Table of Contents

1. ItemReaders and ItemWriters 1
1.1. ItemReader 1
1.2. ItemWriter 2
1.3. ItemProcessor 2
1.3.1. Chaining ItemProcessors 5
1.3.2. Filtering Records 8
1.3.3. Fault Tolerance 9
1.4. ItemStream 9
1.5. The Delegate Pattern and Registering with the Step 9
1.6. Flat Files 11
1.6.1. The FieldSet 11
1.6.2. FlatFileItemReader 12
LineMapper 13
LineTokenizer 14
FieldSetMapper 14
DefaultLineMapper 14
Simple Delimited File Reading Example 15
Mapping Fields by Name 17
Automapping FieldSets to Domain Objects 18
Fixed Length File Formats 18
Multiple Record Types within a Single File 20
Exception Handling in Flat Files 22
1.6.3. FlatFileItemWriter 24
LineAggregator 24
Simplified File Writing Example 24
FieldExtractor 25
Delimited File Writing Example 27
Fixed Width File Writing Example 28
Handling File Creation 30
1.7. XML Item Readers and Writers 30
1.7.1. StaxEventItemReader 32
1.7.2. StaxEventItemWriter 35
1.8. JSON Item Readers And Writers 38
1.8.1. JsonItemReader 39
1.8.2. JsonFileItemWriter 40
1.9. Multi-File Input 40
1.10. Database 41
1.10.1. Cursor-based ItemReader Implementations 41
JdbcCursorItemReader 42
HibernateCursorItemReader 45

StoredProcedureItemReader 47
1.10.2. Paging ItemReader Implementations 51
JdbcPagingItemReader 51
JpaPagingItemReader 53
1.10.3. Database ItemWriters 54
1.11. Reusing Existing Services 56
1.12. Validating Input 57
1.13. Preventing State Persistence 59
1.14. Creating Custom ItemReaders and ItemWriters 60
1.14.1. Custom ItemReader Example 61
Making the ItemReader Restartable 61
1.14.2. Custom ItemWriter Example 63
Making the ItemWriter Restartable 64
1.15. Item Reader and Writer Implementations 64
1.15.1. Decorators 64
SynchronizedItemStreamReader 64
SingleItemPeekableItemReader 64
MultiResourceItemWriter 65
ClassifierCompositeItemWriter 65
ClassifierCompositeItemProcessor 65
1.15.2. Messaging Readers And Writers 65
AmqpItemReader 65
AmqpItemWriter 65
JmsItemReader 66
JmsItemWriter 66
1.15.3. Database Readers 66
Neo4jItemReader 66
MongoItemReader 66
HibernateCursorItemReader 66
HibernatePagingItemReader 66
RepositoryItemReader 66
1.15.4. Database Writers 67
Neo4jItemWriter 67
MongoItemWriter 67
RepositoryItemWriter 67
HibernateItemWriter 67
JdbcBatchItemWriter 67
JpaItemWriter 67
GemfireItemWriter 67
1.15.5. Specialized Readers 68
LdifReader 68
MappingLdifReader 68
1.15.6. Specialized Writers 68
SimpleMailMessageItemWriter 68
1.15.7. Specialized Processors 68
ScriptItemProcessor 68

Chapter 1. ItemReaders and ItemWriters

All batch processing can be described in its most simple form as reading in large amounts of data, performing some type of calculation or transformation, and writing the result out. Spring Batch provides three key interfaces to help perform bulk reading and writing: `ItemReader`, `ItemProcessor`, and `ItemWriter`.

1.1. `ItemReader`

Although a simple concept, an `ItemReader` is the means for providing data from many different types of input. The most general examples include:

- Flat File: Flat-file item readers read lines of data from a flat file that typically describes records with fields of data defined by fixed positions in the file or delimited by some special character (such as a comma).
- XML: XML `ItemReaders` process XML independently of technologies used for parsing, mapping and validating objects. Input data allows for the validation of an XML file against an XSD schema.
- Database: A database resource is accessed to return resultsets which can be mapped to objects for processing. The default SQL `ItemReader` implementations invoke a `RowMapper` to return objects, keep track of the current row if restart is required, store basic statistics, and provide some transaction enhancements that are explained later.

There are many more possibilities, but we focus on the basic ones for this chapter. A complete list of all available `ItemReader` implementations can be found in Appendix A.

`ItemReader` is a basic interface for generic input operations, as shown in the following interface definition:

```
public interface ItemReader<T> {

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

}
```

The `read` method defines the most essential contract of the `ItemReader`. Calling it returns one item or `null` if no more items are left. An item might represent a line in a file, a row in a database, or an element in an XML file. It is generally expected that these are mapped to a usable domain object (such as `Trade`, `Foo`, or others), but there is no requirement in the contract to do so.

It is expected that implementations of the `ItemReader` interface are forward only. However, if the underlying resource is transactional (such as a JMS queue) then calling `read` may return the same logical item on subsequent calls in a rollback scenario. It is also worth noting that a lack of items to process by an `ItemReader` does not cause an exception to be thrown. For example, a database `ItemReader` that is configured with a query that returns 0 results returns `null` on the first invocation of read.

1.2. `ItemWriter`

`ItemWriter` is similar in functionality to an `ItemReader` but with inverse operations. Resources still need to be located, opened, and closed but they differ in that an `ItemWriter` writes out, rather than reading in. In the case of databases or queues, these operations may be inserts, updates, or sends. The format of the serialization of the output is specific to each batch job.

As with `ItemReader`, `ItemWriter` is a fairly generic interface, as shown in the following interface definition:

```
public interface ItemWriter<T> {

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

}
```

As with `read` on `ItemReader`, `write` provides the basic contract of `ItemWriter`. It attempts to write out the list of items passed in as long as it is open. Because it is generally expected that items are 'batched' together into a chunk and then output, the interface accepts a list of items, rather than an item by itself. After writing out the list, any flushing that may be necessary can be performed before returning from the write method. For example, if writing to a Hibernate DAO, multiple calls to write can be made, one for each item. The writer can then call `flush` on the hibernate session before returning.

1.3. `ItemProcessor`

The `ItemReader` and `ItemWriter` interfaces are both very useful for their specific tasks, but what if you want to insert business logic before writing? One option for both reading and writing is to use the composite pattern: Create an `ItemWriter` that contains another `ItemWriter` or an `ItemReader` that contains another `ItemReader`. The following code shows an example:

```
public class CompositeItemWriter<T> implements ItemWriter<T> {

    ItemWriter<T> itemWriter;

    public CompositeItemWriter(ItemWriter<T> itemWriter) {
        this.itemWriter = itemWriter;
    }

    public void write(List<? extends T> items) throws Exception {
        //Add business logic here
       itemWriter.write(items);
    }

    public void setDelegate(ItemWriter<T> itemWriter){
        this.itemWriter = itemWriter;
    }
}
```

The preceding class contains another `ItemWriter` to which it delegates after having provided some business logic. This pattern could easily be used for an `ItemReader` as well, perhaps to obtain more reference data based upon the input that was provided by the main `ItemReader`. It is also useful if you need to control the call to `write` yourself. However, if you only want to 'transform' the item passed in for writing before it is actually written, you need not `write` yourself. You can just modify the item. For this scenario, Spring Batch provides the `ItemProcessor` interface, as shown in the following interface definition:

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

    O process(I item) throws Exception;
}
```

An `ItemProcessor` is simple. Given one object, transform it and return another. The provided object may or may not be of the same type. The point is that business logic may be applied within the process, and it is completely up to the developer to create that logic. An `ItemProcessor` can be wired directly into a step. For example, assume an `ItemReader` provides a class of type `Foo` and that it needs to be converted to type `Bar` before being written out. The following example shows an `ItemProcessor` that performs the conversion:

```java
public class Foo {}

public class Bar {
    public Bar(Foo foo) {}
}

public class FooProcessor implements ItemProcessor<Foo,Bar>{
    public Bar process(Foo foo) throws Exception {
        //Perform simple transformation, convert a Foo to a Bar
        return new Bar(foo);
    }
}

public class BarWriter implements ItemWriter<Bar>{
    public void write(List<? extends Bar> bars) throws Exception {
        //write bars
    }
}
```

In the preceding example, there is a class `Foo`, a class `Bar`, and a class `FooProcessor` that adheres to the `ItemProcessor` interface. The transformation is simple, but any type of transformation could be done here. The `BarWriter` writes `Bar` objects, throwing an exception if any other type is provided. Similarly, the `FooProcessor` throws an exception if anything but a `Foo` is provided. The `FooProcessor` can then be injected into a `Step`, as shown in the following example:

XML Configuration

```xml
<job id="ioSampleJob">
    <step name="step1">
        <tasklet>
            <chunk reader="fooReader" processor="fooProcessor" writer="barWriter"
                   commit-interval="2"/>
        </tasklet>
    </step>
</job>
```

Java Configuration

```
@Bean
public Job ioSampleJob() {
    return this.jobBuilderFactory.get("ioSampleJOb")
                .start(step1())
                .end()
                .build();
}

@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
                .<String, String>chunk(2)
                .reader(fooReader())
                .processor(fooProcessor())
                .writer(barWriter())
                .build();
}
```

1.3.1. Chaining ItemProcessors

Performing a single transformation is useful in many scenarios, but what if you want to 'chain' together multiple `ItemProcessor` implementations? This can be accomplished using the composite pattern mentioned previously. To update the previous, single transformation, example, `Foo` is transformed to `Bar`, which is transformed to `Foobar` and written out, as shown in the following example:

```
public class Foo {}

public class Bar {
    public Bar(Foo foo) {}
}

public class Foobar {
    public Foobar(Bar bar) {}
}

public class FooProcessor implements ItemProcessor<Foo,Bar>{
    public Bar process(Foo foo) throws Exception {
        //Perform simple transformation, convert a Foo to a Bar
        return new Bar(foo);
    }
}

public class BarProcessor implements ItemProcessor<Bar,Foobar>{
    public Foobar process(Bar bar) throws Exception {
        return new Foobar(bar);
    }
}

public class FoobarWriter implements ItemWriter<Foobar>{
    public void write(List<? extends Foobar> items) throws Exception {
        //write items
    }
}
```

A `FooProcessor` and a `BarProcessor` can be 'chained' together to give the resultant `Foobar`, as shown in the following example:

```
CompositeItemProcessor<Foo,Foobar> compositeProcessor =
                                      new CompositeItemProcessor<Foo,Foobar>();
List itemProcessors = new ArrayList();
itemProcessors.add(new FooTransformer());
itemProcessors.add(new BarTransformer());
compositeProcessor.setDelegates(itemProcessors);
```

Just as with the previous example, the composite processor can be configured into the `Step`:

XML Configuration

```
<job id="ioSampleJob">
    <step name="step1">
        <tasklet>
            <chunk reader="fooReader" processor="compositeItemProcessor" writer=
"foobarWriter"
                   commit-interval="2"/>
        </tasklet>
    </step>
</job>

<bean id="compositeItemProcessor"
      class="org.springframework.batch.item.support.CompositeItemProcessor">
    <property name="delegates">
        <list>
            <bean class="..FooProcessor" />
            <bean class="..BarProcessor" />
        </list>
    </property>
</bean>
```

```
@Bean
public Job ioSampleJob() {
    return this.jobBuilderFactory.get("ioSampleJob")
                .start(step1())
                .end()
                .build();
}

@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
                .<String, String>chunk(2)
                .reader(fooReader())
                .processor(compositeProcessor())
                .writer(foobarWriter())
                .build();
}

@Bean
public CompositeItemProcessor compositeProcessor() {
    List<ItemProcessor> delegates = new ArrayList<>(2);
    delegates.add(new FooProcessor());
    delegates.add(new BarProcessor());

    CompositeItemProcessor processor = new CompositeItemProcessor();

    processor.setDelegates(delegates);

    return processor;
}
```

1.3.2. Filtering Records

One typical use for an item processor is to filter out records before they are passed to the `ItemWriter`. Filtering is an action distinct from skipping. Skipping indicates that a record is invalid, while filtering simply indicates that a record should not be written.

For example, consider a batch job that reads a file containing three different types of records: records to insert, records to update, and records to delete. If record deletion is not supported by the system, then we would not want to send any "delete" records to the `ItemWriter`. But, since these records are not actually bad records, we would want to filter them out rather than skip them. As a result, the `ItemWriter` would receive only "insert" and "update" records.

To filter a record, you can return `null` from the `ItemProcessor`. The framework detects that the result is `null` and avoids adding that item to the list of records delivered to the `ItemWriter`. As usual, an exception thrown from the `ItemProcessor` results in a skip.

1.3.3. Fault Tolerance

When a chunk is rolled back, items that have been cached during reading may be reprocessed. If a step is configured to be fault tolerant (typically by using skip or retry processing), any `ItemProcessor` used should be implemented in a way that is idempotent. Typically that would consist of performing no changes on the input item for the `ItemProcessor` and only updating the instance that is the result.

1.4. `ItemStream`

Both `ItemReaders` and `ItemWriters` serve their individual purposes well, but there is a common concern among both of them that necessitates another interface. In general, as part of the scope of a batch job, readers and writers need to be opened, closed, and require a mechanism for persisting state. The `ItemStream` interface serves that purpose, as shown in the following example:

```
public interface ItemStream {

    void open(ExecutionContext executionContext) throws ItemStreamException;

    void update(ExecutionContext executionContext) throws ItemStreamException;

    void close() throws ItemStreamException;
}
```

Before describing each method, we should mention the `ExecutionContext`. Clients of an `ItemReader` that also implement `ItemStream` should call `open` before any calls to `read`, in order to open any resources such as files or to obtain connections. A similar restriction applies to an `ItemWriter` that implements `ItemStream`. As mentioned in Chapter 2, if expected data is found in the `ExecutionContext`, it may be used to start the `ItemReader` or `ItemWriter` at a location other than its initial state. Conversely, `close` is called to ensure that any resources allocated during open are released safely. `update` is called primarily to ensure that any state currently being held is loaded into the provided `ExecutionContext`. This method is called before committing, to ensure that the current state is persisted in the database before commit.

In the special case where the client of an `ItemStream` is a `Step` (from the Spring Batch Core), an `ExecutionContext` is created for each StepExecution to allow users to store the state of a particular execution, with the expectation that it is returned if the same `JobInstance` is started again. For those familiar with Quartz, the semantics are very similar to a Quartz `JobDataMap`.

1.5. The Delegate Pattern and Registering with the Step

Note that the `CompositeItemWriter` is an example of the delegation pattern, which is common in Spring Batch. The delegates themselves might implement callback interfaces, such as `StepListener`. If they do and if they are being used in conjunction with Spring Batch Core as part of a `Step` in a `Job`, then they almost certainly need to be registered manually with the `Step`. A reader, writer, or processor that is directly wired into the `Step` gets registered automatically if it implements `ItemStream` or a `StepListener` interface. However, because the delegates are not known to the `Step`, they need to be injected as listeners or streams (or both if appropriate), as shown in the following example:

XML Configuration

```
<job id="ioSampleJob">
    <step name="step1">
        <tasklet>
            <chunk reader="fooReader" processor="fooProcessor" writer=
"compositeItemWriter"
                   commit-interval="2">
                <streams>
                    <stream ref="barWriter" />
                </streams>
            </chunk>
        </tasklet>
    </step>
</job>

<bean id="compositeItemWriter" class="...CustomCompositeItemWriter">
    <property name="delegate" ref="barWriter" />
</bean>

<bean id="barWriter" class="...BarWriter" />
```

Java Configuration

```
@Bean
public Job ioSampleJob() {
    return this.jobBuilderFactory.get("ioSampleJob")
                .start(step1())
                .end()
                .build();
}

@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
                .<String, String>chunk(2)
                .reader(fooReader())
                .processor(fooProcessor())
                .writer(compositeItemWriter())
                .stream(barWriter())
                .build();
}

@Bean
public CustomCompositeItemWriter compositeItemWriter() {

    CustomCompositeItemWriter writer = new CustomCompositeItemWriter();

    writer.setDelegate(barWriter());

    return writer;
}

@Bean
public BarWriter barWriter() {
    return new BarWriter();
}
```

1.6. Flat Files

One of the most common mechanisms for interchanging bulk data has always been the flat file. Unlike XML, which has an agreed upon standard for defining how it is structured (XSD), anyone reading a flat file must understand ahead of time exactly how the file is structured. In general, all flat files fall into two types: delimited and fixed length. Delimited files are those in which fields are separated by a delimiter, such as a comma. Fixed Length files have fields that are a set length.

1.6.1. The `FieldSet`

When working with flat files in Spring Batch, regardless of whether it is for input or output, one of the most important classes is the `FieldSet`. Many architectures and libraries contain abstractions for helping you read in from a file, but they usually return a `String` or an array of `String` objects.

This really only gets you halfway there. A FieldSet is Spring Batch's abstraction for enabling the binding of fields from a file resource. It allows developers to work with file input in much the same way as they would work with database input. A FieldSet is conceptually similar to a JDBC ResultSet. A FieldSet requires only one argument: a String array of tokens. Optionally, you can also configure the names of the fields so that the fields may be accessed either by index or name as patterned after ResultSet, as shown in the following example:

```
String[] tokens = new String[]{"foo", "1", "true"};
FieldSet fs = new DefaultFieldSet(tokens);
String name = fs.readString(0);
int value = fs.readInt(1);
boolean booleanValue = fs.readBoolean(2);
```

There are many more options on the FieldSet interface, such as Date, long, BigDecimal, and so on. The biggest advantage of the FieldSet is that it provides consistent parsing of flat file input. Rather than each batch job parsing differently in potentially unexpected ways, it can be consistent, both when handling errors caused by a format exception, or when doing simple data conversions.

1.6.2. FlatFileItemReader

A flat file is any type of file that contains at most two-dimensional (tabular) data. Reading flat files in the Spring Batch framework is facilitated by the class called FlatFileItemReader, which provides basic functionality for reading and parsing flat files. The two most important required dependencies of FlatFileItemReader are Resource and LineMapper. The LineMapper interface is explored more in the next sections. The resource property represents a Spring Core Resource. Documentation explaining how to create beans of this type can be found in Spring Framework, Chapter 5. Resources. Therefore, this guide does not go into the details of creating Resource objects beyond showing the following simple example:

```
Resource resource = new FileSystemResource("resources/trades.csv");
```

In complex batch environments, the directory structures are often managed by the EAI infrastructure, where drop zones for external interfaces are established for moving files from FTP locations to batch processing locations and vice versa. File moving utilities are beyond the scope of the Spring Batch architecture, but it is not unusual for batch job streams to include file moving utilities as steps in the job stream. The batch architecture only needs to know how to locate the files to be processed. Spring Batch begins the process of feeding the data into the pipe from this starting point. However, Spring Integration provides many of these types of services.

The other properties in FlatFileItemReader let you further specify how your data is interpreted, as described in the following table:

Table 1. FlatFileItemReader Properties

Property	Type	Description
comments	String[]	Specifies line prefixes that indicate comment rows.

Property	Type	Description
encoding	String	Specifies what text encoding to use. The default is the value of `Charset.defaultCharset()`.
lineMapper	`LineMapper`	Converts a `String` to an `Object` representing the item.
linesToSkip	int	Number of lines to ignore at the top of the file.
recordSeparatorPolicy	RecordSeparatorPolicy	Used to determine where the line endings are and do things like continue over a line ending if inside a quoted string.
resource	`Resource`	The resource from which to read.
skippedLinesCallback	LineCallbackHandler	Interface that passes the raw line content of the lines in the file to be skipped. If `linesToSkip` is set to 2, then this interface is called twice.
strict	boolean	In strict mode, the reader throws an exception on `ExecutionContext` if the input resource does not exist. Otherwise, it logs the problem and continues.

`LineMapper`

As with `RowMapper`, which takes a low-level construct such as `ResultSet` and returns an `Object`, flat file processing requires the same construct to convert a `String` line into an `Object`, as shown in the following interface definition:

```
public interface LineMapper<T> {

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

}
```

The basic contract is that, given the current line and the line number with which it is associated, the mapper should return a resulting domain object. This is similar to `RowMapper`, in that each line is associated with its line number, just as each row in a `ResultSet` is tied to its row number. This allows the line number to be tied to the resulting domain object for identity comparison or for more informative logging. However, unlike `RowMapper`, the `LineMapper` is given a raw line which, as discussed above, only gets you halfway there. The line must be tokenized into a `FieldSet`, which can then be mapped to an object, as described later in this document.

`LineTokenizer`

An abstraction for turning a line of input into a `FieldSet` is necessary because there can be many formats of flat file data that need to be converted to a `FieldSet`. In Spring Batch, this interface is the `LineTokenizer`:

```
public interface LineTokenizer {

    FieldSet tokenize(String line);

}
```

The contract of a `LineTokenizer` is such that, given a line of input (in theory the `String` could encompass more than one line), a `FieldSet` representing the line is returned. This `FieldSet` can then be passed to a `FieldSetMapper`. Spring Batch contains the following `LineTokenizer` implementations:

- `DelimitedLineTokenizer`: Used for files where fields in a record are separated by a delimiter. The most common delimiter is a comma, but pipes or semicolons are often used as well.
- `FixedLengthTokenizer`: Used for files where fields in a record are each a "fixed width". The width of each field must be defined for each record type.
- `PatternMatchingCompositeLineTokenizer`: Determines which `LineTokenizer` among a list of tokenizers should be used on a particular line by checking against a pattern.

`FieldSetMapper`

The `FieldSetMapper` interface defines a single method, `mapFieldSet`, which takes a `FieldSet` object and maps its contents to an object. This object may be a custom DTO, a domain object, or an array, depending on the needs of the job. The `FieldSetMapper` is used in conjunction with the `LineTokenizer` to translate a line of data from a resource into an object of the desired type, as shown in the following interface definition:

```
public interface FieldSetMapper<T> {

    T mapFieldSet(FieldSet fieldSet) throws BindException;

}
```

The pattern used is the same as the `RowMapper` used by `JdbcTemplate`.

`DefaultLineMapper`

Now that the basic interfaces for reading in flat files have been defined, it becomes clear that three basic steps are required:

1. Read one line from the file.
2. Pass the `String` line into the `LineTokenizer#tokenize()` method to retrieve a `FieldSet`.
3. Pass the `FieldSet` returned from tokenizing to a `FieldSetMapper`, returning the result from the

ItemReader#read() method.

The two interfaces described above represent two separate tasks: converting a line into a FieldSet and mapping a FieldSet to a domain object. Because the input of a LineTokenizer matches the input of the LineMapper (a line), and the output of a FieldSetMapper matches the output of the LineMapper, a default implementation that uses both a LineTokenizer and a FieldSetMapper is provided. The DefaultLineMapper, shown in the following class definition, represents the behavior most users need:

```
public class DefaultLineMapper<T> implements LineMapper<>, InitializingBean {

    private LineTokenizer tokenizer;

    private FieldSetMapper<T> fieldSetMapper;

    public T mapLine(String line, int lineNumber) throws Exception {
        return fieldSetMapper.mapFieldSet(tokenizer.tokenize(line));
    }

    public void setLineTokenizer(LineTokenizer tokenizer) {
        this.tokenizer = tokenizer;
    }

    public void setFieldSetMapper(FieldSetMapper<T> fieldSetMapper) {
        this.fieldSetMapper = fieldSetMapper;
    }
}
```

The above functionality is provided in a default implementation, rather than being built into the reader itself (as was done in previous versions of the framework) to allow users greater flexibility in controlling the parsing process, especially if access to the raw line is needed.

Simple Delimited File Reading Example

The following example illustrates how to read a flat file with an actual domain scenario. This particular batch job reads in football players from the following file:

```
ID,lastName,firstName,position,birthYear,debutYear
"AbduKa00,Abdul-Jabbar,Karim,rb,1974,1996",
"AbduRa00,Abdullah,Rabih,rb,1975,1999",
"AberWa00,Abercrombie,Walter,rb,1959,1982",
"AbraDa00,Abramowicz,Danny,wr,1945,1967",
"AdamBo00,Adams,Bob,te,1946,1969",
"AdamCh00,Adams,Charlie,wr,1979,2003"
```

The contents of this file are mapped to the following Player domain object:

```java
public class Player implements Serializable {

    private String ID;
    private String lastName;
    private String firstName;
    private String position;
    private int birthYear;
    private int debutYear;

    public String toString() {
        return "PLAYER:ID=" + ID + ",Last Name=" + lastName +
            ",First Name=" + firstName + ",Position=" + position +
            ",Birth Year=" + birthYear + ",DebutYear=" +
            debutYear;
    }

    // setters and getters...
}
```

To map a `FieldSet` into a `Player` object, a `FieldSetMapper` that returns players needs to be defined, as shown in the following example:

```java
protected static class PlayerFieldSetMapper implements FieldSetMapper<Player> {
    public Player mapFieldSet(FieldSet fieldSet) {
        Player player = new Player();

        player.setID(fieldSet.readString(0));
        player.setLastName(fieldSet.readString(1));
        player.setFirstName(fieldSet.readString(2));
        player.setPosition(fieldSet.readString(3));
        player.setBirthYear(fieldSet.readInt(4));
        player.setDebutYear(fieldSet.readInt(5));

        return player;
    }
}
```

The file can then be read by correctly constructing a `FlatFileItemReader` and calling `read`, as shown in the following example:

```
FlatFileItemReader<Player> itemReader = new FlatFileItemReader<Player>();
itemReader.setResource(new FileSystemResource("resources/players.csv"));
//DelimitedLineTokenizer defaults to comma as its delimiter
DefaultLineMapper<Player> lineMapper = new DefaultLineMapper<Player>();
lineMapper.setLineTokenizer(new DelimitedLineTokenizer());
lineMapper.setFieldSetMapper(new PlayerFieldSetMapper());
itemReader.setLineMapper(lineMapper);
itemReader.open(new ExecutionContext());
Player player = itemReader.read();
```

Each call to `read` returns a new `Player` object from each line in the file. When the end of the file is reached, `null` is returned.

Mapping Fields by Name

There is one additional piece of functionality that is allowed by both `DelimitedLineTokenizer` and `FixedLengthTokenizer` and that is similar in function to a JDBC `ResultSet`. The names of the fields can be injected into either of these `LineTokenizer` implementations to increase the readability of the mapping function. First, the column names of all fields in the flat file are injected into the tokenizer, as shown in the following example:

```
tokenizer.setNames(new String[] {"ID", "lastName","firstName","position","birthYear",
"debutYear"});
```

A `FieldSetMapper` can use this information as follows:

```
public class PlayerMapper implements FieldSetMapper<Player> {
    public Player mapFieldSet(FieldSet fs) {

       if(fs == null){
           return null;
       }

       Player player = new Player();
       player.setID(fs.readString("ID"));
       player.setLastName(fs.readString("lastName"));
       player.setFirstName(fs.readString("firstName"));
       player.setPosition(fs.readString("position"));
       player.setDebutYear(fs.readInt("debutYear"));
       player.setBirthYear(fs.readInt("birthYear"));

       return player;
   }
}
```

Automapping FieldSets to Domain Objects

For many, having to write a specific `FieldSetMapper` is equally as cumbersome as writing a specific `RowMapper` for a `JdbcTemplate`. Spring Batch makes this easier by providing a `FieldSetMapper` that automatically maps fields by matching a field name with a setter on the object using the JavaBean specification. Again using the football example, the `BeanWrapperFieldSetMapper` configuration looks like the following snippet:

XML Configuration

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

<bean id="player"
      class="org.springframework.batch.sample.domain.Player"
      scope="prototype" />
```

Java Configuration

```
@Bean
public FieldSetMapper fieldSetMapper() {
    BeanWrapperFieldSetMapper fieldSetMapper = new BeanWrapperFieldSetMapper();

    fieldSetMapper.setPrototypeBeanName("player");

    return fieldSetMapper;
}

@Bean
@Scope("prototype")
public Player player() {
    return new Player();
}
```

For each entry in the `FieldSet`, the mapper looks for a corresponding setter on a new instance of the `Player` object (for this reason, prototype scope is required) in the same way the Spring container looks for setters matching a property name. Each available field in the `FieldSet` is mapped, and the resultant `Player` object is returned, with no code required.

Fixed Length File Formats

So far, only delimited files have been discussed in much detail. However, they represent only half of the file reading picture. Many organizations that use flat files use fixed length formats. An example fixed length file follows:

```
UK21341EAH4121131.11customer1
UK21341EAH4221232.11customer2
UK21341EAH4321333.11customer3
UK21341EAH4421434.11customer4
UK21341EAH4521535.11customer5
```

While this looks like one large field, it actually represent 4 distinct fields:

1. ISIN: Unique identifier for the item being ordered - 12 characters long.
2. Quantity: Number of the item being ordered - 3 characters long.
3. Price: Price of the item - 5 characters long.
4. Customer: ID of the customer ordering the item - 9 characters long.

When configuring the `FixedLengthLineTokenizer`, each of these lengths must be provided in the form of ranges, as shown in the following example:

XML Configuration

```xml
<bean id="fixedLengthLineTokenizer"
      class="org.springframework.batch.io.file.transform.FixedLengthTokenizer">
    <property name="names" value="ISIN,Quantity,Price,Customer" />
    <property name="columns" value="1-12, 13-15, 16-20, 21-29" />
</bean>
```

Because the `FixedLengthLineTokenizer` uses the same `LineTokenizer` interface as discussed above, it returns the same `FieldSet` as if a delimiter had been used. This allows the same approaches to be used in handling its output, such as using the `BeanWrapperFieldSetMapper`.

> Supporting the above syntax for ranges requires that a specialized property editor, `RangeArrayPropertyEditor`, be configured in the `ApplicationContext`. However, this bean is automatically declared in an `ApplicationContext` where the batch namespace is used.

Java Configuration

```
@Bean
public FixedLengthTokenizer fixedLengthTokenizer() {
    FixedLengthTokenizer tokenizer = new FixedLengthTokenizer();

    tokenizer.setNames("ISIN", "Quantity", "Price", "Customer");
    tokenizer.setColumns(new Range(1-12),
                        new Range(13-15),
                        new Range(16-20),
                        new Range(21-29));

    return tokenizer;
}
```

Because the `FixedLengthLineTokenizer` uses the same `LineTokenizer` interface as discussed above, it returns the same `FieldSet` as if a delimiter had been used. This lets the same approaches be used in handling its output, such as using the `BeanWrapperFieldSetMapper`.

Multiple Record Types within a Single File

All of the file reading examples up to this point have all made a key assumption for simplicity's sake: all of the records in a file have the same format. However, this may not always be the case. It is very common that a file might have records with different formats that need to be tokenized differently and mapped to different objects. The following excerpt from a file illustrates this:

```
USER;Smith;Peter;;T;20014539;F
LINEA;1044391041ABC037.49G201XX1383.12H
LINEB;2134776319DEF422.99M005LI
```

In this file we have three types of records, "USER", "LINEA", and "LINEB". A "USER" line corresponds to a `User` object. "LINEA" and "LINEB" both correspond to `Line` objects, though a "LINEA" has more information than a "LINEB".

The `ItemReader` reads each line individually, but we must specify different `LineTokenizer` and `FieldSetMapper` objects so that the `ItemWriter` receives the correct items. The `PatternMatchingCompositeLineMapper` makes this easy by allowing maps of patterns to `LineTokenizer` instances and patterns to `FieldSetMapper` instances to be configured, as shown in the following example:

XML Configuration

```
<bean id="orderFileLineMapper"
      class="org.spr...PatternMatchingCompositeLineMapper">
    <property name="tokenizers">
        <map>
            <entry key="USER*" value-ref="userTokenizer" />
            <entry key="LINEA*" value-ref="lineATokenizer" />
            <entry key="LINEB*" value-ref="lineBTokenizer" />
        </map>
    </property>
    <property name="fieldSetMappers">
        <map>
            <entry key="USER*" value-ref="userFieldSetMapper" />
            <entry key="LINE*" value-ref="lineFieldSetMapper" />
        </map>
    </property>
</bean>
```

Java Configuration

```
@Bean
public PatternMatchingCompositeLineMapper orderFileLineMapper() {
    PatternMatchingCompositeLineMapper lineMapper =
        new PatternMatchingCompositeLineMapper();

    Map<String, LineTokenizer> tokenizers = new HashMap<>(3);
    tokenizers.put("USER*", userTokenizer());
    tokenizers.put("LINEA*", lineATokenizer());
    tokenizers.put("LINEB*", lineBTokenizer());

    lineMapper.setTokenizers(tokenizers);

    Map<String, FieldSetMapper> mappers = new HashMap<>(2);
    mappers.put("USER*", userFieldSetMapper());
    mappers.put("LINE*", lineFieldSetMapper());

    lineMapper.setFieldSetMappers(mappers);

    return lineMapper;
}
```

In this example, "LINEA" and "LINEB" have separate `LineTokenizer` instances, but they both use the same `FieldSetMapper`.

The `PatternMatchingCompositeLineMapper` uses the `PatternMatcher#match` method in order to select the correct delegate for each line. The `PatternMatcher` allows for two wildcard characters with special meaning: the question mark ("?") matches exactly one character, while the asterisk ("*") matches zero or more characters. Note that, in the preceding configuration, all patterns end with an asterisk, making them effectively prefixes to lines. The PatternMatcher always matches the most specific pattern possible, regardless of the order in the configuration. So if "LINE*" and "LINEA*" were both listed as patterns, "LINEA" would match pattern "LINEA*", while "LINEB" would match pattern "LINE*". Additionally, a single asterisk ("*") can serve as a default by matching any line not matched by any other pattern, as shown in the following example.

XML Configuration

```
<entry key="*" value-ref="defaultLineTokenizer" />
```

Java Configuration

```
...
tokenizers.put("*", defaultLineTokenizer());
...
```

There is also a PatternMatchingCompositeLineTokenizer that can be used for tokenization alone.

It is also common for a flat file to contain records that each span multiple lines. To handle this situation, a more complex strategy is required. A demonstration of this common pattern can be found in the multiLineRecords sample.

Exception Handling in Flat Files

There are many scenarios when tokenizing a line may cause exceptions to be thrown. Many flat files are imperfect and contain incorrectly formatted records. Many users choose to skip these erroneous lines while logging the issue, the original line, and the line number. These logs can later be inspected manually or by another batch job. For this reason, Spring Batch provides a hierarchy of exceptions for handling parse exceptions: FlatFileParseException and FlatFileFormatException. FlatFileParseException is thrown by the FlatFileItemReader when any errors are encountered while trying to read a file. FlatFileFormatException is thrown by implementations of the LineTokenizer interface and indicates a more specific error encountered while tokenizing.

IncorrectTokenCountException

Both DelimitedLineTokenizer and FixedLengthLineTokenizer have the ability to specify column names that can be used for creating a FieldSet. However, if the number of column names does not match the number of columns found while tokenizing a line, the FieldSet cannot be created, and an IncorrectTokenCountException is thrown, which contains the number of tokens encountered, and the number expected, as shown in the following example:

```
tokenizer.setNames(new String[] {"A", "B", "C", "D"});

try {
    tokenizer.tokenize("a,b,c");
}
catch(IncorrectTokenCountException e){
    assertEquals(4, e.getExpectedCount());
    assertEquals(3, e.getActualCount());
}
```

Because the tokenizer was configured with 4 column names but only 3 tokens were found in the file, an `IncorrectTokenCountException` was thrown.

`IncorrectLineLengthException`

Files formatted in a fixed-length format have additional requirements when parsing because, unlike a delimited format, each column must strictly adhere to its predefined width. If the total line length does not equal the widest value of this column, an exception is thrown, as shown in the following example:

```
tokenizer.setColumns(new Range[] { new Range(1, 5),
                                   new Range(6, 10),
                                   new Range(11, 15) });
try {
    tokenizer.tokenize("12345");
    fail("Expected IncorrectLineLengthException");
}
catch (IncorrectLineLengthException ex) {
    assertEquals(15, ex.getExpectedLength());
    assertEquals(5, ex.getActualLength());
}
```

The configured ranges for the tokenizer above are: 1-5, 6-10, and 11-15. Consequently, the total length of the line is 15. However, in the preceding example, a line of length 5 was passed in, causing an `IncorrectLineLengthException` to be thrown. Throwing an exception here rather than only mapping the first column allows the processing of the line to fail earlier and with more information than it would contain if it failed while trying to read in column 2 in a `FieldSetMapper`. However, there are scenarios where the length of the line is not always constant. For this reason, validation of line length can be turned off via the 'strict' property, as shown in the following example:

```
tokenizer.setColumns(new Range[] { new Range(1, 5), new Range(6, 10) });
tokenizer.setStrict(false);
FieldSet tokens = tokenizer.tokenize("12345");
assertEquals("12345", tokens.readString(0));
assertEquals("", tokens.readString(1));
```

The preceding example is almost identical to the one before it, except that `tokenizer.setStrict(false)` was called. This setting tells the tokenizer to not enforce line lengths when tokenizing the line. A `FieldSet` is now correctly created and returned. However, it contains only empty tokens for the remaining values.

1.6.3. `FlatFileItemWriter`

Writing out to flat files has the same problems and issues that reading in from a file must overcome. A step must be able to write either delimited or fixed length formats in a transactional manner.

`LineAggregator`

Just as the `LineTokenizer` interface is necessary to take an item and turn it into a `String`, file writing must have a way to aggregate multiple fields into a single string for writing to a file. In Spring Batch, this is the `LineAggregator`, shown in the following interface definition:

```
public interface LineAggregator<T> {

    public String aggregate(T item);

}
```

The `LineAggregator` is the logical opposite of `LineTokenizer`. `LineTokenizer` takes a `String` and returns a `FieldSet`, whereas `LineAggregator` takes an `item` and returns a `String`.

`PassThroughLineAggregator`

The most basic implementation of the `LineAggregator` interface is the `PassThroughLineAggregator`, which assumes that the object is already a string or that its string representation is acceptable for writing, as shown in the following code:

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

    public String aggregate(T item) {
        return item.toString();
    }
}
```

The preceding implementation is useful if direct control of creating the string is required but the advantages of a `FlatFileItemWriter`, such as transaction and restart support, are necessary.

Simplified File Writing Example

Now that the `LineAggregator` interface and its most basic implementation, `PassThroughLineAggregator`, have been defined, the basic flow of writing can be explained:

1. The object to be written is passed to the `LineAggregator` in order to obtain a `String`.
2. The returned `String` is written to the configured file.

The following excerpt from the FlatFileItemWriter expresses this in code:

```
public void write(T item) throws Exception {
    write(lineAggregator.aggregate(item) + LINE_SEPARATOR);
}
```

A simple configuration might look like the following:

XML Configuration

```
<bean id="itemWriter" class="org.spr...FlatFileItemWriter">
    <property name="resource" value="file:target/test-outputs/output.txt" />
    <property name="lineAggregator">
        <bean class="org.spr...PassThroughLineAggregator"/>
    </property>
</bean>
```

Java Configuration

```
@Bean
public FlatFileItemWriter itemWriter() {
    return  new FlatFileItemWriterBuilder<Foo>()
                    .name("itemWriter")
                    .resource(new FileSystemResource("target/test-outputs/output.txt"
))
                    .lineAggregator(new PassThroughLineAggregator<>())
                    .build();
}
```

FieldExtractor

The preceding example may be useful for the most basic uses of a writing to a file. However, most users of the FlatFileItemWriter have a domain object that needs to be written out and, thus, must be converted into a line. In file reading, the following was required:

1. Read one line from the file.
2. Pass the line into the LineTokenizer#tokenize() method, in order to retrieve a FieldSet.
3. Pass the FieldSet returned from tokenizing to a FieldSetMapper, returning the result from the ItemReader#read() method.

File writing has similar but inverse steps:

1. Pass the item to be written to the writer.
2. Convert the fields on the item into an array.
3. Aggregate the resulting array into a line.

Because there is no way for the framework to know which fields from the object need to be written out, a `FieldExtractor` must be written to accomplish the task of turning the item into an array, as shown in the following interface definition:

```
public interface FieldExtractor<T> {

    Object[] extract(T item);

}
```

Implementations of the `FieldExtractor` interface should create an array from the fields of the provided object, which can then be written out with a delimiter between the elements or as part of a fixed-width line.

`PassThroughFieldExtractor`

There are many cases where a collection, such as an array, `Collection`, or `FieldSet`, needs to be written out. "Extracting" an array from one of these collection types is very straightforward. To do so, convert the collection to an array. Therefore, the `PassThroughFieldExtractor` should be used in this scenario. It should be noted that, if the object passed in is not a type of collection, then the `PassThroughFieldExtractor` returns an array containing solely the item to be extracted.

`BeanWrapperFieldExtractor`

As with the `BeanWrapperFieldSetMapper` described in the file reading section, it is often preferable to configure how to convert a domain object to an object array, rather than writing the conversion yourself. The `BeanWrapperFieldExtractor` provides this functionality, as shown in the following example:

```
BeanWrapperFieldExtractor<Name> extractor = new BeanWrapperFieldExtractor<Name>();
extractor.setNames(new String[] { "first", "last", "born" });

String first = "Alan";
String last = "Turing";
int born = 1912;

Name n = new Name(first, last, born);
Object[] values = extractor.extract(n);

assertEquals(first, values[0]);
assertEquals(last, values[1]);
assertEquals(born, values[2]);
```

This extractor implementation has only one required property: the names of the fields to map. Just as the `BeanWrapperFieldSetMapper` needs field names to map fields on the `FieldSet` to setters on the provided object, the `BeanWrapperFieldExtractor` needs names to map to getters for creating an object array. It is worth noting that the order of the names determines the order of the fields within the array.

Delimited File Writing Example

The most basic flat file format is one in which all fields are separated by a delimiter. This can be accomplished using a `DelimitedLineAggregator`. The following example writes out a simple domain object that represents a credit to a customer account:

```
public class CustomerCredit {

    private int id;
    private String name;
    private BigDecimal credit;

    //getters and setters removed for clarity
}
```

Because a domain object is being used, an implementation of the `FieldExtractor` interface must be provided, along with the delimiter to use, as shown in the following example:

XML Configuration

```
<bean id="itemWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" ref="outputResource" />
    <property name="lineAggregator">
        <bean class="org.spr...DelimitedLineAggregator">
            <property name="delimiter" value=","/>
            <property name="fieldExtractor">
                <bean class="org.spr...BeanWrapperFieldExtractor">
                    <property name="names" value="name,credit"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

Java Configuration

```
@Bean
public FlatFileItemWriter<CustomerCredit> itemWriter(Resource outputResource) throws
Exception {
    BeanWrapperFieldExtractor<CustomerCredit> fieldExtractor = new
BeanWrapperFieldExtractor<>();
    fieldExtractor.setNames(new String[] {"name", "credit"});
    fieldExtractor.afterPropertiesSet();

    DelimitedLineAggregator<CustomerCredit> lineAggregator = new
DelimitedLineAggregator<>();
    lineAggregator.setDelimiter(",");
    lineAggregator.setFieldExtractor(fieldExtractor);

    return new FlatFileItemWriterBuilder<CustomerCredit>()
                .name("customerCreditWriter")
                .resource(outputResource)
                .lineAggregator(lineAggregator)
                .build();
}
```

In the previous example, the `BeanWrapperFieldExtractor` described earlier in this chapter is used to turn the name and credit fields within `CustomerCredit` into an object array, which is then written out with commas between each field.

It is also possible to use the `FlatFileItemWriterBuilder.DelimitedBuilder` to automatically create the `BeanWrapperFieldExtractor` and `DelimitedLineAggregator` as shown in the following example:

Java Configuration

```
@Bean
public FlatFileItemWriter<CustomerCredit> itemWriter(Resource outputResource) throws
Exception {
    return new FlatFileItemWriterBuilder<CustomerCredit>()
                .name("customerCreditWriter")
                .resource(outputResource)
                .delimited()
                .delimiter("|")
                .names(new String[] {"name", "credit"})
                .build();
}
```

Fixed Width File Writing Example

Delimited is not the only type of flat file format. Many prefer to use a set width for each column to delineate between fields, which is usually referred to as 'fixed width'. Spring Batch supports this in file writing with the `FormatterLineAggregator`. Using the same `CustomerCredit` domain object described above, it can be configured as follows:

XML Configuration

```
<bean id="itemWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" ref="outputResource" />
    <property name="lineAggregator">
        <bean class="org.spr...FormatterLineAggregator">
            <property name="fieldExtractor">
                <bean class="org.spr...BeanWrapperFieldExtractor">
                    <property name="names" value="name,credit" />
                </bean>
            </property>
            <property name="format" value="%-9s%-2.0f" />
        </bean>
    </property>
</bean>
```

Java Configuration

```
@Bean
public FlatFileItemWriter<CustomerCredit> itemWriter(Resource outputResource) throws
Exception {
    BeanWrapperFieldExtractor<CustomerCredit> fieldExtractor = new
BeanWrapperFieldExtractor<>();
    fieldExtractor.setNames(new String[] {"name", "credit"});
    fieldExtractor.afterPropertiesSet();

    FormatterLineAggregator<CustomerCredit> lineAggregator = new
FormatterLineAggregator<>();
    lineAggregator.setFormat("%-9s%-2.0f");
    lineAggregator.setFieldExtractor(fieldExtractor);

    return new FlatFileItemWriterBuilder<CustomerCredit>()
                .name("customerCreditWriter")
                .resource(outputResource)
                .lineAggregator(lineAggregator)
                .build();
}
```

Most of the preceding example should look familiar. However, the value of the format property is new and is shown in the following element:

```
<property name="format" value="%-9s%-2.0f" />
```

```
...
FormatterLineAggregator<CustomerCredit> lineAggregator = new FormatterLineAggregator<
>();
lineAggregator.setFormat("%-9s%-2.0f");
...
```

The underlying implementation is built using the same `Formatter` added as part of Java 5. The Java `Formatter` is based on the `printf` functionality of the C programming language. Most details on how to configure a formatter can be found in the Javadoc of Formatter.

It is also possible to use the `FlatFileItemWriterBuilder.FormattedBuilder` to automatically create the `BeanWrapperFieldExtractor` and `FormatterLineAggregator` as shown in following example:

Java Configuration

```
@Bean
public FlatFileItemWriter<CustomerCredit> itemWriter(Resource outputResource) throws
Exception {
    return new FlatFileItemWriterBuilder<CustomerCredit>()
                .name("customerCreditWriter")
                .resource(outputResource)
                .formatted()
                .format("%-9s%-2.0f")
                .names(new String[] {"name", "credit"})
                .build();
}
```

Handling File Creation

`FlatFileItemReader` has a very simple relationship with file resources. When the reader is initialized, it opens the file (if it exists), and throws an exception if it does not. File writing isn't quite so simple. At first glance, it seems like a similar straightforward contract should exist for `FlatFileItemWriter`: If the file already exists, throw an exception, and, if it does not, create it and start writing. However, potentially restarting a `Job` can cause issues. In normal restart scenarios, the contract is reversed: If the file exists, start writing to it from the last known good position, and, if it does not, throw an exception. However, what happens if the file name for this job is always the same? In this case, you would want to delete the file if it exists, unless it's a restart. Because of this possibility, the `FlatFileItemWriter` contains the property, `shouldDeleteIfExists`. Setting this property to true causes an existing file with the same name to be deleted when the writer is opened.

1.7. XML Item Readers and Writers

Spring Batch provides transactional infrastructure for both reading XML records and mapping them to Java objects as well as writing Java objects as XML records.

Constraints on streaming XML

The StAX API is used for I/O, as other standard XML parsing APIs do not fit batch processing requirements (DOM loads the whole input into memory at once and SAX controls the parsing process by allowing the user to provide only callbacks).

We need to consider how XML input and output works in Spring Batch. First, there are a few concepts that vary from file reading and writing but are common across Spring Batch XML processing. With XML processing, instead of lines of records (`FieldSet` instances) that need to be tokenized, it is assumed an XML resource is a collection of 'fragments' corresponding to individual records, as shown in the following image:

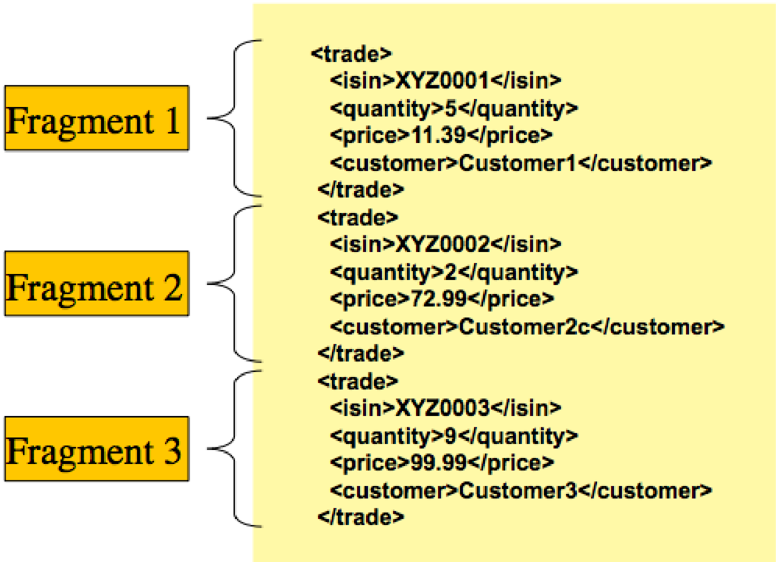

Figure 1. XML Input

The 'trade' tag is defined as the 'root element' in the scenario above. Everything between '<trade>' and '</trade>' is considered one 'fragment'. Spring Batch uses Object/XML Mapping (OXM) to bind fragments to objects. However, Spring Batch is not tied to any particular XML binding technology. Typical use is to delegate to Spring OXM, which provides uniform abstraction for the most popular OXM technologies. The dependency on Spring OXM is optional and you can choose to implement Spring Batch specific interfaces if desired. The relationship to the technologies that OXM supports is shown in the following image:

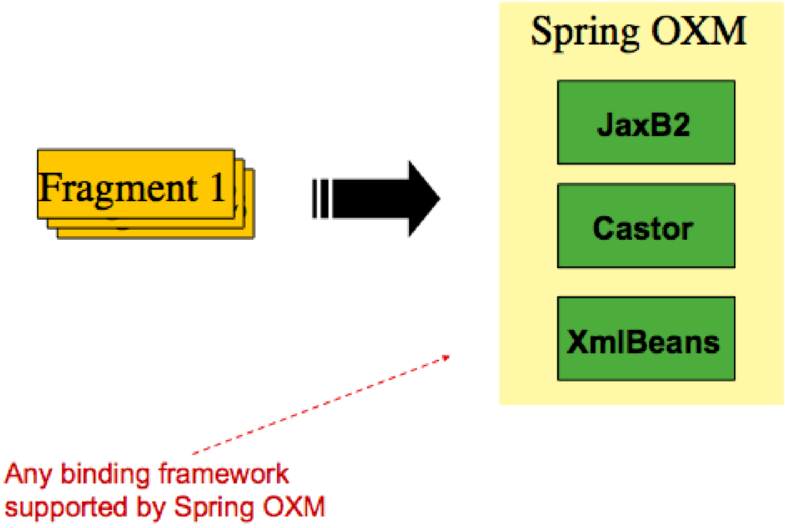

Figure 2. OXM Binding

With an introduction to OXM and how one can use XML fragments to represent records, we can now more closely examine readers and writers.

1.7.1. `StaxEventItemReader`

The `StaxEventItemReader` configuration provides a typical setup for the processing of records from an XML input stream. First, consider the following set of XML records that the `StaxEventItemReader` can process:

```
<?xml version="1.0" encoding="UTF-8"?>
<records>
    <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
        <isin>XYZ0001</isin>
        <quantity>5</quantity>
        <price>11.39</price>
        <customer>Customer1</customer>
    </trade>
    <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
        <isin>XYZ0002</isin>
        <quantity>2</quantity>
        <price>72.99</price>
        <customer>Customer2c</customer>
    </trade>
    <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
        <isin>XYZ0003</isin>
        <quantity>9</quantity>
        <price>99.99</price>
        <customer>Customer3</customer>
    </trade>
</records>
```

To be able to process the XML records, the following is needed:

- Root Element Name: The name of the root element of the fragment that constitutes the object to be mapped. The example configuration demonstrates this with the value of trade.

- Resource: A Spring Resource that represents the file to read.
- `Unmarshaller`: An unmarshalling facility provided by Spring OXM for mapping the XML fragment to an object.

The following example shows how to define a `StaxEventItemReader` that works with a root element named `trade`, a resource of `org/springframework/batch/item/xml/domain/trades.xml`, and an unmarshaller called `tradeMarshaller`.

XML Configuration

```
<bean id="itemReader" class="org.springframework.batch.item.xml.StaxEventItemReader">
    <property name="fragmentRootElementName" value="trade" />
    <property name="resource" value=
"org/springframework/batch/item/xml/domain/trades.xml" />
    <property name="unmarshaller" ref="tradeMarshaller" />
</bean>
```

Java Configuration

```
@Bean
public StaxEventItemReader itemReader() {
    return new StaxEventItemReaderBuilder<Trade>()
            .name("itemReader")
            .resource(new FileSystemResource(
"org/springframework/batch/item/xml/domain/trades.xml"))
            .addFragmentRootElements("trade")
            .unmarshaller(tradeMarshaller())
            .build();

}
```

Note that, in this example, we have chosen to use an `XStreamMarshaller`, which accepts an alias passed in as a map with the first key and value being the name of the fragment (that is, a root element) and the object type to bind. Then, similar to a `FieldSet`, the names of the other elements that map to fields within the object type are described as key/value pairs in the map. In the configuration file, we can use a Spring configuration utility to describe the required alias, as follows:

XML Configuration

```
<bean id="tradeMarshaller"
      class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="aliases">
        <util:map id="aliases">
            <entry key="trade"
                   value="org.springframework.batch.sample.domain.trade.Trade" />
            <entry key="price" value="java.math.BigDecimal" />
            <entry key="isin" value="java.lang.String" />
            <entry key="customer" value="java.lang.String" />
            <entry key="quantity" value="java.lang.Long" />
        </util:map>
    </property>
</bean>
```

Java Configuration

```
@Bean
public XStreamMarshaller tradeMarshaller() {
    Map<String, Class> aliases = new HashMap<>();
    aliases.put("trade", Trade.class);
    aliases.put("price", BigDecimal.class);
    aliases.put("isin", String.class);
    aliases.put("customer", String.class);
    aliases.put("quantity", Long.class);

    XStreamMarshaller marshaller = new XStreamMarshaller();

    marshaller.setAliases(aliases);

    return marshaller;
}
```

On input, the reader reads the XML resource until it recognizes that a new fragment is about to start. By default, the reader matches the element name to recognize that a new fragment is about to start. The reader creates a standalone XML document from the fragment and passes the document to a deserializer (typically a wrapper around a Spring OXM `Unmarshaller`) to map the XML to a Java object.

In summary, this procedure is analogous to the following Java code, which uses the injection provided by the Spring configuration:

```java
StaxEventItemReader<Trade> xmlStaxEventItemReader = new StaxEventItemReader<>();
Resource resource = new ByteArrayResource(xmlResource.getBytes());

Map aliases = new HashMap();
aliases.put("trade","org.springframework.batch.sample.domain.trade.Trade");
aliases.put("price","java.math.BigDecimal");
aliases.put("customer","java.lang.String");
aliases.put("isin","java.lang.String");
aliases.put("quantity","java.lang.Long");
XStreamMarshaller unmarshaller = new XStreamMarshaller();
unmarshaller.setAliases(aliases);
xmlStaxEventItemReader.setUnmarshaller(unmarshaller);
xmlStaxEventItemReader.setResource(resource);
xmlStaxEventItemReader.setFragmentRootElementName("trade");
xmlStaxEventItemReader.open(new ExecutionContext());

boolean hasNext = true;

Trade trade = null;

while (hasNext) {
    trade = xmlStaxEventItemReader.read();
    if (trade == null) {
        hasNext = false;
    }
    else {
        System.out.println(trade);
    }
}
```

1.7.2. StaxEventItemWriter

Output works symmetrically to input. The StaxEventItemWriter needs a Resource, a marshaller, and a rootTagName. A Java object is passed to a marshaller (typically a standard Spring OXM Marshaller) which writes to a Resource by using a custom event writer that filters the StartDocument and EndDocument events produced for each fragment by the OXM tools. The following example uses the StaxEventItemWriter:

XML Configuration

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

Java Configuration

```
@Bean
public StaxEventItemWriter itemWriter(Resource outputResource) {
    return new StaxEventItemWriterBuilder<Trade>()
            .name("tradesWriter")
            .marshaller(tradeMarshaller())
            .resource(outputResource)
            .rootTagName("trade")
            .overwriteOutput(true)
            .build();

}
```

The preceding configuration sets up the three required properties and sets the optional `overwriteOutput=true` attribute, mentioned earlier in this chapter for specifying whether an existing file can be overwritten. It should be noted the marshaller used for the writer in the following example is the exact same as the one used in the reading example from earlier in the chapter:

XML Configuration

```
<bean id="customerCreditMarshaller"
      class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="aliases">
        <util:map id="aliases">
            <entry key="customer"
                   value="org.springframework.batch.sample.domain.trade.Trade" />
            <entry key="price" value="java.math.BigDecimal" />
            <entry key="isin" value="java.lang.String" />
            <entry key="customer" value="java.lang.String" />
            <entry key="quantity" value="java.lang.Long" />
        </util:map>
    </property>
</bean>
```

Java Configuration

```java
@Bean
public XStreamMarshaller customerCreditMarshaller() {
    XStreamMarshaller marshaller = new XStreamMarshaller();

    Map<String, Class> aliases = new HashMap<>();
    aliases.put("trade", Trade.class);
    aliases.put("price", BigDecimal.class);
    aliases.put("isin", String.class);
    aliases.put("customer", String.class);
    aliases.put("quantity", Long.class);

    marshaller.setAliases(aliases);

    return marshaller;
}
```

To summarize with a Java example, the following code illustrates all of the points discussed, demonstrating the programmatic setup of the required properties:

```java
FileSystemResource resource = new FileSystemResource("data/outputFile.xml")

Map aliases = new HashMap();
aliases.put("trade","org.springframework.batch.sample.domain.trade.Trade");
aliases.put("price","java.math.BigDecimal");
aliases.put("customer","java.lang.String");
aliases.put("isin","java.lang.String");
aliases.put("quantity","java.lang.Long");
Marshaller marshaller = new XStreamMarshaller();
marshaller.setAliases(aliases);

StaxEventItemWriter staxItemWriter =
    new StaxEventItemWriterBuilder<Trade>()
                .name("tradesWriter")
                .marshaller(marshaller)
                .resource(resource)
                .rootTagName("trade")
                .overwriteOutput(true)
                .build();

staxItemWriter.afterPropertiesSet();

ExecutionContext executionContext = new ExecutionContext();
staxItemWriter.open(executionContext);
Trade trade = new Trade();
trade.setPrice(11.39);
trade.setIsin("XYZ0001");
trade.setQuantity(5L);
trade.setCustomer("Customer1");
staxItemWriter.write(trade);
```

1.8. JSON Item Readers And Writers

Spring Batch provides support for reading and Writing JSON resources in the following format:

```json
[
  {
    "isin": "123",
    "quantity": 1,
    "price": 1.2,
    "customer": "foo"
  },
  {
    "isin": "456",
    "quantity": 2,
    "price": 1.4,
    "customer": "bar"
  }
]
```

It is assumed that the JSON resource is an array of JSON objects corresponding to individual items. Spring Batch is not tied to any particular JSON library.

1.8.1. `JsonItemReader`

The `JsonItemReader` delegates JSON parsing and binding to implementations of the `org.springframework.batch.item.json.JsonObjectReader` interface. This interface is intended to be implemented by using a streaming API to read JSON objects in chunks. Two implementations are currently provided:

- Jackson through the `org.springframework.batch.item.json.JacksonJsonObjectReader`
- Gson through the `org.springframework.batch.item.json.GsonJsonObjectReader`

To be able to process JSON records, the following is needed:

- `Resource`: A Spring Resource that represents the JSON file to read.
- `JsonObjectReader`: A JSON object reader to parse and bind JSON objects to items

The following example shows how to define a `JsonItemReader` that works with the previous JSON resource `org/springframework/batch/item/json/trades.json` and a `JsonObjectReader` based on Jackson:

```java
@Bean
public JsonItemReader<Trade> jsonItemReader() {
   return new JsonItemReaderBuilder<Trade>()
                 .jsonObjectReader(new JacksonJsonObjectReader<>(Trade.class))
                 .resource(new ClassPathResource("trades.json"))
                 .name("tradeJsonItemReader")
                 .build();
}
```

1.8.2. `JsonFileItemWriter`

The `JsonFileItemWriter` delegates the marshalling of items to the `org.springframework.batch.item.json.JsonObjectMarshaller` interface. The contract of this interface is to take an object and marshall it to a JSON `String`. Two implementations are currently provided:

- Jackson through the `org.springframework.batch.item.json.JacksonJsonObjectMarshaller`
- Gson through the `org.springframework.batch.item.json.GsonJsonObjectMarshaller`

To be able to write JSON records, the following is needed:

- `Resource`: A Spring `Resource` that represents the JSON file to write
- `JsonObjectMarshaller`: A JSON object marshaller to marshall objects to JSON format

The following example shows how to define a `JsonFileItemWriter`:

```
@Bean
public JsonFileItemWriter<Trade> jsonFileItemWriter() {
   return new JsonFileItemWriterBuilder<Trade>()
                 .jsonObjectMarshaller(new JacksonJsonObjectMarshaller<>())
                 .resource(new ClassPathResource("trades.json"))
                 .name("tradeJsonFileItemWriter")
                 .build();
}
```

1.9. Multi-File Input

It is a common requirement to process multiple files within a single `Step`. Assuming the files all have the same formatting, the `MultiResourceItemReader` supports this type of input for both XML and flat file processing. Consider the following files in a directory:

```
file-1.txt  file-2.txt  ignored.txt
```

`file-1.txt` and `file-2.txt` are formatted the same and, for business reasons, should be processed together. The `MultiResourceItemReader` can be used to read in both files by using wildcards, as shown in the following example:

XML Configuration

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

Java Configuration

```
@Bean
public MultiResourceItemReader multiResourceReader() {
    return new MultiResourceItemReaderBuilder<Foo>()
                    .delegate(flatFileItemReader())
                    .resources(resources())
                    .build();
}
```

The referenced delegate is a simple `FlatFileItemReader`. The above configuration reads input from both files, handling rollback and restart scenarios. It should be noted that, as with any `ItemReader`, adding extra input (in this case a file) could cause potential issues when restarting. It is recommended that batch jobs work with their own individual directories until completed successfully.

> Input resources are ordered by using `MultiResourceItemReader#setComparator(Comparator)` to make sure resource ordering is preserved between job runs in restart scenario.

1.10. Database

Like most enterprise application styles, a database is the central storage mechanism for batch. However, batch differs from other application styles due to the sheer size of the datasets with which the system must work. If a SQL statement returns 1 million rows, the result set probably holds all returned results in memory until all rows have been read. Spring Batch provides two types of solutions for this problem:

- Cursor-based `ItemReader` Implementations
- Paging `ItemReader` Implementations

1.10.1. Cursor-based `ItemReader` Implementations

Using a database cursor is generally the default approach of most batch developers, because it is the database's solution to the problem of 'streaming' relational data. The Java `ResultSet` class is essentially an object oriented mechanism for manipulating a cursor. A `ResultSet` maintains a cursor to the current row of data. Calling `next` on a `ResultSet` moves this cursor to the next row. The Spring Batch cursor-based `ItemReader` implementation opens a cursor on initialization and moves the cursor forward one row for every call to `read`, returning a mapped object that can be used for processing. The `close` method is then called to ensure all resources are freed up. The Spring core `JdbcTemplate` gets around this problem by using the callback pattern to completely map all rows in a `ResultSet` and close before returning control back to the method caller. However, in batch, this must wait until the step is complete. The following image shows a generic diagram of how a cursor-based `ItemReader` works. Note that, while the example uses SQL (because SQL is so widely known), any technology could implement the basic approach.

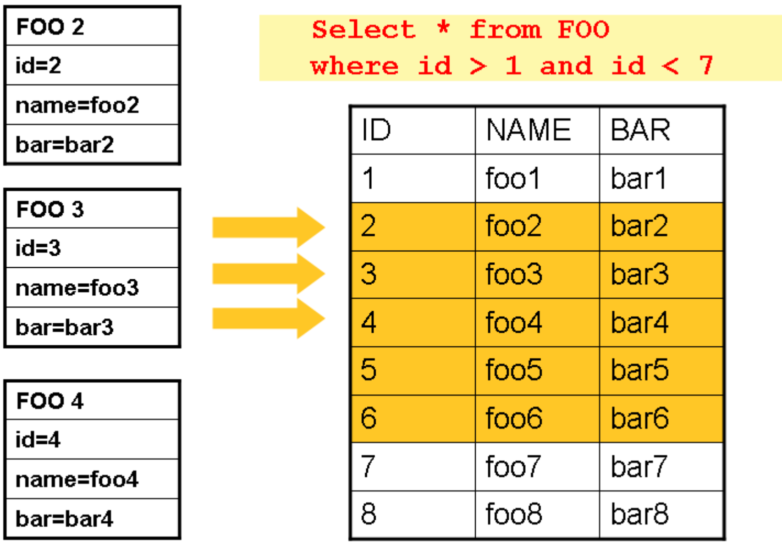

Figure 3. Cursor Example

This example illustrates the basic pattern. Given a 'FOO' table, which has three columns: `ID`, `NAME`, and `BAR`, select all rows with an ID greater than 1 but less than 7. This puts the beginning of the cursor (row 1) on ID 2. The result of this row should be a completely mapped `Foo` object. Calling `read()` again moves the cursor to the next row, which is the `Foo` with an ID of 3. The results of these reads are written out after each `read`, allowing the objects to be garbage collected (assuming no instance variables are maintaining references to them).

`JdbcCursorItemReader`

`JdbcCursorItemReader` is the JDBC implementation of the cursor-based technique. It works directly with a `ResultSet` and requires an SQL statement to run against a connection obtained from a `DataSource`. The following database schema is used as an example:

```
CREATE TABLE CUSTOMER (
   ID BIGINT IDENTITY PRIMARY KEY,
   NAME VARCHAR(45),
   CREDIT FLOAT
);
```

Many people prefer to use a domain object for each row, so the following example uses an implementation of the `RowMapper` interface to map a `CustomerCredit` object:

```
public class CustomerCreditRowMapper implements RowMapper<CustomerCredit> {

    public static final String ID_COLUMN = "id";
    public static final String NAME_COLUMN = "name";
    public static final String CREDIT_COLUMN = "credit";

    public CustomerCredit mapRow(ResultSet rs, int rowNum) throws SQLException {
        CustomerCredit customerCredit = new CustomerCredit();

        customerCredit.setId(rs.getInt(ID_COLUMN));
        customerCredit.setName(rs.getString(NAME_COLUMN));
        customerCredit.setCredit(rs.getBigDecimal(CREDIT_COLUMN));

        return customerCredit;
    }
}
```

Because `JdbcCursorItemReader` shares key interfaces with `JdbcTemplate`, it is useful to see an example of how to read in this data with `JdbcTemplate`, in order to contrast it with the `ItemReader`. For the purposes of this example, assume there are 1,000 rows in the `CUSTOMER` database. The first example uses `JdbcTemplate`:

```
//For simplicity sake, assume a dataSource has already been obtained
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
List customerCredits = jdbcTemplate.query("SELECT ID, NAME, CREDIT from CUSTOMER",
                                          new CustomerCreditRowMapper());
```

After running the preceding code snippet, the `customerCredits` list contains 1,000 `CustomerCredit` objects. In the query method, a connection is obtained from the `DataSource`, the provided SQL is run against it, and the `mapRow` method is called for each row in the `ResultSet`. Contrast this with the approach of the `JdbcCursorItemReader`, shown in the following example:

```
JdbcCursorItemReader itemReader = new JdbcCursorItemReader();
itemReader.setDataSource(dataSource);
itemReader.setSql("SELECT ID, NAME, CREDIT from CUSTOMER");
itemReader.setRowMapper(new CustomerCreditRowMapper());
int counter = 0;
ExecutionContext executionContext = new ExecutionContext();
itemReader.open(executionContext);
Object customerCredit = new Object();
while(customerCredit != null){
    customerCredit = itemReader.read();
    counter++;
}
itemReader.close();
```

After running the preceding code snippet, the counter equals 1,000. If the code above had put the returned `customerCredit` into a list, the result would have been exactly the same as with the `JdbcTemplate` example. However, the big advantage of the `ItemReader` is that it allows items to be 'streamed'. The `read` method can be called once, the item can be written out by an `ItemWriter`, and then the next item can be obtained with `read`. This allows item reading and writing to be done in 'chunks' and committed periodically, which is the essence of high performance batch processing. Furthermore, it is very easily configured for injection into a Spring Batch `Step`, as shown in the following example:

XML Configuration

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

Java Configuration

```
@Bean
public JdbcCursorItemReader<CustomerCredit> itemReader() {
    return new JdbcCursorItemReaderBuilder<CustomerCredit>()
            .dataSource(this.dataSource)
            .name("creditReader")
            .sql("select ID, NAME, CREDIT from CUSTOMER")
            .rowMapper(new CustomerCreditRowMapper())
            .build();

}
```

Additional Properties

Because there are so many varying options for opening a cursor in Java, there are many properties on the `JdbcCursorItemReader` that can be set, as described in the following table:

Table 2. JdbcCursorItemReader Properties

ignoreWarnings	Determines whether or not SQLWarnings are logged or cause an exception. The default is `true` (meaning that warnings are logged).
fetchSize	Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed by the `ResultSet` object used by the `ItemReader`. By default, no hint is given.

maxRows	Sets the limit for the maximum number of rows the underlying `ResultSet` can hold at any one time.
queryTimeout	Sets the number of seconds the driver waits for a `Statement` object to run. If the limit is exceeded, a `DataAccessException` is thrown. (Consult your driver vendor documentation for details).
verifyCursorPosition	Because the same `ResultSet` held by the `ItemReader` is passed to the `RowMapper`, it is possible for users to call `ResultSet.next()` themselves, which could cause issues with the reader's internal count. Setting this value to `true` causes an exception to be thrown if the cursor position is not the same after the `RowMapper` call as it was before.
saveState	Indicates whether or not the reader's state should be saved in the `ExecutionContext` provided by `ItemStream#update(ExecutionContext)`. The default is `true`.
driverSupportsAbsolute	Indicates whether the JDBC driver supports setting the absolute row on a `ResultSet`. It is recommended that this is set to `true` for JDBC drivers that support `ResultSet.absolute()`, as it may improve performance, especially if a step fails while working with a large data set. Defaults to `false`.
setUseSharedExtendedConnection	Indicates whether the connection used for the cursor should be used by all other processing, thus sharing the same transaction. If this is set to `false`, then the cursor is opened with its own connection and does not participate in any transactions started for the rest of the step processing. If you set this flag to `true` then you must wrap the DataSource in an `ExtendedConnectionDataSourceProxy` to prevent the connection from being closed and released after each commit. When you set this option to `true`, the statement used to open the cursor is created with both 'READ_ONLY' and 'HOLD_CURSORS_OVER_COMMIT' options. This allows holding the cursor open over transaction start and commits performed in the step processing. To use this feature, you need a database that supports this and a JDBC driver supporting JDBC 3.0 or later. Defaults to `false`.

`HibernateCursorItemReader`

Just as normal Spring users make important decisions about whether or not to use ORM solutions, which affect whether or not they use a JdbcTemplate or a HibernateTemplate, Spring Batch users have the same options. HibernateCursorItemReader is the Hibernate implementation of the cursor technique. Hibernate's usage in batch has been fairly controversial. This has largely been because Hibernate was originally developed to support online application styles. However, that does not mean it cannot be used for batch processing. The easiest approach for solving this problem is to use a StatelessSession rather than a standard session. This removes all of the caching and dirty checking Hibernate employs and that can cause issues in a batch scenario. For more information on the differences between stateless and normal hibernate sessions, refer to the documentation of your specific hibernate release. The HibernateCursorItemReader lets you declare an HQL statement and pass in a SessionFactory, which will pass back one item per call to read in the same basic fashion as the JdbcCursorItemReader. The following example configuration uses the same 'customer credit' example as the JDBC reader:

```
HibernateCursorItemReader itemReader = new HibernateCursorItemReader();
itemReader.setQueryString("from CustomerCredit");
//For simplicity sake, assume sessionFactory already obtained.
itemReader.setSessionFactory(sessionFactory);
itemReader.setUseStatelessSession(true);
int counter = 0;
ExecutionContext executionContext = new ExecutionContext();
itemReader.open(executionContext);
Object customerCredit = new Object();
while(customerCredit != null){
    customerCredit = itemReader.read();
    counter++;
}
itemReader.close();
```

This configured ItemReader returns CustomerCredit objects in the exact same manner as described by the JdbcCursorItemReader, assuming hibernate mapping files have been created correctly for the Customer table. The 'useStatelessSession' property defaults to true but has been added here to draw attention to the ability to switch it on or off. It is also worth noting that the fetch size of the underlying cursor can be set via the setFetchSize property. As with JdbcCursorItemReader, configuration is straightforward, as shown in the following example:

XML Configuration

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

Java Configuration

```
@Bean
public HibernateCursorItemReader itemReader(SessionFactory sessionFactory) {
    return new HibernateCursorItemReaderBuilder<CustomerCredit>()
            .name("creditReader")
            .sessionFactory(sessionFactory)
            .queryString("from CustomerCredit")
            .build();
}
```

`StoredProcedureItemReader`

Sometimes it is necessary to obtain the cursor data by using a stored procedure. The `StoredProcedureItemReader` works like the `JdbcCursorItemReader`, except that, instead of running a query to obtain a cursor, it runs a stored procedure that returns a cursor. The stored procedure can return the cursor in three different ways:

- As a returned `ResultSet` (used by SQL Server, Sybase, DB2, Derby, and MySQL).
- As a ref-cursor returned as an out parameter (used by Oracle and PostgreSQL).
- As the return value of a stored function call.

The following example configuration uses the same 'customer credit' example as earlier examples:

XML Configuration

```
<bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="procedureName" value="sp_customer_credit"/>
    <property name="rowMapper">
        <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper
"/>
    </property>
</bean>
```

Java Configuration

```
@Bean
public StoredProcedureItemReader reader(DataSource dataSource) {
    StoredProcedureItemReader reader = new StoredProcedureItemReader();

    reader.setDataSource(dataSource);
    reader.setProcedureName("sp_customer_credit");
    reader.setRowMapper(new CustomerCreditRowMapper());

    return reader;
}
```

The preceding example relies on the stored procedure to provide a `ResultSet` as a returned result (option 1 from earlier).

If the stored procedure returned a `ref-cursor` (option 2), then we would need to provide the position of the out parameter that is the returned `ref-cursor`. The following example shows how to work with the first parameter being a ref-cursor:

XML Configuration

```
<bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="procedureName" value="sp_customer_credit"/>
    <property name="refCursorPosition" value="1"/>
    <property name="rowMapper">
        <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper
"/>
    </property>
</bean>
```

Java Configuration

```
@Bean
public StoredProcedureItemReader reader(DataSource dataSource) {
    StoredProcedureItemReader reader = new StoredProcedureItemReader();

    reader.setDataSource(dataSource);
    reader.setProcedureName("sp_customer_credit");
    reader.setRowMapper(new CustomerCreditRowMapper());
    reader.setRefCursorPosition(1);

    return reader;
}
```

If the cursor was returned from a stored function (option 3), we would need to set the property "function" to `true`. It defaults to `false`. The following example shows what that would look like:

XML Configuration

```
<bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="procedureName" value="sp_customer_credit"/>
    <property name="function" value="true"/>
    <property name="rowMapper">
        <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper
"/>
    </property>
</bean>
```

Java Configuration

```
@Bean
public StoredProcedureItemReader reader(DataSource dataSource) {
    StoredProcedureItemReader reader = new StoredProcedureItemReader();

    reader.setDataSource(dataSource);
    reader.setProcedureName("sp_customer_credit");
    reader.setRowMapper(new CustomerCreditRowMapper());
    reader.setFunction(true);

    return reader;
}
```

In all of these cases, we need to define a `RowMapper` as well as a `DataSource` and the actual procedure name.

If the stored procedure or function takes in parameters, then they must be declared and set via the `parameters` property. The following example, for Oracle, declares three parameters. The first one is the out parameter that returns the ref-cursor, and the second and third are in parameters that takes a value of type `INTEGER`.

XML Configuration

```xml
<bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="procedureName" value="spring.cursor_func"/>
    <property name="parameters">
        <list>
            <bean class="org.springframework.jdbc.core.SqlOutParameter">
                <constructor-arg index="0" value="newid"/>
                <constructor-arg index="1">
                    <util:constant static-field="oracle.jdbc.OracleTypes.CURSOR"/>
                </constructor-arg>
            </bean>
            <bean class="org.springframework.jdbc.core.SqlParameter">
                <constructor-arg index="0" value="amount"/>
                <constructor-arg index="1">
                    <util:constant static-field="java.sql.Types.INTEGER"/>
                </constructor-arg>
            </bean>
            <bean class="org.springframework.jdbc.core.SqlParameter">
                <constructor-arg index="0" value="custid"/>
                <constructor-arg index="1">
                    <util:constant static-field="java.sql.Types.INTEGER"/>
                </constructor-arg>
            </bean>
        </list>
    </property>
    <property name="refCursorPosition" value="1"/>
    <property name="rowMapper" ref="rowMapper"/>
    <property name="preparedStatementSetter" ref="parameterSetter"/>
</bean>
```

Java Configuration

```
@Bean
public StoredProcedureItemReader reader(DataSource dataSource) {
    List<SqlParameter> parameters = new ArrayList<>();
    parameters.add(new SqlOutParameter("newId", OracleTypes.CURSOR));
    parameters.add(new SqlParameter("amount", Types.INTEGER);
    parameters.add(new SqlParameter("custId", Types.INTEGER);

    StoredProcedureItemReader reader = new StoredProcedureItemReader();

    reader.setDataSource(dataSource);
    reader.setProcedureName("spring.cursor_func");
    reader.setParameters(parameters);
    reader.setRefCursorPosition(1);
    reader.setRowMapper(rowMapper());
    reader.setPreparedStatementSetter(parameterSetter());

    return reader;
}
```

In addition to the parameter declarations, we need to specify a `PreparedStatementSetter` implementation that sets the parameter values for the call. This works the same as for the `JdbcCursorItemReader` above. All the additional properties listed in Additional Properties apply to the `StoredProcedureItemReader` as well.

1.10.2. Paging `ItemReader` Implementations

An alternative to using a database cursor is running multiple queries where each query fetches a portion of the results. We refer to this portion as a page. Each query must specify the starting row number and the number of rows that we want returned in the page.

`JdbcPagingItemReader`

One implementation of a paging `ItemReader` is the `JdbcPagingItemReader`. The `JdbcPagingItemReader` needs a `PagingQueryProvider` responsible for providing the SQL queries used to retrieve the rows making up a page. Since each database has its own strategy for providing paging support, we need to use a different `PagingQueryProvider` for each supported database type. There is also the `SqlPagingQueryProviderFactoryBean` that auto-detects the database that is being used and determine the appropriate `PagingQueryProvider` implementation. This simplifies the configuration and is the recommended best practice.

The `SqlPagingQueryProviderFactoryBean` requires that you specify a `select` clause and a `from` clause. You can also provide an optional `where` clause. These clauses and the required `sortKey` are used to build an SQL statement.

> It is important to have a unique key constraint on the `sortKey` to guarantee that no data is lost between executions.

After the reader has been opened, it passes back one item per call to `read` in the same basic fashion as any other `ItemReader`. The paging happens behind the scenes when additional rows are needed.

The following example configuration uses a similar 'customer credit' example as the cursor-based `ItemReaders` shown previously:

XML Configuration

```xml
<bean id="itemReader" class="org.spr...JdbcPagingItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="queryProvider">
        <bean class="org.spr...SqlPagingQueryProviderFactoryBean">
            <property name="selectClause" value="select id, name, credit"/>
            <property name="fromClause" value="from customer"/>
            <property name="whereClause" value="where status=:status"/>
            <property name="sortKey" value="id"/>
        </bean>
    </property>
    <property name="parameterValues">
        <map>
            <entry key="status" value="NEW"/>
        </map>
    </property>
    <property name="pageSize" value="1000"/>
    <property name="rowMapper" ref="customerMapper"/>
</bean>
```

Java Configuration

```
@Bean
public JdbcPagingItemReader itemReader(DataSource dataSource, PagingQueryProvider
queryProvider) {
    Map<String, Object> parameterValues = new HashMap<>();
    parameterValues.put("status", "NEW");

    return new JdbcPagingItemReaderBuilder<CustomerCredit>()
                        .name("creditReader")
                        .dataSource(dataSource)
                        .queryProvider(queryProvider)
                        .parameterValues(parameterValues)
                        .rowMapper(customerCreditMapper())
                        .pageSize(1000)
                        .build();
}

@Bean
public SqlPagingQueryProviderFactoryBean queryProvider() {
    SqlPagingQueryProviderFactoryBean provider = new
SqlPagingQueryProviderFactoryBean();

    provider.setSelectClause("select id, name, credit");
    provider.setFromClause("from customer");
    provider.setWhereClause("where status=:status");
    provider.setSortKey("id");

    return provider;
}
```

This configured `ItemReader` returns `CustomerCredit` objects using the `RowMapper`, which must be specified. The 'pageSize' property determines the number of entities read from the database for each query run.

The 'parameterValues' property can be used to specify a `Map` of parameter values for the query. If you use named parameters in the `where` clause, the key for each entry should match the name of the named parameter. If you use a traditional '?' placeholder, then the key for each entry should be the number of the placeholder, starting with 1.

`JpaPagingItemReader`

Another implementation of a paging `ItemReader` is the `JpaPagingItemReader`. JPA does not have a concept similar to the Hibernate `StatelessSession`, so we have to use other features provided by the JPA specification. Since JPA supports paging, this is a natural choice when it comes to using JPA for batch processing. After each page is read, the entities become detached and the persistence context is cleared, to allow the entities to be garbage collected once the page is processed.

The `JpaPagingItemReader` lets you declare a JPQL statement and pass in a `EntityManagerFactory`. It then passes back one item per call to read in the same basic fashion as any other `ItemReader`. The paging happens behind the scenes when additional entities are needed. The following example configuration uses the same 'customer credit' example as the JDBC reader shown previously:

XML Configuration

```
<bean id="itemReader" class="org.spr...JpaPagingItemReader">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
    <property name="queryString" value="select c from CustomerCredit c"/>
    <property name="pageSize" value="1000"/>
</bean>
```

Java Configuration

```
@Bean
public JpaPagingItemReader itemReader() {
    return new JpaPagingItemReaderBuilder<CustomerCredit>()
                        .name("creditReader")
                        .entityManagerFactory(entityManagerFactory())
                        .queryString("select c from CustomerCredit c")
                        .pageSize(1000)
                        .build();
}
```

This configured `ItemReader` returns `CustomerCredit` objects in the exact same manner as described for the `JdbcPagingItemReader` above, assuming the `CustomerCredit` object has the correct JPA annotations or ORM mapping file. The 'pageSize' property determines the number of entities read from the database for each query execution.

1.10.3. Database ItemWriters

While both flat files and XML files have a specific `ItemWriter` instance, there is no exact equivalent in the database world. This is because transactions provide all the needed functionality. `ItemWriter` implementations are necessary for files because they must act as if they're transactional, keeping track of written items and flushing or clearing at the appropriate times. Databases have no need for this functionality, since the write is already contained in a transaction. Users can create their own DAOs that implement the `ItemWriter` interface or use one from a custom `ItemWriter` that's written for generic processing concerns. Either way, they should work without any issues. One thing to look out for is the performance and error handling capabilities that are provided by batching the outputs. This is most common when using hibernate as an `ItemWriter` but could have the same issues when using JDBC batch mode. Batching database output does not have any inherent flaws, assuming we are careful to flush and there are no errors in the data. However, any errors while writing can cause confusion, because there is no way to know which individual item caused an exception or even if any individual item was responsible, as illustrated in the following image:

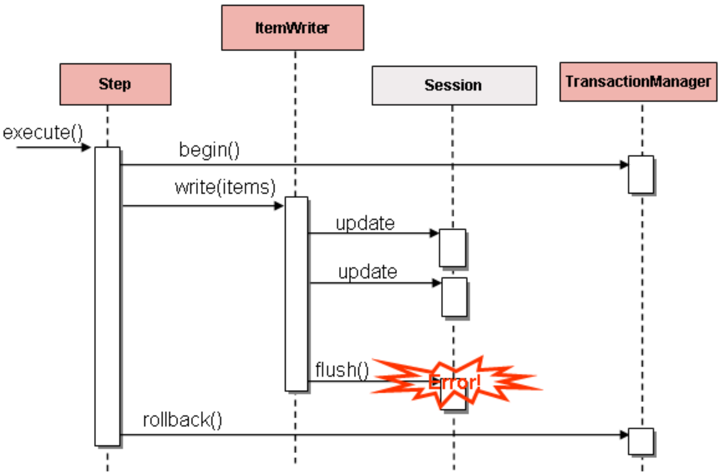

Figure 4. Error On Flush

If items are buffered before being written, any errors are not thrown until the buffer is flushed just before a commit. For example, assume that 20 items are written per chunk, and the 15th item throws a `DataIntegrityViolationException`. As far as the `Step` is concerned, all 20 item are written successfully, since there is no way to know that an error occurs until they are actually written. Once `Session#flush()` is called, the buffer is emptied and the exception is hit. At this point, there is nothing the `Step` can do. The transaction must be rolled back. Normally, this exception might cause the item to be skipped (depending upon the skip/retry policies), and then it is not written again. However, in the batched scenario, there is no way to know which item caused the issue. The whole buffer was being written when the failure happened. The only way to solve this issue is to flush after each item, as shown in the following image:

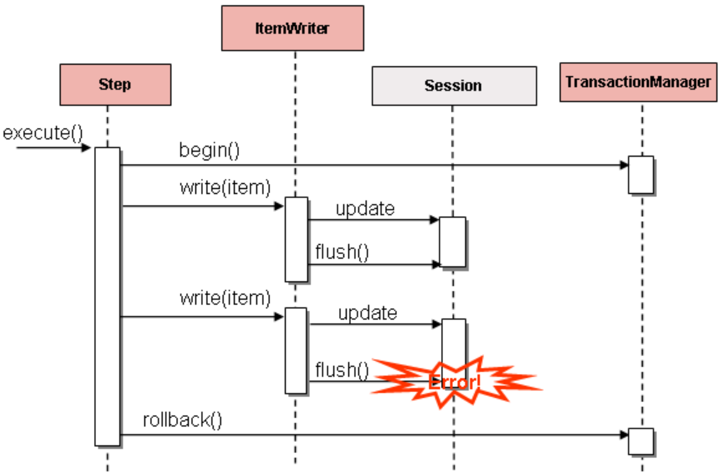

Figure 5. Error On Write

This is a common use case, especially when using Hibernate, and the simple guideline for implementations of `ItemWriter` is to flush on each call to `write()`. Doing so allows for items to be skipped reliably, with Spring Batch internally taking care of the granularity of the calls to `ItemWriter` after an error.

1.11. Reusing Existing Services

Batch systems are often used in conjunction with other application styles. The most common is an online system, but it may also support integration or even a thick client application by moving necessary bulk data that each application style uses. For this reason, it is common that many users want to reuse existing DAOs or other services within their batch jobs. The Spring container itself makes this fairly easy by allowing any necessary class to be injected. However, there may be cases where the existing service needs to act as an `ItemReader` or `ItemWriter`, either to satisfy the dependency of another Spring Batch class or because it truly is the main `ItemReader` for a step. It is fairly trivial to write an adapter class for each service that needs wrapping, but because it is such a common concern, Spring Batch provides implementations: `ItemReaderAdapter` and `ItemWriterAdapter`. Both classes implement the standard Spring method by invoking the delegate pattern and are fairly simple to set up. The following example uses the `ItemReaderAdapter`:

XML Configuration

```
<bean id="itemReader" class="org.springframework.batch.item.adapter.ItemReaderAdapter
">
    <property name="targetObject" ref="fooService" />
    <property name="targetMethod" value="generateFoo" />
</bean>

<bean id="fooService" class="org.springframework.batch.item.sample.FooService" />
```

Java Configuration

```
@Bean
public ItemReaderAdapter itemReader() {
    ItemReaderAdapter reader = new ItemReaderAdapter();

    reader.setTargetObject(fooService());
    reader.setTargetMethod("generateFoo");

    return reader;
}

@Bean
public FooService fooService() {
    return new FooService();
}
```

One important point to note is that the contract of the `targetMethod` must be the same as the contract for `read`: When exhausted, it returns `null`. Otherwise, it returns an `Object`. Anything else prevents the framework from knowing when processing should end, either causing an infinite loop or incorrect failure, depending upon the implementation of the `ItemWriter`. The following example uses the `ItemWriterAdapter`:

XML Configuration

```
<bean id="itemWriter" class="org.springframework.batch.item.adapter.ItemWriterAdapter
">
    <property name="targetObject" ref="fooService" />
    <property name="targetMethod" value="processFoo" />
</bean>

<bean id="fooService" class="org.springframework.batch.item.sample.FooService" />
```

Java Configuration

```
@Bean
public ItemWriterAdapter itemWriter() {
    ItemWriterAdapter writer = new ItemWriterAdapter();

    writer.setTargetObject(fooService());
    writer.setTargetMethod("processFoo");

    return writer;
}

@Bean
public FooService fooService() {
    return new FooService();
}
```

1.12. Validating Input

During the course of this chapter, multiple approaches to parsing input have been discussed. Each major implementation throws an exception if it is not 'well-formed'. The `FixedLengthTokenizer` throws an exception if a range of data is missing. Similarly, attempting to access an index in a `RowMapper` or `FieldSetMapper` that does not exist or is in a different format than the one expected causes an exception to be thrown. All of these types of exceptions are thrown before `read` returns. However, they do not address the issue of whether or not the returned item is valid. For example, if one of the fields is an age, it obviously cannot be negative. It may parse correctly, because it exists and is a number, but it does not cause an exception. Since there are already a plethora of validation frameworks, Spring Batch does not attempt to provide yet another. Rather, it provides a simple interface, called `Validator`, that can be implemented by any number of frameworks, as shown in the following interface definition:

```
public interface Validator<T> {

    void validate(T value) throws ValidationException;

}
```

The contract is that the `validate` method throws an exception if the object is invalid and returns normally if it is valid. Spring Batch provides an out of the box `ValidatingItemProcessor`, as shown in the following bean definition:

XML Configuration

```
<bean class="org.springframework.batch.item.validator.ValidatingItemProcessor">
    <property name="validator" ref="validator" />
</bean>

<bean id="validator" class="org.springframework.batch.item.validator.SpringValidator">
    <property name="validator">
        <bean class=
"org.springframework.batch.sample.domain.trade.internal.validator.TradeValidator"/>
    </property>
</bean>
```

Java Configuration

```
@Bean
public ValidatingItemProcessor itemProcessor() {
    ValidatingItemProcessor processor = new ValidatingItemProcessor();

    processor.setValidator(validator());

    return processor;
}

@Bean
public SpringValidator validator() {
    SpringValidator validator = new SpringValidator();

    validator.setValidator(new TradeValidator());

    return validator;
}
```

You can also use the `BeanValidatingItemProcessor` to validate items annotated with the Bean Validation API (JSR-303) annotations. For example, given the following type `Person`:

```
class Person {

    @NotEmpty
    private String name;

    public Person(String name) {
     this.name = name;
    }

    public String getName() {
     return name;
    }

    public void setName(String name) {
     this.name = name;
    }

}
```

you can validate items by declaring a `BeanValidatingItemProcessor` bean in your application context and register it as a processor in your chunk-oriented step:

```
@Bean
public BeanValidatingItemProcessor<Person> beanValidatingItemProcessor() throws
Exception {
    BeanValidatingItemProcessor<Person> beanValidatingItemProcessor = new
BeanValidatingItemProcessor<>();
    beanValidatingItemProcessor.setFilter(true);

    return beanValidatingItemProcessor;
}
```

1.13. Preventing State Persistence

By default, all of the `ItemReader` and `ItemWriter` implementations store their current state in the `ExecutionContext` before it is committed. However, this may not always be the desired behavior. For example, many developers choose to make their database readers 'rerunnable' by using a process indicator. An extra column is added to the input data to indicate whether or not it has been processed. When a particular record is being read (or written) the processed flag is flipped from `false` to `true`. The SQL statement can then contain an extra statement in the `where` clause, such as `where PROCESSED_IND = false`, thereby ensuring that only unprocessed records are returned in the case of a restart. In this scenario, it is preferable to not store any state, such as the current row number, since it is irrelevant upon restart. For this reason, all readers and writers include the 'saveState' property, as shown in the following example:

XML Configuration

```
<bean id="playerSummarizationSource" class="org.spr...JdbcCursorItemReader">
    <property name="dataSource" ref="dataSource" />
    <property name="rowMapper">
        <bean class="org.springframework.batch.sample.PlayerSummaryMapper" />
    </property>
    <property name="saveState" value="false" />
    <property name="sql">
        <value>
            SELECT games.player_id, games.year_no, SUM(COMPLETES),
            SUM(ATTEMPTS), SUM(PASSING_YARDS), SUM(PASSING_TD),
            SUM(INTERCEPTIONS), SUM(RUSHES), SUM(RUSH_YARDS),
            SUM(RECEPTIONS), SUM(RECEPTIONS_YARDS), SUM(TOTAL_TD)
            from games, players where players.player_id =
            games.player_id group by games.player_id, games.year_no
        </value>
    </property>
</bean>
```

Java Configuration

```
@Bean
public JdbcCursorItemReader playerSummarizationSource(DataSource dataSource) {
    return new JdbcCursorItemReaderBuilder<PlayerSummary>()
                .dataSource(dataSource)
                .rowMapper(new PlayerSummaryMapper())
                .saveState(false)
                .sql("SELECT games.player_id, games.year_no, SUM(COMPLETES),"
                  + "SUM(ATTEMPTS), SUM(PASSING_YARDS), SUM(PASSING_TD),"
                  + "SUM(INTERCEPTIONS), SUM(RUSHES), SUM(RUSH_YARDS),"
                  + "SUM(RECEPTIONS), SUM(RECEPTIONS_YARDS), SUM(TOTAL_TD)"
                  + "from games, players where players.player_id ="
                  + "games.player_id group by games.player_id, games.year_no")
                .build();

}
```

The `ItemReader` configured above does not make any entries in the `ExecutionContext` for any executions in which it participates.

1.14. Creating Custom ItemReaders and ItemWriters

So far, this chapter has discussed the basic contracts of reading and writing in Spring Batch and some common implementations for doing so. However, these are all fairly generic, and there are many potential scenarios that may not be covered by out-of-the-box implementations. This section shows, by using a simple example, how to create a custom `ItemReader` and `ItemWriter` implementation and implement their contracts correctly. The `ItemReader` also implements `ItemStream`, in order to illustrate how to make a reader or writer restartable.

1.14.1. Custom `ItemReader` Example

For the purpose of this example, we create a simple `ItemReader` implementation that reads from a provided list. We start by implementing the most basic contract of `ItemReader`, the `read` method, as shown in the following code:

```
public class CustomItemReader<T> implements ItemReader<T>{

    List<T> items;

    public CustomItemReader(List<T> items) {
        this.items = items;
    }

    public T read() throws Exception, UnexpectedInputException,
       NonTransientResourceException, ParseException {

        if (!items.isEmpty()) {
            return items.remove(0);
        }
        return null;
    }
}
```

The preceding class takes a list of items and returns them one at a time, removing each from the list. When the list is empty, it returns `null`, thus satisfying the most basic requirements of an `ItemReader`, as illustrated in the following test code:

```
List<String> items = new ArrayList<String>();
items.add("1");
items.add("2");
items.add("3");

ItemReader itemReader = new CustomItemReader<String>(items);
assertEquals("1", itemReader.read());
assertEquals("2", itemReader.read());
assertEquals("3", itemReader.read());
assertNull(itemReader.read());
```

Making the `ItemReader` Restartable

The final challenge is to make the `ItemReader` restartable. Currently, if processing is interrupted and begins again, the `ItemReader` must start at the beginning. This is actually valid in many scenarios, but it is sometimes preferable that a batch job restarts where it left off. The key discriminant is often whether the reader is stateful or stateless. A stateless reader does not need to worry about restartability, but a stateful one has to try to reconstitute its last known state on restart. For this reason, we recommend that you keep custom readers stateless if possible, so you need not worry about restartability.

If you do need to store state, then the `ItemStream` interface should be used:

```
public class CustomItemReader<T> implements ItemReader<T>, ItemStream {

    List<T> items;
    int currentIndex = 0;
    private static final String CURRENT_INDEX = "current.index";

    public CustomItemReader(List<T> items) {
        this.items = items;
    }

    public T read() throws Exception, UnexpectedInputException,
        ParseException, NonTransientResourceException {

        if (currentIndex < items.size()) {
            return items.get(currentIndex++);
        }

        return null;
    }

    public void open(ExecutionContext executionContext) throws ItemStreamException {
        if(executionContext.containsKey(CURRENT_INDEX)){
            currentIndex = new Long(executionContext.getLong(CURRENT_INDEX)).intValue
();
        }
        else{
            currentIndex = 0;
        }
    }

    public void update(ExecutionContext executionContext) throws ItemStreamException {
        executionContext.putLong(CURRENT_INDEX, new Long(currentIndex).longValue());
    }

    public void close() throws ItemStreamException {}
}
```

On each call to the `ItemStream` `update` method, the current index of the `ItemReader` is stored in the provided `ExecutionContext` with a key of 'current.index'. When the `ItemStream` `open` method is called, the `ExecutionContext` is checked to see if it contains an entry with that key. If the key is found, then the current index is moved to that location. This is a fairly trivial example, but it still meets the general contract:

```java
ExecutionContext executionContext = new ExecutionContext();
((ItemStream)itemReader).open(executionContext);
assertEquals("1", itemReader.read());
((ItemStream)itemReader).update(executionContext);

List<String> items = new ArrayList<String>();
items.add("1");
items.add("2");
items.add("3");
itemReader = new CustomItemReader<String>(items);

((ItemStream)itemReader).open(executionContext);
assertEquals("2", itemReader.read());
```

Most `ItemReaders` have much more sophisticated restart logic. The `JdbcCursorItemReader`, for example, stores the row ID of the last processed row in the cursor.

It is also worth noting that the key used within the `ExecutionContext` should not be trivial. That is because the same `ExecutionContext` is used for all `ItemStreams` within a `Step`. In most cases, simply prepending the key with the class name should be enough to guarantee uniqueness. However, in the rare cases where two of the same type of `ItemStream` are used in the same step (which can happen if two files are needed for output), a more unique name is needed. For this reason, many of the Spring Batch `ItemReader` and `ItemWriter` implementations have a `setName()` property that lets this key name be overridden.

1.14.2. Custom `ItemWriter` Example

Implementing a Custom `ItemWriter` is similar in many ways to the `ItemReader` example above but differs in enough ways as to warrant its own example. However, adding restartability is essentially the same, so it is not covered in this example. As with the `ItemReader` example, a `List` is used in order to keep the example as simple as possible:

```java
public class CustomItemWriter<T> implements ItemWriter<T> {

    List<T> output = TransactionAwareProxyFactory.createTransactionalList();

    public void write(List<? extends T> items) throws Exception {
        output.addAll(items);
    }

    public List<T> getOutput() {
        return output;
    }
}
```

Making the `ItemWriter` Restartable

To make the `ItemWriter` restartable, we would follow the same process as for the `ItemReader`, adding and implementing the `ItemStream` interface to synchronize the execution context. In the example, we might have to count the number of items processed and add that as a footer record. If we needed to do that, we could implement `ItemStream` in our `ItemWriter` so that the counter was reconstituted from the execution context if the stream was re-opened.

In many realistic cases, custom `ItemWriters` also delegate to another writer that itself is restartable (for example, when writing to a file), or else it writes to a transactional resource and so does not need to be restartable, because it is stateless. When you have a stateful writer you should probably be sure to implement `ItemStream` as well as `ItemWriter`. Remember also that the client of the writer needs to be aware of the `ItemStream`, so you may need to register it as a stream in the configuration.

1.15. Item Reader and Writer Implementations

In this section, we will introduce you to readers and writers that have not already been discussed in the previous sections.

1.15.1. Decorators

In some cases, a user needs specialized behavior to be appended to a pre-existing `ItemReader`. Spring Batch offers some out of the box decorators that can add additional behavior to to your `ItemReader` and `ItemWriter` implementations.

Spring Batch includes the following decorators:

- `SynchronizedItemStreamReader`
- `SingleItemPeekableItemReader`
- `MultiResourceItemWriter`
- `ClassifierCompositeItemWriter`
- `ClassifierCompositeItemProcessor`

`SynchronizedItemStreamReader`

When using an `ItemReader` that is not thread safe, Spring Batch offers the `SynchronizedItemStreamReader` decorator, which can be used to make the `ItemReader` thread safe. Spring Batch provides a `SynchronizedItemStreamReaderBuilder` to construct an instance of the `SynchronizedItemStreamReader`.

`SingleItemPeekableItemReader`

Spring Batch includes a decorator that adds a peek method to an `ItemReader`. This peek method lets the user peek one item ahead. Repeated calls to the peek returns the same item, and this is the next item returned from the `read` method. Spring Batch provides a `SingleItemPeekableItemReaderBuilder` to construct an instance of the `SingleItemPeekableItemReader`.

> SingleItemPeekableItemReader's peek method is not thread-safe, because it would not be possible to honor the peek in multiple threads. Only one of the threads that peeked would get that item in the next call to read.

`MultiResourceItemWriter`

The `MultiResourceItemWriter` wraps a `ResourceAwareItemWriterItemStream` and creates a new output resource when the count of items written in the current resource exceeds the `itemCountLimitPerResource`. Spring Batch provides a `MultiResourceItemWriterBuilder` to construct an instance of the `MultiResourceItemWriter`.

`ClassifierCompositeItemWriter`

The `ClassifierCompositeItemWriter` calls one of a collection of `ItemWriter` implementations for each item, based on a router pattern implemented through the provided `Classifier`. The implementation is thread-safe if all delegates are thread-safe. Spring Batch provides a `ClassifierCompositeItemWriterBuilder` to construct an instance of the `ClassifierCompositeItemWriter`.

`ClassifierCompositeItemProcessor`

The `ClassifierCompositeItemProcessor` is an `ItemProcessor` that calls one of a collection of `ItemProcessor` implementations, based on a router pattern implemented through the provided `Classifier`. Spring Batch provides a `ClassifierCompositeItemProcessorBuilder` to construct an instance of the `ClassifierCompositeItemProcessor`.

1.15.2. Messaging Readers And Writers

Spring Batch offers the following readers and writers for commonly used messaging systems:

- `AmqpItemReader`
- `AmqpItemWriter`
- `JmsItemReader`
- `JmsItemWriter`

`AmqpItemReader`

The `AmqpItemReader` is an `ItemReader` that uses an `AmqpTemplate` to receive or convert messages from an exchange. Spring Batch provides a `AmqpItemReaderBuilder` to construct an instance of the `AmqpItemReader`.

`AmqpItemWriter`

The `AmqpItemWriter` is an `ItemWriter` that uses an `AmqpTemplate` to send messages to an AMQP exchange. Messages are sent to the nameless exchange if the name not specified in the provided `AmqpTemplate`. Spring Batch provides an `AmqpItemWriterBuilder` to construct an instance of the `AmqpItemWriter`.

`JmsItemReader`

The `JmsItemReader` is an `ItemReader` for JMS that uses a `JmsTemplate`. The template should have a default destination, which is used to provide items for the `read()` method. Spring Batch provides a `JmsItemReaderBuilder` to construct an instance of the `JmsItemReader`.

`JmsItemWriter`

The `JmsItemWriter` is an `ItemWriter` for JMS that uses a `JmsTemplate`. The template should have a default destination, which is used to send items in `write(List)`. Spring Batch provides a `JmsItemWriterBuilder` to construct an instance of the `JmsItemWriter`.

1.15.3. Database Readers

Spring Batch offers the following database readers:

- `Neo4jItemReader`
- `MongoItemReader`
- `HibernateCursorItemReader`
- `HibernatePagingItemReader`
- `RepositoryItemReader`

`Neo4jItemReader`

The `Neo4jItemReader` is an `ItemReader` that reads objects from the graph database Neo4j by using a paging technique. Spring Batch provides a `Neo4jItemReaderBuilder` to construct an instance of the `Neo4jItemReader`.

`MongoItemReader`

The `MongoItemReader` is an `ItemReader` that reads documents from MongoDB by using a paging technique. Spring Batch provides a `MongoItemReaderBuilder` to construct an instance of the `MongoItemReader`.

`HibernateCursorItemReader`

The `HibernateCursorItemReader` is an `ItemStreamReader` for reading database records built on top of Hibernate. It executes the HQL query and then, when initialized, iterates over the result set as the `read()` method is called, successively returning an object corresponding to the current row. Spring Batch provides a `HibernateCursorItemReaderBuilder` to construct an instance of the `HibernateCursorItemReader`.

`HibernatePagingItemReader`

The `HibernatePagingItemReader` is an `ItemReader` for reading database records built on top of Hibernate and reading only up to a fixed number of items at a time. Spring Batch provides a `HibernatePagingItemReaderBuilder` to construct an instance of the `HibernatePagingItemReader`.

`RepositoryItemReader`

The `RepositoryItemReader` is an `ItemReader` that reads records by using a `PagingAndSortingRepository`. Spring Batch provides a `RepositoryItemReaderBuilder` to construct an instance of the

RepositoryItemReader.

1.15.4. Database Writers

Spring Batch offers the following database writers:

- Neo4jItemWriter
- MongoItemWriter
- RepositoryItemWriter
- HibernateItemWriter
- JdbcBatchItemWriter
- JpaItemWriter
- GemfireItemWriter

Neo4jItemWriter

The Neo4jItemWriter is an ItemWriter implementation that writes to a Neo4j database. Spring Batch provides a Neo4jItemWriterBuilder to construct an instance of the Neo4jItemWriter.

MongoItemWriter

The MongoItemWriter is an ItemWriter implementation that writes to a MongoDB store using an implementation of Spring Data's MongoOperations. Spring Batch provides a MongoItemWriterBuilder to construct an instance of the MongoItemWriter.

RepositoryItemWriter

The RepositoryItemWriter is an ItemWriter wrapper for a CrudRepository from Spring Data. Spring Batch provides a RepositoryItemWriterBuilder to construct an instance of the RepositoryItemWriter.

HibernateItemWriter

The HibernateItemWriter is an ItemWriter that uses a Hibernate session to save or update entities that are not part of the current Hibernate session. Spring Batch provides a HibernateItemWriterBuilder to construct an instance of the HibernateItemWriter.

JdbcBatchItemWriter

The JdbcBatchItemWriter is an ItemWriter that uses the batching features from NamedParameterJdbcTemplate to execute a batch of statements for all items provided. Spring Batch provides a JdbcBatchItemWriterBuilder to construct an instance of the JdbcBatchItemWriter.

JpaItemWriter

The JpaItemWriter is an ItemWriter that uses a JPA EntityManagerFactory to merge any entities that are not part of the persistence context. Spring Batch provides a JpaItemWriterBuilder to construct an instance of the JpaItemWriter.

GemfireItemWriter

The GemfireItemWriter is an ItemWriter that uses a GemfireTemplate that stores items in GemFire as key/value pairs. Spring Batch provides a `GemfireItemWriterBuilder` to construct an instance of the `GemfireItemWriter`.

1.15.5. Specialized Readers

Spring Batch offers the following specialized readers:

- `LdifReader`
- `MappingLdifReader`

`LdifReader`

The `LdifReader` reads LDIF (LDAP Data Interchange Format) records from a `Resource`, parses them, and returns a `LdapAttribute` object for each `read` executed. Spring Batch provides a `LdifReaderBuilder` to construct an instance of the `LdifReader`.

`MappingLdifReader`

The `MappingLdifReader` reads LDIF (LDAP Data Interchange Format) records from a `Resource`, parses them then maps each LDIF record to a POJO (Plain Old Java Object). Each read returns a POJO. Spring Batch provides a `MappingLdifReaderBuilder` to construct an instance of the `MappingLdifReader`.

1.15.6. Specialized Writers

Spring Batch offers the following specialized writers:

- `SimpleMailMessageItemWriter`

`SimpleMailMessageItemWriter`

The `SimpleMailMessageItemWriter` is an `ItemWriter` that can send mail messages. It delegates the actual sending of messages to an instance of `MailSender`. Spring Batch provides a `SimpleMailMessageItemWriterBuilder` to construct an instance of the `SimpleMailMessageItemWriter`.

1.15.7. Specialized Processors

Spring Batch offers the following specialized processors:

- `ScriptItemProcessor`

`ScriptItemProcessor`

The `ScriptItemProcessor` is an `ItemProcessor` that passes the current item to process to the provided script and the result of the script is returned by the processor. Spring Batch provides a `ScriptItemProcessorBuilder` to construct an instance of the `ScriptItemProcessor`.