Spring Batch IN ACTION

Arnaud Cogoluègnes
Thierry Templier
Gary Gregory
Olivier Bazoud

SAMPLE CHAPTER

MANNING

Spring Batch in Action

by Arnaud Cogoluègnes,
Thierry Templier,
Gary Gregory,
and Olivier Bazoud

Chapter 1

brief contents

Part .

Background

What is Spring Batch? What is it good for? Is it the right tool for you? You'll find the answers to these questions in the next two chapters. Of course, you won't be a Spring Batch expert by the end of this first part, but you'll have a good foundation and understanding of all the features in Spring Batch.

Chapter 1 provides an over view of batch applications and Spring Batch. T o follow the In Action tradition, we also show you how to implement a real-world batch job with Spring Batch. This introduction not only covers how Spring Batch handles the classical read-process-write pattern for large amounts of data but also shows you the techniques used to make a job more robust, like skipping invalid lines in a flat file.

Chapter 2 clearly defines the domain language used in batch applications and explains how Spring Batch captures the essence of batch applications. What are a job, a step, and a job execution? Chapter 2 covers all of this and introduces how Spring Batch tracks the execution of jobs to enable monitoring and restart on failure.

1 Introducing Spring Batch

This chapter covers

- Understanding batch applications in today's architectures
- Describing Spring Batch's main features
- Efficiently reading and writing data
- Implementing processing inside a job with Spring Batch
- Testing a Spring Batch job

Batch applications are a challenge to write, and that's why Spring Batch was created: to make them easier to write but also faster, more robust, and reliable. What are batch applications? Batch applications process large amounts of data without human intervention. You'd opt to use batch applications to compute data for generating monthly financial statements, calculating statistics, and indexing files. You're about to discover more about batch applications in this chapter. You'll see why their requirements—large volumes of data, performance, and robustness—make them a challenge to implement correctly and efficiently. Once you understand the big picture, you'll be ready to meet Spring Batch and its main features: helping to efficiently process data with various types of technologies—databases,

files, and queues. We also honor the *In Action* series by implementing a real-world Spring Batch job. By the end of this first chapter, you'll have an overview of what Spring Batch does, and you'll be ready to implement your first job with Spring Batch. Let's get started with batch applications!

1.1 What are batch applications?

The most common scenario for a batch application is exporting data to files from one system and processing them in another. Imagine you want to exchange data between two systems: you export data as files from system A and then import the data into a database on system B. Figure 1.1 illustrates this example.

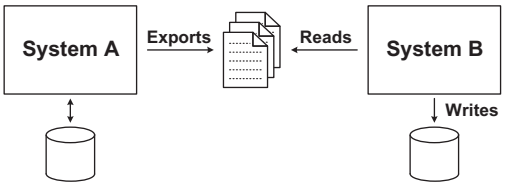

Figure 1.1 A typical batch application: system A exports data to flat files, and system B uses a batch process to read the files into a database.

A batch application processes data automatically, so it must be robust and reliable because there is no human interaction to recover from an error. The greater the volume of data a batch application must process, the longer it takes to complete. This means you must also consider performance in your batch application because it's often restricted to execute within a specific time window. Based on this description, the requirements of a batch application are as follows:

- *Large data volume*—Batch applications must be able to handle large volumes of data to import, export, or compute.
- *Automation*—Batch applications must run without user interaction except for serious problem resolution.
- *Robustness*—Batch applications must handle invalid data without crashing or aborting prematurely.
- *Reliability*—Batch applications must keep track of what goes wrong and when (logging, notification).
- *Performance*—Batch applications must perform well to finish processing in a dedicated time window or to avoid disturbing any other applications running simultaneously.

How batch applications fit in today's software architectures

Performing computations and exchanging data between applications are good examples of batch applications. But are these types of processes relevant today? Computation-based processes are obviously relevant: every day, large and complex calculations take place to index billions of documents, using cutting-edge algorithms like MapReduce. For data exchange, message-based solutions are also popular, having the advantage over batch applications of being (close to) real time. Although messaging is a powerful design pattern, it imposes its own particular set of requirements in terms of application design and implementation.

(continued)

Clearly, messaging isn't a silver bullet, and you should apply it thoughtfully. Note that batch jobs and messaging aren't mutually exclusive solutions: you can use messaging to exchange data and still need batch applications to process the data with the same reliability and robustness requirements as the rest of your application stack. Even in our event- and notification-driven world, batch applications are still relevant!

How does Spring Batch fit in the landscape of batch applications? The next section introduces the Spring Batch framework and its main features. You'll see how Spring Batch helps meet the requirements of batch applications by providing ready-to-use components and processing large amounts of data in an efficient manner.

1.2 Meet Spring Batch

The goal of the Spring Batch project is to provide an open source batch-oriented framework that effectively addresses the most common needs of batch applications. The Spring Batch project was born in 2007 out of the collaboration of Accenture and SpringSource. Accenture brought its experience gained from years of working on proprietary batch frameworks and SpringSource brought its technical expertise and the proven Spring programming model. Table 1.1 lists the main features of Spring Batch.

Table 1.1 Main features of Spring Batch

Feature	Description
Spring Framework foundations	Benefits from enterprise support, dependency injection, and aspect-oriented programming
Batch-oriented processing	Enforces best practices when reading and writing data
Ready-to-use components	Provides components to address common batch scenarios (read and write data to and from databases and files)
Robustness and reliability	Allows for declarative skipping and retry; enables restart after failure

By using Spring Batch, you directly benefit from the best practices the framework enforces and implements. You also benefit from the many off-the-shelf components for the most popular formats and technologies used in the software industry. Table 1.2 lists the storage technologies that Spring Batch supports out of the box.

Table 1.2 Read-write technologies supported by Spring Batch

Data source type	Technology	Description
Database	JDBC	Leverages paging, cursors, and batch updates
Database	Hibernate	Leverages paging and cursors
Database	JPA (Java Persistence API)	Leverages paging

Table 1.2 **Read-write technologies supported by Spring Batch *(continued)***

Data source type	Technology	Description
Database	iBATIS	Leverages paging
File	Flat file	Supports delimited and fixed-length flat files
File	XML	Uses StAX (Streaming API for XML) for parsing; builds on top of Spring OXM; supports JAXB (Java Architecture for XML Binding), XStream, and Castor

As you can see in table 1.2, Spring Batch supports many technologies out of the box, making the framework quite versatile. We study this support thoroughly in chapters 5 and 6.

Spring Batch isn't a scheduler!

Spring Batch drives batch jobs (we use the terms job, batch, and process interchangeably) but doesn't provide advanced support to launch them according to a schedule. Spring Batch leaves this task to dedicated schedulers like Quartz and cron. A scheduler triggers the launching of Spring Batch jobs by accessing the Spring Batch runtime (like Quartz because it's a Java solution) or by launching a dedicated JVM process (in the case of cron). Sometimes a scheduler launches batch jobs in sequence: first job A, and then job B if A succeeded, or job C if A failed. The scheduler can use the files generated by the jobs or exit codes to orchestrate the sequence. Spring Batch can also orchestrate such sequences itself: Spring Batch jobs are made of steps, and you can easily configure the sequence by using Spring Batch XML (covered in chapter 10). This is an area where Spring Batch and schedulers overlap.

How does Spring Batch meet the requirements of robustness and reliability of batch applications?

1.2.1 Robustness and reliability

Should a whole batch fail because of one badly formatted line? Not always. The decision to skip an incorrect line or an incorrect item is declarative in Spring Batch. It's all about configuration.

What happens if you restart a failed batch job? Should it start from the beginning—potentially processing items again and corrupting data—or should it be able to restart exactly where it left off? Spring Batch makes the latter easy: components can track everything they do, and the framework provides them with the execution data on restart. The components then know where they left off and can restart processing at the right place.

Spring Batch also addresses the robustness and reliability requirements. Chapter 2 provides an overview of restart, and chapter 8 covers robustness thoroughly.

Another requirement of batch applications is performance. How does Spring Batch meet the performance requirement?

1.2.2 Scaling strategies

Spring Batch processes items in chunks. We cover chunk processing later in this chapter, but here is the idea: a job reads and writes items in small chunks. Chunk processing allows streaming data instead of loading all the data in memory. By default, chunk processing is single threaded and usually performs well. But some batch jobs need to execute faster, so Spring Batch provides ways to make chunk processing multithreaded and to distribute processing on multiple physical nodes.

Chapter 13 thoroughly discusses the scaling strategies in Spring Batch. Let's take a look at one of these strategies: partitioning.

Partitioning splits a step into substeps, each of which handles a specific portion of the data. This implies that you know the structure of the input data and that you know in advance how to distribute data between substeps. Distribution can take place by ranges of primary key values for database records or by directories for files. The substeps can execute locally or remotely, and Spring Batch provides support for multithreaded substeps. Figure 1.2 illustrates partitioning based on filenames: A through D, E through H, and so on, up to Y through Z.

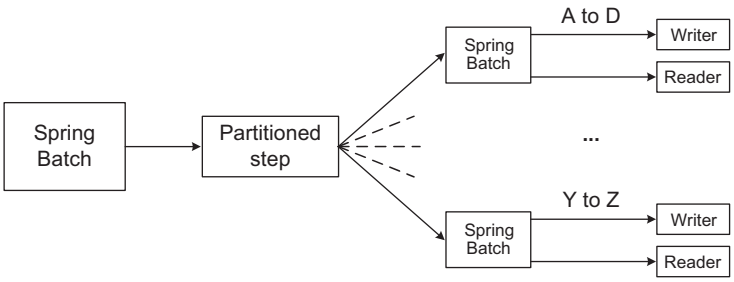

Figure 1.2 Scaling by partitioning: a single step partitions records and autonomous substeps handle processing.

Spring Batch and grid computing

When dealing with large amounts of data—petabytes (10^{15})—a popular solution to scaling is to divide the enormous amounts of computations into smaller chunks, compute them in parallel (usually on different nodes), and then gather the results. Some open source frameworks (Hadoop, GridGain, and Hazelcast, for example) have appeared in the last few years to deal with the burden of distributing units of work so that developers can focus on developing the computations themselves. How does Spring Batch compare to these grid-computing frameworks? Spring Batch is a lightweight solution: all it needs is the Java runtime installed, whereas grid-computing frameworks need a more advanced infrastructure. As an example, Hadoop usually works on top of its own distributed file system, HDFS (Hadoop Distributed File System). In terms of features, Spring Batch provides a lot of support to work with flat files, XML files, and relational databases.

(continued)

Grid-computing frameworks usually don't provide such high-level processing support.

Spring Batch and grid-computing frameworks aren't incompatible: at the time of this writing, projects integrating both technologies are appearing (Spring Hadoop, at https://github.com/SpringSource/spring-hadoop, is an example).

This ends our tour of the Spring Batch framework. You now have a good overview of the most important features of Spring Batch and the benefits it brings to your applications. Let's move on to the more practical part of this chapter with the implementation of a real-world batch job using Spring Batch.

1.3 Introducing the case study

This section introduces a real application that we use throughout this book to illustrate the use of Spring Batch: an online store application. This use case starts out small and simple but remains realistic in terms of technical requirements. It not only demonstrates Spring Batch features but also illustrates how this use case fits into the enterprise landscape.

By implementing this use case using Spring Batch, you gain a practical understanding of the framework: how it implements efficient reading and writing of large volumes of data, when to use built-in components, when to implement your own components, how to configure a batch job 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 this book to find what you need for your batch applications.

1.3.1 The online store application

The ACME Corporation wants to expand its business by selling its products on the web. To do so, ACME chooses to build a dedicated online store application. ACME will use batch jobs to populate the online store database with the catalog from its internal proprietary system, as shown in figure 1.3. The system will process data every night to insert new products in the catalog or update existing products.

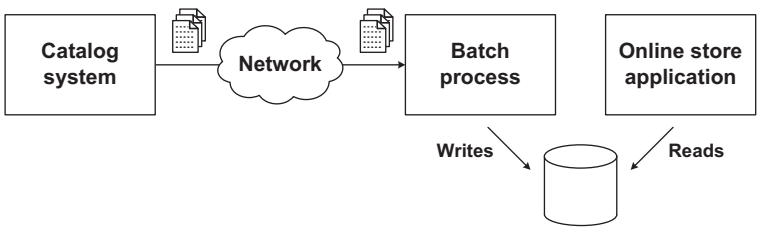

Figure 1.3 Thanks to this new application, anyone can buy ACME's products online. The system sends catalogs to a server where a batch process reads them and writes product records into the online store database.

That's it for the big picture, but you should understand why ACME decided to build an online store in the first place and populate it using batch processes.

1.3.2 Why build an online store with batch jobs?

Why did ACME choose to build an online, web-based application to sell its products? As we mentioned, this is the best way for ACME to expand its business and to serve more customers. Web applications are easy to deploy, easy to access, and can provide a great user experience. ACME plans to deploy the online store application to a local web hosting provider rather than hosting it on its own network. The first version of the online store will provide a simple but efficient UI; ACME focuses on the catalog and transactions first, before providing more features and a more elaborate UI.

Next, why did ACME choose to shuttle data from one system to the other instead of making its onsite catalog and the online store communicate directly? The software that powers the catalog has an API, so why not use it? The main reason is security: as illustrated in figure 1.4, ACME's own network hosts the catalog system, and the company doesn't want to expose the catalog system to the outside world directly, even via another application. This precaution is rather drastic, but that's how things are done at ACME.

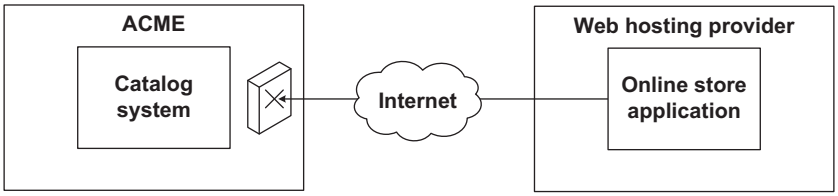

Figure 1.4 Because ACME doesn't want its internal catalog system to be directly accessible from the outside world, it doesn't allow the two applications to communicate directly and exchange data.

Another reason for this architecture is that the catalog system's API and data format don't suit the needs of the online store application: ACME wants to show a summarized view of the catalog data to customers without overwhelming them with a complex catalog structure and supplying too many details. You could get this summarized catalog view by using the catalog system's API, but you'd need to make many calls, which would cause performance to suffer in the catalog system.

To summarize, a mismatch exists between the view of the data provided by the catalog system and the view of the data required by the online store application. Therefore, an application needs to process the data before exposing it to customers through the online store.

1.3.3 Why use batch processes?

The online store application scenario is a good example of two systems communicating to exchange data. ACME updates the catalog system throughout the day, adding new products and updating existing products. The online store application doesn't need to expose live data because buyers can live with day-old catalog information.

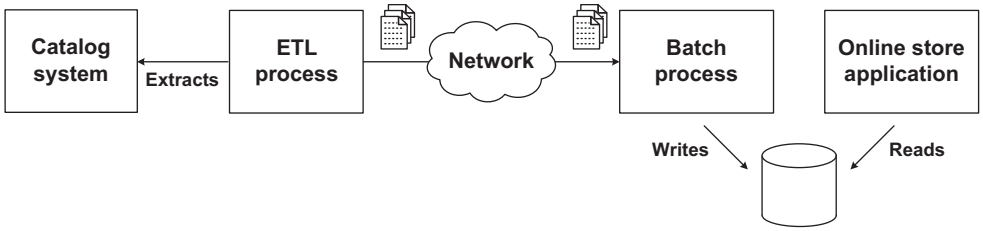

Figure 1.5 An extract, transform, and load (ETL) process extracts and transforms the catalog system data into a flat file, which ACME sends every night to a Spring Batch process. This Spring Batch process is in charge of reading the flat file and importing the data into the online store database.

Therefore, a nightly batch process updates the online store database, using flat files, as shown in figure 1.5.

Extract, transform, and load (ETL)

Briefly stated, ETL is a process in the database and data-warehousing world that performs the following steps:

1. Extracts data from an external data source
2. Transforms the extracted data to match a specific purpose
3. Loads the transformed data into a data target: a database or data warehouse

Many products, both free and commercial, can help create ETL processes. This is a bigger topic than we can address here, but it isn't always as simple as these three steps. Writing an ETL process can present its own set of challenges involving parallel processing, rerunnability, and recoverability. The ETL community has developed its own set of best practices to meet these and other requirements.

In figure 1.5, an ETL process creates the flat file to populate the online store database. It extracts data from the catalog system and transforms it to produce the view expected by the online store application. For the purpose of our discussion, this ETL process is a black box: it could be implemented with an ETL tool (like Talend) or even with another Spring Batch job. We focus next on how the online store application reads and writes the catalog's product information.

1.3.4 The import product use case

The online store application sells products out of a catalog, making the product a main domain concept. The import product batch reads the product records from a flat file created by ACME and updates the online store application database accordingly. Figure 1.6 illustrates that reading and writing products is at the core of this batch job, but it contains other steps as well.

The read-write step forms the core of the batch job, but as figure 1.6 shows, this isn't the only step. This batch job consists of the following steps:

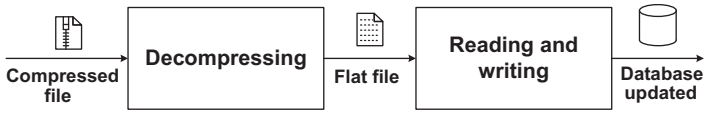

Figure 1.6 The Spring Batch job consists of the following steps: decompression and read-write.

1. *Decompression*—Decompresses the archive flat file received from the ACME network. The file is compressed to speed up transfer over the internet.
2. *Reading and writing*—The flat file is read line by line and then inserted into the database.

This batch process allows us to introduce the Spring Batch features displayed in table 1.3.

Table 1.3 Spring Batch features introduced by the import catalog job

Batch process step	Spring Batch feature
Decompression	Custom processing in a job (but not reading from a data store and writing to another)
Read-write	Reading a flat file Implementing a custom database writing component Skipping invalid records instead of failing the whole process
Configuration	Leveraging Spring's lightweight container and Spring Batch's namespace to wire up batch components Using the Spring Expression Language to make the configuration more flexible

Rather than describe each of Spring Batch's features in the order in which they appear as batch job steps, we start with the core of the process: reading and writing the products. Then we see how to decompress the incoming file before making the process more robust by validating the input parameters and choosing to skip invalid records to avoid the whole job failing on a single error.

1.4 Reading and writing the product data

Reading and writing the product catalog is at the core of the Spring Batch job. ACME provides the product catalog as a flat file, which the job needs to import into the online store database. Reading and writing is Spring Batch's sweet spot: for the import product job, you only have to configure one Spring Batch component to read the content of the flat file, implement a simple interface for the writing component, and create a configuration file to handle the batch execution flow. Table 1.3 lists the Spring Batch features introduced by the import catalog job. Let's start by using Spring Batch to implement the read-write use case.

1.4.1 Anatomy of the read-write step

Because read-write (and copy) scenarios are common in batch applications, Spring Batch provides specific support for this use case. Spring Batch includes many ready-to-use components to read from and write to data stores like files and databases. Spring

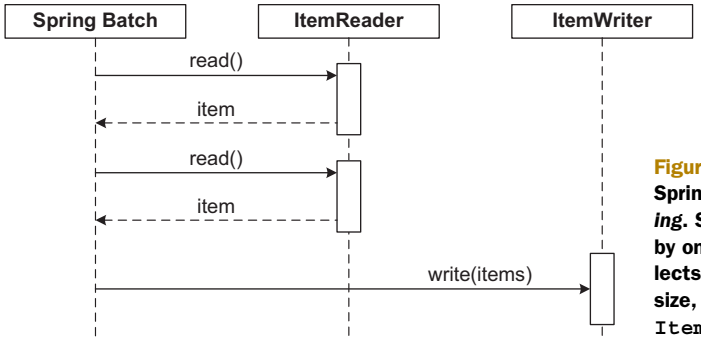

Figure 1.7 In read-write scenarios, Spring Batch uses *chunk processing*. Spring Batch reads items one by one from an `ItemReader`, collects the items in a chunk of a given size, and sends that chunk to an `ItemWriter`.

Batch also includes a batch-oriented algori thm to handle the execution flow , called *chunk processing*. Figure 1.7 illustrates the principle of chunk processing.

Spring Batch handles read-write scenarios by managing an `ItemReader` and an `ItemWriter`. Using chunk processing, Spring Batc h collects items one at a time from the item reader into a configurable-sized chunk. Spring Batch then sends the chunk to the item writer and goes back to usin g the item reader to create another chunk, and so on, until the input is exhausted.

CHUNK PROCESSING Chunk processing is particularly well suited to handle large data operations because a job ha ndles items in small chunks instead of processing them all at once. Practically speaking, a large file won't be loaded in memory; instead it's streamed, whic h is more efficient in terms of memory consumption. Chunk processing allo ws more flexibility to manage the data flow in a job. Spring Batch also handles transactions and errors around read and write operations.

Spring Batch provides an optional processing step in chunk processing: a job can process (transform) read items be fore sending them to the `ItemWriter`. The ability to process an item is useful when you don' t want to write an item as is. The component that handles this transformation is an implementation of the `ItemProcessor` interface. Because item processing in Spring Batch is optional, the illustration of chunk processing shown in figure 1.7 is still va lid. Figure 1.8 illustrates chunk processing combined with item processing.

What can you do in an `ItemProcessor`? You can perform any transformations you need on an item before Sp ring Batch sends it to the `ItemWriter`. This is where you implement the logic to transform the data from the input form at into the format expected by the target system. Spring Batc h also lets you validate and filter input items. If you return `null` from the `ItemProcessor` method `process`, processing for that item stops and Spring Batch won't insert the item in the database.

NOTE Our read-write use case doesn't have an item-processing step.

The following listing shows the definition of the chunk-processing inter faces `ItemReader`, `ItemProcessor`, and `ItemWriter`.

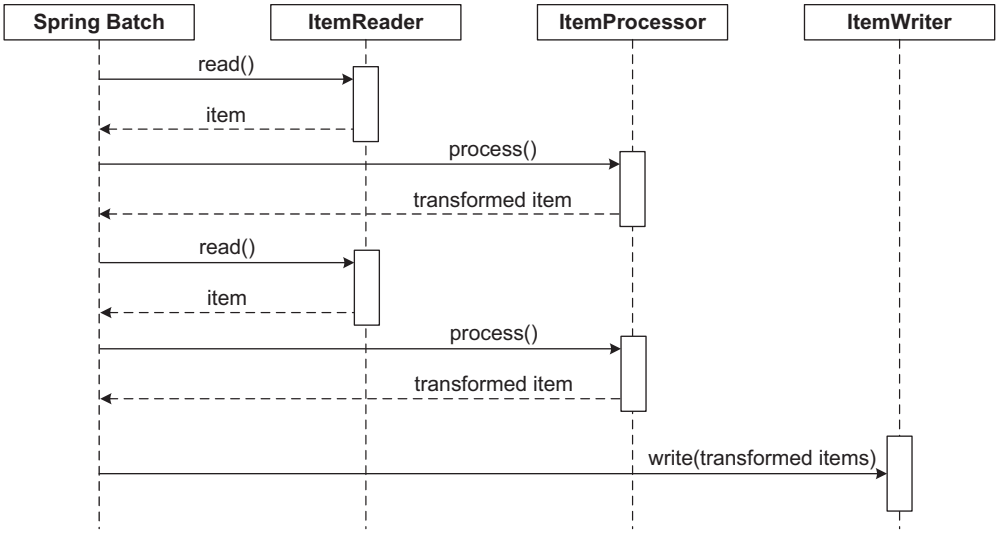

Figure 1.8 Chunk processing combined with item processing: an item processor can transform input items before calling the item writer.

Listing 1.1 Spring Batch interfaces for chunk processing

```
package org.springframework.batch.item;

public interface ItemReader<T> {

  T read() throws Exception, UnexpectedInputException,
                 ParseException,
                 NonTransientResourceException;

}
```

Reads item

```
package org.springframework.batch.item;

public interface ItemProcessor<I, O> {

  O process(I item) throws Exception;

}
```

Transforms item (optional)

```
package org.springframework.batch.item;

import java.util.List;

public interface ItemWriter<T> {

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

}
```

Writes a chunk of items

In chapters 5 and 6, we respecti vely cover all implementations of `ItemReader` and `ItemWriter` provided by Spring Batch. Chapter 7 covers the processing phase used to transform and filter items.

The next two subsections show how to configure the Spring Batch flat file `ItemReader` and how to write your own `ItemWriter` to handle writing products to the database.

1.4.2 Reading a flat file

Spring Batch provides the `FlatFileItemReader` class to read records from a flat file. To use a `FlatFileItemReader`, you need to configure 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 all your old boilerplate I/O code goodbye and focus on your data.

THE FLAT FILE FORMAT

The input flat file format consists of a header line and one line per product record. Here's an excerpt:

```
PRODUCT_ID,NAME,DESCRIPTION,PRICE
PR....210,BlackBerry 8100 Pearl,A cell phone,124.60
PR....211,Sony Ericsson W810i,Yet another cell phone!,139.45
PR....212,Samsung MM-A900M Ace,A cell phone,97.80
PR....213,Toshiba M285-E 14,A cell phone,166.20
PR....214,Nokia 2610 Phone,A cell phone,145.50
```

You may recognize this as the classic comma-separated value (CSV) format. There's nothing out of the ordinary in this flat file: for a given row, the format separates each column value from the next with a comma. Spring Batch maps each row in the flat file to a `Product` domain object.

THE PRODUCT DOMAIN CLASS

The `Product` class maps the different columns of the flat file. Note the instance variable declarations for product attributes like id, name, price, and so on, in this snippet; the getter and setter methods are excluded for brevity:

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

import java.math.BigDecimal;

public class Product {

  private String id;
  private String name;
  private String description;
  private BigDecimal price;

  (...)

}
```

> **NOTE** We use a `BigDecimal` for the product price because the Java float and double primitive types aren't well suited for monetary calculations. For example, it's impossible to exactly represent 0.1.

Let's now use the `FlatFileItemReader` to create `Product` objects out of the flat file.

CREATING DOMAIN OBJECTS WITH A FLATFILEITEMREADER

The `FlatFileItemReader` class handles all the I/O for you: opening the file, streaming it by reading each line, and closing it. The `FlatFileItemReader` class delegates the mapping between an input line and a domain object to an implementation of the `LineMapper` interface. Spring Batch provides a handy `LineMapper` implementation

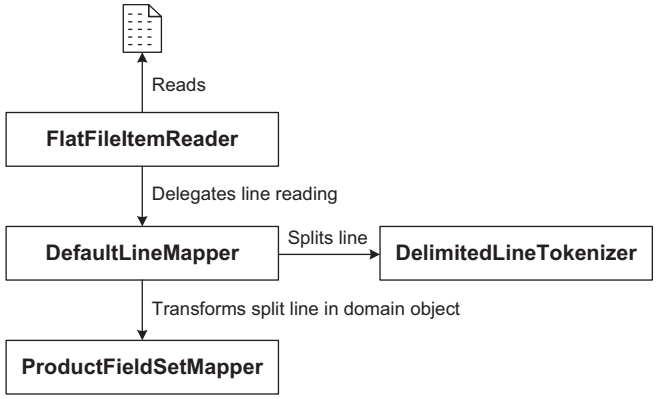

Figure 1.9 The `FlatFileItemReader` reads the flat file and delegates the mapping between a line and a domain object to a `LineMapper`. The `LineMapper` implementation delegates the splitting of lines and the mapping between split lines and domain objects.

called `DefaultLineMapper`, which delegates the mapping to other strategy inter faces. Figure 1.9 shows all of this delegation work.

That's a lot of delegation, and it means you'll have more to configure, but such is the price of reusability and flexibility . You'll be able to configure and use built-in Spring Batch components or provide your own implementations for more specific tasks.

The `DefaultLineMapper` is a typical example; it needs

- *A LineTokenizer to split a line into fields.*
  You'll use a stock Spring Batch implementation for this.
- *A FieldSetMapper to transform the split line into a domain object.*
  You'll write your own implementation for this.

You'll soon see the whole Spring configuration in listing 1.2 (`LineTokenizer` is of particular interest), but next we focus on the `FieldSetMapper` implementation to create `Product` domain objects.

IMPLEMENTING A FIELDSETMAPPER FOR PRODUCT OBJECTS

You use a `FieldSetMapper` to convert 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`. Think of it as an equivalent to the JDBC `ResultSet`: it retrieves field values and per forms conversions between `String` objects and richer objects like `BigDecimal`. The following snippet shows the `ProductFieldSetMapper` implementation:

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

```
public class ProductFieldSetMapper implements FieldSetMapper<Product> {

  public Product mapFieldSet(FieldSet fieldSet) throws BindException {
    Product product = new Product();
    product.setId(fieldSet.readString("PRODUCT_ID"));
    product.setName(fieldSet.readString("NAME"));
    product.setDescription(fieldSet.readString("DESCRIPTION"));
    product.setPrice(fieldSet.readBigDecimal("PRICE"));
    return product;
  }
}
```

The `ProductFieldSetMapper` implementation isn't rocket science, and that's exactly the point: it focuses on retrieving the data from the flat file and converts values into `Product` domain objects. We leave Spring Batch to deal with all of the I/O plumbing and efficiently reading the flat file. Notice in the `mapFieldSet` method the `String` literals `PRODUCT_ID`, `NAME`, `DESCRIPTION`, and `PRICE`. Where do these references come from? They're part of the `LineTokenizer` configuration, so let's study the Spring configuration for `FlatFileItemReader`.

CONFIGURATION OF THE FLATFILEITEMREADER

The `FlatFileItemReader` can be configured like any Spring bean using an XML configuration file, as shown in the following listing.

Listing 1.2 Spring configuration of the `FlatFileItemReader`

```
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="resource"
            value="file:./work/output/output.txt" />
  <property name="linesToSkip" value="1" />
  <property name="lineMapper">
    <bean
     class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">
        <bean class="org.springframework.batch.item.file.transform.
➥ DelimitedLineTokenizer">
          <property name="names" value="PRODUCT_ID,
➥ NAME,DESCRIPTION,PRICE" />
        </bean>
      </property>
      <property name="fieldSetMapper">
        <bean class="com.manning.sbia.ch01.batch.
➥ ProductFieldSetMapper" />
      </property>
    </bean>
  </property>
</bean>
```

(1) Skips first line

Configures tokenization (2)

In this example, the `resource` property defines the input file. Because the first line of the input file contains headers, you ask Spring Batch to skip this line by setting the property `linesToSkip` (1) to `1`. You use a `DelimitedLineTokenizer` (2) to split each input line into fields; Spring Batch uses a comma as the default separator. Then you

define the name of each field. These are the names used in the `ProductFieldSetMapper` class to retrieve values from the `FieldSet`. Finally, you inject an instance of `ProductFieldSetMapper` into the `DefaultLineMapper`.

That's it; your flat file reader is ready! Don't feel overwhelmed because flat file support in Spring Batch uses many components—that's what makes it powerful and flexible. Next up, to implement the database item writer, you need to do less configuration work but more Java coding. Let's dig in.

1.4.3 Implementing a database item writer

To update the database with product data, you have to implement your own `ItemWriter`. Each line of the flat file represents either a new product record or an existing one, so you must decide whether to send the database an `insert` or an `update` SQL statement. Nevertheless, the implementation of the `ProductJdbcItemWriter` is straightforward, as shown in the following listing.

Listing 1.3 Implementing the `ProductJdbcItemWriter`

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

import java.util.List;
import javax.sql.DataSource;
import org.springframework.batch.item.ItemWriter;
import org.springframework.jdbc.core.JdbcTemplate;
import com.manning.sbia.ch01.domain.Product;

public class ProductJdbcItemWriter implements ItemWriter<Product> {

  private static final String INSERT_PRODUCT = "insert into product "+
   "(id,name,description,price) values(?,?,?,?)";

  private static final String UPDATE_PRODUCT = "update product set "+
   "name=?, description=?, price=? where id=?";

  private JdbcTemplate jdbcTemplate;

  public ProductJdbcItemWriter(DataSource ds) {
    this.jdbcTemplate = new JdbcTemplate(ds);
  }

  public void write(List<? extends Product> items) throws Exception {
    for (Product item : items) {
      int updated = jdbcTemplate.update(
        UPDATE_PRODUCT,
        item.getName(),item.getDescription(),
        item.getPrice(),item.getId()
      );
      if (updated == 0) {
        jdbcTemplate.update(
          INSERT_PRODUCT,
          item.getId(),item.getName(),
          item.getDescription(),item.getPrice()
        );
      }
```

1 Uses JDBC template for data access

2 Tries to update a product

3 Inserts new product

```
    }
  }
}
```

The `ProductJdbcItemWriter` uses Spring's `JdbcTemplate` to interact with the database. Spring Batch creates the `JdbcTemplate` with a `DataSource` injected in the constructor (1). In the `write` method, you iterate over a chunk of products and first try to update an existing record (2). If the database tells you the `update` statement didn't update any record, you know this record doesn't exist, and you can insert it (3).

That's it for the implementation! Notice how simple it was to implement this `ItemWriter` because Spring Batch handles getting records from the `ItemReader`, creating chunks, managing transactions, and so on. Next, let's configure the database item writer.

1.4.4 Configuring a database item writer

For the item writer to be configured as a Spring bean, it needs a `DataSource`, as shown in the following XML fragment:

```
<bean id="writer"
      class="com.manning.sbia.ch01.batch.ProductJdbcItemWriter">
  <constructor-arg ref="dataSource" />
</bean>
```

You'll configure the `DataSource` later, in a separate configuration file. You use a separate file because it decouples the application configuration—the item writer—from the infrastructure configuration—the `DataSource`. By doing so, you can use the same application configuration across different environments—production and testing, for example—and switch the infrastructure configuration file.

Now that you've created the two parts of the read-write step, you can assemble them in a Spring Batch job.

1.4.5 Configuring the read-write step

Configuring the read-write step is done through Spring. The step configuration can sit next to the declaration of the reader and writer beans, as shown in the following listing.

Listing 1.4 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="importProducts"
       xmlns="http://www.springframework.org/schema/batch">
    <step id="readWriteProducts">
      <tasklet>
```

(1) **Starts job configuration**

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

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

</beans>
```

❷ Configures chunk processing

❸ Sets commit interval

The configuration file starts with the usual declaration of the namespaces and associated prefixes: the Spring namespace and Spring Batch namespace with the `batch` prefix. The Spring namespace is declared as the default namespace, so you don't need to use a prefix to use its elements. Unfortunately, this is inconvenient for the overall configuration because you must use the `batch` prefix for the batch job elements. To make the configuration more readable, you can use a workaround in XML: when you start the `job` configuration XML element ❶, you specify a default XML namespace as an attribute of the `job` element. The scope of this new default namespace is the `job` element and its child elements.

The `chunk` element ❷ configures the chunk-processing step, in a `step` element, which is itself in a `tasklet` element. In the `chunk` element, you refer to the reader and writer beans with the `reader` and `writer` attributes. The values of these two attributes are the IDs previously defined in the reader and writer configuration. Finally, ❸ the `commit-interval` attribute is set to a chunk size of 100.

Choosing a chunk size and commit interval

First, the size of a chunk and the commit interval are the same thing! Second, there's no definitive value to choose. Our recommendation is a value between 10 and 200. Too small a chunk size creates too many transactions, which is costly and makes the job run slowly. Too large a chunk size makes transactional resources—like databases—run slowly too, because a database must be able to roll back operations. The best value for the commit interval depends on many factors: data, processing, nature of the resources, and so on. The commit interval is a parameter in Spring Batch, so don't hesitate to change it to find the most appropriate value for your jobs.

You're done with the copy portion of the batch process. Spring Batch performs a lot of the work for you: it reads the products from the flat file and imports them into the database. You didn't write any code for reading the data. For the write operation, you only created the logic to insert and update products in the database. Putting these components together is straightforward thanks to Spring's lightweight container and the Spring Batch XML vocabulary.

So far, you've implemented the box labeled "Reading and writing" from figure 1.6. As you've seen, Spring Batch provides a lot of help for this common use case. The framework is even richer and more flexible because a batch process can contain any type of write operation. You'll see an example of this next, when you decompress the input file for your job, as shown in figure 1.6 in the box labeled "Decompressing."

1.5 Decompressing the input file with a tasklet

Remember that the flat file is uploaded to the online store as a compressed archive. You need to decompress this file before starting to read and write products. Decompressing a file isn't a read-write step, but Spring Batch is flexible enough to implement such a task as part of a job. Before showing you how to decompress the input file, let's explain why you must compress the products flat file.

1.5.1 Why compress the file?

The flat file containing the product data is compressed so you can upload it faster from ACME's network to the provider that hosts the online store application. Textual data, as used in the flat file, can be highly compressed, with ratios of 10 to 1 commonly achieved. A 1-GB flat file can compress to 100 MB, which is a more reasonable size for file transfers over the internet.

Note that you could encrypt the file as well, ensuring that no one could read the product data if the file were intercepted during transfer. The encryption could be done before the compression or as part of it. In this case, assume that ACME and the hosting provider agreed on a secure transfer protocol, like Secure Copy (SCP is built on top of Secure Shell [SSH]).

Now that you know why you compress the file, let's see how to implement the decompression tasklet.

1.5.2 Implementing the decompression tasklet

Spring Batch provides an extension point to handle processing in a batch process step: the `Tasklet`. You implement a `Tasklet` that decompresses a ZIP archive into its source flat file. The following listing shows the implementation of the `DecompressTasklet` class.

Listing 1.5 Implementation of decompression tasklet

```
package com.manning.sbia.ch01.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.apache.commons.io.IOUtils;
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 {

  private Resource inputResource;
  private String targetDirectory;
  private String targetFile;

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

    File targetDirectoryAsFile = new File(
      targetDirectory);
    if(!targetDirectoryAsFile.exists()) {
      FileUtils.forceMkdir(targetDirectoryAsFile);
    }
    File target = new File(targetDirectory,targetFile);
    BufferedOutputStream dest = null;
    while(zis.getNextEntry() != null) {
      if(!target.exists()) {
        target.createNewFile();
      }
      FileOutputStream fos = new FileOutputStream(
        target
      );
      dest = new BufferedOutputStream(fos);
      IOUtils.copy(zis,dest);
      dest.flush();
      dest.close();
    }
    zis.close();
    if(!target.exists()) {
      throw new IllegalStateException(
        "Could not decompress anything from the archive!");
    }
    return RepeatStatus.FINISHED;
  }
  /* setters */
  (...)

}
```

1 Implements Tasklet interface

2 Declares Tasklet parameters

3 Opens archive

4 Creates target directory if absent

5 Decompresses archive

6 Tasklet finishes

The `DecompressTasklet` class implements the `Tasklet` interface 1, which has only one method, called `execute`. The tasklet has three fields 2, which represent the archive file, the name of the directory to which the file is decompressed, and the name of the output file. These fields are set when you configure the tasklet with Spring. In the `execute` method, you open a stream to the archive file 3, create the target directory if it doesn't exist 4, and use the Java API to decompress the ZIP archive 5. Note that the `FileUtils` and `IOUtils` classes from the Apache Commons IO project are used to create the target directory and copy the ZIP entry content to the

target file (Apache Commons IO provides handy utilities to deal with files and directories). At 6, you return the `FINISHED` constant from the `RepeatStatus` enumeration to notify Spring Batch that the tasklet finished.

> **Only a data file and no metadata file in the ZIP archive?**
>
> It's common practice to have two files in a ZIP archive used for a batch job. One file contains the data to import, and the other contains information about the data to import (date, identifier, and so on). We wanted to keep things simple in our Spring Batch introduction, especially the tedious unzipping code, so our ZIP archive contains only a data file. Let's say the name of the unzipped file is made up of meaningful information such as the date and an identifier for the import.

Although the `Tasklet` interface is straightforward, its implementation includes a lot of code to deal with decompressing the file. Let's now see how to configure this tasklet with Spring.

1.5.3 Configuring the tasklet

The tasklet is configured as part of the job and consists of two changes in Spring: declare the tasklet as a Spring bean and inject it as a step in the job. To do this, you must modify the configuration you wrote for reading and writing products, as shown in the following listing.

Listing 1.6 Spring configuration of the decompress tasklet

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

<bean id="decompressTasklet"
      class="com.manning.sbia.ch01.batch.
    ➥ DecompressTasklet">
  <property name="inputResource"
            value="file:./input/input.zip" />
  <property name="targetDirectory"
            value="./work/output/" />
  <property name="targetFile"
            value="products.txt" />
</bean>
```

1 Sets tasklet in job

2 Declares tasklet bean

The configuration of a plain `Tasklet` is simpler than for a read-write step because you only need to point the `Tasklet` to the (decompression) bean 1. Note that you control

the job flow through the `next` attribute of the `step` element, which refers to the `readWriteProducts` step by ID. Chapter 10 thoroughly covers how to control the flow of Spring Batch jobs and how to take different paths, depending on how a step ends, for example. The `tasklet` element ❶ refers to the `decompressTasklet` bean, declared at ❷. If you find that the `Tasklet` bean is configured too rigidly in the Spring file (because the values are hardcoded), don't worry: we'll show you later in this chapter how to make these settings more dynamic.

You now have all the parts of the job implemented and configured: you can decompress the input archive, read the products from the decompressed flat file, and write them to the database. You're now about to see how to launch the job inside an integration test.

1.6 Testing the batch process

Batch applications are like any other applications: you should test them using a framework like JUnit. Testing makes maintenance easier and detects regressions after refactoring. Let's test, then! This section covers how to write an integration test for a Spring Batch job. You'll also learn about the launching API in Spring Batch. But don't be too impatient—we need a couple of intermediary steps before writing the test: configuring a test infrastructure and showing you a trick to make the job configuration more flexible.

> **Spring Batch and test-driven development**
>
> Good news: Spring Batch and test-driven development are fully compatible! We introduce here some techniques to test a Spring Batch job, and chapter 14 is dedicated to testing. We don't show tests systematically in this book; otherwise, half of the book would contain testing code! We truly believe in test-driven development, so we test all the source code with automated tests. Download the source code, browse it, and read chapter 14 to discover more about testing techniques.

The next section is about setting up the test infrastructure: the ACME job needs a database to write to, and Spring Batch itself needs a couple of infrastructure components to launch jobs and maintain execution metadata. Let's see how to configure a lightweight test infrastructure to launch the test from an IDE.

1.6.1 Setting up the test infrastructure

Spring Batch needs infrastructure components configured in a Spring lightweight container. These infrastructure components act as a lightweight runtime environment to run the batch process. Setting up the batch infrastructure is a mandatory step for a batch application, which you need to do only once for all jobs living in the same Spring application context. The jobs will use the same infrastructure components to run and to store their state. These infrastructure components are the key to managing and monitoring jobs (chapter 12 covers how to monitor your Spring Batch jobs).

Spring Batch needs two infrastructure components:

- *Job repository*—To store the state of jobs (finished or currently running)
- *Job launcher*—To create the state of a job before launching it

For this test, you use the volatile job repository implementation. It's perfect for testing and prototyping because it stores execution metadata in memory. Chapter 2 covers how to set up a job repository that uses a database. The following listing shows how to configure the test infrastructure.

Listing 1.7 Spring configuration for the batch infrastructure

```
<?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:jdbc="http://www.springframework.org/schema/jdbc"
       xsi:schemaLocation="http://www.springframework.org/schema/jdbc
       http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <jdbc:embedded-database id="dataSource" type="H2">
    <jdbc:script location="/create-tables.sql"/>
  </jdbc:embedded-database>

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

  <bean id="jobRepository"
        class="org.springframework.batch.core.
➥ repository.support.MapJobRepositoryFactoryBean">
    <property name="transactionManager"
              ref="transactionManager" />
  </bean>

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

  <bean class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg ref="dataSource" />
  </bean>

</beans>
```

Declares and populates data source

Declares transaction manager

Declares job repository

Declares job launcher

This listing uses an open source in-memory database called H2; although it may look odd for an online application, it's easy to deploy and you won't have to install any database engine to work with the code samples in this chapter. And remember, this is the testing configuration; the application can use a full-blown, persistent database in

production. For a more traditional relational database management system (RDBMS) setup, you could change the data source configuration to use a database like PostgreSQL or Oracle. Listing 1.7 also runs a SQL script on the database to create the product table and configures a `JdbcTemplate` to check the state of the database during the test.

> **How does a job refer to the job repository?**
>
> You may have noticed that we say a job needs the job repository to run but we don't make any reference to the job repository bean in the job configuration. The XML `step` element can have its `job-repository` attribute refer to a job repository bean. This attribute isn't mandatory, because by default the job uses a `jobRepository` bean. As long as you declare a `jobRepository` bean of type `JobRepository`, you don't need to explicitly refer to it in your job configuration.

This leads us to the following best practice: when configuring a Spring Batch application, the infrastructure and job configuration should be in separate files.

SPLITTING INFRASTRUCTURE AND APPLICATION CONFIGURATION FILES You should always split infrastructure and application configuration files (test-context.xml and import-products-job-context.xml in our example). This allows you to swap out the infrastructure for different environments (test, development, staging, production) and still reuse the application (job, in our case) configuration files.

In a split application configuration, the infrastructure configuration file defines the job repository and data source beans; the job configuration file defines the job and depends on the job repository and data source beans. For Spring to resolve the whole configuration properly, you must bootstrap the application context from both files.

You completed the infrastructure and job configuration in a flexible manner by splitting the configuration into an infrastructure file and a job file. Next, you make the configuration more flexible by leveraging the Spring Expression Language (SpEL) to avoid hardcoding certain settings in Spring configuration files.

1.6.2 Leveraging SpEL for configuration

Remember that part of your job configuration is hardcoded in the Spring configuration files, such as all file location settings (in bold):

```
<bean id="decompressTasklet"
      class="com.manning.sbia.ch01.batch.DecompressTasklet">
  <property name="inputResource" value="file:./input/input.zip" />
  <property name="targetDirectory" value="./work/output/" />
  <property name="targetFile" value="products.txt" />
</bean>
```

These settings aren't flexible because they can change between environments (testing and production, for example) and because rolling files might be used for the incoming archive (meaning the filename would dependon the date). An improvement is to turn

these settings into parameters specified at launch time. When launching a Spring Batch job, you can provide parameters, as in the following:

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

The good news is that you can refer to these parameters in your job configuration, which comes in handy for the `DecompressTasklet` and `FlatFileItemReader` beans, as shown in the following listing.

Listing 1.8 Referring to job parameters in the Spring configuration

```
<bean id="decompressTasklet"
      class="com.manning.sbia.ch01.batch.DecompressTasklet"
      scope="step">
  <property name="inputResource"
    value="#{jobParameters['inputResource']}" />
  <property name="targetDirectory"
    value="#{jobParameters['targetDirectory']}" />
  <property name="targetFile"
    value="#{jobParameters['targetFile']}" />
</bean>

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

1 Uses step scope

2 Refers to job parameters

1 Uses step scope

To be able to refer to job parameters, a bean must use the Spring Batch `step` scope 1. The step scope means that Spring will create the bean only when the step asks for it and that values will be resolved then (this is the lazy instantiation pattern; the bean isn't 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 SpEL, which is available as of Spring 3.0 (Spring Batch falls back to a less powerful language if you don't have Spring 3.0 on your class path). The `jobParameters` variable behaves like a `Map`. That's how you refer to the `inputResource`, `targetDirectory`, and `targetFile` job parameters 2. Note that you're not limited to plain references; you can also use more complex expressions; for example, notice how the target directory and file are concatenated for the `resource` property.

You're done with the configuration: the job and infrastructure are ready, and part of the configuration can come from job parameters, which are set when you launch the job. It's time to write the test for your batch process.

1.6.3 Writing the test for the job

You use good old JUnit to write the test, with some help from the Spring testing support. The following listing shows the integration test for the job.

Listing 1.9 Integration test for the import product test

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

(...)

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={
  "/import-products-job-context.xml",
  "/test-context.xml"
})
public class ImportProductsIntegrationTest {

  @Autowired
  private JobLauncher jobLauncher;

  @Autowired
  private Job job;

  @Autowired
  private JdbcTemplate jdbcTemplate;

  @Before
  public void setUp() throws Exception {
    jdbcTemplate.update("delete from product");
    jdbcTemplate.update("insert into product "+
      "(id,name,description,price) values(?,?,?,?)",
      "PR....214","Nokia 2610 Phone","",102.23
    );
  }

  @Test
  public void importProducts() throws Exception {
    int initial = jdbcTemplate.queryForInt("select count(1) from product");
    jobLauncher.run(
      job, new JobParametersBuilder()
        .addString("inputResource",
           "classpath:/input/products.zip")
        .addString("targetDirectory",
           "./target/importproductsbatch/")
        .addString("targetFile",
           "products.txt")
        .addLong("timestamp",
           System.currentTimeMillis())
         .toJobParameters()
    );
    int nbOfNewProducts = 7;
    Assert.assertEquals(
      initial+nbOfNewProducts,
      jdbcTemplate.queryForInt(
        "select count(1) from product")
    );
  }
}
```

1 Cleans and populates database

2 Launches job with parameters

3 Checks correct item insertion

The test uses the Spring T estContext Framework, which creates a Spring application context during the test and lets you inject some Spring beans into the test (with the `@Autowired` annotation). The `@RunWith` and `@ContextConfiguration` trigger the

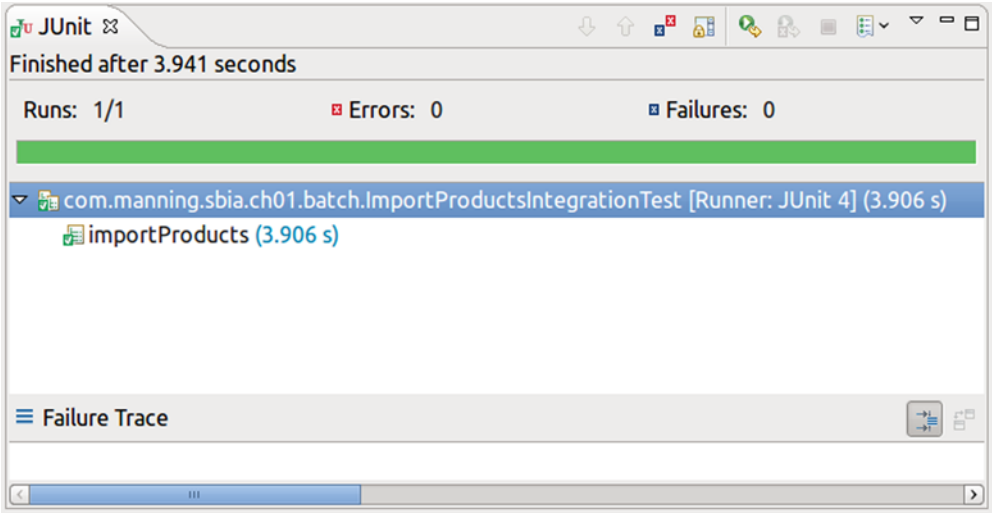

Figure 1.10 Launching the test in Eclipse. Despite all its features, Spring Batch remains lightweight, making jobs easy to test.

Spring TestContext Framework. Chapter 14 is all about testing, so give it a read if you want to learn more about this topic. At ❶, you clean and populate the database. This creates a consistent database environment for each `@Test` method. At ❷, you launch the job with its parameters and check at ❸ that the job correctly inserted the products from the test ZIP archive. The test ZIP archive doesn't have to contain thousands of records: it can be small so the test runs quickly.

You can now run the test with your favorite IDE (Eclipse, IDEA) or build tool (Maven, Ant). Figure 1.10 shows the result of the test execution in Eclipse.

That's it! You have a reliable integration test for your batch job. W asn't it easy? Even if the job handles hundreds of thousands of records daily, you can test in an IDE in a couple of seconds.

> **NOTE** A common requirement is launching jobs from the command line. Chapter 4 covers this topic.

The job works, great, but batch applications aren't common pieces of software: they must be bulletproof. What happens if the input file contains a badly formatted line? Could you live with your job crashing beca use of an extra comma? The next section covers how Spring Batch lets you skip incorrect lines instead of failing.

1.7 Skipping incorrect lines instead of failing

We listed the requirements for batch app lications, including robustness. The import product job isn't robust yet: for example, it crashes abruptly if only a single line of the flat file is formatted incorrectly. The good news is that Spring Batch can help make the job more robust by changing the configuration or by implementing simple interfaces.

Spring Batch's features related to robustness are thoroughly covered in chapter 8. For now, we show you how to handle unexpected entries when you're reading data. By the end of this section, the import product job will be more robust and you'll have a better understanding of how Spring Batch can help improve robustness in general.

On a good day the import product job will decompress the input archive, read each line of the extracted flat file, send data to the database, and then exit successfully. As you know, if something can go wrong, it will. For instance, if the `FlatFileItemReader` fails to read a single line of the flat file—because it's incorrectly formatted, for example—the job immediately stops. Perhaps this is acceptable behavior, but what if you can live with some invalid records? In this case, you could skip an invalid line and keep on chugging. Spring Batch allows you to choose declaratively a skip policy when something goes wrong. Let's apply a skip policy to your job's import step.

Suppose a line of the flat file hasn't been generated correctly, like the price (in bold) of the third product in the following snippet:

```
PRODUCT_ID,NAME,DESCRIPTION,PRICE
PR....210,BlackBerry 8100 Pearl,,124.60
PR....211,Sony Ericsson W810i,,139.45
PR....212,Samsung MM-A900M Ace,,97,80
PR....213,Toshiba M285-E 14,,166.20
```

The format of the price field of the third record is incorrect: it uses a comma instead of a period as the decimal separator. Note that the comma is the field separator Spring Batch uses to tokenize input lines: the framework would see five fields where it expects only four. The `FlatFileItemReader` throws a `FlatFileParseException` and, in the default configuration, Spring Batch immediately stops the process.

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

Listing 1.10 Setting the skip policy when reading records from the flat file

```
<job id="importProducts"
     xmlns="http://www.springframework.org/schema/batch">
  <step id="decompress" next="readWriteProducts">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWriteProducts">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100"
          skip-limit="5">
        <skippable-exception-classes>
          <include class="org.springframework.batch.
          ➥ item.file.FlatFileParseException" />
        </skippable-exception-classes>
      </chunk>
    </tasklet>
  </step>
</job>
```

1 Fails job if Spring Batch skips five records

2 Skips flat file parse exceptions

The skip policy is set in the `chunk` element. The `skip-limit` attribute (1) is set to tell Spring Batch to stop the job when the number of skipped records in the step exceeds this limit. Your application can be tolerant, but not too tolerant! Then, the exception classes that trigger a skip are stated (2). Chapter 8 details all the options of the `skippable-exception-classes` element. Here, we want to skip the offending line when the item reader throws a `FlatFileParseException`.

You can now launch the job with an input file containing incorrectly formatted lines, and you'll see that Spring Batch keeps on running the job as long as the number of skipped items doesn't exceed the skip limit. Assuming the ZIP archive contains incorrect lines, you can add a test method to your test, as shown in the following listing.

Listing 1.11 Testing the job correctly skips incorrect lines with a new test method

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/import-products-job-context.xml","/test-
     context.xml"})
public class ImportProductsIntegrationTest {
  (...)
  @Test
  public void importProductsWithErrors() throws Exception {
    int initial = jdbcTemplate.queryForInt("select count(1) from product");

    jobLauncher.run(job, new JobParametersBuilder()
      .addString("inputResource",
        "classpath:/input/products_with_errors.zip")
      .addString("targetDirectory", "./target/importproductsbatch/")
      .addString("targetFile","products.txt")
      .addLong("timestamp", System.currentTimeMillis())
      .toJobParameters()
    );
    int nbOfNewProducts = 6;
    Assert.assertEquals(
      initial+nbOfNewProducts,
      jdbcTemplate.queryForInt("select count(1) from product")
    );
  }
}
```

Note that this code doesn't do any processing when something goes wrong, but you could choose to log that a line was incorrectly formatted. Spring Batch also provides hooks to handle errors (see chapter 8).

This completes the bullet-proofing of the product import job. The job executes quickly and efficiently, and it's more robust and reacts accordingly to unexpected events such as invalid input records.

That's it—you've implemented a full-blown job with Spring Batch! This shows how Spring Batch provides a powerful framework to create batch jobs and handles the heavy lifting like file I/O. Your main tasks were to write a couple of Java classes and do some XML configuration. That's the philosophy: focus on the business logic and Spring Batch handles the rest.

1.8 Summary

This chapter started with an overview of batch applications: handling large amounts of data automatically. The immediate requirements are performance, reliability, and robustness. Meeting these requirements is tricky, and this is where Spring Batch comes in, by processing data efficiently thanks to its chunk-oriented approach and providing ready-to-use components for the most popular technologies and formats.

After the introduction of batch applications and Spring Batch, the import product job gave you a good overview of what the framework can do. You discovered Spring Batch's sweet spot—chunk processing—by reading records from a flat file and writing them to a database. You used built-in components like the `FlatFileItemReader` class and implemented a database `ItemWriter` for business domain code. Remember that Spring Batch handles the plumbing and lets you focus on the business code. You also saw that Spring Batch isn't limited to chunk processing when you implemented a `Tasklet` to decompress a ZIP file.

Configuring the job ended up being quite simple thanks to the Spring lightweight container and Spring Batch XML. You even saw how to write an automated integration test. Finally, you learned how to make the job more robust by dealing with invalid data.

You're now ready to implement your first Spring Batch job. With this chapter under your belt, you can jump to any other chapter in this book when you need information to use a specific Spring Batch feature: reading and writing components, writing bulletproof jobs with skip and restart, or making your jobs scale. You can also continue on to chapter 2, where we present the Spring Batch vocabulary and the benefits it brings to your applications.

JAVA/SPRING

Spring Batch IN ACTION

Cogoluègnes • Templier • Gregory • Bazoud

Even though running batch jobs is a common task, there's no standard way to write them. Spring Batch is a framework for writing batch applications in Java. It includes reusable components and a solid runtime environment, so you don't have to start a new project from scratch. And it uses Spring's familiar programming model to simplify configuration and implementation, so it'll be comfortably familiar to most Java developers.

Spring Batch in Action is a thorough, in-depth guide to writing efficient batch applications. Starting with the basics, it discusses the best practices of batch jobs along with details of the Spring Batch framework. You'll learn by working through dozens of practical, reusable examples in key areas like monitoring, tuning, enterprise integration, and automated testing.

What's Inside

- Batch programming from the ground up
- Implementing data components
- Handling errors during batch processing
- Automating tedious tasks

No prior batch programming experience is required. Basic knowledge of Java and Spring is assumed.

Arnaud Cogoluègnes, Thierry Templier, and **Olivier Bazoud** are Java EE architects with a focus on Spring. **Gary Gregory** is a Java developer and software integration specialist.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/SpringBatchinAction

Free ebook
SEE INSERT

"Clear, easy to read, and very thorough."
—Rick Wagner, Red Hat

"A must-have for enterprise batch programmers."
—John Guthrie, SAP

"A fresh look at using batch in the enterprise."
—Tray Scates
Unisys Corporation

"The definitive source."
—Cédric Exbrayat
Lyon Java User Group

"Flawlessly written, easily readable, powerfully presented."
—Willhelm Lehman
Websense Inc.

ISBN 13: 978-1-935182-95-5
ISBN 10: 1-935182-95-1
55999
9 781935 182955

MANNING $59.99 / Can $62.99 [INCLUDING eBOOK]