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 5

brief contents

5

Reading data

This chapter covers

- Reading input data
- Reading from files
- Reading from databases
- Reading from Java Message Service and other sources
- Implementing custom readers

In the previous two chapters, we concentrated on configuring and launching batch jobs. It's now time to dig into the features at the heart of batch processes. As described in chapter 2, Spring Batch provides types for batch processes based on the concepts of *job* and *step*. A job uses a tasklet to implement chunk processing. Chunk-oriented processing allows jobs to implement efficiently the most common batch processing tasks: reading, processing, and writing.

We focus here on the first step of this process, reading. We describe general concepts and types implemented by Spring Batch. These built-in types are the foundation used to support the most common use cases. Spring Batch can use different data sources as input to batch processes. Data sources correspond to flat files, XML, and JavaScript Serialized Object Notation (JSON). Spring Batch also supports other types of data sources, such as Java Message Service (JMS), in the message-oriented middleware world.

In some cases, the Spring Batch built-in implementations aren't enough, and it's necessary to create custom implementations. Because Spring Batch is open source, implementing and extending core types for reading is easily achievable.

Another thing to keep in mind is that reading data is part of the general processing performed by the chunk tasklet. Spring Batch guarantees robustness when executing such processing. That's why built-in implementations implicitly provide integration with the execution context to store current state. The stored data is particularly useful to handle errors and restart batch processes. We concentrate here on the data-reading capabilities of Spring Batch and leave chapter 8 to cover in detail these other aspects.

We use our case study to describe concrete use cases taken from the real world. We explain how to import product data from different kinds of input, with different formats, and how to create data objects.

5.1 Data reading concepts

In this section, we introduce key concepts and types related to reading data in Spring Batch. These concepts are the foundation for the Spring Batch reading feature. This feature and its related types operate within the chunk tasklet, as illustrated in figure 5.1.

This chapter focuses on the first part of chunk processing. At this level, the first key type is the `ItemReader` interface that provides a contract for reading data. This interface supports generics and contains a `read` method that returns the next element read:

```
public interface ItemReader<T> {
  T read() throws Exception, UnexpectedInputException,
                  ParseException, NonTransientResourceException;
}
```

If you've toured the Spring Batch documentation, you've noticed that readers implement another key interface: `ItemStream`. The `ItemStream` interface is important because it allows interaction with the execution context of the batch process to store and restore state. It's also useful when errors occur. In this chapter, we concentrate on the `ItemReader`, and we discuss state management in chapter 8. At this point, it's only necessary to know that readers can save state to properly handle errors and restart.

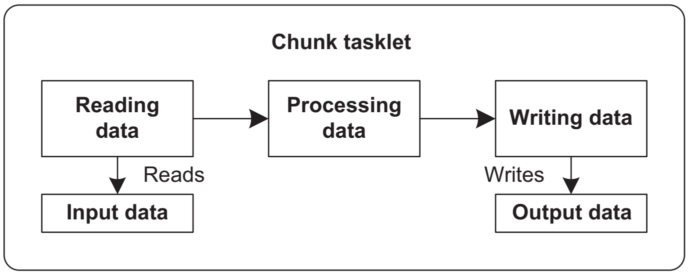

Figure 5.1 A chunk tasklet reads, processes, and writes data.

The following snippet describes the content of the `ItemStream` interface. The `open` and `close` methods open and close the stream. The `update` method allows updating the state of the batch process:

```
public interface ItemStream {
  void open(ExecutionContext executionContext)
                  throws ItemStreamException;
  void update(ExecutionContext executionContext)
                  throws ItemStreamException;
  void close() throws ItemStreamException;
}
```

You can create your own implementations of the `ItemReader` and `ItemStream` interfaces, but Spring Batch provides implementations for common data sources to batch processes. Throughout this chapter, we use our case study as the background story and describe how to import product data into the online store using different data sources.

5.2 Reading flat files

The first data source we describe to input data in batch processes is files. A *file* contains a set of data to integrate into an information system. Each type of file has its own syntax and data structure. Each structure in the file identifies a different data element. To configure a file type in Spring Batch, we must define its format.

Figure 5.2 illustrates the batch process inputs in our case study. Look at each box in figure 5.2 and see how Spring Batch handles that format.

Flat files are pure data files and contain little or no metadata information. Some flat file formats, such as comma-separate value (CSV), may contain one header line as the first line that names columns. In general, though, the file provider defines the file format. This information can consist of field lengths or correspond to a separator splitting data fields. Configuring Spring Batch to handle flat files corresponds to defining the file format to map file records to data objects.

The item reader for flat files is responsible for identifying records in the file and then creating data objects from these records, as shown in figure 5.3.

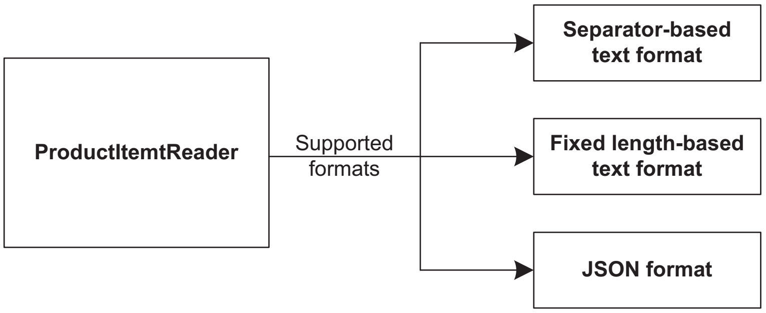

Figure 5.2 The supported file formats in the case study are separator-based text, fixed length-based text, and JSON.

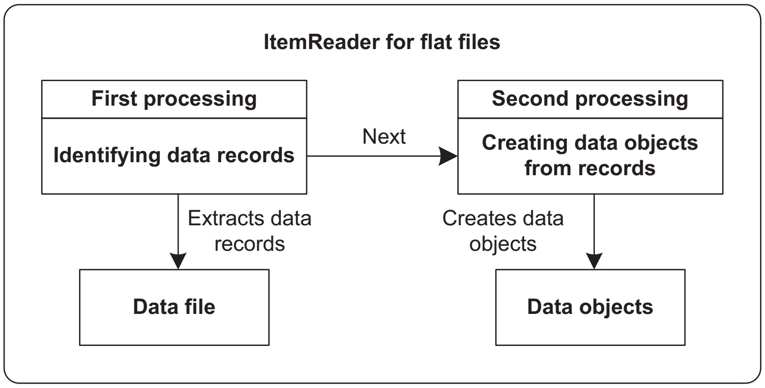

Figure 5.3 `ItemReader` **processing for flat files. The item reader first identifies records, and then creates data objects.**

The `ItemReader` implementation for flat files is the `FlatFileItemReader` class. Several other types work in conjunction with the `FlatFileItemReader` to identify data fields from file lines and to create data objects, as pictured in figure 5.4.

Three inter faces work closely with the `FlatFileItemReader` class. The `RecordSeparatorPolicy` interface identifies data records in a file. The `LineMapper` interface is responsible for extracting data from lines. The `LineCallbackHandler` inter face handles data lines in special cases.

The `DefaultLineMapper` class is the default and most commonly used implementation of the`LineMapper` interface. Two additional interfaces related to the`DefaultLineMapper` class come into play . The `DefaultLineMapper` class holds a `LineTokenizer` responsible for splitting data lines into tokens and a `FieldSetMapper` to create data objects from tokens.

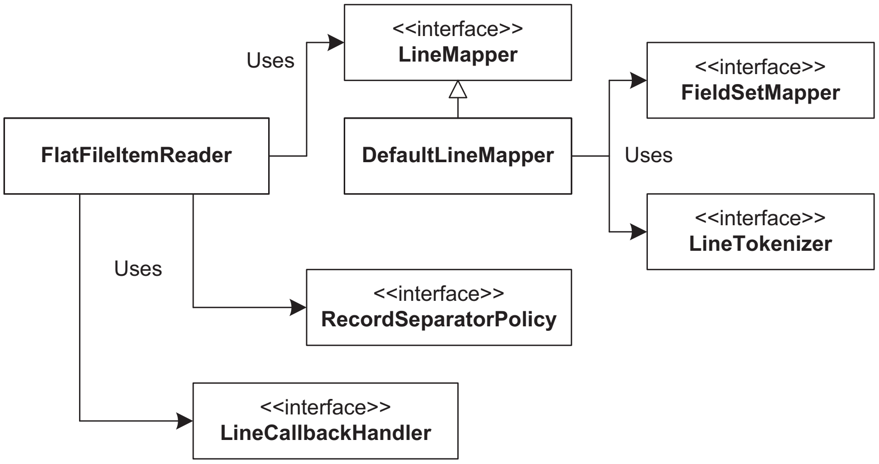
<<interface>> LineMapper
Uses
<<interface>> FieldSetMapper
FlatFileItemReader
DefaultLineMapper
Uses
<<interface>> LineTokenizer
Uses
<<interface>> RecordSeparatorPolicy
<<interface>> LineCallbackHandler

Figure 5.4 **Classes and interfaces involved in reading and parsing flat files**

Table 5.1 summarizes the interfaces from figure 5.4.

Table 5.1 Interfaces for flat file processing with the `FlatFileItemReader` class

Entity	Description
`LineMapper`	Maps a data line to a data object
`FieldSetMapper`	Creates a data object from tokens; invoked by the `DefaultLine-Mapper` class, the default implementation of the `LineMapper` interface
`LineTokenizer`	Splits a data line into tokens; invoked by the `DefaultLineMapper` class, the default implementation of the `LineMapper` interface
`RecordSeparatorPolicy`	Identifies beginning and end of data records
`LineCallbackHandler`	Provides data lines in special cases; the common usage is for lines skipped by the `FlatFileItemReader` class during processing

These interfaces are all involved when configuring the `FlatFileItemReader` class for a file format. You'll use these inter faces and their corresponding implementations to handle various file formats in this chapter.

This section on flat files introduced all concepts and types related to the item reader for flat files to import data files as objects. In the next section, we describe the general configuration of a `FlatFileItemReader` bean in Spring Batch as well as implementations for record-separator policies and line mappers. W e then explain how to handle delimited, fixed-length, and JSON file formats and describe advanced concepts to support records split over several lines and heterogonous records.

5.2.1 Configuring the FlatFileItemReader class

The `FlatFileItemReader` class is configured as a Spring bean and accepts the properties described in table 5.2.

Table 5.2 `FlatFileItemReader` properties

Property	Type	Description
`bufferedReaderFactory`	`BufferedReaderFactory`	Creates `BufferReader` instances for the input file. The default factory (`DefaultBufferedReader-Factory`) provides a suitable instance for text files. Specifying another factory is useful for binary files.
`comments`	`String[]`	Specifies comment prefixes in the input file. When a line begins with one of these prefixes, Spring Batch ignores that line.

Table 5.2 `FlatFileItemReader` properties *(continued)*

Property	Type	Description
`encoding`	`String`	The input file's encoding. The default value is the class's `DEFAULT_CHARSET` constant.
`lineMapper`	`LineMapper<T>`	Creates objects from file data records.
`linesToSkip`	`int`	The number of lines to skip at the beginning of the file. This feature is particularly useful to handle file headers. If the `skippedLinesCallback` property is present, the item reader provides each line to the callback.
`recordSeparatorPolicy`	`RecordSeparatorPolicy`	How the input file delimits records. The provided class can detect single or multiline records.
`resource`	`Resource`	The input resource. You can use standard Spring facilities to locate the resource.
`skippedLinesCallback`	`LineCallbackHandler`	The callback for lines skipped in the input file. Used jointly with the `linesToSkip` property.
`strict`	`boolean`	Whether item reader throws an exception if the resource doesn't exist. The default value is `true`.

The following listing describes how to configure an instance of the `FlatFileItemReader` class in Spring Batch using the properties `linesToSkip`, `recordSeparatorPolicy`, and `lineMapper`. You use this type of configuration for the online store use case with flat files.

Listing 5.1 Configuring a `FlatFileItemReader`

```
<bean id="productItemReader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="resource" value="datafile.txt"/>
  <property name="linesToSkip" value="1"/>
  <property name="recordSeparatorPolicy"
            ref="productRecordSeparatorPolicy"/>
  <property name="lineMapper" ref="productLineMapper"/>
</bean>

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

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

Specifying the value 1 for the linesToSkip property means that the reader doesn't consider the first line as a data line and that the line will be skipped. In the context of the use case, this line corresponds to the file header describing the record fields. The recordSeparatorPolicy property determines how to delimit product records in the file. Finally, the code specifies how to create a product object from a data record using the lineMapper property. To lighten the listing, we elided the beans corresponding to the two last entities but we detail them next.

The first type, the RecordSeparatorPolicy interface, delimits data records with the following methods:

```
public interface RecordSeparatorPolicy {
  boolean isEndOfRecord(String line);
  String postProcess(String record);
  String preProcess(String record);
}
```

The RecordSeparatorPolicy interface detects the end of a record and can preprocess and postprocess lines. Implementations can support continuation markers and unbalanced quotes at line ends. The FlatFileItemReader class uses a RecordSeparatorPolicy to build data records when parsing the data file. Spring Batch provides several implementations of this interface, described in table 5.3.

Table 5.3 RecordSeparatorPolicy built-in implementations

Implementation	Description
SimpleRecordSeparatorPolicy	Separates input as one record per line; the simplest implementation and root class for all other implementations.
DefaultRecordSeparatorPolicy	Supports unbalanced quotes at line end and a continuation string.
JsonRecordSeparatorPolicy	Uses JSON as the record format and can detect JSON objects over multiple lines, based on numbers of tokens delimited by characters { and }.
SuffixRecordSeparatorPolicy	Expects a specific string at line end to mark the end of a record. By default, this string is a semicolon.

Configuring record separation policy classes can be simple because their default constructors cover the most common cases. The following XML fragment is an example:

```
<bean id="productRecordSeparatorPolicy"
      class="org.springframework.batch.item.file
                              ➥ .separator.DefaultRecordSeparatorPolicy">
```

Another important interface present as a property of the FlatFileItemReader class is the LineMapper interface. The LineMapper interface provides data objects from

JavaScript Object Notation (JSON)

JSON is an open and lightweight text-based standard designed for human-readable data interchange. It provides a way to structure text data using braces and brackets. This technology is similar to XML but requires about 30% fewer characters.

The JSON format is commonly associated with JavaScript because the language uses it to perform I/O for data structures and objects. The JSON format is language independent, but you usually see it used in Asynchronous JavaScript + XML (AJAX)-styled web applications.

record lines without knowing how Spri ng Batch obtained the lines. The `LineMapper` interface contains one method called `mapLine`:

```
public interface LineMapper<T> {
  T mapLine(String line, int lineNumber) throws Exception;
}
```

Spring Batch provides several implementations of this interface for different use cases and file formats, as described in table 5.4.

Table 5.4 **`LineMapper` built-in implementations**

Class	Description
`DefaultLineMapper`	The default implementation tokenizes lines and maps items to objects.
`JsonLineMapper`	Supports the JSON format for records and extracts data to a map for each record. This implementation is based on the jackson-mapper-asl.jar file that can be reached at the website http://jackson.codehaus.org/.
`PassThroughLineMapper`	Provides the original record string instead of a mapped object.
`PatternMatchingCompositeLineMapper`	Parses heterogeneous record lines. For each line type, a line tokenizer and a field-set mapper must be configured.

We describe in detail the `DefaultLineMapper`, `JsonLineMapper`, and `PatternMatchingCompositeLineMapper` implementations in the next sections. First, we take a quick look at the `PassThroughLineMapper` class.

The `PassThroughLineMapper` class performs no parsing or data extraction. It's simple to configure because it doesn't define properties. The configuration of this class is shown in the following XML fragment:

```
<bean id="lineMapper" class="org.springframework.batch.item
                                                ➥ .file.mapping.PassThroughLineMapper"/>
```

The DefaultLineMapper class is the most commonly used implementation because it handles files with implicit structures using separators or fixed-length fields. In our use case, we accept several data formats for incoming data. W e describe next how to configure Spring Batch to handle data structures based on separators and fixed-length fields.

5.2.2 Introducing the DefaultLineMapper class

The most commonly used implementation of the LineMapper inter face is the DefaultLineMapper class. It implements line processing in two phases:

- Parses a line to extract fields using an implementation of the LineTokenizer interface
- Creates data objects from fields using an implementation of the FieldSet-Mapper interface

Figure 5.5 illustrates how theLineTokenizer and FieldSetMapper interfaces described in the preceding list interact within the DefaultLineMapper.

The lineTokenizer and fieldSetMapper properties configure the DefaultLine-Mapper class's LineTokenizer and FieldSetMapper. For example:

```
<bean id="productLineMapper"
    class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
  <property name="lineTokenizer" ref="productLineTokenizer"/>
  <property name="fieldSetMapper" ref="productFieldSetMapper"/>
</bean>

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

This example uses bean references (using the ref attribute), but you could also use an inner bean. The lineTokenizer property is set to an instance of LineTokenizer. The fieldSetMapper property is set to an instance of FieldSetMapper.

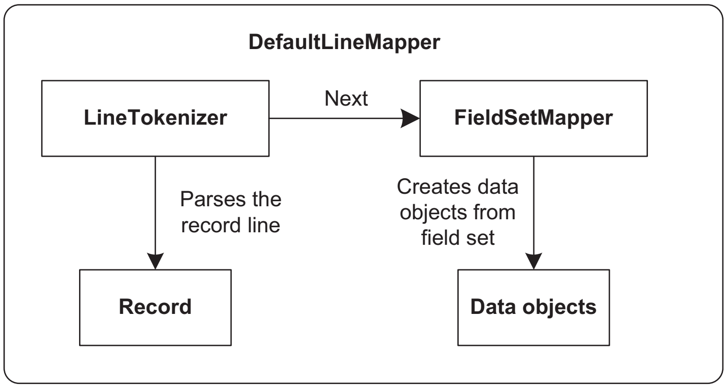

Figure 5.5 Interactions between LineTokenizer and FieldSetMapper in a DefaultLineMapper. The LineTokenizer parses record lines, and the FieldSetMapper creates objects from field sets.

5.2.3 Using the DefaultLineMapper class

It's now time to use DefaultLineMapper in our case study. The FieldSetMapper implementation remains the same; it creates objects from a FieldSet. The LineTokenizer implementation depends on the record format and provides the contract to create a field set from lines in a data file, as defined here:

```
public interface LineTokenizer {
  FieldSet tokenize(String line);
}
```

A FieldSet is returned by the tokenize method, which parses the record. A FieldSet contains all extracted record fields an d offers several methods to manage them. Table 5.5 lists the Spring Batch implementations of the LineTokenizer interface for different ways to delimit fields in data lines.

Table 5.5 **LineTokenizer built-in implementations**

Implementation	Description
DelimitedLineTokenizer	Uses a delimiter to split a data line into fields. Defaults correspond to comma-separated values.
FixedLengthTokenizer	Uses field lengths to split a data line into fields.

5.2.4 Extracting character-separated fields

Let's take concrete examples from our case study. The first example contains product records that use the comma character to separate fields, as shown in the following example:

```
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
PR....214,Nokia 2610 Phone,,145.50
PR....215,CN Clogs Beach/Garden Clog,,190.70
PR....216,AT&T 8525 PDA,,289.20
```

For this example, configuring a LineTokenizer consists of defining a DelimitedLineTokenizer with the names and delimiter properties. The names property defines the names of the fields, and the delimiter property defines the field delimiter; for example:

```
<bean id=" productLineTokenizer"
      class="org.springframework.batch.item.file
                    ➥ .transform.DelimitedLineTokenizer">
  <property name="delimiter" value=","/>
  <property name="names"
            value="id,name,description,price"/>
</bean>
```

The delimiter property specifies the character used to separate fields in a data file—in this case, the comma: ",". The names property defines the field names as an ordered list separated by commas: "id,name,description,price". When the tokenizer extracts a field from the data line, the corresponding name is associated with it so it's possible to get the field value by name.

EXTRACTING FIXED-LENGTH FIELDS

Another supported data structure in the case study is fixed-length fields. Note that such a data structure is potentially larger because all fields must have the same length. Here's an example of this structure:

```
PR....210BlackBerry 8100 Pearl                    124.60
PR....211Sony Ericsson W810i                      139.45
PR....212Samsung MM-A900M Ace                      97.80
PR....213Toshiba M285-E 14                        166.20
PR....214Nokia 2610 Phone                         145.50
PR....215CN Clogs Beach/Garden Clog               190.70
PR....216AT&T 8525 PDA                            289.20
```

For this example, table 5.6 describes the lengths of each field. This format doesn't use a separator character.

Field name	Length in characters
id	9
name	26
description	15
price	6

Table 5.6 Fixed record field names and lengths

To configure the LineTokenizer for this type of example, you define a FixedLengthTokenizer with names and columns properties. The properties respectively define the names of fields and the column ranges used to identify them:

```
<bean id=" productLineTokenizer"
      class="org.springframework.batch.item.file
                    ➥ .transform.FixedLengthTokenizer">
  <property name="columns" value="1-9,10-35,36-50,51-56"/>
  <property name="names"
            value="id,name,description,price"/>
</bean>
```

The columns property configures column ranges for each field. This property accepts a list of Range instances, but you can also configure it using a comma-separated string of ranges. You can see that this property numbers from 1 and not 0 because Spring' s RangeArrayPropertyEditor class is internally used to configure the specified value into the value to set. The names property sets field names, as with the DelimitedLineTokenizer class.

Spring built-in PropertyEditor support

Spring provides support to configure properties as strings. The PropertyEditor interface describes how to convert strings to beans, and vice versa. For nonstring type properties, Spring automatically tries to convert to the appropriate type using the registered property editors. They come from the JavaBean specification and can be seen as a to/from string conversion service.

This support is extensible: you can register your own property editors to convert strings to custom objects with the CustomEditorConfigurer class. Spring Batch uses this mechanism to register its own property editors to make some types easier to configure. For example, the Range class is configured with the RangeArrayPropertyEditor class.

5.2.5 Creating objects from fields

Now that you've configured extracting fields from data lines, it's time to specify how to create data objects from these fields. This process isn't specific to a particular LineMapper but relates to the field set structure.

The FieldSetMapper interface defines this process and uses generics to type implementations to application-specific types. Spring Batch defines the FieldSetMapper interface as

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

The mapFieldSet method implements the mapping where a LineTokenizer implementation has created the FieldSet instance. Table 5.7 lists the Spring Batch implementations of this interface for different ways to handle fields and create data objects.

Table 5.7 **FieldSetMapper built-in implementations**

Class	Description
BeanWrapperFieldSetMapper	Uses field names to set data in properties of data beans.
PassThroughFieldSetMapper	Provides the FieldSet without doing any mappings to objects. Useful if you need to work directly with the field set.

Before using a FieldSetMapper, you must implement the bean to receive the data. In the case study, as you import product data, the bean is a plain old Java object (POJO) that contains the id, name, description, and price properties, as shown in the following listing.

Listing 5.2 The Product bean

```
public class Product {
  private String id;
  private String name;
  private String description;
  price float price;

  public String getId() { return id; }
     public void setId(String id) { this.id = id; }
  (...)
  public float getPrice() { return price; }
  public void setPrice(float price) { this. price = price; }
}
```

To map the FieldSet instance to the Product data bean, you implement a FieldSet-Mapper to populate Product instances from a FieldSet instance. The following listing shows the FieldSetMapper implementation that creates Product instances.

Listing 5.3 Custom FieldSetMapper for creating Product objects

```
public class ProductFieldSetMapper implements FieldSetMapper<Product> {
  public Product mapFieldSet(FieldSet fieldSet) {
    Product product = new Product();
    product.setId(fieldSet.readString("id"));
    product.setName(fieldSet.readString("name"));
    product.setDescription(fieldSet.readString("description"));
    product.setPrice(fieldSet.readFloat("price"));
    return product;
  }
}
```

In the mapFieldSet method implementation, an uninitialized instance of a Product is first created. The mapFieldSet method then uses the FieldSet instance and its various read methods. The FieldSet class provides multiple read methods, one for each primitive Java type, plus String, Date, and BigDecimal. Each read method takes a field name or field index as a parameter; some methods also provide an argument for a default value.

You configure ProductFieldSetMapper as follows:

```
<bean id="productFieldSetMapper"
      class="com.manning.sbia.reading.ProductFieldSetMapper"/>
```

As described in table 5.7, Spring Batch provides a bean-based implementation of the FieldSetMapper interface: the BeanWrapperFieldSetMapper class. This class makes working with field sets easier because you don't have to write a custom FieldSetMapper.

You specify a data template for the bean using the prototypeBeanName property, where the value is the bean name for this template. You must configure the corresponding bean with the prototype scope. When a Product is instantiated, its properties are set using field set data. The bean property names and field names must match exactly for the mapping to take place. The following XML fragment shows this configuration:

```
<bean id="productFieldSetMapper"
      class="org.springframework.batch.item.file
                                ➥ .mapping.BeanWrapperFieldSetMapper">
  <property name="prototypeBeanName" value="product"/>
</bean>
```

```
<bean id="product"
      class="com.manning.sbia.reading.Product"
      scope="prototype"/>
```

After defining a bean of type `BeanWrapperFieldSetMapper`, you set its `prototypeBeanName` property to the identifier of the bean used to create data instances. For our case study, the bean type is `Product`.

Another interesting data format that Spring Batch supports is JSON. We introduced this format in tables 5.3 and 5.4 with dedicated Spring Batch types. In the next section, we describe how to implement and configure processing to support JSON-formatted data in our case study.

5.2.6 Reading JSON

Spring Batch provides support for JSON with a `LineMapper` implementation called `JsonLineMapper`. The following listing shows the JSON content of a data file corresponding to the data presented in the previous section. This is the last format for flat files in our case study.

Listing 5.4 Product data file using JSON

```
{ "id": "PR....210",
  "name": "BlackBerry 8100 Pearl",
  "description": "",
  "price": 124.60 }
{ "id": "PR....211",
  "name": "Sony Ericsson W810i",
  "description": "",
  "price": 139.45 }
{ "id": "PR....212",
  "name": "Samsung MM-A900M Ace",
  "description": "",
  "price": 97.80 }
(...)
```

Configuring the `JsonLineMapper` class is simple because line parsing is built into the class, and each `FieldSet` maps to a `java.util.Map`. No additional types are required to configure the class, as shown in the following XML fragment:

```
<bean id="productsLineMapper"
      class="org.springframework.batch.item.file.mapping.JsonLineMapper"/>
```

Using a `JsonLineMapper`, you get a list of `Map` instances containing all data from the JSON structure. If you were to convert this processing to code, you'd have a listing similar to the following.

Listing 5.5 JSON data processing as Java code

```
List<Map<String,Object>> products = new ArrayList<Map<String,Object>>();

Map<String,Object> product210 = new HashMap<String,Object>();
product210.put("id", "PR....210");
product210.put("name", "BlackBerry 8100 Pearl");
```

```
product210.put("description", "");
product210.put("price", 124.60);
products.add(product210);

Map<String,Object> product211 = new HashMap<String,Object>();
product211.put("id", "PR....211");
product211.put("name", "Sony Ericsson W810i");
product211.put("description", "");
product211.put("price", 139.45);
products.add(product211);

Map<String,Object> product212 = new HashMap<String,Object>();
product212.put("id", "PR....212");
product212.put("name", "Samsung MM-A900M Ace");
product212.put("description", "");
product212.put("price", 97.80);
products.add(product212);
```

Using the JsonLineMapper class is convenient to get data as Map objects, but it's perhaps not exactly what you need. At this point in the case study, you want to support several input data formats homogenously. For every type of format, data must come through as Product instances.

For this reason, your work isn't finished. You need to create an additional class implementing the LineMapper interface to wrap a JsonLineMapper. The purpose of this class, called WrappedJsonLineMapper, is to delegate processing to the target JsonLineMapper instance and then to create Product instances from Map objects. The following listing shows the JsonLineMapperWrapper class.

Listing 5.6 A JsonLineMapper wrapper to create data objects

```
public class JsonLineMapperWrapper implements LineMapper<Product> {
  private JsonLineMapper delegate;

  public Product mapLine(String line, int lineNumber) {
    Map<String,Object> productAsMap                        (1) Delegates to target
                   = delegate.mapLine(line, lineNumber);        JsonLineMapper

    Product product = new Product();
    product.setId((String)productAsMap.get("id"));
    product.setName(                                         (2) Populates
              (String)productAsMap.get("name"));                  product
    product.setDescription(                                       from map
              (String)productAsMap.get("description"));
    product.setPrice(
              new Float((Double)productAsMap.get("price")));

    return product;
  }
}
```

The WrappedJsonLineMapper class is a wrapper for a target JsonLineMapper instance defined with the delegate property. This makes it possible to delegate processing (1) within the mapLine method and to get the corresponding result as a Map. The mapLine method then converts the Map to a Product object (2).

We've described all the supported formats for flat files in Spring Batch. Before moving on to XML data input support in Spring Batch, we explain how to handle records spread over multiple lines and how to support several record types within the same data file.

5.2.7 Multiline records

The `RecordSeparatorPolicy` interface identifies record boundaries using its `isEndOfRecord` method. For files using only one line per record, unbalanced quotes, or continuation markers exceptions, you can use the default implementation of this interface, the `DefaultRecordSeparatorPolicy` class.

When an input source spreads records over several lines, a custom implementation is required to specify the conditions that delimit records. Imagine that each product record extends over two lines. The first line provides general data such as product identifier and name, and the second line includes additional information like the description and price. Here's an example of this format:

```
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
(...)
```

In this case, the implementation of the `isEndOfRecord` method needs to detect if the line starts with a product identifier. If true, this *isn't* the end of the record. The following listing implements this format and assumes that no unbalanced quotes and continuation markers are present.

Listing 5.7 Reading multiline records with a custom `RecordSeparatorPolicy`

```
public class TwoLineProductRecordSeparatorPolicy
                          implements RecordSeparatorPolicy {

  public String postProcess(String record) {
    return record;
  }

  public String preProcess(String line) {
    return line;
  }

  private int getCommaCount(String s) {
    String tmp = s;
    int index = -1;
    int count = 0;
    while ((index=tmp.indexOf(","))!=-1) {
      tmp = tmp.substring(index+1);
      count++;
    }
    return count;
  }

  public boolean isEndOfRecord(String line) {
    return getCommaCount(line)==3;          <-- (1) Checks comma count
  }
}
```

To determine if the current line is the end of the product record, you check if the string contains three commas ❶ because a valid product must have four properties separated by commas (if a valid product required five properties, getCommaCount would be set to check for four commas). The remaining task is to set the implementation on the FlatFileItemReader bean using its recordSeparatorPolicy property, as described in listing 5.2.

To close the topic of reading from flat files, the following section describes heterogonous record handling within the same file.

5.2.8 Reading heterogonous records

Records present in flat files may not always be uniform. Each record still corresponds to one line, including support for unbalanced quotes and a continuation character, but can correspond to different data records. In our case study, this corresponds to having several product types with different data in the same file. The following file example contains mobile phone records as before and new book records:

```
PRM....210,BlackBerry 8100 Pearl,,BlackBerry,124.60
PRM....211,Sony Ericsson W810i,,Sony Ericson,139.45
PRB....734,Spring Batch in action,,Manning,34.95
PRM....212,Samsung MM-A900M Ace,,Samsung,97.80
PRB....735,Spring Roo in action,,Manning,34.95
PRM....213,Toshiba M285-E 14,,Toshiba,166.20
PRB....736,Spring in action,,Manning,44.95
PRM....214,Nokia 2610 Phone,,Nokia,145.50
```

In this data file example, lines beginning with PRM correspond to mobile phones (product-mobile), and lines beginning with PRB to books (product-book). In this case, you use polymorphism to create a basic product class and subclasses for specific types of products, mobile phones and books, as illustrated in figure 5.6.

Because the data file mixes different types of products, you must define rules to detect the product type for a given line. The prefix of the product identifier is used here: an identifier beginning with PRM is a mobile phone, and one with PRB is a book. To associate a line mapper for each line type, you use a LineMapper implementation called PatternMatchingCompositeLineMapper.

The PatternMatchingCompositeLineMapper class detects different records, parses them, and extracts data objects. The following listing describes how to configure the class as a bean to handle a multiproduct data file.

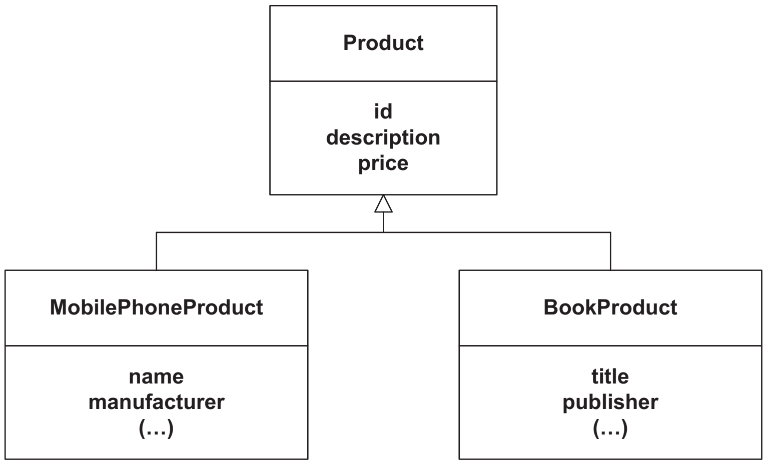

Figure 5.6 Inheritance relationships between different kinds of products. The `MobilePhoneProduct` and `BookProduct` classes inherit from the `Product` class.

Listing 5.8 Configuring a composite `LineMapper`

```
<bean id="productLineMapper"
      class="org.springframework.batch.item.file
                          ➥ .mapping.PatternMatchingCompositeLineMapper">
  <property name="tokenizers">                          ← Sets line tokenizers (1)
    <map>
      <entry key="PRM*" value-ref="mobileProductLineTokenizer"/>
      <entry key="PRB*" value-ref="bookProductLineTokenizer"/>
    </map>
  </property>
  <property name="fieldSetMappers">                     ← Sets field set mappers (2)
    <map>
      <entry key="PRM*" value-ref="mobileProductFieldSetMapper"/>
      <entry key="PRB*" value-ref="bookProductFieldSetMapper"/>
    </map>
  </property>
</bean>

<bean id="mobileProductLineTokenizer" class="(...)"> (...) </bean>
<bean id="mobileProductFieldSetMapper" class="(...)"> (...) </bean>

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

The first property, `tokenizers` (1), registers all `LineTokenizers` in a map. The map keys contain patterns that select a toke nizer for a given line. The wildcard " `*`" can be used as a map key. The `fieldSetMappers` property (2) configures field set mappers.

This section ends our description of flat file support in Spring Batch. This support is powerful, flexible, and can handle varied data formats. T o complete our presentation of using files as input, we look atXML. The main difference between XML and flat files is that Java provides support for XML, which Spring Batch can use.

5.3 Reading XML files

As opposed to flat files, the Java runtime provides supports for XML. Java can process XML input using different techniques, including using a Streaming API for XML (StAX) parser. StAX is a standard XML-processing API that streams XML data to your application. StAX is particularly suitable to batch processing because streaming is a principal Spring Batch feature used to provide the best possible performance and memory consumption profile.

Batch performance and XML

Not all XML parsing approaches are suitable for obtaining the best performance for batch processes. For example, DOM (Document Object Model) loads all content in memory, and SAX (Simple API for XML) implements event-driven parsing. These two approaches aren't suitable and efficient in the context of batch processing because they don't support streaming.

Using XML alone in Java applications (and object-oriented applications) has limitations because of a mismatch between the XML and object-oriented models. To address this limitation and provide efficient conversion between XML and objects, the Spring framework includes the Object/XML Mapping framework (aka OXM or O/X Mapper). Spring OXM provides generic components called *marshaller* and *unmarshaller* to convert, respectively, objects to XML, and vice versa, as shown in figure 5.7.

In addition to the `Marshaller` and `Unmarshaller` interfaces, Spring OXM supports the object-to-XML mapping libraries listed in table 5.8.

Before diving into Spring Batch support for XML files, let's describe the XML vocabulary used for products in our case study. The following listing shows the contents of the file after conversion, as described in the section 5.2.3. This file format is the last supported import format in our case study.

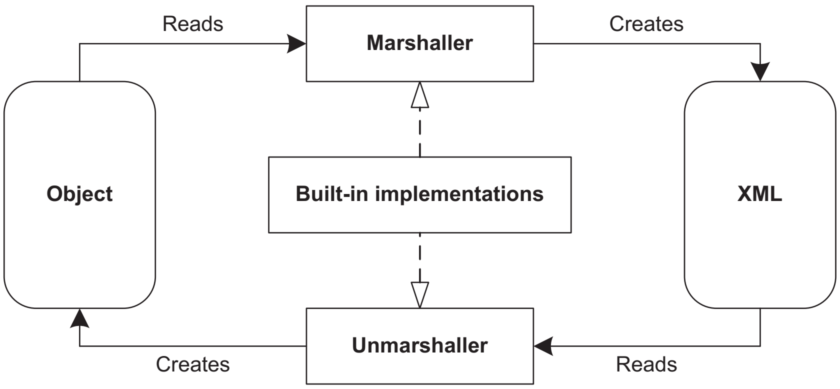

Figure 5.7 **Spring OXM components**

Library	Spring OXM `Marshaller` class
JAXB 1 and 2	`Jaxb1Marshaller` and `Jaxb2Marshaller`
Castor XML	`CastorMarshaller`
XMLBeans	`XmlBeansMarshaller`
JiBX	`JibxMarshaller`
XStream	`XStreamMarshaller`

Table 5.8 Built-in Spring OXM marshallers

Listing 5.9 XML product data converted from JSON

```
<products>
  <product>                                    <-+
    <id>PR....210</id>                           |
    <name>BlackBerry 8100 Pearl</name>           | 1 product
    <description/>                               |   element
    <price>124.60</price>                        |
  </product>                                     |
  <product>                                    <-+
    <id>PR....211</id>
    <name>Sony Ericsson W810i</name>
    <description/>
    <price>139.45</price>
  </product>
  (...)
</products>
```

Each product corresponds to a `product` XML element (1) under the root `products` element. Every product has four XML children elements for identifier, name, description, and price.

The `StaxEventItemReader` class implements the Spring Batch `ItemReader` interface using StAX to read XML documents. Because of its reliance on Spring OXM, it's independent of a parser implementation. Table 5.9 lists the `StaxEventItemReader` properties.

Table 5.9 `StaxEventItemReader` properties

Property	Description
`fragmentRootElementName`	The XML element name to import for each object.
`maxItemCount`	The maximum number of items to retrieve. The default value is `Integer.MAX_VALUE`.
`resource`	The resource to use as input. Because the property is of type `Resource`, you can use Spring to load the resource. See table 5.10.
`strict`	Whether the item reader throws an exception if the resource doesn't exist. The default is `false`.
`unmarshaller`	The Spring OXM `Unmarshaller` implementation used to convert XML to objects.

The key properties of the StaxEventItemReader class are fragmentRootElementName, used to identify the XML element to import, and unmarshaller to define XML-to-object conversions.

Table 5.10 lists most common built-in implementations of the Resource interface.

Table 5.10 Spring Resource implementations

Class	Description
UrlResource	Gets java.net.URL resources
ClassPathResource	Gets classpath resources
FileSystemResource	Gets java.io.File resources
ServletContextResource	Gets ServletContext resources from a web application
InputStreamResource	Gets java.io.InputStream resources
ByteArrayResource	Gets byte[] resources

The following listing describes how to configure a StaxEventItemReader bean to import product data from an XML file.

Listing 5.10 Configuring a StaxEventItemReader

```
<bean id="productItemReader"
      class="org.springframework.batch.item.xml.StaxEventItemReader">
  <property name="resource" value="datafile.xml"/>
  <property name="fragmentRootElementName" value="product"/>
  <property name="unmarshaller" ref="productMarshaller"/>
</bean>

<bean id="productMarshaller"
      class="org.springframework.oxm.castor.CastorMarshaller">
  <property name="mappingLocation"
         value="classpath:/com/manning/sbia/reading/xml/mapping.xml"/>
</bean>
```

You configure the StaxEventItemReader class by setting the value of the fragmentRootElementName property to product, which is the XML element name for a product. The unmarshaller property points to the bean definition used to convert XML to objects. You define this bean with the ID productMarshaller. This marshaller uses Castor through the Spring OXM class CastorMarshaller, which implements both the Marshaller and Unmarshaller interfaces.

Before tackling databases as input sources, we describe how to handle a file set with item readers. This approach is particularly useful for handling files in a directory. In our case study, this corresponds to product data files sent using FTP or Secure Copy (SCP) to an input directory.

5.4 Reading file sets

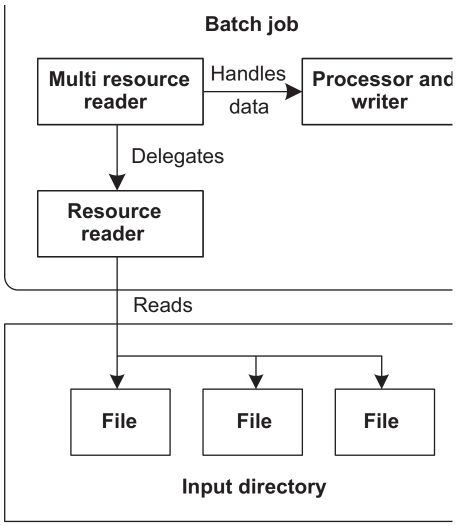

Figure 5.8 How Spring Batch reads data from multiple files. A multiresource reader delegates to a resource reader that reads files from an input directory.

Input can enter an application as a set of files, not only as a single file or resource. For example, files can periodically arrive via FTP or SCP in a dedicated input directory. In this case, the application doesn't know in advance the exact filename, but the names will follow a pattern that you can express as a regular expression. Figure 5.8 shows this architecture with Spring Batch. A dedicated multiresource reader accepts several resources as input and delegates processing to individual resource readers.

The `MultiResourceReader` class is the resource reader used for this input scenario. It handles multiple resources (`Resource[]`) and a delegate, `ResourceAwareItemReader-ItemStream`. The `MultiResourceReader` handles one resource at a time, sequentially, by iterating over all configured resources, and delegates processing to a resource-aware item reader.

This class is powerful because it leverages Spring resource support to easily configure multiple resources with simple patterns from different sources such as the file system or class path. A `MultiResourceReader` has two properties:

- `resources` configures resources with a list or patterns.
- `delegate` specifies the target `ResourceAwareItemReaderItemStream` to delegate processing for each resource.

The following XML fragment configures a `MultiResourceItemReader` to handle several files with an item reader. A file expression defines which files the reader uses as input:

```
<bean id="multiResourceReader"
      class="org.springframework.batch.item.file.MultiResourceItemReader">
  <property name="resources" value="file:/var/data/input/file-*.txt"/>
  <property name="delegate" ref="flatFileItemReader"/>
</bean>

<bean id="flatFileItemReader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  (...)
</bean>
```

The `resources` property configures the resource reader with files that match the pattern /var/data/input/file-*.txt. The `delegate` property references the identifier of the item reader used to process data. This it em reader is configured as a bean in the Spring configuration and must be designed to handle resources. In Spring Batch, these item readers implement the `ResourceAwareItemReaderItemStream` interface.

Going beyond the multiresource item reader with partitioning

When using the `MultiResourceItemReader`, Spring Batch reads files one after the other. The processing is, by default, single threaded. If you face performance issues when dealing with multiple input files, Spring Batch has built-in support to parallelize processing. Spring Batch can process each file on its own thread and calls this technique *partitioning*. Chapter 13 covers scaling strategies like partitioning.

Spring Batch provides broad, flexible, an d extensible support for flat files and XML. You can configure Spring Batch with the mo st common data file formats and integrate custom types. Next, we look at anot her input source for batch processes, relational databases.

5.5 Reading from relational databases

Another standard data input source is the relational database. In this case, data to import come from database tables. Spring Batch provides two approaches for batch processing to stream data from databases: JDBC and Object-Relational Mapping (ORM). W e first see how Spring Batch leverages JDBC to read data from relational databases.

5.5.1 Using JDBC item readers

JDBC is the Java platform component providing the interface to relational databases in Java. JDBC is a Java technology that makes it possible to interact with relational databases using the SQL language for both querying and updating data. JDBC, in principle at least, keeps things simple and independent from databases' implementations. JDBC provides an abstraction over the specifics of different databases by using the concept of a *driver*, which is responsible for implementing communication with a specific database, as shown in figure 5.9. But it doesn' t provide a complete solution to handle the specifics of each SQL dialect. The client application must deal with these issues. Spring uses JDBC but hides the JDBC plumbing and error -prone code and leaves the application to contain business-specific code.

Spring Batch bases its database support on the Spring JDBC layer and hides its use by managing request calls and transactions. In a batch job, you need to configure how to set request parameters and handle results. Spring Batch also bases its database support on the Spring `RowMapper` inter face and JDBC `PreparedStatement` inter face. In the next sections, we look at diff erent reading techniques based on JDBC and supported by Spring Batch.

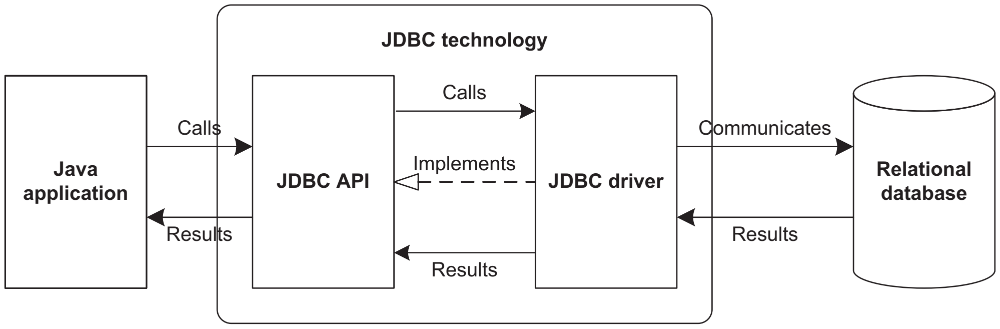

Figure 5.9 **High-level JDBC architecture. An application uses the vendor-neutral JDBC API. A database-specific JDBC driver implements communication with the database system.**

READING WITH DATABASE CURSORS AND JDBC RESULT SETS

In this approach, Spring Batch leaves the responsibility of reading data to the JDBC `ResultSet` interface. This interface is the object representation of a database cursor, which allows browsing result data of a `SELECT` statement. In this case, the result set integrates mechanisms to stream data. With this approach, Spring Batch executes only one request and retrieves result data progressively using JDBC with data batches, as shown in figure 5.10. Spring Batch relies on JDBC configuration and optimizations to perform efficiently.

The Spring Batch `JdbcCursorItemReader` class implements this technique and has the properties listed in table 5.11.

The minimal set of properties to use a `JdbcCursorItemReader` is `dataSource`, `sql`, and `rowMapper`. The properties specify the data source to access the database (`dataSource`), the SQL `SELECT` statement to execute to get data (`sql`), and the class to map

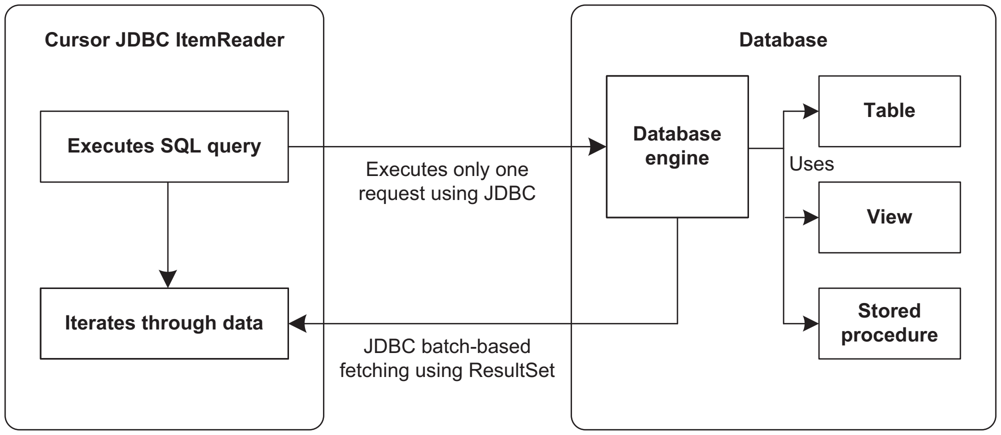

Figure 5.10 **Getting input data from a JDBC `ResultSet` corresponding to result of a SQL request within the `JdbcCursorItemReader` class**

Table 5.11 `JdbcCursorItemReader` properties

Property	Description
`dataSource`	The data source used to access the database.
`driverSupportsAbsolute`	Whether the JDBC driver supports using absolute row positioning on `ResultSet` using the `absolute` method. The default value is `false`.
`fetchSize`	The number of rows to fetch to transparently retrieve data by group. The default value is `-1`.
`ignoreWarnings`	Whether Spring Batch ignores SQL warnings. If `true`, Spring Batch logs warnings. If `false`, it throws exceptions when detecting warnings. The default value is `true`.
`maxRows`	The maximum number of rows that can be retrieved from SQL `SELECT` statements. The default value is `-1`.
`preparedStatementSetter`	The `PreparedStatementSetter` instance to set parameters for SQL statements.
`queryTimeout`	The maximum amount of time to wait for a response. If the timeout is exceeded, a `SQLException` is thrown. The default value is `-1`.
`rowMapper`	The `RowMapper` instance to build objects from `ResultSet` objects.
`sql`	The SQL `SELECT` to execute to get data.
`useSharedExtendedConnection`	Whether the connection is shared by the cursor and all other processing, therefore sharing the same transaction. If `false`, the cursor operates on its own connection and won't participate in any transactions started for the rest of the step processing. The default value is `false`.
`verifyCursorPosition`	Verifies the cursor position after processing the current row with `RowMapper` or `RowCallbackHandler`. The default value is `true`.

data into objects (`rowMapper`). If a statement has parameters, use the `preparedStatementSetter` property to set SQL statement parameters. In our use case, the SQL `SELECT` returns all the products to import from the `product` table and uses the `ProductRowMapper` class to convert data in the `ResultSet` to instances of the `Product` class. The following listing describes how to configure the `JdbcCursorItemReader` for this case.

Listing 5.11 Configuring a `JdbcCursorItemReader`

```
<bean id="productItemReader"
      class="org.springframework.batch.item.database.JdbcCursorItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="sql"
```

```
            value="select id, name, description, price from product"/>
  <property name="rowMapper"
            ref="productRowMapper"/>
</bean>

<bean id="productRowMapper"
      class="com.manning.sbia.reading.jdbc.ProductRowMapper"/>
```

You set the SQL `SELECT` statement to get product data from the `product` table using the `sql` property of a `JdbcCursorItemReader` instance. Next, you set the `RowMapper` implementation for the product mapper to a bean reference. Finally, you declare the product row mapper to use the `ProductRowMapper` class.

The following listing defines a `RowMapper` implementation called `ProductRowMapper`, which is used in all our JDBC examples.

Listing 5.12 `RowMapper` implementation for `Product`

```
public class ProductRowMapper implements RowMapper<Product> {
  public Product mapRow(ResultSet rs, int rowNum)          ❶ Maps result set
      throws SQLException {                                   to domain object
    Product product = new Product();
    product.setId(rs.getString("id"));
    product.setName(rs.getString("name"));
    product.setDescription(rs.getString("description"));
    product.setPrice(rs.getFloat("price"));
    return product;
  }
}
```

The `mapRow` method ❶ is a factory method that creates `Product` instances based on a given JDBC `ResultSet` and row number.

You use the `preparedStatementSetter` property if the SQL statement includes parameters. The property value is a class that works directly on the `PreparedStatement` instance managed internally by Spring to set the parameters. The following listing describes how to configure this feature in a`JdbcCursorItemReader` instance. In this listing, you get the subset of products for the names that start with "Samsung."

Listing 5.13 Setting SQL statement parameters in a `JdbcCursorItemReader`

```
<bean id="productItemReader"
      class="org.springframework.batch.item.database.JdbcCursorItemReader">
  <property name="dataSource" ref="datasource"/>
  <property name="sql"
            value="select id, name, description, price from product
                        where name like ?"/>
  <property name="preparedStatementSetter"
            ref="samsungStatementSetter"/>
  <property name="rowMapper" ref="productRowMapper"/>
</bean>

<bean id="samsungStatementSetter"
      class=" com.manning.sbia.reading.jdbc.SamsungStatementSetter"/>
```

You set the property `preparedStatementSetter` to reference the bean `samsungStatementSetter`. Then, you define the bean `samsungStatementSetter` as a `SamsungStatementSetter` that implements the `PreparedStatementSetter` interface. The `SamsungStatementSetter` class is

```
public class SamsungStatementSetter implements PreparedStatementSetter {
  void setValues(PreparedStatement ps) throws SQLException {
    ps.setString(1, "Samsung%");
  }
}
```

The `JdbcCursorItemReader` class provides advanced configuration for tuning batch processes. You can specify the maximum number of rows to retrieve through the `maxRows` property. The `fetchSize` property allows a reader to transparently retrieve data in fixed-sized groups. The following XML fragment describes how to configure these two properties:

```
<bean id="productItemReader"
      class="org.springframework.batch.item.database.JdbcCursorItemReader">
  <property name="dataSource" ref="datasource"/>
  <property name="sql"
            value="select id, name, description, price from product"/>
  <property name="rowMapper" ref="productRowMapper"/>
  <property name="maxRows" value="3000"/>
  <property name="fetchSize" value="100"/>
</bean>
```

In this example, you set the maximum number of rows to read to 3000 rows and the fetch group size to 100.

You don't always define SQL statements (here, a `SELECT`) outside the database in a configuration file; instead, a stored procedure can define the SQL to execute. This feature is supported by JDBC and Spring, and Spring Batch provides an `ItemReader` implementation for stored procedures using the cursor-based approach. This item reader class is called `StoredProcedureItemReader` and accepts the properties listed in table 5.12 in addition to the properties for `JdbcCursorItemReader` listed in table 5.11.

Table 5.12 `StoredProcedureItemReader` properties

Property	Description
`function`	Whether the stored procedure is a function. A function returns a value; a stored procedure doesn't.
`parameters`	Parameter types for the stored procedure.
`procedureName`	Name of the stored procedure.
`refCursorPosition`	When results of the stored procedure are returned using a ref cursor in an out parameter, this value specifies the position of this parameter in the parameter list. This index is 0-based, and its default value is `0`.

Using the `StoredProcedureItemReader` class is similar to using the `JdbcCursorItemReader` class. For common cases, replace the `sql` property with `procedureName`. In our case study, we have a stored procedure called `sp_product` that returns a product result set. The following XML fragment configures a `StoredProcedureItemReader`:

```
<bean id="reader"
 class="org.springframework.batch.item.database.StoredProcedureItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="procedureName" value="sp_product"/>     <-- Sets stored
  <property name="rowMapper" ref="productRowMapper"/>         procedure name
</bean>
```

By default, the `procedureName` property configures a stored procedure that returns a `ResultSet` instance.

Databases also provide another way to obtain results, reflected in Spring Batch with the `function` and `refCursorPosition` properties of the `StoredProcedureItemReader` class. When the SQL code executed in the database is a stored function call, the `function` property must be set to `true`. If a ref cursor in an out parameter returns data, then the `cursorRefPosition` property must be set to the position of the cursor in the output parameter list, as described in the following XML fragment:

```
<bean id="reader"
 class="org.springframework.batch.item.database.StoredProcedureItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="procedureName" value="sp_product"/>      1 Sets out
  <property name="parameters">                          <--    parameters
    <list>
      <bean class="org.springframework.jdbc.core.SqlOutParameter">
        <constructor-arg index="0" value="products"/>
        <constructor-arg index="1">
          <util:constant static-field="oracle.jdbc.OracleTypes.CURSOR"/>
        </constructor-arg>
      </bean>
    </list>
  </property>                                              2 Sets cursor
  <property name="cursorRefPosition" value="1"/>       <--    ref position
  <property name="rowMapper" ref="productRowMapper"/>
</bean>
```

In this example, you define parameters for a stored procedure. The procedure has one output parameter that corresponds to a cursor 1. Because only one parameter is involved, the cursor ref position is `1` and is set in the `StoredProcedureItemReader` using the `cursorRefPosition` property 2.

The Spring Batch cursor-based technique relies on JDBC and leverages streaming results using JDBC's own `ResultSet`. This mechanism allows retrieving data in batches and is useful with large data sets, as is often the case in batch processing. Spring Batch also allows you to control data set retrieval using data pages, which we see next.

USING PAGING TO MANAGE JDBC DATA RETRIEVAL

Instead of leaving JDBC to manage the data retrieval, Spring Batch allows you to handle this process using paging. In this case, retrieving data consists in successively executing

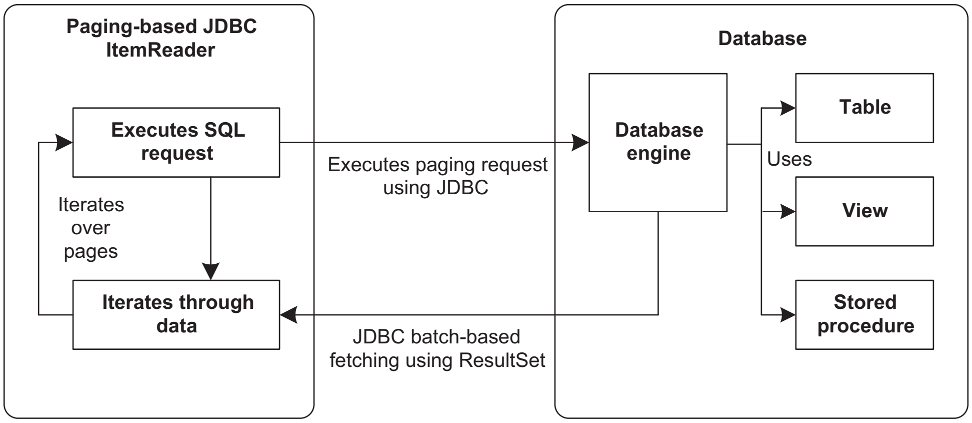

Figure 5.11 Using JDBC batch-based fetching to provide input data to an `ItemReader` by pages with fixed size

several requests with criteria. Spring Ba tch dynamically builds requests to execute based on a sort key to delimit data for a page. To retrieve each page, Spring Batch executes one request to retrieve the corresponding data. Figure 5.11 shows JDBC paging with an item reader.

Choosing between cursor-based and page-based item readers

Why does Spring Batch provide two ways to read from a database? The reason is that there's no one-size-fits-all solution. Cursor-based readers issue one query to the database and stream the data to avoid consuming too much memory. Cursor-based readers rely on the cursor implementation of the database and of the JDBC driver. Depending on your database engine and on the driver, cursor-based readers can work well . . . or not. Page-based readers work well with an appropriate page size (see the sidebar on page size). The trick is to find the best page size for your use case. With Spring Batch, switching from cursor- to page-based item readers is a matter of configuration and doesn't affect your application code. Don't hesitate to test both!

The `JdbcPagingItemReader` class is the new component we present here to implement JDBC paging with Spring Batch. The `JdbcPagingItemReader` class defines the properties listed in table 5.13.

Table 5.13 `JdbcPagingItemReader` properties

Property	Description
`dataSource`	Data source used to access the database
`fetchSize`	Number of rows to fetch to retrieve data in groups
`parameterValues`	Parameter values for the query

Table 5.13 `JdbcPagingItemReader` properties *(continued)*

Property	Description
`queryProvider`	`PagingQueryProvider` responsible for creating SQL requests used to retrieve data for pages
`rowMapper`	`RowMapper` instance used to build data object from the returned result set
`pageSize`	Number of rows per page

The following listing describes how to configure a `JdbcPagingItemReader` to retrieve product data using paging.

Listing 5.14 Configuring a `JdbcPagingItemReader`

```
<bean id="productItemReader"
      class="org.springframework.batch.item.database.JdbcPagingItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="queryProvider" ref="productQueryProvider"/>
  <property name="pageSize" value="1500"/>
  <property name="rowMapper" ref="productRowMapper"/>
</bean>

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

The `queryProvider` property configures an instance of `PagingQueryProvider` responsible for creating SQL queries to retrieve paged data. The generated SQL query limits the number of retrieved rows to the page size specified by the `pageSize` property. Finally, you set the `RowMapper` that creates business domain objects from `ResultSet` objects. For this example, the product `RowMapper` from listing 5.12 is reused.

Choosing a page size

There's no definitive value for the page-size setting. Let's venture a hint though: the size of a page should be around 1,000 items—this is a rule of thumb. The page size is usually higher than the commit interval (whose reasonable values range from 10 to 200 items). Remember, the point of paging is to avoid consuming too much memory, so large pages aren't good. Small pages aren't good either. If you read 1 million items in pages of 10 items (a small page size), you'll send 100,000 queries to the database. The good news is that the page size is a parameter in Spring Batch, so you can test multiple values and see which works best for your job.

Spring Batch provides a dedicated factory class called `SqlPagingQueryProvider-FactoryBean` used to configure a SQL paging query provider.

Table 5.14 lists the `SqlPagingQueryProviderFactoryBean` properties.

In our use case, we always need to have a SQL query returning products for cursor-based data retrieval. This query uses the `product` table and returns the id, name,

The Spring `FactoryBean` classes

When using a constructor (with the `new` keyword) to create beans doesn't fit your needs, use a Spring `FactoryBean`. A `FactoryBean` provides a level of indirection from the target bean by acting as a factory. After the Spring `BeanFactory` (be careful, the names are similar) initializes the factory bean, the target bean instances are obtained through the `getObject` method.

Table 5.14 **`SqlPagingQueryProviderFactoryBean` properties**

Property	Description
`ascending`	Whether sorting is ascending (or descending). The default value is `true` (ascending).
`databaseType`	The underlying database type. If you omit this property, the value is determined directly using the database through the specified data source.
`dataSource`	The data source to obtain connections to the database. Note that it's unnecessary to specify this field if the database type is set.
`fromClause`	The `FROM` clause of the SQL statement.
`selectClause`	The `SELECT` clause of the SQL statement.
`sortKey`	The sort column to identify data pages.
`whereClause`	Specifies the `WHERE` clause of the SQL statement.

description, and price columns. The following XML fragment describes how to configure this quer y for page-based data retrieval using the `SqlPagingQueryProviderFactoryBean` class:

```
<bean id=" productQueryProvider"
      class="org.springframework.batch.item.database
                          ➥ .support.SqlPagingQueryProviderFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="selectClause"
            value="select id, name, description, price"/>
  <property name="fromClause" value="from product"/>
  <property name="sortKey" value="id"/>
</bean>
```

After configuring the `SqlPagingQueryProviderFactoryBean` as a bean in the Spring configuration, you specify the SQL `SELECT` and `FROM` clauses followed by the sort key set to the `id` column. With this configuration, the `SqlPagingQueryProviderFactoryBean` configures and returns the appropriate class according to the database type in use. For example, for PostgreSQL, the `PostgresPagingQueryProvider` class is instantiated, configured, and returned. The returned class is then responsible for generating SQL paging queries. The quer y pattern is as follows: the first quer y is simple, using configured `SELECT`, `FROM`, and `WHERE` clauses with a hint limit for the number of returned rows:

```
SELECT id, name, description, price FROM product LIMIT 1500
```

For the next pages, queries include addition al clauses to specify the beginning of the page using the specified sort key:

```
SELECT id, name, description, price FROM product where id>? LIMIT 1500
```

As we've seen in this section, Spring Ba tch provides sophisticated integration with JDBC to support batch processes. As for nonbatch Java applications, we must explicitly define SQL queries to execute.

ORM tools provide interesting solutions to address this issue but don't account for batch processes in their design. Next, we se e the solutions Spring Batch provides for using ORM with batch applications.

5.5.2 Using ORM item readers

In traditional Java and Java EE applications, ORM is commonly used to interact with relational databases. The goal of ORM is to handle the mismatch between the relational database model and the object-oriented model. ORM tools efficiently manage conversions between these models. ORM tools also remove the need to explicitly specify SQL statements by automatically generating SQL on your behalf.

Is ORM the right tool for batch applications?

ORM works great for online applications but can be difficult to deal with in batch applications. It's not that ORM is a bad match for batch applications, but the high-level features ORM tools provide—such as lazy loading—don't always work well in batch scenarios. In Spring Batch, reading takes place in a separate transaction from processing and writing (this is a constraint of cursor- and page-based readers). This works well in normal cases, but when Murphy's law kicks in, the combination of a failure, the separate transaction, and lazy loading is explosive. Failure scenarios are numerous and tricky to solve. A solution is to apply the driving query pattern: the reader reads only item identifiers (using JDBC cursors or paging), and the processor uses the ORM tool to load the corresponding objects. In this case, a second-level cache can help performance. The goal is to have the ORM tool use the same transaction as the writer. Chapter 7 covers the driving query pattern.

As it does for JDBC, Spring provides supports for ORM. We don't describe here how Spring provides this support because Spring Batch does a good job of hiding it. Next, we focus on solutions (which are similar to JDBC) that Spring Batch provides to use ORM with batch processes efficiently.

READING WITH ORM CURSORS

Reading with ORM cursors implies that code responsible for managing domain classes doesn't use a first-level cache. Only Hibernate supports this feature through an interface called StatelessSession that provides the same methods as the classic Session but without caching and checking dirty state.

You first define a model class mapping to a relational database entity . For the online store case study, you define a Product class to map the product database table using Hibernate annotations, as shown in the following listing.

Listing 5.15 ORM mapping class

```
@Entity("product")
public class Product {
  @Id("id")
  private String id;
  @Column("label")
  private String label;
  @Column("description")
  private String description;
  @Column("price")
  private float price;
  (...)
}
```

The Entity annotation on the Product class specifies that the class maps to the product table. The Id and Column annotations map class properties to table columns using the name in the annotation values.

Using ORM cursors is similar to JDBC, shown in figure 5.10. The only difference is that ORM is an additional layer on top of JDBC. For ORM, Spring Batch only supports Hibernate through the HibernateCursorItemReader class, which accepts the properties listed in table 5.15.

The following listing shows how to configure a HibernateCursorItemReader bean for our case study to retrieve products from the product table.

Table 5.15 HibernateCursorItemReader properties

Property	Description
fetchSize	Number of rows to fetch transparently when retrieving data in groups.
maxItemCount	Maximum number of items to retrieve. The default value is Integer.MAX_VALUE.
parameterValues	Statement parameter values.
queryProvider	HibernateQueryProvider for creating Hibernate Query Language (HQL) queries to retrieve data pages.
queryString	The HQL query to retrieve entities.
sessionFactory	Hibernate SessionFactory for interacting with the database through Hibernate.
useStatelessSession	Whether the Hibernate session must be stateless. The default value is true.

Listing 5.16 Configuring a `HibernateCursorItemReader`

```
<bean id="productItemReader"
 class="org.springframework.batch.item.database.HibernateCursorItemReader">
  <property name="sessionFactory" ref="sessionFactory"/>
  <property name="queryString" value="from Product"/>
</bean>

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource"/>

  <property name="configurationClass"
            value="org.hibernate.cfg.AnnotationConfiguration"/>
  <property name="configLocation"
      value="classpath:/com/manning/sbia/reading/dao/hibernate.cfg.xml"/>
  <property name="hibernateProperties">
    <value>
      hibernate.dialect=org.hibernate.dialect.HSQLDialect
      hibernate.show_sql=true
    </value>
  </property>
</bean>
```

After configuring the `SessionFactory` using facilities provided by Spring Hibernate, this entity is configured with a `HibernateCursorItemReader` bean using the `sessionFactory` property. The `queryString` property sets the query to retrieve products.

The following XML fragment describes the content of `hibernate.cfg.xml`, which specifies that Hibernate must manage the `Product` class:

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <mapping class="com.manning.sbmia.reading.model.Product"/>
  </session-factory>
</hibernate-configuration>
```

Defines Product as managed class

As emphasized at the beginning of this section, only Hibernate supports this approach. For other ORM providers, Spring Batch provides a paging mechanism similar to the one described for JDBC.

USING PAGING TO MANAGE ORM DATA RETRIEVAL

ORM frameworks don't usually support the approach described in the previous section regarding Hibernate. For example, the Java Persistence API (JPA) technology doesn't provide cacheless support. In this case, paging is the natural solution because ORM caches only a page of objects in memory.

As with JDBC, ORM paging retrieves data in batches. Spring Batch performs data retrieval by successively executing several queries, as shown in figure 5.11. The only difference is that ORM is an additional layer on top of JDBC. For Hibernate, Spring Batch provides the `HibernatePagingItemReader` and `JpaPagingReader` classes for

Java Persistence API

The Java EE 5 specification includes ORM support defined as the JPA in Java Specification Request 220: Enterprise JavaBeans 3. Its aim is to provide a standardized layer so that ORM tools are implementations of this specification. The specification describes how to map managed entities to database tables and an API to interact with databases. New features are the ability to use this technology outside an Enterprise JavaBeans (EJB) container and to use local transactions instead of global transactions with JTA.

The main JPA implementations are Hibernate JPA, Apache OpenJPA, and EclipseLink JPA (based on TopLink).

JPA. Properties are almost the same as those described in table 5.15 with the addition of the `pageSize` property to specify the number of items per data page. Note that for the `JpaPagingReader` class, the `useStateless` property doesn't apply and the `queryProvider` property is of type `JpaQueryProvider`.

The following XML fragment describes how to configure a `HibernatePagingItemReader` bean to retrieve products from the `product` table:

```
<bean id="productItemReader"
 class="org.springframework.batch.item.database.HibernatePagingItemReader">
  <property name="sessionFactory" ref="sessionFactory"/>
  <property name="queryString" value="from Product"/>
</bean>

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

As you can see, configuring paging is similar to configuring cursors, and properties are generally the same; here you set the factory for the ORM and the query.

This section closes the description of relational databases as input sources using JDBC directly and through ORM with cursors and paging. Spring Batch integrates mechanisms to guarantee performance and memory consumption when using ORM for batch processes. In the next section, we focus on other input sources for importing data.

5.6 Using other input sources

Files and databases are the main data sources used as input for batch processes, but they aren't the only ones. You may want to reuse services provided by existing applications or integrate with an information system with asynchronous and event-driven features. Spring Batch provides implementations of the `ItemReader` interface for such cases, which this section examines.

5.6.1 Services as input

Reusability of existing services is a key concept of modern applications. This avoids reinventing the wheel, provides robustness, and saves time. Batch processes can integrate in existing systems that already provide entities to read data. These sources can

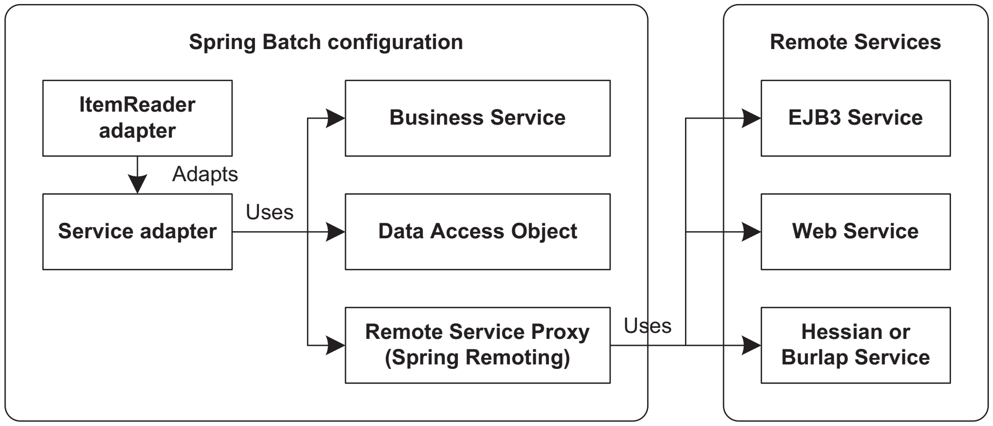

Figure 5.12 Reusing methods of existing entities and services to get data to provide as input for batch processes

be POJOs that implement business services; data access objects; or more complex entities managed by a container, such as EJB3 or by services accessible with lightweight protocols provided by Caucho, such as Hessian and Burlap. Figure 5.12 describes different patterns for batch processes to read data from existing entities.

To implement these patterns, Spring Batch provides the `ItemReaderAdapter` class, which makes it possible to see an existing entity as an `ItemReader`. The `ItemReaderAdapter` class holds the bean and method to delegate data retrieval. The only constraints at this level are that the delegated method must not have parameters and that it returns the same type as the `read` method of the `ItemReader` interface.

For this reason, it isn't possible to use the target service directly, and you must implement an adapter class for the service. The `ItemReader` adapter retrieves elements one by one, which isn't the case for services because they usually return a set of elements. The following listing shows the `ProductServiceAdapter` class, which implements this technique.

Listing 5.17 Service adapter for the `ProductService` service

```
public class ProductServiceAdapter implements InitializingBean {
  private ProductService productService;
  private List<Product> products;

  public void afterPropertiesSet() throws Exception {
    this.products = productService.getProducts();        ← ❶ Initializes products from service
  }

  public Product nextProduct() {
    if (products.size()>0) {
      return products.remove(0);        ← ❷ Gets products one by one
    } else {
      return null;
    }
  }
  public void setProductService(ProductService productService) {
    this.productService = productService;
  }
}
```

The ProductServiceAdapter class initializes the product list at startup using the afterPropertiesSet method ❶ from the Spring InitializingBean callback interface. Products are then retrieved one by one with the getProduct method ❷ using the product list initially loaded.

The next listing shows how to configure this mechanism for a POJO configured in Spring to reuse the ProductService entity managing products.

Listing 5.18 Configuring the ProductService as an ItemReader

```
<bean id="productItemReader"
      class="org.springframework.batch.item.adapter.ItemReaderAdapter">
  <property name="targetObject"                                   Sets product service object ❶
            ref="productServiceAdapter"/>
  <property name="targetMethod" value="nextProduct"/>             Sets target method ❷
</bean>

<bean id="productServiceAdapter"
      class="com.manning.sbia.reading.service.ProductServiceAdapter">
  <property name="productService" ref="productService"/>
</bean>

<bean id="productService"
      class="com.manning.sbia.reading.service.ProductServiceImpl">
  (...)
</bean>
```

Having configured the ProductService as a bean in the Spring configuration, you can reference it as the target object for the ItemReader adapter through the targetObject property ❶. The ItemReaderAdapter delegates import processing to the productService bean. You then specify which method to use to get product data with the targetMethod property ❷. In this case, the method is getProducts.

You adapted listing 5.18 to remotely access an EJB3 rather than a POJO using the http://www.springframework.org/schema/jee Spring XML namespace and jee namespace prefix. This vocabulary provides facilities for Java EE–related configuration for JNDI and EJB. The remote-slsb XML element configures a remote EJB3 proxy as a bean, which transparently provides a delegated business implementation. For our case study, the EJB3 corresponds to a remote service that manages products. The following snippet shows how to use a remote EJB3 with an ItemReader adapter:

```
<bean id="productItemReader"
      class="org.springframework.batch.item.adapter.ItemReaderAdapter">
  <property name="targetObject" ref="productService"/>
  <property name="targetMethod" value="nextProduct"/>
</bean>

<jee:remote-slsb id="productService"
                 jndi-name="ejb/remoteProductService">
```

For a remote EJB3 service, the configuration of the `ItemReader` adapter remains the same. For the `productService` bean, the configuration changes and the `remote-slsb` element's `jndi-name` property is set to the name in the JNDI entry for the EJB3 session.

Be aware that the target entity is entirely responsible for importing data in this case, and there's no possible interaction with the Spring Batch execution context. In fact, existing entities aren't linked to Spring Batch mechanisms and objects. The consequence is that state can't be stored. You also need to check that the use of the target entity in a batch process performs efficiently.

Before describing advanced issues regarding importing data with Spring Batch, we see how to implement and configure importing data using message-oriented middleware (MOM) and JMS.

5.6.2 Reading from JMS

The JMS API is to MOM what JDBC is to databases. JMS defines a vendor-neutral API and wrapper classes for MOM vendors to implement. MOM systems guarantee message delivery to applications and integrate fault tolerance, load scaling, and loosely coupled distributed communication and transaction support. JMS uses communication channels named *destinations* (like a *queue* or *topic*) to implement asynchronous communication.

The JMS specification tackles application messaging by providing a generic framework to send and receive messages synchronously and asynchronously. JMS provides a standard abstraction level for MOM providers to implement. In the context of batch processes, this makes it possible to handle incoming data automatically.

If JMS is event driven, why use it in batch applications?

One benefit of JMS is notification of new messages queued on destinations. Java objects wait to be called by containers like Spring's `MessageListenerContainers`. These Java objects are message-driven objects. Sometimes, you don't want to process messages as soon as they arrive because their processing is costly and you want to postpone this processing to reduce load on the server (which is busy doing something else at the moment). A batch job can consume JMS messages while throttling processing. You can choose to trigger the job when appropriate (every 10 minutes or at night, for example). The message-driven and batch approaches can work together: you can enable JMS listeners when your servers aren't too busy and disable them when there's too much load. You're also free to launch a dequeuing batch job whenever you want. This approach helps optimize usage of your hardware resources.

Spring Batch bases its JMS support on Spring's JMS support. The Spring Batch class `JmsItemReader` implements the `ItemReader` interface and internally uses the Spring `JmsTemplate` class. The `JmsItemReader` class reads data directly from a JMS destination (queue or topic). In the case study, the import job receives products as payload from JMS messages read from a JMS queue. The following listing shows how to configure reading from a JMS destination using a `JmsItemReader`.

Listing 5.19 Configuring a `JmsItemReader` class

```
<bean id="productItemReader"
      class="org.springframework.batch.item.jms.JmsItemReader">
  <property name="itemType"
            value="com.manning.sbia.reading.Product"/>        ❶ Defines item type
  <property name="jmsTemplate" ref="jmsTemplate"/>             ❷ Sets JMS template
</bean>

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" bean="jmsFactory"/>
  <property name="defaultDestination" ref="productDestination"/>
  <property name="receiveTimeout" value="500"/>
  <property name="sessionTransacted" value="true" />
</bean>

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

You first define the data object type contained in JMS messages ❶. For our case study, this type is `Product`. You then configure how to interact with the JMS provider through Spring's JMS support and its `JmsTemplate` class. To configure a `JmsTemplate`, you specify the JMS connection factory and destination. This template must then be set in the item reader using the `jmsTemplate` property ❷.

JMS and transactions

JMS provides transactional features for consuming messages. Chapter 9 includes guidelines to properly deal with JMS transactions.

In this chapter, we described all built-in capabilities of Spring Batch used to import data from different input sources. Spring Batch supports several file formats, relational databases, MOM, and reusing existing ser vices. In some cases, the `ItemReader` implementations provided by Spring Batc h aren't enough, and you must implement custom readers, which we discuss next.

5.7 Implementing custom readers

If Spring Batch `ItemReader` implementations don't suit your needs, you can provide your own implementations. We don't describe interacting with the execution context here because it's covered in chapter 8.

Imagine that you want to handle all file s present in a director y at batch startup. The list of files in the director y is loaded when the item reader is instantiated. For each read, you return the first list element af ter removing it from the list. The following listing shows the implementation of the `ListDirectoryItemReader` class.

Listing 5.20 Custom `ItemReader` implementation

```
public class ListDirectoryItemReader
        implements ItemReader<File> {
  private List<File> files;
  public ListDirectoryItemReader(File directory) {
    if (directory==null) {
      throw new IllegalArgumentException("The directory can be null.");
    }
    if (!directory.isDirectory()) {
      throw new IllegalArgumentException(
                           "The specified file must be a directory.");
    }
    files = Arrays.asList(directory.listFiles());
  }

  public File read() throws Exception, UnexpectedInputException,
                         ParseException, NonTransientResourceException {
    if (!files.isEmpty()) {
      return files.remove(0);
    }
    return null;
  }
}
```

As a custom item reader, this class implements the `ItemReader` interface. Because the `ItemReader` interface supports generics, you specify the associated type for the class (`File`). In the constructor, you initialize the list of files in the given director y. Implementing the `ItemReader` interface requires defining the `read` method. Spring Batch calls this method until it returns `null`, indicating that the method returned all files in the list, one at a time.

When you create a custom reader in Spring Batch, you implement the `ItemReader` interface. The `read` method performs all read processing, which returns elements one by one.

5.8 Summary

Reading data from batch processes is the first step of a chunk-based tasklet. Spring Batch provides support for this step with the generic `ItemReader` and `ItemStream` interfaces. Spring Batch implements these interfaces for common technologies used in batch processes to import data. Using ou r case study as an ex ample, we described how to read data from vari ous types of flat files and XML files. We also described how to get data from a database, how to integrat e with existing ser vices, and how to interact with a MOM like JMS.

We briefly mentioned that reading is in volved in the complete batch process execution. This aspect is fundamental to restart batch processes and avoid reading data again when things go wrong later in processing and writing. W e didn't go into details about reading, but we deal with this issue in chapter 8.

In chapter 6, we describe the other side of reading data—writing data—where Spring Batch supports the same technologies we saw in this chapter.

JAVA/SPRING

Spring Batch IN ACTION

Cogoluègnes • Templier • Gregory • Bazoud

Free ebook
SEE INSERT

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.

"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.

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

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]