portion before reading the next chunk. The size of the chunk is referred to as the "commit interval".

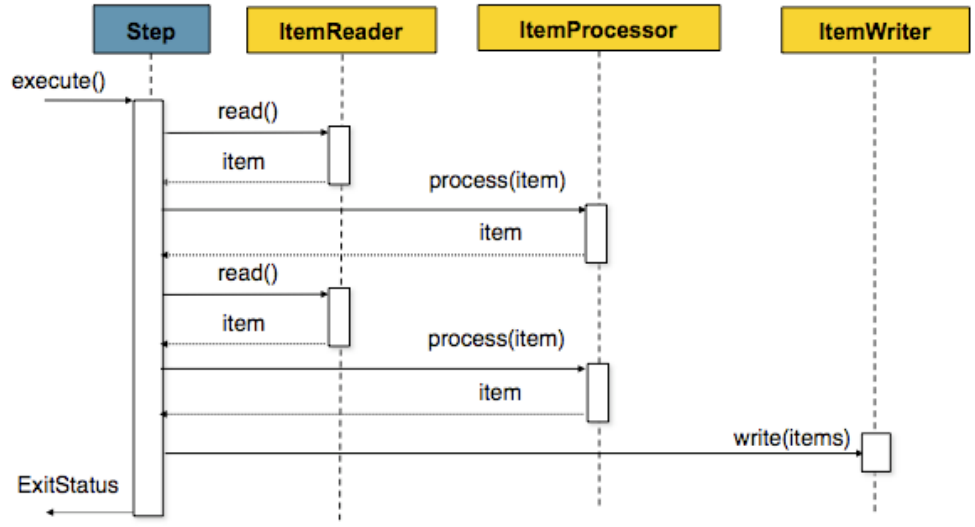

Figure 4. Chunk-Oriented Processing

A chunk-oriented step is configured in Spring Batch by using the <chunk/> element. A chunk-oriented step requires a reader, a writer, and a commit-interval. The transaction-manager attribute will default to "transactionManager" if it is unspecified.

Example 6. Chunk-Oriented Processing Configuration

```
<job id="sampleJob" job-repository="jobRepository">
    <step id="step1">
```

```
        <tasklet transaction-manager="transactionManager">
            <chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>
        </tasklet>
    </step>
</job>
```

Configuring Skip Logic

In the language of chunk-oriented processing, a skip occurs when a record is found to contain an error and thus is should not be processed. The term comes from the fact that bad records are "skipped" over and processing is allowed to continue.

For example, assume a step has five input records, and the fourth causes an exception when it was read, processed, or written. When the step executes, if skips are allowed, then records 1, 2, 3, and 5 will be processed normally while record 4 will be skipped and ignored.

To allow for skips, the skip-limit of the chunk-oriented step must be greater than zero. The skip-limit indicates the number of items that are allowed to be skipped before the job fails.

Example 7. Configuring For Skip

```
<step id="step1">
    <tasklet>
        <chunk reader="itemReader" writer="itemWriter"
               commit-interval="10" skip-limit="100"/>
    </tasklet>
</step>
```

In order to tell the framework which Exceptions should cause skips, the <chunk/> element allows for two configurable lists of exceptions. The skippable-exception-classes list configures which exceptions should cause the item to skip. The fatal-exception-classes list configures which exceptions should cause the item to fail the step. Both lists are comma- or newline-delimited.

Example 8. Exception Class Configuration

```
<step id="step1">
    <tasklet>
        <chunk reader="itemReader" writer="itemWriter"
               commit-interval="10" skip-limit="100">
            <skippable-exception-classes>
                com...MySkippableException
            </skippable-exception-classes>
            <fatal-exception-classes>
                com...MyFatalException
            </fatal-exception-classes>
        </chunk>
    </tasklet>
</step>
```

The decision to skip or fail on a particular exception is made as follows:

- If the exception is listed as "fatal", then fail
- Else, if the exception is listed as "skippable", then skip
- Else, fail

Tasklet Step

While the chunk-oriented paradigm is the most common type for a step, not all steps require this type of processing. Some steps simply need to execute a single method. The **tasklet step** is used for this purpose. For example, if a batch job needs a step that runs a stored procedure in the database or executes a method to delete a file from the file system, these would be best handled with a tasklet step.

A tasklet step is defined with the batch namespace by referencing an implement ion of the Tasklet interface:

Example 9. Tasklet Step Configuration

```
<job ... >
    <step id="step1" ... >
        <tasklet ref="myTasklet"/>
    </step>
</job>

<beans:bean id="myTasklet" class="com...MyTasklet />
```

A tasklet implementation is very simple: it only requires the execute method to be written. All of the processing will be contained in this method since it will be called exactly once by the framework.

Example 10. Tasklet Implementation

```
public class MyTasklet implements Tasklet {
    public RepeatStatus execute(StepContribution contribution,
                                ChunkContext chunkContext) throws Exception {
        // All processing done here
        return RepeatStatus.FINISHED;
    }
}
```

READING AND WRITING

Working with the Database

Reading from the database

If the input items to a chunk-oriented step come from the database, then an ItemReader that reads from the database must be used. For this purpose, the JdbcCursorItemReader is provided. This class can be configured with a DataSource that references the database and an SQL select-statement that will return the records through which the step will loop.

Example 11. Database Item Reader Configuration

```
<bean id="itemReader" class="org.spr...JdbcCursorItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="sql" value="select ID, NAME, CREDIT from CUSTOMER"/>
    <property name="rowMapper">
        <bean class="com.mycompany...CustomerCreditRowMapper"/>
    </property>
</bean>
```

In addition to the "dataSource" and "sql" properties, the class requires that a RowMapper be set. The RowMapper is used to convert the ResultSet returned from the query into the item that will passed to

the processor (optionally) and then to the writer. A simple RowMapper will simply create the item and copy values from the ResultSet to it.

Example 12. Row Mapper Implementation

```
public class CustomerCreditRowMapper implements RowMapper {
    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        CustomerCredit customerCredit = new CustomerCredit();
        customerCredit.setId(rs.getInt("ID"));
        customerCredit.setName(rs.getString("NAME"));
        customerCredit.setCredit(rs.getBigDecimal("CREDIT"));
        return customerCredit;
    }
}
```

Writing to the database

Writing to the database can be done in a custom-written ItemWriter. A DAO can be wired into the writer and called to write an item to the database. In the following example, the writer expects items of the type CustomerCredit. Its write method simply loops through the list of items it receives calls the DAO to create each one in the database.

Example 13. Database Item Writer Configuration

```
public class CustomerCreditItemWriter implements ItemWriter<CustomerCredit> {
    private CustomerCreditDao customerCreditDao = null; //setter omitted...

    public void write(List<? extends CustomerCredit> items) throws Exception {
        for (CustomerCredit customerCredit : items) {
            this.customerCreditDao.create(customerCredit);
        }
    }
}
```

Note that ItemReader, ItemProcessor, and ItemWriter all make use of Java 5's "generic type" features. The generic type corresponds to the type of the item that is read, processed, or written.

Working with Flat Files

The term "flat file" is used to describe a text file that contains records as strings. In general, each line will represent one record. There are two main types of flat files: delimited and fixed-width.

Delimited flat files are those whose records have fields that are separated by some "delimiter". The most common delimiter is the comma (","), though any character can be used.

Example 14. Delimited File

```
372,Alan Turing,572.12
1242,Alonzo Church,532.22
738,Edsger Dijkstra,4131.63
```

Fixed-width files are those whose records are made up of fields with specific, pre-determined lengths.

Example 15. Fixed-Width File

```
372 Alan Turing      572.12
1242Alonzo Church    532.22
738 Edsger Dijkstra  4131.63
```

Reading from a Flat File

To read from a flat file, Spring Batch includes the FlatFileItemReader. This class requires two properties. The first is a Resource which indicates the file from which records will be read. The second is a LineMapper, whose job it is to convert a line from the file (a String) into an object representing the item. The most common LineMapper implementation is DefaultLineMapper. The DefaultLineMapper is made up of two parts: the LineTokenizer and the FieldSetMapper.

Example 16. Flat File Item Reader Configuration

```
<bean class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="resource" value="classpath:/data/input/myFile.txt"
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer" ref="myLineTokenizer"/>
            <property name="fieldSetMapper" ref="myFieldSetMapper"/>
        </bean>
    </property>
</bean>
```

The job of the LineTokenizer is to split the line into a FieldSet, which is an ordered collection of fields that make up the record. The two main LineTokenizer implementations provided for the framework are DelimitedLineTokenizer and FixedLengthTokenizer. The DelimitedLineTokenizer has a delimiter character as a property while the FixedLengthTokenizer requires a list of Ranges corresponding to the fields in the file.

Example 17. Delimited Line Tokenizer Configuration

```
<bean class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
    <property name="delimiter" value=","/>
</bean>
```

Example 18. Fixed Length Line Tokenizer Configuration

```
<bean class="org.springframework.batch.item.file.transform.FixedLengthTokenizer">
    <property name="columns" value="1-4,5-21,22-30" />
</bean>
```

A FieldSetMapper is used to convert the FieldSet into an object that will be passed to the writer.

Example 19. Field Set Mapper Implementation

```
public class CustomerCreditFieldSetMapper implements FieldSetMapper<CustomerCredit> {
    public CustomerCredit mapFieldSet(FieldSet fieldSet) {
        CustomerCredit customerCredit = new CustomerCredit();
        customerCredit.setId(fieldSet.getInt(0));
        customerCredit.setName(fieldSet.readString(1));
        customerCredit.setCredit(fieldSet.readBigDecimal(2));
        return customerCredit;
    }
}
```

Writing to a Flat File

Spring Batch includes the FlatFileItemWriter for writing to flat files. This writer requires a Resource to indicate the location of the file that should be written. It also requires a LineAggregator, whose job is to convert an item of a particular type into a String that can be written to the file.

Example 20. Flat File Item Writer Configuration

```
<bean id="itemWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" value="classpath:/data/output/myFile.txt"/>
    <property name="lineAggregator" ref="myLineAggregator"/>
</bean>
```

To write to a delimited file, a DelimitedLineAggregator should be used while the FormatterLineAggregator should be used for a fixed-length file. Both of these aggregators make use of a FieldExtractor whose job is to convert the item into an array of fields.

Example 21. Field Extractor Implementation

```
public class CustomerCreditFieldExtractor implements FieldExtractor<CustomerCredit> {
    public Object[] extract(CustomerCredit item) {
        Object[] fields = new Object[3];
        fields[0] = item.getId();
        fields[1] = item.getName();
        fields[2] = item.getCredit();
        return fields;
    }
}
```

The DelimitedLineAggregator allows for the specification of a delimiter that will be inserted between the fields in the array as the String is created. The FormatterLineAggregator requires the specification of a "format" string that the aggregator will use to format the array of fields into a String using Java's Formatter.

Example 22. Delimited Line Aggregator Configuration

```
<bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
    <property name="fieldExtractor" ref="customerCreditFieldExtractor"/>
    <property name="delimiter" value=","/>
</bean>
```

Example 23. Formatter Line Aggregator Configuration

```
<bean class="org.springframework.batch.item.file.transform.FormatterLineAggregator">
    <property name="fieldExtractor" ref="customerCreditFieldExtractor"/>
    <property name="format" value="%-4s%-15s%-6.2f" />
</bean>
```

LISTENERS

A **listener** is a class that is able to intercept the execution of a batch job by executing specified methods at various times during the life cycle of the job. For example, listeners exist that can execute methods before a job, after each chunk, when a record is skipped, and various other times.

In order for the framework to know about a listener, it must be registered with the job or step. That is, it must be configured as a property of the job or step in the job configuration.

Listeners are represented using various interfaces. A listener can be defined by extending one of these interfaces and implementing the particular method desired.

In addition to the listener interfaces, an annotation is provided for every method that appears on a listener interface. For instance the annotation @BeforeStep performs the same way as the implementing StepExecutionListener and implementing the beforeStep method. The annotations are simpler to use than the interfaces because an annotation can be placed on a method with any name and the class does not need to implement the interface. Keep in mind that a class with listener annotations must still be registered in order for the framework to know about it.

Job Listeners

A job listener can be configured to execute certain methods before or after the job. A job listener should implement the JobExecutionListener interface or use the corresponding annotations. Job listeners are configured using the <listeners/> element on the <job/>.

Example 24. Job Listener Configuration

```
<job ... >
    <step ... />
    <listeners>
        <listener class="com...MyJobExecutionListener"/>
    <listeners>
</job>
```

Step Listeners

There are several step listener interfaces, all of which extend StepListener: ChunkListener, ItemProcessListener, ItemReadListener, ItemWriteListener, SkipListener, and StepExecutionListener. Step listeners are configured using the <listeners/> element on the <tasklet/>.

Example 25. Step Listener Configuration

```
<step ... >
    <tasklet ... >
        <listeners>
            <listener class="com...MyStepListener"/>
        <listeners>
    </tasklet>
</step>
```

RUNNING A JOB

Batch jobs are usually launched from the command line. This can be done by directly running a Spring Batch class or, more frequently, through the use of a shell script. To facilitate the launching of jobs through a command line interface, Spring Batch provides the CommandLineJobRunner, which is a class with a main method that can be executed from the shell. The takes CommandLineJobRunner takes all of the parameters required for launching a batch job:

Table 2. CommandLIneJobRunner Parameters Table

Parameter	Description
jobPath	The location of the job's XML configuration file that will be used to create the ApplicationContext
jobName	The name of the batch job as specified by the "id" attribute of the <job/> element in the XML configuration
jobParameters	The job parameters used to create or identify the Job Instance

The batch job can be called in the following way:

```
bash$ java CommandLineJobRunner endOfDayJob.xml endOfDay
schedule.date(date)=2008/01/01
```

The job parameters are given in the form "key=value" and are space delimited. Therefore, if more job parameters are required, they can simply be listed at the end a command such as the ones above.

TESTING

As with other application styles, it is important to unit test at all levels. Unit tests should be written for all custom components in a batch application including services, item processors, item writers, as well as others. In addition to these unit tests, it also useful to write "functional" or "integration" tests for batch jobs. These tests apply an "end-to-end" approach to testing. For a functional test, one should provide input data like the data that will be used in production. The batch job can be run with this data as input and its result can be verified with assertions.

Creating a Test Class

In order for JUnit to be able to execute a batch job, the ApplicationContext must be loaded. Spring provides two annotations for this purpose.

Example 26. Job Test Class Annotations

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "/simple-job-launcher-context.xml",
                                    "/jobs/myJob.xml" })
public class MyFunctionalTests extends AbstractJobTests { ... }
```

Notice that the @ContextConfiguration annotation is where one specifies the locations of the XML files containing the ApplicationContext configuration.

The test class should extend AbstractJobTests, which is provided by Spring Batch's test project. This base class contains methods that will be useful for launching batch jobs from the test.

Testing a Job

To run an end-to-end batch job test, AbstractJobTests contains a method launchJob that will execute the batch job. This allows a test to set up data, launch the job, and then verify that the job was successful.

Example 27. Job Test Class Implementation

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "/simple-job-launcher-context.xml",
                                    "/jobs/myJob.xml" })
public class MyFunctionalTests extends AbstractJobTests {

    @Test
    public void testJob() throws Exception {
        // Set up data
        // ...

        JobExecution jobExecution = this.launchJob();

        // Validate results
        Assert.assertEquals("COMPLETED", jobExecution.getExitStatus());
        // ...
```

```
    }
}
```

The launchJob method optionally takes a JobParameters object to use when launching the job. This is useful when testing a job that requires specific parameters. When no JobParameters are given, the test class automatically generates a unique set of parameters so that each launch will, by default, be a new Job Instance.

Testing a Step

To create an end-to-end test for a step, the test class should again extend AbstractJobTests. This class contains a method launchStep that takes a step name as a parameter and will execute the step by itself. This allows a test to set up data, launch the individual step, and then verify that the step was successful, all without having to worry about the other steps in the job.

Example 28. Step Test Method

```
JobExecution jobExecution = this.launchStep("loadFileStep");
```

REFERENCES