Table of Contents

Chapter 1. Common Batch Patterns

Some batch jobs can be assembled purely from off-the-shelf components in Spring Batch. For instance, the `ItemReader` and `ItemWriter` implementations can be configured to cover a wide range of scenarios. However, for the majority of cases, custom code must be written. The main API entry points for application developers are the `Tasklet`, the `ItemReader`, the `ItemWriter`, and the various listener interfaces. Most simple batch jobs can use off-the-shelf input from a Spring Batch `ItemReader`, but it is often the case that there are custom concerns in the processing and writing that require developers to implement an `ItemWriter` or `ItemProcessor`.

In this chapter, we provide a few examples of common patterns in custom business logic. These examples primarily feature the listener interfaces. It should be noted that an `ItemReader` or `ItemWriter` can implement a listener interface as well, if appropriate.

1.1. Logging Item Processing and Failures

A common use case is the need for special handling of errors in a step, item by item, perhaps logging to a special channel or inserting a record into a database. A chunk-oriented `Step` (created from the step factory beans) lets users implement this use case with a simple `ItemReadListener` for errors on `read` and an `ItemWriteListener` for errors on `write`. The following code snippet illustrates a listener that logs both read and write failures:

```
public class ItemFailureLoggerListener extends ItemListenerSupport {

    private static Log logger = LogFactory.getLog("item.error");

    public void onReadError(Exception ex) {
        logger.error("Encountered error on read", e);
    }

    public void onWriteError(Exception ex, List<? extends Object> items) {
        logger.error("Encountered error on write", ex);
    }
}
```

Having implemented this listener, it must be registered with a step, as shown in the following example:

XML Configuration

```
<step id="simpleStep">
...
<listeners>
    <listener>
        <bean class="org.example...ItemFailureLoggerListener"/>
    </listener>
</listeners>
</step>
```

Java Configuration

```
@Bean
public Step simpleStep() {
    return this.stepBuilderFactory.get("simpleStep")
                ...
                .listener(new ItemFailureLoggerListener())
                .build();
}
```

if your listener does anything in an `onError()` method, it must be inside a transaction that is going to be rolled back. If you need to use a transactional resource, such as a database, inside an `onError()` method, consider adding a declarative transaction to that method (see Spring Core Reference Guide for details), and giving its propagation attribute a value of `REQUIRES_NEW`.

1.2. Stopping a Job Manually for Business Reasons

Spring Batch provides a `stop()` method through the `JobLauncher` interface, but this is really for use by the operator rather than the application programmer. Sometimes, it is more convenient or makes more sense to stop a job execution from within the business logic.

The simplest thing to do is to throw a `RuntimeException` (one that is neither retried indefinitely nor skipped). For example, a custom exception type could be used, as shown in the following example:

```
public class PoisonPillItemProcessor<T> implements ItemProcessor<T, T> {

    @Override
    public T process(T item) throws Exception {
        if (isPoisonPill(item)) {
            throw new PoisonPillException("Poison pill detected: " + item);
        }
        return item;
    }
}
```

Another simple way to stop a step from executing is to return `null` from the `ItemReader`, as shown in the following example:

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

    private ItemReader<T> delegate;

    public void setDelegate(ItemReader<T> delegate) { ... }

    public T read() throws Exception {
        T item = delegate.read();
        if (isEndItem(item)) {
            return null; // end the step here
        }
        return item;
    }

}
```

The previous example actually relies on the fact that there is a default implementation of the `CompletionPolicy` strategy that signals a complete batch when the item to be processed is `null`. A more sophisticated completion policy could be implemented and injected into the `Step` through the `SimpleStepFactoryBean`, as shown in the following example:

XML Configuration

```
<step id="simpleStep">
    <tasklet>
        <chunk reader="reader" writer="writer" commit-interval="10"
               chunk-completion-policy="completionPolicy"/>
    </tasklet>
</step>

<bean id="completionPolicy" class="org.example...SpecialCompletionPolicy"/>
```

Java Configuration

```
@Bean
public Step simpleStep() {
    return this.stepBuilderFactory.get("simpleStep")
                .<String, String>chunk(new SpecialCompletionPolicy())
                .reader(reader())
                .writer(writer())
                .build();
}
```

An alternative is to set a flag in the `StepExecution`, which is checked by the `Step` implementations in the framework in between item processing. To implement this alternative, we need access to the

current StepExecution, and this can be achieved by implementing a StepListener and registering it with the Step. The following example shows a listener that sets the flag:

```
public class CustomItemWriter extends ItemListenerSupport implements StepListener {

    private StepExecution stepExecution;

    public void beforeStep(StepExecution stepExecution) {
        this.stepExecution = stepExecution;
    }

    public void afterRead(Object item) {
        if (isPoisonPill(item)) {
            stepExecution.setTerminateOnly(true);
       }
    }

}
```

When the flag is set, the default behavior is for the step to throw a JobInterruptedException. This behavior can be controlled through the StepInterruptionPolicy. However, the only choice is to throw or not throw an exception, so this is always an abnormal ending to a job.

1.3. Adding a Footer Record

Often, when writing to flat files, a "footer" record must be appended to the end of the file, after all processing has be completed. This can be achieved using the FlatFileFooterCallback interface provided by Spring Batch. The FlatFileFooterCallback (and its counterpart, the FlatFileHeaderCallback) are optional properties of the FlatFileItemWriter and can be added to an item writer as shown in the following example:

XML Configuration

```
<bean id="itemWriter" class="org.spr...FlatFileItemWriter">
    <property name="resource" ref="outputResource" />
    <property name="lineAggregator" ref="lineAggregator"/>
    <property name="headerCallback" ref="headerCallback" />
    <property name="footerCallback" ref="footerCallback" />
</bean>
```

Java Configuration

```
@Bean
public FlatFileItemWriter<String> itemWriter(Resource outputResource) {
    return new FlatFileItemWriterBuilder<String>()
            .name("itemWriter")
            .resource(outputResource)
            .lineAggregator(lineAggregator())
            .headerCallback(headerCallback())
            .footerCallback(footerCallback())
            .build();
}
```

The footer callback interface has just one method that is called when the footer must be written, as shown in the following interface definition:

```
public interface FlatFileFooterCallback {

    void writeFooter(Writer writer) throws IOException;

}
```

1.3.1. Writing a Summary Footer

A common requirement involving footer records is to aggregate information during the output process and to append this information to the end of the file. This footer often serves as a summarization of the file or provides a checksum.

For example, if a batch job is writing `Trade` records to a flat file, and there is a requirement that the total amount from all the `Trades` is placed in a footer, then the following `ItemWriter` implementation can be used:

```
public class TradeItemWriter implements ItemWriter<Trade>,
                                        FlatFileFooterCallback {

    private ItemWriter<Trade> delegate;

    private BigDecimal totalAmount = BigDecimal.ZERO;

    public void write(List<? extends Trade> items) throws Exception {
        BigDecimal chunkTotal = BigDecimal.ZERO;
        for (Trade trade : items) {
            chunkTotal = chunkTotal.add(trade.getAmount());
        }

        delegate.write(items);

        // After successfully writing all items
        totalAmount = totalAmount.add(chunkTotal);
    }

    public void writeFooter(Writer writer) throws IOException {
        writer.write("Total Amount Processed: " + totalAmount);
    }

    public void setDelegate(ItemWriter delegate) {...}
}
```

This `TradeItemWriter` stores a `totalAmount` value that is increased with the `amount` from each `Trade` item written. After the last `Trade` is processed, the framework calls `writeFooter`, which puts the `totalAmount` into the file. Note that the `write` method makes use of a temporary variable, `chunkTotal`, that stores the total of the `Trade` amounts in the chunk. This is done to ensure that, if a skip occurs in the `write` method, the `totalAmount` is left unchanged. It is only at the end of the `write` method, once we are guaranteed that no exceptions are thrown, that we update the `totalAmount`.

In order for the `writeFooter` method to be called, the `TradeItemWriter` (which implements `FlatFileFooterCallback`) must be wired into the `FlatFileItemWriter` as the `footerCallback`. The following example shows how to do so:

XML Configuration

```
<bean id="tradeItemWriter" class="..TradeItemWriter">
    <property name="delegate" ref="flatFileItemWriter" />
</bean>

<bean id="flatFileItemWriter" class="org.spr...FlatFileItemWriter">
   <property name="resource" ref="outputResource" />
   <property name="lineAggregator" ref="lineAggregator"/>
   <property name="footerCallback" ref="tradeItemWriter" />
</bean>
```

Java Configuration

```
@Bean
public TradeItemWriter tradeItemWriter() {
    TradeItemWriter itemWriter = new TradeItemWriter();

    itemWriter.setDelegate(flatFileItemWriter(null));

    return itemWriter;
}

@Bean
public FlatFileItemWriter<String> flatFileItemWriter(Resource outputResource) {
    return new FlatFileItemWriterBuilder<String>()
            .name("itemWriter")
            .resource(outputResource)
            .lineAggregator(lineAggregator())
            .footerCallback(tradeItemWriter())
            .build();
}
```

The way that the `TradeItemWriter` has been written so far functions correctly only if the `Step` is not restartable. This is because the class is stateful (since it stores the `totalAmount`), but the `totalAmount` is not persisted to the database. Therefore, it cannot be retrieved in the event of a restart. In order to make this class restartable, the `ItemStream` interface should be implemented along with the methods `open` and `update`, as shown in the following example:

```
public void open(ExecutionContext executionContext) {
    if (executionContext.containsKey("total.amount") {
        totalAmount = (BigDecimal) executionContext.get("total.amount");
    }
}

public void update(ExecutionContext executionContext) {
    executionContext.put("total.amount", totalAmount);
}
```

The update method stores the most current version of `totalAmount` to the `ExecutionContext` just before that object is persisted to the database. The open method retrieves any existing `totalAmount` from the `ExecutionContext` and uses it as the starting point for processing, allowing the `TradeItemWriter` to pick up on restart where it left off the previous time the `Step` was run.

1.4. Driving Query Based ItemReaders

In the chapter on readers and writers, database input using paging was discussed. Many database vendors, such as DB2, have extremely pessimistic locking strategies that can cause issues if the table being read also needs to be used by other portions of the online application. Furthermore, opening cursors over extremely large datasets can cause issues on databases from certain vendors.

Therefore, many projects prefer to use a 'Driving Query' approach to reading in data. This approach works by iterating over keys, rather than the entire object that needs to be returned, as the following image illustrates:

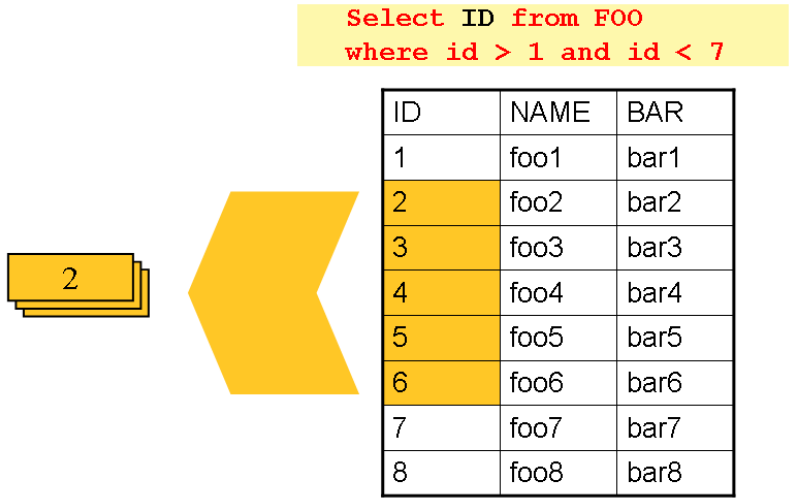

Figure 1. Driving Query Job

As you can see, the example shown in the preceding image uses the same 'FOO' table as was used in the cursor-based example. However, rather than selecting the entire row, only the IDs were selected in the SQL statement. So, rather than a `FOO` object being returned from `read`, an `Integer` is returned. This number can then be used to query for the 'details', which is a complete `Foo` object, as shown in the following image:

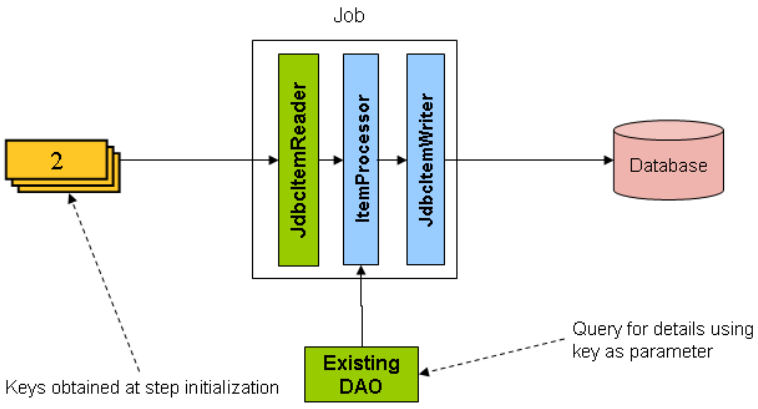

Figure 2. Driving Query Example

An `ItemProcessor` should be used to transform the key obtained from the driving query into a full 'Foo' object. An existing DAO can be used to query for the full object based on the key.

1.5. Multi-Line Records

While it is usually the case with flat files that each record is confined to a single line, it is common that a file might have records spanning multiple lines with multiple formats. The following excerpt

from a file shows an example of such an arrangement:

```
HEA;0013100345;2007-02-15
NCU;Smith;Peter;;T;20014539;F
BAD;;Oak Street 31/A;;Small Town;00235;IL;US
FOT;2;2;267.34
```

Everything between the line starting with 'HEA' and the line starting with 'FOT' is considered one record. There are a few considerations that must be made in order to handle this situation correctly:

- Instead of reading one record at a time, the `ItemReader` must read every line of the multi-line record as a group, so that it can be passed to the `ItemWriter` intact.
- Each line type may need to be tokenized differently.

Because a single record spans multiple lines and because we may not know how many lines there are, the `ItemReader` must be careful to always read an entire record. In order to do this, a custom `ItemReader` should be implemented as a wrapper for the `FlatFileItemReader`, as shown in the following example:

XML Configuration

```
<bean id="itemReader" class="org.spr...MultiLineTradeItemReader">
    <property name="delegate">
        <bean class="org.springframework.batch.item.file.FlatFileItemReader">
            <property name="resource" value="data/iosample/input/multiLine.txt" />
            <property name="lineMapper">
                <bean class="org.spr...DefaultLineMapper">
                    <property name="lineTokenizer" ref="orderFileTokenizer"/>
                    <property name="fieldSetMapper" ref="orderFieldSetMapper"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

Java Configuration

```
@Bean
public MultiLineTradeItemReader itemReader() {
    MultiLineTradeItemReader itemReader = new MultiLineTradeItemReader();

    itemReader.setDelegate(flatFileItemReader());

    return itemReader;
}

@Bean
public FlatFileItemReader flatFileItemReader() {
    FlatFileItemReader<Trade> reader = new FlatFileItemReaderBuilder<Trade>()
            .name("flatFileItemReader")
            .resource(new ClassPathResource("data/iosample/input/multiLine.txt"))
            .lineTokenizer(orderFileTokenizer())
            .fieldSetMapper(orderFieldSetMapper())
            .build();
    return reader;
}
```

To ensure that each line is tokenized properly, which is especially important for fixed-length input, the `PatternMatchingCompositeLineTokenizer` can be used on the delegate `FlatFileItemReader`. See `FlatFileItemReader` in the Readers and Writers chapter for more details. The delegate reader then uses a `PassThroughFieldSetMapper` to deliver a `FieldSet` for each line back to the wrapping `ItemReader`, as shown in the following example:

XML Content

```
<bean id="orderFileTokenizer" class="org.spr...PatternMatchingCompositeLineTokenizer">
    <property name="tokenizers">
        <map>
            <entry key="HEA*" value-ref="headerRecordTokenizer" />
            <entry key="FOT*" value-ref="footerRecordTokenizer" />
            <entry key="NCU*" value-ref="customerLineTokenizer" />
            <entry key="BAD*" value-ref="billingAddressLineTokenizer" />
        </map>
    </property>
</bean>
```

Java Content

```
@Bean
public PatternMatchingCompositeLineTokenizer orderFileTokenizer() {
    PatternMatchingCompositeLineTokenizer tokenizer =
            new PatternMatchingCompositeLineTokenizer();

    Map<String, LineTokenizer> tokenizers = new HashMap<>(4);

    tokenizers.put("HEA*", headerRecordTokenizer());
    tokenizers.put("FOT*", footerRecordTokenizer());
    tokenizers.put("NCU*", customerLineTokenizer());
    tokenizers.put("BAD*", billingAddressLineTokenizer());

    tokenizer.setTokenizers(tokenizers);

    return tokenizer;
}
```

This wrapper has to be able to recognize the end of a record so that it can continually call `read()` on its delegate until the end is reached. For each line that is read, the wrapper should build up the item to be returned. Once the footer is reached, the item can be returned for delivery to the `ItemProcessor` and `ItemWriter`, as shown in the following example:

```
private FlatFileItemReader<FieldSet> delegate;

public Trade read() throws Exception {
    Trade t = null;

    for (FieldSet line = null; (line = this.delegate.read()) != null;) {
        String prefix = line.readString(0);
        if (prefix.equals("HEA")) {
            t = new Trade(); // Record must start with header
        }
        else if (prefix.equals("NCU")) {
            Assert.notNull(t, "No header was found.");
            t.setLast(line.readString(1));
            t.setFirst(line.readString(2));
            ...
        }
        else if (prefix.equals("BAD")) {
            Assert.notNull(t, "No header was found.");
            t.setCity(line.readString(4));
            t.setState(line.readString(6));
          ...
        }
        else if (prefix.equals("FOT")) {
            return t; // Record must end with footer
        }
    }
    Assert.isNull(t, "No 'END' was found.");
    return null;
}
```

1.6. Executing System Commands

Many batch jobs require that an external command be called from within the batch job. Such a process could be kicked off separately by the scheduler, but the advantage of common metadata about the run would be lost. Furthermore, a multi-step job would also need to be split up into multiple jobs as well.

Because the need is so common, Spring Batch provides a `Tasklet` implementation for calling system commands, as shown in the following example:

XML Configuration

```
<bean class="org.springframework.batch.core.step.tasklet.SystemCommandTasklet">
    <property name="command" value="echo hello" />
    <!-- 5 second timeout for the command to complete -->
    <property name="timeout" value="5000" />
</bean>
```

Java Configuration

```
@Bean
public SystemCommandTasklet tasklet() {
    SystemCommandTasklet tasklet = new SystemCommandTasklet();

    tasklet.setCommand("echo hello");
    tasklet.setTimeout(5000);

    return tasklet;
}
```

1.7. Handling Step Completion When No Input is Found

In many batch scenarios, finding no rows in a database or file to process is not exceptional. The `Step` is simply considered to have found no work and completes with 0 items read. All of the `ItemReader` implementations provided out of the box in Spring Batch default to this approach. This can lead to some confusion if nothing is written out even when input is present (which usually happens if a file was misnamed or some similar issue arises). For this reason, the metadata itself should be inspected to determine how much work the framework found to be processed. However, what if finding no input is considered exceptional? In this case, programmatically checking the metadata for no items processed and causing failure is the best solution. Because this is a common use case, Spring Batch provides a listener with exactly this functionality, as shown in the class definition for `NoWorkFoundStepExecutionListener`:

```
public class NoWorkFoundStepExecutionListener extends StepExecutionListenerSupport {

    public ExitStatus afterStep(StepExecution stepExecution) {
        if (stepExecution.getReadCount() == 0) {
            return ExitStatus.FAILED;
        }
        return null;
    }

}
```

The preceding `StepExecutionListener` inspects the `readCount` property of the `StepExecution` during the 'afterStep' phase to determine if no items were read. If that is the case, an exit code of FAILED is returned, indicating that the `Step` should fail. Otherwise, `null` is returned, which does not affect the status of the `Step`.

1.8. Passing Data to Future Steps

It is often useful to pass information from one step to another. This can be done through the `ExecutionContext`. The catch is that there are two `ExecutionContexts`: one at the `Step` level and one at the `Job` level. The `Step ExecutionContext` remains only as long as the step, while the `Job ExecutionContext` remains through the whole `Job`. On the other hand, the `Step ExecutionContext` is

updated every time the Step commits a chunk, while the Job ExecutionContext is updated only at the end of each Step.

The consequence of this separation is that all data must be placed in the Step ExecutionContext while the Step is executing. Doing so ensures that the data is stored properly while the Step runs. If data is stored to the Job ExecutionContext, then it is not persisted during Step execution. If the Step fails, that data is lost.

```
public class SavingItemWriter implements ItemWriter<Object> {
    private StepExecution stepExecution;

    public void write(List<? extends Object> items) throws Exception {
        // ...

        ExecutionContext stepContext = this.stepExecution.getExecutionContext();
        stepContext.put("someKey", someObject);
    }

    @BeforeStep
    public void saveStepExecution(StepExecution stepExecution) {
        this.stepExecution = stepExecution;
    }
}
```

To make the data available to future Steps, it must be "promoted" to the Job ExecutionContext after the step has finished. Spring Batch provides the ExecutionContextPromotionListener for this purpose. The listener must be configured with the keys related to the data in the ExecutionContext that must be promoted. It can also, optionally, be configured with a list of exit code patterns for which the promotion should occur (COMPLETED is the default). As with all listeners, it must be registered on the Step as shown in the following example:

XML Configuration

```
<job id="job1">
    <step id="step1">
        <tasklet>
            <chunk reader="reader" writer="savingWriter" commit-interval="10"/>
        </tasklet>
        <listeners>
            <listener ref="promotionListener"/>
        </listeners>
    </step>

    <step id="step2">
       ...
    </step>
</job>

<beans:bean id="promotionListener" class="
org.spr....ExecutionContextPromotionListener">
    <beans:property name="keys">
        <list>
            <value>someKey</value>
        </list>
    </beans:property>
</beans:bean>
```

Java Configuration

```
@Bean
public Job job1() {
    return this.jobBuilderFactory.get("job1")
                .start(step1())
                .next(step1())
                .build();
}

@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
                .<String, String>chunk(10)
                .reader(reader())
                .writer(savingWriter())
                .listener(promotionListener())
                .build();
}

@Bean
public ExecutionContextPromotionListener promotionListener() {
    ExecutionContextPromotionListener listener = new
ExecutionContextPromotionListener();

    listener.setKeys(new String[] {"someKey" });

    return listener;
}
```

Finally, the saved values must be retrieved from the `Job ExecutionContext`, as shown in the following example:

```
public class RetrievingItemWriter implements ItemWriter<Object> {
    private Object someObject;

    public void write(List<? extends Object> items) throws Exception {
        // ...
    }

    @BeforeStep
    public void retrieveInterstepData(StepExecution stepExecution) {
        JobExecution jobExecution = stepExecution.getJobExecution();
        ExecutionContext jobContext = jobExecution.getExecutionContext();
        this.someObject = jobContext.get("someKey");
    }
}
```