Spring Batch
IN ACTION

Thierry Templier
Arnaud Cogoluègnes
Gary Gregory
Olivier Bazoud

MEAP

MANNING

MEAP Edition
Manning Early Access Program

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

1 Introducing Spring Batch

This chapter covers

- the specificities of batch applications
- how batch applications fit in today's architectures
- the necessity of a batch framework like Spring Batch
- the main features of Spring Batch

The software industry, like any industry, is all about following trends. And over the last few years web, SOA-based and event-driven applications have become all the rage. Each of them bring many advantages: web applications are easy to deploy and can scale "easily", SOA allows for more decoupled applications – for your own information system or for interaction with partners – and event-driven architecture helps to reach more real-time (and also decoupled) applications. But another kind of application has been around for a long time yet hasn't raised any eyebrows: batch applications. What are batch applications? Remember, those obscure scripts that process tons of data during the night. This is a lousy definition, but it should mean something for anyone who works in the software industry! Don't worry, this first chapter is about setting up the context, so it will give a more precise definition of batch applications as well as listing their specificities.

There are many reasons why batch applications don't get much coverage: they're not new, they don't have a fancy nice-looking user interface and there are many newer (sometimes ephemeral) technologies to talk about! Nevertheless batch applications remain critical in many systems and undoubtedly have their place in today's software architectures. The goal of this introductory chapter and even of this whole book is to prove that writing batch applications is not synonym of integrating with old systems or writing boring code: it can be challenging and rewarding. It can also mean building modern architectures and applying cutting-edge algorithms and patterns.

Spring Batch, a batch-oriented framework and the topic of this book, is the living proof that you can write batch applications and still use up-to-date technologies: it builds on top of the Spring Framework, a de facto standard for writing enterprise applications on the Java platform. This book is not only about helping you master the Spring Batch framework, but also about making your batch applications move

with the times! This first chapter will introduce the main features of Spring Batch, so you can see how this modern framework helps you write batch applications that are anything but dull. But before we get into Spring Batch, let's first tackle batch applications. What are they and why should you care.

1.1 What are batch applications?

Batch applications are mainly about dealing with large amounts of data, without any human intervention. What does that mean? Imagine you want to exchange data between two systems: you can export those data as files from system A and then import them into the database of system B. Figure 1.1 shows this scenario. Exchanging data is not the only reason to have batch applications: systems need sometimes to compute data, for generating monthly financial statement, calculating statistics, indexing files, and so on.

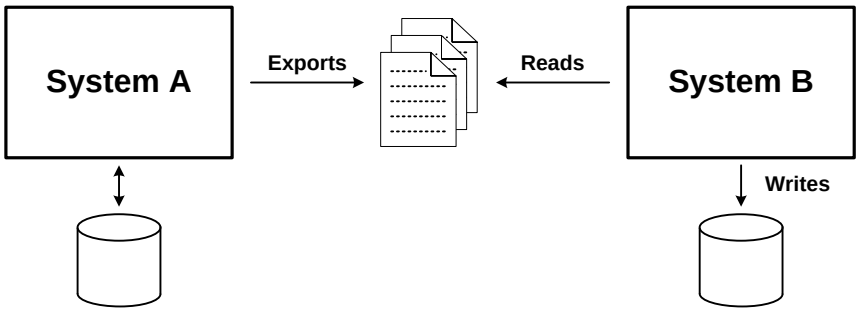

Figure 1.1 A typical batch application scenario, where system A exports data as flat files and system B uses a batch process to read the flat files and writes the data in its own database.

From these two examples we can extract the specificities of batch applications.

1.1.1 Specificities of batch applications

As a batch application deals with lots of data, it must be careful about performances (volume equals time to summarize). As a batch application processes this data automatically, it must be very robust and reliable, because there will no human intervention to recover from an error. These two characteristics (large volumes of data and automatic processing) imply the basic requirements of batch applications (robustness, reliability and performances). This is summarized in table 1.1.

Table 1.1 Summary of characteristics and implied requirements of batch applications

Specificity	Description
Large amounts of data	Batch applications handle large volumes of data to do import/export operations or computation.
Automatic processing	Batch applications process without user interaction, except for serious problem resolution.
Robustness	Batch applications must be able to process improper records without

	crashing brutally.
Reliability	Batch applications must take the right decisions (logging, notification) when something goes wrong.
Performances	Batch applications must perform well to finish processing in the dedicated window or to avoid disturbing any other applications running simultaneously.

As a batch application is in charge of dealing with a lot of data automatically (understand without human intervention), it has to be very robust and reliable. Robust means it should scale even for very large amounts of data and that it shouldn't stop the processing when something goes wrong. In the event of an error, the batch application should log the error and be able to decide whether it should continue (error that can be skipped) or stop immediately (fatal error). That's what makes it reliable.

If you're dealing with tons of data, you need to take specific care of how you process them, so that the batch application doesn't perform poorly. This is especially true because batch applications have usually only a window they can execute in, for example during the night, when nobody uses the system. So batch applications must be very careful with the way they work with memory, network and inputs/outputs (I/O). For example, large files shouldn't be loaded in memory at once, especially if memory is a sensitive resource. If online applications and several batch processes run on the same machine – and despite memory can be considered as cheap – the machine can run out of memory. This can lead to brutal crashes or performance bottlenecks, because the operating system falls back using swap memory heavily.

A solution for reading large file efficiently is to *stream* them, that is to say reading small part of them one after the other. This is more effective in terms of memory usage, but more difficult to achieve, especially when it comes to exploit the data, as the exploiting code ends up being lost in the middle of technical code.

What about databases? For reading data, solutions like paging or cursors allow reading large datasets little by little, instead of retrieving them at once. This is equivalent to streaming database results, but again, streaming code is tricky and exploiting the data is hard to isolate from the reading. For writing data to a database, Java provides batch features that reduce the number of roundtrips to the database server. By using batch updates, the optimization is made on the I/O, as one network roundtrip is used to send several operations instead of one. Batch updates can lead to drastic performance improvements.

We're not going to list all the techniques to handle large volumes of data to, as several chapters of this book are dedicated to this topic. The point is that batch applications must implement such techniques to perform well.

Now you know more about the characteristics and implied requirements of batch applications, let's compare them with more traditional applications.

1.1.2 Differences between batch and other Java applications

Today, resources like books and online documentations don't cover the design of batch applications; they are more about creating web and desktop applications, or deal with web services technologies. So, the vast majority of developers, architects, project managers are more familiar with these topics than with the design and the constraints of batch applications. We're going to explain the main differences between a traditional web application and a batch application. Table 1.2 summarizes these differences.

Table 1.2 Main differences between web and batch applications

Feature	Web applications	Batch applications
Volumes of data	Small *	Large
Transaction management	One transaction per request **	Transactions can span the processing of several records **
User interaction	Yes	No ***

* paging for search results, one record edited at once, mainly for ergonomics reasons
** most common transaction management policy
*** only for serious problem resolution

AMOUNT OF DATA INVOLVED

The first specificity of batch applications is the amount of data they can deal with. A web application can also work with lots of data, but usually, it won't read 1 million of rows from the database. Why? Because it will be catastrophic for performances? This is true, but mainly because the end-user won't be able to exploit all these data! The bottleneck here is the user itself. Web applications use then techniques like filtering through search forms and paging to limit the amount of data that displayed to the user.

What about writing data? Again, a web application can't ask a user to fill in a web form with thousands of records and submit it; this is against the rules of ergonomics. Usually, a web application displays a form dealing with one or several related records, the user fills in the form and submits it to the server. This will trigger some update operations in the database and these operations will happen in a single transaction.

TRANSACTION MANAGEMENT

This leads us to a major difference between web and batch applications: the management of transactions. In web applications, a user request is usually handled in a single transaction. This is a simple scenario, but it is the most common case. This makes things quite simple: you deal with the user inputs (HTTP parameters for a web application), open a transaction, call some business services and commit the transaction (this is the "sunny day" case; you'll have to rollback the transaction on a "rainy day"). What about transaction management with batch applications? The answer is the worst one you can make to a desperate developer: it depends. Let's say you are importing data into a database. If you want this import to be an all-or-nothing operation, you should use a single transaction for the whole import. It means that if you fail to import one single record, you'll end up with nothing imported. If you want to import as more records as possible, you should create a transaction for each record of the import. But transactions are expensive: creating one for each record can take too long, so perhaps you should create one for each hundred of records. A good trade-off can consist in opening a transaction and processing a fixed number of records in it. Figure 1.2 illustrates common transaction management strategies for both web and batch applications. Like the optimization of read/write operations, transaction management is a very tricky topic, and that's where a framework like Spring Batch can help.

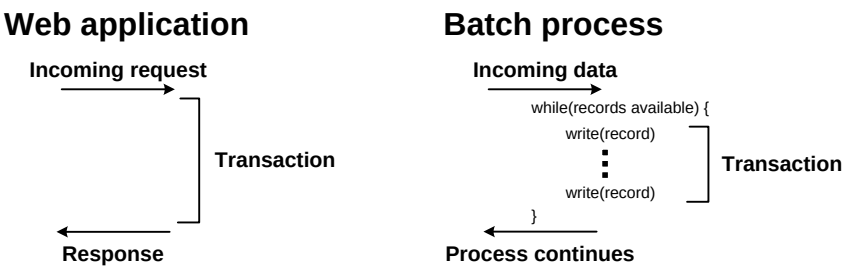

Figure 1.2 In a web application, a transaction usually spans the whole incoming request. In a batch process, transaction management policies are driven by business and technical requirement. Creating a transaction for a fixed number of records is quite common, as it offers a good trade-off between robustness and performances.

USER INTERACTION

Another major difference between a batch and a web application is the user. In a web application, if some inputs are incorrect, you can ask the user to modify them. If the user meets a fatal error because of a bug, you can ask them to come back later or to raise an issue. This is not an ideal solution – the user can be stuck (meaning you loose money!) – but at least, they can decide to retry, leave, phone, or anything else. There's no user in batch applications. It means that, when something goes wrong (and there'll always be something going wrong), the batch needs to take a decision. What to do? Again, it depends, mainly on the business requirements. We mentioned some solutions in the previous section, but the point is that batch applications are on their own, and if you don't want to find that you're processing data in a inconsistent way for months or that you're not processing them because the batch process fails at the beginning each night, you should write robust and reliable batch applications. Monitoring, notifications, logging will be your best friends!

WHAT DO THESE DIFFERENCES IMPLY?

So, batch applications are quite different from other applications. We don't say here that web applications are simpler to write than batch applications, as writing web applications can be a very challenging task. But web applications have benefited these last years from more efforts, like studies, books, patterns, and frameworks, than batch applications. To summarize, batch applications have their own requirements that differ from those of other applications: batch applications should then not be designed and implemented in the exact same way, by using the same patterns, techniques and tools.

By now, you should have a good overview of what batch applications are, but do we really *need* batch applications in today's architecture?

1.1.3 How batch applications fit in today's software architectures

To define batch applications, we took 2 examples which were simple but quite representative of the two most common kinds of batch processes, which aim at:

- Doing computation
- Exchanging data between applications

The two kinds are not exclusive: an application can export data, and another application reads them, does some computation and writes the results in a persistent store (the second application uses the same data, but under another representation). Let's see if both kinds of processes are relevant today.

ARE COMPUTATION-BASED PROCESSES OBSOLETE?

Computation-based processes are obviously relevant: don't you hear quite often about monstrous calculations done to index billions of documents, using cutting-edge algorithm like MapReduce and distributed over hundreds or even thousands of nodes? Ok, these are not batch processes you meet everyday, but they exist and many enterprises need at least to compute data, on a daily, weekly, or monthly basis, even if these data do not represent billions of documents. So computation-based processes are still relevant, more than ever, as data are everywhere and we need some ways to digest, aggregate, summarize them, just to be able to make something out of them.

IS EXCHANGING DATA BETWEEN APPLICATIONS WITH FILES OBSOLETE?

What about exchanging data between applications? Making applications communicate with each other is usually referred to as *integrating* applications and we'll use the term *enterprise integration* to refer to the practice of integrating applications in enterprise systems. In their "Enterprise Integration Patterns" book, Gregor Hohpe and Bobby Woolf distinguish 4 approaches to enterprise integration:

- File transfer – an application writes files that another reads. This is a very common use case for batch processes.
- Shared database – applications share the same database, so that no transfer is needed.
- Remote procedure invocation – applications expose their functionalities so that they can be accessed remotely by other applications.
- Messaging – applications send messages that other applications can consume on channels.

File transfer seems to be the most basic approaches and in an event-driven world, where you get notified by your cell phone and applications on your computer, messaging looks more appealing and in fashion. So using batch processes for exchanging data between applications should not be used in new systems? Well, this is not so simple. First, messaging is a very powerful and interesting way to design applications but it has requirements of its own in terms of design and technical solutions. So a good idea is to avoid putting messaging everywhere in your system. Second, batch processes and messaging are not exclusive: messages can aim at exchanging data and you can still need to process these data in batch, with the same reliability and robustness requirements as when reading large files. We believe so much in the ability of making batch and messaging solutions cohabit that we cover in chapter 11 (dedicated to enterprise integration) how to use Spring Batch and Spring Integration.

So the 2 most common use cases for batch applications (computation and exchange-based processes) are still relevant in today's architectures. This should be enough to be confident in the usefulness of batch applications and also in the usefulness to discover a framework like Spring Batch.

1.2 Meeting Spring Batch

Spring Batch is about simplifying the development of batch applications. Table 1.3 lists the main features of Spring Batch, we'll detail them in this section, starting with the building block of Spring Batch, the Spring Framework.

Table 1.3 The main features of Spring Batch

Feature	Description
Foundations in the Spring Framework	Benefits from dependency injection, aspect-oriented programming and enterprise support.
Batch-oriented runtime	Drives efficiently the flow of an batch application.
Efficient data processing	Enforces best practices for reading and writing data.
Ready-to-use components	Provides components to address common batch scenarios (read/write data from/to databases, files)

1.2.1 The Spring Framework as a foundation

Spring Batch stands on the shoulders of a giant: the Spring Framework. This has several implications on the support that Spring Batch builds on (as shown on figure 1.x) and also on the programming approach that it promotes.

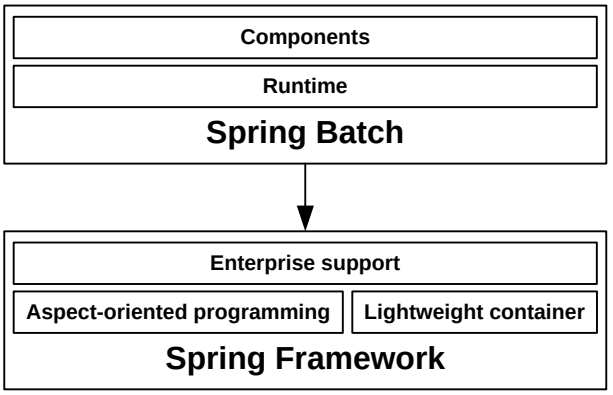

Figure 1.x Spring Batch builds on top the Spring Framework, so it can leverage its lightweight container for configuration, the aspect-oriented programming framework to address cross-cutting and the enterprise support to integrate with enterprise systems like databases.

The origins of the Spring Batch project

The Spring Batch project was born in 2007, from the collaboration of Accenture and SpringSource. Accenture brought its experience gained on years working on proprietary batch frameworks and SpringSource came with its technical experience and the proven Spring's programming model. The primary goal of the Spring Batch project is to provide an open source batch-oriented framework that addresses in an effective way the most common needs of batch applications.

NON-INTRUSIVE DEVELOPMENT APPROACH

Spring enforces separation of concerns: application (understand business) code should not be strongly coupled with technical code. Spring Batch follows the same path, even you'll need to implement some interfaces or extends some classes from the framework API. Note that less intrusive solutions are sometimes available as Spring Batch provides annotations or adapters to plug on POJOs (Plain Ordinary Java Objects).

POWERFUL CONFIGURATION FEATURES

Spring is well known as a lightweight container: it lets you provide configuration instructions (for example in XML files) and takes care of assembling the components of your application. All Spring Batch's components are meant to be configured by Spring's lightweight container, leveraging dependency injection and some common hooks (complex object instantiation, initialization callbacks, dedicated scope).

Spring Batch provides a very useful extension of the Spring framework: a dedicated XML namespace to configure batch processes. This namespace provides XML elements that ease configuration and make it more readable, like shown in the following snippet:

```
<job id="importInvoices">
  <validator ref="jobParametersValidator" />                          #A
  <step id="decompress" next="readWriteInvoices">                     #B
    <tasklet ref="decompressTasklet" />                               #B
  </step>                                                             #B
  <step id="readWriteInvoices">                                       #C
    <tasklet>                                                         #C
      <chunk reader="reader" writer="writer"                          #C
             commit-interval="3" skip-limit="5">                      #C
        <skippable-exception-classes>                                 #C
          <batch:include class="                                      #C
[CA]  o.s.b.item.file.FlatFileParseException"/>                       #C
        </skippable-exception-classes>                                #C
      </chunk>
    </tasklet>
  </step>
</job>
```

#A Sets input validation strategy
#B Defines first step (decompression)
#C Defines second step (read/write)

Even if you don't know anything about Spring Batch, the XML elements are meaningful enough to understand what different steps of the batch process are about and how they transition.

POWERFUL DATA ACCESS SUPPORT

Spring provides integration with popular data access technologies: JDBC, JPA, Hibernate, and iBatis to name some of them. This support is about configuring resources easily (a `DataSource` for example) but also about providing helper classes that provides more straightforward API or missing key features, like automatic conversion of `SQLExceptions`.

With this data access support comes an abstraction for transaction management, which can be used programmatically or declaratively, allowing making Java objects transactional with XML configuration or annotations. This transaction management abstraction is portable across the data access technologies

previously mentioned, meaning you can use the same transaction API or configuration for different technologies. This is exactly what Spring Batch does: all its transaction management is coded against Spring's transaction API, which means that your Spring Batch applications will be able to use any data access technology supported by Spring and benefit from Spring Batch's powerful, batch-oriented transaction management.

SMOOTH INTEGRATION WITH SPRING APPLICATIONS AND PROJECTS

The Spring Framework is used massively in enterprise applications and any Spring application can easily embed Spring Batch and benefit from its features. Once embedded, a Spring Batch job can re-use existing business code, as long as the latter conforms to batch processing. Embedded, but where? Anywhere Spring can run: Java EE web container (Tomcat, Jetty, Resin…), Java EE application servers (Glassfish, JBoss…), OSGi container (Eclipse Equinox, Apache Felix…) or as a standalone JVM process.

So you can use Spring Batch in your Spring applications or standalone. That's good news, but note that Spring Batch can also benefit from the projects of the Spring portfolio. Want to run your Spring Batch processes in your OSGi container? Don't omit to use Spring Dynamic Modules (also known as Eclipse Gemini Blueprint), the bridge between Spring and OSGi. Want to add an event-driven taste to your batch application? Try Spring Integration! We cover in this book how to trigger your batch processes, to monitor them and making them scale with this framework.

We're not trying to overwhelm you with everything you can do with Spring Batch but rather emphasize that this framework has very strong foundations as it builds on top of the Spring Framework. Moreover, if you're familiar with Spring, you're not starting from scratch when choosing Spring Batch for your batch applications. It's time to see now the second feature listed in table 1.x, the batch-oriented runtime that Spring Batch provides.

1.2.2 Batch-oriented runtime

When we speak about a batch-oriented runtime, we refer to the way Spring Batch can drive the flow of a batch process. As soon as you'll use Spring Batch, it will be in charge of orchestrating the flow of your batch application: when and how to read records from the database, when to open a stream to file, when to commit the transaction and so on. At some places in this flow, let's say inside a transaction, Spring Batch will call you own code to perform the core business operation. Figure 1.x illustrates how Spring Batch can drive the application flow and call business code appropriately.

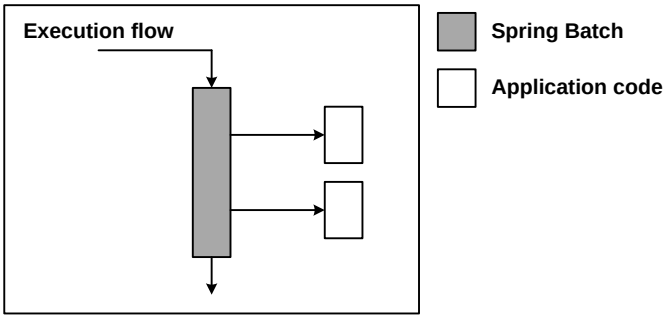

Figure 1.x Spring Batch handles the execution flow of the batch application and calls the business code appropriately. Around the business code, Spring Batch can deal with I/O, transactions or any other technical concern.

The good news is Spring Batch drives the flow of the application in a way adapted to batch processes. You can tweak the way the flow is running: imagine paging the records from the database is useful in your case, then you'll be able to choose the size of the page (10, 100 or 1000 records). If tweaking is not enough, Spring Batch can let you drive the flow yourself: imagine paging is not adapted, you can implement your own way to read records from the database.

An immediate benefit of the control that Spring Batch has over the execution flow is that it can enforce best practices to achieve efficient data processing.

1.2.3 Efficient data processing

We spoke about dealing with large volumes of data in batch applications, let's see how Spring Batch does this efficiently through a classic scenario: reading records from a source (for example a flat file) and write them to another data store (for example a database). A simple approach is to load the whole file in memory and handle its Java representation (a `List` object or a DOM tree) to a writing component that will iterate over the records. As mentioned earlier, loading the whole file in memory is not a good practice, especially when the file is large (more than hundreds of megabytes, which is quite common).

Spring Batch uses a more efficient approach, called chunk processing: it streams the input source and handles a fixed number of records to the writing component. It re-iterates this operation as long as the input source provides records. Figure 1.4 illustrates chunk processing.

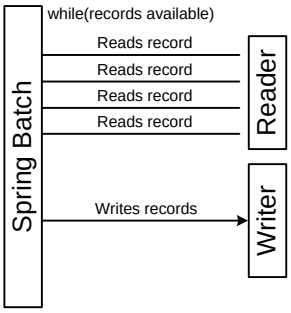

Figure 1.4 For read/write scenarios, Spring Batch uses chunk processing, by streaming the input source to read chunks of records and sending them to the writing component

Chunk processing has many advantages:

- Flexibility (you can stick to a record-per-record processing by choosing one as the chunk size)
- Optimization (streaming on the source side, allows for batch updates on the writer side)
- Decoupling between the infrastructure, the reader and the writer

Spring Batch provides the infrastructure to perform chunk processing and delegates I/O to dedicated components, namely the reader and the writer. Let's see what kind of I/O implementations Spring Batch provides.

1.2.4 Ready-to-use components for common batch scenarios

We introduced in the previous sub-section a very common scenario in batch applications: reading data from a data source and writing the data (transformed or not) in another data source. We also emphasized that the amount of data can be very large, so the reading and writing processes should use appropriate techniques, like streaming files or paging database results. The good news is Spring Batch provides ready-to-use components for the most common data sources. We're going to see how to use them before listing the implementations that Spring Batch provides.

CONFIGURING A READ COMPONENT

Imagine that in your read/write scenario the data to be read are in a database. To do this, Spring Batch provides a `JdbcPagingItemReader` that will read the data in an efficient way, as it will use paging. What you end up doing is mainly some configuration work to tell the `JdbcPagingItemReader` where to read the data from and the size of the page, as shown in the following snippet:

```
<bean id="reader" class="o.s.b.item.database.JdbcPagingItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="queryProvider">
    <bean class=" o.s.b.item.database.support.
[CA] SqlPagingQueryProviderFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="selectClause" value="                          #A
[CA] select id,customer_id,issue_date,amount"/>                      #A
      <property name="fromClause"                                    #A
                value="from invoice"/>                               #A
      <property name="sortKey" value="id"/>                          #A
    </bean>
  </property>
  <property name="pageSize" value="100"/>                            #B
  <property name="rowMapper">                                        #C
    <bean class="com.manning.sbia.ch01.chunk.                        #C
[CA] InvoiceRowMapper" />                                            #C
  </property>
</bean>
```

#A Specifies SQL query
#B Sets page size
#C Specifies application component for processing results

By providing a ready-to-use reader component, Spring Batch lets you focus your business code. What is the business code in this case? It's made of two parts: the SQL query configured in the Spring configuration file and an application class – the `InvoiceRowMapper` – that converts the database result set into domain objects (`Invoice` instances in the example). Note the reader component implements a typical batch application best practice, as it uses paging, to avoid loading many instances in memory. You, as an application developer, don't have to deal with technical concerns like paging, JDBC, transaction and so on.

Let's see now how Spring Batch can help us to write data in another data source.

CONFIGURING A WRITE COMPONENT

Imagine the write part of your read/write scenario should be done in a flat file. Spring Batch provides a `FlatFileItemWriter` that can write your domain objects into an output file. Again, you won't have to deal with technical code; your job will consist in some configuration, as shown in the following snippet:

```
<bean id="writer" class="o.s.f.item.file.FlatFileItemWriter">
  <property name="resource"                                    #A
            value="file:target/output.txt" />                  #A
  <property name="lineAggregator">                             #B
    <bean class="o.s.f.item.file.transform.                    #B
[CA] PassThroughLineAggregator"/>                              #B
  </property>                                                  #B
</bean>
```

#A Specifies output file
#B Specifies domain object to line converter

When you use the `FlatFileItemWriter`, the conversion between a domain object and a line in the output file is done by a `LineAggregator`. For the simplicity's sake, we used here a very basic implementation (`PassThroughLineAggregator`) that calls the `toString` method of a domain object to generate the content of the line (remember that your domain objects are produced by the `InvoiceRowMapper` injected into our database reader). Imagine the `toString` method of our `Invoice` class looks like the following:

```
public class Invoice {

  (...)

  @Override
  public String toString() {
    return id + "," + customerId + "," + amount + "," + issueDate;
  }

}
```

Then a line in the output file would look like the following:

```
PR....214,9737,102.23,2010-11-18
```

If this rather simple line generation strategy does not suit your needs, Spring Batch provides other configurable implementations. You can also provide your own implementation, as implementing the `LineAggregator` interface is straightforward (it has only one method).

This write example shows again that Spring Batch handles all the technical concerns. It provides many off-the-shelf components but is also flexible enough to let you plug on your own implementations.

Now that you have a good idea of what we mean when we speak about ready-to-use components provided by Spring Batch, let's see a more comprehensive list of them.

LISTING FORMATS AND TECHNOLOGIES SUPPORTED BY SPRING BATCH

We covered how to configure read and write components for a database and a flat file. These are typical uses of Spring Batch's components, but Spring Batch's support for read/write scenarios is by no mean restricted to these data sources (database, file) and formats (JDBC, flat file). Table 1.2 gives a more comprehensive list of the storage technologies that Spring Batch supports out-of-the-box.

Table 1.2 Supported technologies by Spring Batch for read/write scenarios

Data source type	Technology/format	Description
Database	JDBC	Leverages paging, cursors and batch updates.
Database	Hibernate	Leverages cursors.
Database	JPA	Leverages paging.
Database	iBatis	Leverages paging.
File	Flat file	Supports delimited and fixed length flat files.
File	XML	Uses StAX for parsing, builds on top Spring OXM, so support tools like JAXB, XStream or Castor.

You see from table 1.2 that Spring Batch supports many technologies out-of-the-box, making it quite versatile. We'll study thoroughly this support in chapters 6 and 7.

Spring Batch is not limited to this support as it is also flexible at different levels: each component provides many hooks where you can plug your own implementations. Then, if no Spring Batch's component fits your needs, you can implement your own read or write components, by implementing straightforward interfaces (`ItemReader` and `ItemWriter`, respectively). And at last, Spring Batch does not limit you to read/write scenarios, as batch applications are also about moving files, calling stored procedures or web services and so on. So a Spring Batch process is usually made of read/write steps but also of more specific steps.

Spring Batch is not a scheduler!

Spring Batch drives batch processes but does not provide advanced support to launch them, especially on a time basis. Spring Batch leaves this job to dedicated tools like schedulers (Quartz or Cron to name of few). A scheduler usually triggers the launching of Spring Batch processes, by accessing to the Spring Batch runtime if it can (Quartz for example, as it is Java solution) or by launching a dedicated JVM process (Cron for example). Sometimes a scheduler launches batch processes in sequence: first process A and then process B if A succeeded or process C if A failed. The scheduler can use files generated by the processes or exit codes to organize the sequence. Spring Batch is also capable to orchestrate such sequences: Spring Batch's jobs are made of steps and the sequence of steps can be easily configured thanks to the XML namespace or Java annotations (this is covered in chapter 10). This is where we can say that Spring Batch and a scheduler overlap.

This ends our discovery of the core features of Spring Batch. We're sure that you've been happy to learn how Spring Batch can free you from cumbersome technical code like I/O handling to let you focus on the business code. In the next section, we're going to explore other Spring Batch's features through use cases. These features are mainly about making your batch applications more robust and scalable.

1.3 Use cases

We covered in the very first section of this chapter the specificities of batch applications: they handle large amounts of data through automatic processing and as so, they must be very robust and reliable.

Spring Batch provides strong foundations for handling these specificities, especially in the way it runs batch processes. This is what we can call the runtime side of Spring Batch features. To show you in which situations Spring Batch can be very useful for you batch applications, we're going to see different runtime scenarios (transaction management, error handling and making batch application scale) and how Spring Batch handles them.

1.3.1 Handling transactions

In read/write scenarios, Spring Batch is able to manage transactions for you. It means that any operation on a database will be run inside a transaction. Spring Batch handles the transaction creation and whether it should be committed (in case of success) or rolled back (in case of failure). This is a very interesting feature because your code won't be cluttered with transaction management. Moreover, as Spring Batch's transaction management builds on top Spring's one, it supports native database transaction, JTA, Hibernate and so on. It means you can switch from transaction management technology to another without impact on your code.

Another benefit of letting Spring Batch drive transactions for you is that it can do so in a batch-oriented way, which helps you handle large volumes of data. When doing a bulk of inserts in a database, you usually don't want to have only one transaction spanning all of them: in case of error, all the inserts will be lost, and it forces the database to maintain a large rollback segment. You don't want either to have on transaction for each insert: transactions aren't cheap and doing so can have dramatic impacts on performances. The best strategy is usually to handle records in… batch! It means that you want to handle 10 or 100 or 1000 records in one transaction, the number of records being called the *batch size*.

Doing so is not difficult, but doing it for all your read/write operations becomes quickly cumbersome, especially when it comes to handling errors (more on this later!). For read/write scenarios, Spring Batch allows to set a batch size, as shown in the following snippet:

```
<batch:chunk reader="reader" writer="writer" commit-interval="100" />
```

By setting the `commit-interval` attribute to 100, we tell Spring Batch to ask 100 records to the reader, open a transaction, send the records to the writer and commit the transaction. Externalizing the batch size is very interesting as there's no "best" value for this setting: it depends on many factors like the writing instructions, the data, or the database. Being able to set the batch size without impact on the code simplifies tweaking the batch processes, when doing performance tests.

Committing transaction happens on sunny days, but how Spring Batch handles errors? Errors can have impacts on transaction but also on the whole batch process.

1.3.2 Handling errors

Batch processes handle a lot of data automatically and many things can go wrong: incorrect format in input files, violation of database constraints, bugs and so forth. Usually, a batch process is not all-or-nothing operation: you don't want to stop the whole process because of a tiny error. This is one of the specificities that can make batch applications really tricky to write: foresee errors and handle them.

Imagining reading records from a flat file before inserting them into a database: if a line doesn't respect the format, what should we do? By default, Spring Batch will launch an exception and stop the process. The following snippet shows the configuration for this default behavior:

```
<chunk reader="reader" writer="writer" commit-interval="100" />
```

But perhaps you can live with an incorrect line, so why not skipping it and letting the show going? If you know about the types of exception that the reader can launch in case of incorrect data, you can ask Spring Batch to skip the item and read the next one, as shown in the following snippet:

```
<chunk reader="reader" writer="writer"
       commit-interval="100" skip-limit="5">
  <skippable-exception-classes>
    <batch:include class="o.s.b.item.file.FlatFileParseException"/>
  </skippable-exception-classes>
</chunk>
```

The previous example implies that the reader can launch `FlatFileParseExceptions` when something cause wrong, and it this case, we want Spring Batch just to ignore the exception and keep reading. Any other exception would make Spring Batch stop the batch process. Note the `skip-limit` attribute set to 5: it tells Spring Batch to stop the process as soon as 5 records have been skipped, meaning you can live with incorrect lines, but not too many.

And what happens if Spring Batch had no other choice than stopping the process? A common need is to restart the job exactly where it failed once the reason of the failure has been found and solved. Indeed, it would be a pity to re-process millions of records when only one of the lasts couldn't have been processed. The good news is Spring Batch can store in a database all the information about the batch processes it performs. We'll see in chapter 3 what kinds of information are stored. The point is Spring Batch can use them to restart a batch process where it exactly stopped. Storing batch execution data is also interesting for monitoring, as you can plug a user interface on the database to browse the data and detect any problem. We'll study monitoring Spring Batch applications in chapter 12 and learn more about error handling with Spring Batch in chapter 9.

We saw how Spring Batch handles robustness and reliability, now let's see how it can help to scale batch processes when they face performance issues.

1.3.3 Scaling batch processes

Batch processes have usually a window they can execute in, but sometimes this window gets too small and you'll need to make your batch processes execute faster. Spring Batch provides support for executing your batch jobs in the same JVM process or in several JVM processes. There are also different strategies to choose from for the execution of the steps of the batch (splitting chunk writings, execute steps in parallel). We are not going to cover all the combinations here but only give an overview, as chapter 13 is dedicated to the topic.

In a read/write scenario, a first strategy for scaling is to split the chunk writings. You can choose to share the chunk writings between several threads, in the same process. This is especially useful when you're running on multi-core hardware. Figure 1.5 illustrates splitting chunk writings between threads.

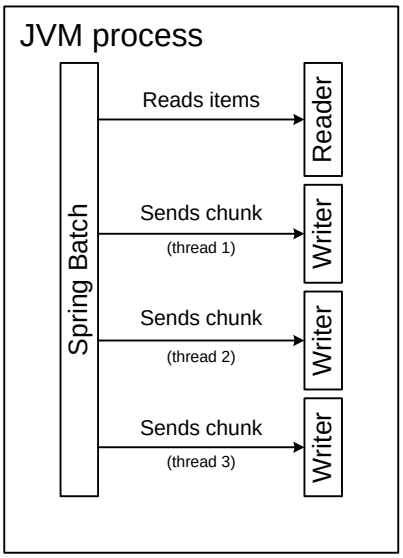

Figure 1.5 Scaling by splitting chunk writings between threads. Spring Batch reads records and accumulates in chunks to send these chunks to writers, in dedicated threads. This can help scaling when reading is faster than writing and when using multi-core hardware.

You can also distribute the chunk writings on several servers. In such a configuration, records are read from a master node, grouped into chunks and sent to slave nodes. The communication between the master and the slaves implies using a middleware and Spring Batch leverages Spring Integration to stay agnostic to the underlying middleware implementation (JMS is an example of such a middleware). Figure 1.6 illustrates splitting chunk writing between several processes.

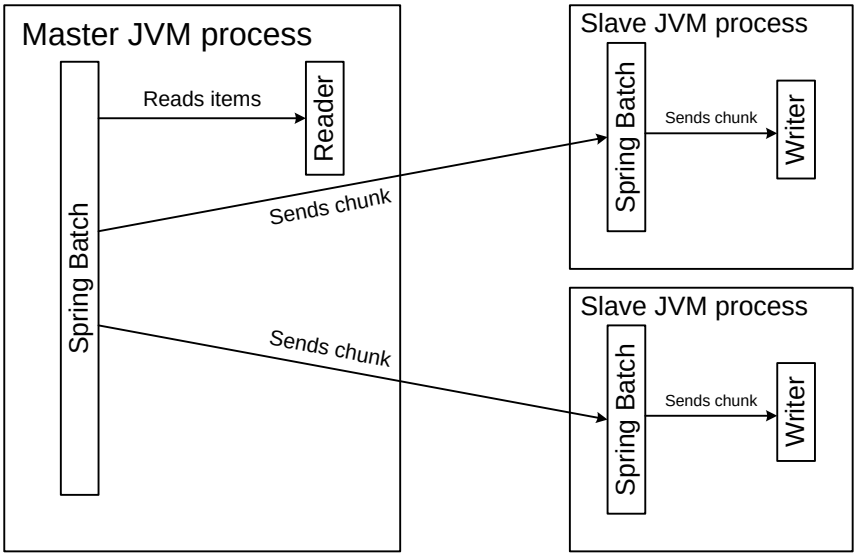

Figure 1.6 Scaling by splitting chunk writings between 2 processes. Spring Batch reads records, accumulates them in chunks and sends these chunks to writers but in different processes. The processes can live on the same machine or on different machines. This can help scaling when reading is faster than writing.

Note that splitting chunk writings is interesting only when read operations are faster than write operations, so that the writing workers remain always busy (but this is usually the case).

Another strategy for scaling is partitioning. The idea behind partitioning is to split a whole step into sub-steps that each handles a specific part of the data. This implies that you know about the structure of the input data, meaning you know in advance how to distribute them between the sub-steps (by ranges of primary keys for database records or by directories for files). The sub-steps can execute locally or remotely (Spring Batch provides support for multi-threaded sub-steps). Figure 1.7 illustrates partitioning (records could be files and they would be partitioned thanks to their names: [A-D], [E-H], up to Z).

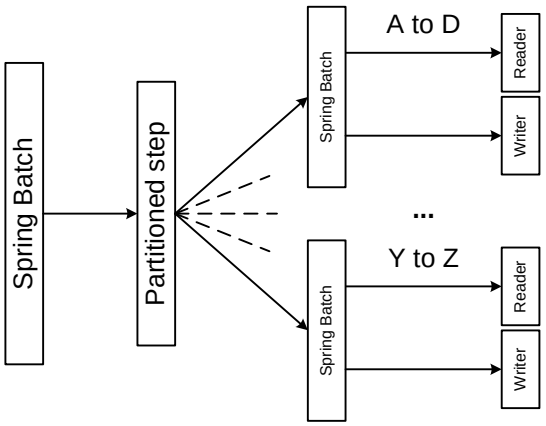

Figure 1.7 Scaling with partitioning, where records are partitioned and handled by autonomous sub-steps

This is the end of our tour of Spring Batch! You have by now a good overview of the most important features of the framework, how you can use it and how it can help you to write your batch applications. Before writing our first application with Spring Batch, we are going to compare the framework with other technical solutions for batch applications, grid computing frameworks.

1.4 How Spring Batch compares to grid computing frameworks

We emphasized that batch applications are about dealing with large amount of data and computations and that Spring Batch provides good foundations to address these requirements. We even introduced some ways to scale the processes with Spring Batch. But what happens when we really, really, really need to scale? A popular solution to scale nowadays is to divide the enormous amount of computations into smaller ones, compute them in parallel (usually on different nodes) and gather the results.

Doing so looks simple on paper, but it ends up being really tricky in practice, when you need to deal with parallelization, failover, and coordination between the nodes. Some open source frameworks (Hadoop, GridGain, Hazelcast to name a few) appeared these years to deal with the technical burden implied by distribution, so that developers can focus on developing the computations themselves.

What does “enormous amounts of data” mean?

People usually oppose grid computing frameworks to more traditional tools like relational databases. Comparing the volume that both systems are meant to deal with can give an idea of what we mean by “enormous amounts of data”. Relational databases handle pretty well gigabytes (10^9) of data and grid computing frameworks are interesting (or mandatory!) if you deal with petabytes (10^15) of data.

These frameworks are good candidates for developing batch applications, so let’s see in what these solutions differ from Spring Batch, before giving some guidelines about what to use.

1.4.1 In what grid computing frameworks differ from Spring Batch?

The two main differences between grid computing frameworks and Spring Batch are the way both solutions are deployed and the features they provide.

DIFFERENCES IN TERMS OF DEPLOYMENT

Whereas Spring Batch is mainly a framework, grid computing frameworks are also about providing a runtime environment. They also drive the execution flow, but they not only call your code when necessary, they will also manage to call it where it is necessary: they handle all the distributing part. For example, Hadoop drives its batch processes around an implementation of the MapReduce algorithm and the framework deals with distributing the processing on the nodes, gathering the results; all of this usually on top of its distributed file system, HDFS. Hadoop comes even with a dedicated database engine, HBase.

Spring Batch is more lightweight: it can run in any Java program or embedded in any Java-based runtime (a web or an OSGi container for example). Spring Batch will be able to plug on an existing system to scale (we mentioned JMS for distributing the processing of chunks) but by no means it tries to provide the necessary infrastructure to make the distributing reliable.

DIFFERENCES IN TERMS OF FEATURES

Obviously grid computing frameworks drive the flow of batch applications very efficiently, by distributing automatically the processing within a cluster of nodes. This is a very important feature for batch processing. They can also provide support to read from or write to popular data stores, like files or databases.

Usually, grid computing frameworks provide low level API to deal with I/O and that's exactly the point: they provide the minimum, but this minimum is run on an engine that can scale very easily. Thanks to that, you won't have a barrier like an SQL engine between your application and your data, but you'll have to be very careful in the way you store, query and, update your data.

Spring Batch provides advanced support for file and database access, but also for technologies that build on top of them (XML for files, Hibernate for databases for example). It means that you'll be able to leverage all these features, but you'll also have to live with their limitations, the difficulty to scale for databases for example.

1.4.2 What to use in which circumstances?

Grid computing frameworks are very interesting when you really need to scale, as they let you write applications on top of a distributed infrastructure. This scaling has a price as these frameworks do not provide some features (like easy-to-use API to deal files or databases) or loosen others you can be used to when working on more traditional systems (like transactions). That's why people usually associate such frameworks with enormous amounts of data and computations.

Spring Batch doesn't provide (yet?) any integration with grid computing framework (at least in its core distribution). It follows a more classical path, as it doesn't provide any support for NoSQL database engines either (these kinds of database engines are often associated with grid computing frameworks).

So Spring Batch for small to medium computations and grid computing frameworks for very large computations? Even if this is lousy and quick conclusion, it could be a guideline. But keep in mind that this is an architecture decision to take, so when you face such a choice, you should base you decision on real tests on both solutions.

The good news after this comparison is that most enterprise applications do not deal with petabytes of data, making Spring Batch a suitable solution for them! Enough theory for now, let's move on to a more practical part by writing a Hello World application with Spring Batch!

1.5 Spring Batch Hello World

Even a batch application can be a Hello World! We are not going to process here some data, but mainly discover the Spring Batch API, how to configure a Spring Batch's job with Spring and how to launch the process. The core of our batch job will just consist in issuing "Hello World" on the console.

1.5.1 Writing the Hello World task

In Spring Batch, a batch process is called a job and is divided in a succession of steps. Spring Batch provides different ways to implement what should happen in a step, let's see the most adapted to a Hello World application: the `Tasklet`. A `Tasklet` is an example of the command pattern as it is an interface made of one method, `execute`, which will be called by Spring Batch during the job execution.

Let's not wait no more and see the `Tasklet` implementation in the following code snippet:

```
package com.manning.sbia.ch01;

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 HelloTasklet implements Tasklet {

  @Override
  public RepeatStatus execute(                                    #A
      StepContribution contribution,                              #A
      ChunkContext chunkContext) throws Exception {               #A
    System.out.println("Hello Spring Batch World!");
    return RepeatStatus.FINISHED;                                 #B
  }
}
```

#A Called by Spring Batch
#B Tells the task is over (no need to repeat)

This is obviously a very simplistic task – real implementations of `Tasklet` would delete files once they have been processed or call stored procedures – but an Hello World cannot be complex by nature! Keep in mind that the execution of the `Tasklet` is handled by Spring Batch and we'll see all along the book that the framework is able to address many concerns around our code: transaction management, skip or retry and so on.

That's it for the task; let's see now how to configure everything Spring Batch needs to launch a job: its infrastructure and the job itself.

1.5.2 Setting up Spring Batch's infrastructure

Spring Batch relies on some Spring beans to fulfill its infrastructure work: transaction management, storage of job executions and states, launching of jobs and so on. Following Spring's philosophy, Spring Batch heavily relies on an interface-based programming model: this will help us setting a simple – yet not reliable – batch infrastructure by using mock implementations. The infrastructure configuration takes place in a Spring file, as shown in the following snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:batch="http://www.springframework.org/schema/batch"
  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-2.1.xsd">

  <bean id="transactionManager"                                          #1
        class="org.springframework.batch.support.                        #1
[CA]  transaction.ResourcelessTransactionManager" />                     #1

  <bean id="jobRepository"                                               #2
        class="org.springframework.batch.core.                           #2
[CA]  repository.support.MapJobRepositoryFactoryBean">                   #2
    <property name="transactionManager"                                  #2
              ref="transactionManager" />                                #2
  </bean>                                                                #2

  <bean id="jobLauncher"                                                 #3
        class="org.springframework.batch.core.launch.                   #3
[CA]  support.SimpleJobLauncher">                                        #3
    <property name="jobRepository"                                       #4
              ref="jobRepository" />                                     #4
  </bean>

</beans>
```

#1 Declares mock transaction manager
#2 Declares in-memory job repository
#3 Declares job launcher
#4 Injects job repository

The file starts with the declaration of namespaces: we use the common `beans` namespace for standard bean declarations. Note we also declare the `batch` namespace, which we'll use later, for the job configuration. Spring Batch uses the Spring's transaction support – based on the `PlatformTransactionManager` interface – and so needs a transaction manager bean (#1). As the Hello World does not involve any transactional data source (database, JMS), we can use a mock implementation that Spring Batch provides. A very important bean in the batch infrastructure is the job repository, which handles the storage of the jobs' data (state of execution, exit status and so forth). We use here a in-memory implementation (#2), but we'll see in the next chapters that Spring Batch provides also a database implementation. The last bean needed in the job launcher (#3), whose role is to start the job execution. Note it needs a reference to the job repository (#4) because the last execution of the job could have failed and the job repository would then provide enough information to the launcher to start the job at the exact same step it failed before.

We're done with the configuration of the batch infrastructure. Keep in mind that this configuration must be done only once (all the jobs can use it) and that Spring Batch provides other implementations for the main components, depending on the needs. Let's move on to the job configuration.

1.5.3 Configuring the job

In Spring Batch, jobs are also configured using Spring, but usually by using the dedicated `batch` namespace. The latter provides XML elements that make the configuration of jobs easier and more readable (the namespace feature of Spring is sometimes referred to as a way to define *domain specific languages*). In a real batch application, the infrastructure and jobs configuration should be split in different files but we'll keep them in the same file for the sake of simplicity. Here is the configuration of the job, which takes place just after the infrastructure part (note we eluded the infrastructure configuration):

```
<beans (...)>

  (...)

  <bean id="helloTasklet"                                          #1
        class="com.manning.sbia.ch01.HelloTasklet" />              #1

  <batch:job id="helloJob">                                        #2
    <batch:step id="helloStep">                                    #2
      <batch:tasklet ref="helloTasklet" />                         #3
    </batch:step>
  </batch:job>

</beans>
```

#1 Declares Hello Tasklet bean
#2 Declares Hello job and its step
#3 Declares tasklet-based step using Hello Tasklet

The `Tasklet` is declared as a Spring bean (`#1`), we'll refer to it in the job configuration. The job configuration starts with the `batch:job` element at `#2`. Behind the scenes, Spring Batch instantiates one or more beans from this element, but we don't care about that: this is the goal of the `batch` namespace, which encapsulates the framework mechanisms. A job is made of steps and ours has only one. We declare it with the `batch:step` element. As our step is a tasklet-based step, we use the `batch:tasklet` element and refer to the `helloTasklet` bean (`#3`).

That's it! We're done with the configuration, let's see now how to launch our batch process.

1.5.4 Launching the batch

There are many ways to launch batch processes, so many that chapter XX is dedicated to this topic! We've chosen here to launch the job in a plain old main program: this is simple and gives a first taste of Spring Batch's API. Here is the content of the launching program:

```
package com.manning.sbia.ch01;

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 SpringBatchHelloWorld {
```

```
    public static void main(String ... args) throws Exception {
      ApplicationContext context =                                    #1
        new ClassPathXmlApplicationContext(                           #1
        "spring-batch-hello-world.xml"                                #1
      );                                                              #1
      JobLauncher jobLauncher = context.getBean(                      #2
        JobLauncher.class);                                           #2
      Job job = context.getBean(Job.class);                           #2
      JobExecution jobExecution = jobLauncher.run(                    #3
        job,                                                          #3
        new JobParameters()                                           #3
      );                                                              #3
      System.out.println(                                             #4
        "Job ended with status: "+                                    #4
        jobExecution.getExitStatus()                                  #4
      );                                                              #4
    }

}
```

#1 Bootstraps Spring application context
#2 Looks up launcher and job beans
#3 Launches job
#4 Displays execution exit status

The program starts by loading the Spring application context from the configuration file (#1). This creates the infrastructure, tasklet and job beans. We then look up the launcher and the job beans from the Spring container (#2). We then pass the job to the job launcher with an empty instance of job parameters (#3). Internally, the launcher will ask to the job repository to create a job execution instance and then will launch it. The batch processing happens synchronously: this is the default behavior of the `SimpleJobLauncher`. We then display on the console the exit status of the execution of the job (#4).

By executing the program, you should see the following:

```
Hello Spring Batch World!
Job ended with status: exitCode=COMPLETED;exitDescription=
```

Congratulations for your first use of Spring Batch! This sample is simple but it shows you that Spring Batch will now care of the execution of your batch processes. Imagine that you will be able to choose easily to commit the database transaction every 10 or 20 rows, to fail a batch when a specific exception is thrown or just skip the offended row, etc. All of that without changing your business code, thanks to the declarative approach Spring Batch provides.

1.6 Summary

This chapter covered the concepts of batch applications: they are critical, sensitive applications that need special care to be written. In our data-oriented world, batch applications can help us to digest and analyze all these data and that's what makes them important and even interesting to write. Batch applications can be very challenging to code and a framework like Spring Batch can relieve the developer from writing some cumbersome and difficult technical code.

Spring Batch has strong foundations thanks to the Spring Framework and provides many ready-to-use components, making the writing of batch applications easier and quicker. Spring Batch is also very flexible and extensible to fit all the needs a batch application can have. This chapter introduced the main features of Spring Batch and covered even how to make your batch processes scale with it or how it

compares to grid computing solutions. The conclusion is that Spring Batch is very well suited for the vast majority of enterprise applications.

This quick tour of Spring Batch's features and the Hello World sample should have given you of good idea of what the framework is, but you must be wondering how to write a complete batch process with it. This is the goal of chapter 2, where we'll introduce a case study and see how to build a Spring Batch application from it, before covering all the introduced features in depth in the following chapters.

2

Getting started with Spring Batch

This chapter covers:

- how to address a real enterprise scenario with a batch process
- how to efficiently read and write data with Spring Batch
- how to implement specific processing inside a Spring Batch job
- how to validate input parameters and skip incorrect records

Chapter 1 gave an overview of batch applications and of Spring Batch's features. It also introduced the framework with a very simple Hello World application. It's time to get down to business and see in practice how Spring Batch can address real enterprise scenarios. In this chapter, we're going to introduce you to a real use case that we'll use throughout the book to illustrate the use of Spring Batch. The use case will start out small and simple in this chapter, but it remains realistic in terms of technical requirements. It won't only show some Spring Batch features but also will also illustrate how this batch scenario fits into the enterprise.

By addressing this use case with Spring Batch, you'll get a very practical understanding of the framework: how it implements efficient reading and writing of large volumes of data, when to use off-the-shelf components and when to implement your own, how to configure the batch with the Spring lightweight container and much more! By the end of this chapter, you'll have a good overview of how Spring Batch works and you'll know exactly where to go in the book to find what you need for your batch applications. Let's get going and introduce our use case application.

2.1 Introducing the invoice front-end application

ACME company wants to share some data from its billing system with its customers. The billing system is unfortunately not the right entry point for customers, so ACME chose to build a dedicated web, read-only front-end application to expose the data. ACME decided to use batch processes to feed the front-end application's database with the data from the billing system, as shown in figure 2.1. Data will be sent every night to insert new invoice records or to update existing ones.

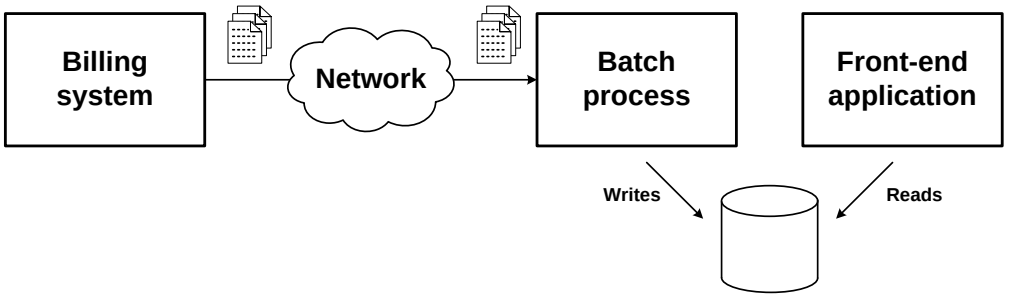

Figure 2.1 The front-end web application exposes billing data to customers. These data are retrieved from the company's billing system and sent to a batch process that writes them into the front-end application's database.

That's it for the big picture, but we need to really understand why ACME decided to build the front-end application and to feed it using batch processes.

2.1.1 Why the invoice front-end application?

First, why did ACME choose web applications to expose part of its billing data? Mainly because web applications are easy to deploy, easy to access, and provide a good user experience. ACME plans to deploy the front-end application to a local web hosting provider, rather than hosting it in its own network. Customers will only need a web browser to access the application and access will be restricted thanks to a username and a password. The first version of the front-end application will provide a simple but efficient user interface; ACME will focus on the data first, before providing more features and a more elaborate user interface.

Second, why didn't ACME choose to make the billing system and the front-end application communicate directly instead of importing data from one system to the other? Indeed, the billing system provides an API, so why not use it? The main reason is security: as shown in figure 2.2, the billing system is hosted in ACME's own network, and the company does not want to directly expose the billing system to the outside world, even through another application. This is drastic, but that's how things work at ACME.

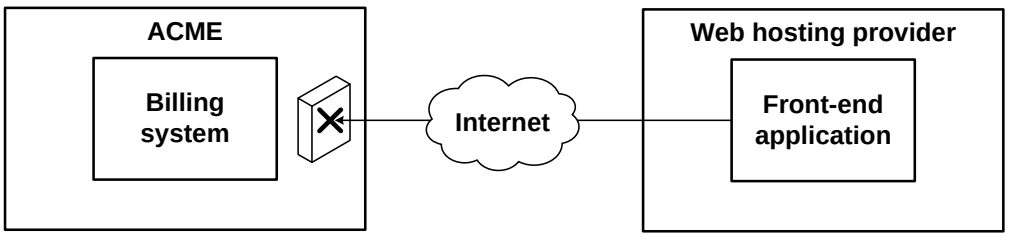

Figure 2.2 Instead of exchanging data, the two applications could communicate directly, but the company doesn't want its billing system to be directly accessible from the outside world.

Another reason to build the front-end application is that the billing system's API does not exactly suit the needs of the front-end application: ACME wants to show a digested version of the data to the

customers, so they're not bothered by some complex billing details. We could get to this digested version by using the billing system's API, but it would imply a lot of live calls, which could cause a performance overhead on the billing system. To summarize, there is a mismatch between the view of the billing system and the view of the front-end application: the data needs to be processed before being exposed.

2.1.2 Why use batch processes?

The front-end application scenario is a good example of two systems communicating to exchange some data. The billing system is updated all through the day, with new invoices inserted or existing ones updated. The front-end application does not need to expose live data: customers can live with one-day-old data. So a nightly batch process will feed the front-end application data store, using flat files, as shown in figure 2.3.

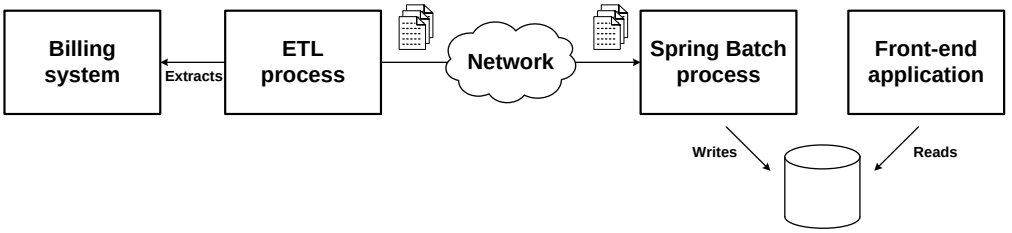

Figure 2.3 An ETL process extracts and transforms the billing system data into a flat file, which is sent every night to the Spring Batch process (the focus of this chapter). The Spring Batch process is in charge of reading the flat file and importing the data into the front-end application's database.

Figure 2.3 shows that an ETL (Extract, Transform, Load) process handles the creation of the flat files that will feed the front-end application. This ETL process extracts the data from the billing system and transforms them to produce the view that the front-end application's model expects. This ETL process is a black box in our current discussion: it could be implemented with an ETL tool (like Talend for example) or even with another Spring Batch job. What we are going to focus on is the way the front-end application reads and writes the billing data, which in our case are invoices.

2.1.3 The use case, importing invoices

The front-end application is about displaying customers' billing data, so the main concept is an invoice. The batch process will read the invoice records from the flat file coming from ACME and update the front-end application's database accordingly. Figure 2.4 shows that reading and writing invoices is at the core of the batch process, but also that it's not only made of this step.

Figure 2.4 The Spring Batch process is made of a validation stage, a decompressing step and a read/write step.

This read/write step is the core of the batch process but figure 2.4 shows that this is not only what the batch process is made of. Indeed, it will be constituted of:

- a validation stage, at the very beginning of the batch, to check for example that the input file is there
- a decompressing step, as the flat file comes from ACME's network as a compressed archive, to speed up the transfer
- the read/write step, where each line of the flat file is read and then inserted into the database

The different parts of the batch process allows us to introduce many Spring Batch's features, as shown in table 2.x.

Table 2.x Spring Batch's features introduced by the different parts of the invoice process

Part of the batch process	Spring Batch's features introduced
Validation stage	Validating the job parameters, to stop the job immediately if some of the inputs are incorrect.
Decompressing step	Implementing specific processing (not read/write) inside a job.
Read/write step	Using Spring Batch's flat file support to read data, implementing a custom database writing component, skipping incorrect records instead of failing the whole process.
Configuration	Leveraging Spring's lightweight container and Spring Batch's namespace to wire the batch components, using Spring Expression Language to make the configuration more flexible.

To introduce you to all these features, we won't follow the flow of the batch process, because it wouldn't provide the best learning experience. Imagine for example starting by the validation stage whereas you don't even know yet about the input parameters of the job! We'll then start by the core of the process, reading and writing the invoices, and then we'll see how to decompress the incoming compressed file before making the process more robust by validating the input parameters and choosing to skip incorrect records to avoid failing the whole job.

But before diving into the writing of the job, let's think about the up-coming versions of the front-end application.

2.1.4 How the application will evolve

ACME adopts an iterative process for the development of its front-end application. It's not that ACME doesn't think big, but it wants to stay cautious, to avoid creating fancy and useless features. First, we should be able to polish up some technical parts in the up-coming versions. For example, we'll choose to launch the batch process as a standalone Java process, from the command. It implies using a system scheduler. It works but this is not ideal, so a nice evolution would be to embed a scheduler in the web application itself to launch the batch process. Chapter 5 covers how to use a Java scheduler.

Secondly, some business requirements could impact the batch process, or rather on the way it is triggered. If for example the customers want more live data, instead of sending a very large file each night, we could think of sending smaller files but more often. The scheduler could then fire up the batch

process every hour or we could even scan a directory to launch the batch process once a file has been copied in it. It would give an event-driven taste to our front-end application and this is covered in chapter 11, where we deal with enterprise integration.

This is about the future, let's stick to the present by covering how to read and write the invoices with Spring Batch.

2.2 Reading and writing the invoices

Reading and writing the invoices is the core of our Spring Batch process. Remember that the billing data comes as a flat file and needs to be inserted in the database of the front-end application. Reading and writing scenarios are Spring Batch's sweet spot: for the invoice import process you'll only have to configure a Spring Batch's component to read the content of the flat file, implement a simple interface for the writing component, write some configuration and Spring Batch will handle the execution flow. Let's start by seeing how Spring Batch drives a read and write scenario.

2.2.1 Anatomy of read-write step

As read and write (or copy) scenarios are fairly common in batch applications, Spring Batch provides specific support for them. This support comes with a lot of ready-to-use components to read and write from and to common data stores like files or databases, but it is also made of a specific batch-oriented way to handle the execution flow: chunk processing. Figure 2.x shows the principle of chunk processing.

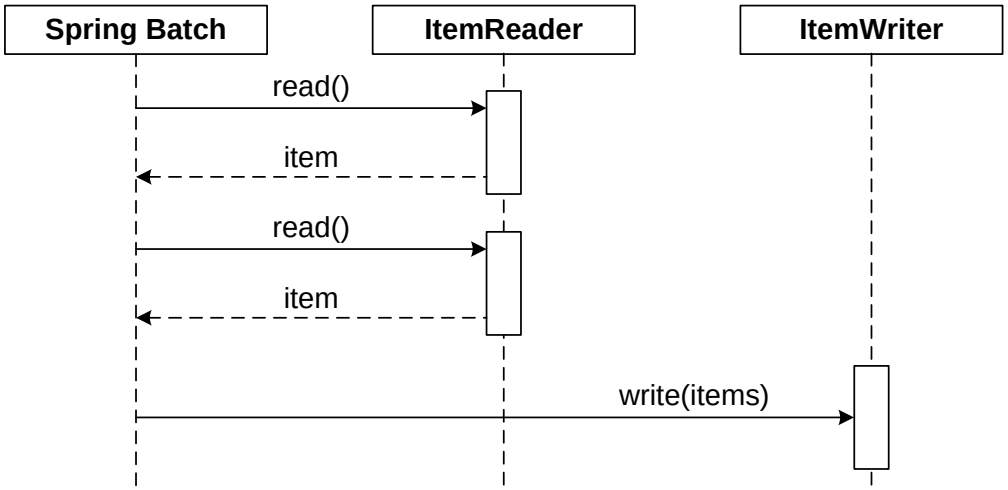

Figure 2.x For read and write scenarios, Spring Batch uses chunk processing. It reads items one by one from an item reader, gathers the items in a chunk of a given size and sends the chunk to the item writer

When performing chunk processing Spring Batch asks items one by one to an `ItemReader`, accumulates them and then sends them (as a chunk) to an `ItemWriter`.

CHUNK PROCESSING

Chunk processing is particularly adapted to large copy operations as the items are handled by small chunks instead of being handled as a big whole. Practically, it means that a large file won't be loaded in memory but rather streamed: this is more efficient in terms of memory consumption. Chunk processing allows more flexibility to handle the flow of the copy: Spring Batch can handle concerns like transactions or errors around the read and write operations.

Speaking about flexibility, Spring Batch adds a processing step in its chunk processing: items read can be processed (transformed) before being sent to the `ItemWriter`. Being able to process an item is useful when you don't want to write the exact same item you've just read. The component that handles the transformation is called an `ItemProcessor`. Item processing is optional in Spring Batch so it doesn't mean the chunk processing shown in figure 2.x is incorrect. Figure 2.x illustrates chunk processing, but this time with an item processing stage.

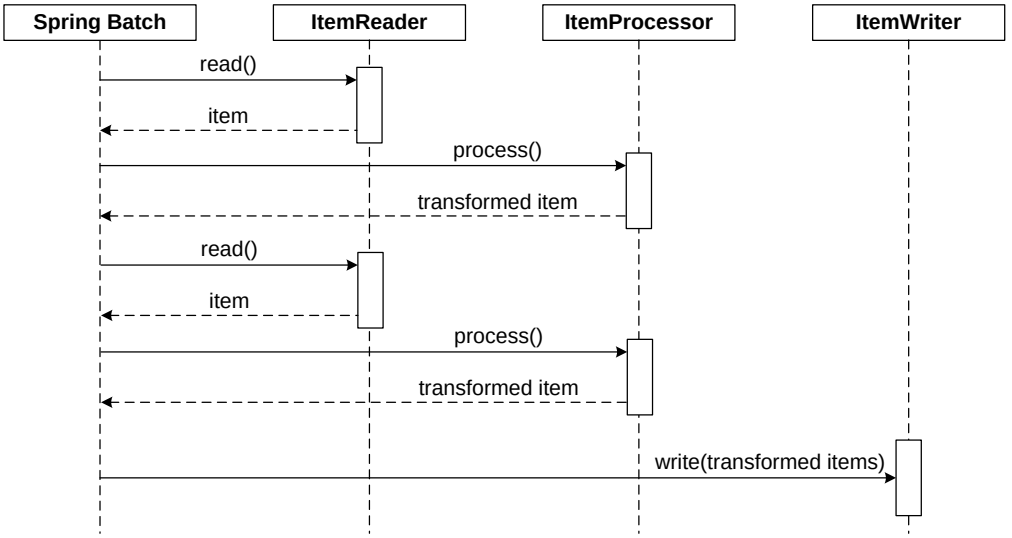

Figure 2.x Chunk processing with the optional processing stage, where an item processor can transform the read item before it is sent to the item writer

What can be done in an `ItemProcessor`? Any needed transformation on the item read before it is sent to the `ItemWriter`. This is where you can implement the logic to transform the data view from the input system into the data view expected by the target system. Spring Batch lets you also validate and filter a read item: if you choose to return nothing from the `ItemProcessor`, there won't be anything written.

NOTE

Our read/write use case doesn't have an item processing stage.

Listing 2.1 shows the definition of the `ItemReader`, `ItemProcessor` and `ItemWriter` interfaces.

Listing 2.1 The Spring Batch's interfaces for chunk processing

```
package org.springframework.batch.item;                          #A
                                                                 #A
public interface ItemReader<T> {                                 #A
                                                                 #A
  T read() throws Exception, UnexpectedInputException,           #A
                  ParseException;                                #A
                                                                 #A
}                                                                #A

package org.springframework.batch.item;                          #B
                                                                 #B
public interface ItemProcessor<T, O> {                           #B
                                                                 #B
  O process(T item) throws Exception;                            #B
                                                                 #B
}                                                                #B

package org.springframework.batch.item;                          #C
                                                                 #C
import java.util.List;                                           #C
                                                                 #C
public interface ItemWriter<O> {                                 #C
                                                                 #C
  void write(List<? extends O> items) throws Exception;          #C
                                                                 #C
}                                                                #C
```

#A Reads item
#B Transforms item (optional)
#C Writes chunk of items

Chapters 6 and 7 cover all the implementations of `ItemReader` and `ItemWriter` that Spring Batch provides, respectively. Chapter 8 covers the processing phase, to transform items but also to filter them. Note that each interface uses Java generics, so it means that the chain made of a reader, a processor and a writer should be type-compatible, as shown in figure 2.x.

Figure 2.x As the item interfaces (reader, processor, and writer) are parameterized, they need to be compatible when they are assembled into a chain for chunk processing

In the next two sub-sections, we'll see how to configure Spring Batch's flat `ItemReader` and how to write our own `ItemWriter` to handle the writing of our invoices in the database.

2.2.2 Reading from a flat file

Spring Batch provides the `FlatFileItemReader` class to read records from a flat file. By using it, you'll end up configuring some Spring beans and implement a component that creates domain objects

from what the FlatFileItemReader reads, Spring Batch will handle the rest. You can kiss goodbye all the I/O boilerplate code and focus on the structure of the data and how to exploit it.

THE FORMAT OF THE FLAT FILE

The input flat file is made of a header line and then one line for each invoice record. Here is an excerpt:

```
INVOICE_ID,CUSTOMER_ID,DESCRIPTION,ISSUE_DATE,AMOUNT
PR....210,9834,Paid,2010-09-15,124.60
PR....211,9834,Last call,2010-10-25,139.45
PR....212,6765,Paid,2010-08-26,97.80
PR....213,4525,Paid,2010-12-19,166.20
PR....214,9737,Paid,2010-11-18,145.50
```

Nothing fancy in this flat file: in a row, columns are separated by a comma. Each row of the file will be mapped to an Invoice domain object.

THE INVOICE DOMAIN CLASS

The Invoice class just maps the different columns of the flat file (we eluded the getter and setter methods):

```
package com.manning.sbia.ch02.domain;

import java.math.BigDecimal;
import java.util.Date;

public class Invoice {

  private String id;
  private Long customerId;
  private String description;
  private Date issueDate;
  private BigDecimal amount;

  (...)

}
```

Let's see now how to use the FlatFileItemReader to create Invoice objects from the flat file.

SPRING BATCH'S FLATFILEITEMREADER

The FlatFileItemReader will handle the I/O for us (opening the file, streaming it to read each line) and will delegate the mapping between a line and a domain object to a LineMapper. Spring Batch provides us with a handy LineMapper implementation, DefaultLineMapper, which itself delegates the mapping to other strategy interfaces. Figure 2.x shows all this delegation work.

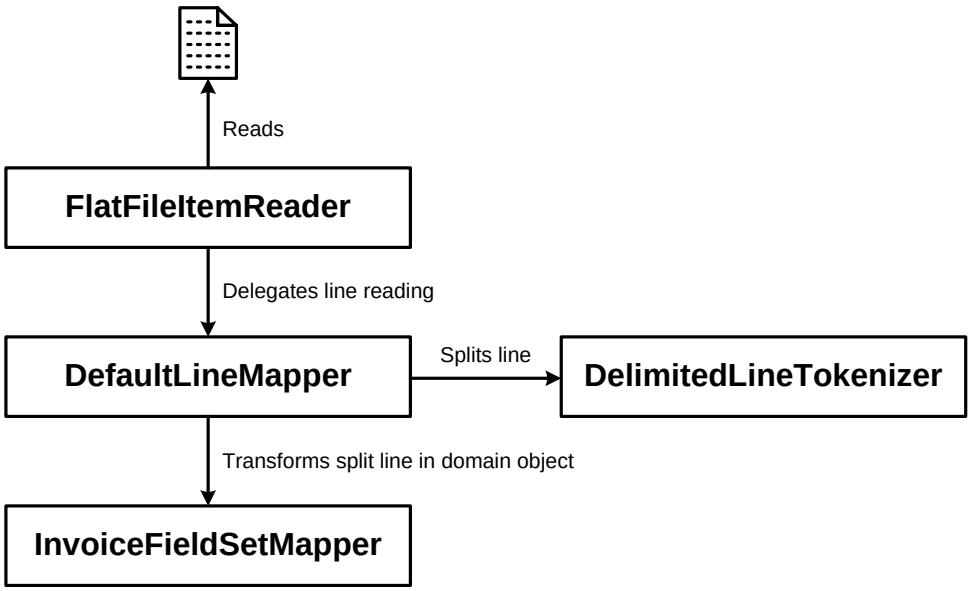

Figure 2.x The FlatFileItemReader reads the file but delegates the mapping between a line and a domain object to a LineMapper. The LineMapper implementation we use delegates the splitting of lines and the mapping between split lines and domain objects.

That's a lot of delegation and it means you'll have more to configure. This is the price of re-usability and flexibility: you'll be able to configure and use implementations that Spring Batch provides out-of-the-box or provides your own implementations for something specific.

The `DefaultLineMapper` is a typical example as it needs:

- a `LineTokenizer` to split the line into fields – you'll use a Spring Batch implementation for this
- a `FieldSetMapper` to transform the split line into a domain object – you'll write your own implementation for this

We'll see the whole Spring configuration soon (especially for the `LineTokenizer`) – in listing 2.2 – but we are going to focus now on our `FieldSetMapper` implementation.

THE CUSTOM INVOICE FIELDSETMAPPER

The `FieldSetMapper` is about converting the line split by the `LineTokenizer` into a domain object. The `FieldSetMapper` interface is straightforward:

```
public interface FieldSetMapper<T> {
  T mapFieldSet(FieldSet fieldSet) throws BindException;
}
```

The `FieldSet` parameter comes from the `LineTokenizer` and you can see it as a flat file equivalent to the JDBC `ResultSet`: it will help you retrieve field values and do some conversion between `String` and richer object like `Date`. The following snippet shows our `InvoiceFieldSetMapper` implementation:

```
package com.manning.sbia.ch02.batch;
```

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

import com.manning.sbia.ch02.domain.Invoice;

public class InvoiceFieldSetMapper implements FieldSetMapper<Invoice> {

  public Invoice mapFieldSet(FieldSet fieldSet) throws BindException {
    Invoice invoice = new Invoice();
    invoice.setId(fieldSet.readString("INVOICE_ID"));
    invoice.setCustomerId(fieldSet.readLong("CUSTOMER_ID"));
    invoice.setDescription(fieldSet.readString("DESCRIPTION"));
    invoice.setIssueDate(fieldSet.readDate("ISSUE_DATE"));
    invoice.setAmount(fieldSet.readBigDecimal("AMOUNT"));
    return invoice;
  }
}
```

Our `InvoiceFieldSetMapper` implementation is not rocket science and that's exactly the point: we focus on retrieving the data from the flat file to handle them as a domain object to the reading component. Spring Batch deals with the plumbing: reading efficiently the flat file. Perhaps you noticed in the `mapFieldSet` method some references to field like INVOICE_ID or CUSTOMER_ID. Where do these references come from? They're part of the `LineTokenizer` configuration, so let's study now the Spring configuration of our `FlatFileItemReader`.

SPRING CONFIGURATION OF THE FLATFILEITEMREADER

The `FlatFileItemReader` can be configured like any Spring bean, using XML configuration, as shown in listing 2.2:

Listing 2.2 The Spring configuration of the FlatFileItemReader

```
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="resource"                                          #1
            value="file:invoices.txt" />                             #1
  <property name="linesToSkip" value="1" />                          #2
  <property name="lineMapper">
    <bean
     class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">
        <bean class="org.springframework.batch.item.file.transform.
[CA] DelimitedLineTokenizer">
          <property name="names" value="INVOICE_ID                 #3
[CA] ,CUSTOMER_ID,DESCRIPTION,ISSUE_DATE,AMOUNT" />                  #3
        </bean>
      </property>
      <property name="fieldSetMapper">                               #4
        <bean class="com.manning.sbia.ch02.batch.                    #4
[CA] InvoiceFieldSetMapper" />                                       #4
      </property>                                                    #4
    </bean>
  </property>
</bean>
```

#1 Sets input file
#2 Skips first line
#3 Sets columns names

#4 Uses invoice field set mapper

The input file can be set with the `resource` property (#1). As the first line of the flat file contains headers, we can skip it thanks to the `linesToSkep` property (#2). We use a `DelimitedLineTokenizer` to split each line into fields (it uses a comma as the default separator). At #3, we set the name of each field. These are the names we used in the `InvoiceFieldSetMapper` to retrieve values from the `FieldSet`. We then inject an instance of our `InvoiceFieldSetMapper` at #4.

That's it, your flat file reader is ready! Don't feel overwhelmed: the flat file support in Spring Batch implies many components, but that's what makes it powerful and flexible. You are going to do less configuration and more Java code to implement now the database item writer.

2.2.3 Implementing a database item writer

Updating the database with our invoices data is a little special, that's why we'll have to implement our own `ItemWriter`. Why is it so special? Each line of the flat file represents either a new invoice record or an existing one, so we'll have to decide if we should send an `insert` or an `update` SQL statement. Nevertheless, the implementation of the `InvoiceJdbcItemWriter` is quite straightforward, as shown in listing 2.3.

Listing 2.3 The Java implementation of the InvoiceJdbcItemWriter

```
package com.manning.sbia.ch02.batch;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.batch.item.ItemWriter;
import org.springframework.jdbc.core.simple.SimpleJdbcTemplate;
import com.manning.sbia.ch02.domain.Invoice;

public class InvoiceJdbcItemWriter implements ItemWriter<Invoice> {

  private static final String INSERT_INVOICE = "insert into invoice "+
   "(id,customer_id,description,issue_date,amount) values(?,?,?,?,?)";

  private static final String UPDATE_INVOICE = "update invoice set "+
    "customer_id=?, description=?, issue_date=?, amount=? where id = ?";

  private SimpleJdbcTemplate jdbcTemplate;                          #1
                                                                    #1
  public InvoiceJdbcItemWriter(DataSource ds) {                     #1
    this.jdbcTemplate = new SimpleJdbcTemplate(ds);                 #1
  }                                                                 #1

  public void write(List<? extends Invoice> items) throws Exception {
    for(Invoice item : items) {
      int updated = jdbcTemplate.update(                            #2
        UPDATE_INVOICE,                                             #2
        item.getCustomerId(),item.getDescription(),                 #2
        item.getIssueDate(),item.getAmount(),item.getId()           #2
      );                                                            #2
      if(updated == 0) {
        jdbcTemplate.update(                                        #3
          INSERT_INVOICE,                                           #3
```

```
            item.getId(),item.getCustomerId(),                          #3
            item.getDescription(),item.getIssueDate(),                  #3
            item.getAmount()                                            #3
          );                                                            #3
        }
      }
    }
  }
```

#1 Uses JDBC template for data access
#2 Tries to update invoice
#3 Inserts new invoice

The `InvoiceJdbcItemWriter` uses Spring's `JdbcTemplate` to interact with the database. The `JdbcTemplate` is created with the `DataSource` injected to the constructor (#1). In the `write` method, we iterate over the chunk's invoices and try to update an existing record (#2). If the database returns that the `update` statement didn't update any record, we are sure that this is a new record and we can insert it (#3).

That's it for the implementation! Notice how implementing an `ItemWriter` can be simple, mainly because Spring Batch handles all the work around the writing (asking records to the `ItemReader`, creating chunks, opening transactions and so on).

The item writer can then be configured as a Spring bean, it just needs a `DataSource` dependency:

```
<bean id="writer"
      class="com.manning.sbia.ch02.batch.InvoiceJdbcItemWriter">
  <constructor-arg ref="dataSource" />
</bean>
```

Now we have the two parts of our read-write step, we can see how to assemble them in a Spring Batch job.

2.2.4 Configuring the read-write step

The configuration of the read-write step is part of a Spring configuration. It can lie besides the declaration of our reader and writer beans. Listing 2.4 shows this configuration.

Listing 2.4 The Spring configuration of the read-write step

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:batch="http://www.springframework.org/schema/batch"
    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-2.1.xsd">

  <job id="importInvoices"                                              #1
       xmlns="http://www.springframework.org/schema/batch">             #1
    <step id="readWriteInvoices">
      <tasklet>
        <chunk reader="reader" writer="writer"                          #2
               commit-interval="100" />                                 #3
      </tasklet>
    </step>
  </job>

  <bean id="reader" (...)
  </bean>
```

```
    <bean id="writer" (...)
    </bean>

</beans>
```

#1 Starts job configuration
#2 Configures chunk step with reader and writer
#3 Sets commit interval

The configuration file starts with the declaration of the namespaces we want to use: the usual Spring's `beans` namespace and Spring Batch's `batch` namespace. The `beans` namespace is declared as the top-level namespace, so we don't need to use a prefix to use its elements. Unfortunately this is not convenient for the overall configuration, as we'll have to use the `batch` prefix for the batch job parts. But we have a workaround for this: when we start the job configuration at #1, we can specify the namespace as an attribute of the `job` element we're using for it and all its siblings. This makes the configuration more readable. Our chunk-oriented step is configured with a `chunk` element, inside `step` and `tasklet` elements. At #2, we refer to the reader and writer beans thanks to the `reader` and `writer` attributes, respectively. We use also at #3 the `commit-interval` attribute to specify the size of chunks.

We're done with the copy part of our batch process. Spring Batch helped us a lot to read the invoices from the flat file and import them in the database: we didn't write any code Java for the reading and wrote only the logic to insert or update invoices in the database. Putting the parts together ended up being straightforward thanks to Spring's lightweight container and Spring Batch's XML namespace.

So we've just implemented the "Reading and writing" box shown in figure 2.4: a copy scenario is a common requirement in batch applications and Spring Batch provides lots of help for this, but the framework is not restricted to this as it allows to writing any kind of operation as part of batch process, as we'll see this by decompressing the input file of our job, which is the "Decompressing" box in figure 2.4.

2.3 Decompressing the input file with a tasklet

Remember that our flat file is uploaded on the front-end application system as a compressed archive. So we need to decompress this file before starting the reading and the writing of the invoices. Decompressing a file is not a read-write step, but Spring Batch is flexible enough to implement such a task as a part of one of its job. But before showing you how to decompress the input file, let's explain why we chose to compress the invoice data.

2.3.1 Why a compressed file?

The flat file containing the invoice data is compressed because this allows us to upload it faster from ACME's network to the web hosting provider that hosts the front-end application. Textual data like flat files can be compressed with effective compression ratios (1 to 10 for example). So a 1 GB flat file can be compressed to a 100 MB archive file, which is a more reasonable size for file transfer.

Note that the file could have been encrypted too; to ensure that nobody could read the invoice data if they were intercepted during the transfer. The encryption could have been done before the compression or be part of it. In our case, we assume that ACME and the hosting provider agreed on a secured transfer protocol, like Secure Copy (SCP), which builds on top of SSH.

Now the choice of a compressed is justified, let's see how to implement the decompressing tasklet.

2.3.2 Implementing the tasklet

Spring Batch provides an extension point to handle the processing in a step of a batch process: the `Tasklet`. You discovered the `Tasklet` interface in chapter 1, when you wrote the Hello World application with Spring Batch. Here we're going to implement a `Tasklet` that decompresses a ZIP archive into a large flat file. Listing 2.5 shows the implementation of the `DecompressTasklet`.

Listing 2.5 The implementation of the DecompressTasklet

```
package com.manning.sbia.ch02.batch;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.util.zip.ZipInputStream;
import org.apache.commons.io.FileUtils;
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;
import org.springframework.core.io.Resource;

public class DecompressTasklet implements Tasklet {                    #1

  private Resource inputResource;                                       #2
  private String targetDirectory;                                       #2
  private String targetFile;                                            #2

  private final int BUFFER = 2048;

  public RepeatStatus execute(StepContribution contribution,
       ChunkContext chunkContext) throws Exception {
    ZipInputStream zis = new ZipInputStream(                            #3
      new BufferedInputStream(                                          #3
        inputResource.getInputStream()));                               #3

    File targetDirectoryAsFile = new File(                              #4
      targetDirectory);                                                 #4
    if(!targetDirectoryAsFile.exists()) {                               #4
      FileUtils.forceMkdir(targetDirectoryAsFile);                      #4
    }                                                                   #4
    BufferedOutputStream dest = null;
    while(zis.getNextEntry() != null) {                                 #5
      int count;                                                        #5
      byte data[] = new byte[BUFFER];                                   #5
      File target = new File(targetDirectory,                           #5
        targetFile);                                                    #5
      if(!target.exists()) {                                            #5
        target.createNewFile();                                         #5
      }                                                                 #5
      FileOutputStream fos = new FileOutputStream(                      #5
        target);                                                        #5
      dest = new BufferedOutputStream(fos, BUFFER);                     #5
      while ((count = zis.read(data, 0, BUFFER))                        #5
          != -1) {                                                      #5
        dest.write(data, 0, count);                                     #5
      }                                                                 #5
```

```
        dest.flush();                                                    #5
        dest.close();                                                    #5
      }                                                                  #5
      zis.close();
      if(!target.exists()) {
        throw new IllegalStateException(
          "Could not decompress anything from the archive!");
      }
      return RepeatStatus.FINISHED;                                      #6
    }
    /* setters */
    (...)

  }
```

#1 Implements Tasklet
#2 Declares Tasklet parameters
#3 Opens archive
#4 Creates target directory if necessary
#5 Decompresses archive
#6 Declares Tasklet as finished

The `DecompressTasklet` class implements the `Tasklet` interface (#1), which has only one method, `execute` (you'll learn more about the method parameters in chapters 4 and 10, we don't use them right now). The tasklet has three fields (#2), which represent the archive file, the name of the directory where to decompress the file and the name of the output file. These fields will be set when we'll configure the tasklet with Spring. In the execute method, we open a stream to the archive file (#3), create the target directory if necessary (#4) and use the Java API to decompress the ZIP archive (#5). Note we use the `FileUtils` class from the Commons Apache IO project to create the target directory, as it provides handy utilities to deal with files and directory. At #6, we return the `FINISHED` value from the `RepeatStatus` enumeration to say to Spring Batch that the tasklet is over.

The `Tasklet` interface is straightforward to implement, but in our case it implies some boilerplate code as we deal with decompressing, which is verbose in Java. Let's see now how to configure the tasklet with Spring.

2.3.3 Configuring the tasklet

The tasklet configuration is part of the Spring configuration of our job and it's made of two steps: declaring the tasklet as a Spring bean and injecting it as a step of the job. So we need to modify the configuration we wrote for the reading and writing of the invoices, as shown in listing 2.6.

Listing 2.6 The Spring configuration of the job with the decompress tasklet

```
<job id="importInvoices"
     xmlns="http://www.springframework.org/schema/batch">
  <step id="decompress" next="readWriteInvoices">                        #1
    <tasklet ref="decompressTasklet" />                                  #1
  </step>                                                                #1
  <step id="readWriteInvoices">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100" />
    </tasklet>
```

```
    </step>
  </job>

  <bean id="decompressTasklet"                                    #2
        class="com.manning.sbia.ch02.batch.                       #2
  [CA]    DecompressTasklet">                                     #2
    <property name="inputResource"                                #2
              value="file:./input/input.zip" />                   #2
    <property name="targetDirectory"                              #2
              value="./work/output/" />                           #2
    <property name="targetFile"                                   #2
              value="invoices.txt" />                             #2
  </bean>                                                         #2
```

#1 Sets tasklet in the job
#2 Declares tasklet bean

The configuration of a plain `Tasklet` is simpler than for read/write step, as we only refer to the `Tasklet` bean (#1). Note we control the job flow thanks to `next` attribute of the `step` element, which refers to the `readWriteInvoices` step. Chapter 10 covers thoroughly how you can control the flow of your Spring Batch jobs and take different paths depending on how a step ended for example. The `tasklet` element at #1 refers to the `decompressTasklet` bean, declared at #2. Perhaps you find the settings of the `Tasklet` bean rigid because they're hard coded in the Spring configuration file, but don't worry, we'll show you later in the chapter how to make them more dynamic.

We now have the parts of the job implemented and configured: we can decompress the input archive and read the invoices from the decompressed flat file and write them into the database. What we need now is the infrastructure to drive the processing of our job and we'll see in the next section how to configure this infrastructure.

2.4 Configuring the batch process

As stated in chapter 1 the Spring Batch framework will help you with your batch applications by driving the flow of your batch processes. To do so, it needs some infrastructure components that we'll configure in the Spring lightweight container. These infrastructure components will act as a lightweight runtime environment to run our batch process.

Setting up the batch infrastructure is a mandatory step for a batch application but you need to do it only once for all the Spring Batch jobs that live in the same Spring application context. The jobs will use the same infrastructure components to run but also to store their state, so these components are key components for managing and monitoring the jobs (chapter 12 covers how to monitor your Spring Batch jobs). Let's see immediately how to configure the minimal infrastructure for Spring Batch.

2.4.1 Setting up the batch infrastructure

Chapter 1 already introduced the concepts of job repository and job launcher. The former is about storing the state of the jobs (finished or currently running) and the latter is about creating the state of a job before launching its execution. Spring Batch provides several implementations for both but we'll stick to the simplest ones in this chapter (chapter 3 covers how to quickly set up a job repository that uses a database). Listing 2.7 shows how to configure the infrastructure.

Listing 2.7 The Spring configuration for the batch infrastructure

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...) >

  <bean id="dataSource"                                                    #A
        class="org.springframework.jdbc.datasource.                        #A
[CA]  SingleConnectionDataSource">                                         #A
    <property name="driverClassName"                                       #A
              value="org.h2.Driver" />                                     #A
    <property name="url"                                                   #A
    value="jdbc:h2:mem:invoices;DB_CLOSE_DELAY=-1 "/>                      #A
    <property name="username" value="sa" />                                #A
    <property name="password" value="" />                                  #A
  </bean>                                                                  #A

  <bean id="transactionManager"                                            #B
        class="org.springframework.jdbc.datasource.                        #B
[CA]  DataSourceTransactionManager">                                       #B
    <property name="dataSource" ref="dataSource" />                        #B
  </bean>                                                                  #B

  <bean id="jobRepository"                                                 #C
        class="org.springframework.batch.core.                             #C
[CA]  repository.support.MapJobRepositoryFactoryBean">                     #C
    <property name="transactionManager"                                    #C
              ref="transactionManager" />                                  #C
  </bean>                                                                  #C

  <bean id="jobLauncher"                                                   #D
        class="org.springframework.batch.core.launch.                      #D
[CA]  support.SimpleJobLauncher">                                          #D
    <property name="jobRepository"                                         #D
              ref="jobRepository" />                                       #D
  </bean>                                                                  #D

</beans>
```

#A Declares data source
#B Declares transaction manager
#C Declares job repository
#D Declares job launcher

We're not going to focus on the job repository and the job launcher as we already covered them in chapter 1 (and you'll learn more about their configuration in chapter 4). The previous snippet declared a data source, which connects to the front-end application's database. Perhaps you noticed we used an in-memory database (H2), which looks weird for an online application. You could change the data source configuration to use a database like PostgreSQL or Oracle but the previous snippet comes from the code samples of the book, which use an in-memory database, as it is easier to deploy (you won't have to install any database engine to work with the code samples).

How does the job refer to the job repository?

Careful readers may have noticed that we say the job needs the job repository to run but we didn't make any reference to the job repository bean in the job configuration. The XML `step` element can

have `job-repository` attribute to refer to a job repository bean but this attribute is not mandatory because the job uses by default a `jobRepository` bean. This implies that as long as you declare a `jobRepository` bean of type `JobRepository`, you don't need to explicitly refer to it in the configuration of your jobs.

This explanation about the data source leads to a very important practice: when configuring a Spring Batch application, the infrastructure configuration should be separated from the jobs' configuration.

SPLITTING INFRASTRUCTURE AND APPLICATION CONFIGURATION FILES

You should always split infrastructure and application Spring configuration files. This allows to swap out the infrastructure between environments (test, development, staging, production) while reusing the application – jobs' in our case – configuration files.

The batch infrastructure configuration is in its own file and the job configuration only depends on the job repository and data source beans, so as long a Spring configuration provides them when we load the Spring application context, we're fine.

We're done with the infrastructure configuration and we even did it in a flexible way as we separated it from the job configuration. We can go further in making the configuration more flexible by leveraging the Spring Expression Language to avoid hard coding some settings in the Spring configuration files.

2.4.2 Leveraging the Spring Expression Language for configuration

Remember that part of our job configuration is hard coded in the Spring configuration files, like for example all the settings related the location of files:

```
<bean id="decompressTasklet"
      class="com.manning.sbia.ch02.batch.DecompressTasklet">
  <property name="inputResource" value="file:./input/input.zip" />
  <property name="targetDirectory" value="./work/output/" />
  <property name="targetFile" value="invoices.txt" />
</bean>
```

This is not flexible, as these settings can change between environments (testing and production for example) or because we could use rolling files for the incoming archive (meaning its name would depend on the date). So a very nice improvement would be to make these settings more flexible, especially when we know that we can pass parameters when we launch. Remember from chapter 1 when we launched our Hello World job:

```
JobExecution jobExecution = jobLauncher.run(job, new JobParameters());
```

We didn't pass any parameter because we used an empty instance of `JobParameters`, but we could have used a `JobParametersBuilder` to launch the job with some parameters:

```
jobLauncher.run(job, new JobParametersBuilder()
  .addString("parameter1", "value1")
  .addString("parameter2", "value2")
  .toJobParameters());
```

The good news is you can refer to these parameters in your job configuration and this could be very useful for your `DecompressTasklet` and `FlatFileItemReader` beans, as shown in the listing 2.8.

Listing 2.8 Referring to job parameters in the Spring configuration

```
<bean id="decompressTasklet"
      class="com.manning.sbia.ch02.batch.DecompressTasklet"
```

```
        scope="step">                                                    #1
    <property name="inputResource"                                       #2
      value="#{jobParameters['inputResource']}" />                       #2
    <property name="targetDirectory"                                     #2
      value="#{jobParameters['targetDirectory']}" />                     #2
    <property name="targetFile"                                          #2
      value="#{jobParameters['targetFile']}" />                          #2
  </bean>

  <bean id="reader"
        class="org.springframework.batch.item.file.FlatFileItemReader"
        scope="step">                                                    #1
    <property name="resource"                                            #3
      value="file:#{jobParameters['targetDirectory']                     #3
[CA]  +jobParameters['targetFile']}" />                                  #3
```

#1 Uses step scope
#2 Refers to job parameters
#3 Concatenates job parameters

To be able to refer to job parameters, a bean must use the Spring Batch's `step` scope (#1). It means that Spring will create the bean only when the step asks for it and the values will be resolved at this time (this is a lazy instantiation, as the bean is not created during the Spring application context's bootstrapping). To trigger the dynamic evaluation of a value, you must use the `#{expression}` syntax. The expression must be in the Spring Expression Language (SpEL), which is available as of Spring 3.0 (Spring Batch is able to fall back to a less powerful language if you don't have Spring 3.0 on your class path). There's a `jobParameters` variable available and you can use it as a `Map`. That's how we refer to the `inputResource`, `targetDirectory` and `targetFile` job parameters at #2. Note you're not limited to plain references, you can also use more complex expressions, like we do to concatenate the target directory and file for the file reader at #3.

We're done now with the configuration: we have our job and infrastructure ready, and part of the configuration can come from the job parameters, which are set when we launch the batch. So this is time to see how we can execute our batch process.

2.5 Launching the job

We're going to cover two ways to launch our job, both implies launching a simple Java program from the command line. The first way will consist in writing our own main program and the second way will consist in using a Java program that Spring Batch provides. By writing our own main program, we'll learn about the Spring Batch API and figure out easily what happens behind the scenes when we'll use the runner provided by Spring Batch.

Both ways are based on the command line, so it means the job launching is triggered either manually or by a system scheduler like CRON. Spring Batch is not limited to these scenarios and chapter 5 covers other ways to launch your batch jobs, like embedding Spring Batch in a web application and using a Java scheduler.

2.5.1 Using a main program

To launch our invoices importing job in a main program we need to validate the job parameters passed on the command line, to bootstrap the Spring application context and to launch the `Job` bean with the `JobLauncher` bean. Listing 2.9 shows this main program.

Listing 2.9 The Java program to launch the batch process

```
package com.manning.sbia.ch02.batch;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ImportInvoicesBatchProgram {

  public static void main(String[] args) throws Exception {
    if (args == null || args.length != 3) {                          #1
      throw new IllegalArgumentException(                            #1
        "Batch job needs 3 parameters: " +                           #1
        "input archive, target directory, target file"               #1
      );                                                             #1
    }                                                                #1

    ApplicationContext context =                                     #2
    new ClassPathXmlApplicationContext(                              #2
      "/import-invoices-job-context.xml",                            #2
      "/batch-infrastructure-context.xml"                            #2
    );                                                               #2

    JobLauncher jobLauncher = context.getBean(JobLauncher.class);
    Job job = context.getBean(Job.class);
    JobExecution jobExecution = jobLauncher.run(job,                 #3
      new JobParametersBuilder()                                     #3
        .addString("inputResource", args[0])                         #3
        .addString("targetDirectory", args[1])                       #3
        .addString("targetFile", args[2])                            #3
        .toJobParameters());                                         #3

    System.out.println("Job ended with status: "
      + jobExecution.getExitStatus());

  }
}
```

#1 Checks command line parameters
#2 Bootstraps Spring application context
#3 Launches job with parameters

The main program starts by validating the command line parameters at #1: we need these parameters to launch the job properly. We then bootstrap the Spring application context at #2 and launch the job with its parameters at #3, thanks to the job launcher bean.

NOTE

When launching the job from the command line, ensure that the target database is properly initialized. We configured an in-memory database in this chapter, but this database is meant to be used in automated integration tests, where we directly created the necessary tables for the code samples.

Assuming you gathered all the JAR files a `lib` directory, you can launch the job with the following command in Unix-based shell:

```
java -classpath "./lib/*"
[CA] com.manning.sbia.ch02.batch.ImportInvoicesBatchProgram
[CA] file:./invoices.zip ./temp/ invoices.txt
```

This solution works but implies writing a main program for each Spring Batch job, which is very tedious if you have a lot of jobs. That's why Spring Batch provides a generic command line job runner.

2.5.2 Using Spring Batch's command line runner

Spring Batch provides the `CommandLineJobRunner` main program that we can use for running all our jobs from the command line. It prevents us from creating any specific main program. The `CommandLineJobRunner` takes care of the following:

- bootstrapping a Spring application context
- parsing the command line to convert parameters into job parameters
- launching the specified job

The syntax to use the `CommandLineJobRunner` is the following (assuming you have all the need JAR files in a `lib` directory and you're using a Unix-based shell):

```
java -classpath "./lib/*"
[CA] org.springframework.batch.core.launch.support.CommandLineJobRunner
[CA] springApplicationContextFile jobName
[CA] param1=value1 param2=value2
```

The `CommandLineJobRunner` is handy but it can use only one file to bootstrap the Spring application context, so we need to gather our 2 two files in one. Fortunately this is simple thanks to the `import` XML element:

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

  <import resource="/import-invoices-job-context.xml" />
  <import resource="/batch-infrastructure-context.xml" />

</beans>
```

If we called `import-invoices-batch-context.xml` the file that groups our configuration and infrastructure configuration files, we can launch our import invoices job with the following command line:

```
java -classpath "./lib/*"
[CA] org.springframework.batch.core.launch.support.CommandLineJobRunner
[CA] import-invoices-batch-context.xml importInvoices
[CA] inputResource=file:./invoices.zip targetDirectory=./temp/
[CA] targetFile=invoices.txt
```

Note that the last 3 command line parameters will be converted automatically into job parameters. As we can refer to them in the Spring configuration, it makes the job very flexible: we could for example change the location of the input archive without modifying the job itself.

Our import invoices job is now operational: it decompresses a ZIP archive, reads the content of a flat file to write it into a database. It even offers some configuration options from the command line. But perhaps some questions came to your mind when reading the previous sections: what would happen if the archive file doesn't exist? What would happen if one line of the flat file is not correctly formatted? These are legitimate questions, directly related to the robustness of our job. The next section will introduce some features that Spring Batch provides out-of-the-box to deal with these concerns.

2.6 Making the job more robust

Chapter 1 listed the requirements for batch applications and among them was robustness. Our invoice import job is not that robust yet: for example it would crash abruptly if only one single line of the flat file was not correctly formatted. The good news is Spring Batch can help you very quickly to make the job more robust and does this by only changing the configuration or implementing simple interfaces.

We're not going to see all Spring Batch's features related to robustness here, as chapter 9 covers them thoroughly, but we're going to show how you can add some validation to check that everything is in place before executing the steps of your job and how to handle unexpected entries when you're reading data. By the end of the section, the invoice import job will be more robust and you'll have a better understanding of how Spring Batch can help improve robustness.

2.6.1 Validating the job parameters

Usually a batch process needs some inputs, like a file to read data from, and trying to launch the process without these inputs available would lead the process to inevitably crash at some moment. This is exactly the case of our `DecompressingTasklet`: it needs the archive file as an input. So before decompressing the archive, we need to check it's part of the job parameters and that it exists. To make it short, we need to validate our inputs. There are several ways to achieve such validation.

We could implement the validation directly in the concerned component – the `DecompressTarget` in our example. It would work but the core code of the tasklet – decompressing – would be tangled with validation code.

We could also validate the parameters before launching the job. We did that – even if the validation was a bit lousy – when we implemented our own main program to launch the job: we checked there were at least 3 command line parameters, but we could have done much more. This also would work, but we'd be stuck with our main program and we wouldn't be able to use the `CommandLineJobRunner` that Spring Batch provides.

Anyway, it's not a good idea to tie the job runner with the validation of parameters: the job should embed it own validation logic, so that it can be launched from the command line or from another Java process. Fortunately, Spring Batch provides a hook to validate job parameters before the first step of the batch is launched. The first thing to do is to implement the validation component (a `JobParametersValidator`) and the second is to add it in the job configuration.

IMPLEMENTING THE JOB PARAMETERS VALIDATOR

To validate the parameters of your job, you need to implement the `JobParametersValidator`. Listing 2.10 shows the implementation for the invoice import job.

Listing 2.10 Job parameters validator for the invoices import job

```
package com.manning.sbia.ch02.batch;

import java.util.ArrayList;
import java.util.Collection;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersInvalidException;
import org.springframework.batch.core.JobParametersValidator;
import org.springframework.context.ResourceLoaderAware;
import org.springframework.core.io.ResourceLoader;

public class ImportInvoicesJobParametersValidator implements
    JobParametersValidator,ResourceLoaderAware {

  private static final String PARAM_INPUT_RESOURCE = "inputResource";
  private static final String PARAM_TARGET_DIRECTORY = "targetDirectory";
  private static final String PARAM_TARGET_FILE = "targetFile";

  private ResourceLoader resourceLoader;

  public void validate(JobParameters parameters)
      throws JobParametersInvalidException {
    Collection<String> missing = new ArrayList<String>();
    checkParameter(PARAM_INPUT_RESOURCE,                              #1
      parameters, missing);                                           #1
    checkParameter(PARAM_TARGET_DIRECTORY,                            #1
      parameters, missing);                                           #1
    checkParameter(PARAM_TARGET_FILE,                                 #1
      parameters, missing);                                           #1
    if(!missing.isEmpty()) {                                          #1
      throw new JobParametersInvalidException(                        #1
        "Missing job parameter(s): "+missing);                        #1
    }                                                                 #1
    if(!resourceLoader.getResource(parameters.                        #2
        getString(PARAM_INPUT_RESOURCE)).exists()) {                  #2
      throw new JobParametersInvalidException(                        #2
        "The input file: "+                                           #2
        parameters.getString(PARAM_INPUT_RESOURCE)+                   #2
        " does not exist");                                           #2
    }                                                                 #2
  }

  private void checkParameter(String parameterKey,
      JobParameters parameters,Collection<String> missing) {
    if(!parameters.getParameters().containsKey(parameterKey)) {
      missing.add(parameterKey);
    }
  }

  public void setResourceLoader(                                      #3
      ResourceLoader resourceLoader) {                                #3
    this.resourceLoader = resourceLoader;                             #3
  }                                                                   #3
}
```

#1 Checks parameters availability
#2 Checks input archive exists
#3 Injects resource loader

A `JobParametersValidator` gets a reference to the `JobParameters` in its `validate` method, so it can access to them to do any kind of validation. At #1 we check that the parameters we need are at least part of the job parameters. Remember we need specific parameters because we refer to them in the job configuration thanks to the Spring Expression Language. At #2 we check that the input archive file exists. We use the Spring resource abstraction to do so, and we delegate the loading to the `ResourceLoader` that is automatically passed by Spring at #3 (this is why we implemented the `ResourceLoaderAware` interface). Note we launch a `JobParametersInvalidException` if the validation fails.

Once we implemented the validation component, we can add it to the job configuration.

CONFIGURING THE JOB PARAMETERS VALIDATOR

You can add the validator by using the `validator` element in the job configuration, as shown in the following snippet:

```
<job id="importInvoices"
     xmlns="http://www.springframework.org/schema/batch">
  <validator ref="jobParametersValidator" />                    #A
  <step id="decompress" next="readWriteInvoices">
  (...)
</job>

<bean id="jobParametersValidator"                               #B
      class="com.manning.sbia.ch02.batch.                       #B
[CA]     ImportInvoicesJobParametersValidator" />               #B
```

#A Sets job parameters validator
#B Declares job parameters validator bean

That's it, your invoice import job is able to validate its parameters before starting to decompress the archive. Sure you'll sleep better knowing your batch jobs won't crash miserably in the middle of their processing because someone started them without correct parameters - and Spring Batch makes this validation a breeze. Moreover the validation is confined in the job itself, so it doesn't depend on the way we launch it. That was the first step to make the job more robust, now let's see how to face some incorrect lines into the input flat file.

2.6.2 Skipping instead of failing

On a sunny day, the invoices import job will decompress the input archive, read each line of the extracted flat file, send them to the database and then exit successfully. Unfortunately, bad things happen and something can wrong. For example, if the `FlatFileIteReader` fails to read a single line of the flat file – because it's not correctly formatted for example – the job will immediately stops. Perhaps this is the behavior to adopt, but what if we can live with some incorrect records? In this case, we could just skip an incorrect line and keeps on reading. Spring Batch allows to declaratively choosing the skip policy when something goes wrong. Let's apply this to the importing step of our job.

Let's imagine a line of the flat file hasn't been generated correctly, like in the following snippet:

```
INVOICE_ID,CUSTOMER_ID,DESCRIPTION,ISSUE_DATE,AMOUNT
PR....210,9834,,2010-09-15,124.60
PR....211,9834,,2010-10-25,139.45
PR....212,6765,,2010-08-26,97,80                                #A
PR....213,4525,,2010-12-19,166.20
```

#A Amount incorrectly formatted

The amount field of the third record is incorrectly formatted (a comma instead of a dot as the decimal separator) and we're really out of luck as the comma is the field separator that helps Spring Batch to

tokenize a line. The `FlatFileItemReader` will throw a `FlatFileParseException` and with the default configuration we used, Spring Batch would stop immediately the process.

Assuming you can live with skipping some records instead of failing the whole job, you could change the job configuration to keep on reading when a `FlatFileParseException` is thrown, as shown in listing 2.11.

Listing 2.11 Setting the skip policy when reading records from the flat file

```
<job id="importInvoices"
     xmlns="http://www.springframework.org/schema/batch">
  <validator ref="jobParametersValidator" />
  <step id="decompress" next="readWriteInvoices">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWriteInvoices">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100"
          skip-limit="5">                                          #1
        <skippable-exception-classes>                              #2
          <include class="org.springframework.batch.               #2
[CA]        item.file.FlatFileParseException" />                   #2
        </skippable-exception-classes>                             #2
      </chunk>
    </tasklet>
  </step>
</job>
```

#1 Fails job if 5 records are skipped
#2 Skips flat file parse exceptions

The skip policy settings take place in the `chunk` element. We must set the `skip-limit` attribute (#1) to tell Spring Batch to stop the job if the number of skipped records exceeds this limit. Spring Batch can be tolerant but not too much! Then we can choose the exception classes that should trigger a skip (#2). We detail all the options of the `skippable-exception-classes` element in chapter 9, where we cover batch applications robustness. Here we just want to skip the line when the item reader throws a `FlatFileParseException`.

You can now launch the job with a flat file that contains incorrectly formatted lines and you'll see that Spring Batch will keep on running the job as long as the number of skipped items doesn't exceed the skip limit. Note we don't do any processing when something goes wrong: we could for example log that a line was incorrectly formatted. Don't worry, Spring Batch provides hooks to react to errors and we'll speak about them in chapter 10, which covers the execution flow of Spring Batch's jobs.

This ends the strengthening of the invoice import process: by now it doesn't only do its job quickly and efficiently but it's also more robust and can react accordingly to unexpected events, like incorrect parameters or fancy record line.

2.7 Summary

Congratulations, you've just implemented a whole batch process with Spring Batch! The import invoices job from the ACME company allowed you to get a good overview of what the framework is capable of. You discovered Spring Batch's sweet spot – chunk processing – by reading records from a flat file and writing them to a database. You've been able to use off-the-shelf components like the

`FlatFileItemReader` and implement a database `ItemWriter` for the business code. The thing to remember is that Spring Batch handles the plumbing and lets you focus on this business code. You've also seen that Spring Batch is not limited to chunk processing as you implemented a `Tasklet` to decompress a ZIP file.

Configuring the job ended up being quite simple thanks to the Spring lightweight container and the `batch` namespace. You even got help to run the batch from the command line, thanks to the `CommandLineJobRunner`.

At last you learned how to make the job more robust by validating the input parameters and dealing with incorrect data.

This is a lot of material, but you're now ready to implement your first Spring Batch jobs. Hopefully this chapter gave enough pointers to find the appropriate chapter when you'll need some information to address concerns with Spring Batch. You can also choose to stay on the track on the book and read chapter 3, where we cover the batch vocabulary that Spring Batch introduces and the benefits it brings to your applications.

3

Spring Batch's concepts

This chapter covers

- the domain language of batch application
- Spring Batch's infrastructure components
- The structure of a Spring Batch job
- How Spring Batch handles the execution of jobs

Chapter 2 provided a hands-on introduction to Spring Batch, as we saw how to implement a batch process from the business requirement to the batch implementation and to the running of the process. This introduction showed us how to get started with Spring Batch and gave us a good overview of the framework's features. It's time to strengthen this newly acquired knowledge by diving into batch applications' concepts and vocabulary.

Indeed, batch applications are complex and imply a lot of components, so we're going to cover the domain language of batch applications: we'll analyze their structure and give a name to their constituting parts. This vocabulary will be a great tool for communication, for us in this book, but also for you when you work on your own batch applications in a project team. Then we'll see that Spring Batch provides some technical services with implementations of infrastructure components, namely the job launcher and the job repository. Then we'll dive into the heart of a batch process: the job. We'll study how we structure a Spring Batch job and how the framework handles the execution of its processes.

By the end of this chapter, you'll have a better idea of how Spring Batch models batch applications and of the support the framework provides to implement them. Indeed, all these concepts are the foundations for efficient configuration and runtime behavior as well as features like restart: these are the starting points to unleash the power of Spring Batch. Let's start immediately by discovering the domain language of batch applications.

3.1 The domain language of batch

We've been a little negligent in this book so far: we used many technical terms in a lousy way as we didn't define them strictly. Be sure it was for your own good as we didn't want to hammer you with

concepts and definitions at the very beginning of the book. We chose the practical way by giving you a practical introduction to Spring Batch but it's time now to step back a little and take a more Cartesian approach. That's not a reason to bore you right now, so we'll make this short and even interesting!

We'll see in this section the domain language of batch: we'll decorticate the ecosystem of batch applications and give a name to each of its elements. Naming is hard but we're in luck as the Spring Batch project already did some analysis for us. Let's start first by looking at the benefits of using a domain language for batch applications.

3.1.1 Why use a domain language?

Using a well defined set of words in your batch applications will help you to structure them, enforce best practices and communicate with others. If there's a word for something, it means the corresponding concept matters and is part of the ecosystem of a batch application. By analyzing your business requirements and all the elements of the batch ecosystem, you'll end up finding a match and it'll help you to design your batch applications. Remember when we introduced chunk processing in chapter 2. This clearly identified important components in a typical batch process: the reader, the processor and the writer. By using the concept of chunk processing, you're more likely to structure your batch applications with readers, processors and writers.

The good news about chunk processing is it's a pattern really suited for batch applications. Indeed, it's a best practice in terms of memory consumption and performances. That's another benefit of the domain language: by following the structure it promotes, you're likely to enforce best practices. It doesn't mean you'll end up with perfect batch applications immediately, but at least you should avoid the most common pitfalls and you'll benefit immediately of years of experience in batch processing.

At last, you'll have a common vocabulary with other people working on batch applications. This will greatly improve communication and avoid confusion. You'll be able to switch from one project or company to another without having to learn a brand new vocabulary for the same notions.

Now you're aware of the benefits of having a domain language as a new tool to work with batch applications, so let's discover all these definitions.

3.1.2 Main elements of the domain language

In this sub-section, we're going to focus on the core components of Spring Batch applications and the next sub-section will cover external components that a Spring Batch application can communicate with. Figure 3.1 shows the main Spring Batch's components.

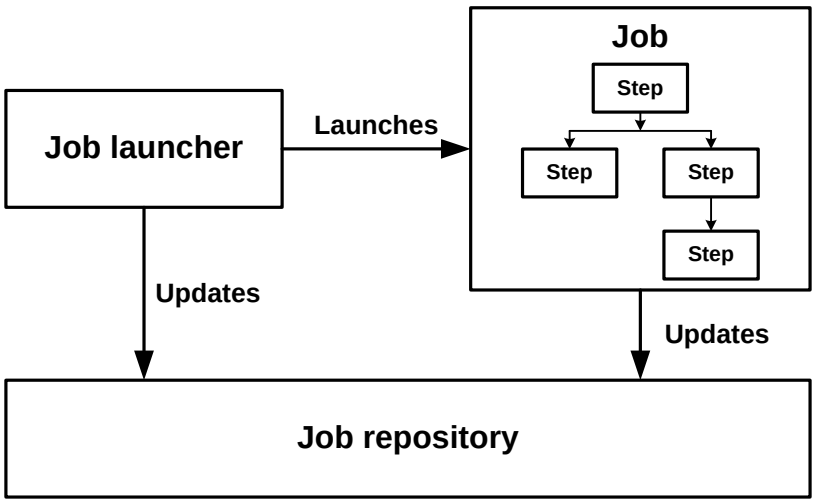

Figure 3.1 the main components of Spring Batch. The framework provides a job repository to store jobs' metadata, a job launcher to launch the jobs and the application developer configures and implements parts of jobs.

Figure 3.1 shows two kinds of components: the infrastructure components and the application components. The infrastructure components are the job repository and the job launcher. Spring Batch provides implementations for both and you'll need to configure these components but there's little chance you'll have to create your own implementations.

The application components in Spring Batch are the jobs and their constituting parts. From the previous chapters you know that Spring Batch provides components like item readers and writers that you'll only need to configure, but it's rather common to also implement your own. So writing batch jobs with Spring Batch is a combination of configuration and Java programming.

Figure 3.1 gives only the big picture to remain readable but table 3.1 gives a more comprehensive list of the components of a Spring Batch application and their interactions.

Table 3.1 The main components of a Spring Batch application

Component	Description
Job repository	The infrastructure components that persists the execution metadata of jobs.
Job launcher	The infrastructure component that starts the execution of jobs.
Job	The application component that represents a batch process.
Step	A phase in a job. A job is made of a succession of steps.
Tasklet	A transactional, potentially repeatable, processing occurring in a step.
Item	A record read from or written a data source.
Chunk	A set of items of a given size.

Item reader	A component responsible for reading items from a data source.
Item processor	A component responsible for processing (transforming, validating or filtering) a read item before it's written.
Item writer	A component responsible for writing a chunk to a data source.

From now we'll use the terms listed in a table 3.1. The remainder of this chapter will be dedicated to describe the concepts behind these terms, but before that we're going to see how the components of Spring Batch application interact with the outside world.

3.1.3 How Spring Batch connects to the world

A batch application isn't an island: it needs to interact with the outside world, just like any enterprise application. Figure 3.2 shows how a Spring Batch application interacts with the outside world.

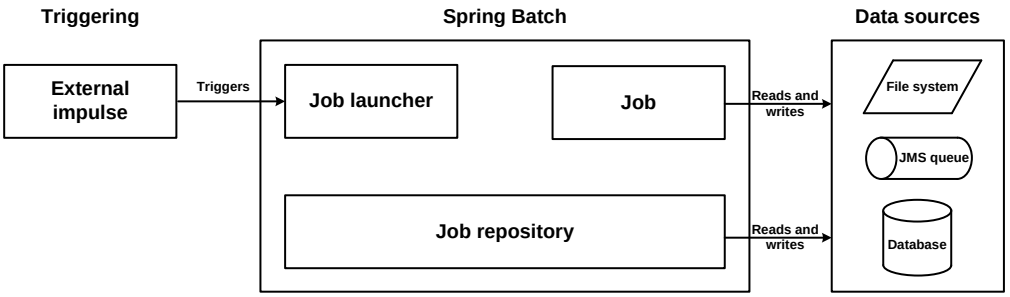

Figure 3.2 A Spring Batch application interacts mainly with triggering systems like schedulers and data sources like databases, JMS queue or files.

A job starts thanks to an impulse. You always end using the `JobLauncher` interface and `JobParameters`, but the impulse can come from anything: a system scheduler like cron that runs periodically a script which itself launches a Spring Batch process, a HTTP request that calls a web controller which launches the job and so on. Chapter 5 covers different scenarios to trigger the launching of your Spring Batch jobs.

Batch jobs are about processing data and that's why figure 3.2 shows that Spring Batch communicates with data sources. These data sources can be of any kind, the file system and a database being the most common, but a job can also read messages from a JMS queue or send messages.

NOTE

The job communicates with data sources, but so does the job repository. Indeed, the job repository stores the jobs' execution metadata in a database, to provide reliable monitoring and restart features.

Figure 3.2 doesn't show if Spring Batch needs to run in a specific container. Chapter 5 will say more about that, but for now, you just need to know that Spring Batch can run anywhere the Spring Framework can run: in its own Java process, in a web container, in an application or even in an OSGi container. This really depends on your requirements and Spring Batch is very flexible on this topic.

Now you know more about Spring Batch's core components, how they interact with each other and with the outside world, let's focus on the framework's infrastructure components, the job launcher and the job repository.

3.2 The Spring Batch's infrastructure

Spring Batch's infrastructure is about components that launch your batch jobs and store some metadata about their execution. As a batch application developer, you don't have to deal directly with those components, as they're here to provide transverse technical services. But you'll need to configure this infrastructure at least once in your Spring Batch application, so that's a good reason to mention them here, to know what these technical services are about.

This section will give an overview of the job launcher and job repository and their interactions, before showing how to configure the persistent implementation of job repository.

3.2.1 Launching jobs and storing their metadata

The Spring Batch's infrastructure is quite complex, but you'll need mainly to deal with two components: the job launcher and the job repository. These components are not nebulous concepts as they match two quite straightforward Java interfaces: `JobLauncher` and `JobRepository`. Let's start by studying the job launcher.

THE JOB LAUNCHER

As figure 3.2 shows, the job launcher is the entry point to launch Spring Batch's jobs: this is where the external world meets Spring Batch. The `JobLauncher` interface is quite simple:

```
package org.springframework.batch.core.launch;

(...)

public interface JobLauncher {

 public JobExecution run(Job job, JobParameters jobParameters)
    throws JobExecutionAlreadyRunningException,
           JobRestartException, JobInstanceAlreadyCompleteException,
           JobParametersInvalidException;

}
```

The `run` method accepts two parameters: a `Job`, which is typically a Spring bean configured with the batch namespace, and `JobParameters`, which are usually created on the fly, by the launching mechanism.

Who calls the job launcher? Stated in the sub-section about Spring Batch and the outside world, your own main Java program can use the job launcher to launch a job, or it can be a command line program, a system or Java scheduler.

The job launcher encapsulates launching strategies, like the fact that the job executes synchronously or asynchronously. Spring Batch provides an implementation of `JobLauncher`: the `SimpleJobLauncher`. We'll say more on its configuration in chapter 4 and its tweaking in chapter 5. Let's just say now that the `SimpleJobLauncher` class only launches a job, it doesn't create it, as it delegates this work to the job repository.

THE JOB REPOSITORY

The job repository maintains all the metadata related to the execution of jobs. Here is the definition of the `JobRepository` interface:

```
package org.springframework.batch.core.repository;

(...)

public interface JobRepository {

  boolean isJobInstanceExists(String jobName, JobParameters jobParameters);

  JobExecution createJobExecution(
    String jobName, JobParameters jobParameters)
      throws JobExecutionAlreadyRunningException, JobRestartException,
             JobInstanceAlreadyCompleteException;

  void update(JobExecution jobExecution);

  void add(StepExecution stepExecution);

  void update(StepExecution stepExecution);

  void updateExecutionContext(StepExecution stepExecution);

  void updateExecutionContext(JobExecution jobExecution);

  StepExecution getLastStepExecution(JobInstance jobInstance,
    String stepName);

  int getStepExecutionCount(JobInstance jobInstance, String stepName);

  JobExecution getLastJobExecution(String jobName,
    JobParameters jobParameters);

}
```

If you take a quick look at the `JobRepository` interface declaration, you'll see that the job repository provides all the services to manage the lifecycle of the execution of a batch job: creating it, updating it and so on.

This explains the interactions that figure 3.2 showed previously: the job launcher delegates the job creation to the job repository and a job calls the job repository during its execution to store its current state. This is really useful if you want to monitor how the execution of your jobs goes on but also if you want Spring Batch to restart a failed job exactly where it left off. Note that the Spring Batch's runtime handles all the calls to the job repository: the persistence of the job execution metadata is transparent for the application developer.

But what are these runtime metadata about? They can be the list of the executed steps, how many items were read, written or skipped, how long each step took and so on. We won't list them all here as you'll learn more about them when we'll study the anatomy of a job in section 3.3.

Spring Batch provides two implementations of `JobRepository`: the first one persists the metadata in memory, it's then useful for testing or when you don't want monitoring or restart capabilities. The second implementation stores the metadata in a relational database, and we're going to see immediately how to configure this implementation.

3.2.2 Configuring Spring Batch's infrastructure for a database

Thanks to the job repository implementation Spring Batch provides you can store your jobs' metadata in a database. This allows you to monitor the execution of your batch processes and know if they have failed or completed successfully. These persistent metadata also makes it possible to restart a failed job exactly where it stopped.

We're going to show you quickly how to configure this persistent job repository, to help us illustrate the concepts we're about to introduce in this chapter. You'll find all the details of the persistent job repository configuration in chapter 4.

The persistent job repository support provided by Spring Batch is made of the following:

- SQL scripts to create the necessary database tables – Spring Batch provides the scripts for the most popular database engines
- An database implementation of `JobRepository` – this implementation executes all the necessary SQL statements to insert, update and query the job repository's tables

Let's see now how to configure step by step the persistent repository.

CREATING THE JOB REPOSITORY'S DATABASE TABLES

The scripts to create the database tables are located in the core JAR file of Spring Batch, in the `org.springframework.batch.core` package. The scripts follow a naming convention: `schema-[database].sql` for creating the table and `schema-drop-[database].sql` for dropping them, where `[database]` is the name of the database engines. We're going to use PostgreSQL, so we'll use the `schema-postgresql.sql` file. Note you can use any other database engine that Spring Batch supports, like MySQL, Oracle or SQLServer.

All you have to do is to create the database and then execute the corresponding script on it.

CONFIGURING THE JOB REPOSITORY WITH SPRING

We already configured a job repository bean in chapters 1 and 2, but this was the in-memory implementation. Listing 3.1 shows how to configure the database implementation of job repository:

Listing 3.1 The configuration of a persistent job repository

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:batch="http://www.springframework.org/schema/batch"
  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-2.1.xsd">

  <batch:job-repository id="jobRepository"                          #1
    data-source="dataSource"                                        #1
    transaction-manager="transactionManager" />                     #1

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

  <bean id="dataSource"                                             #2
        class="org.springframework.jdbc.datasource.                 #2
```

```
[CA]  SingleConnectionDataSource">                                    #2
    <property name="driverClassName"                                  #2
      value="org.postgresql.Driver" />                                #2
    <property name="url" value="                                      #2
[CA]  jdbc:postgresql://localhost:5432/sbia_ch03" />                  #2
    <property name="username" value="app" />                          #2
    <property name="password" value="app" />                          #2
    <property name="suppressClose" value="true" />                    #2
  </bean>                                                             #2

  <bean id="transactionManager" class="org.springframework.jdbc.datasource.
[CA]  DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>

</beans>
```

#1 Declares persistent job repository
#2 Declares data source

There's a dedicated `job-repository` XML element in the `batch` namespace to create a persistent job repository (#1). The persistent job repository needs a data source and a transaction manager to work properly. Note that at #2 we used a data source implementation which holds a single JDBC connection and reuse it for each query. We did so because it's convenient and enough for a single-threaded application (like a batch process) but if you plan to use the data source for a concurrent application, you should use a connection pool, like Commons DBCP or c3p0. Note also that using the persistent job repository implementation doesn't change much the whole Spring Batch's infrastructure configuration, as we used the same implementation of job launcher as with the in-memory implementation.

Now we have the persistent job repository ready, why not take a look at it?

ACCESSING TO THE JOBS' METADATA

If you take a look at the job repository database, you'll see that the SQL script created 6 tables. If you're impatient, you can launch the job of the previous chapter to see how Spring Batch feeds the tables (that's what the `GeneratesJobMetaData` program does in the code samples of this chapter), but don't worry we'll analyze the content of the job repository later in this chapter because it'll help you understand how Spring Batch manages the execution of jobs.

Nevertheless, as we can't help showing you a teaser, figure 3.3 shows how you can access the job repository tables with the Spring Batch Admin web application to see the executions of a job.

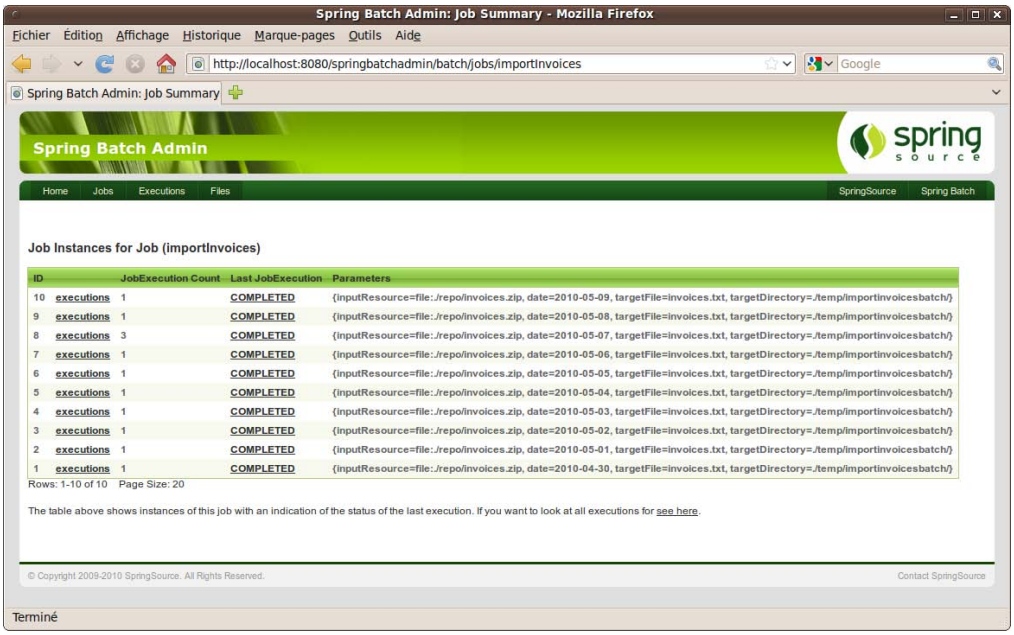

Figure 3.3 Jobs' metadata can help monitoring the executions of jobs. The Spring Batch Admin application lists here all the executions of the import invoices job.

You now have a ready persistent job repository that will be useful to illustrate the forthcoming runtime concepts. Let's see now the structural and runtime aspects of the core entity of Spring Batch, the job.

3.3 Anatomy of a job

The job is the central entity in a batch application: it is the batch process itself. Spring Batch defines a strict structure for its jobs and provides tools – the XML namespace – to configure this structure. This is the static part of the story. Spring Batch also provides strong foundations for the dynamic part, when it comes to execute a job. These foundations are about providing a reliable way to control which instance of a job is about to be executed and to be able to restart a job where it left off when it failed. This section explains these two sides – static for structure, dynamic for runtime – of a job.

3.3.1 The static story, a job is made of steps

A Spring Batch job is made of a succession of steps, which are configured in a Spring configuration file, thanks to the `batch` namespace. Let's delve into these concepts to see what they can bring to your batch applications.

STRUCTURE OF A JOB

Remember the import invoices job from chapter 2: it's made of a first step to decompress the incoming archive and of second step to import the records from the extracted file to the database. We could also have added a cleaning step, to delete the extracted file. Figure 3.4 depicts this job and its 3 successive steps.

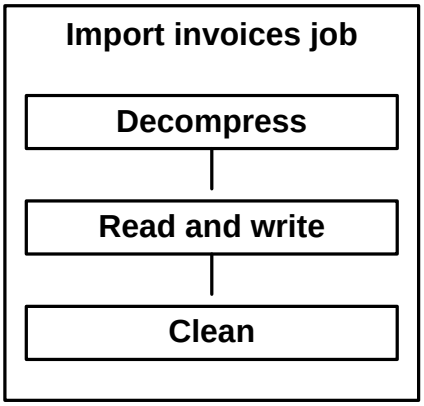

Figure 3.4 A Spring Batch job is made of steps, like the import invoices job, which is made in this case of 3 successive steps.

Decomposing a job in steps is convenient as steps are then easily testable, at least more than one monolithic job. You can also reuse the same step in several jobs: the decompress step in the import invoices job could be reused in any job that needs to decompress an archive, only the configuration would change.

Figure 3.4 shows a job made of 3 successive steps, but the sequence of steps doesn't have to be linear, as in figure 3.5, which shows a more advanced version of the import invoices job.

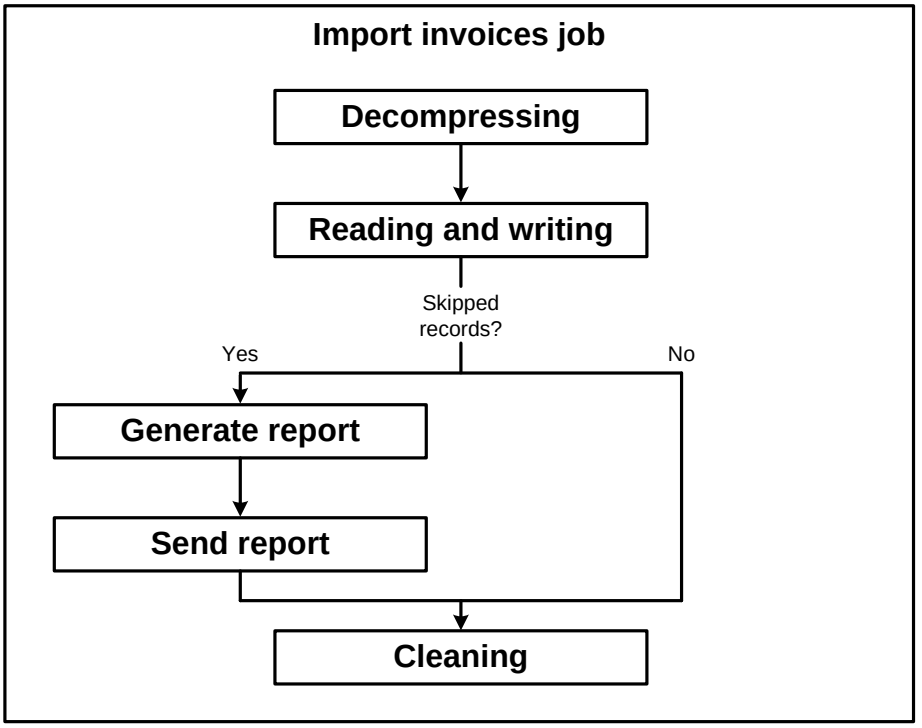

Figure 3.5 A Spring Batch can be made of a non-linear sequence of steps, like this version of the import invoices job which sends a report if some records were skipped.

Figure 3.5 illustrates a version of the import invoices job that generates a report and sends it to an administrator if the read/write step skipped some records. To decide which path a job should take, Spring Batch allows for decisions based on the status the previous step ended with (completed, failed) but also based on a custom logic (check the content of a database table for example). You can then create jobs with complex flows and react accordingly to any kind of eventuality (no incoming files to process, records skipped and so on). This brings flexibility and robustness to jobs, as you choose the granularity that best suits a job, instead of splitting a big, monolithic job into a set of smaller jobs and trying to orchestrate them thanks to exit codes or produced files with a scheduler. You'll also benefit from a clear separation of concerns between processing – implemented in steps – and the execution flow – configured declaratively or implemented in dedicated decision components. There will be less temptation to implement transition logic in steps and thus coupling them with each other.

Let's see some examples of the configuration of the structure of a job.

JOB CONFIGURATION

Spring Batch provides the `batch` namespace to configure the steps within a job. Listing 3.2 shows the code for the linear version of the import invoices job.

Listing 3.2 Configuring a linear flow with the batch namespace

```
<job id="importInvoicesJob">
  <step id="decompress" next="readWrite">                          #A
    <tasklet ref="decompressTasklet" />                            #B
  </step>
  <step id="readWrite" next="clean">                               #A
    <tasklet>
      <chunk reader="reader" writer="writer"                       #B
        commit-interval="100" />
    </tasklet>
  </step>
  <step id="clean">
    <tasklet ref="cleanTasklet" />                                 #B
  </step>
</job>
```

#A Sets execution flow
#B Refers to Spring beans

When a job is made only of a linear sequence of steps, using the `next` attribute of the `step` elements is enough. Listing 3.3 shows how the configuration looks like for the non-linear version of the import invoices job (figure 3.5).

Listing 3.3 Configuring a non-linear flow with the batch namespace

```
<job id="importInvoicesJob"
     xmlns="http://www.springframework.org/schema/batch">
  <step id="decompress" next="readWrite">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWrite" next="skippedDecision">                     #A
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100" />
    </tasklet>
  </step>
  <decision id="skippedDecision"                                   #B
            decider="skippedDecider">                              #B
    <next on="SKIPPED" to="generateReport"/>                       #B
    <next on="*" to="clean" />                                     #B
  </decision>                                                      #B
  <step id="generateReport" next="sendReport">                     #C
    <tasklet ref="generateReportTasklet" />                        #C
  </step>                                                          #C
  <step id="sendReport" next="clean">                              #C
    <tasklet ref="sendReportTasklet" />                            #C
  </step>                                                          #C
  <step id="clean">
    <tasklet ref="cleanTasklet" />
  </step>
</job>

<bean id="skippedDecider"                                          #D
      class="com.manning.sbia.ch03.structure.                      #D
[CA]  SkippedDecider" />                                           #D
```

#A Refers to flow decision logic
#B Defines decision logic
#C Declares optional steps
#D Declares decision logic bean

You notice from the previous snippet that the `batch` namespace is expressive and allows for readable job configuration. You'll also benefit from code completion and code validation when editing your XML job configuration, as long as your editor supports XML. An integrated development environment like SpringSource Tool Suite (based on Eclipse) provides also a graphical view of a job configuration, as shown in figure 3.6.

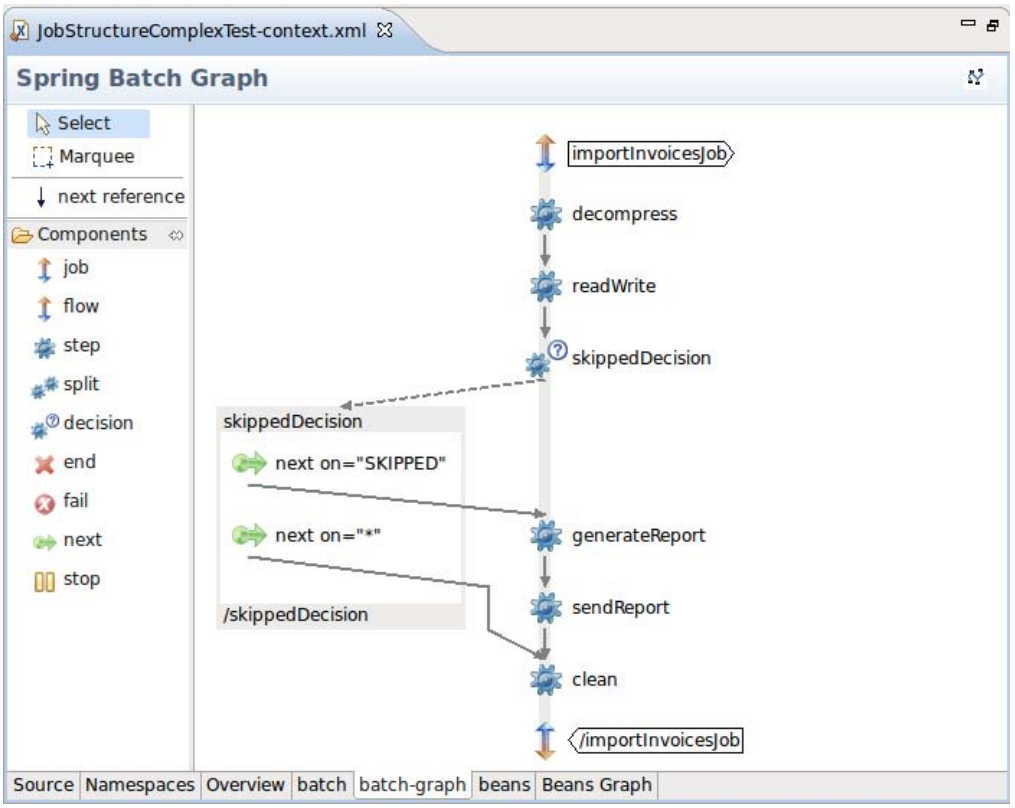

Figure 3.6 The flow of job viewed in SpringSource Tool Suite. As we can strictly define the structure of a Spring Batch job with XML, tools can provide a graphical view of this structure.

We won't go further into the details of configuration and execution flow, as these topics are thoroughly covered in chapters 4 and 10, respectively. Now you know that a Spring Batch job is made of steps and that you can control their flow, let's see of what a step can be made of.

THE TASKLETSTEP FOR PROCESSING

Spring Batch defines the `Step` Java interface to embody the concept of step and provides some implementations. As an application developer, the only implementation you care about is the `TaskletStep`, which delegates the processing to a `Tasklet` object. As you discovered in chapter 1, the `Tasklet` Java interface contains only one method, `execute`, to do some processing. So a Spring

Batch step will consist either in writing a custom implementation of `Tasklet` or using one provided by Spring Batch.

You'll implement your own `Tasklet` when you'll need to perform processing like decompressing files, calling a stored procedure or deleting some temporary files at the end of job.

If one of our steps follows the classic read-process-write batch pattern, you'll use Spring Batch's `ChunkOrientedTasklet`, which helps you read data efficiently, process them and write them in chunks.

NOTE

The `chunk` element of the `batch` namespace hides the use of the `ChunkOrientedTasklet`, so there's little chance you refer directly to this class.

You know by now all the static story of a job: it's made of steps, you can easily choose the succession of steps with Spring Batch and steps are implemented with `Tasklets`, either custom or chunk-oriented. Let's move on to the dynamic story.

3.3.2 The dynamic story, job instances and executions

As batch processes handle a lot of data automatically, being able to monitor what they're doing or what they did is a must-have for critical systems. This is especially useful when something goes wrong and you need to decide whether you should restart the job from the beginning or from where it left off. To do this, you need to strictly define the identity of a job run and also store reliably everything the job did during the run. This is quite a difficult task, but Spring Batch brings all the concepts and tools to reach these goals.

JOB, JOB INSTANCE AND JOB EXECUTION

We defined a *job* as a batch process, composed of a succession of steps. Spring Batch introduces the notions of job instance and job execution, which are related to the way the framework handles jobs at runtime. Table 3.2 gives the definition of these new notions and provides some examples.

Table 3.2 Definition of job, job instance and job execution

Term	Description	Example
Job	Batch process made of steps	The import invoices job
Job instance	Logical run of a job	The run of May 4th of the import invoices job
Job execution	Effective execution of a job instance	The first attempt to run the job instance of May 4th of the import invoices job

Figure 3.7 illustrates the cardinality between a job, its instances and their executions, for 2 days of executions of the import invoices job.

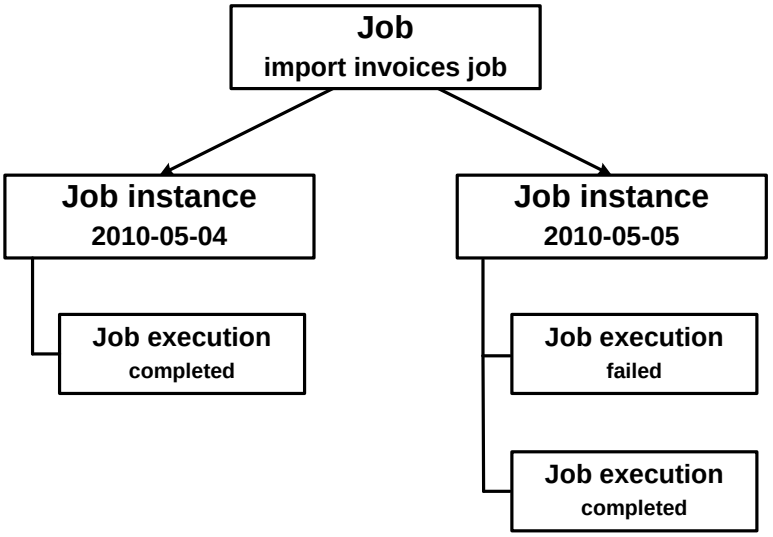

Figure 3.7 A job can have several job instances, which can have themselves several executions. The import invoices job executes daily, so it should have an instance for everyday and one or more corresponding executions, depending on whether the first one succeeds or fails.

Now we've know the identity of a job run with the job instance, let's see how to define a job instance in Spring Batch.

HOW TO DEFINE A JOB INSTANCE

In Spring Batch, a job instance is made of a job and of job parameters. When we speak about the instance of May 4th of our import invoices job, the date is the parameter that defines the job instance (along with the job itself). This is a simple yet powerful way to define a job instance, as you have full control over the job parameters, as shown in the following snippet:

```
jobLauncher.run(job, new JobParametersBuilder()
  .addString("date", "2010-05-04")
  .toJobParameters()
);
```

But what happens if you try to run the same job several times with the same parameters? It depends, as this is related to the lifecycle of job instances and job executions.

THE LIFECYCLE OF JOB INSTANCES AND JOB EXECUTIONS

There are several rules about the lifecycle of job instances and job executions:

- When you launch a job for the first time, the corresponding job instance and a first execution are created
- You can't launch the execution of a job instance if a previous one has already completed successfully
- You can't launch multiple executions of the same instance at the same time

Hopefully by now all these new concepts are clear for you, but let's make them crystal clear by doing some runs of our import invoices job and analyze the job metadata that Spring Batch stores in the database.

MULTIPLE RUNS OF THE IMPORT INVOICES JOB

The import invoices job we introduced in chapter 2 is supposed to run once a day, to import all the new and updated invoices coming from the billing system. We're going to run it several times to see how Spring Batch updates the job metadata with the persistent job repository we configured previously. Here's the sequence of the run we're going to make:

- Run the job for May 4th – the run will succeed
- Run the job a second time for May 4th – Spring Batch should not launch the job again, because it's already completed for this date
- Run the job for May 5th, but with a corrupted archive – the run will fail
- Run the job for May 5th, but with a correct archive – the run will succeed

WARNING

If you to try these runs yourself, you need first to create a `sbia_ch03` database in your PostgreSQL server and launch the `CreateJobRepositoryAndInvoiceTables` Java program to create the job repository and invoice tables.

Running the job for May 4th

Step	Description
1	Copy the `invoices.zip` file from the `input` directory into the root directory of the `ch03` project.
2	Open the `LaunchImportInvoicesJob` Java class and launch it: this will run the job for May 4th, 2010.
3	Launch the `LaunchSpringBatchAdmin` program from the code samples. It starts an embedded web container with the Spring Batch Admin application running on it.
4	Reach the instances of the import invoices job at the following URL: http://localhost:8080/springbatchadmin/batch/jobs/importInvoices. Figure 3.8 shows the graphical interface with the job instance the job repository created for this run.

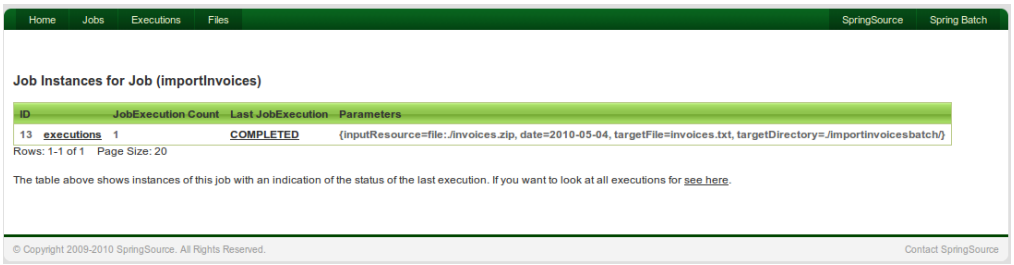

Figure 3.8 After the run of May 4th, Spring Batch created a corresponding instance in the job repository.

	The instance is marked as completed as the first and only execution completed successfully.
5	Follow the links from the job instance view and to get to the details of the corresponding execution, as shown in figure 3.9.

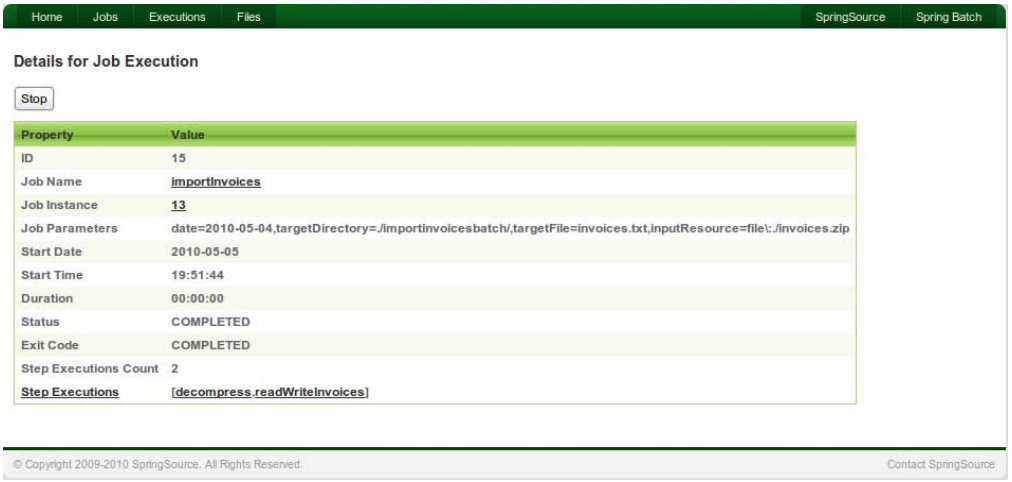

Figure 3.9 Details of the first and sole execution of May 4th (duration, number of steps executed). You can also learn about the job instance as the job parameters appear.

Note: you must read the job parameters to be sure of the identity of the execution. For example, the `date` job parameter tells you that this is an execution of the May 4th instance. The "Start Date" attribute indicates exactly when I ran the job.

Running the job a second time for May 4th

Step	Description
1	Execute the `LaunchImportInvoicesJob` class. You get an exception: that's because an execution already completed successfully, so you cannot launch another execution for the same instance.

Run the job for May 5th with a corrupted archive

Step	Description
1	Delete the `invoices.zip` file you copied and the `importinvoicesbatch` directory that's been created to decompress the archive.
2	Copy the `invoices_corrupted.zip` from the `input` directory into the root of the project and rename it to `invoices.zip`.

3	Simulate you're launching the job for May 5th, 2010 by changing the job parameters in `LaunchImportInvoicesJob`, like shown in the following snippet:
	`jobLauncher.run(job, new JobParametersBuilder()`
	`.addString("inputResource", "file:./invoices.zip")`
	`.addString("targetDirectory", "./importinvoicesbatch/")`
	`.addString("targetFile","invoices.txt")`
	`.addString("date", "2010-05-05")`

	`.toJobParameters()` `);`
4	Execute the `LaunchImportInvoicesJob` class. You get an exception.
5	Go to http://localhost:8080/springbatchadmin/batch/jobs/importInvoices and you'll see that the import invoices job has another instance, but this time it's not completed because the execution failed.

Run the job for May 5th with a correct archive

Step	Description
1	Replacing the corrupted archive by the correct one (the same as for the very first run).
2	Launch the job again.
3	Check on Spring Batch Admin that the instance of the instance of May 5th is now completed. Figure 3.10 shows the 2 executions of the May 5th instance.

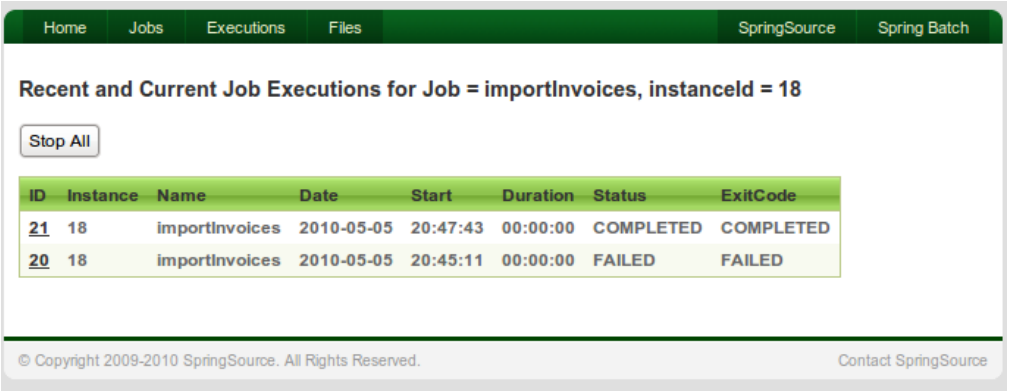

Figure 3.10 The 2 executions of the job instance of May 5th. The first failed because of a corrupted archive but the second completed successfully, so the instance is considered completed as well.

Note: if you want to do some fresh tests after you ran the job several times, you need to empty all the job repository tables. On PostgreSQL, you can do this with the following command: `truncate batch_job_instance cascade`.

We've just put the concepts of job instance and job execution in practice. To do so, we used a persistent job repository, which is convenient to visualize the instances and the executions. Here the job metadata was useful to illustrate the concepts but they will also be used for monitoring in a production system. They're also essential to restart a job that has failed – a topic that chapter 9 covers in depth.

3.4 Summary

This chapter was rich in new terms and concepts! Perhaps you already knew some of them, but now the picture is less blurry and you're now able to use a well defined vocabulary for all the parts of your batch applications. You even know how the Spring Batch framework chose to model them and that's a very important requirement to understand how it will help you solve problems. For example, if restarting

your failed jobs is an important requirement in your batch applications, you know now how Spring Batch implements this feature – by storing all the details of job executions – and you'll know exactly about the possibilities and limitations of this mechanism.

With the picture of the Spring Batch model in mind, let's move on to the next chapter to see how to effectively configure our batch applications with the Spring lightweight container and the `batch` namespace.

4

Batch configuration

This chapter covers

- Configuring batch processes using Spring Batch XML
- Configuring batch jobs and related entities
- Using advanced techniques to improve configuration

In the previous chapter, we explored Spring Batch foundations, described all Spring Batch concepts and entities, and looked at their interactions. The previous chapter introduced configuring and implementing the structure of batch jobs and related entities with Spring Batch XML.

In this chapter, we continue with our case study: reading products from files, processing products, and integrating products in the database. Configuring this process will serve as the backdrop describing all of Spring Batch's configuration capabilities.

After describing Spring Batch XML capabilities, we show how to configure batch jobs and related entities with this dedicated XML vocabulary. We will also look at configuring a repository for batch execution metadata. In last part of this chapter, we focus on advanced configuration topics and describe how to make configuration easier.

How should you read this chapter? You can use it as a reference for configuring Spring Batch jobs and either skip it or come back to it for a specific configuration need. Alternatively, you can read it in its entirety to get an overview of nearly all the features available in Spring Batch. We say an "overview" because when you learn about configuring a skipping policy for example, you won't learn all the subtleties of this topic. That's why you'll find information in dedicated sections in other chapter to drill down into difficult topics.

Chapter author name: Templier, Gregory

4.1 The Spring Batch XML vocabulary

Like all projects in the Spring portfolio and the Spring framework itself, Spring Batch provides a dedicated XML vocabulary and namespace to configure its entities. This feature leverages the Spring XML schema-based configuration introduced in Spring 2 and simplifies bean configuration. This allows configurations to operate at a high-level by hiding complexity when configuring entities and related Spring-based mechanisms.

In this section, we describe how to use Spring Batch XML and the facilities it offers for batch configuration. Without this vocabulary, we would need intimate knowledge of Spring Batch internals and entities that make up the batch infrastructure, which can be tedious and complex to configure.

4.1.1 Using the Spring Batch XML namespace

Like most components in the Spring portfolio, Spring Batch configuration is based on a dedicated Spring XML vocabulary and namespace. By hiding internal Spring Batch implementation details, this vocabulary provides a simple way to configure core components like jobs and steps, as well as the job repository used for job metadata. Note that the previous chapter describes all these components. The vocabulary also provides simple ways to define and customize batch behaviors.

Before we get into the XML vocabulary and component capabilities, you first need to know how use the Spring Batch XML in Spring configuration files. In listing 4.1, we declare the `batch` namespace prefix and use it in child XML elements mixed with other namespace prefixes, like the `beans` namespace prefix mapped to Spring XML.

Listing 4.1 Spring Batch XML namespace and prefix

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

  <batch:job id="importProductsJob">
#D
    (...)
  </batch:job>                                                            #D

</beans>
```

#A Uses beans namespace as default
#B Declares Spring Batch namespace prefix
#C Imports schemas
#D Uses Spring Batch namespace prefix

Spring Batch uses the Spring standard mechanism to configure a custom XML namespace: The Spring Batch XML vocabulary is implemented in Spring Batch jars, automatically

discovered, and handled by Spring. In listing 4.1, because Spring XML Beans uses the default namespace, we qualify each Spring Batch XML element with the `batch` namespace prefix.

Note that a namespace prefix can be whatever you want; in our examples, we use the `batch` and `beans` prefixes by convention.

Spring XML schema-based configuration

As of version 2.0, Spring uses an XML schema-based configuration system. XML schemas replace the previous DTD-driven configuration system that mainly used two tags: `bean` for declaring a bean and `property` for injecting dependencies (with our apologies to Spring for this quick-and-dirty summary.) While this approach works for creating beans and injecting dependencies, it is not sufficient to define complex tasks. The DTD bean and property mechanism is unable to hide complex bean creation. This is a shortcoming in configuring advanced features like AOP and security and should be clear to former Spring Security users. Before version 2.0, XML configuration used to be non-intuitive and verbose.

Spring 2.0 introduced a new, extensible, schema-based XML configuration system. On the XML side, XML schemas describe the syntax; and on the Java side, corresponding namespace handlers encapsulate the bean creation logic. The Spring framework provides namespaces for its modules (AOP, transaction, JMS and so on) and other Spring-based projects can benefit from the namespace extension mechanism to provide their own namespaces. Each Spring portfolio project comes with one or more dedicated vocabularies and namespaces to provide the most natural and appropriate way to configure objects using module-specific tags.

Listing 4.2 declares the Spring Batch namespace as the default namespace in the root XML element. In this case, the elements without a prefix correspond to Spring Batch elements. Using this configuration style, you do not need to repeat using the Spring Batch namespace prefix for each Spring Batch XML element.

Listing 4.2 Using the Spring Batch namespace as the default namespace

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

  <job id="importProductsJob">                                           #D
    (...)
```

```
    </job>                                                          #D

</beans:beans>
```

#A Uses Spring Batch namespace as default
#B Declares Spring beans namespace with beans prefix
#C Imports schemas
#D Uses Spring Batch namespace without prefix

We've defined a configuration using Spring Batch XML and can now leverage all the facilities it provides. In the next section, we focus on Spring Batch XML features and describe how to configure and use those capabilities.

4.1.2 Spring Batch XML features

Spring Batch XML is the central feature in Spring Batch configurations. You'll use this XML vocabulary to configure all batch entities described in chapter 3, "Spring Batch concepts". Table 4.1 lists and describes the main tags in Spring Batch XML.

Table 4.1 Main tags in Spring Batch XML

Tag name	Description
job	Configures a batch job.
step	Configures a batch step.
tasklet	Configures a tasklet in a step.
chunk	Configures a chunk in a step.
job-repository	Configures a job repository for metadata.

Spring Batch XML configures the structure of batches but specific entities need to be configured using Spring features. Spring Batch XML provides the ability to interact easily with standard Spring XML. You can configure other entities like item readers and writers as simple beans and then reference them from entities configured with the Spring Batch XML namespace. Figure 4.1 describes the possible interactions between the Spring Batch namespace and the Spring default namespace.

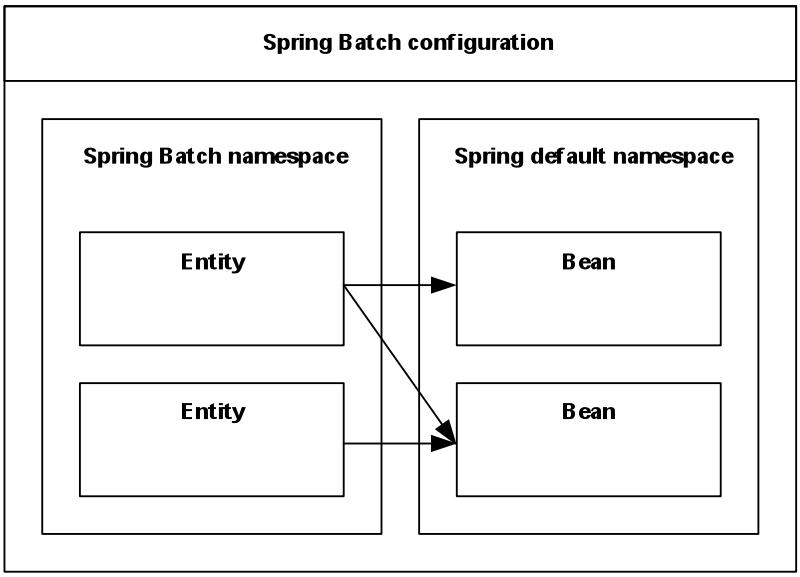

Figure 4.1 Interactions between Spring Batch and Spring vocabularies. The batch vocabulary defines the structure of batches. Some batch entities like a job can refer to Spring beans defined with the beans vocabulary, like item readers and writers.

Now that you've seen the capabilities provided for Spring Batch configuration, it's time to dive into details. In our case study, these capabilities let us configure a batch job and its steps. Next, let's describe how to create such a configuration.

4.2 Configuring jobs and steps

As described in the previous chapter, the central entities in Spring Batch are jobs and steps, which describe the details of batch processing. The use case here consists in defining what the batch must do and how to organize its processing. For our examples, we will use our case study, the online store. After having defined the job, we will progressively extend it by adding internal processing.

We focus here on how to configure the core entities of batch processes; we also examine their relationships at the configuration level. Before focusing on jobs, steps and transactions, let's look at the big picture.

4.2.1 Job entities hierarchy

Spring Batch XML makes the configuration of jobs and related entities easier. We don't need to configure Spring beans corresponding to internal Spring Batch objects, instead we can

work at a higher level of abstraction, specific to Spring Batch. Spring Batch XML configures batch components such as job, step, tasklet, and chunk, as well as their relationships. Together, all these elements make up batch processes. Figure 4.2 depicts this hierarchy.

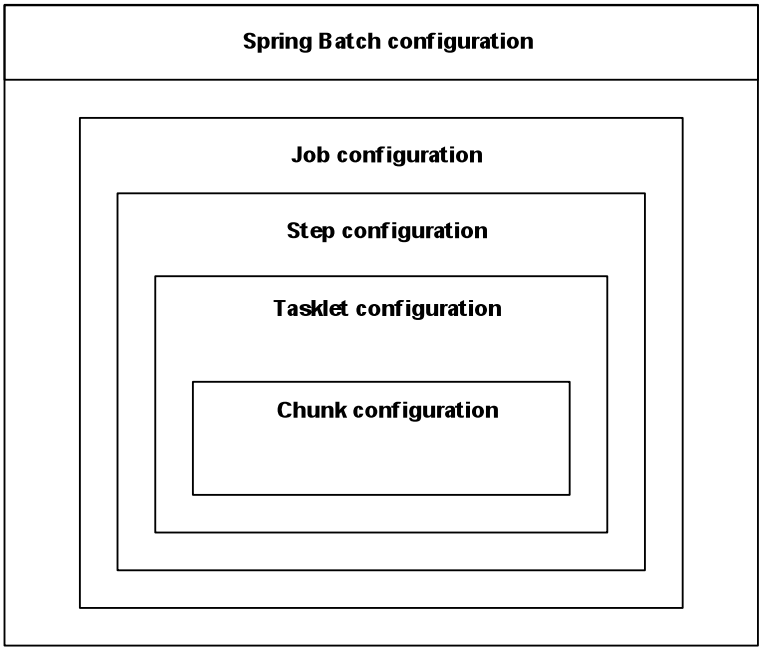

Figure 4.2 The entity configuration hierarchy for batch processing

Within Spring Batch XML, this hierarchy corresponds to nested XML elements with the ability to specify parameters at each level, as shown in the listing 4.3. For example, a job defined with the `job` element can contain one or more steps, which you define with `step` elements within the `job` element. Similar types of configurations can be used for steps, tasklets and chunks.

Listing 4.3 Nested configuration of a job

```
<batch:job id="importProductsJob">
  (...)
  <batch:step id="readWriteStep">
    <batch:tasklet transaction-manager="transactionManager">
      <batch:chunk
               reader="productItemReader"
               processor="productItemProcessor"
               writer="productItemWriter"
               commit-interval="10"/>
    </batch:tasklet>
```

```
    </batch:step>
  </batch:job>
```

For our case study, we use these nested elements in order to define each job, particularly the reading, processing, and writing logic.

Now that we've described high-level configuration concepts for Spring Batch entities, let's examine configuration details using Spring Batch XML.

4.2.2 Configuring jobs

When implementing a batch application with Spring Batch, the top-level entity is the job and it is the first entity we configure when defining a batch process. In the context of our case study, the job is a processing flow that imports and handles products for the web store. We introduced the job concept in chapter 3, section 3.3, "Anatomy of a job".

To configure a job, we use the Spring Batch XML `job` element and the attributes listed in Table 4.2.

Table 4.2 Job attributes

Job attribute name	Description
id	Identifies the job.
restartable	Whether Spring Batch can restart the job. The default is `true`.
incrementer	Refers to an entity used to set job parameter values. This entity is required when trying to launch a batch job through the `startNextInstance` method of the `JobOperator` interface.
abstract	Whether the job definition is abstract. If `true`, this job is a parent job configuration for other jobs. It doesn't correspond to a concrete job configuration.
parent	Defines the parent of this job.
job-repository	Specifies the job repository bean used for the job. Defaults to a `jobRepository` bean if none specified.

The attributes `parent` and `abstract` deal with configuration inheritance in Spring Batch; for details, see section 4.5.3, "Configuration inheritance". Let's focus here on the `restartable`, `incrementer` and `job-repository` attributes.

The `restartable` attribute specifies whether Spring Batch can restart a job. If `false`, Spring Batch cannot start the job more than once; if you try, Spring Batch throws the exception `JobRestartException`. The following snippet describes how to configure this behavior for a job:

```
<batch:job id="importProductsJob" restartable="false">
    (...)
```

```
</batch:job>
```

The `job-repository` attribute is a bean identifier that specifies which job repository to use for the job. In section 4.4, "Configuring the job repository", we describe this task.

The `incrementer` attribute provides a convenient way to create new job parameter values. Note that the `JobLauncher` does not need this feature because we must provide all parameter values explicitly. This isn't the case when the `startNextInstance` method of the `JobOperator` class launches a job. In that case, the method needs to determine new parameter values and use an instance of the `JobParametersIncrementer` interface:

```
public interface JobParametersIncrementer {
  JobParameters getNext(JobParameters parameters);
}
```

The `getNext` method can use the parameters from the previous job instance in order to create new values.

You specify an incrementer object in the job configuration using the `incrementer` attribute of the `job` element, for example:

```
<batch:job id="importProductsJob" incrementer="customIncrementer">
  (...)
</batch:job>

<bean id="customIncrementer" class="com.manning.sbia
                        [CA].configuration.job.CustomIncrementer"/>
```

Code continuation character in code

Section XX, "XX" in the next chapter describes the use of the `JobOperator` class in more detail.

Besides these attributes, the `job` element also supports nested elements to configure listeners and validators. We describe listeners in section 4.4.5, "Listening entities".

You can configure Spring Batch to validate job parameters and check that all required parameters are present before starting a job. To validate job parameters, you implement the `JobParametersValidator` interface:

```
public interface JobParametersValidator {
  void validate(JobParameters parameters)
    throws JobParametersInvalidException;
}
```

The `validate` method throws a `JobParametersInvalidException` if a parameter is invalid. Spring Batch provides a default implementation of this interface with the `DefaultJobParametersValidator` class that suits most use cases. This class allows you to specify which parameters are required and which are optional. Listing 4.4 describes how to configure and use this class in a job.

Listing 4.4 Configuring a job parameter validator

```
<batch:job id="importProductsJob">
  (...)
  <batch:validator ref="validator"/>                                    #1
```

```
</batch:job>

<bean id="validator" class="org.springframework.batch                    #2
                        [CA].core.job.DefaultJobParametersValidator">
  <property name="requiredKeys">                                         #3
    <set>
      <value>date</value>
    </set>
  </property>
  <property name="optionalKeys">                                         #3
    <set>
      <value>productId</value>
    </set>
  </property>
</bean>
```

#1 Specifies the job validator
#2 Configures a default validator
#3 Sets required and optional parameter keys

Cueballs in code and text

The `validator` element's `ref` attribute at #1 references the validator bean #2. This configuration uses a bean of type `DefaultJobParametersValidator` at #2 to specify the required and optional parameter keys. We use the `requiredKeys` and `optionalKeys` properties of the validator class to set these values at #3.

Now that we've described how to configure a job element, let's look at configuring job steps to define exactly what processing takes place in that job.

4.2.3 Configuring steps

Here, we go down a level in the job configuration and describe what makes up a job: steps. Don't hesitate to refer back to figure 4.4 to get a view of the relationships between all the batch entities. A step is a phase in a job; chapter 3, section 3.3.1 "Modeling jobs with steps" describes these concepts. Steps define the sequence of actions a job will take, one at a time. In our use case, we receive products in a compressed file; the job first decompresses the file before importing and saving the products in the database, as described in figure 4.3.

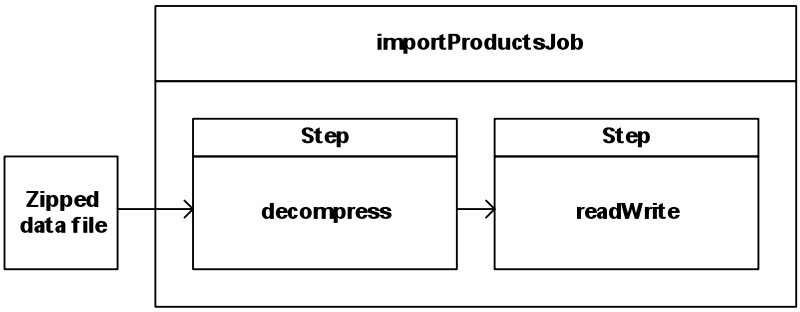

Figure 4.3 The steps of the import products job

You configure a job step using the `step` element and the attributes listed in table 4.3.

Table 4.3 Step attributes

Step attribute name	Description
next	The next step to execute in a sequence of steps.
parent	The parent of the step configuration.
abstract	Whether the step definition is abstract. If `true`, this step is a parent step configuration for other steps. It doesn't correspond to a concrete step configuration.

The attributes `parent` and `abstract` deal with configuration inheritance in Spring Batch; for details, see section 4.5.3, "Configuration inheritance".

Configuring a step is simple because it is a container for a task, executed at a specific point in the flow of a batch process. For our use case, we define what steps take place and in what order. To start things off, product files come in as compressed data files to optimize file uploads. A first step decompresses the data file and a second step processes the data. The following fragment describes how to configure this flow:

```
<job id="importProductsJob">
  <step id="decompress" next="readWrite">                    #1
    (...)
  </step>
  <step id="readWrite">                                      #2
    (...)
  </step>
</job>
```

#1 Configures the decompress step
#2 Configures the readWrite step

Cueballs in code and text

We always define the `step` element #1 as child element of the `job` element. Spring Batch uses the `id` attribute to identify objects in a configuration. This aspect is particularly important to order to use steps in a flow. The `next` attribute of the `step` element is set to define which step to execute next. In the previous fragment, the step identified as `decompress` is executed before the step identified as `readWrite`.

Now that we've described the job and step elements, let's continue our tour of Spring Batch core objects with tasklet and chunk, which define what happens in a step.

4.2.4 Configuring tasklets and chunks

The tasklet and chunk are step elements used to specify processing. Chapter 3, section 3.3.1 "Modeling jobs with steps" describes these concepts. To import products, we successively configure how to import product data, how to process products, and where to put products in the database, as described in figure 4.4.

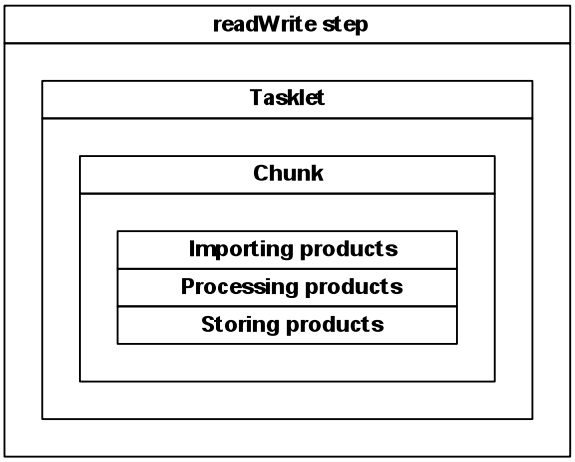

Figure 4.4 Import product tasklet and chunk configurations in our use case

In the next sections, we describe how to configure tasklets and chunks. Let's start with the tasklet.

TASKLET

A tasklet corresponds to a transactional, potentially repeatable, process occurring in a step. You can write your own tasklet class by implementing the `Tasklet` interface or use the tasklet implementations provided by Spring Batch. Implementing your own tasklets is useful for specific tasks, like decompressing an archive or cleaning a directory. Spring Batch

provides more generic tasklet implementations, to call system commands or to do chunk-oriented processing.

To configuring a tasklet, you define a `tasklet` element within a `step` element. Table 4.4 lists the attributes of the `tasklet` element.

Table 4.4 Tasklet attributes

Attribute	Description
ref	A Spring bean identifier whose class implements the `Tasklet` interface. You must use this attribute when implementing a custom tasklet.
transaction-manager	The Spring transaction manager to use for tasklet transactions. By default, a tasklet is transactional and the default value of the attribute is `transactionManager`.
start-limit	The number of times Spring Batch can restart a tasklet for a retry.
allow-start-if-complete	Whether Spring Batch can restart a tasklet even if it completed for a retry.

Listing 4.5 shows how to use the last three attributes of the `tasklet` element, whatever the tasklet type. Because we want to write the products from the compressed import file to a database, we need to specify a transaction manager to handle transactions associated with inserting products in the database. Listing 4.5 also specifies additional parameters to define our restart behavior.

Listing 4.5 Configuring tasklet attributes

```
<batch:job id="importProductsJob">
  (...)
  <batch:step id="readWriteStep">
    <batch:tasklet
              transaction-manager="transactionManager"                    #1
              start-limit="3"                                             #2
              allow-start-if-complete="true">                             #2
      (...)
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="transactionManager" class="(...)">
  (...)
</bean>
```

#1 Specifies transaction manager
#2 Defines retry and restart behaviour

Cueballs in code and text

The `transaction-manager` attribute #1 contains the bean identifier corresponding to the Spring transaction manager to use during the tasklet's processing. The bean must implement

the Spring `PlatformTransactionManager` interface. The attributes `start-limit` and `allow-start-if-complete` #2 specify that Spring Batch can restart the tasklet three times in the context of a retry even if the tasklet has completed. We describe in section 4.3.5, "Configuring transactions", how to control rollback for a step.

In the case of a custom tasklet, you can reference the Spring bean implementing the `Tasklet` interface with the `ref` attribute. Spring Batch delegates processing to this class when executing the step. In our use case, decompressing import files does not correspond to processing that Spring Batch natively supports, so we need a custom tasklet to implement decompression. The following snippet describes how to configure this tasklet:

```
<job id="importProductsJob">
  <step id="decompress" next="readWrite">
    <tasklet ref="decompressTasklet" />                                    #A
  </step>
</job>
```

#A Configures tasklet

Listing 2.5 in chapter 2 shows the code for the `decompressTasklet` tasklet.

Spring Batch also supports using chunks in tasklets. The `chunk` child element of the `tasklet` element configures chunk processing. Note that you do not need to use the `ref` attribute of the `tasklet` element. On the Java side, the `ChunkOrientedTasklet` class implements chunk processing.

Configuring a tasklet can be simple, but to implement chunk processing, the configuration gets more complex because more objects are involved. Next, let's look at chunk processing with a tasklet.

CHUNK-ORIENTED TASKLET

Spring Batch provides a tasklet class to process data in chunks: the `ChunkOrientedTasklet`. You typically use chunk-oriented tasklets for read-write processing. In chunk processing, Spring Batch reads data chunks from a source, transforms, validates, and then writes data chunks to a destination. In our case study, this corresponds to importing products into the database.

To configure chunks objects, we use an additional level of configuration using the `chunk` element with the attributes listed in table 4.5.

Table 4.5 Chunk attributes

Chunk attribute name	Description
reader	The bean identifier used to read data from a chunk. The bean must implement the Spring Batch `ItemReader` interface.
processor	The bean identifier used to process data from a the chunk. The bean must implement the Spring Batch `ItemProcessor` interface.

writer	The bean identifier used to write data from a chunk. The bean must implement the Spring Batch `ItemWriter` interface.
commit-interval	The number of items to process before issuing a commit.
skip-limit	The maximum number of skips during processing of the step. If processing reaches the skip limit, the next exception thrown on item processing (read, process, or write) will cause the step to fail.
skip-policy	A skip policy bean that implements the `SkipPolicy` interface.
retry-policy	A retry policy bean that implements the `RetryPolicy` interface.
retry-limit	The maximum number of retries.
cache-capacity	The cache capacity of the retry policy.
reader-transactional-queue	When reading item from a JMS queue, whether reading is transactional.
processor-transactional	Whether the processor used includes transactional processing.
chunk-completion-policy	A completion policy bean for the chunk that implements the `CompletionPolicy` interface.

The four first attributes (`reader`, `writer`, `processor`, `commit-interval`) in table 4.5 are the most commonly used in chunk configuration. These attributes defines which entities are involved in processing chunks and the number of items to process before committing.

Listing 4.6 Using tasklet configuration attributes

```
<batch:job id="importProductsJob">
  (...)
  <batch:step id="readWrite">
    <batch:tasklet>
      <batch:chunk
             reader="productItemReader"                    #1
             processor="productItemProcessor"              #1
             writer="productItemWriter"                    #1
             commit-interval="15"/>                        #2
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="productItemReader" class="(...)">
  (...)
</bean>

<bean id="productItemProcessor" class="(...)">
  (...)
</bean>

<bean id="productItemWriter" class="(...)">
  (...)
</bean>
```

#1 Specifies entities used by the chunk
#2 Specifies commit interval

Cueballs in code and text

The attributes `reader`, `processor` and `writer` #1 correspond to Spring bean identifiers defined in the configuration. For more information on these topics, we refer you to chapter 6, "Reading data" for configuring item readers; chapter 7, "Writing data", for configuring item writers; and chapter 8, "Processing data", for configuring item processors. The `commit-interval` attribute #2 defines that Spring Batch will execute a database commit after processing each fifteenth elements.

Other attributes deal with configuring the skip limit, retry limit and completion policy aspects of a chunk. Listing 4.7 shows how to use these attributes.

Listing 4.7 Configuring chunk retry, skip and completion

```
<batch:job id="importProductsJob">
  (...)
  <batch:step id="readWrite">
    <batch:tasklet>
      <batch:chunk
             (...)
             skip-limit="20"                                          #1
             retry-limit="3" cache-capacity="100"                     #2
             chunk-completion-policy="timeoutCompletionPolicy"/>      #3
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="timeoutCompletionPolicy"                                    #3
      class="org.springframework.batch.repeat
                    [CA].policy.TimeoutTerminationPolicy">
  <constructor-arg value="60"/>
</bean>
```

#1 Configures chunk skip limit
#2 Configures chunk retry limit
#3 Configures chunk completion policy

Code continuation character in code, cueballs in code and text

In listing 4.7, the `skip-limit` attribute #1 configures the maximum number of items that Spring Batch can skip. The `retry-limit` attribute #2 sets the maximum number of retries, and the `cache-capacity` attribute sets the cache capacity for retries. The `chunk-completion-policy` attribute #3 configures the completion policy in order to define a chunk processing timeout.

Chapter author name: Templier, Gregory

We've described rather briefly how to configure skip, retry, and completion in steps. We will look at this topic in more detail in chapter 9, "Making batch processes more robust" where we aim for batch robustness and define error handlers.

The last attributes correspond to more advanced configurations regarding transactions. We will describe these in section 4.3.5, "Configuring transactions".

Most of the attributes described in table 4.5 have equivalent child elements to allow embedding beans within the chunk configuration. These beans are anonymous and specially defined for the chunk. Table 4.6 describes `chunk` children elements usable in this context.

Table 4.6 Chunk child elements

Chunk child element	Description
reader	Corresponds to the `reader` attribute.
processor	Corresponds to the `processor` attribute.
writer	Corresponds to the `writer` attribute.
skip-policy	Corresponds to the `skip-policy` attribute.
retry-policy	Corresponds to the `retry-policy` attribute.

Listing 4.8 describes how to rewrite listing 4.6 using child elements instead of attributes for the `reader`, `processor`, and `writer`.

Listing 4.8 Using child elements in the tasklet configuration

```
<batch:job id="importProductsJob">
  (...)
  <batch:step id="readWrite">
    <batch:tasklet>
      <batch:chunk commit-interval="15">
        <batch:reader>                                          #A
          <bean class="(...)">
            (...)
          </bean>
        </batch:reader>
        <batch:processor>                                       #A
          <bean class="(...)">
            (...)
          </bean>
        </batch:processor>
        <batch:writer>                                          #A
          <bean class="(...)">
            (...)
          </bean>
        </batch:writer>
      </batch:chunk>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

#A Uses child element instead of attribute

You can configure other objects with child elements in chunks. Table 4.7 lists these additional elements.

Table 4.7 Additional chunk child elements

Chunk child element name	Description
retry-listeners	Retry listeners; see section 4.5.3, “Listening entities”.
skippable-exception-classes	A list of exceptions triggering skips.
retryable-exception-classes	A list of exceptions triggering retries.
streams	Each stream element involved in the step. By default, objects referenced using a `reader`, `processor` and `writer` are automatically registered. You don’t need to specify them again here.

The `chunk` element can configure which exceptions trigger skips and retries using, respectively, the elements `skippable-exception-classes` and `retryable-exception-classes`. Listing 4.9 shows these elements specifying which exceptions will trigger an event (`include` child element) and which ones to will not (`exclude` child element).

Listing 4.9 Configuring skippable exceptions

```
<batch:job id="importProductsJob">
  (...)
  <batch:step id="readWrite">
    <batch:tasklet>
      <batch:chunk commit-interval="15"
                   skip-limit="10">
        <skippable-exception-classes>
          <include class="org.springframework.batch                    #1
                      [CA].item.file.FlatFileParseException"/>
          <exclude class="java.io.FileNotFoundException"/>             #2
        </skippable-exception-classes>
      </batch:chunk>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

#1 Specifies exceptions triggering skips
#2 Specifies exceptions to exclude

Code continuation character in code, cueballs in code and text

You can use the same mechanism with the `retryable-exception-classes` element as we did for the `skippable-exception-classes` element to configure retries. The

following fragment configures retries when `DeadlockLoserDataAccessException` exceptions are caught:

```
<batch:chunk commit-interval="15" retry-limit="3">
  <retryable-exception-classes>
    <include
         class="org.springframework.dao.DeadlockLoserDataAccessException"/>
  </retryable-exception-classes>
</batch:chunk>
```

The last feature in table 4.7 deals with streams. We provide here a short description of the feature and show how to configure it. Chapter 9, "Making batch processes more robust" provides more details on this topic. Streams provide the ability to save state between executions for step restarts. In fact, the step needs to know which instance is a stream (by implementing the `ItemStream` interface.) Spring Batch automatically registers as streams everything specified in the reader, processor, and writer attributes. Note that it's not the case when the step does not directly reference entities. That's why these entities must latter be explicitly registered as stream as described in figure 4.5.

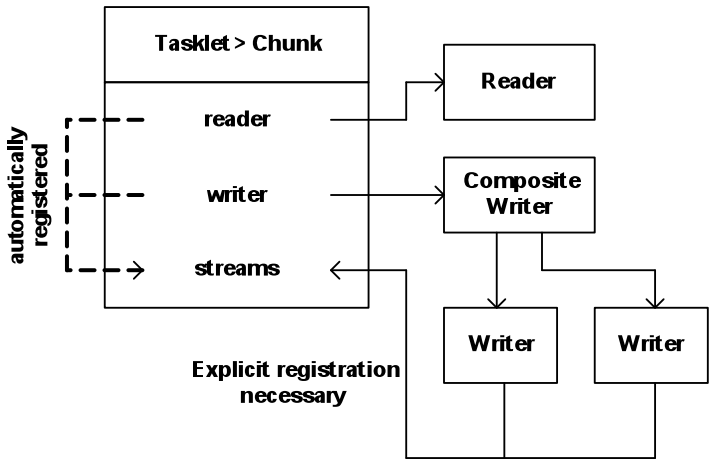

Figure 4.5 Registration of entities as stream. Spring Batch automatically registers readers, processors, and writers if they implement the ItemStream interface. Explicit registration is necessary if Spring Batch doesn't know about the streams to register, like the writers in the figure used through a composite writer.

Let's look at an example. If you use a composite item writer which isn't a stream and which internally uses stream writers, the step doesn't have references to those writers. In this case, we need to define explicitly the writers as streams for the step in order avoid problems on restarts when errors occur. Listing 4.10 describes how to configure this aspect using the `streams` child element of the `chunk` element.

Listing 4.10 Configuring streams in a chunk

```
<batch:job id="importProductsJob">
  (...)
  <batch:step id="readWrite">
    <batch:tasklet>
      <batch:chunk reader="productItemReader" writer="compositeWriter"/>
        <streams>
          <stream ref="productItemWriter1"/>                          #1
          <stream ref="productItemWriter2"/>                          #1
        </streams>
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="compositeWriter"
       class="org.springframework.batch.item.support.CompositeItemWriter">
  <property name="delegates">
    <list>
      <ref bean="productItemWriter1"/>                                #2
      <ref bean="productItemWriter2"/>                                #2
    </list>
  </property>
</bean>
```

#1 References writers as stream
#2 References writers in composite writer

Cueballs in code and text

In listing 4.10, we must register as streams #1 the item writers #2 involved in the composite item writer for the step. We do this using `streams` element as a child of the `chunk` element. The `streams` element then defines one or more `stream` elements, in our example, two `stream` elements #1.

In this section, we described how to configure a batch job with Spring Batch. We detailed the configuration of each related core object. We saw that transactions guarantee batch robustness and are involved at several levels in the configuration. Because this is an important issue, we gather all configuration aspects related to transactions in the next section.

4.2.5 Configuring transactions

Transactions in Spring Batch are an important topic: Transactions contribute to the robustness of batch processes and work in combination with chunk processing. You configure transactions at different levels because transactions involve several types of objects. In our use case, we validate a set of products during processing.

The first thing to configure is the Spring transaction manager because Spring Batch is based on the Spring framework and uses Spring's transaction support. Spring provides built-in transaction managers for common persistent technologies and frameworks. For JDBC, we use the `DataSourceTransactionManager` class as configured in the following snippet:

```
<bean id="transactionManager">
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="batchDataSource"/>
</bean>
```

Every Spring transaction manager needs to be configured using the factory provided by the framework to create connections and sessions. In the case of JDBC, the factory is an instance of the `DataSource` interface.

Once we configure the transaction manager, other configuration elements can refer to it from different level within the batch configuration, like from the tasklet level. The next snippet configures the transaction manager using the `transaction-manager` attribute:

```
<batch:job id="importProductsJob">
  (...)
  <batch:step id="readWrite">
    <batch:tasklet transaction-manager="transactionManager" (...)>       #A
      (...)
    </batch:tasklet>
  </batch:step>
</batch:job>
```

#A Specifies the transaction manager

We will see in section 4.4, "Configuring the job repository", that we must also use a transaction manager to configure entities to interact with a persistent job repository.

The Spring transaction manager

The Spring framework provides generic transactional support. Spring bases this support on the `PlatformTransactionManager` interface that provides a contract to handle transaction demarcations: beginning and ending transactions with a commit or rollback. Spring provides implementations for a large range of persistent technologies and frameworks like JDBC, JPA, and so on. For JDBC, the implementing class is `DataSourceTransactionManager` and for JPA it is `JpaTransactionManager`.

Spring builds on this interface to implement standard transactional behavior and allows configuring transactions with Spring beans using AOP or annotations. This generic support doesn't need to know which persistence technology is used and is completely independent of it.

Now that we have specified which Spring transaction manager to use, we can define how transactions will be handled during processing. As described in chapter 2, section 2.2.1, "Anatomy of the read-write step", Spring Batch uses chunk processing to handle items. That's why Spring Batch provides a commit interval tied to the chunk size. The `commit-interval` attribute configures this setting at the chunk level and ensures that Spring Batch executes a commit after processing a given number of items. The following example sets the commit interval to fifteen items:

```
<batch:tasklet>
  <batch:chunk (...) commit-interval="15"/>                          #A
</batch:tasklet>
```

#A Sets commit interval

Transactions have several attributes defining transactional behaviour, isolation and timeout. These attributes specify how transactions behave and can affect performance. Since Spring Batch is based on Spring transactional support, configuring these attributes is generic and applies to all persistent technologies supported by the Spring framework. Spring Batch provides the `transaction-attributes` element in the `tasklet` element for this purpose, as described in following snippet:

```
<batch:tasklet>
  <batch:chunk reader="productItemReader"
         writer="productItemReader"
         commit-interval="2"/>
    <batch:transaction-attributes isolation="DEFAULT"                    #1
                          propagation="REQUIRED"                         #1
                          timeout="30"/>                                 #1
  </batch:chunk>
</batch:tasklet>
```

#1 Configures transaction attributes

Cueballs in code and text

Transactional attributes are configured using the `transaction-attributes` element #1 and its attributes. The `isolation` attribute specifies the isolation level used for the database. The `READ_COMMITTED` level prevents dirty reads; `READ_UNCOMMITTED` that dirty reads, non-repeatable reads and phantom reads can occur; `REPEATABLE_READ` prevents dirty reads and non-repeatable reads; and `SERIALIZABLE` prevents dirty reads, non-repeatable reads and phantom reads. Choosing the `DEFAULT` value leaves the choice of the isolation level to the database.

The `propagation` attribute specifies the transactional behavior to use. Choosing `REQUIRED` implies that processing will use the current transaction if it exists and create a new one if it does not. The Spring class `TransactionDefinition` declares all valid values for these two attributes. Finally, the `timeout` attribute defines the timeout in seconds for the transaction. If the `timeout` attribute is absent, the default timeout of the underlying system is used.

Rollback and commit conventions in Spring and Java EE

Java defines two types of exceptions: checked and unchecked. A checked exception extends the `Exception` class, and a method must explicitly handle it in a try-catch block or declare it its signature's `throws` clause. An unchecked exception extends the `RuntimeException` class and a method does not need to catch or declare it. You commonly see checked exceptions used as business exceptions (recoverable) and unchecked exceptions as lower-level exceptions (unrecoverable by the business logic).

Chapter author name: Templier, Gregory

By default, in Java EE and Spring, commit and rollback are automatically triggered by exceptions. If Spring catches a checked exceptions, a commit is executed. If Spring catches an unchecked exceptions, a rollback is executed. You can configure Spring's transactional support to customize this behavior by setting which exceptions trigger commits and rollbacks.

The Spring framework follows the conventions outlined in the sidebar "Rollback and commit conventions in Spring and Java EE." In addition, Spring Batch lets you configure specific exception classes that do not trigger rollbacks when thrown. You configure this feature in a tasklet using the `no-rollback-exception-classes` element, as described in the following snippet:

```
<batch:tasklet>
  (...)
  <batch:no-rollback-exception-classes>
    <batch:include
     class="org.springframework.batch.item.validator.ValidationException"/>
  </batch:no-rollback-exception-classes>
</batch:tasklet>
```

In the previous snippet, Spring issues a commit even if the unchecked Spring Batch exception `ValidationException` is thrown during batch processing.

Spring Batch also provides parameters for special cases. The first case is readers built on a transactional resource like a JMS queue. For JMS, a step doesn't need to buffer data because JMS already provides this feature. For this type of resource, you need to specify the `reader-transactional-queue` attribute on the corresponding chunk, as shown in listing 4.11.

Listing 4.11 Configuring a transactional JMS item reader

```
<batch:tasklet>
  <batch:chunk reader="productItemReader"
               is-reader-transactional-queue="true" (...)/>              #A
</batch:tasklet>

<bean id="productItemReader"                                             #B
      class="org.springframework.batch.item.jms.JmsItemReader">
  <property name="itemType"
            value="com.manning.sbia.reader.jms.ProductBean"/>
  <property name="jmsTemplate" ref="jmsTemplate"/>
  <property name="receiveTimeout" value="350"/>
</bean>
```

#A Specifies reader as transactional
#B Configures JMS item reader

As described throughout this section, configuring batch processes can involve many concepts and objects. Spring Batch eases the configuration of core entities like job, step, tasklet and chunk. Spring Batch also lets you configure transaction behavior and define you own error handling. The next section covers configuring the Spring Batch job repository to store batch execution data.

4.3 Configuring the job repository

Along with the batch feature, the job repository is a key feature of the Spring Batch infrastructure because it provides information about batch processing. Chapter 3, section 3.2, "The Spring Batch infrastructure", describes the job repository: it saves information about the details of job executions. In this section, we focus on configuring the job repository in Spring Batch XML. The job repository is part of the more general topic concerning batch process monitoring. Chapter 12, "Batch monitoring", is dedicated to this topic.

4.3.1 Choosing a job repository

Spring Batch provides the `JobRepository` interface for the batch infrastructure and job repository to interact with each other. Chapter 3, section 3.2.1, "Launching jobs and storing job metadata", shows this interface. The interface provides all the methods required to interact with the repository. We don't present more here because we describe the job repository in detail in chapter 12, "Batch monitoring".

For the `JobRepository` interface, Spring Batch provides only one implementation: the `SimpleJobRepository` class. Spring Batch bases this class on a set of DAOs used for dedicated interactions and data management. Spring Batch provides two kinds of DAOs at this level:

- In-memory with no persistence.
- Persistent with metadata using JDBC.

You can use the in-memory DAO for tests but you should not use it in production environments. In fact, batch data is lost between job executions. You should prefer the persistent DAO when you want to have robust batch processing with checks on start-up. Because the persistent DAO uses a database, you need additional information in the job configuration. Database access configuration includes data source and transactional behavior.

Let's now describe how to configure these two repositories using Spring Batch XML.

4.3.2 Specifying job repository parameters

In this section, we configure the in-memory and persistent job repositories.

CONFIGURING AN IN-MEMORY JOB REPOSITORY

The first kind of job repository is the in-memory repository. Spring Batch provides the `MapJobRepositoryFactoryBean` class to make its configuration easier. The persistent repository uses a Spring bean for configuration and requires a transaction manager. Spring Batch provides the `ResourcelessTransactionManager` class as a NOOP implementation of the `PlatformTransactionManager` interface.

Listing 4.12 describes how to use the Spring Batch `MapJobRepositoryFactoryBean` and `ResourcelessTransactionManager` classes to configure an in-memory job repository.

Listing 4.12 Configuring an in-memory job repository

```
<bean id="jobRepository"                                                    #1
      class="org.springframework.batch.core.repository
                      [CA].support.MapJobRepositoryFactoryBean"
  <property name="transactionManager-ref" ref="transactionManager"/>
</bean>

<bean id="transactionManager"                                               #2
      class="org.springframework.batch.support
                      [CA].transaction.ResourcelessTransactionManager"/>

<batch:job id="importInvoicesJob"
           job-repository="jobRepository">                                  #3
  (...)
</batch:job>
```

#1 Configures job repository
#2 Configures transaction manager
#3 Links job repository to job

Code continuation character in code, cueballs in code and text

The in-memory job repository is first defined using the `MapJobRepositoryFactory` class #1 provided by Spring Batch. We specify the `transactionManager-ref` attribute to reference a configured transaction manager. This particular transaction manager is a `ResourcelessTransactionManager` #2 because the job repository is in-memory. Finally, we reference the job repository from the job using the `job-repository` attribute of the `job` element #3. The value of this attribute is the identifier of the job repository bean.

CONFIGURING A PERSISTENT JOB REPOSITORY

Configuring a persistent job repository isn't too complicated thanks to Spring Batch XML, which hides all the bean configuration details that would otherwise be required with Spring XML. Our configuration uses the `job-repository` element and specifies the attributes listed in table 4.8.

Table 4.8 job-repository attributes

Repository attribute name	Description
data-source	Bean identifier for the repository data source used to access the database.
transaction-manager	Bean identifier for the Spring transaction manager used to handle transactions for the job repository.

isolation-level-for-create	Isolation level used for SQL DDL.
max-varchar-length	Maximum length for VARCHAR columns in the database.
table-prefix	Table prefix used by the job repository in the database. This prefix allows identifying the tables used by the job repository from the tables used by the batch itself.
lob-handler	Handler for LOB type columns. Only use this attribute with Oracle or if Spring Batch does not detect the database type.

Listing 4.13 shows how to use the `job-repository` element and its attributes in order to configure a persistent job repository for a relational database.

Listing 4.13 Configuration of a persistent job repository

```
<bean id="dataSource"                                                    #1
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <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" lazy-init="true"                           #2
      class="org.springframework.jdbc.datasource
                      [CA].DataSourceTransactionManager">
   <property name="dataSource" ref="dataSource" />
</bean>

<batch:job-repository id="jobRepository"                                 #3
                data-source="dataSource"
                transaction-manager="transactionManager"
                isolation-level-for-create="SERIALIZABLE"
                table-prefix="BATCH_"
/>

<batch:job id="importInvoicesJob"
           job-repository="jobRepository">                               #4
  (...)
</batch:job>
```

#1 Configures data source with Apache DBCP
#2 Configures transaction manager
#3 Configures persistent job repository
#4 Links job repository in job

Code continuation character in code, cueballs in code and text

The configuration in listing 4.13 uses a data source named `dataSource` and the Apache DBCP library for connection pooling at #1. The listing also configures a transaction manager named `transactionManager` as a Spring `DataSourceTransactionManager` #2, which uses JDBC for database access.

The `job-repository` element can then configure the persistent job repository #3. This element references the data source and transaction manager previously configured. It also uses additional parameters to force the use of the `SERIALIZABLE` isolation level for DDL requests and to identify Spring Batch tables with the `BATCH_` prefix. We then reference this job repository from the job configuration at #4.

Spring Batch automatically adds transactions to entities for job repository management. If you don't use the `job-repository` element, you don't need to apply transactions manually.

The job repository is an important part of the Spring Batch infrastructure because it records batch processing information to track which jobs succeed and fail. Although Spring Batch provides an in-memory job repository, you should only use it for tests. Use the persistent job repository in production. In chapter 12, we monitor batch applications using the job repository.

In the next section, we describe advanced Spring Batch configurations that leverage the Spring expression language, modularize configurations with inheritance and use listeners.

4.4 Advanced configuration topics

In the previous sections, we described how to use Spring Batch XML to configure Spring Batch entities like jobs, steps, tasklets, chunks, and the job repository. In the next sections, we focus on more advanced configuration topics including optimizing configurations. Our goal is to simplify batch-processing configuration. Let's begin with the Spring Batch step scope feature.

4.4.1 Using Step scopes

Spring Batch provides a special bean scope class -- `StepScope` -- implemented as a custom Spring bean scope. The goal of the step scope is to link beans with steps within batches. This mechanism allows instantiation of beans configured in Spring only when steps begin, and to specify configuration and parameters for a step.

> **Spring custom scopes**
>
> Starting in version 2, Spring supports custom bean scopes. A bean scope specifies how to create instances of the class for a given bean definition. Spring provides scopes like `singleton`, `prototype`, `request`, and `session` but also allows custom scopes to be plugged-in. You must register a custom scope in the container using the `CustomScopeConfigurer` Spring class.

A custom scope implementation handles how an instance is served in a given Spring container. For example, with the `singleton` scope, the same instance is always provided by Spring. With the `prototype` scope, Spring always creates a new instance.

The `scope` attribute of the `bean` element defines the bean scope, for example:

```
<bean id="myBean" class="(...)" scope="prototype">
```

If you use Spring Batch XML, the step scope is automatically registered for the current Spring container and is usable without additional configuration. If you don't use Spring Batch XML, you need to define the step scope with its `StepScope` class, as described in the following snippet:

```
<bean class="org.springframework.batch.core.scope.StepScope"/>
```

Developers using custom Spring scopes may be surprised by this configuration. In fact, the `StepScope` class implements the Spring `BeanFactoryPostProcessor` interface, which automatically applies scopes to beans.

Using the step scope is particularly useful and convenient when jointly used with the Spring Expression Language in order to implement late binding of properties. Let's describe this feature next.

4.4.2 Leveraging the Spring Expression Language

In order to configure Spring Batch entities, the Spring Expression Language (SpEL) offers interesting possibilities. SpEL handles cases when values cannot be known during development and configuration because they depends on the runtime execution context.

Spring Expression Language (SpEL)

Spring version 3 introduced the Spring Expression Language (SpEL) to facilitate configuration. SpEL was created to provide a single language for use across the whole Spring portfolio, but it is not directly tied to Spring and can be used independently. SpEL supports a large set of expression types like literal, class, property access, collection, assignment, and method invocation.

The power of this language consists in its ability to use expressions to reference bean properties present in a particular context. You can view this feature as a more advanced and generic Spring `PropertyPlaceholderConfigurer`. The expression language can resolve expression not only in a properties file but also in beans managed within Spring application contexts.

Spring Batch leverages SpEL in order to access entities associated with jobs and steps, and to provide *late binding* in configurations. The typical use case of late binding is to use batch parameters specified at launch-time in the batch configuration. The values are unknown at

development-time when configuring batch processes. Spring evaluates these values at runtime during the batch process execution.

Table 4.9 describes all entities available from the step scope.

Table 4.9 Entities available from the step scope

Entity name	Description
jobParameters	Parameters specified for the job.
jobExecutionContext	Execution context of the current job.
stepExecutionContext	Execution context of the current step.

With this approach, it's now possible to specify property values filled in at launch-time as shown in listing 4.14.

Listing 4.14 Configuring batch parameters with SpEL

```
<bean id="decompressTasklet"
      class="com.manning.sbia.ch02.batch.DecompressTasklet"
      scope="step">                                                    #1
  <property name="inputResource"
            value="#{jobParameters['inputResource']}" />               #2
  <property name="targetDirectory"
            value="#{jobParameters['targetDirectory']}" />             #2
  <property name="targetFile"
            value="#{jobParameters['targetFile']}" />                  #2
</bean>
```

#1 Specifies step scope
#2 Specifies job parameters

Cueballs in code and text

We configure the `decompressTasklet` bean using the Spring Batch `step` scope #1. Specifying this scope allows us to use SpEL's late binding feature for job parameters within values of bean properties #2. The `jobParameters` object acts as map for a set of parameters and elements and is accessed using notation delimited by `#{` and `]` . We also use this format in the example with the objects `jobExecutionContext` and `stepExecutionContext`.

In the context of our case study, this mechanism makes it possible to specify at batch startup the file to use to import product data. We do not need to hard-code the file name in the batch configuration, as described in figure 4..

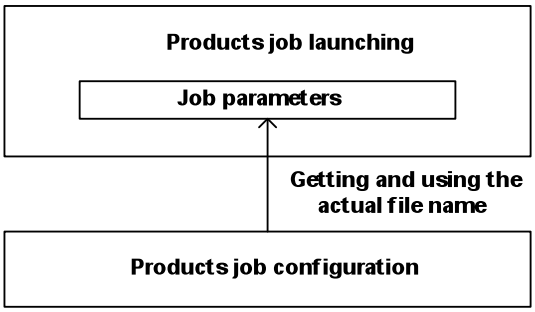

Figure 4.6 Using the file name to import from the job configuration in the use case

Like most frameworks, Spring Batch provides the ability to specify listeners during processing. Spring Batch supports this feature at the job and step levels. The next section describes how to implement and configure such listeners.

4.4.3 Listeners

Spring Batch provides the ability to specify and use listeners at different levels within batches. This feature is particularly useful and powerful because Spring Batch can notify each level of batch processing, where we can plug-in additional processing. For example, in our case study, we can add a listener that Spring Batch calls when a batch fails or to record which products Spring Batch skips because of errors, as described in figure 4.7.

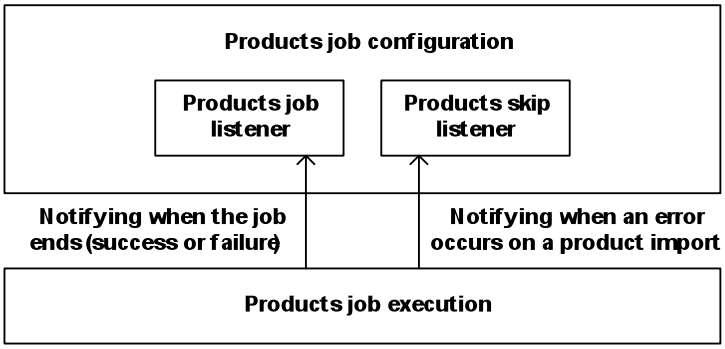

Figure 4.7 Notifications of errors during job execution

We will provide concrete examples of this feature in chapter 9, "Making batch processes more robust". Table 4.10 describes the listener types provided by Spring Batch.

Table 4.10 Listener types

Listener type	Description
Job listener	Listens to processing at the job level.
Step listeners	Listens to processing at the step level.
Item listeners	Listens to item repeat or retry.

Next, let's describe in more details how to implement and configure these listeners and when to use them.

JOB LISTENERS

The first kind of listener intercepts job execution and supports the before and after job execution events. These events add processing before a job and after a job according to the completion type of a batch process. They are particularly useful to notify batch failures to external systems. Such listeners are implementations of the `JobExecutionListener` interface:

```
public interface JobExecutionListener {
  void beforeJob(JobExecution jobExecution);
  void afterJob(JobExecution jobExecution);
}
```

You configure a listener in a job configuration with the `listeners` element, as a child of the `job` element. The `listeners` element can configure several listeners by referencing Spring beans. The following snippet describes how to register the `ImportProductsJobListener` class as a listener for the `importProductsJob` job.

```
<batch:job id="importProductsJob">
  <batch:listeners>
    <batch:listener ref="importProductsJobListener"/>
  </batch:listeners>
</batch:job>

<bean id="importProductsJobListener" class="ImportProductsJobListener"/>
```

The `ImportProductsJobListener` class will receive notifications when Spring Batch starts and stops a job, whether the job ends successfully or in failure. A `JobExecution` instance provides information regarding job execution status with `BatchStatus` constants. Listing 4.15 shows the `ImportProductsJobListener` class.

Listing 4.15 Listening to job execution with a listener

```
public class ImportProductsJobListener
                        implements JobExecutionListener {                #1
  public void beforeJob(JobExecution jobExecution) {                    #2
    // Called when job starts
  }

  public void afterJob(JobExecution jobExecution) {                     #3
    if (jobExecution.getStatus()==BatchStatus.COMPLETED) {
      // Called when job ends successfully
    } else if (jobExecution.getStatus()==BatchStatus.FAILED) {
```

```
        // Called when job ends in failure
      }
    }
}
```

#1 Implements listener interface
#2 Called before a job starts
#3 Called after a job ends

Cueballs in code and text

The listener class must implement the `JobExecutionListener` interface #1 and define the `beforeJob` method #2 that Spring Batch calls before the job starts and the `afterJob` method #3 called after the job ends.

Spring Batch can also use annotated classes as listeners. In this case, we do not implement the `JobExecutionListener` interface. In order to specify which methods do the listening, we use the `BeforeJob` and `AfterJob` annotations. Listing 4.16 shows the annotated class (`AnnotedImportProductsJobListener`) corresponding to the standard listener class (`ImportProductsJobListener`).

Listing 4.16 Listening to job execution with annotations

```
public class AnnotedImportProductsJobListener {
  @BeforeJob                                                    #1
  public void executeBeforeJob(JobExecution jobExecution) {
    //Notifying when job starts
  }

  @AfterJob                                                     #2
  public void executeAfterJob(JobExecution jobExecution) {
    if (jobExecution.getStatus()==BatchStatus.COMPLETED) {
      //Notifying when job successfully ends
    } else if (jobExecution.getStatus()==BatchStatus.FAILED) {
      //Notifying when job ends with failure
    }
  }
}
```

#1 Called before a job starts
#2 Called after a job ends

Cueballs in code and text

This listener class is a POJO and defines which methods to execute before the job starts with the `BeforeJob` annotation #1 and after the job ends with the `AfterJob` annotation #2.

Next, we describe listeners at the job level.

STEP LISTENERS

Steps also have a matching set of listeners to track processing during step execution. You use step listeners to track item processing and to define error-handling logic. All step listeners extend the `StepListener` marker interface. `StepListener` acts as a parent to all step domain listeners. Table 4.11 describes the step listeners provided by Spring Batch.

Table 4.11 Step Listeners provided by Spring Batch

Listener interface	Description
ChunkListener	Called before and after chunk execution.
ItemProcessListener	Called before and after an ItemProcessor gets an item, and when that processor throws an exception.
ItemReadListener	Called before and after an item is read and when an exception occurs reading an item.
ItemWriteListener	Called before and after an item is written and when an exception occurs writing an item.
SkipListener	Called when a skip occurs while reading, processing, or writing an item.
StepExecutionListener	Called before and after a step.

The `StepExecutionListener` and `ChunkListener` interfaces relate to lifecycle. They are respectively associated to the step and chunk and provide methods for before and after events. The `StepExecutionListener` interface uses a `StepExecution` parameter for each listener method to access current step execution data. Furthermore, the `afterStep` method triggered after step completion must return the status of the current step with an `ExistStatus` instance. The following snippet describes the `StepExecutionListener` interface:

```
public interface StepExecutionListener extends StepListener {
  void beforeStep(StepExecution stepExecution);
  ExitStatus afterStep(StepExecution stepExecution);
}
```

The `ChunkListener` interface provides methods called before and after the current chunk. These methods have no parameter and return void, as described in the following snippet:

```
public interface ChunkListener extends StepListener {
  void beforeChunk();
  void afterChunk();
}
```

The item listener interfaces (read, process, and write) listed in table 4.11 each deal with a single item and support Java 5 generics to specify item types. Each interface provides three methods triggered before, after, and on error. Each interface accepts as a parameter a single item from a list of handled entities for before and after methods. For error-handling methods, Spring Batch passes the thrown exception as a parameter.

For item processing, these methods are `beforeProcess`, `afterProcess` and `onProcessError`. The following snippet lists the `ItemProcessingListener` interface:

```
public interface ItemProcessListener<T, S> extends StepListener {
  void beforeProcess(T item);
  void afterProcess(T item, S result);
  void onProcessError(T item, Exception e);
}
```

In the case of the `ItemReadListener` interface, these methods are `beforeRead`, `afterRead` and `onReadError`, as shown in the following snippet:

```
public interface ItemReadListener<T> extends StepListener {
  void beforeRead();
  void afterRead(T item);
  void onReadError(Exception ex);
}
```

In the case of the `ItemWriteListener` interface, these methods are `beforeWrite`, `afterWrite` and `onWriteError`, as shown in the following snippet:

```
public interface ItemWriteListener<S> extends StepListener {
  void beforeWrite(List<? extends S> items);
  void afterWrite(List<? extends S> items);
  void onWriteError(Exception exception, List<? extends S> items);
}
```

The last kind of interface in table 4.11 listens for skip events. Spring Batch calls the `SkipListener` interface when processing skips an item. The interface provides three methods corresponding to when the skip occurs: `onSkipInRead` during reading, `onSkipInProcess` during processing and `onSkipInWrite` during writing. The following snippet lists the `SkipListener` interface:

```
public interface SkipListener<T,S> extends StepListener {
  void onSkipInRead(Throwable t);
  void onSkipInProcess(T item, Throwable t);
  void onSkipInWrite(S item, Throwable t);
}
```

You can also define listeners for all these events as annotated POJOs. Spring Batch leaves the choice up to you. Spring Batch provides annotations corresponding to each method defined by the interfaces in table 4.11. For example, in the case of the `ExecutionListener` interface, the `BeforeStep` annotation corresponds to the `beforeStep` method and the `AfterStep` annotation to the `afterStep` method. Configuring listeners using annotations follows the same rules as the interface-based configurations described in the next section. Listing 4.17 describes how to implement an annotation-based listener for step execution.

Listing 4.17 Implementing an annotation-based step listener

```
public class ImportProductsExecutionListener {
  @BeforeStep                                                    #A
  public void handlingBeforeStep(StepExecution stepExecution) {
    (...)
```

```
    }

    @AfterStep                                                          #B
    public ExitStatus afterStep(StepExecution stepExecution) {
      (...)
      return ExitStatus.FINISHED;
    }
  }
```

#A Called before step execution
#B Called after step execution

As for a job, configuring a step listener is done using a `listeners` element as a child of the `tasklet` element. You can configure all kinds of step listeners at this level in the same manner. For example:

```
<batch:job id="importProductsJob">
  <batch:step id="decompress" next="readWrite">
    <batch:tasklet ref="decompressTasklet">
      <batch:listeners>                                                 #A
        <batch:listener ref="stepListener"/>                            #B
      </batch:listeners>
    </batch:tasklet>
  </batch:step>
</ batch:job>
```

#A Specifies listeners for step
#B Registers a listener

Note that you can also specify several listeners at the same time.

Next, we look at another type of listener that deals with batch robustness and provides notification when repeats and retries occur.

REPEAT AND RETRY LISTENERS

The last kind of listeners provided by Spring Batch deals with repeats and retries for items. These listeners support the methods listed in table 4.12 and allow processing during repeat or retry.

Table 4.12 Methods for retry and repeat listeners

Method	Description
after (repeat listener only)	Called after each try or repeat.
before (repeat listener only)	Called before each try or repeat.
close	Called after the last try or repeat on an item, whether successful or not in the case of a retry.
onError	Called after every unsuccessful attempt at a retry or every repeat failure with a thrown exception.
open	Called before the first try or repeat on an item.

The following snippet lists the `RepeatListener` interface called when repeating an item:

```
public interface RepeatListener {
  void before(RepeatContext context);
```

```
  void after(RepeatContext context, RepeatStatus result);
  void open(RepeatContext context);
  void onError(RepeatContext context, Throwable e);
  void close(RepeatContext context);
}
```

The following snippet lists the content of the `RetryListener` interface called when retrying an item:

```
public interface RetryListener {
  <T> void open(RetryContext context, RetryCallback<T> callback);
  <T> void onError(RetryContext context,
            RetryCallback<T> callback, Throwable e);
  <T> void close(RetryContext context,
            RetryCallback<T> callback, Throwable e);
}
```

Such listeners must be configured like step listeners using the `listeners` child element of the `tasklet` element, as described at the end of the previous section, "Step listeners".

The next and last feature in our advanced configuration discussion is the Spring Batch inheritance feature used to modularize entity configurations.

4.4.4 Configuration inheritance

As emphasized in section 4.2.2, "Configuring jobs", Spring Batch XML provides facilities to ease configuration of batch jobs. While this XML vocabulary improves Spring Batch configuration, duplication can remain and that's why the vocabulary supports configuration inheritance like Spring XML.

This feature is particularly useful when configuring similar jobs and steps. Rather than duplicating XML fragments, Spring Batch allows you to define abstract entities to modularize configuration data. In our case study, we define several jobs with their own steps. As a best practice, we want to apply default values of our batch processes. To implement this, we define abstract jobs and steps. The default configuration parameters then apply to all child jobs and steps. Modifying one parameter affects all children automatically, as described in figure 4.8.

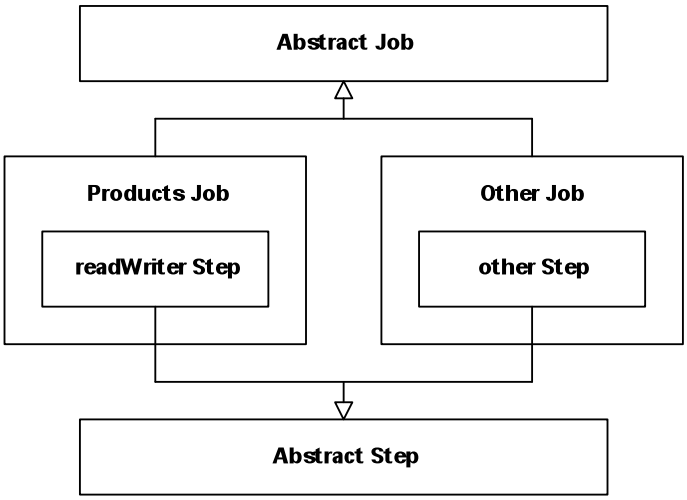

Figure 4.8 Using configuration inheritance

Configuration inheritance in Spring

Configuration inheritance is a built-in feature of the default Spring XML vocabulary since version 2.0. Its aim is to allow modularizing configuration and preventing configuration duplication. It targets the same issue that Spring XML addresses: making configuration less complex.

Spring allows defining abstract bean definitions that don't correspond to instances. Such bean definitions are useful to modularize common bean properties and avoid duplication within Spring configurations. Spring provides the `abstract` and `parent` attributes on the `bean` element. A bean with the attribute `abstract` set to `true` defines a virtual bean and using the `parent` attribute allows linking two beans in a parent-child relationship.

Inheriting from a parent bean means the child bean can use all attributes and properties from the parent bean. You can also use overriding from a child bean.

The following fragment describes how to use the `abstract` and `parent` attributes:

```
<bean id="parentBean" abstract="true">
  <property name="propertyOne" value="(...)"/>
</bean>
<bean id="childBean" parent="parentBean">
  <property name="propertyOne" value="(...)"/>
```

```
  <property name="propertyTwo" value="(...)"/>
</bean>
```

You can use Spring Batch configuration inheritance at both the job and step levels. Spring Batch also supports configuration merging. Table 4.13 describes the `abstract` and `parent` configuration inheritance attributes.

Table 4.13 Configuration inheritance attributes

Attribute	Description
abstract	When `true`, specifies that the job or step element isn't a concrete element but an abstract one only used for configuration. Abstract configuration entities are not instantiated.
parent	The parent element used to configure a given element. The child element will have all properties of its parent and can override them.

As we've seen, configuration inheritance in Spring Batch is inspired by Spring and is based on the `abstract` and `parent` attributes. Configuration inheritance allows you to define abstract batch entities that aren't instantiated but are only present within the configuration in order to modularize other configuration elements.

Let's look at an example that uses steps. A common use case is to define a parent step that modularizes common and default step parameters. Listing 4.18 shows how to use configuration inheritance to configure `step` elements.

Listing 4.18 Using configuration inheritance for steps

```
<step id="parentStep">                                          #1
  <tasklet allow-start-if-complete="true">
    <chunk commit-interval="100"/>
  </tasklet>
</step>

<step id="productStep" parent="parentStep">                     #2
  <tasklet start-limit="5">
    <chunk reader="productItemReader"
           writer="productItemWriter"
           processor="productItemProcessor"
           commit-interval="15"/>
  </tasklet>
</step>
```

#1 Defines the parent step
#2 Defines the child step

Cueballs in code and text

The parent step named `parentStep` #1 includes a `tasklet` element, which also includes a `chunk` element. Each element includes several attributes for its configuration, `allow-start-if-complete` for `tasklet` and `commit-interval` for `chunk`. We name the second step `productStep` at #2 and reference the previous step as its parent. The `productStep` step has the same element hierarchy as its parent, which includes all elements and attributes. In some cases, a parent defines attributes but children do not, so Spring Batch adds the attributes to the child configurations. In other cases, attributes are present in both parent and child steps and the values of child elements override the values of parent elements.

An interesting feature of Spring Batch related to configuration inheritance is the ability to merge lists. By default, Spring Batch does not enable this feature; lists in the child element overrides lists in the parent element. You can change this behavior by setting the `merge` attribute to `true`. Listing 4.19 combines the list of listeners present in a parent job with the list in the child job.

Listing 4.19 Merging two list with configuration inheritance

```
<job id="parentJob" abstract="true">
  <listeners>
    <listener ref="globalListener"/>
  <listeners>
</job>

<job id="importProductsJob" parent="parentJob">
  (...)
  <listeners merge="true">                                          #1
    <listener ref="specificListener"/>
  <listeners>
</job>
```

#1 Enables merging

Cueballs in code and text

We specify the merge attribute at #1 for the `listeners` element in the configuration of the child job. In this case, Spring Batch merges the listeners of the parent and child jobs and the resulting list contains listeners named `globalListener` and `specificListener`.

With configuration inheritance in Spring Batch, we close our advanced configuration section. These features allow easier and more concise configurations of Spring Batch jobs.

4.5 Summary

Spring Batch provides facilities to ease configuration of batch processes. Spring Batch bases these facilities on Spring Batch XML – an XML vocabulary dedicated to the batch domain – which leverages Spring XML. This vocabulary can configure all batch entities described in chapter 3, "Spring Batch concepts" like jobs, tasklets, and chunks. Spring Batch supports

entity hierarchies in its XML configuration and closely interacts with Spring XML to use Spring beans.

XML configuration of batch processes is built-in and allows plug-in strategies to handle errors and transactions. All these features contribute to making batches more robust by providing support for commit intervals, skip handling, and retry, among others. We focus on batch robustness in chapter 9, "Making batch processes more robust".

Spring Batch provides support to configure access to the job repository used to store job execution data. This feature relates to the more general topic of batch monitoring and chapter 12, "Batch monitoring" addresses it in detail.

Spring Batch XML also includes advanced features making configuration flexible and convenient. Features include the step scope and the ability to interact with the batch execution context using SpEL late binding of parameter values at runtime. You can implement job, step, and chunk listeners with interfaces or by using annotations on POJOs. Finally, we saw that Spring Batch provides modularization at the configuration level with the ability to use inheritance to eliminate configuration duplication.

After having described batch configuration in this chapter, the next chapter will focus on execution and describe the different ways to launch batch processes in real systems.

5 Running batch jobs

This chapter covers

- Running jobs from the command line
- Scheduling jobs
- Embedding Spring Batch in a web application
- Stopping job executions

If you've been reading this book from page one, you know the basics of Spring Batch, you know about jobs, steps, and chunks. And you must be eager to get your jobs up and running. Launching a Spring Batch job is easy because the framework provides a Java-based API for this purpose. However, how you call this API is another matter and depends on your system. Perhaps you'll use something simple like the `cron` scheduler to launch a Java program. Alternatively, you may want to trigger your jobs manually from a web application. Either way, we have you covered as this chapter covers both scenarios.

This chapter does cover many launching scenarios, so you may not want to read it from beginning to end, especially if you're in a hurry. You may read this chapter "à la carte": think about your scenario and read only what you need. Nevertheless, you should read section 5.1 covering the concepts of launching Spring Batch jobs and especially sub-section 5.1.3 that guides you through the chapter to pick up the launching solution that best suits your needs.

5.1 Launching concepts

It's time to launch your Spring Batch job! You're about to see that launching a Spring Batch job is quite simple thanks to the Spring Batch launcher API. But how you end up launching your batch jobs depends on many parameters, so we provide you with basic concepts and some guidelines. By the end of this section, you'll know where to look in this chapter to set up a launching environment for your jobs.

5.1.1 Introducing the Spring Batch launch API

The heart of the Spring Batch launcher API is the `JobLauncher` interface. Here is a shortened version of this interface (we removed the exceptions for brevity):

```
public interface JobLauncher {

  public JobExecution run(Job job, JobParameters jobParameters) throws (…);

}
```

The `JobLauncher` itself and the `Job` you pass to the `run` method are Spring beans. The call site typically builds the `JobParameters` argument on the fly. The following snippet shows how to use the job launcher to start a job execution with two parameters:

```
ApplicationContext context = (...)
JobLauncher jobLauncher = context.getBean(JobLauncher.class);
Job job = context.getBean(Job.class);

jobLauncher.run(
  job,
  new JobParametersBuilder()
    .addString("inputFile", "file:./products.txt")
    .addDate("date", new Date())
    .toJobParameters()
);
```

Note the use a `JobParametersBuilder` to create a `JobParameters` instance. The `JobParametersBuilder` class provides a fluent-style API to construct job parameters. A job parameter consists of a key and a value. Spring Batch supports four types for job parameters: string, long, double and date.

> Job parameters and job instance Remember that job parameters define the *instance* of a job and that a job instance can have one or more corresponding *executions*. You can view an execution as an attempt to run a batch process. If the notions of job, job instance, and job execution aren't clear to you, please refer to chapter 3, which covers these concepts.

Spring Batch provides an implementation of `JobLauncher`, whose only mandatory dependency is a job repository. The following snippet shows how to declare a job launcher with a persistent job repository:

```
<batch:job-repository id="jobRepository" data-source="dataSource" />

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

That's it; you know everything about the Spring Batch launcher API! Ok, not everything, we did not describe the `JobExecution` object returned by the `run` method. As you can guess, this object represents the execution coming out of the run method. The `JobExecution` interface provides the API to query the status of an execution: if it's running, if it has finished or if it has failed. Because batch processes are often quite long to execute, Spring Batch offers both synchronous and asynchronous ways to launch jobs.

5.1.2 Synchronous vs. asynchronous launches

By default, the `JobLauncher run` method is synchronous: the caller waits until the job execution ends (successfully or not.) Figure 5.1 illustrates a synchronous launch.

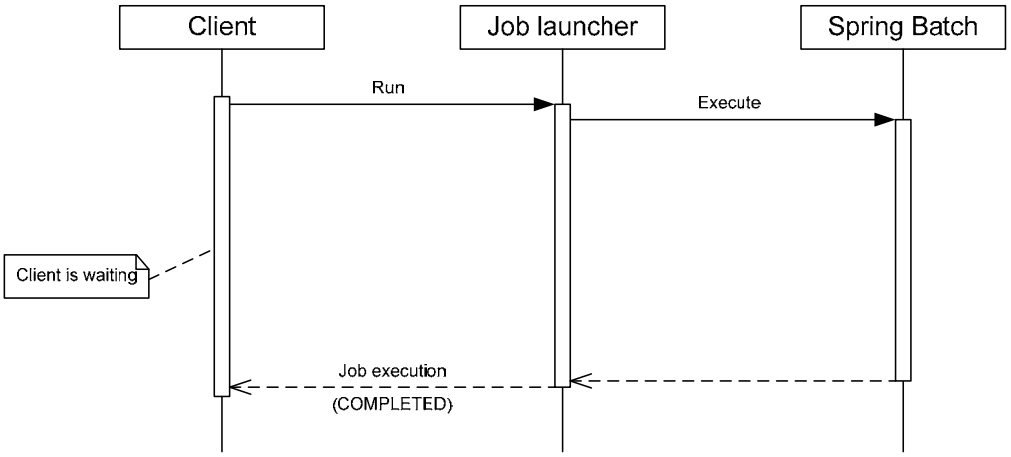

Figure 5.2 The job launcher is synchronous by default. The client waits until the job execution ends (successfully or not) before the job launcher returns the corresponding job execution object. Synchronous execution can be problematic, for example, when the client is a controller from a web application.

Synchronous launching is good in some cases: if you write a Java main program that a system scheduler like cron launches periodically, you want to exit the program only when the execution ends. But imagine that an HTTP request triggers the launching of a job. Writing a web controller that uses the job launcher to start Spring Batch jobs on HTTP requests is a handy way to integrate with external triggering systems. What happens if the launch is synchronous? The batch process will execute in the calling thread, monopolizing web container resources. Submit many batch processes in this way and they will use up all the threads of the web container, making it unable to process any other requests.

The solution is to make the job launcher asynchronous. Figure 5.2 shows how launching behaves when the job launcher is asynchronous.

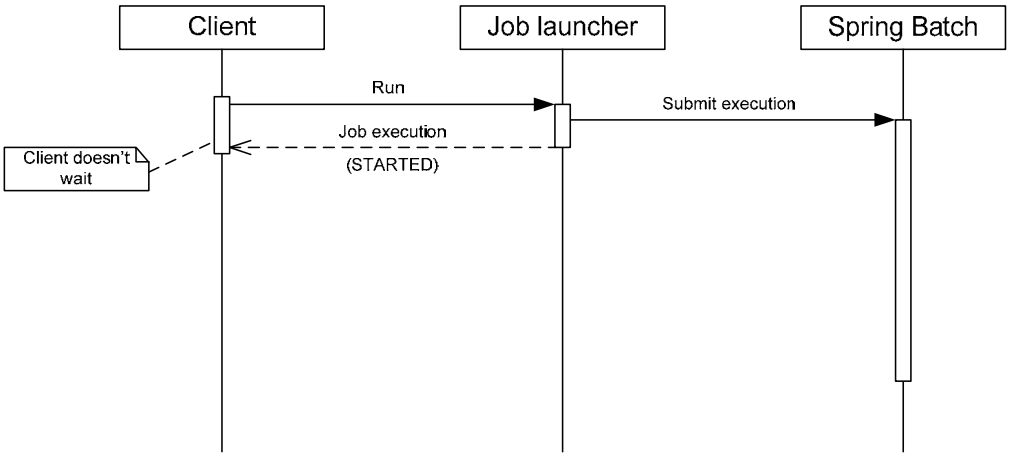

Figure 5.2 The job launcher can use a task executor to launch job executions asynchronously. The task executor

handles the threading strategy and the client has immediate access to the job execution object.

To make the job launcher asynchronous, just provide it with an appropriate `TaskExecutor`, as shown in the following snippet:

```
<task:executor id="executor" pool-size="10" />

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

In this example, we use a task executor with a thread pool of size 10. The executor will reuse threads from its pool to launch job executions asynchronously. Note the use of the `executor` XML element from the `task` namespace. This is a shortcut provided in Spring 3.0, but you can also define a task executor like any other bean (by using an implementation like `ThreadPoolTaskExecutor`).

It is now time to guide you through the launching solutions that this chapter covers.

5.1.3 Overview of launching solutions

This chapter covers many solutions to launch your Spring Batch jobs and there's little chance you'll use them all in one project. Many factors can lead you to choose a specific launching solution: launching frequency, number of jobs to launch, nature of the triggering event, type of job, duration of the job execution, and so on. Let's explore some cases and present some guidelines.

LAUNCHING FROM THE COMMAND LINE

A straightforward way to launch a Spring Batch job is to use the command line, which spawns a new Java Virtual Machine process for the execution, as figure 5.3 illustrates.

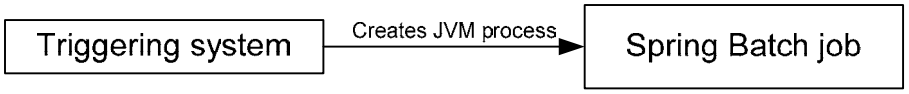

Figure 5.3 You can launch a Spring Batch job as a plain Java Virtual Machine process. The triggering system can be a scheduler or a human operator. This solution is simple but implies initializing the batch environment for each run.

The triggering event can be a system scheduler like cron or even a human operator that knows when to launch the job. If you're interested in launching jobs this way, read section 5.2 on command line launching. You'll see that you can write your own Java launcher program but also that Spring Batch provides a generic command line launcher that you can reuse. If you choose the scheduler option, you should also look at section 5.3.1, which covers cron.

EMBEDDING SPRING BATCH AND A SCHEDULER IN A CONTAINER

Spawning a JVM process for each execution can be costly, especially if it opens new connections to a database or creates object-relational mapping contexts. Such initializations are resource-intensive and you probably don't want the associated costs if your jobs run every minute. Another option is to embed Spring Batch into a container such that your Spring Batch environment is ready to run at anytime and

there is no need to set up Spring Batch for each job execution. You can also choose to embed a Java-based scheduler to start your jobs. Figure 5.4 illustrates this solution.

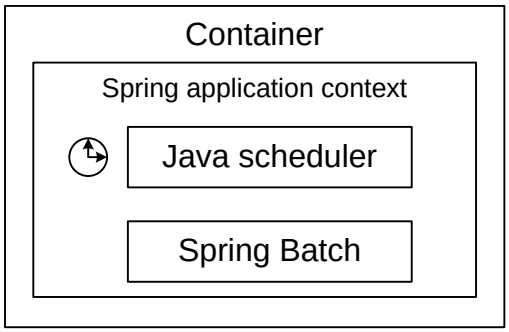

Figure 5.4 You can embed Spring Batch in a container, along with a Java scheduler. A web container is a good candidate, as Spring integrates easily in web applications.

A web container is a popular way to embed a Spring Batch environment. Remember that Spring Batch runs everywhere the Spring Framework runs. If you want to learn how to deploy Spring Batch in a web application, read section 5.4.1. Java-based schedulers also run in Spring, so read section 5.3.2 to learn about Spring scheduling support and section 5.3.3 to learn about the Quartz scheduler.

EMBEDDING SPRING BATCH AND TRIGGERING JOBS BY AN EXTERNAL EVENT

You can also have a mix of solutions: use cron because it is a popular solution in your company and embed Spring Batch in a web application because it avoids costly recurring initializations. The challenge here is to give cron access to the Spring Batch environment. Figure 5.5 illustrates this deployment.

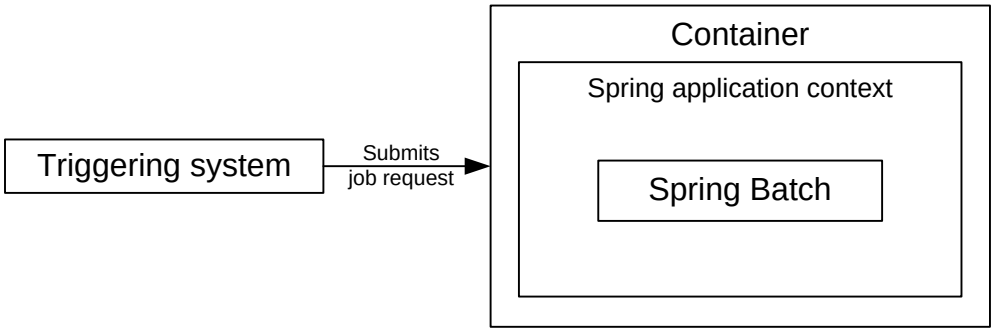

Figure 5.5 An external system submits a job request to the container where the Spring Batch environment is deployed. An example would be a cron scheduler submitting an HTTP request to a web controller. The web controller would then use the job launcher API to start the job execution.

To see how the Spring Batch job launcher works with HTTP, please see section 5.4.2, which covers Spring MVC.

The list of launching solutions this chapter covers is by no means exhaustive. The Spring Batch launcher API is simple to use, so you can imagine building other types of solutions, for example: event-driven with JMS or remote with JMX. And don't forget to read section 5.5 on stopping job executions when you're done launching all these jobs!

5.2 Launching from the command line

Using the command line is perhaps the most common way to launch a batch process. Triggering the process can be done manually (by a human) but most of the time you'll be using a system scheduler (cron on UNIX systems for example) to trigger the launch. Why? Because batch processes are launched at specific times (at night, on the last Sunday of the month, and so on). We cover schedulers later; in this section, we focus on how to launch a batch process through the command line.

Because Spring Batch is a Java-based framework, launching a Spring Batch process means spawning a new Java Virtual Machine process for a class and using the Spring Batch launcher API in that class' `main` method.

The Spring Batch launcher API is straightforward; the `JobLauncher` has one method – `run` – that takes a `Job` and `JobParameters` argument, so writing a main method to launch a job is quick and easy. We look at this option in the next section. But writing a dedicated Java program for each job doesn't scale as the number of jobs increases. That's why Spring Batch provides a generic Java class to launch jobs. You'll probably use this second option to launch Spring Batch jobs from the command line. For now, let's keep things simple and write our own Java program to launch a job.

5.2.1 Writing a custom command line launcher

Our custom command line launcher is a Java class that we launch from the command line. This custom launcher is just a plain Java class with a `main` method. We'll see how to launch it from the command line, which will require packaging and knowledge of the `java` command (as for any Java program.) We'll see that this solution works but that it has some drawbacks that the Spring Batch generic launcher addresses.

IMPLEMENTING THE CUSTOM LAUNCHER

Listing 5.1 shows a simple Java program that performs the necessary operations to launch a Spring Batch job: bootstrapping the application context, looking up the job and the job launcher, and launching the job with parameters.

Listing 5.1 A simple Java program to launch a Spring Batch job

```
package com.manning.sbia.ch05;

import java.util.Date;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LaunchImportProductsJob {

  public static void main(String[] args) throws Exception {
    ApplicationContext context =                              #1
```

```
      new ClassPathXmlApplicationContext(                    #1
    "import-products-job.xml"                              #1
  );                                                  #1
  JobLauncher jobLauncher = context.getBean(                    #2
    JobLauncher.class                                 #2
  );                                                  #2
  Job job = context.getBean(Job.class);                         #2

  jobLauncher.run(                                    #3
    job,                                              #3
    new JobParametersBuilder()                              #3
      .addString("inputFile", "file:./products.txt")          #3
      .addDate("date", new Date())                          #3
      .toJobParameters()                              #3
  );                                                  #3
 }

}
```

#1 Bootstraps Spring application context
#2 Looks up job and job launcher beans
#3 Launches job

Everything that Spring Batch does takes place in an instance of a Spring container, so the launching program starts by bootstrapping a Spring application context at #1. The application context provides us with the `JobLauncher` and the target `Job` at #2; we launch the job at #3. Note that the provided job parameters uniquely identify the job instance and we'll see later how this is a shortcoming of this launcher program.

TRIGGERING THE LAUNCHER FROM THE COMMAND LINE

Now that we have a Java class with a `main` method, how do we launch it from the command line? The first step is to package everything in a JAR file: all application classes – the launcher class itself but also custom item readers, writers, processors, data access objects, and so on – as well as resources, like the import-products-job.xml file. We can do all of this with a tool like Maven to end up with an `import-products.jar` file.

The second step is to create a neat layout on the file system such that the JVM can locate all the necessary Java classes and resources on the classpath. What should be on the classpath? The `import-products.jar` file of course, but also all the dependencies of our batch: Spring Batch, the corresponding dependencies from the Spring Framework and any other dependencies for your application (XML, persistence, and database connection pooling libraries for example.) The launch layout can look like the following (we elide dependencies for brevity):

```
batch-launching-dir/
  lib/
    import-products.jar
    spring-batch-core.jar
    spring-batch-infrastructure.jar
    spring-core.jar
    ...
```

The following command launches the batch process:

```
java -classpath "./lib/*" com.manning.sbia.ch05.LaunchImportProductsJob
```

Nice! This launch solution is simple; it works for a nightly batch job like the import new products job of our online store application. But, batch processes launched from the command line can get complex and have strict requirements; for example: accepting parameters and returning exit codes that must reflect the result of the batch process. Let's analyze of our custom launching program and see its limitations.

LIMITATIONS OF THE CUSTOM COMMAND LINE LAUNCHER

To see the shortcomings of the custom command line launcher, let's consider the following scenarios:

- The input file changes for each batch instance – meaning the class should not hard-code the location of the input file as it is now[1]. To remedy this, the launcher must be able to use parameters from the command line.
- The triggering system (human or scheduler) needs an exit code to know if the batch succeeded or failed – the custom launcher we wrote has poor exception handling. It re-throws any exceptions thrown by the `JobLauncher`.

We can address both scenarios easily: the `main` method's argument is a `String` array that includes parameters to the class from the `java` executable invocation. We can surround the job launch with a try-catch statement and use the `System.exit(int)` method to exit the JVM with an appropriate code.

Now, imagine doing this for each job, each with their own set of requirements, repeatedly. Yikes! You would think twice before adding a new job to your batch system. Fortunately, Spring Batch includes a generic command line launcher that handles these requirements.

Using the generic command line launcher

Spring Batch provides the `CommandLineJobRunner` class to launch jobs. This launcher should remove any need for custom command line launchers because of its flexibility. Table 5.1 lists the `CommandLineJobRunner` settings.

Table 5.1 Settings for the generic command line launcher

Setting	Description
Spring configuration file	The file used to start the Spring application context. The file configures the Spring Batch infrastructure, jobs, and necessary components (data source, readers, writers, and so forth.)
Job	The name of the job to execute (refers to a Spring bean name).
Job parameters	The job parameters to pass to the job launcher.
Exit code mapping	A strategy to map the executed job exit status to a system exit status.

[1] The location of the input file is one thing, but we use the current date as one of the parameters, which helps to identify uniquely the job instance. Unfortunately, if the execution fails, we will never be able to re-start it! For the next attempt, the date will be different, meaning this is a different job instance (remember chapter 3, which explained that a job instance is defined by a job and job parameters.) We lose all the restart features that Spring Batch provides. Therefore, if we want to be able to restart the job, the date also needs to be a job parameter coming from the command line.

Before we see how to use the `CommandLineJobRunner`, don't forget that it's a plain Java class with a `main` method, so you need to set up your launching layout on the file system properly, just as we did for the custom command line launcher.

To cover the different uses of the `CommandLineJobRunner`, imagine that we have an `importProductsJob` job defined in an `import-products-job.xml` file located at the root of the classpath.

LAUNCHING WITHOUT JOB PARAMETERS

The simplest use of the `CommandLineJobRunner` is to launch a job that doesn't require any parameters. We launch the `importProductsJob` job this way:

```
java -classpath "./lib/*"
[CA] org.springframework.batch.core.launch.support.CommandLineJobRunner
[CA] import-products-job.xml importProductsJob
```

The first parameter to the `CommandLineJobRunner` is the location of the Spring configuration file and the second parameter is the name of the `Job` (the name of the corresponding Spring bean.)

> Note The `CommandLineJobRunner` uses a `ClassPathXmlApplicationContext`, which means it locates the configuration file on the classpath by default. You can use Spring's resource abstraction prefixes to override this default, for example `file:./import-products-job.xml` if your configuration file is on the file system, in the current directory.

There's little chance that your jobs won't need any job parameters, especially if the job instance identity is relevant (to benefit from Spring Batch's restart features for instance), so let's see how to specify job parameters to the command line job runner.

LAUNCHING WITH JOB PARAMETERS

Recall that our custom command line launcher uses two job parameters for the import products jobs: the location of the input file and the current date. The following snippet shows how to specify those parameters from the command line:

```
java -classpath "./lib/*"
[CA] org.springframework.batch.core.launch.support.CommandLineJobRunner [CA] import-products-job.xml importProductsJob
[CA] inputFile=file:./products.txt date=2010/12/08
```

The syntax is simple: you specify job parameters after the name of the job, using the `name=value` syntax. Remember that a job parameter can have a data type in Spring Batch. The way we define our parameters in the previous snippet creates `String`–typed parameters. What if the parameter type is relevant? Spring Batch offers a way to specify the type of a parameter, by using the syntax `name(type)=value`, where `type` can be a `string`, `date`, `long` or `double` (`string` being the default.) Let's now launch our job by passing in the `date` parameter as real `Date` object:

```
java -classpath "./lib/*"
[CA] org.springframework.batch.core.launch.support.CommandLineJobRunner
[CA] import-products-job.xml importProductsJob
[CA] inputFile=file:./products.txt date(date)=2010/12/08
```

Note the format of the date: yyyy/MM/dd. Table 5.2 lists the different types of job parameters along with examples.

Table 5.2 Job parameters types for CommandLineJobRunner

Type	Java type	Example
string	java.lang.String	`inputFile(string)=products.txt`
date	java.util.Date	`date(date)=2010/12/08`
long	Long	`timeout(long)=1000`
double	Double	`delta(double)=20.1`

This completes our tour of the `CommandLineJobRunner class`. This command line launcher is handy as it allows you to specify a Spring configuration file, the name of the job you want to start and job parameters (with some advanced type-conversion).

Let's now see an advanced feature of the runner that you use when you need to set the system exit code returned by the launcher. Use this feature if you want to run a series of jobs and choose precisely which job should follow a previous job.

HANDLING EXIT CODES

The `CommandLineJobRunner` lets you set the exit code to return when the job execution ends. The triggering system (a system scheduler for example) can use this exit code to decide what to do next (see the callout on the use of exit codes.) For example, after the execution of job A, you want to run either job B or job C. The scheduler decides based on the exit code returned by A.

If you use the `CommandLineJobRunner` but don't care about exit codes, because you don't execute sequences of jobs or you organize all the sequencing of your batch processes as Spring Batch steps, you can easily skip this sub-section. But, if your batch system relies on exit codes to organize the sequencing of your jobs, you'll learn here how Spring Batch lets you easily choose which exit code to return from a job execution.

What's the deal with exit codes?

A system process always returns an integer exit code when it terminates. As previously mentioned, system schedulers commonly trigger batch processes launched from the command line and these schedulers can be interested in the exit code of the batch process. Why? To determine the course of action. An exit code of 0 could mean that everything went ok, 1 could mean that a fatal error occurred, and 2 could mean that the job must be restarted. That's why the Spring Batch command line launcher provides advanced support to map job exit statuses (string) with system exit codes (integer).

The `CommandLineJobRunner` uses an *exit code mapper* to map a job's exit status (a String) with a system exit code (an integer). Figure 5.6 illustrates this mapping.

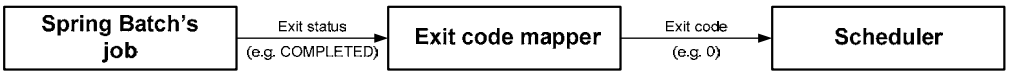

Figure 5.6 The command line job runner uses an exit code mapper to translate the String exit status of a Spring Batch job into an integer system exit code. The triggering system – a system scheduler here – can then use this system exit code to decide want to do next.

What is the exit code for a Spring Batch job? A job's exit code is a property of the job's exit status, which is itself part of the job execution returned by the job launcher. Spring Batch provides an `ExitStatus` class, which includes an exit code typed as a `String`. Don't confuse `BatchStatus` (an enumeration) and `ExitStatus` (a simple String)! These are different concepts, even if in most cases, the exit status is directly determined from the batch status. Chapter 10 – on handling the execution of job steps – provides in-depth coverage of the batch and exit status. For now, just remember that by default Spring Batch gets the exit status from the batch status (either COMPLETED or FAILED) and that you can override this default behavior if you want to return a specific exit status.

Table 5.3 explains the `CommandLineJobRunner` default behavior for exit code mappings (the `SimpleJvmExitCodeMapper` class implements this behavior).

Table 5.3 Default exit code mappings

System exit code	Job's exit status
0	The job completed successfully (COMPLETED).
1	The job failed (FAILED).
2	Used for errors from the command line job runner. For example, the runner could not find the job in the Spring application context.

You can override the defaults listed in table 5.3 if they don't suit your needs. How do you do that? Write an implementation of the `ExitCodeMapper` interface and declare a Spring bean of the corresponding type in the job's Spring application context. There is nothing more to do, because the `CommandLineJobRunner` will automatically use the `ExitCodeMapper`.

Let's look at an example to illustrate overriding the default exit code mapper. Remember the goal is to use the exit code returned by a job to decide what to do next. Let's imagine that this job (let's call it job A) deals with importing items from a file into a database. The system scheduler we're using will run job A and will behave as follows depending on the exit code returned by job A:

- 0 – starts job B (job A completed)
- 1 – does nothing (job A failed)
- 2 – does nothing (job A exited with an unknown job exit status)
- 3 – starts job C (job A completed, but skipped some items during processing)

Your job as the developer of job A is to return the correct exit code, such that the system scheduler uses it to decide what to do next. To do so, you write an implementation of `ExitCodeMapper` to handle the exit code strategy and install it in job A. Listing 5.2 shows the implementation of an `ExitCodeMapper` that honors this contract.

Listing 5.2 Writing an ExitCodeMapper to map job and system exit codes

```
package com.manning.sbia.ch05;

import org.springframework.batch.core.ExitStatus;
import org.springframework.batch.core.launch.support.ExitCodeMapper;
```

```
public class SkippedAwareExitCodeMapper implements ExitCodeMapper {

  @Override
  public int intValue(String exitCode) {
    if(ExitStatus.COMPLETED.getExitCode().equals(exitCode)) {
      return 0;
    } else if(ExitStatus.FAILED.getExitCode().equals(exitCode)) {
      return 1;
    } else if("COMPLETED WITH SKIPS".equals(exitCode)) {
      return 3;
    } else {
      return 2;
    }
  }

}
```

Note that the `exitCode` argument of the `intValue` method comes from the `ExitStatus` object of the job, which has a `getExitCode()` method. Implementing an exit code mapper is straightforward: you get a `String` and return a matching integer. But how can the job's exit code (the `String` argument of the `exitCode` method) get values like "COMPLETED WITH SKIPS"? This is not magic, you set the value at the step level, and chapter 10 teaches you how to do that. Let's assume here that we configured our job correctly to receive the appropriate exit status if the job skipped some items.

Now that the exit code mapper is implemented, we need to declare it in the Spring configuration, as shown in the following snippet:

```
<bean class="com.manning.sbia.ch05.SkippedAwareExitCodeMapper" />

<job id="importProductsJob"
    xmlns="http://www.springframework.org/schema/batch">
  (...)
</job>
(...)
```

That's it; you map exactly what Spring Batch exit code maps to what system exit code! All you do is declare an exit code mapper bean alongside your job configuration, the `CommandLineJobRunner` detects and uses the mapper automatically.

You now know how to launch Spring Batch jobs from the command line. When using the command line to launch a batch job, you need someone or something to trigger this command line. There are many ways to trigger batch jobs and job schedulers are great tools to trigger jobs at specific times or periodically. This is the topic of our next section.

5.3 Job schedulers

A job scheduler is a program in charge of periodically launching other programs, in our case, batch processes. Imagine that you have a time frame between 2 and 4 in the morning to re-index your product catalog (because there are few users connected to the online application at that time,) or that you want to scan a directory every minute between 6 AM and 8 PM for new files to import. How would you do that? You can implement a solution yourself using a programming language like Java, but this is time consuming and error prone, and system utilities are probably not the focus of your business.

Alternatively, Job schedulers are perfect for this work: triggering a program at a specific time, periodically or not.

> Warning Don't confuse job scheduling – our topic – with process scheduling, which is about assigning processes to CPUs, at the operating system level.

Our goal here is to use several job schedulers to launch Spring Batch jobs. We don't cover these job schedulers just for fun. We picked popular, mainstream, and free job schedulers to provide you with guidelines for choosing one over of another, depending on the context of your applications. Before we dive in the description of each solution, table 5.4 lists the job schedulers we cover, their main characteristics, and features.

Table 5.4 Overview of the job schedulers covered in this section

Job scheduler	Description
Cron	A job scheduler available on Unix-like systems. Uses cron expressions to periodically launch commands or shell scripts.
Spring scheduler	The Spring framework scheduler. Configurable with XML or annotations, it supports cron expressions.
Quartz	A Java-based job scheduler. Supports cron expressions and enterprise features like JTA and clustering.

The descriptions in table 5.4 might already have helped you make up your mind: if your application doesn't run on a Unix-like system, you won't be using cron! In any case, we invite you to read about cron expressions in the cron section, as they're a great way to configure other job schedulers, like the Spring scheduler and Quartz.

5.3.1 Using cron and cron expressions

cron is the de facto job scheduler program on Unix-like systems. The name cron comes from the Greek "chronos" (for "time".) cron enables launching commands or shell scripts periodically, using *cron expressions*. Configuring cron is simple: you set up commands to launch and when to launch them in the *crontab* file.

CONFIGURING CRON WITH CRONTAB

The system wide crontab file is stored in the /etc/ directory. Figure 5.7 shows the structure of a line of the crontab file.

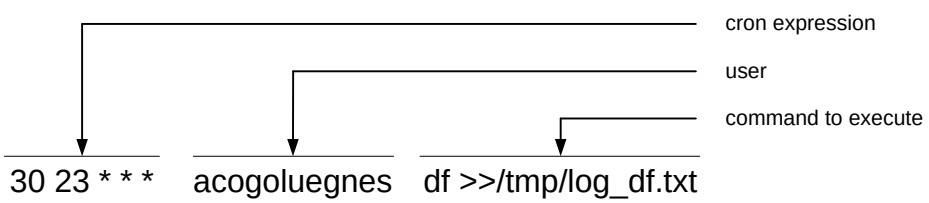

Figure 5.7 An entry in the crontab file has three parts. First is the cron expression, which schedules the job execution. Second is the user who runs the command. Third is the command to execute. Note that some cron implementations

don't have the user option.

The command can be anything, in our case, it can be something we covered in section 5.2 –command line launching. The following snippet shows an entry to launch a job with Spring Batch's command line job launcher, with the acogoluegnes user:

```
0 4 * * ?    acogoluegnes    java -classpath "/usr/local/bin/sb/lib/*"
[CA] org.springframework.batch.core.launch.support.CommandLineJobRunner
[CA] import-products-job.xml importProductsJob
[CA] inputFile=file:/home/sb/import/products.txt date=2010/12/08
```

From the previous snippet, you should recognize the structure of a cron entry (cron expression, user, and command). The command is long: it must set the classpath, the Java class to launch, the Spring configuration file to use, the name of the job to launch, and the job parameters. You can use any command in a crontab entry: Spring Batch's command line launcher or any other command to launch a job process. Next is choosing when to trigger the command, which is where we use a cron expression.

If you're new to cron, the start of the entry in the previous snippet must be puzzling: this is a cron expression, which says to launch the job every day at four in the morning. cron expressions are to scheduling what regular expressions are to string matching. Depending on your background with regular expressions, this assertion can be appealing or scary! Don't worry; the next sub-section will make a cron expression master out of you.

DEMYSTIFYING CRON EXPRESSIONS

cron expressions are a powerful yet easy tool to schedule events like "every Wednesday at 18" or "the last weekday of the month at 23:00". Figure 5.8 shows the components of a cron expression.

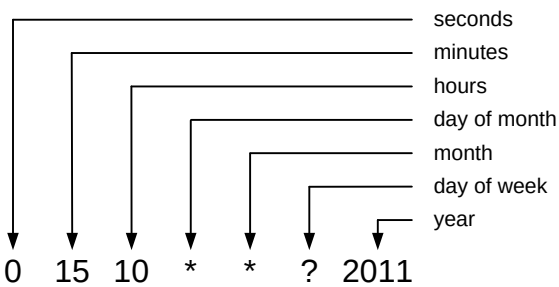

Figure 5.8 A cron expression has 6 or 7 fields (the year field is optional). Each field can accept a set of values, special characters or a range of values. This expression means "every day of the year 2011 at 10:15 in the morning."

You see in figure 11.8 that a cron expression has of 6 or 7 fields (the year field is optional), corresponding to seconds, minutes, hours, and the day of the month.

> Warning The cron implementation in Unix-like system doesn't support seconds, so on these systems, you need to remove the seconds field.

These fields can take values that make a cron expression look like a black magic formula! Don't worry; each field can have a set of values (0, 1, and 2, up to 23 for hours) and some specials characters. Table 5.5 lists the different values and special characters that each field accepts.

Table 5.5 Field syntax in a cron expression

Field	Mandatory	Value range	Special characters
Seconds	Yes	0-59	* / , -
Minutes	Yes	0-59	* / , -
Hours	Yes	0-23	* / , -
Day of month	Yes	1-31	* / , - ? L W
Month	Yes	1-12 or JAN-DEC	* / , -
Day of week	Yes	1-7 or SUN-SAT	* / , - ? L #
Year	No	1970-2099	* / , -

You can't go very far with cron expressions without special characters. For example, these characters allow starting a job on the last Friday of the month or every minute between 8:00 and 18:00. Table 5.6 lists the special characters and provides examples.

Table 5.6 Special characters in cron expressions

Character	Description	Examples
*	Matches all values of the field (* in the month field means "every month").	`* * * * * ?` : every second
/	Means increment.	`0/15` in seconds field: for seconds 0, 15, 30 and 45
		`5/15` in seconds field: for seconds 5, 20, 15 and 50
,	Specifies additional values.	`MON,WED,FRI` : Monday, Wednesday and Friday
-	Specifies ranges.	`8-10` in hours field: for hours 8, 9 and 10
		`MON-WED` : for Monday, Tuesday and Wednesday
?	Specifies the value is irrelevant (for the day of the week or the day of the month fields).	`0 15 8 10 * ?` : the 10th of every month at 8:15 in the morning
		`0 15 8 ? * MON` : every Monday at 8:15 in the morning.
L	Stands for "last".	`0 15 8 L * ?` : the last day of every

		month, at 8:10 in the morning. `0 15 8 ? * 6L` : the last Friday of every month, at 8:10 in the morning.
W	Specifies the nearest weekday to the given day.	`0 15 8 15W * ?` : the nearest weekday to the 15th of every month, at 8:15 in the morning. `0 15 8 LW * ?` : the last weekday of every month, at 8:15 in the morning.
#	Allows for constructs like the second Friday of the month.	`0 15 8 ? * 2#1` : the first Monday of the month, at 8:15 in the morning. `0 15 8 ? * 6#1` : the first Friday of the month, at 8:15 in the morning.

Let's see a couple of examples. Cynthia – our development editor at Manning – asks us to send her our status report every Wednesday. Let's say that 18:00 (6 PM) is a convenient time and that the writing of the book will span only – hopefully – the years 2010 and 2011. Here is the cron expression that will trigger an alert to remind us to send an email to Cynthia:

```
0 0 18 ? * 4 2010-2011
```

Imagine that the online store application should create a monthly sales report and send it to an administrator. This should happen the last weekday of every month, at 23:00 (11 PM.) Here is the corresponding expression:

```
0 0 23 LW * ?
```

Table 5.7 lists common cron expressions. If you're ever desperately looking for a cron expression and you feel lucky, check table 5.7.

Table 5.7 Examples of cron expressions

Expression	Description
`0 0 4 * * ?`	Everyday at 4 in the morning.
`0 0 4 * * ? 2011`	Everyday at 4 in the morning in the year 2011.
`0 0/5 8-18 ? * MON-FRI`	Every 5 minutes, from 8 to 17:55, on weekdays (Monday through Friday).
`0 0 23 L * ?`	The last day of every month, at 23:00.
`0 0 23 LW * ?`	The last weekday of every month, at 23:00.
`0 0 23 15W * ?`	The closest weekday to the 15 of every month, at 23:00.
`0 0 23 ? * 6#3`	The third Friday of every month, at 23:00.

By now, you should be a master of cron expressions and be able to schedule your job processes. Let's now see some recommendations about the use of cron.

CRON FOR MY SPRING BATCH JOBS?

Is cron suited to launch your Spring Batch job? Remember, cron is a system scheduler: it spawns a new JVM process for each Spring Batch command line launcher. Imagine that you need to launch a job every night. cron triggers the command line launcher, which creates a Spring application context before launching the job itself. Everything is fine. But imagine now that you need to launch another job which scans a directory for new files to import. You set up cron to trigger this job every minute. If bootstrapping the Spring application context is CPU-intensive – because it initializes a Hibernate `SessionFactory` or a JPA context for example – the job execution will perhaps be faster than the creation of the Spring application context! In this second case, you prefer to have your Spring application context already running and then simply launch the job from the existing `JobLauncher`. You can't easily achieve this from the command line (hence with cron) but a Java scheduler like the Spring scheduler or Quartz will do the trick.

5.3.2 Using the Spring scheduler

Let's now look at the second scheduling option from table 5.4: the Spring scheduler. Do you want to schedule a job with a simple-to-deploy and yet powerful solution? Good news, Spring includes such a feature: As of version 3.0, the Spring Framework offers a declarative way to schedule jobs, without requiring extra dependencies for your Spring Batch jobs, because Spring Batch sits on top of Spring.

Spring's lightweight scheduling provides features like cron expressions, customization of threading policy, and declarative configuration with XML or annotations. The Spring scheduler needs a running Spring application context to work, so you typically embed it in a web application, but you can use any other managed environment, like an OSGi container. We cover how to embed Spring Batch in a web application in section 5.4.

Note Spring's scheduler supports seconds in cron expressions.

The steps to use the Spring scheduler are:

- Set up the scheduler – this is where you decide to use a thread pool or not. This set up is optional and Spring uses a single-threaded scheduler by default.
- Set up the Java methods to launch periodically – you can use XML or annotations on the target methods. In our case, those methods use the Spring Batch API to launch jobs.

The next sub-sections cover these steps, but let's first see what kind of scheduling configuration Spring supports.

SCHEDULING OPTIONS

Your scheduling requirements can be as simple as "every minute" or as complex as "the last weekday of the month at 23:00". Sections 5.3.1 shows that cron expressions meet both requirements, but do you really need to unleash the big guns for "every minute?" Could we use something simple for simple requirements and fall back to cron expressions only when necessary? Spring allows you to do that by supporting cron expressions – with its own engine – but also lets you trigger a job at a fixed rate, without resorting to cron expressions. Table 5.8 lists the scheduling options that Spring offers.

Table 5.8 Spring scheduling options

Scheduling option	XML attribute	Annotation attribute	Description
Fixed rate	`fixed-rate`	`fixedRate`	Launches periodically, using the *start* time of the previous task to measure the interval.
Fixed delay	`fixed-delay`	`fixedDelay`	Launches periodically, using the *completion* time of the previous task to measure the interval.
Cron	`cron`	`cron`	Launches using a cron expression.

The fixed-rate and fixed-delay options are the simple options, depending on whether you want to launch job executions independently (fixed-rate) or depending on the completion time of the previous execution (fixed-delay.) For cases that are more complex, use cron expressions. The next sub-sections show you the use of the fixed-rate option with both XML and annotations; remember that you can use the attributes in table 5.8 for fixed-rate or cron.

SCHEDULER SETUP

Spring uses a dedicated bean to schedule jobs. You can declare this bean using the `task` namespace prefix:

```
<task:scheduler id="scheduler" />
```

> Note Remember that declaring a scheduler is optional. Spring will use the default single-threaded scheduler as soon as you declare scheduled tasks.

Even though Spring uses reasonable defaults, declaring a scheduler explicitly is good practice because it reminds you that an infrastructure bean takes care of the actual scheduling. It also serves as a reminder that that you can tweak this scheduler to use a thread pool:

```
<task:scheduler id="scheduler" pool-size="10" />
```

Using multiple threads is useful when you need to schedule multiple jobs and their launch time can overlap. You don't want some jobs to wait because the single thread of your scheduler is busy launching another job.

Now that the scheduler is ready, let's schedule a job using XML.

SCHEDULING WITH XML

Imagine you have the following Java code that launches your Spring Batch job, and you want Spring to execute this code periodically:

```
package com.manning.sbia.ch05;

import java.util.Date;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;

public class SpringSchedulingLauncher {
```

```
  private Job job;

  private JobLauncher jobLauncher;

  public void launch() throws Exception {                    #A
    JobParameters jobParams = createJobParameters();              #A
    jobLauncher.run(job, jobParams);                          #A
  }

  private JobParameters createJobParameters() {                  #B
    (...)                                                 #B
  }                                                       #B
  (...)
}
```

#A Launches job
#B Creates job parameters

Note that the previous snippet elides setter methods for brevity. It also elides the creation of job parameters, as job parameters are job specific. Most of the time, you'll be using a timestamp or a sequence to change the job identity for each run. Finally, exception handling is up to you: here, the `launch` method just propagates any exception that the job launcher throws. You could also catch the exception and log it.

We now need to tell Spring to call this code periodically. Fortunately, we inherit from all of Spring's configuration features: dependency injection and the `task` namespace to configure our scheduling. Listing 5.3 shows a scheduling configuration using XML.

Listing 5.3 Scheduling with Spring and XML

```
<bean id="springSchedulingLauncher"                          #1
    class="com.manning.sbia.ch05.                          #1
[CA] SpringSchedulingLauncher">                             #1
  <property name="job" ref="job" />                         #1
  <property name="jobLauncher" ref="jobLauncher" />              #1
</bean>                                                  #1

<task:scheduler id="scheduler" />

<task:scheduled-tasks scheduler="scheduler">                    #2
  <task:scheduled ref="springSchedulingLauncher"                #2
            method="launch"                              #2
            fixed-rate="1000" />                         #2
</task:scheduled-tasks>                                    #2
```

#1 Declares launcher bean
#2 Schedules job at fixed rate

We declare the bean that launches the Spring Batch job at #1. The `task:scheduled-tasks` element starting at #2 contains the tasks to schedule. For each task we schedule, we use the `task:scheduled` element and refer to the bean and the method to call, using the `ref` and `method` attributes, respectively. Here, we use a fixed rate, but remember that we can also schedule with a fixed delay or a cron expression.

An XML configuration has many advantages: it doesn't affect your Java code – making it easier to reuse – and it's flexible because you can externalize part of your configuration in a property file, using a

Spring property placeholder. This allows switching the scheduling configuration between the development and production environments for example. The XML configuration is external to your code: when you look at your Java code, you have no idea a scheduler launches it periodically. If you change the name of the Java method to launch periodically, you need to reflect this change in the XML configuration. If you want the scheduling configuration to be closer to your code than a separate XML file, then annotations are the way to go.

SCHEDULING WITH ANNOTATIONS

Spring lets you schedule your jobs by annotating Java methods. The following snippet shows how to schedule a job with the Spring `@Scheduled` annotation:

```
package com.manning.sbia.ch05;

import java.util.Date;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.scheduling.annotation.Scheduled;

public class SpringSchedulingAnnotatedLauncher {

  private Job job;

  private JobLauncher jobLauncher;

  @Scheduled(fixedRate=1000)                                    #A
  public void launch() throws Exception {
    JobParameters jobParams = createJobParameters();
    jobLauncher.run(job, jobParams);
  }

  private JobParameters createJobParameters() {
    (...)
  }
  (...)
}
```

#A Schedules job with fixed rate

Note Don't forget: you can use the `fixedDelay` or `cron` annotation attributes instead of `fixedRate`.

When using the `@Scheduled` annotation, the Java class does part of the configuration itself. The XML configuration is shorter but you need to tell Spring to look for `@Scheduled` annotations with the `task:annotation-driven` element, as shown in the following snippet:

```
<bean id="springSchedulingAnnotatedLauncher"
    class="com.manning.sbia.ch05.
[CA] SpringSchedulingAnnotatedLauncher">
  <property name="job" ref="job" />
  <property name="jobLauncher" ref="jobLauncher" />
</bean>

<task:scheduler id="scheduler" />
```

```
<task:annotation-driven scheduler="scheduler" />                    #A
```

#A Detects @Scheduled annotations

Using the `@Scheduled` annotation is straightforward: you activate its support with the `task:annotation-driven` element and you add new tasks directly in Java, without going back to the XML configuration. However, the annotation solution is less flexible: the scheduling configuration is hard-coded and it works only on that code (you can't annotate code you don't control.)

Spring scheduling offers a good trade-off: it's free, handles cron expressions and lets you customize the threading strategy. Nevertheless, it's not a full-blown scheduler and it lacks advanced features like clustering for example. If you reach the limits of the Spring scheduler, take a look at Quartz, a Java-based job scheduler.

5.3.3 Using Quartz

We've now reached the last scheduling solution listed in table 5.4: the Quartz scheduler. Quartz is a Java-based job scheduler that you can integrate in any Java environment (standalone or Java EE.) Quartz can schedule thousands of jobs and you can distribute the processing on multiple Quartz instances that use a centralized database configuration. Spring integrates with Quartz, so this sub-section covers how to use Quartz to trigger a Spring Batch job.

Note Quartz supports seconds in cron expressions.

Listing 5.4 shows a simple class that launches a Spring Batch job. Because the class implements the Quartz `Job` interface, we can hand off an instance of this class to the Quartz scheduler.

Listing 5.4 Launching a Spring Batch job as a Quartz job

```
package com.manning.sbia.ch05;

import java.util.Date;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;

public class QuartzLauncher                                    #A
   implements org.quartz.Job {                                 #A

  private Job job;

  private JobLauncher jobLauncher;

  @Override                                                    #B
  public void execute(JobExecutionContext context)                  #B
     throws JobExecutionException {                                  #B
    try {                                                  #B
      JobParameters jobParams = createJobParameters();                   #B
      jobLauncher.run(job, jobParams);                                   #B
    } catch(Exception e) {                                 #B
```

```
    throw new JobExecutionException(                          #B
      "error while executing Spring Batch job", e);           #B
   }                                                  #B
 }                                                    #B

 private JobParameters createJobParameters() {
  (...)
 }

 public void setJob(Job job) {
  this.job = job;
 }

 public void setJobLauncher(JobLauncher jobLauncher) {
  this.jobLauncher = jobLauncher;
 }

}
```

#A Implements Quartz's Job interface
#B Launches Spring Batch job

warning Spring Batch and Quartz both have an interface called `Job`, don't confuse them!

Our implementation of the `execute` method from the Quartz `Job` interface launches Spring Batch jobs. As usual, we need references to the `Job` and the `JobLauncher`. The next step is to configure Quartz in a Spring application context, as shown in listing 5.5.

Listing 5.5 Configuring Quartz in a Spring application context

```
<bean id="jobDetail"                                      #1
    class="org.springframework.scheduling.quartz.              #1
[CA] JobDetailBean">                                      #1
  <property name="jobClass"                                  #1
        value="com.manning.sbia.ch05.                          #1
[CA] QuartzLauncher" />                                   #1
</bean>                                                #1

<bean id="jobDetailTrigger"                                  #2
    class="org.springframework.scheduling.quartz.              #2
[CA] CronTriggerBean">                                    #2
  <property name="jobDetail" ref="jobDetail" />                 #2
  <property name="cronExpression"                             #2
        value="* * * * * ?" />                             #2
</bean>                                                #2

<bean id="scheduler"
    class="org.springframework.scheduling.quartz.
[CA] SchedulerFactoryBean">
  <property name="triggers">                                  #3
   <list>                                              #3
    <ref bean="jobDetailTrigger" />                           #3
   </list>                                             #3
  </property>                                             #3
  <property name="jobFactory">
   <bean class="org.springframework.scheduling.
```

```
[CA] quartz.SpringBeanJobFactory" />
  </property>
  <property name="schedulerContextAsMap">                    #4
    <map>                                              #4
      <entry key="job" value-ref="job" />                    #4
      <entry key="jobLauncher"                             #4
           value-ref="jobLauncher" />                      #4
    </map>                                             #4
  </property>                                           #4
</bean>
```

#1 Declares job
#2 Declares trigger
#3 Adds trigger to Quartz scheduler
#4 Adds Spring Batch beans to scheduler context

The `jobDetail` bean at #1 contains the details of the Quartz job we want to trigger. Note that the bean refers to our `QuartzLauncher` class from listing 5.4. The `jobDetailTrigger` bean at #2 is in charge of triggering the job. You need to declare two such beans for each job you want to launch. You declare the next bean – `scheduler` – only once, as it defines the Quartz infrastructure. This scheduler `triggers` property contains all the triggers Quartz should activate: in our example, that is the `jobDetailTrigger` bean #3. At #4, we add the Spring Batch job and job launcher beans to the scheduler's context. By doing this, the job and job launcher will be injected in our `QuartzLauncher` instance. For this injection to work the `QuartzLauncher` class must have a corresponding setter method and the scheduler must use a `SpringBeanJobFactory` (this is what we injected in the scheduler's `jobFactory` property.) Figure 5.9 shows a simplified view of the usefulness of each beans in the integration with Spring and Quartz.

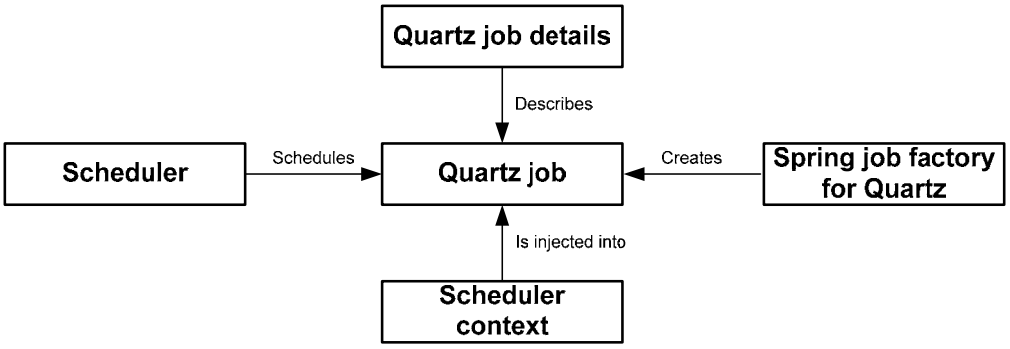

Figure 5.9 The integration with Spring and Quartz. The job details bean describes the job that the job factory creates. The scheduler context contains beans that can be injected in the Quartz job. Spring Batch's job launcher and job can be in the scheduler context, so the Quartz job can use them to start an execution.

This ends the coverage of schedulers used to launch Spring Batch jobs. You can use a system scheduler like cron to launch Spring Batch jobs, which spawns a plain Java process for each job. But cron isn't suited for all cases, especially if bootstrapping the Spring application context is resource-intensive and the job is triggered every second, for example. In such cases, use a Java-based

scheduler. For simple scenarios, use the Spring scheduler. To run hundreds of jobs in a cluster environment, Quartz is a better solution.

Now, you know everything about schedulers, especially Java-based schedulers. Remember, when using a Java scheduler, you already have a Spring Batch environment ready; you don't need to spawn a new JVM process for every job (as you do with cron, for example.) You now have everything ready, assuming you found a container to run your application. A popular way to embed a Spring Batch environment and a scheduler is to use a web application. This is the second scenario presented in section 5.1.3. In the next section, we see how to embed Spring Batch in a web application

5.4 Launching from a web application

Spring Batch is a lightweight framework that can live in a simple Spring application context. Here, we look at configuring a Spring Batch environment in a web application. This makes Spring Batch available at any time; there is no need to spawn a dedicated Java process to launch a job. We can also embed a Java scheduler in the same web application context and become independent of any system schedulers. Figure 5.10 illustrates what a Spring application context can contain in a web application. Note that the job beans can also use any available services, like data sources, data access objects, and business services.

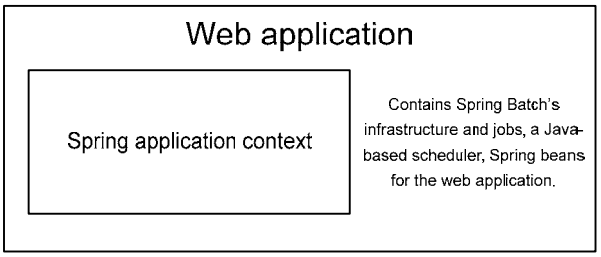

Figure 5.10 A web application can contain a Spring application context. This Spring application context can host Spring Batch's infrastructure (job launcher, job repository) and jobs. The context can also host a Java-based scheduler like Quartz and any Spring beans related to the web application (data access objects, business services.)

Hosting Spring Batch in a web application is convenient, but what about pushing this architecture further and triggering jobs through HTTP requests? This is useful when an external system triggers jobs and that system cannot easily communicate with the Spring Batch environment. But before we study how to use HTTP to trigger jobs, let's see how to configure Spring Batch in a web application.

5.4.1 Embedding Spring Batch in a web application

The Spring Framework provides a servlet listener class – the `ContextLoaderListener` – that manages the application context's lifecycle according to the web application lifecycle. We refer to this application context as the *root* application context of the web application. We configure the servlet listener in the web.xml file of the web application, as shown in listing 5.6.

Listing 5.6 Configuring Spring in a web application

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">
  <display-name>Spring Batch in a web application</display-name>

  <listener>
    <listener-class>
    org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

</web-app>
```

By default, the `ContextLoaderListener` class uses an applicationContext.xml file in the WEB-INF directory of the web application to create the application context. This file should contain the configuration of the Spring Batch infrastructure, the jobs, the scheduler (if any) and application services. A best practice consists in splitting up this configuration into multiple files. This avoids having a large and monolithic configuration file and encourages reuse of configuration files. Should you redefine all your jobs for integration testing? No, so define the jobs in a dedicated file and import this file from a master Spring file. The following snippet shows how the default applicationContext.xml imports other files to create a more maintainable and reusable configuration:

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

  <import resource="batch-infrastructure.xml"/>
  <import resource="batch-jobs.xml"/>
  <import resource="scheduling.xml"/>

</beans>
```

If you follow the configuration of the previous snippet, the structure of the web application on disk should be as follows:

```
web application root directory/
  WEB-INF/
    applicationContext.xml
    batch-infrastructure.xml
    batch-jobs.xml
    scheduling.xml
    web.xml
```

What's next? If you use the Spring scheduler or Quartz to start your jobs, the scheduling.xml file contains the corresponding configuration and you're done! You can deploy the web application in your favorite web container and the embedded Java scheduler will trigger jobs according to the configuration. Figure 5.11 shows this configuration.

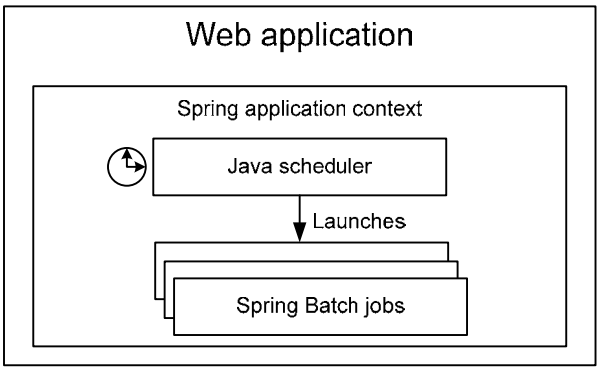

Figure 5.11 Once Spring Batch is in a web application, you can use an embedded Java scheduler like Quartz to launch jobs periodically.

In many cases, this configuration will be fine. But in others, the triggering event doesn't come from an embedded scheduler but from an external system. Next, we use HTTP to let this external system get access to our Spring Batch environment.

5.4.2 Launching a job with an HTTP request

Imagine that you deployed your Spring Batch environment in a web application but that a system scheduler is in charge of triggering your Spring Batch jobs. A system scheduler like `cron` is easy to configure and that might be what your administration team prefers to use. But how can `cron` get access to Spring Batch, which is now in a web application? By using a command that performs an HTTP request and schedule that command in the crontab! Here is how to perform an HTTP request with a command line tool like `wget`:

```
wget "http://localhost:8080/sbia/joblauncher?job=importProductsJob&
[CA] date=20101218"
```

Figure 5.12 illustrates launching a Spring Batch job with an HTTP request.

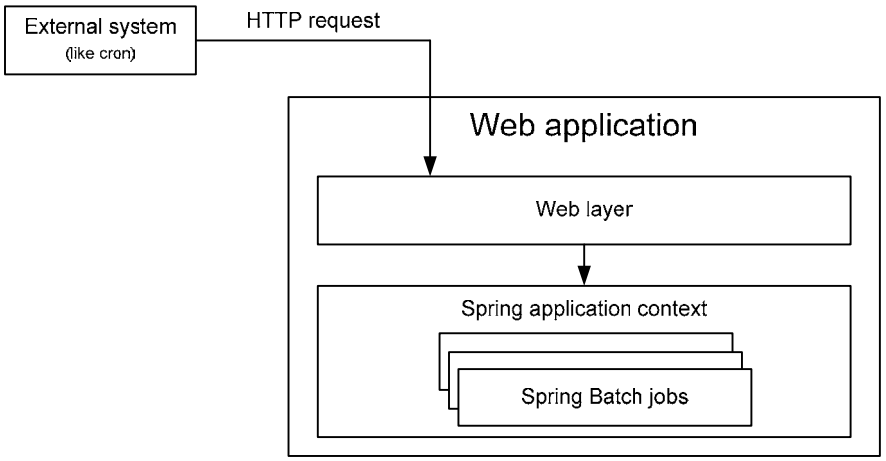

Figure 5.12 Once Spring Batch is in a web application, you can add a web layer to launch Spring Batch jobs on incoming HTTP requests. This solution is convenient when the triggering system is external to Spring Batch (like cron.)

To implement this architecture, you need a web controller that analyzes the HTTP parameters and triggers the corresponding job with its parameters. We're going to use Spring MVC to do that but we could have used any other web framework. We chose Spring MVC because it's part of the Spring Framework, so it's free to our Spring Batch application.

IMPLEMENTING A SPRING MVC CONTROLLER TO LAUNCH JOBS

Spring MVC is part of the Spring Framework and provides a simple yet powerful way to write web applications or REST web services. In Spring MVC, controllers are plain Java classes, with some annotations. Listing 5.7 shows our job launcher controller.

Listing 5.7 A Spring MVC controller job launcher

```
package com.manning.sbia.ch05.web;

import java.util.Enumeration;
import javax.servlet.http.HttpServletRequest;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.configuration.JobRegistry;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;

@Controller
public class JobLauncherController {

  private static final String JOB_PARAM = "job";
```

```
 private JobLauncher jobLauncher;
 private JobRegistry jobRegistry;

 public JobLauncherController(JobLauncher jobLauncher,
    JobRegistry jobRegistry) {
  super();
  this.jobLauncher = jobLauncher;
  this.jobRegistry = jobRegistry;
 }

 @RequestMapping(value="joblauncher",method=RequestMethod.GET)
 @ResponseStatus(HttpStatus.ACCEPTED)
 public void launch(@RequestParam String job,                    #1
    HttpServletRequest request) throws Exception {
  JobParametersBuilder builder = extractParameters(              #2
    request                                                      #2
  );                                                             #2
  jobLauncher.run(                                               #3
    jobRegistry.getJob(request.getParameter(JOB_PARAM)),         #3
    builder.toJobParameters()                                    #3
  );                                                             #3
 }

 private JobParametersBuilder extractParameters(
     HttpServletRequest request) {
  JobParametersBuilder builder = new JobParametersBuilder();
  Enumeration<String> paramNames = request.getParameterNames();
  while(paramNames.hasMoreElements()) {
    String paramName = paramNames.nextElement();
    if(!JOB_PARAM.equals(paramName)) {
      builder.addString(paramName,request.getParameter(paramName));
    }
  }
  return builder;
 }

}
```

#1 Gets job name from HTTP parameter
#2 Converts HTTP parameters into job parameters
#3 Launches job

The `@RequestMapping` annotation tells Spring MVC which URL and which HTTP operation to bind to the `launch` method. With the `@RequestParam` annotation on the `job` parameter at #1, we tell Spring MVC to pass the value of the job HTTP parameter to the method. As you probably guessed, this parameter is the name of the job we want to launch. At #2, we extract HTTP parameters and convert them to job parameters. At #3, we use the job launcher to launch the job. We use the `@ResponseStatus` annotation to return an empty HTTP response, with a 202 (ACCEPTED) status code.

> Note When using an HTTP request to start jobs, you should consider making the Spring Batch job launcher asynchronous, otherwise, the job execution will monopolize the web container's thread.

The launching request URL path should follow the following syntax:
/launcher?job=importProductsJob&param1=value1&param2=value2

Finally, you may have noticed the `jobRegistry` property in the web controller in listing 5.7. The `JobRegistry` is a Spring Batch interface used to lookup `Job` beans configured in the Spring application context. This is exactly what the launching controller does: from the job name passed in the request, it retrieves the corresponding `Job` bean. You need to declare the job registry in the Spring application context, typically where you declare the Spring Batch infrastructure. Following the structure we previously listed, we add the following code in the /WEB-INF/batch-infrastructure.xml file to declare the job registry:

```
<bean id="jobRegistry"
    class="org.springframework.batch.core.configuration.support.
[CA] MapJobRegistry" />
<bean class="org.springframework.batch.core.configuration.support.
[CA] JobRegistryBeanPostProcessor">
  <property name="jobRegistry" ref="jobRegistry" />
</bean>
```

Now the controller is ready, let's configure Spring MVC!

CONFIGURING SPRING MVC

At the heart of Spring MVC is a servlet class – DispatcherServlet – which we declare in the web.xml file of our web application, as shown in listing 5.8.

Listing 5.8 Declaring Spring MVC's servlet in web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">
  <display-name>Spring Batch in a web application</display-name>

  <listener>
    <listener-class>
   org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>                                          #A
    <servlet-name>sbia</servlet-name>                    #A
    <servlet-class>                                  #A
   org.springframework.web.servlet.DispatcherServlet       #A
    </servlet-class>                                 #A
  </servlet>                                         #A

  <servlet-mapping>                                    #B
    <servlet-name>sbia</servlet-name>                      #B
    <url-pattern>/*</url-pattern>                        #B
  </servlet-mapping>                                   #B
</web-app>
```

#A Declares Spring MVC servlet
#B Maps servlet to URLs

A Spring MVC servlet creates its own Spring application context. By default, its configuration file is {servlet-name}-servlet.xml, in our case, we create a sbia-servlet.xml file in the WEB-INF directory of the web application. We must declare the web controller in this file, as shown in the following snippet:

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

  <bean class="com.manning.sbia.ch05.web.                    #A
[CA] JobLauncherController">                              #A
    <constructor-arg ref="jobLauncher" />                    #A
    <constructor-arg ref="jobRegistry" />                    #A
  </bean>                                                 #A

</beans>
```

#A Declares job launcher web controller

In this configuration, we declare the controller and inject some dependencies, but where do these dependencies come from? From the root application context configured with the `ContextLoaderListener`. The Spring application context of the Spring MVC servlet can see the beans from the root application context, because they share a parent-child relationship, as figure 5.13 shows.

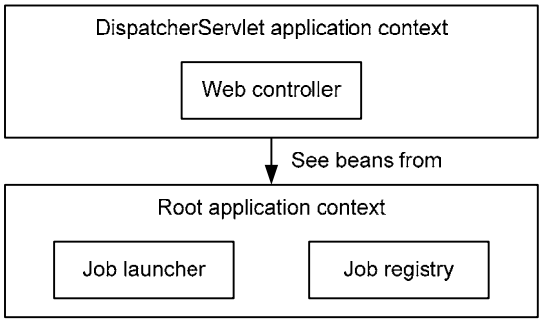

Figure 5.13 The web controller is defined in the servlet's application context. The job registry and the job launcher are defined in the root application context. As the two application contexts share a parent-child relation, the web controller can be injected with beans from the root application context.

We can now launch our Spring Batch jobs with a simple HTTP request! You should use this launching mechanism when an external system triggers your jobs and that system doesn't have direct access to your Spring Batch environment. Otherwise, you can just deploy your Spring Batch environment in a web application and use an embedded Java-based scheduler to trigger your jobs. Remember, you can use Spring Batch wherever you can use the Spring Framework, and web applications are no exception.

We covered a lot on triggering and launching Spring Batch jobs. By now, you should know which solution to adopt for your batch system. Next, we learn how to stop all these jobs.

5.5 Stopping jobs gracefully

We started many jobs in this chapter, but how do we stop them? Stopping a job is unfortunate because it means that something went wrong. If everything is OK, a job execution ends by itself, without any external intervention. When it comes to stopping job executions, we distinguish two points of view. The first is the operator's point of view. The operator monitors batch processes, he doesn't know much about Spring Batch, he just receives an alert when something goes wrong and stops a job execution, by using a JMX console for example.

The second is the developer's point of view. The developer writes Spring Batch jobs and knows that under certain circumstances he should stop a job. What are these certain circumstances? We refer here to any business decision that should prevent the job from going any further: the job shouldn't import more than 1,000 products a day for example, so the developer should count the imported items and make the execution stops just after the 1,000th item.

Spring Batch provides techniques to stop a job for both points of view, let's start with the operator.

5.5.1 Stopping a job for the operator

Imagine that the import job has been running for two hours and that you now receive the following phone call: "The import file contains bad data, there's no use letting the import running!" Obviously, you want the import to stop as soon as possible, to avoid wasting system resources on your server. Spring Batch provides the `JobOperator` interface to perform such an operation. The following snippet shows how to stop a job execution through a `JobOperator`:

```
Set<Long> runningExecs = jobOperator.getRunningExecutions("importJob");
Long executionId = runningExecs.iterator().next();
boolean stopMessageSent = jobOperator.stop(executionId);
```

The steps are simple: the job operator returns the identifiers of the running job executions for a given job name. You then ask the job operator to send a stop message to an execution using an execution ID. We discuss the notion of sending a stop message in the section titled "Understanding the stop message."

INVOKING THE JOB OPERATOR

The next question is how do you invoke this code? The most common way is to expose the job operator to JMX and call its method from a JMX console, as figure 5.14 illustrates using JConsole.

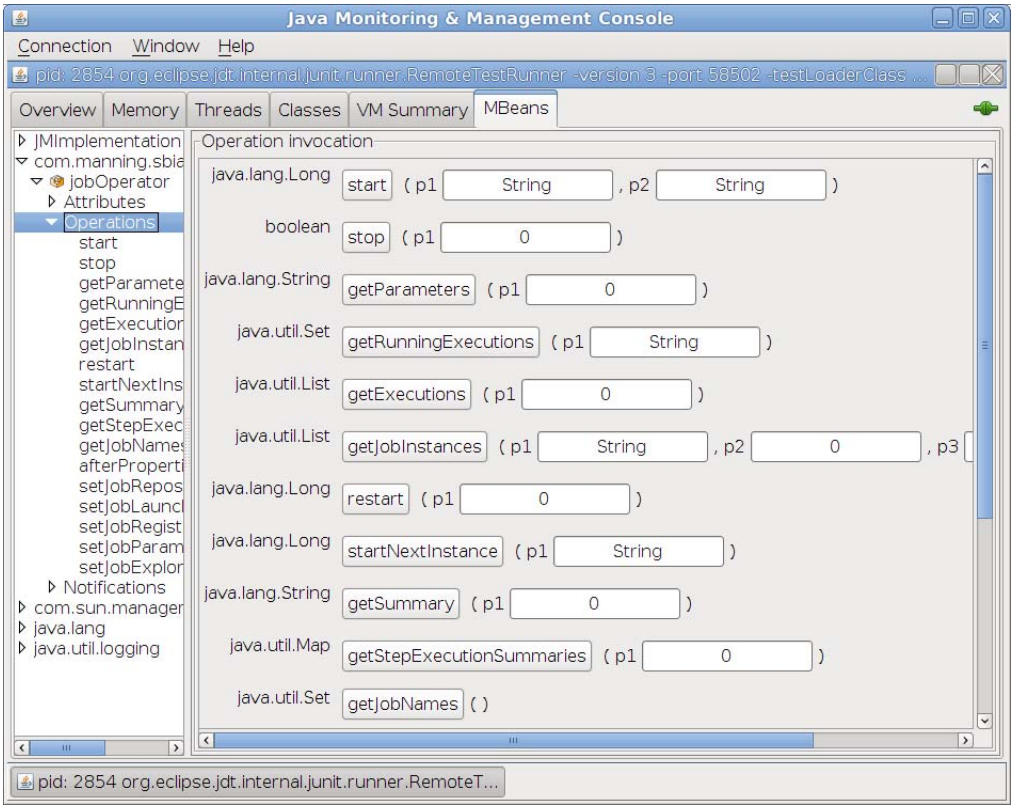

Figure 5.14 You can expose the job operator bean to JMX and then call its methods remotely, from a JMX client like JConsole. An operator can learn about the Spring Batch runtime and stop or restart jobs.

Another way to call job operator methods is to provide a user interface in your application that lets an administrator stop any job execution. You can create this user interface yourself but you can also use Spring Batch Admin, the web administration application introduced in chapter 3.

> Note Chapter 12 covers how to expose a Spring bean to JMX as well as how to monitor Spring Batch with the Spring Batch Admin application.

Now that you know how to use the job operator, let's see how to configure it.

CONFIGURING THE JOB OPERATOR

The job operator isn't automatically available; you need to declare it in your Spring configuration. Listing 5.9 shows the Spring configuration required to declare the job operator.

Listing 5.9 Configuring the job operator in Spring

```
<bean id="jobOperator" class="org.springframework.                #A
[CA] batch.core.launch.support.SimpleJobOperator">                #A
  <property name="jobRepository" ref="jobRepository"/>              #A
  <property name="jobLauncher" ref="jobLauncher" />                 #A
```

```
  <property name="jobRegistry" ref="jobRegistry" />                #A
  <property name="jobExplorer" ref="jobExplorer" />                #A
</bean>                                                  #A

<batch:job-repository id="jobRepository" data-source="dataSource" />

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

<bean class="org.springframework.batch.core.configuration.
[CA] support.JobRegistryBeanPostProcessor">
  <property name="jobRegistry" ref="jobRegistry" />
</bean>

<bean id="jobRegistry"class="org.springframework.batch.core.
[CA] configuration.support.MapJobRegistry" />

<bean id="jobExplorer" class="org.springframework.batch.core.explore.
[CA] support.JobExplorerFactoryBean">
  <property name="dataSource" ref="dataSource" />
</bean>
```

#A Declares job operator bean

The job operator has four dependencies: the job repository, job launcher, job registry, and job explorer. By now, we are used to seeing the job repository and the job launcher, as they're essential parts of the Spring Batch infrastructure. You need to declare the job registry and the job explorer only for specific tasks, and configuring the job operator is one.

As a bonus, the following configuration exposes the job operator to JMX. This saves you a round-trip to chapter 12.

```
<bean class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="com.manning.sbia:name=jobOperator"
           value-ref="jobOperator" />
    </map>
  </property>
</bean>
```

You can now explain to your administration team how to stop a job execution. But a member of the administration team might tell you that a job execution doesn't stop. The next sub-section goes behind the scenes to understand what happens when you request to stop a job execution.

UNDERSTANDING THE STOP MESSAGE

When we showed the job operator in action, you may have found this line intriguing:

```
boolean stopMessageSent = jobOperator.stop(executionId);
```

The job operator returns a boolean when you request to stop a job execution. This boolean value tells you whether the stop message has been sent successfully. A stop *message*? When you call the `stop` method on a job operator, there's no guaranty that the execution has immediately stopped after the call. Why? In Java, you cannot to stop code from executing immediately.

When does job execution stop after you request it? Let's imagine some business code is executing when you send the stop message. There are two possibilities:

1. The business code takes into account that the thread can be interrupted by checking `Thread.currentThread().isInterrupted()`. If the code detects the thread interruption, it can choose to end processing by throwing an exception or returning immediately. This means that the execution will stop almost immediately.
2. The business code doesn't deal with thread interruption. As soon as the business code finishes and Spring Batch gets control again, the framework stops the job execution. This means that the execution will stop only after the code finishes. If the code is in the middle of a long processing sequence, the execution can take a long time to stop.

Stopping while in a middle of chunk-oriented step shouldn't be a problem: Spring Batch drives all the processing in this case, so the execution should stop quickly (unless some custom reader, processor, or writer takes a very long time to execute.) But, if you write a custom tasklet whose processing is long, you should consider checking for thread interruption.

Understanding the stop message is a first step towards the developer's point of view, so let's now see how to stop a job execution from application code.

5.5.2 Stopping a job for the application developer

We saw that an administrator can use the job operator to stop a job execution, but sometimes stopping the execution from within the job itself is necessary. Imagine you're indexing your product catalog with a Spring Batch job. The online store application can work with some un-indexed products, but the job execution shouldn't overlap with periods of high activity, so it shouldn't run after eight in the morning. You can check the time in various places in the job and decide to stop the execution after eight.

The first way to stop execution is to throw an exception. This works all the time, unless you configured the job to skip some exceptions in a chunk-oriented step!

The second and preferred way to stop execution is to set a stop flag in the step execution object. To set this stop flag, call the method `StepExecution.setTerminateOnly()`, this is equivalent to sending a stop message. As soon as Spring Batch gets control of the processing, it will stop the job execution. The next topic to cover is how to get access to the `StepExecution` object from a job. Getting access to the `StepExecution` depends on whether you're working directly with a tasklet or in a chunk-oriented step. Let's study both cases now.

STOPPING FROM A TASKLET

A tasklet has direct access to the `StepExecution` through the step context, itself in the chunk context. Listing 5.10 shows a tasklet that processes items, checks a stop condition, and sets the stop flag accordingly. The stop condition could be any business decision, like the time restriction we mentioned previously.

Listing 5.10 Setting the stop flag from a tasklet

```
package com.manning.sbia.ch05.stop;

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 ProcessItemsTasklet implements Tasklet {

 @Override
 public RepeatStatus execute(StepContribution contribution,
    ChunkContext chunkContext) throws Exception {
  if(shouldStop()) {
    chunkContext.getStepContext()                          #A
      .getStepExecution().setTerminateOnly();              #A
  }
  processItem();
  if(moreItemsToProcess()) {
    return RepeatStatus.CONTINUABLE;
  } else {
    return RepeatStatus.FINISHED;
  }

 }

 (...)

}
```

#A Sets stop flag

Setting the stop flag in a tasklet is straightforward; let's now see how to do this in a chunk-oriented step.

STOPPING FROM A CHUNK-ORIENTED STEP

Remember how a chunk-oriented step works: Spring Batch drives the flow and lets you plug in your business logic or reuse off-the-shelf components to read, process or write items. If you look at the `ItemReader`, `ItemProcessor`, and `ItemWriter` interfaces, you won't see a `StepExecution`. We get access to the `StepExecution` to stop the execution using *listeners*.

> Note Not dealing with stopping a job in item readers, processors, and writers is a good thing. These components should focus on their processing, to enforce separation of concerns.

Chapter 10 covers listeners (or interceptors), but we're going to give you enough background here to use them for our job stopping purposes. The idea of a listener is to react to the lifecycle events of a step. We register a listener on a step and Spring Batch calls corresponding methods throughout the lifecycle of that step. We can use annotations or implement interfaces to register these methods. What are the lifecycle events we can listen for? There are a lot of them: step start, after each read, processed or written item, step end, and so on. Listing 5.11 shows a listener that keeps a reference to the `StepExecution` and checks a stopping condition after each read item. This listener uses annotations.

Listing 5.11 An annotated listener to stop a job execution

```
package com.manning.sbia.ch05.stop;

import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.annotation.AfterRead;
import org.springframework.batch.core.annotation.BeforeStep;
```

```
public class StopListener {

  private StepExecution stepExecution;

  @BeforeStep                                          #A
  public void beforeStep(                               #A
     StepExecution stepExecution) {                        #A
    this.stepExecution = stepExecution;                     #A
  }                                                  #A

  @AfterRead                                           #B
  public void afterRead() {                             #B
    if(stopConditionsMet()) {                            #B
      stepExecution.setTerminateOnly();                       #B
    }                                                  #B
  }                                                    #B

  (...)

}
```

#A Registers step execution
#B Sets stop flag if necessary

The real work is to implement the stopping condition, which is a business decision (the body of the `stopConditionsMet` method in our example.) Listing 5.12 shows how to register the listener on the chunk-oriented step.

Listing 5.12 Registering the stop listener on the step

```
<bean id="stopListener"                                  #A
    class="com.manning.sbia.ch05.stop.StopListener"             #A
/>                                                     #A

<batch:job id="importProductsJob">
  <batch:step id="importProductsStep">
    <batch:tasklet>
      <batch:chunk reader="reader" writer="writer" commit-interval="10" />
      <batch:listeners>                                #B
        <batch:listener ref="stopListener" />               #B
      </batch:listeners>                               #B
    </batch:tasklet>
  </batch:step>
</batch:job>
```

#A Declares listener bean
#B Registers listener

Note how the listener mechanism makes the stopping decision a crosscutting concern: no component in the step knows about stopping, only the dedicated listener.

This concludes the coverage of stopping a Spring Batch job. You saw how to stop an job execution from the operator's point of view. You configure a job operator bean that you can expose to JMX and call the appropriate sequence of methods to stop a specific job execution. Don't forget that stopping an execution is only a request message: your code must be aware of this message if you want the execution to stop quickly. As soon as soon Spring Batch gets control of the processing, it will do its best

to stop the execution gracefully. Finally, remember that you can choose to stop the execution from within your business code.

5.6 Summary

Launching Spring Batch jobs is easy, even if the length of this chapter doesn't reflect this simplicity. We indeed covered many scenarios, which are to us the most common scenarios you'll meet in batch systems. With Spring Batch, you can stick to the popular "cron + command line" scenario, by using either your own Java program or Spring Batch's generic command line runner. You can also choose to embed Spring Batch in a web application combined with a Java scheduler. Springs provides lightweight support for scheduling, but if you reach the limits of this feature, consider using Quartz.

We provided you with the following guidelines:

- The generic command line launcher + cron solution is good for jobs that don't run with a high frequency. You shouldn't use this solution when initializing the batch environment is costly and the batch job runs every 30 seconds for example.
- If you want to have your batch environment ready all the time, embed your Spring Batch environment in a web application.
- Once your batch environment is in a web application, also embed a Java scheduler to start your jobs. If the triggering event comes from an external system that doesn't have direct access to Spring Batch, use an HTTP request to trigger the execution.
- Imagine any launching system that suits your needs, the Spring Batch launching API is in Java, so you're only limited by the Java language and your imagination!
- Stopping a job execution uses a stop message. You should take into account this message in your code, but you can also count on Spring Batch to stop gracefully when it takes control of the flow again.

It's now time to go back to the heart of Spring Batch: chunk-oriented processing. The next three chapters cover the three corresponding phases of chunk processing: reading, processing, and writing.

8 Processing data

This chapter covers

- Writing business logic in a chunk-oriented step
- Processing items in a chunk-oriented step
- Transforming items
- Filtering items
- Validating items

The two previous chapters focused heavily on Spring Batch input and output: how to read and write data from various types of data stores. You learned that Spring Batch enforces best practices to optimize I/O and provides many ready-to-use components. This is important for batch applications because exchanging data between systems is common. Batch applications aren't limited to I/O; they also have business logic to carry on: enforcing business rules before sending items to a database, transforming data from a source representation to one expected by a target data store, and so on.

In Spring Batch applications, you embed this business logic in the *processing phase*: after you read an item but before you write it. Thanks to its chunk-oriented architecture, Spring Batch provides first-class support for this type of processing in a dedicated component – the item processor – that you insert between the item reader and the item writer. After explaining item processing and its configuration in Spring Batch, we'll see how to use item processors to modify, filter, and validate items. For validation, we'll examine two techniques: programmatic validation in Java and validation through configuration files using a validation language. Finally, we cover how to chain item processors following the composite design pattern. By the end of this chapter, you'll know exactly where and how to write the business logic for your batch application. You'll also learn about advanced topics, like the distinction between filtering and skipping items. Let's start with processing items in Spring Batch.

8.1 Processing items

Spring Batch provides a convenient way to handle a large amount of records: the chunk-oriented step. So far, we've covered the read and write phases of the chunk-oriented step; in this section, we'll see

how to add a processing phase. This processing phase is the perfect place to embed your application-specific business logic. The processing phase also avoids tangling your business code in the reading and writing phases (input and output). We'll see what kind of business logic the processing phase can handle, how to configure an item processor in a chunk-oriented step, and the item processor implementations delivered with Spring Batch.

8.1.1 Processing items in a chunk-oriented step

Recall that a chunk-oriented step includes a reading component (to read items one by one) and a writing component (to handle writing several items in one chunk). The two previous chapters covered how to read and write items from different kinds of data stores and in various formats. Spring Batch can insert an optional processing component between the reading and writing phase. This component – the item processor – will embed some business logic between reading and writing, like transforming or filtering items. Figure 8.1 illustrates where item processing takes place in a chunk-oriented step.

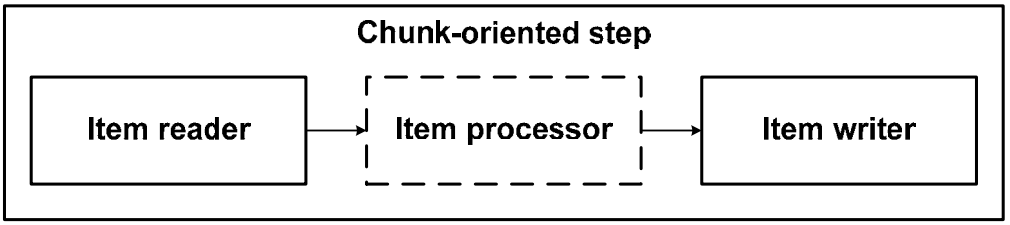

Figure 8.1 Spring Batch allows insertion of an optional processing phase between the reading and writing phase of a chunk-oriented step. The processing phase usually contains some business logic, implemented as an item processor.

When there's no processing phase in a chunk-oriented step, items read are sent as-is to the writer and Spring Batch takes care of aggregating items in chunks. Now, imagine that an application can't allow writing items as-is, because some kind of processing must be applied the items first. Do you recall our online store example? Let's add a new business requirement: We want to apply discounts to products before the job imports them in the online store database. To do so, we need to modify the products imported from the flat file in the item-processing phase. Let's learn more about item processing now.

8.1.2 Use cases for item processing

The processing phase is a good place for business logic. A common use case in Spring Batch is to use built-in readers and writers to deal with data stores – like flat files and databases – and to add an item processor to hold any custom business logic. Table 8.1 lists the categories of business logic that can take place in the item-processing phase.

Table 8.1 Categories of business logic that can take place in the item-processing phase

Category	Description
Transformation	The item processor transforms read items before sending them to the writer. The item processor can change the state of the read item, or create a new object. In the latter case, written items may not be of the same type as read items.
Filtering	The item processor decides whether to send each read item to the writer.

The processing phase is an interesting link between the reading and writing phase. It allows you to go beyond the simple "read an item – write that item" pattern. The remaining of this chapter examines the subtleties of item processing – with realistic use cases – in order to illustrate the many possibilities of this pattern. Let's start with the basic configuration of an item processor in Spring Batch.

8.1.3 Configuring an item processor

Spring Batch defines the item-processing contract with the `ItemProcessor` interface as follows:

```
package org.springframework.batch.item;

public interface ItemProcessor<I, O> {

  O process(I item) throws Exception;

}
```

The `ItemProcessor` interface uses two type arguments, `I` and `O`:

- Spring Batch passes a read item of type `I` to the `process` method. The type `I` must be compatible with the item reader type.
- The `process` method returns an item of type `O`, which Spring Batch will in turn send to the item writer also of a type compatible with `O`.

You define the concrete types `I` and `O` in your `ItemProcessor` implementation. If the `process` method returns `null`, Spring Batch will not send the item to the writer, as defined by the filtering contract. Listing 8.1 shows how to implement a filtering `ItemProcessor`.

Listing 8.1 Implementation of a filtering item processor

```
package com.manning.sbia.ch08;

import org.apache.commons.lang.math.NumberUtils;
import org.springframework.batch.item.ItemProcessor;
import com.manning.sbia.ch02.domain.Product;

public class FilteringProductItemProcessor implements                  #A
               ItemProcessor<Product, Product> {                        #A

  @Override
  public Product process(Product item) throws Exception {
    return needsToBeFiltered(item) ? null : item;
  }

  private boolean needsToBeFiltered(Product item) {                     #B
    String id = item.getId();                                           #B
```

```
      String lastDigit = id.substring(                                  #B
        id.length()-1, id.length());                                    #B
      if(NumberUtils.isDigits(lastDigit)) {                             #B
        return NumberUtils.toInt(lastDigit) % 2 == 1;                   #B
      } else {                                                          #B
        return false;                                                   #B
      }                                                                 #B
    }                                                                   #B

}
```

#A Receives and returns the same type or objects
#B Implements filtering logic

Our `ItemProcessor` implementation has the following characteristics:

- No transformation – the processor receives a `Product` object and returns a `Product` object. Therefore, the `I` and `O` type arguments of the `FilteringProductItemProcessor` class both use the `Product` class.
- Filtering – depending on the result of the `needsToBeFiltered` method, Spring Batch will send the item to the writer or discard it. Our filtering logic is a simple: if an item ID's last character is an even digit, the filter will accept the item.

Our `ItemProcessor` example is not useful beyond showing you how to configure item processing in a chunk-oriented step. Listing 8.2 shows how to configure this item processor.

Listing 8.2 Configuring an item processor in a chunk-oriented step

```
<batch:job id="readWriteJob">
  <batch:step id="readWriteStep">
    <batch:tasklet>
      <batch:chunk
        reader="reader"
        processor="processor"                                           #A
        writer="writer"
        commit-interval="100" />
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="processor"                                                    #B
      class="com.manning.sbia.ch08.                                     #B
[CA] FilteringProductItemProcessor" />                                  #B

<bean id="reader" (...) >
  (...)
</bean>

<bean id="writer" (...)>
  (...)
</bean>
```

#A Adds item processor to chunk-oriented step
#B Defines item processor bean

Adding an item processor is straightforward with the Spring Framework and Spring Batch XML: you write a Spring bean that implements the `ItemProcessor` interface and then refer to it in your chunk-oriented step configuration with the `processor` attribute of a `chunk` element.

Now that you know the basics of item processing in Spring Batch, let's see what the framework offers in term of ready-to-use `ItemProcessor` implementations.

8.1.4 Item processor implementations

As you've seen in the previous section, implementing an `ItemProcessor` is simple. This is what you end up doing most of the time to implement business logic. Nevertheless, Spring Batch provides implementations of `ItemProcessors` that can come in handy; table 8.2 lists these implementations.

Table 8.2 Spring Batch implementations of ItemProcessor

Implementation class	Description
`ItemProcessorAdapter`	Invokes a custom method on a delegate POJO, which isn't required to implement `ItemProcessor`.
`ValidatingItemProcessor`	Delegates filtering logic to a `Validator` object.
`CompositeItemProcessor`	Delegates processing to a chain of `ItemProcessors`.

We won't go further in the description of these `ItemProcessor` implementations as we'll find some opportunities to illustrate their use in the remaining of the chapter. Now that you have the big picture of item processing with Spring Batch, let's dive into the details of transforming items.

8.2 Transforming items

Transforming read items and then writing them out is the typical use case for an item processor. In Spring Batch, we distinguish two kinds of transformation: changing the state of the read item and producing a new object based on the read item. In the later case, the object the processor returns can be of a different type than the incoming item.

We illustrate both kinds of transformation with our online store application. Imagine that the application is successful and that other companies ask ACME to add their products to the online catalog; for this service, ACME will take a cut of each partners' product sold. For the application, this means importing products from different files: a file for its own catalog and a file for each partner catalog. With this use case in mind, let's first explore transforming the state of read items.

8.2.1 Changing the state of read items

ACME needs to import a flat file for each partner's product catalog. In our scenario, the model of the ACME product and of each partner product is similar, nevertheless there are some modifications to make to all partner's imported products before writing them to the database. These modifications will require some custom business logic, so we embed this logic in a dedicated application component. We then use this component from an item processor.

INTRODUCING THE USE CASE

The model of the ACME product and of each partner product is similar, but each partner maintains its own product identifiers. ACME needs to map identifiers of partner products to its own identifiers in order

to avoid collisions. Figure 8.2 shows that the item reader, processor, and writer of the chunk-oriented step all use the same type of object.

Figure 8.2 In the processing phase of a chunk-oriented step we can choose to only change the state of read items. In this case, the item reader, processor, and writer all use the same type of object (illustrated by the small squares in the diagram).

The custom mapping between partner IDs and online store IDs takes place in our `PartnerIdMapper` class as shown in listing 8.3.

Listing 8.3 Mapping partner IDs with store IDs in a business component

```
package com.manning.sbia.ch08;

import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import com.manning.sbia.ch02.domain.Product;

public class PartnerIdMapper {

  private static final String SQL_SELECT_STORE_PRODUCT_ID =
    "select store_product_id from partner_mapping " +
    "where partner_id = ? and partner_product_id = ?";

  private String partnerId;

  private JdbcTemplate jdbcTemplate;

  public Product map(Product partnerProduct) {
    String storeProductId=jdbcTemplate.queryForObject(                #A
      SQL_SELECT_STORE_PRODUCT_ID,                                    #A
      String.class,                                                   #A
      partnerId, partnerProduct.getId()                               #A
    );                                                                #A
    partnerProduct.setId(storeProductId);                             #B
    return partnerProduct;
  }

  public void setPartnerId(String partnerId) {
    this.partnerId = partnerId;
  }

  public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
  }

}
```

#A Finds store product ID in mapping table
#B Modifies incoming product

To perform the product ID mapping, we search the product ID for the online store in a mapping database table, using the partner identifier and the identifier of the product in the partner namespace as the criteria. In an alternative implementation, the `PartnerIdMapper` class could generate the ID on the fly and store it if it did not find it in the mapping table.

> Note The `PartnerIdMapper` class is a POJO: it does not depend on the Spring Batch API but it does use the JdbcTemplate class from the Spring JDBC Core package. This is important because this class implements our business logic and we do not want to couple it tightly to the Spring Batch infrastructure.

Let's now plug our business component into Spring Batch.

IMPLEMENTING A CUSTOM ITEM PROCESSOR

We implement a dedicated `ItemProcessor` with a plug-in slot for a `PartnerIdMapper`. The following snippet shows our implementation:

```
package com.manning.sbia.ch08;

import org.springframework.batch.item.ItemProcessor;
import com.manning.sbia.ch02.domain.Product;

public class PartnerIdItemProcessor implements
    ItemProcessor<Product, Product> {

  private PartnerIdMapper mapper;

  @Override
  public Product process(Product item) throws Exception {
    return mapper.map(item);                                    #A
  }

  public void setMapper(PartnerIdMapper mapper) {
    this.mapper = mapper;
  }

}
```

#A Delegates processing to business component

What we need now is to wire these components together in the configuration of a chunk-oriented step, as shown in listing 8.4.

Listing 8.4 Configuring the dedicated item processor to map product IDs

```
<batch:job id="readWriteJob">
  <batch:step id="readWriteStep">
    <batch:tasklet>
      <batch:chunk reader="reader"
                   processor="processor"                        #A
                   writer="writer"
                   commit-interval="100" />
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="processor"
      class="com.manning.sbia.ch08.PartnerIdItemProcessor">
  <property name="mapper" ref="partnerIdMapper" />              #B
```

```
</bean>

<bean id="partnerIdMapper"                                              #C
      class="com.manning.sbia.ch08.PartnerIdMapper">                    #C
  <property name="partnerId" value="PARTNER1" />                        #C
  <property name="dataSource" ref="dataSource" />                       #C
</bean>                                                                 #C

<bean id="reader" (...)>                                                #D
  (...)                                                                 #D
</bean>                                                                 #D
                                                                        #D
<bean id="writer" (...)>                                                #D
  (...)                                                                 #D
</bean>                                                                 #D
```

#A Sets item processor in step
#B Injects ID mapper in item processor
#C Declares ID mapper bean
#D Defines item reader and writer

That's it! You configured processing that converts the IDs of the incoming products into the IDs that the online store uses. We managed to isolate the business logic from Spring Batch –separation of concerns – but we had to implement a custom `ItemProcessor` to call our business logic. Next, we see a way to achieve the same goal without this extra custom class.

PLUGGING IN AN EXISTING COMPONENT WITH THE ITEMPROCESSORADAPTER

Sometimes an existing business component is similar to a Spring Batch interface like `ItemProcessor`, but as it doesn't implement the interface, the framework is unable to call it directly. That's why we implemented the `PartnerIdItemProcessor` class in the previous section: to be able to call our business code from Spring Batch. It worked nicely but isn't it a shame to implement a dedicated class to delegate a call? Fortunately, Spring Batch provides the `ItemProcessorAdapter` class that you can configure to call any method on a POJO. Using the `ItemProcessorAdapter` class removes the need to implement a class like `PartnerIdItemProcessor`. All you end up doing is a bit of Spring configuration. Listing 8.5 shows how to use the `ItemProcessorAdapter` to call the `PartnerIdMapper` without a custom `ItemProcessor`.

Listing 8.5 Using the ItemProcessorAdapter to plug in an existing Spring bean

```
<batch:job id="readWriteJob">
  <batch:step id="readWriteStep">
    <batch:tasklet>
      <batch:chunk reader="reader"
                   processor="processor"                                #A
                   writer="writer"
                   commit-interval="100" />
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="processor"
      class="org.springframework.batch.item.adapter.ItemProcessorAdapter">
  <property name="targetObject" ref="partnerIdMapper" />                #B
  <property name="targetMethod" value="map" />                          #B
</bean>
```

```
<bean id="partnerIdMapper"                                          #C
      class="com.manning.sbia.ch08.PartnerIdMapper">                #C
  <property name="partnerId" value="PARTNER1" />                    #C
  <property name="dataSource" ref="dataSource" />                   #C
</bean>                                                             #C

<bean id="reader" (...)>                                            #D
  (...)                                                             #D
</bean>                                                             #D
                                                                    #D
<bean id="writer" (...)>                                            #D
  (...)                                                             #D
</bean>                                                             #D
```

#A Sets item processor in step
#B Sets target bean and method to call
#C Declares ID mapper bean
#D Defines item reader and writer

Using an `ItemProcessorAdapter` should remove the need to implement a dedicated `ItemProcessor`. This reminds us that it's a best practice to have your business logic implemented in POJOs. The `ItemProcessorAdapter` class helps in reducing the proliferation of classes if you often need to use a processing phase.

The `ItemProcessorAdapter` class has a couple of drawbacks though: it's not as type-safe as a dedicated `ItemProcessor` class, and you can make typos on the target method name when configuring it in XML. The good news is that the `ItemProcessorAdapter` checks its configuration when it's created; you get an exception when the Spring application context starts and also at runtime.

We're done looking at a processing phase that changes the state of read items. We use such processing when read items are of the same type as written items, but need some sort modification, like the identifier conversion for imported products. The next section covers a processing phase that produces a different type of object from the read item.

8.2.2 Producing new objects from read items

As ACME finds more and more partners, it will have to deal with different product lines. ACME must then deal with mismatches between the model of partner products and its own product model. These model differences make the importing job more complex. We still base the import on an input flat file, but ACME needs a processing phase to transform the partner's products into products that fits in the online store database.

INTRODUCING THE USE CASE

The processing phase of our chunk-oriented step will transform `PartnerProduct` objects read by the `ItemReader` into `Product` objects that the `ItemWriter` will write into the online store database. This is a case where the reader, the processor, and the writer don't manipulate the same kind of objects at every step, as shown in figure 8.3.

Figure 8.3 The item processor of a chunk-oriented step can produce objects of a different type (represented by circles) than the read items (represented by squares). The item writer then receives and handles these new objects.

The logic to transform a `PartnerProduct` object into an ACME `Product` object takes place in a dedicated business component – the `PartnerProductMapper` class – that implements the `PartnerProductMapper` interface:

```
package com.manning.sbia.ch08;

import com.manning.sbia.ch02.domain.Product;

public interface PartnerProductMapper {

    Product map(PartnerProduct partnerProduct);

}
```

We don't show an implementation of the `PartnerProductMapper` interface because it's all business logic not directly related to Spring Batch, and therefore not relevant to our presentation of Spring Batch. You can find a simple implementation in the source code for this book. What we need to do now is to plug this business logic in a Spring Batch job.

What could the partner product mapper do?

Here's an example of what a `PartnerProductMapper` implementation could do in a real-world online store application. Up to now – for simplicity's sake – we've used a static structure for the products of our online store application. Most of the time, online store applications don't have a static structure for the products in their catalog: they use a meta-model configured with the structure of the products and a generic engine that uses this meta-model to display products dynamically. For example, a meta-model for products in a "book" category could have fields for author, title, publication date, and so on. We could imagine that the ACME online application uses such a meta-model but that the partners do not. The goal of the `PartnerProductMapper` would be to map statically structured partner products (from input flat files) to the online store's products model. Such a mapper would rely heavily on the meta-model to do its job.

Let's plug in our `PartnerProductMapper` in an item processor.

IMPLEMENTING A CUSTOM ITEM PROCESSOR

Listing 8.6 shows a custom `ItemProcessor` called `PartnerProductItemProcessor` that calls our `PartnerProductMapper` class. This item processor is Spring Batch specific as it implements `ItemProcessor` and delegates processing to its `PartnerProductMapper`, itself a business POJO.

Listing 8.6 A dedicated item processor to call the partner product mapper

```
package com.manning.sbia.ch08;

import org.springframework.batch.item.ItemProcessor;
import com.manning.sbia.ch02.domain.Product;

public class PartnerProductItemProcessor implements
    ItemProcessor<PartnerProduct, Product> {

  private PartnerProductMapper mapper;

  @Override
  public Product process(PartnerProduct item) throws Exception {
    return mapper.map(item);                                          #A
  }

  public void setMapper(PartnerProductMapper mapper) {
    this.mapper = mapper;
  }

}
```

#A Delegates processing to business component

Note that the actual argument types of the generic `ItemProcessor` interface now take two different types: `PartnerProduct` (for the input type) and `Product` (for the output type.) Now that we have our `ItemProcessor`, let's see its configuration in listing 8.7.

Listing 8.7 Configuring an item processor to map partner products to store products

```
<batch:job id="readWriteJob">
  <batch:step id="readWriteStep">
    <batch:tasklet>
      <batch:chunk reader="reader"
                   processor="processor"                              #A
                   writer="writer"
                   commit-interval="100" />
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="processor"
      class="com.manning.sbia.ch08.PartnerProductItemProcessor">
  <property name="mapper" ref="partnerProductMapper" />               #B
</bean>

<bean id="partnerProductMapper"                                       #C
      class="com.manning.sbia.ch08.                                   #C
[CA] SimplePartnerProductMapper" />                                   #C

<bean id="reader" (...)>                                              #D
  (...)                                                               #D
</bean>                                                               #D
                                                                      #D
<bean id="writer" (...)>                                              #D
  (...)                                                               #D
</bean>                                                               #D
```

#A Sets item processor in step

#B Injects product mapper in item processor

#C Declares product mapper bean
#D Defines item reader and writer

Thanks to the product mapping that takes place during the processing phase of our step, we're able to transform the product representation of a partner to the product representation expected by the online store application. If you need to perform a different conversion for another partner, you only need to implement another item processor, you'd re-use the item reader and writer.

> Note As we did in the "Changing the state of read items" section, you can use the `ItemProcessorAdapter` class to plug in the business component (`PartnerProductMapper`) in the processing phase. This would avoid the need for the `PartnerProductItemProcessor` class.

Before ending this section on using the item-processing phase to modify or transform read items, we're going to see how to use an `ItemProcessor` to implement a common pattern in batch applications: the "driving query" pattern.

8.2.3 Implementing the driving query pattern with an item processor

The driving query pattern is an optimization pattern used with databases. The pattern consists of two parts:

- Execute one query to load the identifiers of the items you want to work with. This first query – the driving query – returns N identifiers
- Execute queries to retrieve a database row for each item. In total, N additional queries load the corresponding objects.

This seems counter-intuitive but using this pattern can end up being faster than loading the whole content of each object in one single query. How is that possible?

Some database engines tend to use pessimistic locking strategies on large cursor-based result sets. This can lead to poor performance or even deadlocks if applications other than the batch application access the same tables. The trick is to use a driving query to select the identifiers and then load complete objects one by one. This execution pattern prevents the database from handling large datasets.

Spring Batch can easily match and implement the driving query pattern in a chunk-oriented step:

- An `ItemReader` executes the driving query
- An `ItemProcessor` receives the identifiers and loads the objects
- The loaded objects go to the `ItemWriter`

Figure 8.4 illustrates the driving query pattern in Spring Batch.

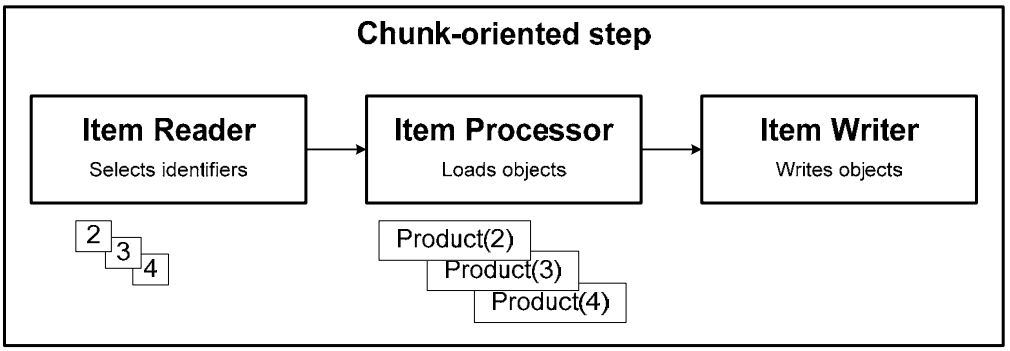

Figure 8.4 The driving query pattern implemented in Spring Batch. The item reader executes the driving query. The item processor receives the identifiers and loads the objects. The item writer then receives these objects to for example write a file, update the database or an index.

Let's use the driving query pattern in our online application. Imagine the application features a search engine whose index needs to be updated from time to time (a complete re-indexing is rare, because it would take too long). The indexing batch job consists of selecting recently updated products and updating the index accordingly. As the online store is running during the indexing, we shouldn't load large datasets to avoid locking overhead; therefore, the driving query pattern is a good match. Let's see how to implement this with Spring Batch!

EXECUTING THE DRIVING QUERY WITH A JDBC ITEM READER

We use a Spring Batch cursor-based JDBC item reader to retrieve the product identifiers. The product table contains an `update_timestamp` column updated each time a product row changes. Who performs the update? The application can perform the update, a database trigger, or a persistence layer like Hibernate. We use the `update_timestamp` column to select the products that the database must index. Listing 8.x shows how to configure a `JdbcCursorItemReader` to execute the driving query for our use case.

Listing 8.8 Configuring an item reader to execute the driving query

```
<bean id="reader"
      class="org.springframework.batch.item.database.JdbcCursorItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="sql"
            value="select id from product where update_timestamp &gt; ?" />
  <property name="preparedStatementSetter">                              #1
    <bean class="org.springframework.jdbc.core.                          #1
[CA] ArgPreparedStatementSetter"                                         #1
          scope="step">                                                  #1
      <constructor-arg                                                   #1
      value="#{jobParameters['updateTimestampBound']}"                   #1
      />                                                                 #1
    </bean>                                                              #1
  </property>                                                            #1
  <property name="rowMapper">                                            #2
    <bean class="org.springframework.jdbc.core.                          #2
[CA] SingleColumnRowMapper">                                             #2
```

```
        <constructor-arg value="java.lang.String" />                    #2
      </bean>                                                           #2
    </property>                                                         #2
  </bean>
```

#1 Assigns parameter to SQL query
#2 Returns a String for each row

If you are not familiar with the `JdbcCursorItemReader` class, please see chapter 6. We use the `JdbcCursorItemReader` `sql` property to set the SQL query to execute. This query selects the identifiers of the products which have been updated after a given date (by using the `where > ?` clause). To pass a parameter to the query, we set the `preparedStatemenSetter` property at `#1` with an `ArgPreparedStatementSetter` (this is a class from the Spring Framework). We use the Spring Expression Language to get the date query parameter from the job parameters. To retrieve the identifiers from the JDBC result set, we use the Spring class `SingleColumnRowMapper` at `#2`.

That's it! We configured our item reader to execute the driving query. Note that we didn't write any Java code: we configured only existing components provided by Spring Batch and the Spring Framework. Next, let's see how to load the products from their identifiers within an item processor.

LOADING ITEMS IN AN ITEM PROCESSOR

We need to load a product based on its identifier. This is a simple operation and a data access object used in the online store application already implements this feature. Listing 8.9 shows the implementation of this data access object.

Listing 8.9 Implementing a data access object to load a product from its ID

```
package com.manning.sbia.ch08;

import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import com.manning.sbia.ch02.domain.Product;

public class JdbcProductDao implements ProductDao {

  private static final String SQL_SELECT_PRODUCT =
    "select id,name,description,price " +
    "from product where id = ?";

  private JdbcTemplate jdbcTemplate;

  private RowMapper<Product> rowMapper = new ProductRowMapper();

  @Override
  public Product load(String productId) {
    return jdbcTemplate.queryForObject(                                 #A
      SQL_SELECT_PRODUCT, rowMapper, productId                          #A
    );                                                                  #A
  }

  public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
  }
```

```
}
```

#A Loads entire object from database

We use a `ProductRowMapper` to map a JDBC result set to `Product` objects. Remember that the `RowMapper` is a Spring interface. You can use `RowMapper` implementations in a JDBC-based data access layer for your online applications. You can also use a `RowMapper` with a Spring Batch JDBC-based item reader.

What we need to do now is to connect our data access logic with Spring Batch. This is what the item processor in listing 8.10 does.

Listing 8.10 Implementing an item processor to call the data access object

```
package com.manning.sbia.ch08;

import org.springframework.batch.item.ItemProcessor;
import com.manning.sbia.ch02.domain.Product;

public class IdToProductItemProcessor implements
    ItemProcessor<String, Product> {

  private ProductDao productDao;

  @Override
  public Product process(String productId) throws Exception {
    return productDao.load(productId);
  }

  public void setProductDao(ProductDao productDao) {
    this.productDao = productDao;
  }

}
```

The class `IdToProductItemProcessor` delegates product loading to the product DAO. To avoid writing a dedicated class, we could have used an `ItemProcessorAdapter`, but with the `IdToProductItemProcessor` class, the input and output types are easier to picture: `String` for the identifiers returned by the driving query (input), and `Product` instances loaded by the item processor (output.)

CONFIGURING A CHUNK-ORIENTED STEP FOR THE DRIVING QUERY PATTERN

The configuration of a chunk-oriented step using the driving query pattern is like any other chunk-oriented step, except that it needs to have an item processor set. Listing 8.11 shows this configuration (we elided the reader and writer details.)

Listing 8.11 Configuring a driving query

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

```
  </batch:job>

  <bean id="reader"
        class="org.springframework.batch.item.database.JdbcCursorItemReader">
    (...)
  </bean>

  <bean id="processor"
        class="com.manning.sbia.ch08.IdToProductItemProcessor">
    <property name="productDao" ref="productDao" />                    #A
  </bean>

  <bean id="productDao"                                                #B
        class="com.manning.sbia.ch08.JdbcProductDao">                  #B
    <property name="dataSource" ref="dataSource" />                    #B
  </bean>                                                              #B

  <bean id="writer" (...)>
    (...)
  </bean>
```

#A Injects product DAO in item processor
#B Declares product DAO bean

The implementation of the driving query pattern with Spring Batch ends this section. We covered how to use the processing phase of a chunk-oriented step as a way to modify read items and create new items. The next section covers how to use the processing phase as a way to filter read items before sending them to an item writer.

8.3 Filtering and validating items

The processing phase of a chunk-oriented step doesn't only modify read items; it can also filter them. Imagine reading products from a flat file, some belong in the database and some don't. For example, some products don't belong to any of the categories of items sold in the store; some products are already in the database, and so on. You can use an item processor to decide whether to send a read item to the item writer, as shown in figure 8.5.

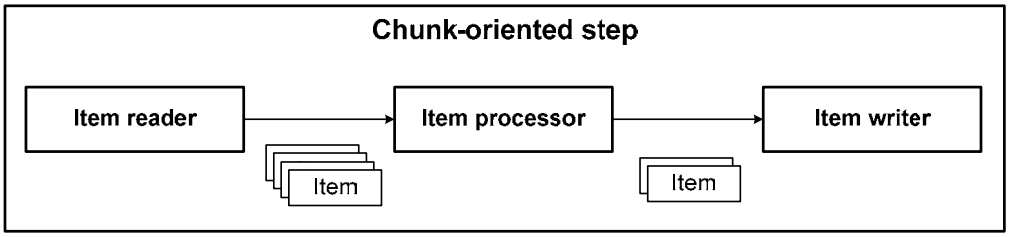

Figure 8.5 An item processor filters read items. It implements logic to decide whether to send a read item to the item writer.

We're going to cover how to implement a typical filtering item processor and how to filter using validation. We'll implement programmatic validation but we'll also use declarative validation using integration between Spring Batch and the Spring Modules project. First, let's learn more about the filtering contract in the item-processing phase.

8.3.1 Filtering in the item processing phase

The basic contract for filtering in an item processor is simple: if the item processor's `process` method returns `null`, the read item will not go to the item writer. This defines the main contract, but there are subtleties, let's look at the filtering rules for item processors:

- If the `process` method returns `null`, Spring Batch filters out the item, it will not go to the item writer.
- Filtering is different from skipping.
- An exception thrown by an item processor will result in a skip (you configure the skip strategy accordingly.)

The basic contract for filtering is clear, but we must point out the distinction between filtering and skipping.

> Filtering - means that Spring Batch shouldn't write a given record. For example, the item writer cannot handle a record.
>
> Skipping – means that a given record is invalid. For example, the format a phone number is invalid.
>
> Note The job repository stores the number of filtered items for each chunk-oriented step execution. You can easily lookup this information using a tool like Spring Batch Admin or by consulting the corresponding database table.

The last detail of the filtering contract we need to examine is that an item processor can filter items by returning `null` for some items but it can also *modify* read items, like any other item processor. You shouldn't mix filtering and transformation in a same item processor (separation of concerns) but it is your right to do so!

> **Best practice: separate filtering and transformation**
>
> If your application needs to both filter items and transform items, then follow the separation of concerns pattern by using two item processors: one to filter and one to transform.

Now that you know all about the filtering contract, let's see how to implement a filtering item processor.

8.3.2 Implementing a filtering item processor

Let's look back at our import product job from chapter 2 and see in which circumstances it could use filtering. Remember that this job consists in reading a flat file containing product records and creating or updating the database accordingly. We get into trouble if we execute the import job while the online store application hits the database: updating products from the job locks database rows and makes the online store less responsive. Nevertheless, we want the database to be as up-to-date as possible. A good compromise is to read the flat file, create new product records, but discard updating existing products. We update existing products later, in a separate job, when there's less traffic in the online store.

We meet this requirement by inserting a filtering item processor between the reading of the flat file and the writing to the database. This item processor checks the existence of the record in the database and discards it if it already exists. Figure 8.6 illustrates how the import products job works with a filtering phase.

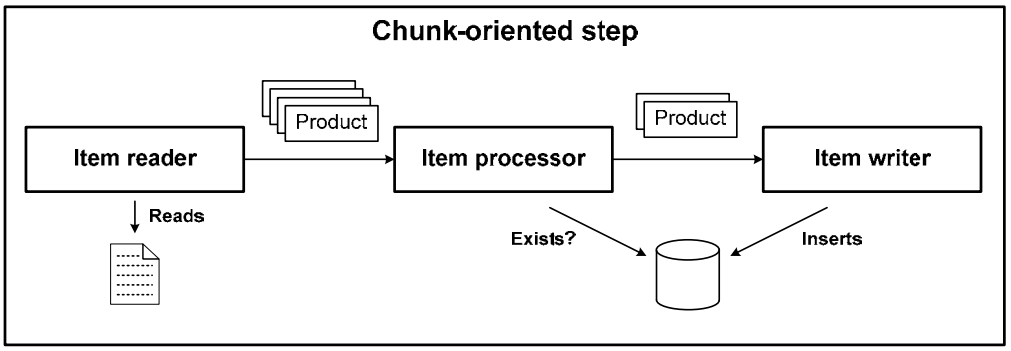

Figure 8.6 Our filtering item processor discards products that are already in the database. This item writer only inserts new records and doesn't interfere with the online application. A different job updates existing records, when there's less traffic in the online store.

Listing 8.12 shows the implementation of the filtering item processor.

Listing 8.12 Filtering existing products with an item processor

```
package com.manning.sbia.ch08;

import javax.sql.DataSource;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.jdbc.core.JdbcTemplate;
import com.manning.sbia.ch02.domain.Product;

public class ExistingProductFilterItemProcessor implements
    ItemProcessor<Product, Product> {

  private static final String SQL_COUNT_PRODUCT =
    "select count(1) from product where id = ?";

  private JdbcTemplate jdbcTemplate;

  @Override
  public Product process(Product item) throws Exception {
    return needsToBeFiltered(item) ? null : item;                    #A
  }

  private boolean needsToBeFiltered(Product item) {
    return jdbcTemplate.queryForInt(                                 #B
      SQL_COUNT_PRODUCT, item.getId()) != 0;                         #B
  }

  public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
  }
```

```
}
```

#A Filters existing product records
#B Checks product existence in database

Note A more advanced version of the filtering item processor could let record updates pass through in a given time window, like between two and four o'clock in the morning, when there isn't much activity in the online store. This would make the filtering more dynamic and could avoid the necessity of having two distinct jobs (one for inserts only and one for inserts and updates).

Listing 8.13 shows the configuration of the import products job with the filtering item processor.

Listing 8.13 Configuring the filtering item processor

```
<batch:job id="readWriteJob">
  <batch:step id="readWriteStep">
    <batch:tasklet>
      <batch:chunk reader="reader"
                   processor="processor"                              #A
                   writer="writer"
                   commit-interval="100" />
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  (...)
</bean>

<bean id="processor" class="com.manning.sbia.ch08.                   #B
[CA] ExistingProductFilterItemProcessor">                            #B
  <property name="dataSource" ref="dataSource" />                    #B
</bean>                                                              #B

<bean id="writer"
      class="com.manning.sbia.ch02.batch.ProductJdbcItemWriter">
  <constructor-arg ref="dataSource" />
</bean>
```

#A Sets filtering item processor in step
#B Declares filtering item processor bean

The item processor we implemented is a typical case of using the item processor phase as a filter. The item processor receives valid items from the item reader and decides which items to pass on to the item writer. The item processor effectively filters out the other items.

Let's now see another case where we can use an item processing to prevent read items from reaching the item writer: validation.

8.3.3 Validating items

As validation is business logic, the standard location to enforce validation rules is in the item-processing phase of a chunk-oriented step. A common practice in Spring Batch is for an item processor to perform

validation checks on read items and decide whether to send the item to the item writer. As an example, we'll see how to validate the price of imported products and check that prices are not negative numbers (products with a negative price shouldn't reach the database - we don't want to credit our customers!) Should we consider an item that fails the validation check filtered or skipped? Skipping is semantically closer to a validation failure, but this remains questionable and the business requirements usually lead us to the correct answer.

A validation failure should lead to a skipped or filtered item, but what you care about is that the item writer does not receive the item in question. Remember that the corresponding step execution metadata stored in the job repository are distinct (skip and filter count) and this distinction can be relevant for some use cases. If you want to enforce validation rules in your item processor, use the following semantics for validation failure in the item processor's `process` method:

- If validation means skip – throw a runtime exception
- If validation means filter – return `null`

What kind of validation can an item processor perform? You can do almost anything: state validation of the read object, consistency check with other data, and so forth. In the import products job, for example, we can check that the price of products from the flat file is positive. A well-formatted negative price would pass the reading phase (no parsing exception) but we shouldn't write the product to the database. The job of the item processor is to enforce this check and discard invalid products.

We can implement an item processor corresponding to this example and follow the semantics outlined above, but Spring Batch already provides a class called `ValidatingItemProcessor` to handle this task. Next, we study the `ValidatingItemProcessor` class by first coding the validation rules in a dedicated validator class. Then, we see how to use declarative validation to achieve the same goal.

VALIDATION WITH A VALIDATINGITEMPROCESSOR

The Spring Batch class `ValidatingItemProcessor` has two interesting characteristics:

- It delegates validation to an implementation of the Spring Batch `Validator` interface.
- It has a `filter` boolean property that can be set to `false` to throw an exception (skip) or `true` to return `null` (filter) if the validation fails. The default value is `false` (skip).

By using the `ValidatingItemProcessor` class, you can embed your validation rules in dedicated `Validator` implementations (which you can re-use) and choose your validation semantics by setting the `filter` property.

The Spring Batch `Validator` interface is:

```
package org.springframework.batch.item.validator;

public interface Validator<T> {

     void validate(T value) throws ValidationException;
}
```

When you decide to use the `ValidatingItemProcessor` class, you can either code your validation logic in `Validator` implementations or create a `Validator` bridge to a full-blown validation framework. We illustrate both next.

VALIDATION WITH A CUSTOM VALIDATOR

Let's say we want to check that a product doesn't have a negative price. The following snippet shows how this to implement this feature as a `Validator`:

```
package com.manning.sbia.ch08.validation;

import java.math.BigDecimal;
import org.springframework.batch.item.validator.ValidationException;
import org.springframework.batch.item.validator.Validator;
import com.manning.sbia.ch02.domain.Product;

public class ProductValidator implements Validator<Product> {

  @Override
  public void validate(Product product) throws ValidationException {
    if(BigDecimal.ZERO.compareTo(product.getPrice()) >= 0) {
      throw new ValidationException("Product price cannot be negative!");
    }
  }

}
```

This validator isn't rocket science, but as you configure it with Spring, it benefits from the ability to use dependency injection to, for example, access the database through a Spring JDBC template. The configuration for this validating item processor example has some interesting aspects, so let's examine listing 8.14.

Listing 8.14 Configuring a validating item processor

```
<batch:job id="readWriteJob">
  <batch:step id="readWriteStep">
    <batch:tasklet>
      <batch:chunk reader="reader"
                   processor="processor"
                   writer="writer"
                   commit-interval="100"
                   skip-limit="5">
        <batch:skippable-exception-classes>                              #A
          <batch:include                                                 #A
            class="org.springframework.batch.item.                       #A
[CA] validator.ValidationException"/>                                    #A
        </batch:skippable-exception-classes>                             #A
      </batch:chunk>
    </batch:tasklet>
   </batch:step>
</batch:job>

<bean id="processor" class="org.springframework.batch.item.validator.
[CA] ValidatingItemProcessor">
  <property name="filter" value="false" />                               #B
  <property name="validator">
    <bean class="com.manning.sbia.ch08.validation.ProductValidator" />
  </property>
</bean>

<bean id="reader" (...) >
  (...)
</bean>
```

```
<bean id="writer" (...)>
  (...)
</bean>
```

#A Skips validation exceptions
#B Re-throws validation exceptions to enforce skipping

Most of this configuration isn't elaborate: an XML `chunk` element for a chunk-oriented step, positioning of the item processor between the reader and the writer, and injection of our product validator in the `ValidatingItemProcessor`.

As we set the `filter` property of the validating item processor to `false` -- this is the default value but we wanted to make this example explicit -- the item processor will re-throw any `ValidationException` thrown by its validator.

This implies the configuration of a skip strategy if we don't want to fail the whole job execution in case of a validation failure. Our skip configuration consists in setting a skip limit and skipping `ValidationExceptions`.

If we were only to *filter* products that have a negative price, we would set the filter property of the `ValidatingItemProcessor` to `true` and we wouldn't need any skip configuration.

Implementing our own validator gave us the occasion to understand how Spring Batch handles validation. Writing dedicated validator classes can be overkill and result in overall code bloat. An alternative is to make the validation *declarative*: instead of coding the validation in Java, you implement it with a dedicated validation language in the configuration file.

VALIDATION WITH THE VALANG VALIDATOR FROM SPRING MODULES

The Spring Modules project provides a simple yet powerful validation language: Valang (for *Va*-lidation *Lang*-uage). You can easily integrate Valang with Spring Batch to write your validation rules without Java code. For example, to verify that the product price isn't negative, you write the following rule in Valang (assuming the evaluation context is a `Product` object):

```
{ price : ? >= 0 : 'Product price cannot be negative!' }
```

Valang has a rich syntax to create validation expressions; we won't cover this syntax here, as our point is to show how to integrate Valang rules within a validating item processor[2].

As Valang is not Java code, we use the Spring configuration to implement validating the product price, as shown in listing 8.15.

Listing 8.15 Embedding validation logic in the configuration with Valang

```
<batch:job id="readWriteJob">
  <batch:step id="readWriteStep">
    <batch:tasklet>
      <batch:chunk reader="reader"
                   processor="processor"
                   writer="writer"
                   commit-interval="100"
                   skip-limit="5">
        <batch:skippable-exception-classes>
          <batch:include
```

[2] You can learn more about the Valang syntax at https://springmodules.dev.java.net/docs/reference/0.9/html/validation.html.

```
            class="org.springframework.batch.item.
[CA] validator.ValidationException"/>
        </batch:skippable-exception-classes>
      </batch:chunk>
    </batch:tasklet>
   </batch:step>
</batch:job>

<bean id="processor"class="org.springframework.batch.item.validator.
[CA] ValidatingItemProcessor">
  <property name="filter" value="false" />
  <property name="validator" ref="validator" />
</bean>

<bean id="validator" class="org.springframework.batch.item.validator.
[CA] SpringValidator">
  <property name="validator">
    <bean class="org.springmodules.validation.valang.ValangValidator">
      <property name="valang">
      <value><![CDATA[
{price : ? >= 0 : 'Product price cannot be negative!'}                #A
      ]]></value>
      </property>
    </bean>
  </property>
</bean>

<bean id="reader" (...) >
  (...)
</bean>

<bean id="writer" (...)>
  (...)
</bean>
```

#A Uses Valang for validation

The key to this configuration is the link between the Spring Batch `ValidatingItemProcessor` class and the Spring Modules `ValangValidator` class. The Spring Batch `ValidatingItemProcessor` class needs a Spring Batch `Validator`, so we provide it a Spring Batch `SpringValidator` class, which itself needs a Spring `Validator` – the interface the `ValangValidator` class implements! In short, the Spring Batch `SpringValidator` class is the bridge between the Spring Batch and Spring validation systems, and the `ValangValidator` builds upon the Spring system (figure 8.7 illustrates the relation between these interfaces and classes, you can also read the note about validator interfaces if you want the whole story.) The `valang` property of the `ValangValidator` accepts one or more validation rules (we used only one in the example.) Note we explicitly set the validating item processor to skip mode (the `filter` property is `false`), so we need to set up a skip strategy to avoid failing the job if the validation fails.

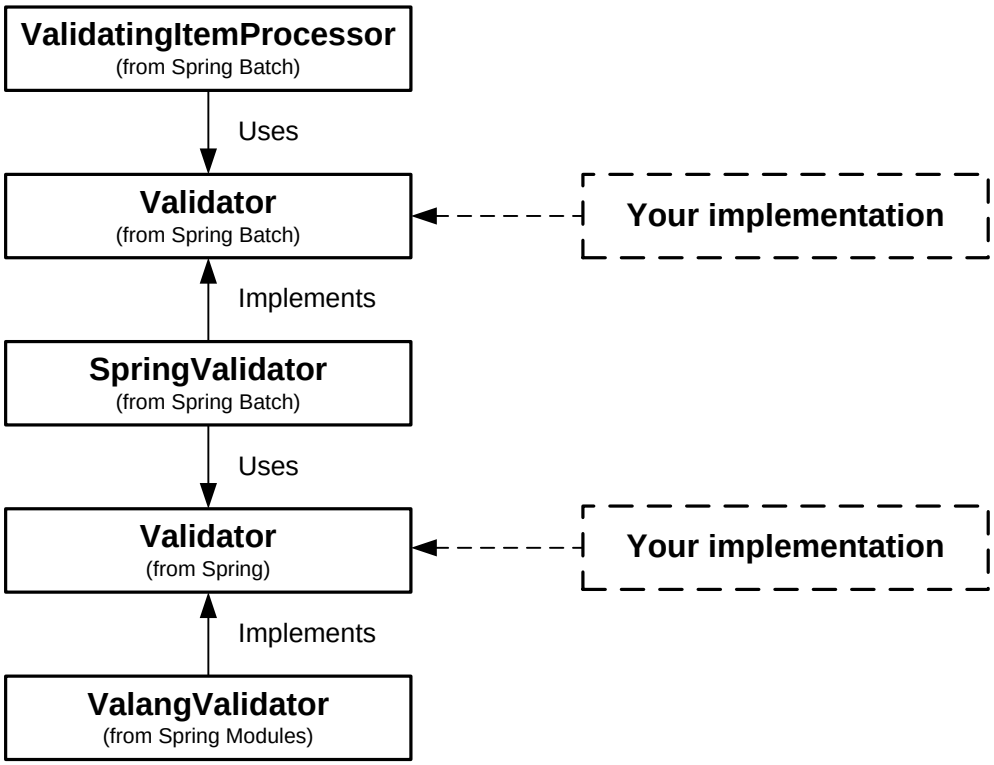

Figure 8.7 The relationship between Spring Batch, Spring and your validation logic. Spring Batch provides a level of abstraction with its Validator interface and an implementation (SpringValidator) that uses the Spring Validator interface. The ValangValidator implementation – from Spring Modules – depends on the Spring Validator interface. Both Validator interfaces (from Spring Batch and Spring) are potential extension points for your own implementations.

Validator interfaces everywhere!

Spring Batch does not intend for application-specific validators to be the only implementers of the Spring Batch `Validator` interface. It can also provide a level of indirection between Spring Batch and your favorite validation framework. Spring Batch provides one implementation of `Validator`: `SpringValidator`, which plugs in the Spring Framework's validation mechanism. Spring bases its validation mechanism on a `Validator` interface, but this one lies in the `org.springframework.validation` package and is part of the Spring Framework. This can look confusing, but the Spring Batch team didn't want to directly tie Spring Batch's validation system to Spring's. By using the Spring Batch `SpringValidator` class, you can use any Spring `Validator` implementation, like the one from Spring Modules for Valang.

Our introduction to Valang ends our coverage of the use of the item-processing phase for validation. Remember that you must follow strict rules if you don't want to confuse skip with filter when validating items. Spring Batch includes the `ValidatingItemProcessor` class that you can configure to skip or filter when validation fails. Finally, you can implement your validation rules programmatically – in Java –

or choose a declarative approach with a validation language like Valang. Let's see now how to apply the composite pattern to chain item processors.

8.4 Chaining item processors

As we stated at the beginning of this chapter, the item-processing phase of a chunk-oriented step is a good place to embed business logic. Assuming that each item processor in our application implements one single business rule (this is simplistic, but enough to illustrate our point), how could we enforce several business rules in the item processing phase of a single step? Moreover, recall that we can only insert a single item processor between an item reader and an item writer. The solution consists in applying the composite pattern by using a composite item processor that maintains a list of item processors (the delegates.) The composite item processor delegates calls to all the members in its list, one after the next. Figure 8.8 illustrates the model of a composite item processor.

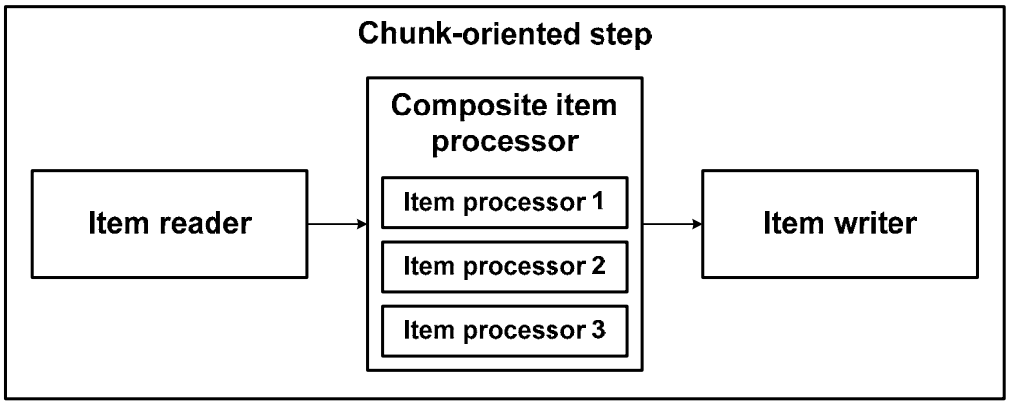

Figure 8.8 Using a composite item processor allows item processors to be chained in order to apply a succession of business rules, transformations or validations.

When using a composite item processor, the delegates should form a type-compatible chain: the type of object an item processor returns must be compatible with the type of object the next item processor expects.

Spring Batch provides the `CompositeItemProcessor` class and we illustrate its use with the import of partner products into our online store. In section 8.2, we covered the transformation of read items, where we distinguished two types of transformations:

- Changing the state of the read item – we illustrated this by mapping the product identifier in the partner's namespace to the product identifier in the ACME system.
- Producing another object from the read item – we illustrated this by producing instances of our `Product` class from `PartnerProduct` objects (created by the item reader).

What do we do if our import job needs to do both? We use two item processors: the first item processor reads raw `PartnerProduct` objects from the flat file and transforms them into `Product` objects, then

the second item processor maps partner product IDs to ACME IDs. Figure 8.9 illustrates the sequence of our import step.

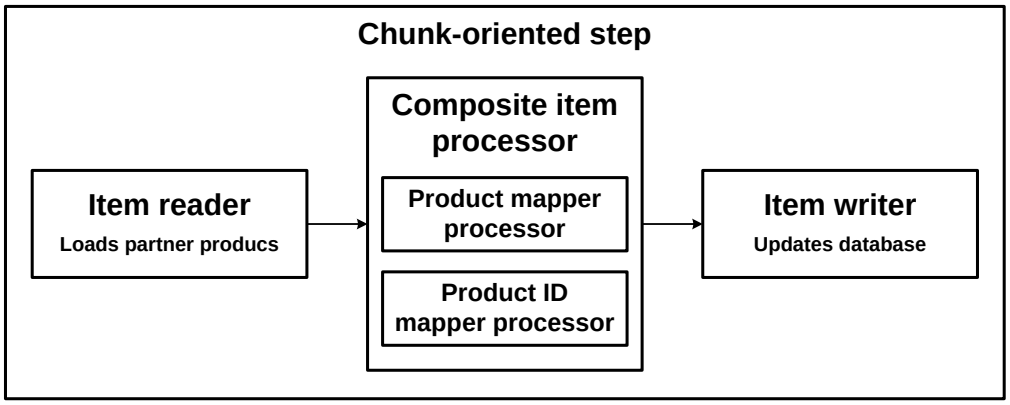

Figure 8.9 Applying the composite item processor pattern to the import products job. The first delegate item processor converts partner product objects into online store product objects. The second delegate item processor maps partner identifiers with ACME identifiers. We re-use and combine item processors without any modification.

We already implemented all the Java code in section 8.2, so we only need to configure a Spring Batch `CompositeItemProcessor` with our two delegate item processors, as shown in listing 8.16.

Listing 8.16 Chaining item processors with the composite item processor

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

<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  (...)
</bean>

<bean id="processor"                                                    #A
      class="org.springframework.batch.item.support.                    #A
[CA] CompositeItemProcessor">                                           #A
  <property name="delegates">                                           #A
    <list>                                                              #A
      <ref bean="productMapperProcessor" />                             #A
      <ref bean="productIdMapperProcessor" />                           #A
    </list>                                                             #A
  </property>                                                           #A
</bean>                                                                 #A
```

```
<bean id="productMapperProcessor"                                    #B
      class="com.manning.sbia.ch08.                                  #B
[CA] PartnerProductItemProcessor">                                   #B
  <property name="mapper">                                           #B
    <bean class="com.manning.sbia.ch08.                              #B
[CA] SimplePartnerProductMapper" />                                  #B
  </property>                                                        #B
</bean>                                                              #B

<bean id="productIdMapperProcessor"                                  #C
      class="com.manning.sbia.ch08.                                  #C
[CA] PartnerIdItemProcessor">                                        #C
  <property name="mapper">                                           #C
    <bean id="partnerIdMapper"                                       #C
          class="com.manning.sbia.ch08.                              #C
[CA] PartnerIdMapper">                                               #C
      <property name="partnerId" value="PARTNER1" />                 #C
      <property name="dataSource" ref="dataSource" />                #C
    </bean>                                                          #C
  </property>                                                        #C
</bean>                                                              #C

<bean id="writer"
      class="com.manning.sbia.ch02.batch.ProductJdbcItemWriter">
  <constructor-arg ref="dataSource" />
</bean>
```

#A Sets two delegates to compose item processor
#B Converts partner products into store products
#C Maps partner product IDs to store product IDs

This example shows the power of the composite pattern applied to building a processing chain: we haven't modified our two existing item processors, we re-use them as-is. Spring Batch encourages separation of concerns by isolating business logic in item processors. We benefit from this here by reusing of our item processors.

It's now time to conclude our journey in the world of Spring Batch data processing.

8.5 Summary

Spring Batch isn't only about reading and writing data: in a chunk-oriented step, you can insert an item processor between the item reader and the item writer to perform any kind of operation. The typical job of an item processor is to implement business logic. For example, an item processor can convert read items into other kinds of objects Spring Batch sends to the item writer. As batch applications often exchange data between two systems, going from one representation to another falls into the domain of item processors.

Spring Batch defines another contract in the processing phase: filtering. For example, if items already exist in the target data store, the application should not insert them again. You can filter items such that they will never get to the writing phase. We made a clear distinction between filtering items and skipping items. Skipping denotes that an item is invalid. This distinction became even more relevant when we covered validation. Thanks to the Spring Batch `ValidatingItemProcessor` class, you can easily switch from skipping to filtering semantics. We used the `ValidatingItemProcessor` class to validate that the price of imported products isn't negative before the job writes them to the database. We saw that we can isolate validation rules in dedicated validator components and we used this feature to plug in a declarative validation framework, Valang.

9

Implementing bulletproof jobs

This chapter covers:

- handling errors with retry and skip
- logging errors with listeners
- restarting an execution after a failure

Batch jobs manipulate large amounts of data automatically. Previous chapters showed how Spring Batch helps to read, process, and write data efficiently and without much Java code, thanks to ready-to-use I/O components. It's time to deal with the automatic aspect of batch jobs. Batch jobs operate over long periods, at night for example, and without human intervention. Even if a batch job can send an email to an operator when there's something wrong, it's on its own most of the time. A batch job isn't automatic if it fails each time something goes wrong, it needs to be able to handle errors correctly and not crash abruptly. Perhaps you know how frustrating it is to sit at your desk in the morning and see that some nightly jobs have crashed because of a missing comma in an input file.

This chapter explains techniques to make your batch jobs more robust and reliable when errors occur during processing. By the end of this chapter, you will know how to build bulletproof batch jobs and be confident to see green lights for your batch job results.

The first section of this chapter explains how a batch job should behave when errors or edge-cases emerge during processing. Spring Batch has built-in support to handle errors when a job is executing, with its skip and retry features. Skip and retry are about avoiding crashes, but crashes are inevitable, so Spring Batch also supports restarting a job after a failed execution. Sections 9.2, 9.3, and 9.4 covers skip, retry, and restart, respectively.

By following the guidelines and the techniques in this chapter, you'll go from "my job failed miserably because of a missing comma" to "bring in your fancy-formatted input file, nothing scares my job anymore". Let's start and get bulletproof!

9.1 What is a bulletproof job?

A bulletproof job is able to handle errors gracefully; it won't fail miserably because of a minor error like a missing comma. It won't fail abruptly either for a major problem like a constraint violation in the database. Before giving some guidelines on the design of a robust job, let's list some requirements that a job must meet.

9.1.1 What makes a job bulletproof?

A bulletproof batch job should meet the following general requirements:

- Robust – the job should fail only for fatal exceptions and should recover gracefully from any non-fatal exception. As software developers, we can't do anything about a power cut, but we can properly handle incorrectly formatted lines or a missing input file.
- Traceable – the job should record any abnormal behavior. A job can skip as many incorrectly formatted lines as it wants, but it should log them to record what did not make it in the database and allow someone to do something about it.
- Restartable – in case of an abrupt failure, the job should be able to restart properly. Depending on the use case, the job could restart exactly where it left off, or even forbid a restart because it would process the same data again.

Good news: Spring Batch provides all the features to meet these requirements! You can activate these features through configuration or by plugging in your own code through extension points (to log errors for example.) A tool like Spring Batch isn't enough to write a bulletproof job, you also need to design the job properly before leveraging the tool.

9.1.2 Designing a bulletproof job

To make your batch jobs bulletproof, you first need to think about failure scenarios. What can go wrong in this batch job? Anything can happen but the nature of the operations in a job helps to narrow the failure scenarios. The batch job we introduced in chapter 2 starts by decompressing a ZIP archive to a working directory before reading the lines of the extracted file and inserting them in the database. Many things can go wrong: the archive can be corrupt (if it's there!); the OS does not allow the process to write in the working directory, some lines in the files can be in an incorrect format, and the list goes on.

> **Testing failure scenarios**
>
> Remember that Spring Batch is a lightweight framework. It means you can easily test failure scenarios in integration tests. You can simulate many failure scenarios thanks to testing techniques like mock objects for example. Chapter 14 covers how to test batch applications. For JUnit testing techniques in general, you can also refer to *JUnit in Action*, from Manning Publications.

Once you've identified failure scenarios, you must think about how to deal with them. If there's no ZIP archive at the beginning of the execution, there's not much the job can do, but that's not a reason to fail abruptly. How should the job handle incorrectly formatted lines? Should it skip them or fail the whole execution as soon as it finds a bad line? In our case, we could skip incorrect lines and ensure that we log them somewhere.

Spring Batch has built-in support for error handling, but it doesn't mean you can make batch jobs bulletproof by setting some magical attribute in an XML configuration file (even if sometimes that's the case). Rather, it means that Spring Batch provides infrastructure, deals with tedious plumbing, but you must always know what you're doing: when and why to use Spring Batch error handling. That's what makes batch programming interesting! Let's now see how to deal with errors in Spring Batch.

9.1.3 Techniques for bulletproofing jobs

Unless you control your batch jobs as Neo controls the Matrix, you'll always end up getting errors in your batch applications. Spring Batch includes three features to deal with errors: skip, retry, and restart. Table 9.1 describes these features.

Table 9.1 Error handling support in Spring Batch

Feature	When?	What?	Where?
Skip	For non-fatal exceptions.	Keeps processing for an incorrect item.	Chunk-oriented step.
Retry	For transient exceptions.	Makes new attempts on an operation for a transient failure.	Chunk-oriented step, application code.
Restart	After an execution failure.	Restarts a job instance where the last execution failed.	On job launch.

The features listed in table 9.1 are independent from each other: you can use one without the others, or combine them. Remember that skip and retry are about avoiding a crash on an error, whereas restart is useful when the job has crashed, to restart it where it left off.

Skipping allows for moving processing along to the next line in an input file if the current line is in an incorrect format. If the job doesn't process a line, perhaps you can live without it and the job can process the remaining lines in the file.

Retry attempts an operation several times: the operation can fail at first, but another attempt can succeed. Retry isn't useful for errors like badly formatted input lines; it's useful for transient errors, like concurrency errors. Skip and retry contribute to making job executions more robust, because they deal with error handling during processing.

Restart is useful after a failure, when the execution of a job crashed. Instead of starting the job from scratch, Spring Batch allows for restarting it exactly where the failed execution left off. Restarting can avoid potential corruption of the data in case of reprocessing. Restarting can also save a lot of time if the failed execution was close to the end.

Before covering each feature, let's see how skip, retry, and restart can apply to our import products job.

9.1.4 Skip, retry, and restart in action

Recall our import products job: The core of the job reads a flat file containing one product description per each line and updates the online store database accordingly. Here is how skip, retry, and restart could apply to this job.

- Skip – a line in the flat file is not in the correctly format. We don't want to stop the job execution because of a couple of bad lines: this could mean losing an unknown amount of updates and inserts. We can tell Spring Batch to skip the line that caused the item reader to throw an exception on a formatting error.
- Retry – because some products are already in the database, the flat file data is used to update the products (description, price, and so). Even if the job runs during period of low activity in the online store, users sometimes access the updated products, causing the database to lock the corresponding rows. The database throws a concurrency exception when the job tries to update a product in a locked row, but retrying the update again a few milliseconds later works. You can configure Spring Batch to retry automatically.
- Restart – if Spring Batch has to skip more than 10 products because of badly formatted lines, we consider the input file invalid and should go through a validation phase. The job fails as soon as we reach 10 skipped products, as defined in the configuration. An operator will analyze the input file and correct it before restarting the import. Spring Batch is able to restart the job on the line that caused the failed execution. The work performed by the previous execution isn't lost.

The import products job is robust and reliable thanks to Spring Batch. Now that we defined the roles of skip, retry, and restart, let's study them individually.

9.2 Skipping instead of failing

Sometimes errors aren't fatal: a job execution shouldn't stop when something goes wrong. In our online store application, when importing products from a flat file, should we stop the job execution because one line is in an incorrect format? We could stop the whole execution, but the job would not insert the subsequent lines from the file, which means fewer products in the catalog and less money coming in! A better solution would be to skip the incorrectly formatted line and move on to the next line.

Skipping items or not in a chunk-oriented step is a business decision. The good news is that Spring Batch makes the decision of skipping a matter of configuration, there's no impact on the application code. Let's see how to tell Spring Batch to skip items and then how to tune the skip policy.

9.2.1 Configuring exceptions to be skipped

Recall our import product job: it reads products from a flat file and then inserts them into the database. It would be a shame to stop the whole execution for a couple of incorrect lines in a file containing thousands or even tens of thousands of lines. We can tell Spring Batch to skip incorrect lines by specifying which exceptions it should ignore. To do this, we use the `skippable-exception-classes` element, as shown in listing 9.1.

Listing 9.1 Configuring exceptions to skip in a chunk-oriented step

```
<job id="importProductsJob">
  <step id="importProductsStep">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100"
             skip-limit="10">
        <skippable-exception-classes>                                  #A
          <include class="org.springframework.batch                    #A
[CA] .item.file.FlatFileParseException" />                             #A
        </skippable-exception-classes>                                 #A
      </chunk>
    </tasklet>
  </step>
</job>
```

#A Sets exceptions to be skipped

In the `skippable-exception-classes` element, you specify the exceptions to skip with the `include` element. You can specify several exception classes (with several `include` elements). When using the `include` element, you don't only specify one class of exception to skip, but also all the subclasses of the exception. Listing 9.1 configures Spring Batch to skip a `FlatFileParseException` and *all its subclasses*.

Note also in listing 9.1 the use of the `skip-limit` attribute, which sets the maximum number of items to skip in the step before failing the execution. Skipping is useful, but skipping too many items can signify that the input file is corrupt. As soon as Spring Batch reaches the skip limit, it will stop processing and fail the execution. When you declare an exception to skip, you must specify a skip limit.

The `include` element skips a whole exception hierarchy, but what if you don't want to skip all the subclasses of the specified exception? In this case, you use the `exclude` element. The following snippet shows how to skip `ItemReaderExceptions` but excludes `NonTransientResourceException`:

```
<skippable-exception-classes>
  <include
    class="org.springframework.batch.item.ItemReaderException"/>
  <exclude
    class="org.springframework.batch.item.NonTransientResourceException"/>
</skippable-exception-classes>
```

Figure 9.1 shows the relationship between `ItemReaderException` and `NonTransientResourceException`. With the settings from the previous snippet, a `FlatFileParseException` will trigger a skip, whereas a

`NonTransientFlatFileException` won't. Expressing this requirement in English, we would say that we want to skip any error due to bad formatting in the input file (`ParseException`) and that we don't want to skip errors due to I/O problems (`NonTransientResourceException`).

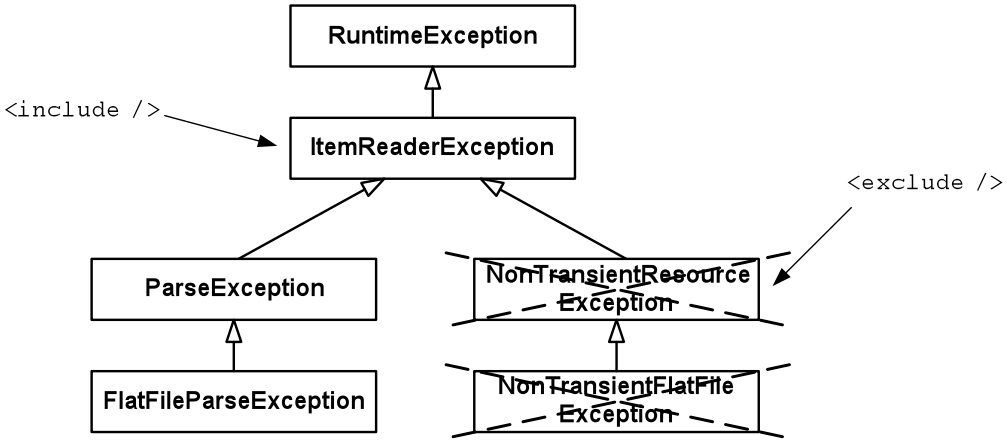

Figure 9.1 The include element specifies an exception class and all its subclasses. In you want to exclude part of the hierarchy, use the exclude element. The exclude element also works transitively, as it excludes a class and its subclasses.

Specifying exceptions to skip and a skip limit is straightforward and fits most cases. Can you avoid using a skip limit and import as many items as possible? Yes. When importing products in the online store, we could process the entire input file, no matter how many lines are incorrect and skipped. As we log these skipped lines, we can correct them and import them the next day. Spring Batch gives you full control over the skip behavior by specifying a *skip policy*.

9.2.2 Configuring a SkipPolicy for complete control

Who decides if an item should be skipped or not in a chunk-oriented step? Spring Batch calls the skip policy when an item reader, processor, or writer throws an exception, as figure 9.2 shows. When using the `skippable-exception-classes` element, Spring Batch uses a default skip policy implementation (`LimitCheckingItemSkipPolicy`), but you can declare your own skip policy as a Spring bean and plug it in your step. This gives you more control if the `skippable-exception-classes` and `skip-limit` pair isn't enough.

Figure 9.2 When skip is on, Spring Batch asks a skip policy whether it should skip an exception thrown by an item reader, processor, or writer. The skip policy's decision can depend on the type of the exception and on the number of skipped items so far in the step.

Let's say we know exactly on which exceptions we want to skip items but we don't care about the number of skipped items. We can implement our own skip policy, as shown in listing 9.2.

Listing 9.2 Implementing a skip policy with no skip limit

```
package com.manning.sbia.ch09.skip;

import org.springframework.batch.core.step.skip.SkipLimitExceededException;
import org.springframework.batch.core.step.skip.SkipPolicy;

public class ExceptionSkipPolicy implements SkipPolicy {

  private Class<? extends Exception> exceptionClassToSkip;

  public ExceptionSkipPolicy(
      Class<? extends Exception> exceptionClassToSkip) {
    super();
    this.exceptionClassToSkip = exceptionClassToSkip;
  }

  @Override
  public boolean shouldSkip(Throwable t, int skipCount)
      throws SkipLimitExceededException {
    return exceptionClassToSkip.isAssignableFrom(                    #A
      t.getClass()                                                   #A
    );                                                               #A
  }

}
```

#A Skips on Exception class and subclasses

Once you implement your own skip policy, you can plug it in a step by using the `skip-policy` attribute, as shown in listing 9.3.

Listing 9.3 Plugging in a skip policy in a chunk-oriented step

```
<bean id="skipPolicy" class="com.manning.sbia.ch09                   #A
[CA] .skip.ExceptionSkipPolicy">                                     #A
  <constructor-arg value="org.springframework.batch                  #A
[CA] .item.file.FlatFileParseException" />                           #A
</bean>                                                              #A

<job id="importProductsJobWithSkipPolicy"
     xmlns="http://www.springframework.org/schema/batch">
```

```
    <step id="importProductsStepWithSkipPolicy">
      <tasklet>
        <chunk reader="reader" writer="writer" commit-interval="100"
               skip-policy="skipPolicy" />                                #B
      </tasklet>
    </step>
  </job>
```

#A Declares skip policy bean
#B Sets skip policy in step

Table 9.2 lists the skip policy implementations Spring Batch provides. Don't hesitate to look them up before implementing your own.

Table 9.2 Skip policy implementations provided by Spring Batch

Skip policy class*	Description
`LimitCheckingItemSkipPolicy`	Skips items depending on the exception thrown and the total number of skipped items. This is the default implementation.
`ExceptionClassifierSkipPolicy`	Delegates skip decision to other skip policies depending on the exception thrown.
`AlwaysSkipItemSkipPolicy`	Always skips, no matter the exception or the total number of skipped items.
`NeverSkipItemSkipPolicy`	Never skips.

* from the `org.springframework.batch.core.step.skip` package

When it comes to skipping, you can stick to the `skippable-exception-classes` and `skip-limit` pair, which have convenient behavior and are easy to configure, with dedicated XML elements. You will typically use to the default skip policy if you care about the total number of skipped items and you don't want to exceed a given limit. If you don't care about the number of skipped items, you can implement your own skip policy and easily plug it in a chunk-oriented step.

How Spring Batch drives chunks with skipped items

We focused on skipping items during the reading phase, but the skip configuration also applies to the processing and writing phases of a chunk-oriented step. Spring Batch doesn't drive a chunk-oriented step the same way when a "skippable" exception is thrown in the reading, processing, or writing phase. When an item reader throws a skippable exception, Spring Batch just calls the `read` method again on the item reader to get the

next item. There's no rollback on the transaction. When an item processor throws a skippable exception, Spring Batch rolls back the transaction of the current chunk and re-submits the read items to the item processor, except for the one that triggered the skippable exception in the previous run. Figure 9.3 shows what Spring Batch does when the item writer throws a skippable exception. As the framework doesn't know which item threw the exception, it re-processes each item in the chunk one by one, in its own transaction.

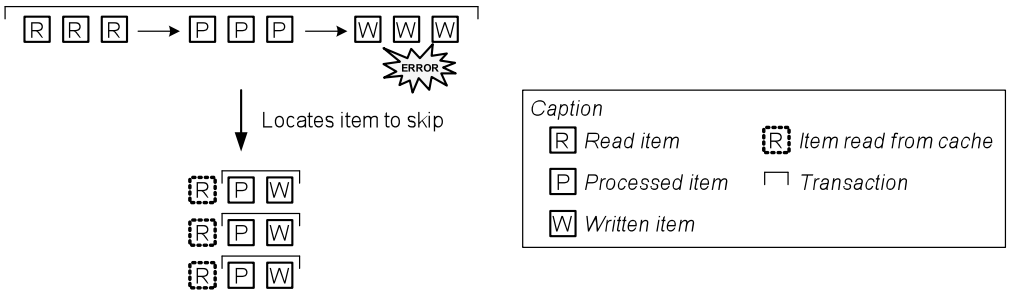

Figure 9.3 When a writer throws a skippable exception, Spring Batch can't know which item triggered the exception. Spring Batch then rolls back the transaction and process the chunk item by item. Note that Spring Batch doesn't read the items again, by default, because it maintains a chunk-scoped cache.

Skipping incorrect items make a job more robust, but you might want to keep track of these items. Let's see how Spring Batch lets you do that, with a skip listener.

9.2.3 Listening and logging skipped items

Ok, your job doesn't fail miserably anymore because of a single incorrect line in your 500 megabytes input file, fine, but how do you easily spot these incorrect line? One solution is to log each skipped item, with the skip callbacks provided by Spring Batch. Once you have the skipped items in a file or in a database, you can deal with them: correct the input file, do some manual processing to deal with the error, and so on. The point is have a record of what went wrong!

Spring Batch provides the `SkipListener` interface to listen to skipped items:

```
public interface SkipListener<T,S> extends StepListener {
  void onSkipInRead(Throwable t);
  void onSkipInProcess(T item, Throwable t);
  void onSkipInWrite(S item, Throwable t);
}
```

You can implement a skip listener and plug it in a step, as figure 9.4 shows. Spring Batch will call the appropriate method on the listener when it skips an item. To implement a skip listener, you can directly implement the `SkipListener` interface, but this implies implementing three methods, even if you expect skipped items only during the reading phase. To avoid implementing empty methods, you can inherit from the

`SkipListenerSupport` adapter class, which provides no-op implementations: you only override the method you need.

There's one more solution: using annotations on a simple class (no interface, no abstract class.) Spring Batch provides one annotation for each method of the `SkipListener` interface: `@OnSkipInRead`, `@OnSkipInProcess`, `@OnSkipInWrite`.

Figure 9.4 Spring Batch lets you register skip listeners. Whenever a chunk-oriented step throws a skippable exception, Spring Batch calls the listener accordingly. A listener can then log the skipped item for later processing.

Next, we use the annotation solution, with `@OnSkipInRead` to skip items during the reading phase. Listing 9.4 shows our skip listener, which logs the incorrect line to a database.

Listing 9.4 Logging skipped items with a skip listener

```
package com.manning.sbia.ch09.skip;

import javax.sql.DataSource;
import org.springframework.batch.core.annotation.OnSkipInRead;
import org.springframework.batch.item.file.FlatFileParseException;
import org.springframework.jdbc.core.JdbcTemplate;

public class DatabaseSkipListener {

  private JdbcTemplate jdbcTemplate;

  public DatabaseSkipListener(DataSource ds) {
    this.jdbcTemplate = new JdbcTemplate(ds);
  }

  @OnSkipInRead                                                    #A
  public void log(Throwable t) {
    if(t instanceof FlatFileParseException) {
      FlatFileParseException ffpe = (FlatFileParseException) t;
      jdbcTemplate.update(                                         #B
        "insert into skipped_product " +                           #B
        "(line,line_number) values (?,?)",                         #B
        ffpe.getInput(),ffpe.getLineNumber()                       #B
      );                                                           #B
    }
  }

}
```

#A Uses annotation to trigger callback
#B Logs line information in database

The skip listener logs the incorrect line in the database, but it could use any other logging system, a logging framework like Java's own `java.util.logging`, Apache Log4J or SLF4J for example.

Once you implement the skip listener, you need to register it. Listing 9.5 below shows how to register the skip listener on a step, using the `listeners` element in the `tasklet` element.

Listing 9.5 Registering a skip listener

```
<bean id="skipListener" class="com.manning                              #A
[CA] .sbia.ch09.skip.DatabaseSkipListener">                             #A
  <constructor-arg ref="dataSource" />                                  #A
</bean>                                                                 #A

<job id="importProductsJob"
     xmlns="http://www.springframework.org/schema/batch">
  <step id="importProductsStep">
    <tasklet>
      <chunk reader="reader" writer="writer"
             commit-interval="100" skip-limit="10">
        <skippable-exception-classes>
          <include class="org.springframework.batch.item.file
[CA] .FlatFileParseException" />
        </skippable-exception-classes>
      </chunk>
      <listeners>                                                       #B
        <listener ref="skipListener" />                                 #B
      </listeners>                                                      #B
    </tasklet>
  </step>
</job>
```

#A Declares skip listener bean
#B Registers skip listener

A couple of details that are worth mentioning in our skip listener configuration:

1. You can have several skip listeners – we registered only one skip listener, we could have as many as we want.
2. Spring Batch is smart enough to figure out the listener type – we used the generic `listener` element to register the skip listener. Spring Batch will detect this is a skip listener (Spring Batch provides many different kinds of listeners.)

When does Spring Batch call a skip listener method? Just after the item reader, processor, or writer throws the to-be-skipped exception you may think. No, not just after. Spring Batch postpones the call to skip listeners to right before committing the transaction for the chunk. Why is that? Because something wrong can happen *after* Spring Batch skips an item and Spring Batch could then rollback the transaction. Imagine that the item reader throws a to-be-skipped exception. Later on, something goes wrong during the writing phase of the same chunk and Spring Batch rolls back the transaction, and could even fail the job execution. You

wouldn't want to log the skipped item during the reading phase, because Spring Batch rolled back the whole chunk! That's why Spring Batch calls skip listeners just before the commit of the chunk, when it's almost certain nothing unexpected could happen.

We're done with skipping, a feature Spring Batch provides to make jobs more robust when errors aren't fatal. Do you want your jobs to get even more robust? Perhaps skipping an item immediately is too pessimistic, what about making additional attempts before skipping. This is what we call *retry* and Spring Batch offers first-class support for this feature.

9.3 Retrying on error

By default, an exception in a chunk-oriented step causes the step to fail. You can skip the exception if you don't want to fail the whole step. Skipping works well for deterministic exceptions, for example an incorrect line in a flat file. Exceptions aren't always deterministic; sometimes they can be *transient*. We call an exception transient when an operation fails at first, but a new attempt – even immediately after – is successful.

Have you ever used your cell phone in a place where the connection would arguably be bad? In a tunnel, for example, or on a ferry, sailing on the Baltic Sea on a Friday night while watching a Finnish clown show[1]. You start speaking on the cell phone, but the line drops out. Do you give up and start watching the clown show or do you try to dial the number again? Maybe the connection will be better on the second attempt or in a couple of minutes. Transient errors happen all the time in the real world, when using the phone or online conference tools like Skype. You usually retry several times after a failure, before giving up and trying later if the call does not go through.

What are transient exceptions in batch applications? Concurrency exceptions are a typical example. If a batch job tries to update a row that another process holds a lock on, the database can cause an error. Retrying the operation immediately after can be successful, because the other process may have released the lock in the meantime. Any operation involving an unreliable network – like a web service call – can also throw transient exceptions. A new attempt, with a new request (or connection,) can succeed.

You can configure Spring Batch to retry operations transparently when they throw exceptions, without any impact on the application code. Because transient failures cause these exceptions, we call them *retryable* exceptions.

9.3.1 Configuring retryable exceptions

You configure retryable exceptions inside the `chunk` element, using the `retryable-exception-classes` element, as shown in listing 9.6.

Listing 9.6 Configuring retryable exceptions

[1] It happened… while working on this book!

```
<job id="importProducsJob">
  <step id="importProductsStep">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100"
             retry-limit="3">                                          #A
        <retryable-exception-classes>                                  #B
          <include class="org.springframework.dao                      #B
[CA].OptimisticLockingFailureException" />                             #B
        </retryable-exception-classes>                                 #B
      </chunk>
    </tasklet>
  </step>
</job>
```

#A Sets max number of retries
#B Sets exceptions to retry on

Notice the `retry-limit` attribute, used to specify how many times Spring Batch should retry an operation. Just like for skipping, you can include a complete exception hierarchy with the `include` element and exclude some specific exceptions with the `exclude` element. You can use both XML elements several times. The following snippet illustrates the use of the exclude element for retry:

```
<retryable-exception-classes>
  <include
    class="org.springframework.dao.TransientDataAccessException"/>
  <exclude
    class="org.springframework.dao.PessimisticLockingFailureException"/>
</retryable-exception-classes>
```

Figure 9.5 shows the relationship between the exceptions `TransientDataAccessException` and `PessimisticLockingFailureException`. In the previous snippet, we tell Spring Batch to retry when Spring throws transient exceptions, except when the exceptions are related to pessimistic locking.

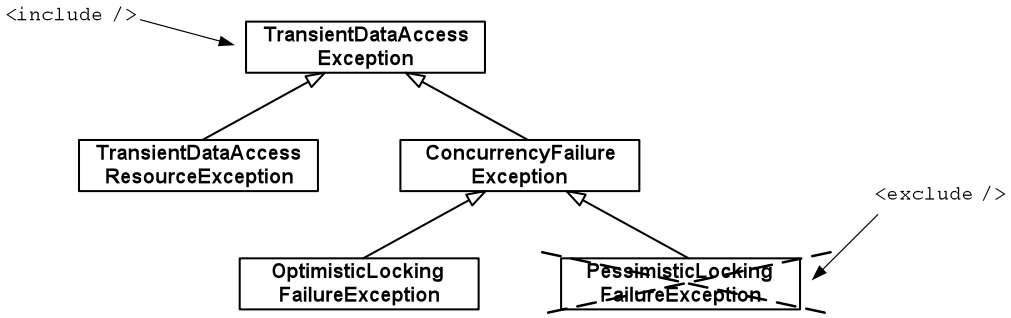

Figure 9.6 Spring Batch configured to retry exceptions: the include tag includes an exception class and all its subclasses. By using the exclude tag, we specify a part of the hierarchy that Spring Batch shouldn't retry. Here, Spring Batch retries any transient exception, except pessimistic locking exceptions.

Spring Batch only retries the item processing and item writing phases. By default, a retryable exception triggers a rollback, so you should be careful with retry, as retrying too many times for too many items can degrade performance. You should use retryable exception only for exceptions that are non-deterministic, not exception related to format or constraint violations, which are typically deterministic. Figure 9.7 summarizes the retry behavior in Spring Batch.

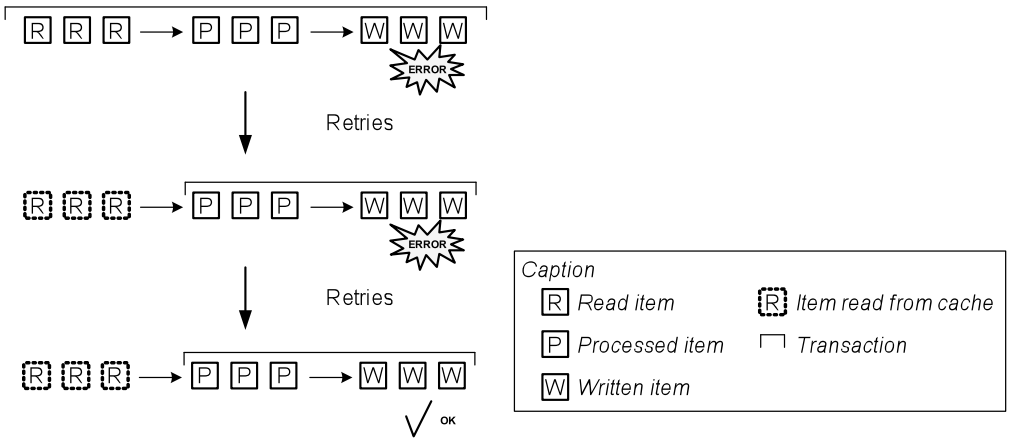

Figure 9.7 Spring Batch retries only for exceptions thrown during item processing or item writing. Retry triggers a rollback, so retrying is costly, don't abuse it! Note that Spring Batch doesn't read the items again, by default, because it maintains a chunk-scoped cache.

Override equals() and hashCode() when using retry

In a chunk-oriented step, Spring Batch handles retry on the item processing and writing phases. By default, a retry implies a rollback, so Spring Batch needs to restore the context of retried operations across transactions. Spring Batch needs to track items closely to know which item could have triggered the retry. Remember that Spring Batch can't always know which item triggers an exception during the writing phase, as an item writer handles a list of items. Spring Batch relies on the identity of items to track them, so for Spring Batch retry to work correctly you should override the `equals` and `hashCode` methods of your items' classes – by using a database identifier for example.

COMBINING RETRY AND SKIP

You can combine retry with skip: a job retries an unsuccessful operation several times and then skips it. Remember that once Spring Batch reaches the retry limit, the exception causes the step to exit and, by default, fail. Use combine retry and skip when you don't want than a persisting transient error to fail a step. Listing 9.7 below shows how to combine retry and skip.

Listing 9.7 Combining retry and skip

```
<job id="job">
  <step id="step">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100"
             retry-limit="3" skip-limit="10">
        <retryable-exception-classes>                                    #A
          <include class="org.springframework.dao                        #A
[CA] .DeadlockLoserDataAccessException" />                               #A
        </retryable-exception-classes>                                   #A
        <skippable-exception-classes>                                    #B
          <include class="org.springframework.dao                        #B
[CA] .DeadlockLoserDataAccessException" />                               #B
        </skippable-exception-classes>                                   #B
      </chunk>
    </tasklet>
  </step>
</job>
```

#A Specifies retryable exceptions
#B Specifies skippable exceptions

Automatic retry in a chunk-oriented step can make jobs more robust. It's indeed a shame to fail a step because of an unstable network, whereas retrying a few milliseconds later could have worked. You now know about the default retry configuration in Spring Batch and this should be enough for most cases. The next section explores how to control retry by setting a *retry policy*.

9.3.2 Controlling retry with a retry policy

By default, Spring Batch lets you configure retryable exceptions and the retry count. Sometimes, retry is more complex: some exceptions deserve more attempts than others do or you want to keep retrying as long as the operation doesn't exceed a given timeout. Spring Batch delegates the decision to retry or not to a *retry policy*. When configuring retry in Spring Batch, you can use the `retryable-exception-classes` element and `retry-limit` pair or provide a `RetryPolicy` bean instead.

Table 9.3 lists the `RetryPolicy` implementations included in Spring Batch. You can use these implementations as-is or choose to implement your own retry policy for specific needs.

Table 9.3 RetryPolicy implementations provided by Spring Batch

Class	Description
`SimpleRetryPolicy`	Retries on given exception hierarchies, a given number of times. This is the default implementation, configured with `retryable-exception-classes/retry-limit`.
`TimeoutRetryPolicy`	Stops retrying when an operation takes too long.
`ExceptionClassifierRetryPolicy`	Combines multiple retry policies depending on the

exception thrown.

Let's see how to set a retry policy with an example. Imagine you want to use retry on concurrent exceptions, but you have several kinds of concurrent exceptions to deal with and you don't want the same retry behavior for all of them. Sprint Batch should retry all generic concurrent exceptions three times, whereas it should retry the deadlock concurrent exceptions five times, which is more aggressive.

The `ExceptionClassifierRetryPolicy` implementation is a perfect match: it delegates the retry decision to different policies depending on the class of the thrown exception. The trick is to encapsulate two `SimpleRetryPolicy` beans in the `ExceptionClassifierRetryPolicy`, one for each kind of exception, as shown in listing 9.8.

Listing 9.8 Using a retry policy for a different behavior with concurrent exceptions

```
<job id="retryPolicyJob"
     xmlns="http://www.springframework.org/schema/batch">
  <step id="retryPolicyStep">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100"
             retry-policy="retryPolicy" />                              #A
    </tasklet>
  </step>
</job>

<bean id="retryPolicy" class="org.springframework
[CA].batch.retry.policy.ExceptionClassifierRetryPolicy">
  <property name="policyMap">                                           #B
    <map>
      <entry key="org.springframework.dao.ConcurrencyFailureException">
        <bean class="org.springframework.batch.retry
[CA].policy.SimpleRetryPolicy">
          <property name="maxAttempts" value="3" />                     #C
        </bean>
      </entry>
      <entry key="org.springframework.dao
[CA] .DeadlockLoserDataAccessException">
        <bean class="org.springframework.batch.retry
[CA] .policy.SimpleRetryPolicy">
          <property name="maxAttempts" value="5" />                     #D
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

#A Sets retry policy on chunk
#B Maps policies to exception classes
#C Sets max number of attempts for concurrent exceptions
#D Sets max number of attempts for deadlock exceptions

Listing 9.8 shows that setting a retry policy allows for flexible retry behavior: the number of retries can be different, depending on the kind of exceptions thrown during processing.

Transparent retries make jobs more robust. Listening to retries also helps learn about the causes of retries. Let's see then how to plug in a retry listener in a step.

9.3.3 Listening to retry

Spring Batch provides the `RetryListener` interface to react to any retried operation. A retry listener can be useful to log retried operations and to gather information. Once you know more about transient failures, you're more likely to change the system to avoid them in subsequent executions (remember, retried operations always degrade performance.)

You can directly implement the `RetryListener` interface; it defines two lifecycle methods – `open` and `close` – that remain often empty, because we usually care only about the error thrown in the operation. A better way is to extend the `RetryListenerSupport` adapter class and override the `onError` method, as shown in listing 9.9 below.

Listing 9.9 Implementing a retry listener to log retried operations

```
package com.manning.sbia.ch09.retry;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.retry.RetryCallback;
import org.springframework.batch.retry.RetryContext;
import org.springframework.batch.retry.listener.RetryListenerSupport;

public class Slf4jRetryListener extends RetryListenerSupport {

  private static final Logger LOG =
LoggerFactory.getLogger(Slf4jRetryListener.class);

  @Override
  public <T> void onError(RetryContext context, RetryCallback<T> callback,
      Throwable throwable) {
    LOG.error("retried operation",throwable);
  }

}
```

Our retry listener uses the SLF4J logging framework to log the exception the operation throws. It could also use JDBC to log the error to a database. The next step is to register the listener in the step, using the `retry-listeners` XML element, as shown in listing 9.10.

Listing 9.10 Registering a retry listener

```
<bean id="retryListener" class="com.manning.sbia.ch09                    #A
[CA] .retry.Slf4jRetryListener" />                                       #A

<job id="job" xmlns="http://www.springframework.org/schema/batch">
  <step id="step">
    <tasklet>
      <chunk reader="reader" writer="writer"
             commit-interval="10" retry-limit="3">
        <retryable-exception-classes>
          <include class="org.springframework.dao
```

```
[CA] .OptimisticLockingFailureException" />
        </retryable-exception-classes>
        <retry-listeners>                                              #B
          <listener ref="retryListener" />                             #B
        </retry-listeners>                                             #B
      </chunk
    </tasklet>
  </step>
</job>
```

#A Declares retry listener bean
#B Registers retry listener

Any time you need to know about retried operations – for example, to get rid of them! – Spring Batch lets you register retry listeners and log errors.

Retry is a built-in feature of chunk-oriented steps. What can you do if you need to retry in your own code, for example in a tasklet?

9.3.4 Retrying in application code with the RetryTemplate

Imagine you use a web service in a custom tasklet to retrieve data that a subsequent step will then use. A call to a web service can cause transient failures, so being able to retry this call would make the tasklet more robust. You can benefit from Spring Batch's retry feature in a tasklet, with to the `RetryOperations` interface and its `RetryTemplate` implementation. The `RetryTemplate` allows for programmatic retry in application code.

The online store uses a tasklet to retrieve the latest discounts from a web service. The discount data are small enough to keep in memory, for later use in the next step. The `DiscountService` interface hides the call to the web service. Listing 9.11 shows the tasklet that retrieves the discounts (we omit the setter methods for brevity.) The tasklet uses a `RetryTemplate` to retry in case of failure.

Listing 9.11 Programmatic retry in a tasklet

```
package com.manning.sbia.ch09.retry;

import java.util.List;
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;
import org.springframework.batch.retry.RetryCallback;
import org.springframework.batch.retry.RetryContext;
import org.springframework.batch.retry.policy.SimpleRetryPolicy;
import org.springframework.batch.retry.support.RetryTemplate;

public class DiscountsWithRetryTemplateTasklet implements Tasklet {

  private DiscountService discountService;
  private DiscountsHolder discountsHolder;

  @Override
```

```
  public RepeatStatus execute(StepContribution contribution,
      ChunkContext chunkContext) throws Exception {
    RetryTemplate retryTemplate = new RetryTemplate();                  #A
    SimpleRetryPolicy retryPolicy =                                     #A
      new SimpleRetryPolicy();                                          #A
    retryPolicy.setMaxAttempts(3);                                      #A
    retryTemplate.setRetryPolicy(retryPolicy);                          #A
    List<Discount> discounts = retryTemplate.execute(                   #B
        new RetryCallback<List<Discount>>() {                           #B
      @Override                                                         #B
      public List<Discount> doWithRetry(                                #B
            RetryContext context)                                       #B
          throws Exception {                                            #B
        return discountService.getDiscounts();                          #B
      }                                                                 #B
    });                                                                 #B
    discountsHolder.setDiscounts(discounts);                            #C
    return RepeatStatus.FINISHED;
  }
  (...)
}
```

#A Configures RetryTemplate
#B Calls web service with retry
#C Stores result for later use

The use of the `RetryTemplate` is straightforward. Note how we configure the `RetryTemplate` with a `RetryPolicy` directly in the tasklet. We could have also defined a `RetryOperations` property in the tasklet and used Spring to inject a `RetryTemplate` bean as a dependency. Thanks to the `RetryTemplate`, we shouldn't fear transient failures on the web service call anymore.

Use of the `RetryTemplate` is simple, but the retry logic is hard-coded in the tasklet. Let's go further to see how to remove the retry logic from the application code.

9.3.5 Retrying transparently with the RetryTemplate and AOP

Can we remove all the retry logic from the tasklet? It would make it easier to test, because the tasklet would be free of any retry code and the tasklet could focus on its core logic. Furthermore, a unit test wouldn't necessarily deal with all retry cases.

Aspect-oriented programming (AOP)

Aspect-oriented programming is a programming paradigm that allows modularizing crosscutting concerns. The idea of AOP is to remove crosscutting concerns from an application's main logic and implement them in dedicated units called aspects. Typical crosscutting concerns are transaction management, logging, security, and retry. The Spring Framework provides first-class support for AOP, thanks to its interceptor-based approach: Spring intercepts application code and calls aspect code to address crosscutting concerns. Thanks to AOP, boilerplate code does not clutter the application code and code aspects address crosscutting concerns in their own units, which also avoids code scattering.

Spring Batch provides an AOP interceptor for retry called `RetryOperationsInterceptor`. By using this interceptor, the tasklet is able to use a `DiscountService` object directly. The interceptor delegates calls to the real `DiscountService` and handles the retry logic. No more dependency on the `RetryTemplate` in the tasklet, the code becomes simpler! Listing 9.12 shows the new version of the tasklet, which doesn't handle retries anymore.

Listing 9.12 Calling the web service without retry logic

```
package com.manning.sbia.ch09.retry;

import java.util.List;
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 DiscountsTasklet implements Tasklet {

  private DiscountService discountService;
  private DiscountsHolder discountsHolder;

  @Override
  public RepeatStatus execute(StepContribution contribution,
      ChunkContext chunkContext) throws Exception {
    List<Discount> discounts = discountService.getDiscounts();
    discountsHolder.setDiscounts(discounts);
    return RepeatStatus.FINISHED;
  }
  (...)
}
```

If we want to keep the tasklet this simple, we need the magic of AOP to handle the retry transparently. Spring AOP will wrap the target `DiscountService` – the one that makes the web service call – in a proxy. This proxy will handle the retry logic thanks to the retry interceptor. The tasklet ends up using this proxy. Listing 9.13 below shows the Spring configuration for transparent, AOP-based retry.

Listing 9.13 Configuring transparent retry with Spring AOP

```
<bean id="discountService" class="com.manning.sbia                    #A
[CA] .ch09.retry.DiscountServiceImpl" />                              #A

<bean id="retryAdvice"                                                #B
      class="org.springframework.batch.retry                          #B
[CA] .interceptor.RetryOperationsInterceptor">                        #B
  <property name="retryOperations">
    <bean class="org.springframework.batch.retry.support.RetryTemplate">
      <property name="retryPolicy">
        <bean class="org.springframework.batch.retry.policy
[CA] .SimpleRetryPolicy">
          <property name="maxAttempts" value="3" />
        </bean>
```

```
      </property>
    </bean>
  </property>
</bean>

<aop:config>                                                          #C
  <aop:pointcut id="retriedOperations"                                #C
    expression="execution(* com.manning.sbia.ch09                     #C
[CA] .retry.DiscountService.*(..))" />                                #C
  <aop:advisor pointcut-ref="retriedOperations"                       #C
               advice-ref="retryAdvice" />                            #C
</aop:config>                                                         #C

<bean class="com.manning.sbia.ch09.retry.DiscountsTasklet">
  <property name="discountService" ref="discountService" />
  <property name="discountsHolder" ref="discountsHolder" />
</bean>

<bean id="discountsHolder"
      class="com.manning.sbia.ch09.retry.DiscountsHolder" />
```

#A Declares target discount service
#B Declares retry interceptor with RetryTemplate
#C Applies interceptor on target service

That's it! Not only should you no longer fear transient failures when calling the web service, but the calling tasklet doesn't even know that there's some retry logic on the `DiscountService`. In addition, retry support isn't limited to batch applications: you can use it in a web application, whenever a call is subject to transient failures.

This ends our coverage of retry. Spring Batch allows for transparent, configurable retry, which allows you to decouple the application code from any retry logic. Retry is useful for transient, non-deterministic errors, like concurrency errors. The default behavior is to retry on given exception classes, until Spring Batch reaches the retry limit. Note that you can also control the retry behavior by plugging in a retry policy.

Skip and retry help to avoid job failures; they make jobs more robust. Thanks to skip and retry, you'll have fewer red-light screens in the morning. But crashes are inevitable. What do you say when a job ran all night and crashed two minutes before reaching the end? If you answer, "I restart it and wait another day," keep on reading, as the next section will teach you that you can answer, "I restart it and it's going to take two minutes."

9.4 Restart on error

Ok, so your job is running, there are some transient errors, retry comes to the rescue, but these errors aren't that transient after all. The job ends up skipping the errors. Is the job finished? Not yet, more errors come up and the job finally reaches the skip limit. Spring Batch must fail the job! Despite all of our bulletproofing techniques, jobs can't dodge bullets forever and can fail. Can't someone honor the exchange format you spent weeks to establish?

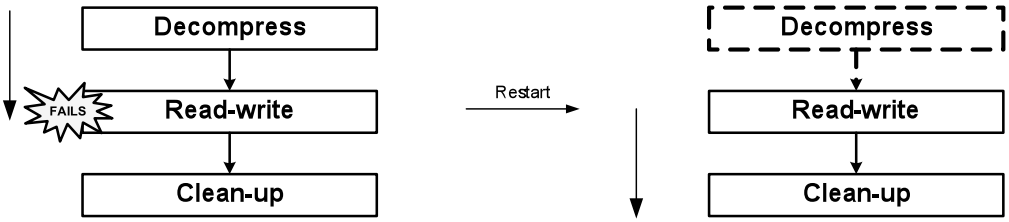

Figure 9.8 If a job fails in the middle of processing, Spring Batch can restart it exactly where it left off.

There's still hope, because Spring Batch lets you restart a job exactly where it left off. This is useful if the job was running for hours and was getting close to the end when it failed. Figure 9.8 illustrates a new execution of the import products job that continues processing where the previous execution failed.

9.4.1 How to enable restart between job executions

How does Spring Batch know where to restart a job execution? Because of the metadata it maintains for each job execution. If you want to benefit from restart with Spring Batch, you need a persistent implementation for the job repository. This enables restart across job executions, even if these executions aren't part of the same Java process. Chapter 3 shows how to configure a persistent job repository and illustrates a simple restart scenario. Chapter 3 also discusses Spring Batch metadata and job executions, as figure 9.9 shows.

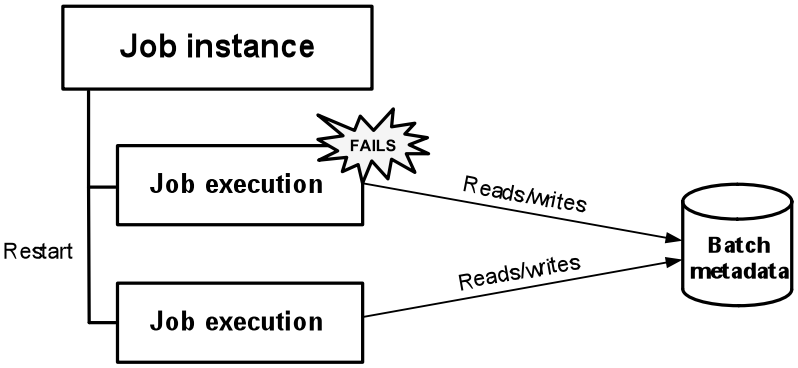

Figure 9.9 Restart is possible thanks to batch metadata that Spring Batch maintains during job executions.

Spring Batch has a default behavior for restart, but because there's no one-size-fits-all solution for batch jobs, it provides hooks to control exactly how a restarted job execution should behave. Let's focus first on the default restart behavior.

DEFAULT RESTART BEHAVIOR

Spring Batch uses the following defaults to restart jobs:

- You can only restart a failed job execution – this seems obvious but has the following implications: you must provide the job launcher with the job and the exact same job parameters as the failed execution you want to restart. Using Spring Batch terminology: when using restart, you start a new job execution of an existing, not completed job instance.
- You can restart any job – this means that you can start a new execution for any failed job instance. You can disable restart for a specific job, but you need to disable it explicitly.
- A job restarts exactly where the last execution left off – this implies that the job skips already completed steps.
- You can restart a job as many times as you want – well, almost, the limitation being a couple of billions of restarts.

You can override the defaults and Spring Batch lets you change the restart behavior. Table 9.4 summarizes the restart settings available in the configuration of a job. The default for these settings match the default behavior we described.

Table 9.4 Configuration of the restart behavior

Attribute	XML element	Possible values	Description
`restartable`	`job`	`true` / `false`	Whether the job can be restarted. Default is `true`.
`allow-start-if-complete`	`tasklet`	`true` / `false`	Whether a step should be started even if it's already completed. Default is `false`.
`start-limit`	`tasklet`	Integer value.	The number of times a step can be started. Default is `Integer.MAX_VALUE`.

Let's learn more about restart options in Spring Batch by covering some typical scenarios.

9.4.2 No restart please!

The simplest restart option is *no restart*! When a job is sensitive and you want to examine each failed execution closely, preventing restarts is useful. After all, a command line mistake or an improperly configured scheduler can easily restart a job execution. Forbid restart on jobs that are just unable to restart with correct semantics. Forbidding an accidental restart can prevent a job from processing data again, potentially corrupting a database.

To disable restart, set the attribute `restartable` to `false` on the job element:

```
<job id="importProductsJob"
     restartable="false">
```

```
    (...)
  </job>
```

Remember that jobs are restartable by default. If you're worried that you'll forget that, set the `restartable` flag explicitly on all your jobs.

Restart is a nice feature, so let's assume now that our jobs are restartable and explore more scenarios.

9.4.3 Whether or not to restart already completed steps

Remember our import products job: it consists of two steps, the first one decompresses a ZIP archive and the second one reads products from the uncompressed file and writes into the database. Imagine the first step succeeds and the second step fails after several chunks. When restarting the job instance, should we re-execute the first step or not? Figure 9.10 illustrates both alternatives. There's no definitive answer to such questions, this is a business decision, which determines how to handle failed executions.

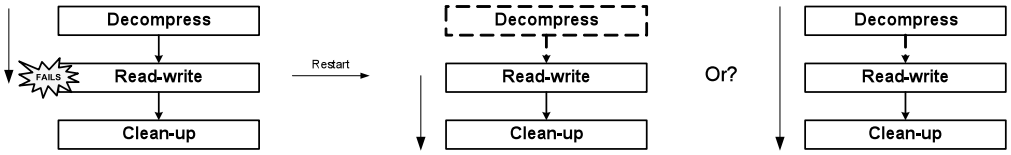

Figure 9.10 Spring Batch lets you choose if it should re-execute already-completed steps on restart. Spring Batch does not re-execute already-completed steps by default.

When is skipping the first, already-completed step, a good choice? If you check the Spring Batch logs and fix the uncompressed input file after the failure, restarting directly on the second step is the way to go. The chunk-oriented step should complete correctly, or at least not fail for the same reason. If you stick to this scenario, you have nothing to do: skipping already completed steps is the default restart behavior.

Let's consider now re-executing the first, already completed step on restart. When the batch operator sees that the execution failed during the second step, their reaction can be to send the log to the creator of the archive and tell them to provide a correct one. In this case, we should restart the import for this specific job instance with a new archive. So re-executing the first step to decompress the new archive makes sense. The second step would then execute and restart exactly on the line where it left off (as long as its item reader is able to do so and assuming the input has no lines removed, moved, or added.)

To re-execute a completed step on a restart, set the `allow-start-if-complete` flag to `true` on the `tasklet` element, as shown in the following snippet:

```
<job id="importProductsJob">
  <step id="decompressStep" next="readWriteProductsStep">
    <tasklet allow-start-if-complete="true">                    #A
      (...)
    </tasklet>
```

```
    </step>
    <step id="readWriteProductsStep" next="cleanStep">
      <tasklet>
        (...)
      </tasklet>
    </step>
    <step id="cleanStep">
      <tasklet>
        (...)
      </tasklet>
    </step>
  </job>
```

#A Sets step to re-execute on restart

Restarting a job is like many things, don't do it too much. Let's see how to avoid restarting a job indefinitely.

9.4.4 Limiting the number of restarts

Repeatedly restarting the same job instance can mean there's something wrong and we should simply give up with this job instance. That's where the restart limit comes in: you can set the number of times a step can be started for the same job instance. If you reach this limit, Spring Batch will forbid any new execution of the step for the same job instance.

You set the start limit at the step level, using the `start-limit` attribute on the `tasklet` element. The following snippet shows how to set a start limit on the second step of our import products job:

```
  <job id="importProductsJob">
    <step id="decompressStep" next="readWriteProductsStep">
      <tasklet>
        (...)
      </tasklet>
    </step>
    <step id="readWriteProductsStep">
      <tasklet start-limit="3">                                    #A
        (...)
      </tasklet>
    </step>
  </job>
```

#A Sets start limit on step

Let's see a scenario where the start limit is useful. We launch a first execution of the import products job. The first decompression step succeeds but the second step fails after a while. We start the job again. This second execution starts directly where the second step left off (the first step completed and we didn't ask to execute it again on restart. The second execution also fails and a third execution fails as well.) On the fourth execution – we're stubborn – Spring Batch sees we reached the start limit (3) for the step and doesn't even try to execute the second step again. The whole job execution fails and the job instance never completes. We need to move on and create a new job instance.

Spring Batch can restart a job exactly at the step where it left off. Can we push restart further and restart a chunk-oriented step exactly on the item where it failed?

9.4.5 Restarting in the middle of a chunk-oriented step

When a job execution fails in the middle of a chunk-oriented step and has already processed a large amount of items, you probably don't want to reprocess these items again on restart. Reprocessing wastes time and can duplicate data or transactions, which could have dramatic side effects. Spring Batch can restart a chunk-oriented step exactly on the chunk where the step failed, as shown in figure 9.11, where the item reader restarts on the exact same input line the previous execution failed.

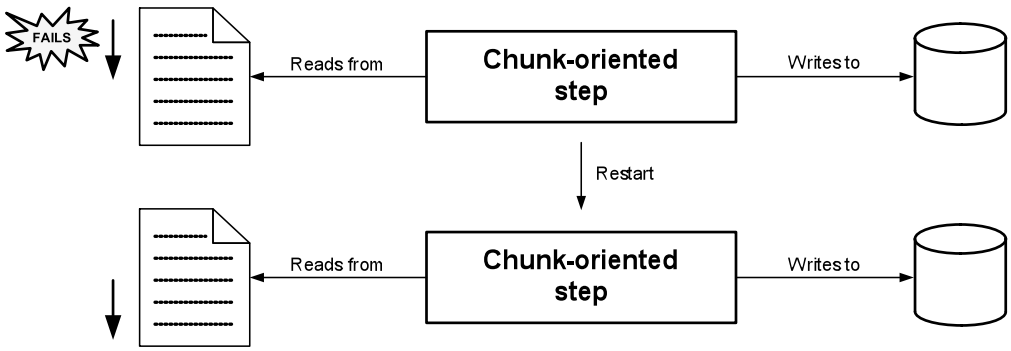

Figure 9.11 A chunk-oriented step can restart exactly where it left off. The figure shows an item reader that restarts on the line where the previous execution failed (it assumes the line has been corrected.) To do so, the item reader uses batch metadata to store its state.

The item reader drives a chunk-oriented step and provides the items to process and write. The item reader is in a charge when it comes to restarting a chunk-oriented step. Again, the item reader knows where it failed in the previous execution thanks to metadata stored in the step execution context. There's no magic here: the item reader must track what it is doing and use this information in case of failure.

> **NOTE** Item writers can also be restartable. Imagine a file item writer that will directly move to the end of the written file on a restart.

A restartable item reader can increment a counter for each item it reads, and store the value of the counter each time a chunk is committed. In case of failure and on restart, the item reader queries the step execution context for the value of the counter. Spring Batch helps by storing the step execution context between executions but the item reader must implement the restart logic. After all, Spring Batch has no idea of what the item reader is doing: reading lines from a flat file, reading rows from a database, and so on.

WARNING When using a counter-based approach for restart, you assume that the list of items doesn't change between executions (no new or deleted items and the order stays the same.)

Most of the item readers that Spring Batch provides are restartable. You should always carefully read the Javadoc of the item reader you're using to know how it behaves on restart. How the item reader implements its restart behavior can also have important consequences. For example, an item reader may not be thread-safe because of its restart behavior, which prevents multi-threading in the reading phase. Chapter 13 covers how to scale Spring Batch jobs and the impacts of multi-threading on a chunk-oriented step.

What if you write your own item reader and you want it to be restartable? You need not only to read the items but also to access the step execution context to store the counter and query it in case of failure. Spring Batch provides a convenient interface – `ItemStream` – that defines a contract to interact with the execution context in key points of the step lifecycle.

Let's take an example where an item reader returns the files in a directory. Listing 9.14 shows the code of the `FilesInDirectoryItemReader` class. This item reader implements the `ItemStream` interface to store its state periodically in the step execution context.

Listing 9.14 Implementing ItemStream to make an item reader restartable

```
package com.manning.sbia.ch09.restart;

import java.io.File;
import java.io.FileFilter;
import org.apache.commons.io.filefilter.FileFilterUtils;
import org.springframework.batch.item.ExecutionContext;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemStream;
import org.springframework.batch.item.ItemStreamException;
import org.springframework.batch.item.NonTransientResourceException;
import org.springframework.batch.item.ParseException;
import org.springframework.batch.item.UnexpectedInputException;

public class FilesInDirectoryItemReader
       implements ItemReader<File>, ItemStream {

  private File [] files;

  private int currentCount;

  private String key = "file.in.directory.count";

  public void setDirectory(String directory) {
    this.files = new File(directory).listFiles(                    #1
      (FileFilter) FileFilterUtils.fileFileFilter()                #1
    );                                                             #1
  }

  @Override                                                        #2
```

```
  public void open(ExecutionContext executionContext)                  #2
      throws ItemStreamException {                                     #2
    currentCount = executionContext.getInt(key, 0);                    #2
  }                                                                    #2

  @Override
  public File read() throws Exception, UnexpectedInputException,
      ParseException, NonTransientResourceException {
    int index = ++currentCount - 1;                                    #3
    if(index == files.length) {
      return null;
    }
    return files[index];
  }

  @Override                                                            #4
  public void update(ExecutionContext executionContext)                #4
      throws ItemStreamException {                                     #4
    executionContext.putInt(key, currentCount);                        #4
  }                                                                    #4

  @Override
  public void close() throws ItemStreamException { }

}
```

#1 Initializes file array to read from
#2 Initializes file array counter
#3 Increments counter on read
#4 Stores counter in step context

The reader implements both the `ItemReader` (`read` method) and `ItemStream` (`open`, `update`, and, `close` methods) interfaces. The code at #1 initializes the file array to read files; where you would typically set it from a Spring configuration file. When the step begins, Spring Batch calls the `open` method first, in which we initialize the counter at #2. We retrieve the counter value from the execution context, with a key we defined. On the first execution, there is no value for this key, so the counter value is zero. On a restart, we get the last value we stored in the previous execution. This allows us to start exactly where we left off. In the `read` method, we increment the counter at #3 and return the corresponding file from the file array. Spring Batch calls the `update` method just before saving the execution context. This typically happens before a chunk is committed. In `update`, we have a chance to store the state of our reader, the value of the counter (#4). `ItemStream` provides the `close` method, to clean up any resources the reader has opened (like a file stream if the reader reads from a file). We leave the method empty, as we have nothing to close.

Listing 9.14 shows you the secret to restarting in a chunk-oriented step. You can achieve this thanks to the `ItemStream` interface. An `ItemStream` is one kind of listener that Spring Batch provides: you can use the interface for item processors, writers, and on plain steps, not only chunk-oriented steps. To enable restart, `ItemStream` defines a convenient

contract to store the state of a reader at key points in a chunk-oriented step. Note that Spring Batch automatically registers an item reader that implements `ItemStream`.

> **NOTE** We implement the `FilesInDirectoryItemReader` class mainly to illustrate creating a custom, restartable item reader. If you want an item reader to read files, look at the more powerful `MultiResourceItemReader` provided by Spring Batch.

This ends our tour of restart in Spring Batch. Remember that Spring Batch can restart a job instance where the last execution left off thanks to the metadata it stores in the job repository. Spring Batch has reasonable defaults for restart, but you can override them, to re-execute an already completed step or limit the number of executions of a step. Restarting in the middle of a chunk-oriented step is also possible, if the item reader stores its state periodically in the execution context. To use this feature, it is best to implement the `ItemStream` interface.

Remember that restart only makes sense when a job execution fails. You can configure restart to avoid re-processing, potentially avoiding data corruption issues. Restart also avoids wasting time and processes the remaining steps of a failed job execution. Skip and retry are techniques to use before relying on restart. Skip and retry allow jobs to handle errors safely and avoid abrupt failures.

Congratulations for getting through this chapter! You're now ready to make your Spring Batch jobs bulletproof. It's time to wrap everything up.

9.5 Summary

Spring Batch has built-in support to make jobs more robust and reliable. Spring Batch jobs can meet the requirements of reliability, robustness, and traceability, which are essential for automatic processing of large amount of data. This chapter covered a lot of material, but we can summarize this material as follows:

- Always think about failure scenarios. Don't hesitate to write tests to simulate these scenarios and check that your jobs behave correctly.
- Use skip for deterministic, non-fatal exceptions.
- Use retry for transient, non-deterministic errors, like concurrency exceptions.
- Use listeners to log errors.
- Make a job restartable in case of failure if you're sure it won't corrupt data on restart. Many Spring Batch components are already restartable and you can implement restartability by using the execution context.
- Disable restart on a job that could corrupt data on a restart.

There's another key point to consider when you want to implement bulletproof jobs: transaction management. Proper transaction management is essential to a batch application because an error during processing can corrupt a database. In such cases, the application

can trigger a rollback to put the database back in a consistent state. The next chapter covers transactions in Spring Batch applications and is the natural transition after the coverage of the bulletproofing techniques in this chapter. So keep on reading for extra bulletproof jobs!