

Spring Batch - Reference Documentation

Spring Batch 1.0

Copyright © 2005-2007 Dave Syer, Wayne Lund, Lucas Ward

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

1. Spring Batch Introduction	1
1.1. Introduction	1
1.1.1. Background	1
1.1.2. Usage Scenarios	2
1.1.3. Spring Batch Architecture	2
2. The Domain Language of Batch	4
2.1. Introduction	4
2.2. Batch Application Style Interactions and Services	4
2.3. Job Stereotypes	5
2.3.1. Job	6
2.3.2. JobInstance	6
2.3.3. JobParameters	7
2.3.4. JobExecution	7
2.4. Step Stereotypes	9
2.4.1. Step	9
2.4.2. StepExecution	10
2.4.3. ExecutionContext	11
2.5. JobRepository	11
2.6. JobLauncher	12
2.7. JobLocator	12
2.8. Item Reader	12
2.9. Item Writer	12
2.10. Tasklet	13
3. ItemReaders and ItemWriters	14
3.1. Introduction	14
3.2. ItemReader	14
3.3. ItemWriter	14
3.4. ItemStream	15
3.5. Flat Files	15
3.5.1. The FieldSet	16
3.5.2. FlatFileItemReader	16
3.5.3. FlatFileItemWriter	21
3.6. XML Item Readers and Writers	22
3.6.1. StaxEventItemReader	24
3.6.2. StaxEventItemWriter	26
3.7. Creating File Names at Runtime	27
3.8. Database	28
3.8.1. Cursor Based ItemReaders	28
3.8.2. Driving Query Based ItemReaders	31
3.8.3. Database ItemWriters	35
3.9. Reusing Existing Services	36
3.10. Item Transforming	36
3.10.1. The Delegate Pattern and Registering with the Step	38
3.10.2. Chaining ItemTransformers	38
3.11. Validating Input	39
3.11.1. The Delegate Pattern and Registering with the Step	40
3.12. Creating Custom ItemReaders and ItemWriters	40
3.12.1. Custom ItemReader Example	40
3.12.2. Custom ItemWriter Example	43
4. Configuring and Executing A Job	45
4.1. Introduction	45

4.2. Run Tier ...	45
4.2.1. Running Jobs from the Command Line ...	46
4.3. Job Tier ...	48
4.3.1. SimpleJobLauncher ...	48
4.3.2. SimpleJobRepository ...	50
4.3.3. SimpleJob ...	52
4.3.4. JobFactory and Stateful Components in Steps ...	54
4.4. Application Tier ...	54
4.4.1. ItemOrientedStep ...	55
4.4.2. TaskletStep ...	61
4.5. Examples of Customized Business Logic ...	62
4.5.1.	62
4.5.2. Logging Item Processing and Failures ...	62
4.5.3. Stopping a Job Manually for Business Reasons ...	63
4.5.4. Adding a Footer Record ...	64
5. Repeat ...	66
5.1. RepeatTemplate ...	66
5.1.1. RepeatContext ...	66
5.1.2. ExitStatus ...	67
5.2. Completion Policies ...	67
5.3. Exception Handling ...	68
5.4. Listeners ...	68
5.5. Parallel Processing ...	68
5.6. Declarative Iteration ...	69
6. Retry ...	70
6.1. RetryTemplate ...	70
6.1.1. RetryContext ...	70
6.2. Retry Policies ...	70
6.2.1. Stateless Retry ...	71
6.2.2. Stateful Retry ...	72
6.3. Backoff Policies ...	73
6.4. Listeners ...	73
6.5. Declarative Retry ...	73
7. Unit Testing ...	75
7.1. End To End Testing Batch Jobs ...	75
7.2. Extending Unit Test frameworks ...	76
A. List of ItemReaders ...	78
A.1. Item Readers ...	78
A.2. Item Writers ...	79
B. Meta-Data Schema ...	81
B.1. Overview ...	81
B.1.1. Version ...	81
B.1.2. Identity ...	81
B.2. BATCH_JOB_INSTANCE ...	82
B.3. BATCH_JOB_PARAMS ...	82
B.4. BATCH_JOB_EXECUTION ...	83
B.5. BATCH_STEP_EXECUTION ...	84
B.6. BATCH_STEP_EXECUTION_CONTEXT ...	85
B.7. Archiving ...	85
Glossary ...	87

Chapter 1. Spring Batch Introduction

1.1. Introduction

Many applications within the enterprise domain require bulk processing to perform business operations in mission critical environments. These business operations include automated, complex processing of large volumes of information that is most efficiently processed without user interaction. These operations typically include time based events (e.g. month-end calculations, notices or correspondence), periodic application of complex business rules processed repetitively across very large data sets (e.g. Insurance benefit determination or rate adjustments), or the integration of information that is received from internal and external systems that typically requires formatting, validation and processing in a transactional manner into the system of record. Batch processing is used to process billions of transactions every day for enterprises.

Spring Batch is a lightweight, comprehensive batch framework designed to enable the development of robust batch applications vital for the daily operations of enterprise systems. Spring Batch builds upon the productivity, POJO-based development approach, and general ease of use capabilities people have come to know from the Spring Framework, while making it easy for developers to access and leverage more advanced enterprise services when necessary. Spring Batch is not a scheduling framework. There are many good enterprise schedulers available in both the commercial and open source spaces such as Quartz, Tivoli, Control-M, etc. It is intended to work in conjunction with a scheduler, not replace a scheduler.

Spring Batch provides reusable functions that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management. It also provides more advanced technical services and features that will enable extremely high-volume and high performance batch jobs through optimization and partitioning techniques. Simple as well as complex, high-volume batch jobs can leverage the framework in a highly scalable manner to process significant volumes of information.

1.1.1. Background

While open source software projects and associated communities have focused greater attention on web-based and SOA messaging-based architecture frameworks, there has been a notable lack of focus on reusable architecture frameworks to accommodate Java-based batch processing needs, despite continued needs to handle such processing within enterprise IT environments. The lack of a standard, reusable batch architecture has resulted in the proliferation of many one-off, in-house solutions developed within client enterprise IT functions.

SpringSource and Accenture have collaborated to change this. Accenture's hands-on industry and technical experience in implementing batch architectures, SpringSource's depth of technical experience, and Spring's proven programming model together mark a natural and powerful partnership to create high-quality, market relevant software aimed at filling an important gap in enterprise Java. Both companies are also currently working with a number of clients solving similar problems developing Spring-based batch architecture solutions. This has provided some useful additional detail and real-life constraints helping to ensure the solution can be applied to the real-world problems posed by clients. For these reasons and many more, SpringSource and Accenture have teamed to collaborate on the development of Spring Batch.

Accenture has contributed previously proprietary batch processing architecture frameworks, based upon decades worth of experience in building batch architectures with the last several generations of platforms, (i.e., COBOL/Mainframe, C++/Unix, and now Java/anywhere) to the Spring Batch project along with committer resources to drive support, enhancements, and the future roadmap.

The collaborative effort between Accenture and SpringSource aims to promote the standardization of software processing approaches, frameworks, and tools that can be consistently leveraged by enterprise users when creating batch applications. Companies and government agencies desiring to deliver standard, proven solutions to their enterprise IT environments will benefit from Spring Batch.

1.1.2. Usage Scenarios

A typical batch program generally reads a large number of records from a database, file, or queue, processes the data in some fashion, and then writes back data in a modified form. Spring Batch automates this basic batch iteration, providing the capability to process similar transactions as a set, typically in an offline environment without any user interaction. Batch jobs are part of most IT projects and Spring Batch is the only open source framework that provides a robust, enterprise-scale solution.

Business Scenarios

- Commit batch process periodically
- Concurrent batch processing: parallel processing of a job
- Staged, enterprise message-driven processing
- Massively parallel batch processing
- Manual or scheduled restart after failure
- Sequential processing of dependent steps (with extensions to workflow-driven batches)
- Partial processing: skip records (e.g. on rollback)
- Whole-batch transaction: for cases with a small batch size or existing stored procedures/scripts

Technical Objectives

- Batch developers use the Spring programming model: concentrate on business logic; let the framework take care of infrastructure.
- Clear separation of concerns between the infrastructure, the batch execution environment, and the batch application.
- Provide common, core execution services as interfaces that all projects can implement.
- Provide simple and default implementations of the core execution interfaces that can be used ‘out of the box’.
- Easy to configure, customize, and extend services, by leveraging the spring framework in all layers.
- All existing core services should be easy to replace or extend, without any impact to the infrastructure layer.
- Provide a simple deployment model, with the architecture JARs completely separate from the application, built using Maven.

1.1.3. Spring Batch Architecture

Spring Batch is designed with extensibility and a diverse group of end users in mind. The figure below shows a sketch of the layered architecture that supports the extensibility and ease of use for end-user developers.

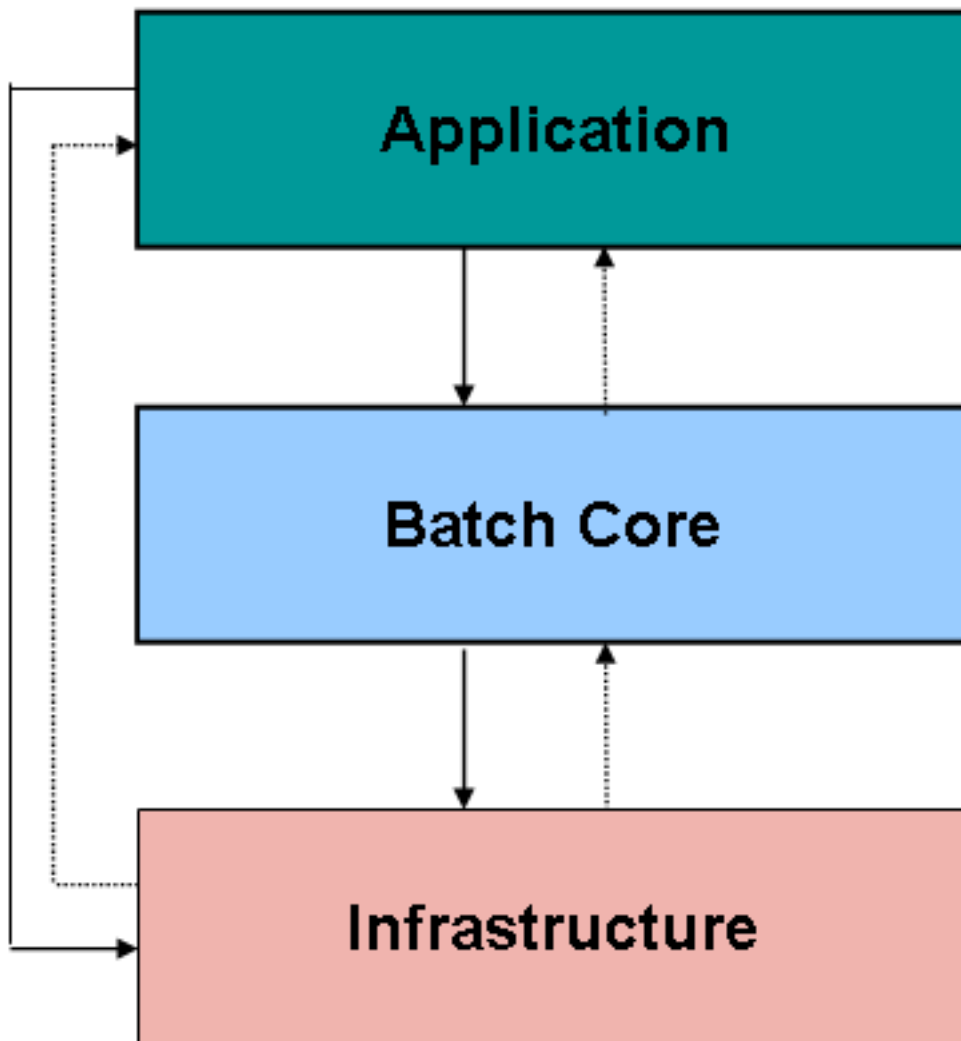


Figure 1.1: Spring Batch Layered Architecture

This layered architecture highlights three major high level components: Application, Core, and Infrastructure. The application contains all batch jobs and custom code written by developers using Spring Batch. The Batch Core contains the core runtime classes necessary to launch and control a batch job. It includes things such as a `JobLauncher`, `Job`, and `Step` implementations. Both Application and Core are built on top of a common infrastructure. This infrastructure contains common readers and writers, and services such as the `RetryTemplate`, which are used both by application developers (`ItemReader` and `ItemWriter`) and the core framework itself. (retry)

Chapter 2. The Domain Language of Batch

2.1. Introduction

To any experienced batch architect, the overall concepts of batch processing used in Spring Batch should be familiar and comfortable. There are “Jobs” and “Steps” and developer supplied processing units called ItemReaders and ItemWriters. However, because of the Spring patterns, operations, templates, callbacks, and idioms, there are opportunities for the following:

- significant improvement in adherence to a clear separation of concerns
- clearly delineated architectural layers and services provided as interfaces
- simple and default implementations that allowed for quick adoption and ease of use out-of-the-box
- significantly enhanced extensibility

The diagram below is only a slight variation of the batch reference architecture that has been used for decades. It provides an overview of the high level components, technical services, and basic operations required by a batch architecture. This architecture framework is a blueprint that has been proven through decades of implementations on the last several generations of platforms (COBOL/Mainframe, C++/Unix, and now Java/anywhere). JCL and COBOL developers are likely to be as comfortable with the concepts as C++, C# and Java developers. Spring Batch provides a physical implementation of the layers, components and technical services commonly found in robust, maintainable systems used to address the creation of simple to complex batch applications, with the infrastructure and extensions to address very complex processing needs.

2.2. Batch Application Style Interactions and Services

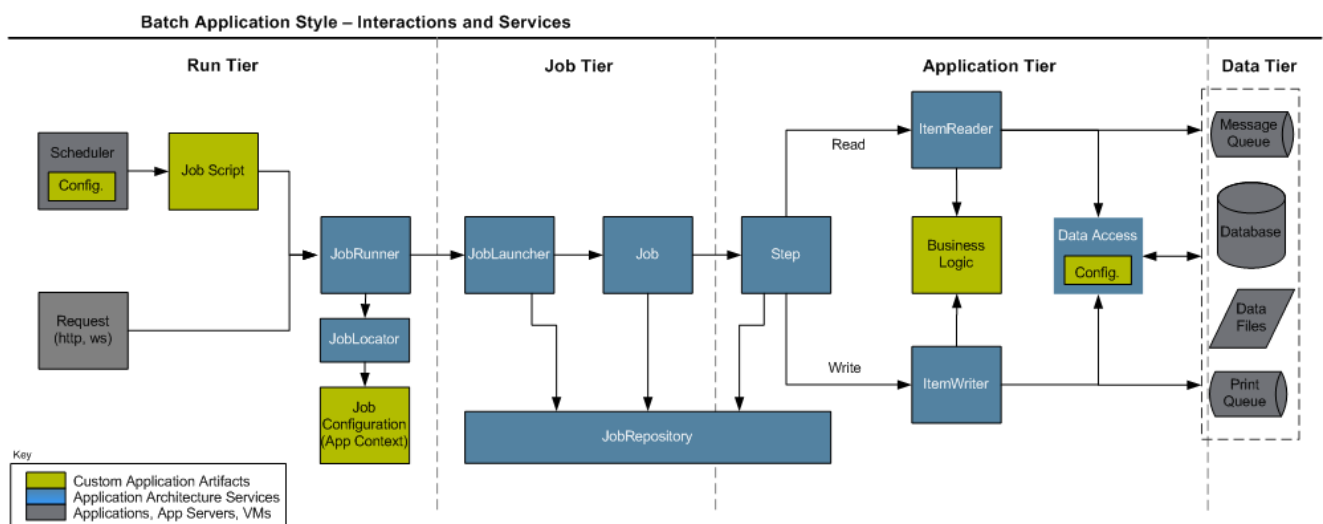


Figure 2.1: Batch Stereotypes

The above diagram highlights the interactions and key services provided by the Spring Batch framework. The colors used are important to understanding the responsibilities of a developer in Spring Batch. Grey represents an external application such as an enterprise scheduler or a database. It's important to note that scheduling is grey, and should thus be considered separate from Spring Batch. Blue represents application architecture services. In most cases these are provided by Spring Batch with out of the box implementations, but an

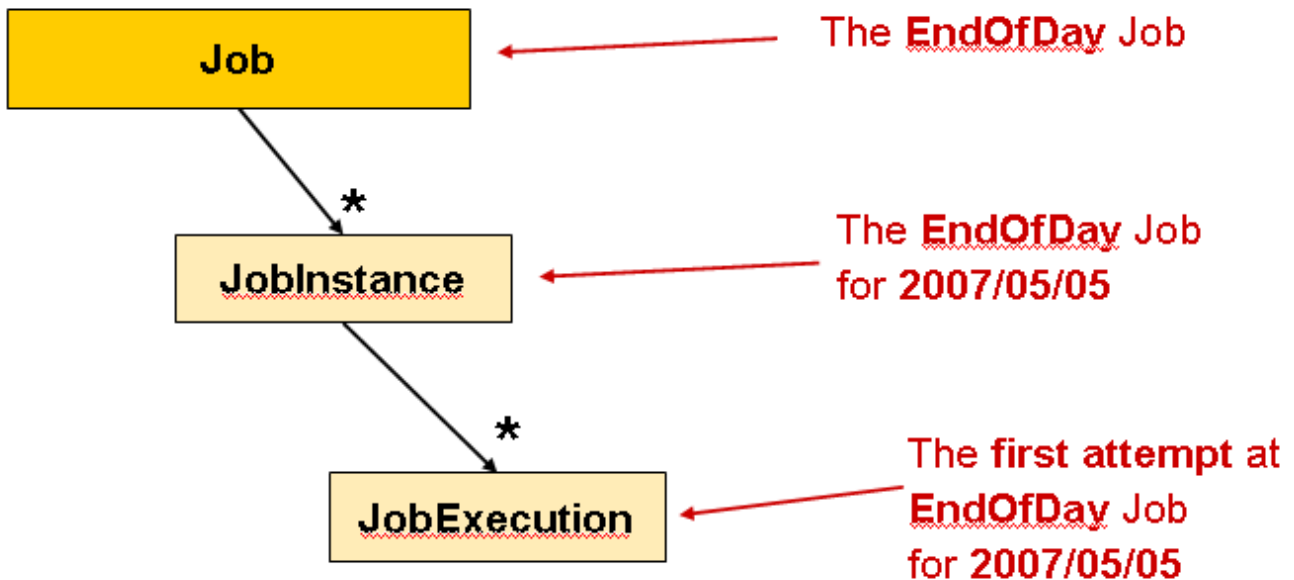
architecture team may make specific implementations that better address their specific needs. Yellow represents the pieces that must be configured by a developer. For example, they need to configure their job schedule so that the job is kicked off at the appropriate time. They also need to create a job configuration that defines how their job will be run. It is also worth noting that the `ItemReader` and `ItemWriter` used by an application may just as easily be a custom one made by the developer for the specific batch job, rather than one provided by Spring Batch or even an architecture team.

The Batch Application Style is organized into four logical tiers, which include Run, Job, Application, and Data. The primary goal for organizing an application according to the tiers is to embed what is known as "separation of concerns" within the system. These tiers can be conceptual but may prove effective in mapping the deployment of the artifacts onto physical components like Java runtimes and integration with data sources and targets. Effective separation of concerns results in reducing the impact of change to the system. The four conceptual tiers containing batch artifacts are:

- **Run Tier:** The Run Tier is concerned with the scheduling and launching of the application. A vendor product is typically used in this tier to allow time-based and interdependent scheduling of batch jobs as well as providing parallel processing capabilities.
- **Job Tier:** The Job Tier is responsible for the overall execution of a batch job. It sequentially executes batch steps, ensuring that all steps are in the correct state and all appropriate policies are enforced.
- **Application Tier:** The Application Tier contains components required to execute the program. It contains specific tasklets that address the required batch functionality and enforces policies around a tasklet execution (e.g., commit intervals, capture of statistics, etc.)
- **Data Tier:** The Data Tier provides the integration with the physical data sources that might include databases, files, or queues.

2.3. Job Stereotypes

This section describes stereotypes relating to the concept of a batch job. A job is an entity that encapsulates an entire batch process. As is common with other Spring projects, a `Job` will be wired together via an XML configuration file. This file may be referred to as the "job configuration". However, `Job` is just the top of an overall hierarchy:



2.3.1. Job

A job is represented by a Spring bean that implements the `Job` interface and contains all of the information necessary to define the operations performed by a job. A job configuration is typically contained within a Spring XML configuration file and the job's name is determined by the "id" attribute associated with the job configuration bean. The job configuration contains

- The simple name of the job
- Definition and ordering of Steps
- Whether or not the job is restartable

A default simple implementation of the `Job` interface is provided by Spring Batch in the form of the `SimpleJob` class which creates some standard functionality on top of `Job`, namely a standard execution logic that all jobs should utilize. In general, all jobs should be defined using a bean of type `SimpleJob`:

```

<bean id="footballJob"
      class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">
    <list>
      <!-- Step Bean details omitted for clarity -->
      <bean id="playerload" parent="simpleStep" />
      <bean id="gameLoad" parent="simpleStep" />
      <bean id="playerSummarization" parent="simpleStep" />
    </list>
  </property>
  <property name="restartable" value="true" />
</bean>
  
```

2.3.2. JobInstance

A `JobInstance` refers to the concept of a logical job run. Let's consider a batch job that should be run once at the end of the day, such as the 'EndOfDay' job from the diagram above. There is one 'EndOfDay' `Job`, but each individual run of the `Job` must be tracked separately. In the case of this job, there will be one logical `JobInstance` per day. For example, there will be a January 1st run, and a January 2nd run. If the January 1st run fails the first time and is run again the next day, it's still the January 1st run. (Usually this corresponds with

the data its processing as well, meaning the January 1st run processes data for January 1st, etc) That is to say, each `JobInstance` can have multiple executions. (`JobExecution` is discussed in more detail below) and only one `JobInstance` corresponding to a particular `Job` can be running at a given time. The definition of a `JobInstance` has absolutely no bearing on the data the will be loaded. It is entirely up to the `ItemReader` implementation used to determine how data will be loaded. For example, in the `EndOfDay` scenario, there may be a column on the data that indicates the 'effective date' or 'schedule date' to which the data belongs. So, the January 1st run would only load data from the 1st, and the January 2nd run would only use data from the 2nd. Because this determination will likely be a business decision, it is left up to the `ItemReader` to decide. What using the same `JobInstance` will determine, however, is whether or not the 'state' (i.e. the `ExecutionContext`, which is discussed below) from previous executions will be used. Using a new `JobInstance` will mean 'start from the beginning' and using an existing instance will generally mean 'start from where you left off'.

2.3.3. JobParameters

Having discussed `JobInstance` and how it differs from `Job`, the natural question to ask is: "how is one `JobInstance` distinguished from another?" The answer is: `JobParameters`. `JobParameters` are any set of parameters used to start a batch job, which can be used for identification or even as reference data during the run. In the example above, where there are two instances, one for January 1st, and another for January 2nd, there is really only one `Job`, one that was started with a job parameter of 01-01-2008 and another that was started with a parameter of 01-02-2008. Thus, the contract can be defined as: `JobInstance = Job + JobParameters`. This allows you to effectively control how you define a `JobInstance`, since you control what parameters are passed in.

2.3.4. JobExecution

A `JobExecution` refers to the technical concept of a single attempt to run a `Job`. An execution may end in failure or success, but the `JobInstance` corresponding to a given execution will not be marked as complete unless the execution completes successfully. For instance, if we have a `JobInstance` of the `EndOfDay` job for 01-01-2008, as described above, that fails to successfully complete its work the first time it is run, when we attempt to run it again (with the same job parameters of 01-01-2008), a new job execution will be created.

A `Job` defines what a job is and defines how it is to be executed, and `JobInstance` is a purely organization object to group executions together, primarily to enable correct restart. A `JobExecution`, however, is the primary storage mechanism for what actually happened during a run, and as such contains many more properties that must be controlled and persisted:

Table 2.1. JobExecution properties

status	A <code>BatchStatus</code> object that indicates the status of the execution. While it's running, it's <code>BatchStatus.STARTED</code> , if it fails it's <code>BatchStatus.FAILED</code> , and if it finishes successfully it's <code>BatchStatus.COMPLETED</code>
startTime	A <code>java.util.Date</code> representing the current system time when the execution was started.
endTime	A <code>java.util.Date</code> representing the current system time when the execution finished, regardless of whether or not it was successful.
exitStatus	The <code>ExitStatus</code> indicating the result of the run. It is most important because it contains an exit code that

	will be returned to the caller. See chapter 5 for more details.
--	---

These properties are important because they will be persisted and can be used to completely determine the status of an execution. For example, if the EndOfDay job for 01-01 is executed at 9:00 PM, and fails at 9:30, the following entries will be made in the batch meta data tables:

Table 2.2. BATCH_JOB_INSTANCE

JOB_INSTANCE_ID	JOB_NAME
1	EndOfDayJob

Table 2.3. BATCH_JOB_PARAMS

JOB_INSTANCE_ID	TYPE_CD	KEY_NAME	DATE_VAL
1	DATE	schedule.Date	2008-01-01 00:00:00

Table 2.4. BATCH_JOB_EXECUTION

JOB_EXECUTION_ID	JOB_INSTANCE_ID	START_TIME	END_TIME	STATUS
1	1	2008-01-01 21:00:23.571	2008-01-01 21:30:17.132	FAILED

Note

extra columns in the table have been removed for added clarity.

Now that the job has failed, let's assume that it took the entire course of the night for the problem to be determined, so that the 'batch window' is now closed. Assuming the window starts at 9:00 PM, the job will be kicked off again for 01-01, starting where it left off and completing successfully at 9:30. Because it's now the next day, the 01-02 job must be run as well, which is kicked off just afterwards at 9:31, and completes in its normal one hour time at 10:30. There is no requirement that one `JobInstance` be kicked off after another, unless there is potential for the two jobs to attempt to access the same data, causing issues with locking at the database level. It is entirely up to the scheduler to determine when to run. Since they're separate `JobInstances`, Spring Batch will make no attempt to stop them from being run concurrently. (Attempting to run the same `JobInstance` while another is already running will result in a `JobExecutionAlreadyRunningException` being thrown) There should now be an extra entry in both the `JobInstance` and `JobParameters` tables, and two extra entries in the `JobExecution` table:

Table 2.5. BATCH_JOB_INSTANCE

JOB_INSTANCE_ID	JOB_NAME
1	EndOfDayJob
2	EndOfDayJob

Table 2.6. BATCH_JOB_PARAMS

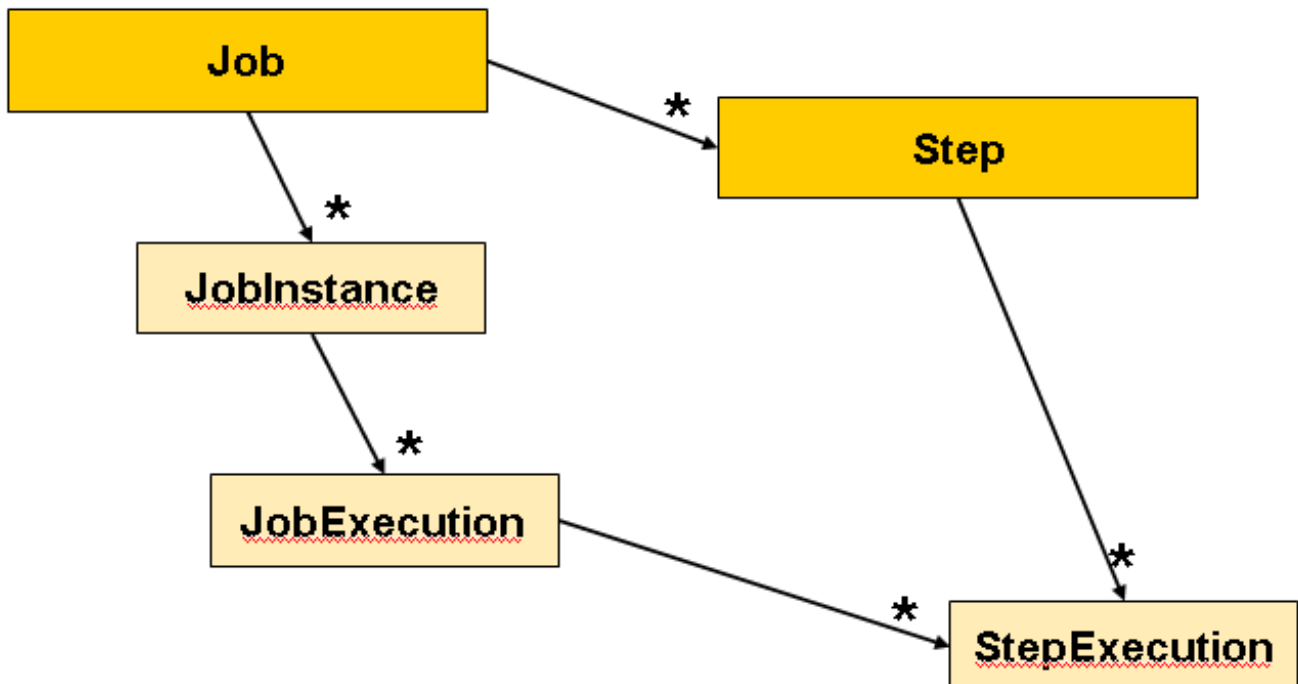
JOB_INSTANCE_ID	TYPE_CD	KEY_NAME	DATE_VAL
1	DATE	schedule.Date	2008-01-01 00:00:00
2	DATE	schedule.Date	2008-01-02 00:00:00

Table 2.7. BATCH_JOB_EXECUTION

JOB_EXECUTION_ID	JOB_INSTANCE_ID	START_TIME	END_TIME	STATUS
1	1	2008-01-01 21:00	2008-01-01 21:30	FAILED
2	1	2008-01-02 21:00	2008-01-02 21:30	COMPLETED
3	2	2008-01-02 21:31	2008-01-02 22:29	COMPLETED

2.4. Step Stereotypes

A `Step` is a domain object that encapsulates an independent, sequential phase of a batch job. Therefore, every `Job` is composed entirely of one or more steps. A `Step` should be thought of as a unique processing stream that will be executed in sequence. For example, if you have one step that loads a file into a database, another that reads from the database, validates the data, preforms processing, and then writes to another table, and another that reads from that table and writes out to a file. Each of these steps will be performed completely before moving on to the next step. The file will be completely read into the database before step 2 can begin. As with `Job`, a `Step` has an individual `StepExecution` that corresponds with a unique `JobExecution`:



2.4.1. Step

A `Step` contains all of the information necessary to define and control the actual batch processing. This is a necessarily vague description because the contents of any given `Step` are at the discretion of the developer writing a `Job`. A `Step` can be as simple or complex as the developer desires. A simple `Step` might load data from a file into the database, requiring little or no code. (depending upon the implementations used) A more complex `Step` may have complicated business rules that are applied as part of the processing.

Steps are defined by instantiating implementations of the `Step` interface. Two step implementation classes are available in the Spring Batch framework, and they are each discussed in detail in Chapter 4 of this guide. For most situations, the `ItemOrientedStep` implementation is sufficient, but for situations where only one call is needed, such as a stored procedure call or a wrapper around existing script, a `TaskletStep` may be a better option.

2.4.2. StepExecution

A `StepExecution` represents a single attempt to execute a `Step`. Using the example from `JobExecution`, if there is a `JobInstance` for the "EndOfDayJob", with `JobParameters` of "01-01-2008" that fails to successfully complete its work the first time it is run, when it is executed again, a new `StepExecution` will be created. Each of these step executions may represent a different invocation of the batch framework, but they will all correspond to the same `JobInstance`, just as multiple `JobExecutions` belong to the same `JobInstance`.

Step executions are represented by objects of the `StepExecution` class. Each execution contains a reference to its corresponding step and `JobExecution`, and transaction related data such as commit and rollback count and start and end times. Additionally, each step execution will contain an `ExecutionContext`, which contains any data a developer needs persisted across batch runs, such as statistics or state information needed to restart. The following is a listing of the properties for `StepExecution`:

Table 2.8. StepExecution properties

status	A <code>BatchStatus</code> object that indicates the status of the execution. While it's running, the status is <code>BatchStatus.STARTED</code> , if it fails the status is <code>BatchStatus.FAILED</code> , and if it finishes successfully the status is <code>BatchStatus.COMPLETED</code>
startTime	A <code>java.util.Date</code> representing the current system time when the execution was started.
endTime	A <code>java.util.Date</code> representing the current system time when the execution finished, regardless of whether or not it was successful.
exitStatus	The <code>ExitStatus</code> indicating the result of the execution. It is most important because it contains an exit code that will be returned to the caller. See chapter 5 for more details.
executionContext	The 'property bag' containing any user data that needs to be persisted between executions.
commitCount	The number transactions that have been committed for this execution
itemCount	The number of items that have been processed for this execution.

2.4.3. ExecutionContext

An `ExecutionContext` represents a collection of key/value pairs that are persisted and controlled by the framework in order to allow developers a place to store persistent state that is scoped to a `StepExecution`. For those familiar with Quartz, it is very similar to `JobDataMap`. The best usage example is restart. Using flat file input as an example, while processing individual lines, the framework periodically persists the `ExecutionContext` at commit points. This allows the `ItemReader` to store its state in case a fatal error occurs during the run, or even if the power goes out. All that is needed is to put the current number of lines read into the context, and the framework will do the rest:

```
executionContext.putLong(getKey(LINES_READ_COUNT), reader.getPosition());
```

The call above will store the current number of lines read into the `ExecutionContext`. It should be made just before the framework commits. Being notified before a commit requires one of the various `StepListeners`, or an `ItemStream`, which are discussed in more detail later in this guide. When the `ItemReader` is opened, it can check to see if it has any stored state in the context, and initialize itself from there:

```
if (executionContext.containsKey(getKey(LINES_READ_COUNT))) {
    log.debug("Initializing for restart. Restart data is: " + executionContext);

    long lineCount = executionContext.getLong(getKey(LINES_READ_COUNT));

    LineReader reader = getReader();

    Object record = "";
    while (reader.getPosition() < lineCount && record != null) {
        record = readLine();
    }
}
```

The `ExecutionContext` can also be used for statistics that need to be persisted about the run itself. For example, if a flat file contains orders for processing that exist across multiple lines, it may be necessary to store how many orders have been processed (which is much different from than the number of lines read) so that an email can be sent at the end of the `Step` with the total orders processed in the body. The framework handles storing this for the developer, in order to correctly scope it with an individual `JobInstance`. It can be very difficult to know whether an existing `ExecutionContext` should be used or not. For example, using the 'EndOfDay' example from above, when the 01-01 run starts again for the second time, the framework recognizes that it is the same `JobInstance` and on an individual `Step` basis, pulls the `ExecutionContext` out of the database and hands it as part of the `StepExecution` to the `Step` itself. Conversely, for the 01-02 run the framework recognizes that it is a different instance, so an empty context must be handed to the `Step`. There are many of these types of determinations that the framework makes for the developer to ensure the state is given to them at the correct time. It is also important to note that exactly one `ExecutionContext` exists per `StepExecution` at any given time. Clients of the `ExecutionContext` should be careful because this creates a shared keyspace, so care should be taken when putting values in to ensure no data is overwritten, however, the `Step` stores absolutely no data in the context, so there is no way to adversely affect the framework.

2.5. JobRepository

`JobRepository` is the persistence mechanism for all of the Stereotypes mentioned above. When a job is first launched, a `JobExecution` is obtained by calling the repository's `createJobExecution` method, and during the course of execution, `StepExecution` and `JobExecution` are persisted by passing them to the repository:

```
public interface JobRepository {

    public JobExecution createJobExecution(Job job, JobParameters jobParameters)
        throws JobExecutionAlreadyRunningException, JobRestartException;
```

```

void saveOrUpdate(JobExecution jobExecution);

void saveOrUpdate(StepExecution stepExecution);

void saveOrUpdateExecutionContext(StepExecution stepExecution);

StepExecution getLastStepExecution(JobInstance jobInstance, Step step);

int getStepExecutionCount(JobInstance jobInstance, Step step);

}

```

2.6. JobLauncher

`JobLauncher` represents a simple interface for launching a `Job` with a given set of `JobParameters`:

```

public interface JobLauncher {

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

}

```

It is expected that implementations will obtain a valid `JobExecution` from the `JobRepository` and execute the `Job`.

2.7. JobLocator

`JobLocator` represents an interface for locating a `Job`:

```

public interface JobLocator {

    Job getJob(String name) throws NoSuchJobException;

}

```

This interface is very necessary due to the nature of Spring itself. Because we can't guarantee one `ApplicationContext` equals one `Job`, an abstraction is needed to obtain a `Job` for a given name. It becomes especially useful when launching jobs from within a Java EE application server.

2.8. Item Reader

`ItemReader` is an abstraction that represents the retrieval of input for a `Step`, one item at a time. When the `ItemReader` has exhausted the items it can provide, it will indicate this by returning null. More details about the `ItemReader` interface and its various implementations can be found in Chapter 3.

2.9. Item Writer

`ItemWriter` is an abstraction that represents the output of a `Step`, one item at a time. Generally, an item writer has no knowledge of the input it will receive next, only the item that was passed in its current invocation. More details about the `ItemWriter` interface and its various implementations can be found in Chapter 3.

2.10. Tasklet

A `Tasklet` represents the execution of a logical unit of work, as defined by its implementation of the Spring Batch provided `Tasklet` interface. A `Tasklet` is useful for encapsulating processing logic that is not natural to split into read-(transform)-write phases, such as invoking a system command or a stored procedure.

Chapter 3. ItemReaders and ItemWriters

3.1. Introduction

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 two key interfaces to help perform bulk reading and writing: `ItemReader` and `ItemWriter`.

3.2. 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 describe records with fields of data defined by fixed positions in the file or delimited by some special character (e.g. Comma).
- XML - XML ItemReaders process XML independently of technologies used for parsing, mapping and validating objects. Input data allows for the validation of and XML file against an XSD schema.
- Database - A database resource is accessed that returns resultsets which can be mapped to objects for processing. The default SQL Input Sources invoke a `RowMapper` to return objects, keep track of the current row if restart is required, basic statistics, and some transaction enhancements that will be explained later.

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

`ItemReader` is a basic interface for generic input operations:

```
public interface ItemReader {  
    Object read() throws Exception;  
    void mark() throws MarkFailedException;  
    void reset() throws ResetFailedException;  
}
```

The `read` method defines the most essential contract of the `ItemReader`, calling it returns one `Item`, returning 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 will be mapped to a usable domain object (i.e. Trade, Foo, etc) but there is no requirement in the contract to do so.

The `mark` and `reset` methods are important due to the transactional nature of batch processing. `Mark()` will be called before reading begins. Calling `reset` at anytime will position the `ItemReader` to its position when `mark` was last called. The semantics are very similar to `java.io.Reader`.

3.3. ItemWriter

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

the serialization of the output is specific for every batch job.

As with `ItemReader`, `ItemWriter` is a fairly generic interface:

```
public interface ItemWriter {  
    void write(Object item) throws Exception;  
    void flush() throws FlushFailedException;  
    void clear() throws ClearFailedException;  
}
```

As with `read` on `ItemReader`, `write` provides the basic contract of `ItemWriter`, it will attempt to write out the item passed in as long as it is open. As with `mark` and `reset`, `flush` and `clear` are necessary due to the transactional nature of batch processing. Because it is generally expected that items will be 'batched' together into a chunk, and then output, it is expected that an `ItemWriter` will perform some type of buffering. `flush` will empty the buffer by actually writing the items out, whereas `clear` will simply throw the contents of the buffer away. In most cases, a `Step` implementation will call `flush` before a commit and `clear` in case of rollback. It is expected that implementations of the `Step` interface will call these methods.

3.4. 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:

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

Before describing each method, it's worth briefly mentioning the `ExecutionContext`. Clients of an `ItemReader` that also implements `ItemStream` should call `open` before any calls to `read`, to open any resources such as files or obtain connections. A similar restriction applies to an `ItemWriter` is also implements `ItemStream`. As mentioned before, 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` will be called to ensure any resources allocated during `open` will be released safely. `update` is called primarily to ensure that any state currently being held is loaded into the provided `ExecutionContext`. This method will be 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 will be returned if the same `JobInstance` is started again. For those familiar with Quartz, the semantics are very similar to a Quartz `JobDataMap`.

3.5. 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 general types:

Delimited and Fixed Length.

3.5.1. 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 `Strings`. 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 very similar to a `Jdbc ResultSet`. `FieldSets` only require one argument, a `String` array of tokens. Optionally, you can also configure in the names of the fields so that the fields may be accessed either by index or name as patterned after `ResultSet`. In code it means it's as simple as:

```
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`, etc. 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 erroring out due to a format exception, or when doing simple data conversions.

3.5.2. 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 `FlatFileItemReader`, which provides basic functionality for reading and parsing flat files. `FlatFileItemReader` class has several properties. The three most important of these properties are `Resource`, `FieldSetMapper` and `LineTokenizer`. The `FieldSetMapper` and `LineTokenizer` interfaces will be 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 4.Resources](#). Therefore, this guide will not go into the details of creating `Resource` objects. A `resource` is used to locate, open, and close resources. It can be as simple as:

```
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. It's sufficient to know that 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.

The other properties in `FlatFileItemReader` allow you to further specify how your data will be interpreted:

Table 3.1. Flat File Item Reader Properties

Property	Type	Description
encoding	String	Specifies what text encoding to use

Property	Type	Description
		- default is "ISO-8859-1"
comments	String[]	Specifies line prefixes that indicate comment rows
linesToSkip	int	Number of lines to ignore at the top of the file
firstLineIsHeader	boolean	Indicates that the first line of the file is a header containing field names. If the column names have not been set yet and the tokenizer extends <code>AbstractLineTokenizer</code> , field names will be set automatically from this line
recordSeparatorPolicy	RecordSeparatorPolicy	Used to determine where the line endings are and do things like continue over a line ending if inside a quoted string.

3.5.2.1. FieldSetMapper

The `FieldSetMapper` interface defines a single method, `mapLine`, which takes a `FieldSet` object and maps its contents to an object. This object may be a custom DTO or domain object, or it could be as simple as an array, depending on your needs. 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:

```
public interface FieldSetMapper {
    public Object mapLine(FieldSet fs);
}
```

As you can see, the pattern used is exactly the same as `RowMapper` used by `JdbcTemplate`.

3.5.2.2. LineTokenizer

Because there can be many formats of flat file data, which all need to be converted to a `FieldSet` so that a `FieldSetMapper` can create a useful domain object from them, an abstraction for turning a line of input into a `FieldSet` is necessary. In Spring Batch, this is called a `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 will be returned. This will then be passed to a `FieldSetMapper`. Spring Batch contains the following `LineTokenizers`:

- `DelimitedLineTokenizer` - Used for files that separate records by a delimiter. The most common is a comma,

but pipes or semicolons are often used as well

- `FixedLengthTokenizer` - Used for tokenizing files where each record is separated by a 'fixed width' that must be defined per record.
- `PrefixMatchingCompositeLineTokenizer` - Tokenizer that determines which among a list of Tokenizers should be used on a particular line by checking against a prefix.

3.5.2.3. Simple Delimited File Reading Example

Now that the basic interfaces for reading in flat files have been defined, a simple example explaining how they work together is helpful. In it's most simple form, the flow when reading a line from a file is the following:

1. Read one line from the file.
2. Pass the string 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

The following example will be used to illustrate this using 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"
```

We want to map this data to the following `Player` domain object:

```
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...
}
```

In order to map a `FieldSet` into our `Player` object, we need to create a `FieldSetMapper` that returns players:

```
protected static class PlayerFieldSetMapper implements FieldSetMapper {
    public Object mapLine(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;
}
}

```

We can then read in from the file by correctly constructing our `FlatFileItemReader` and calling `read()`:

```

FlatFileItemReader itemReader = new FlatFileItemReader();
itemReader.setResource = new FileSystemResource("resources/players.csv");
//DelimitedLineTokenizer defaults to comma as it's delimiter
itemReader.setLineTokenizer(new DelimitedLineTokenizer());
itemReader.setFieldSetMapper(new PlayerFieldSetMapper());
itemReader.read();

```

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

3.5.2.4. Mapping fields by name

There is one additional functionality line tokenizers that is similar in function to a `JDBC ResultSet`. The names of the fields can be injected into the `LineTokenizer` to increase the readability of the mapping function. First, we tell the `LineTokenizer` what the names of the fields in the fieldset are:

```

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

```

and provide a `FieldSetMapper` that uses this information as follows:

```

public class PlayerMapper implements FieldSetMapper {
    public Object mapLine(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;
    }
}

```

3.5.2.5. 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 using the JavaBean spec. Again using the football example, the `FieldSetMapper` configuration looks like the following:

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

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

For each entry in the `FieldSet`, the mapper will look 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 will look for setters matching a property name. Each available field in the `FieldSet` will be mapped, and the resultant `Player` object will be returned, with no code required.

3.5.2.6. 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 is below:

```
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 order - 12 characters long.
2. Quantity: Number of this item being ordered - 3 characters long.
3. Price: Price of the item - 4 characters long.
4. Customer: Id of the customer ordering the item - 8 characters long.

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

```
<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>
```

This `LineTokenizer` will return the same `FieldSet` as if a `delimiter` had been used, allowing the same approaches above to be used such as the `BeanWrapperFieldSetMapper`, in a way that is ignorant of how the actual line was parsed.

3.5.2.7. 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: one record equals one line. However, this may not always be the case. Its very common that a file might have records spanning multiple lines with multiple formats. The following excerpt from a file illustrates this:

```

HEA;0013100345;2007-02-15
NCU;Smith;Peter;;T;20014539;F
BAD;;Oak Street 31/A;;Small Town;00235;IL;US
SAD;Smith, Elizabeth;Elm Street 17;;Some City;30011;FL;United States
BIN;VISA;VISA-12345678903
LIT;1044391041;37.49;0;0;4.99;2.99;1;45.47
LIT;2134776319;221.99;5;0;7.99;2.99;1;221.87
SIN;UPS;EXP;DELIVER ONLY ON WEEKDAYS
FOT;2;2;267.34

```

Everything between the line starting with 'HEA' and the line starting with 'FOT' is considered one record. The `PrefixMatchingCompositeLineTokenizer` makes this easier by matching the prefix in a line with a particular tokenizer:

```

<bean id="orderFileDescriptor"
      class="org.springframework.batch.io.file.transform.PrefixMatchingCompositeLineTokenizer">
  <property name="tokenizers">
    <map>
      <entry key="HEA" value-ref="headerRecordDescriptor" />
      <entry key="FOT" value-ref="footerRecordDescriptor" />
      <entry key="BCU" value-ref="businessCustomerLineDescriptor" />
      <entry key="NCU" value-ref="customerLineDescriptor" />
      <entry key="BAD" value-ref="billingAddressLineDescriptor" />
      <entry key="SAD" value-ref="shippingAddressLineDescriptor" />
      <entry key="BIN" value-ref="billingLineDescriptor" />
      <entry key="SIN" value-ref="shippingLineDescriptor" />
      <entry key="LIT" value-ref="itemLineDescriptor" />
      <entry key="" value-ref="defaultLineDescriptor" />
    </map>
  </property>
</bean>

```

This ensures that the line will be parsed correctly, which is especially important for fixed length input, with the correct field names. Any users of the `FlatFileItemReader` in this scenario must continue calling `read` until the footer for the record is returned, allowing them to return a complete order as one 'item'.

3.5.3. FlatFileItemWriter

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

3.5.3.1. LineAggregator

Just as the `LineTokenizer` interface is necessary to take a string and split it into tokens, 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`:

```

public interface LineAggregator {

    public String aggregate(FieldSet fieldSet);

}

```

The `LineAggregator` is exactly the opposite of a `LineTokenizer`. `LineTokenizer` takes a `String` and returns a `FieldSet`, whereas `LineAggregator` takes a `FieldSet` and returns a `String`. As with reading there are two types: `DelimitedLineAggregator` and `FixedLengthLineAggregator`.

3.5.3.2. FieldSetCreator

Because the `LineAggregator` interface uses a `FieldSet` as its mechanism for converting to a string, there needs

to be an interface that describes how to convert from an object into a `FieldSet`:

```
public interface FieldSetCreator {
    FieldSet mapItem(Object data);
}
```

As with `LineTokenizer` and `LineAggregator`, `FieldSetCreator` is the polar opposite of `FieldSetMapper`. `FieldSetMapper` takes a `FieldSet` and returns a mapped object, whereas a `FieldSetCreator` takes an `Object` and returns a `FieldSet`.

3.5.3.3. Simple Delimited File Writing Example

Now that both the `LineAggregator` and `FieldSetCreator` interfaces have been defined, the basic flow of writing can be explained:

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

The following excerpt from the `FlatFileItemWriter` expresses this in code:

```
public void write(Object data) throws Exception {
    FieldSet fieldSet = fieldSetCreator.mapItem(data);
    getOutputState().write(lineAggregator.aggregate(fieldSet) + LINE_SEPARATOR);
}
```

A simple configuration with the smallest amount of setters would look like the following:

```
<bean id="itemWriter"
    class="org.springframework.batch.io.file.FlatFileItemWriter">
    <property name="resource"
        value="file:target/test-outputs/20070122.testStream.multilineStep.txt" />
    <property name="fieldSetCreator">
        <bean class="org.springframework.batch.io.file.mapping.PassThroughFieldSetMapper"/>
    </property>
</bean>
```

3.5.3.4. 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 straight forward contract should exist for `FlatFileItemWriter`, if the file already exists, throw an exception, if it does not, create it and start writing. However, potentially restarting a `Job` can cause issues. In the normal restart scenario, the contract is reversed, if the file exists start writing to it from the last known good position, 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 will cause an existing file with the same name to be deleted when the writer is opened.

3.6. 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 allowing the user only to provide callbacks).

Lets take a closer look 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 (FieldSets) that need to be tokenized, it is assumed an XML resource is a collection of 'fragments' corresponding to individual records. Note that OXM tools are designed to work with standalone XML documents rather than XML fragments cut out of an XML document, therefore the Spring Batch infrastructure needs to work around this fact, as described below:

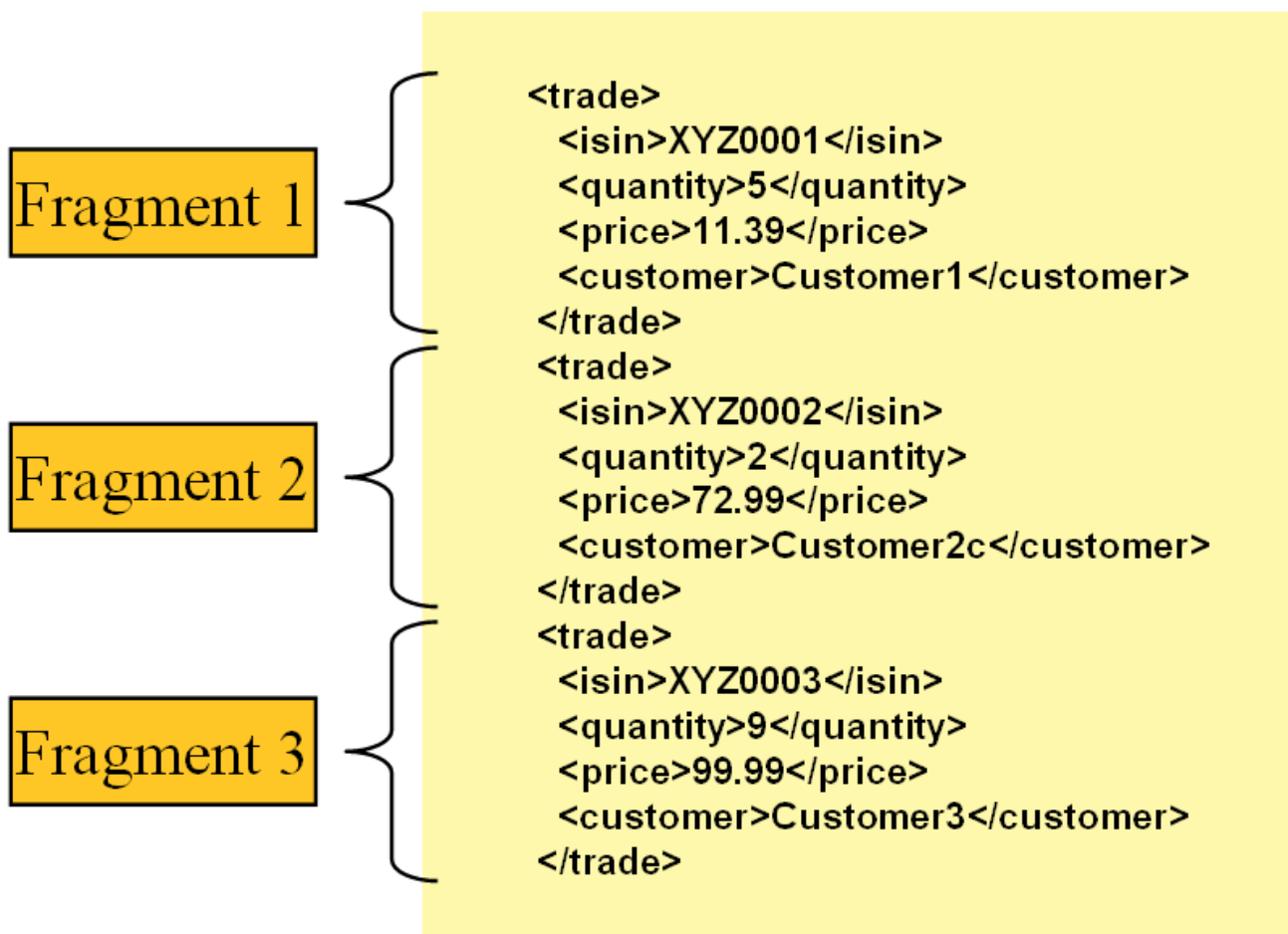


Figure 3.1: XML Input

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 can be shown as the following:

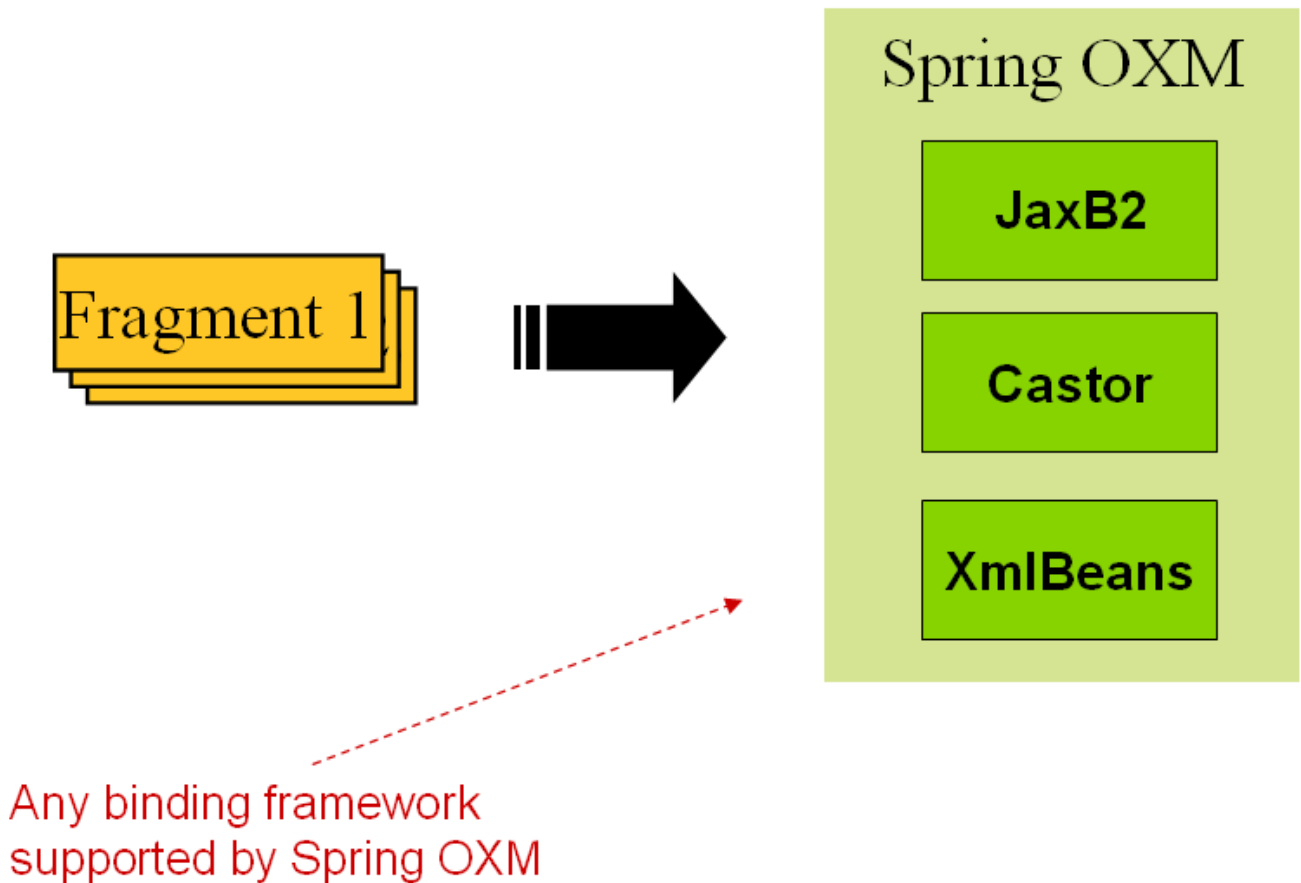


Figure 3.2: OXM Binding

Now with an introduction to OXM and how one can use XML fragments to represent records, let's take a closer look at Item Readers and Item Writers.

3.6.1. StaxEventItemReader

The `StaxEventItemReader` configuration provides a typical setup for the processing of records from an XML input stream. First, let's examine a 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>Customer2</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 we need the following:

- Root Element Name - 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 - Spring Resource that represents the file to be read.
- FragmentDeserializer - UnMarshalling facility provided by Spring OXM for mapping the XML fragment to an object.

```
<property name="itemReader">
  <bean class="org.springframework.batch.io.xml.StaxEventItemReader">
    <property name="fragmentRootElementName" value="trade" />
    <property name="resource" value="data/staxJob/input/20070918.testStream.xmlFileStep.xml" />
    <property name="fragmentDeserializer">
      <bean class="org.springframework.batch.io.xml.oxm.UnmarshallingEventReaderDeserializer">
        <constructor-arg>
          <bean class="org.springframework.oxm.xstream.XStreamMarshaller">
            <property name="aliases" ref="aliases" />
          </bean>
        </constructor-arg>
      </bean>
    </property>
  </bean>
</property>
```

Notice that in this example we have chosen to use an `XStreamMarshaller` that requires an alias passed in as a map with the first key and value being the name of the fragment (i.e. 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:

```
<util:map id="aliases">
  <entry key="trade"
    value="org.springframework.batch.sample.domain.Trade" />
  <entry key="isin" value="java.lang.String" />
  <entry key="quantity" value="long" />
  <entry key="price" value="java.math.BigDecimal" />
  <entry key="customer" value="java.lang.String" />
</util:map>
```

On input the reader reads the XML resource until it recognizes a new fragment is about to start (by matching the tag name by default). The reader creates a standalone XML document from the fragment (or at least makes it appear so) 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, if you were to see this in scripted code like Java the injection provided by the spring configuration would look something like the following:

```
StaxEventItemReader xmlStaxEventItemReader = new StaxEventItemReader()
Resource resource = new ByteArrayResource(xmlResource.getBytes())

Map aliases = new HashMap();
aliases.put("trade", "org.springframework.batch.sample.domain.Trade");
aliases.put("isin", "java.lang.String");
aliases.put("quantity", "long");
aliases.put("price", "java.math.BigDecimal");
```

```

aliases.put("customer","java.lang.String");
Marshaller marshaller = new XStreamMarshaller();
marshaller.setAliases(aliases);
xmlStaxEventItemReader.setFragmentDeserializer(new UnmarshallingEventReaderDeserializer(marshaller));
xmlStaxEventItemReader.setResource(resource);
xmlStaxEventItemReader.setFragmentRootElementName("trade");
xmlStaxEventItemReader.open(new ExecutionContext());

boolean hasNext = true

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

```

3.6.2. StaxEventItemWriter

Output works symmetrically to input. The `StaxEventItemWriter` needs a `Resource`, a serializer, and a `rootTagName`. A Java object is passed to a serializer (typically a wrapper around Spring OXM `Marshaller`) which writes to a `Resource` using a custom event writer that filters the `StartDocument` and `EndDocument` events produced for each fragment by the OXM tools. We'll show this in an example using the `MarshallingEventWriterSerializer`. The Spring configuration for this setup looks as follows:

```

<bean class="org.springframework.batch.item.xml.StaxEventItemWriter" id="tradeStaxWriter">
  <property name="resource" value="file:target/test-outputs/20070918.testStream.xmlFileStep.output.xml" />
  <property name="serializer" ref="tradeMarshallingSerializer" />
  <property name="rootTagName" value="trades" />
  <property name="overwriteOutput" value="true" />
</bean>

```

The configuration sets up the three required properties and optionally sets the `overwriteOutput=true`, mentioned earlier in the chapter for specifying whether an existing file can be overwritten. The `TradeMarshallingSerializer` is configured as follows:

```

<bean class="org.springframework.batch.item.xml.oxm.MarshallingEventWriterSerializer" id="tradeMarshallingSerializer">
  <constructor-arg>
    <bean class="org.springframework.oxm.xstream.XStreamMarshaller">
      <property name="aliases" ref="aliases" />
    </bean>
  </constructor-arg>
</bean>

```

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

```

StaxEventItemWriter staxItemWriter = new StaxEventItemWriter()
FileSystemResource resource = new FileSystemResource(File.createTempFile("StaxEventWriterOutputSourceTests",
    ".xml"));

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

MarshallingEventWriterSerializer tradeMarshallingSerializer = new MarshallingEventWriterSerializer(marshaller)
staxItemWriter.setResource(resource)

```

```

staxItemWriter.setSerializer(tradeMarshallingSerializer)
staxItemWriter.setRootTagName("trades")
staxItemWriter.setOverwriteOutput(true)

ExecutionContext executionContext = new ExecutionContext()
staxItemWriter.open(executionContext)
Trade trade = new Trade()
trade.isin = "XYZ0001"
trade.quantity = 5
trade.price = 11.39
trade.customer = "Customer1"
println trade
staxItemWriter.write(trade)
staxItemWriter.flush()

```

For a complete example configuration of XML input and output and a corresponding Job see the sample `xmlStaxJob`.

3.7. Creating File Names at Runtime

Both the XML and Flat File examples above use the Spring `Resource` abstraction to obtain the file to read or write from. This works because `Resource` has a `getFile` method, that returns a `java.io.File`. Both XML and Flat File resources can be configured using standard Spring constructs:

```

<bean id="flatFileItemReader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="resource"
            value="file://outputs/20070122.testStream.CustomerReportStep.TEMP.txt" />
</bean>

```

The above `Resource` will load the file from the file system, at the location specified. Note that absolute locations have to start with a double slash ("`file://`"). In most spring applications, this solution is good enough because the names of these are known at compile time. However, in batch scenarios, the file name may need to be determined at runtime as a parameter to the job. This could be solved using '-D' parameters, i.e. a system property:

```

<bean id="flatFileItemReader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="resource" value="\${input.file.name}" />
</bean>

```

All that would be required for this solution to work would be a system argument (`-Dinput.file.name="file://file.txt"`). (Note that although a `PropertyPlaceholderConfigurer` can be used here, it is not necessary if the system property is always set because the `ResourceEditor` in Spring already filters and does placeholder replacement on system properties.)

Often in a batch setting it is preferable to parameterize the file name in the `JobParameters` of the job, instead of through system properties, and access them that way. To allow for this, Spring Batch provides the `StepExecutionResourceProxy`. The proxy can use either job name, step name, or any values from the `JobParameters`, by surrounding them with `%`:

```

<bean id="inputFile"
      class="org.springframework.batch.core.resource.StepExecutionResourceProxy" />
  <property name="filePattern" value="//%JOB_NAME%/%STEP_NAME%/%file.name%" />
</bean>

```

Assuming a job name of 'fooJob', and a step name of 'fooStep', and the key-value pair of 'file.name="fileName.txt"' is in the `JobParameters` the job is started with, the following filename will be passed

as the Resource: `"/fooJob/fooStep/fileName.txt"`. It should be noted that in order for the proxy to have access to the `StepExecution`, it must be registered as a `StepListener`:

```
<bean id="fooStep" parent="abstractStep"
  p:itemReader-ref="itemReader"
  p:itemWriter-ref="itemWriter">
  <property name="listeners" ref="inputFile" />
</bean>
```

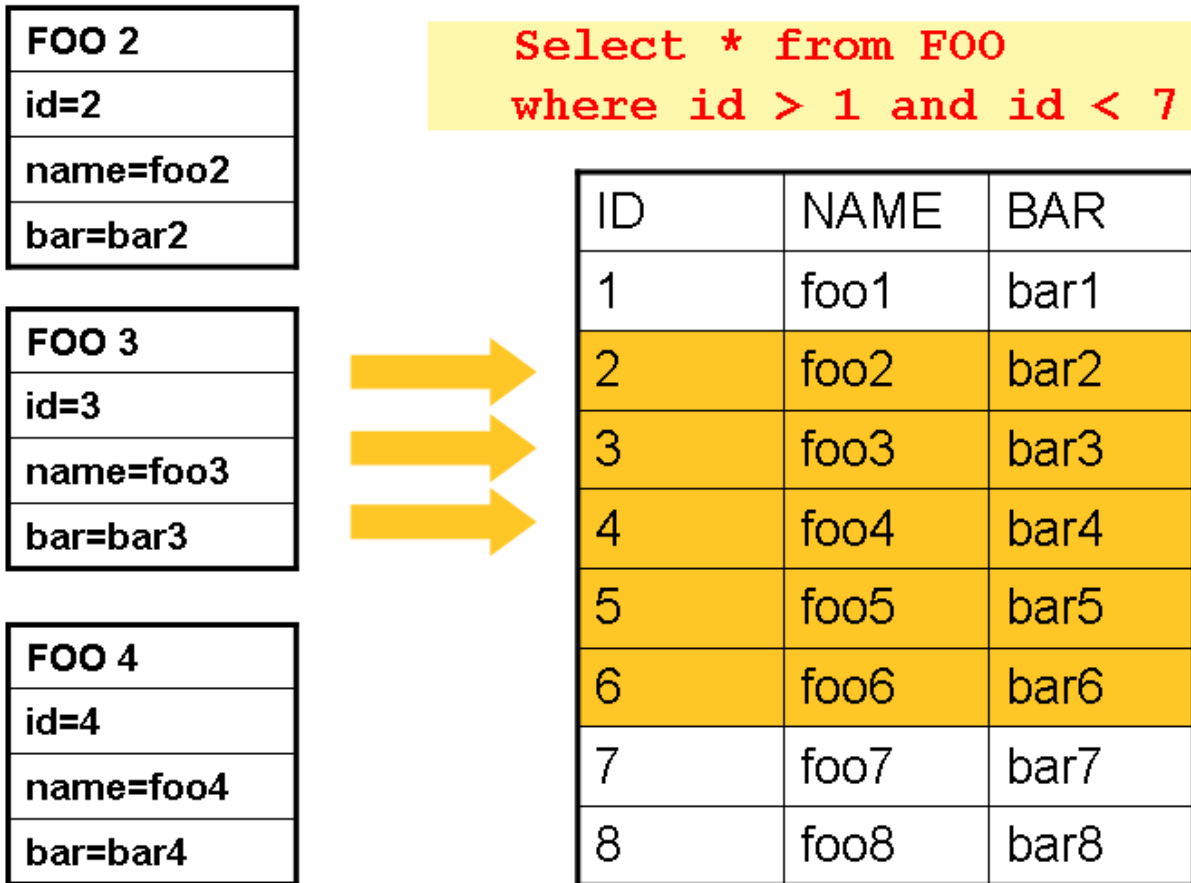
The `StepListener` interface will be discussed in more detail in Chapter 4. For now, it is sufficient to know that the proxy must be registered.

3.8. 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 that must be worked with. The Spring Core `JdbcTemplate` illustrates this problem well. If you use `JdbcTemplate` with a `RowMapper`, the `RowMapper` will be called once for every result returned from the provided query. This causes few issues in scenarios where the dataset is small, but the large datasets often necessary for batch processing would cause any JVM to crash quickly. If the sql statement returns 1 million rows, the `RowMapper` will be called 1 million times. Spring Batch provides two types of solutions for this problem: `Cursor` and `DrivingQuery` `ItemReaders`.

3.8.1. Cursor Based ItemReaders

Using a database cursor is generally the default approach of most batch developers. This is because it is the database's solution to the problem of 'streaming' relational data. The Java `ResultSet` class is essentially an object orientated 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. Spring Batch cursor based `ItemReaders` open the a cursor on initialization, and move the cursor forward one row for every call to `read`, returning a mapped object that can be used for processing. The `close` method will then be 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. Below is a generic diagram of how a cursor based `ItemReader` works, and while a SQL statement is used as an example since it is so widely known, any technology could implement the basic approach:



The 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 one 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.

3.8.1.1. JdbcCursorItemReader

JdbcCursorItemReader is the JDBC implementation of the cursor based technique. It works directly with a ResultSet and requires a SQL statement to run against a connection obtained from a DataSource. The following database schema will be 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 we'll use an implementation of the RowMapper interface to map a CustomerCredit object:

```
public class CustomerCreditRowMapper implements RowMapper {

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

  public Object 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 `JdbcTemplate` is so familiar to users of Spring, and the `JdbcCursorItemReader` shares key interfaces with it, it's useful to see an example of how to read in this data with `JdbcTemplate`, in order to contrast it with the item reader. For the purposes of this example, let's assume there are 1,000 rows in the `CUSTOMER` database. The first example will be using `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 this code snippet the `customerCredits` list will contain 1,000 `CustomerCredit` objects. In the query method, a connection will be obtained from the `DataSource`, the provided SQL will be run against it, and the `mapRow` method will be called for each row in the `ResultSet`. Let's contrast this with the approach of the `JdbcCursorItemReader`:

```
JdbcCursorItemReader itemReader = new JdbcCursorItemReader();
itemReader.setDataSource(dataSource);
itemReader.setSql("SELECT ID, NAME, CREDIT from CUSTOMER");
itemReader.setMapper(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(executionContext);
```

After running this code snippet the counter will equal 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, and the item written out via an `ItemWriter`, and then the next item obtained via `read`. This allows item reading and writing to be done in 'chunks' and committed periodically, which is the essence of high performance batch processing.

3.8.1.1.1. 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:

Table 3.2. JdbcCursorItemReader Properties

<code>ignoreWarnings</code>	Determines whether or not <code>SQLWarnings</code> are logged or cause an exception - default is true
<code>fetchSize</code>	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 <code>ResultSet</code> object used by the <code>ItemReader</code> . By default, no hint is given.
<code>maxRows</code>	Sets the limits for the maximum number of rows the underlying <code>ResultSet</code> can hold at any one time.
<code>queryTimeout</code>	Sets the number of seconds the driver will wait for a

	Statement object to execute to the given number of seconds. If the limit is exceeded, a <code>DataSourceException</code> is thrown. (consult your driver vendor documentation for details).
<code>verifyCursorPosition</code>	Because the same <code>ResultSet</code> held by the <code>ItemReader</code> is passed to the <code>RowMapper</code> , it's possible for users to call <code>ResultSet.next()</code> themselves, which could cause issues with the reader's internal count. Setting this value to true will cause an exception to be thrown if the cursor position is not the same after the <code>RowMapper</code> call as it was before.
<code>saveState</code>	Indicates whether or not the reader's state should be saved in the <code>ExecutionContext</code> provided by <code>ItemStream#update(ExecutionContext)</code> . The default value is false.

3.8.1.2. HibernateCursorItemReader

Just as normal Spring users make important decisions about whether or not to use ORM solutions, which affects 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 doesn't mean it can't 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 that can cause issues when using it 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` allows you to 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`. Below is an example configuration using 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(executionContext);

```

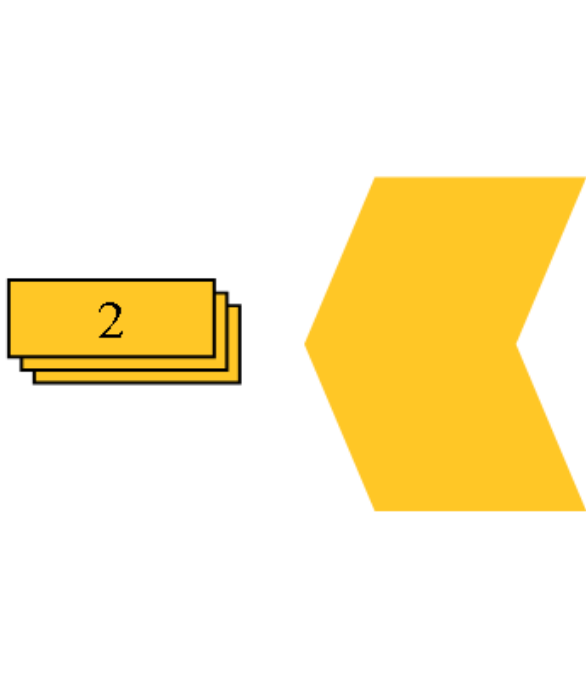
This configured `ItemReader` will return `CustomerCredit` objects in the exact same manner as described by the `JdbcCursorItemReader`, assuming hibernate mapping files have been created correctly for the `Customer` table. The 'useStatelessSession' property defaults to true, but has been added here to draw attention to the ability to switch it on or off.

3.8.2. Driving Query Based ItemReaders

In the previous section, Cursor based database input was discussed. However, it isn't the only option. Many

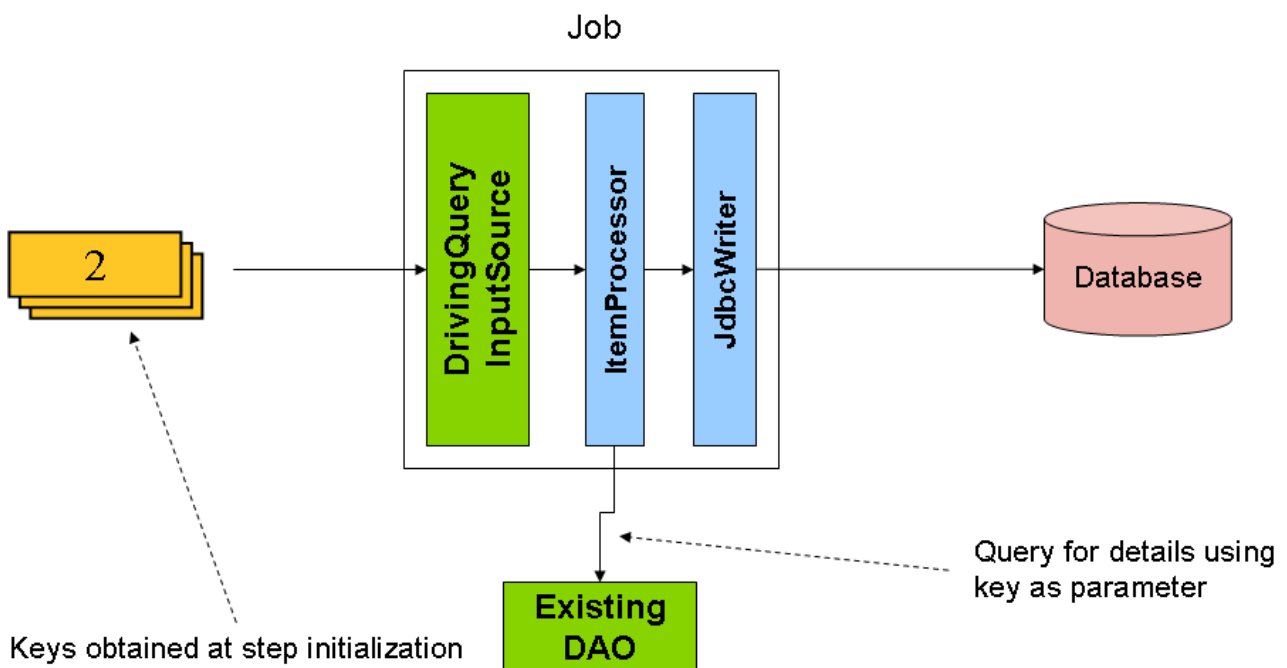
database vendors, such as DB2, have extremely pessimistic locking strategies that can cause issues if the table being read also needs to be used by other portions of the online application. Furthermore, opening cursors over extremely large datasets can cause issues on certain vendors. Therefore, many 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 example illustrates:

```
Select ID from FOO
where id > 1 and id < 7
```



ID	NAME	BAR
1	foo1	bar1
2	foo2	bar2
3	foo3	bar3
4	foo4	bar4
5	foo5	bar5
6	foo6	bar6
7	foo7	bar7
8	foo8	bar8

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



As you can see, an existing DAO can be used to obtain a full 'Foo' object using the key obtained from the driving query. In Spring Batch, driving query style input is implemented with a `DrivingQueryItemReader`, which has only one dependency: a `KeyCollector`

3.8.2.1. KeyCollector

As the previous example illustrates, the `DrivingQueryItemReader` is fairly simple. It simply iterates over a list of keys. However, the real complication is how those keys are obtained. The `KeyCollector` interface abstracts this:

```
public interface KeyCollector {
    List retrieveKeys(ExecutionContext executionContext);
    void updateContext(Object key, ExecutionContext executionContext);
}
```

The primary method in this interface is the `retrieveKeys` method. It is expected that this method will return the keys to be processed regardless of whether or not it is a restart scenario. For example, if a job starts processing keys 1 through 1,000, and fails after processing key 500, upon restarting keys 500 through 1,000 should be returned. This functionality is made possible by the `updateContext` method, which saves the provided key (which should be the current key being processed) in the provided `ExecutionContext`. The `retrieveKeys` method can then use this value to retrieve a subset of the original keys:

```
ExecutionContext executionContext = new ExecutionContext();
List keys = keyStrategy.retrieveKeys(executionContext);
//Assume keys contains 1 through 1,000
keyStrategy.updateContext(new Long(500), executionContext);
keys = keyStrategy.retrieveKeys(executionContext);
//keys should now contains 500 through 1,000
```

This generalization illustrates the `KeyCollector` contract. If we assume that initially calling `retrieveKeys` returned 1,000 keys (1 through 1,000), calling `updateContext` with key 500 should mean that calling `retrieveKeys` again with the same `ExecutionContext` will return 500 keys (501 through 1,000).

3.8.2.2. SingleColumnJdbcKeyCollector

The most common driving query scenario is that of input that has only one column that represents its key. This is implemented as the `SingleColumnJdbcKeyCollector` class, which has the following options:

Table 3.3. SinglecolumnJdbcKeyCollector properties

<code>jdbcTemplate</code>	The <code>JdbcTemplate</code> to be used to query the database
<code>sql</code>	The sql statement to query the database with. It should return only one value.
<code>restartSql</code>	The sql statement to use in the case of restart. Because only one key will be used, this query should require only one argument.
<code>keyMapper</code>	The <code>RowMapper</code> implementation to be used to map the keys to objects. By default, this is a Spring Core <code>SingleColumnRowMapper</code> , which maps them to well known types such as <code>Integer</code> , <code>String</code> , etc. For more information, check the documentation of your

	specific Spring release.
--	--------------------------

The following code helps illustrate how to setup and use a `SingleColumnJdbcKeyCollector`:

```
SingleColumnJdbcKeyCollector keyCollector = new SingleColumnJdbcKeyCollector(getJdbcTemplate(),
"SELECT ID from T_FOOS order by ID");

keyCollector.setRestartSql("SELECT ID from T_FOOS where ID > ? order by ID");

ExecutionContext executionContext = new ExecutionContext();

List keys = keyStrategy.retrieveKeys(new ExecutionContext());

for (int i = 0; i < keys.size(); i++) {
    System.out.println(keys.get(i));
}
```

If this code were run in the proper environment with the correct database tables setup, then it would output the following:

```
1
2
3
4
5
```

Now, let's modify the code slightly to show what would happen if the code were started again after a restart, having failed after processing key 3 successfully:

```
SingleColumnJdbcKeyCollector keyCollector = new SingleColumnJdbcKeyCollector(getJdbcTemplate(),
"SELECT ID from T_FOOS order by ID");

keyCollector.setRestartSql("SELECT ID from T_FOOS where ID > ? order by ID");

ExecutionContext executionContext = new ExecutionContext();

keyStrategy.updateContext(new Long(3), executionContext);

List keys = keyStrategy.retrieveKeys(executionContext);

for (int i = 0; i < keys.size(); i++) {
    System.out.println(keys.get(i));
}
```

Running this code snippet would result in the following:

```
4
5
```

The key difference between the two examples is the following line:

```
keyStrategy.updateContext(new Long(3), executionContext);
```

This tells the key collector to update the provided `ExecutionContext` with the key of three. This will normally be called by the `DrivingQueryItemReader`, but is called directly for simplicities sake. By calling `retrieveKeys` with the `ExecutionContext` that was updated to contain 3, the argument of 3 will be passed to the `restartSql`:

```
keyCollector.setRestartSql("SELECT ID from T_FOOS where ID > ? order by ID");
```

This will cause only keys 4 and 5 to be returned, since they are the only ones with an ID greater than 3.

3.8.2.3. Mapping multiple column keys

The `SingleColumnJdbcKeyCollector` is extremely useful for generating keys, but only if one column uniquely identifies your record. What if more than one column is required to be able to uniquely identify your record? This should be a minority scenario, but it is still possible. In this case, the `MultipleColumnJdbcKeyCollector` should be used. It allows for mapping multiple columns by sacrificing simplicity. The properties needed to use the multiple column collector are the same as the single column version except one difference: instead of a regular `RowMapper`, an `ExecutionContextRowMapper` must be provided. Just like the single column version, it requires a normal SQL statement and a restart SQL statement. However, because the restart SQL statement will require more than one argument, there needs to be more complex handling of how keys are mapped to an execution context. An `ExecutionContextRowMapper` provides this:

```
public interface ExecutionContextRowMapper extends RowMapper {
    public void mapKeys(Object key, ExecutionContext executionContext);
    public PreparedStatementSetter createSetter(ExecutionContext executionContext);
}
```

The `ExecutionContextRowMapper` interface extends the standard `RowMapper` interface to allow for multiple keys to be stored in an `ExecutionContext`, and a `PreparedStatementSetter` be created so that arguments to a the restart SQL statement can be set for the key returned.

By default a implementation of the `ExecutionContextRowMapper` that uses a `Map` will be used. It is recommended that this implementation not be overridden. However, if a specific type of key needs to be returned, then a new implementation can be provided.

3.8.2.4. iBatisKeyCollector

Jdbc is not the only option available for key collectors, iBatis can be used as well. The usage of iBatis doesn't change the basic requirements of a `KeyCollector`: query, restart query, and `DataSource`. However, because iBatis is used, both queries are simply iBatis query ids, and the data source is a `SqlMapClient`.

3.8.3. Database ItemWriters

While both Flat Files and XML have specific `ItemWriters`, there is no exact equivalent in the database world. This is because transactions provide all the functionality that is needed. `ItemWriters` 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 DAO's 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. The one exception to this is buffered output. This is most common when using hibernate as an `ItemWriter`, but could have the same issues when using Jdbc batch mode. Buffering database output doesn't have any inherent flaws, assuming there are no errors in the data. However, any errors while writing out can cause issues because there is no way to know which individual item caused an exception. An example would be a record that causes a `DataIntegrityViolationException`, perhaps because of a primary key violation. If items are buffered before being written out, this error will not be thrown until the buffer is flushed just before a commit. For example, let's assume that 20 items will be written per chunk, and the 15th item throws a `DataIntegrityViolationException`. As far as the `Step` is concerned, all 20 item will be written out successfully, since there's no way to know that an error will occur until they are actually written out. Once `ItemWriter#flush()` is called, the buffer will be emptied and the exception will be hit. At this point, there's nothing the `Step` can do, the transaction must be rolled back. Normally, this exception will cause the `Item` to be skipped (depending upon the skip/retry policies), and then it won't be written out again. However, in this

scenario, there's no way for it to know which item caused the issue, the whole buffer was being written out when the failure happened. Because this is a common enough use case, especially when using Hibernate, Spring Batch provides an implementation to help: `HibernateAwareItemWriter`. The `HibernateAwareItemWriter` solves the problem in a straightforward way: if a chunk fails the first time, on subsequent runs it will be flushed after after each time. This effectively lowers the commit interval to one for the length of the chunk. Doing so allows for items to be skipped reliably. The following example illustrates how to configure the `HibernateAwareItemWriter`:

```
<bean id="hibernateItemWriter"
      class="org.springframework.batch.item.database.HibernateAwareItemWriter">
  <property name="sessionFactory" ref="sessionFactory" />
  <property name="delegate" ref="customerCreditWriter" />
</bean>

<bean id="customerCreditWriter"
      class="org.springframework.batch.sample.dao.HibernateCreditDao">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

3.9. 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's fairly trivial to write an adaptor class for each service that needs wrapping, but because it's such a common concern, Spring Batch provides implementations: `ItemReaderAdapter` and `ItemWriterAdapter`. Both classes implement the standard Spring method invoking delegator pattern and are fairly simple to set up. Below is an example of the reader:

```
<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" />
```

One important point to note is that the contract of the `targetMethod` must be the same as the contract for `read`. That is, when exhausted it will return null, otherwise an `Object`. Anything else will prevent the framework from correctly knowing when processing should end, either causing an infinite loop or incorrect failure, depending upon the implementation of the `ItemWriter`. The `ItemWriter` implementation is equally as simple:

```
<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" />
```

3.10. Item Transforming

The `ItemReader` and `ItemWriter` interfaces have been discussed in detail in this chapter, but what if you want to insert business logic before writing? One option for both reading and writing is to use the composite pattern.

That is, create an `ItemWriter` that contains another `ItemWriter`, or an `ItemReader` that contains another `ItemReader`. For example:

```
public class CompositeItemWriter implements ItemWriter {

    ItemWriter itemWriter;

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

    public void write(Object item) throws Exception {

        //Add business logic here

        itemWriter.write(item);
    }

    public void clear() throws ClearFailedException {
        itemWriter.clear();
    }

    public void flush() throws FlushFailedException {
        itemWriter.flush();
    }
}
```

The class above contains another `ItemWriter` that it delegates to after having provided some business logic. It should be noted that the `clear` and `flush` methods must be propagated as well so that the delegate `ItemWriter` is notified. 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`. This pattern is very 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 actual written, there isn't much need to call `write` yourself, you just want to modify the item. For this scenario, Spring Batch provides the `ItemTransformer` interface:

```
public interface ItemTransformer {

    Object transform(Object item) throws Exception;
}
```

An `ItemTransformer` is very simple, given one object, transform it and return another. The object provided may or may not be of the same type. The point is that business logic may be applied within `transform`, and is completely up to the developer to create. An `ItemTransformer` is used as part of the `ItemTransformerItemWriter`, which accepts an `ItemWriter` and an `ItemTransformer`, passing the item first to the transformer, before writing it. For example, assuming an `ItemReader` provides a class of type `Foo`, and it needs to be converted to type `Bar` before being written out. An `ItemTransformer` can be written that performs the conversion:

```
public class Foo {}

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

public class FooTransformer implements ItemTransformer{

    //Perform simple transformation, convert a Foo to a Bar
    public Object transform(Object item) throws Exception {
        assertTrue(item instanceof Foo);
        Foo foo = (Foo)item;
        return new Bar(foo);
    }
}

public class BarWriter implements ItemWriter{

    public void write(Object item) throws Exception {
        assertTrue(item instanceof Bar);
    }
}
```



```

    }

    //rest of class omitted for clarity
}

```

In the very simple example above, there is a class `Foo`, a class `Bar`, and a class `FooTransformer` that adheres to the `ItemTransformer` interface. The transformation is simple, but any type of transformation could be done here. The `BarWriter` will be used to write out 'Bars', throwing an exception if any other type is provided. Similarly, the `FooTransformer` will throw an exception if anything but a `Foo` is provided. An `ItemTransformerItemWriter` can then be used like a normal `ItemWriter`. It will be passed a `Foo` for writing, which will be passed to the transformer, and a `Bar` returned. The resulting `Bar` will then be written:

```

ItemTransformerItemWriter itemTransformerItemWriter = new ItemTransformerItemWriter();
itemTransformerItemWriter.setItemTransformer(new FooTransformer());
itemTransformerItemWriter.setDelegate(new BarWriter());
itemTransformerItemWriter.write(new Foo());

```

3.10.1. The Delegate Pattern and Registering with the Step

Note that the `ItemTransformerItemWriter` and the `CompositeItemWriter` are examples of a delegation pattern, which is common in Spring Batch. The delegates themselves might implement callback interfaces like `ItemStream` or `StepListener`. If they do, and 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`. Registration is automatic when using the factory beans (`*StepFactoryBean`), but only for the `ItemReader` and `ItemWriter` injected directly. The delegates are not known to the `Step`, so they need to be injected as listeners or streams (or both if appropriate).

3.10.2. Chaining ItemTransformers

Performing a single transformation is useful in many scenarios, but what if you want to 'chain' together multiple `ItemTransformers`? This can be accomplished using a `CompositeItemTransformer`. To update the previous, single transformation, example, `Foo` will be transformed to `Bar`, which will be transformed to `Foobar` and written out:

```

public class Foo {}

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

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

public class FooTransformer implements ItemTransformer{

    //Perform simple transformation, convert a Foo to a Bar
    public Object transform(Object item) throws Exception {
        assertTrue(item instanceof Foo);
        Foo foo = (Foo)item;
        return new Bar(foo);
    }
}

public class BarTransformer implements ItemTransformer{

    public Object transform(Object item) throws Exception {
        assertTrue(item instanceof Bar);
        return new Foobar((Bar)item);
    }
}

```

```
public class FoobarWriter implements ItemWriter{

    public void write(Object item) throws Exception {
        assertTrue(item instanceof Foobar);
    }

    //rest of class ommitted for clarity
}
```

A `FooTransformer` and `BarTransformer` can be 'chained' together to give the resultant `Foobar`:

```
CompositeItemTransformer compositeTransformer = new CompositeItemTransformer();
List itemTransformers = new ArrayList();
itemTransformers.add(new FooTransformer());
itemTransformers.add(new BarTransformer());
compositeTransformer.setItemTransformers(itemTransformers);
```

The `compositeTransformer` could be said to accept a `Foo` and return a `Foobar`. Clients of the composite transformer don't need to know that there are actually two separate transformations taking place. By updating the example from above to use the composite transformer, the correct class can be passed to `FoobarWriter`:

```
ItemTransformerItemWriter itemTransformerItemWriter = new ItemTransformerItemWriter();
itemTransformerItemWriter.setItemTransformer(compositeTransformer);
itemTransformerItemWriter.setDelegate(new FoobarWriter());
itemTransformerItemWriter.write(new Foo());
```

3.11. Validating Input

During the course of this chapter, multiple approaches to parsing input have been discussed. Each major implementation will throw exception if it is not 'well-formed'. The `FixedLengthTokenizer` will throw an exception if a range of data is missing. Similarly, attempting to access an index in a `RowMapper` or `FieldSetMapper` that doesn't exist or is in a different format than the one expected will cause an exception to be thrown. All of these types of exceptions will be thrown before `read` returns. However, they don't 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 will parse correctly, because it existed and is a number, but it won't cause an exception. Since there are already a plethora of Validation frameworks, Spring Batch does not attempt to provide yet another, but rather provides a very simple interface that can be implemented by any number of frameworks:

```
public interface Validator {

    void validate(Object value) throws ValidationException;

}
```

The contract is that the `validate` method will throw an exception if the object is invalid, and return normally if it is valid. Spring Batch provides an out of the box `ItemReader` that delegates to another `ItemReader` and validates the returned item:

```
<bean class="org.springframework.batch.item.validator.ValidatingItemReader">
    <property name="itemReader">
        <bean class="org.springframework.batch.sample.item.reader.OrderItemReader" />
    </property>
    <property name="validator" ref="validator" />
</bean>

<bean id="validator"
    class="org.springframework.batch.item.validator.SpringValidator">
    <property name="validator">
        <bean id="orderValidator"
            class="org.springframework.validation.valang.ValangValidator">
```

```

<property name="valang">
  <value>
    <![CDATA[
      { orderId : ? > 0 AND ? <= 9999999999 : 'Incorrect order ID' : 'error.order.id' }
      { totalLines : ? = size(lineItems) : 'Bad count of order lines' : 'error.order.lines.badcount'}
      { customer.registered : customer.businessCustomer = FALSE OR ? = TRUE : 'Business customer must be
      { customer.companyName : customer.businessCustomer = FALSE OR ? HAS TEXT : 'Company name for busin
    ]]>
  </value>
</property>
</bean>
</property>
</bean>

```

This simple example shows a simple `ValangValidator` that is used to validate an order object. The intent is not to show Valang functionality as much as to show how a validator could be added.

3.11.1. The Delegate Pattern and Registering with the Step

Note that the `ValidatingItemReader` is another example of a delegation pattern, and the delegates themselves might implement callback interfaces like `ItemStream` or `StepListener`. If they do, and 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`. Registration is automatic when using the factory beans (`*StepFactoryBean`), but only for the `ItemReader` and `ItemWriter` injected directly - the delegates are not known to the step, so they need to be injected as listeners or streams (or both if appropriate).

3.12. Creating Custom ItemReaders and ItemWriters

So far in this chapter the basic contracts that exist for reading and writing in Spring Batch and some common implementations have been discussed. 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 will show, using a simple example, how to create a custom `ItemReader` and `ItemWriter` implementation and implement their contracts correctly. The `ItemReader` will also implement `ItemStream`, in order to illustrate how to make a reader or writer restartable.

3.12.1. Custom ItemReader Example

For the purpose of this example, a simple `ItemReader` implementation that reads from a provided list will be created. We'll start out by implementing the most basic contract of `ItemReader`, `read`:

```

public class CustomItemReader implements ItemReader{

    List items;

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

    public Object read() throws Exception, UnexpectedInputException,
        NoWorkFoundException, ParseException {

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

    public void mark() throws MarkFailedException { };

    public void reset() throws ResetFailedException { };
}

```

This very simple class takes a list of items, and returns one at a time, removing it from the list. When the list empty, it returns null, thus satisfying the most basic requirements of an `ItemReader`, as illustrated below:

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

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

3.12.1.1. Making the `ItemReader` transactional

This most basic `ItemReader` will work, but what happens if the transaction needs to be rolled back? This will usually be caused by an error in the `ItemWriter`, since the `ItemReader` generally won't do anything that invalidates the transaction, but without supporting it, there would be erroneous results. `ItemReaders` are notified about rollbacks via the `mark` and `reset` methods. In the example above they're empty, but we'll need to add code to them in order to support the rollback scenario:

```
public class CustomItemReader implements ItemReader{

    List items;
    int currentIndex = 0;
    int lastMarkedIndex = 0;

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

    public Object read() throws Exception, UnexpectedInputException,
        NoWorkFoundException, ParseException {

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

    public void mark() throws MarkFailedException {
        lastMarkedIndex = currentIndex;
    };

    public void reset() throws ResetFailedException {
        currentIndex = lastMarkedIndex;
    };
}
```

The `CustomItemReader` has now been modified to keep track of where it is currently, and where it was when `mark()` was last called. This allows the new `ItemReader` to fulfill the basic contract that calling `reset` returns the `ItemReader` to the state it was in when `mark` was last called:

```
//Assume same setup as last example, a list with "1", "2", and "3"
itemReader.mark();
assertEquals("1", itemReader.read());
assertEquals("2", itemReader.read());
itemReader.reset();
assertEquals("1", itemReader.read());
```

In most real world scenarios, there will likely be some kind of underlying resource that will require tracking. In the case of a file, `mark` will hold the current location within the file, and `reset` will move it back. The `JdbcCursorItemReader`, for example, holds on to the current row number, and on `reset` moves the cursor back

by calling the `ResultSet absolute` method, which moves the current cursor to the row number supplied. The `CustomItemReader` now completely adheres to the entire `ItemReader` contract. `read` will return the appropriate items, returning null when empty, and `reset` returns the `ItemReader` back to its state as of the last call to `mark`, allowing for correct support of a rollback. (It's assumed a `Step` implementation will call `mark` and `reset`).

3.12.1.2. Making the `ItemReader` restartable

The final challenge now is to make the `ItemReader` restartable. Currently, if the power goes out, and processing 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 starts off at 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 and reconstitute its last known state on restart. For this reason, we recommend that you keep custom readers stateless as far as possible, so you don't have to worry about restartability.

If you do need to store state, then in Spring Batch, this is implemented with the `ItemStream` interface:

```
public class CustomItemReader implements ItemReader, ItemStream{

    List items;
    int currentIndex = 0;
    int lastMarkedIndex = 0;
    private static String CURRENT_INDEX = "current.index";

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

    public Object read() throws Exception, UnexpectedInputException,
        NoWorkFoundException, ParseException {

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

    public void mark() throws MarkFailedException {
        lastMarkedIndex = currentIndex;
    };

    public void reset() throws ResetFailedException {
        currentIndex = lastMarkedIndex;
    }

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

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

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

On each call to `ItemStream update` method, the current index of the `ItemReader` will be 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, and if so the current index is moved to that location. This is a fairly trivial example, but it still meets the general contract:

```
ExecutionContext executionContext = new ExecutionContext();
((ItemStream)itemReader).open(executionContext);
```

```

assertEquals("1", itemReader.read());
((ItemStream)itemReader).update(executionContext);

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

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

```

Most `ItemReaders` have much more sophisticated restart logic. The `DrivingQueryItemReader`, for example, only loads up the remaining keys to be processed, rather than loading all of them and then moving to the correct index.

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 need for output) then a more unique name will be needed. For this reason, many of the Spring Batch `ItemReader` and `ItemWriters` have a `setName()` property that allows this key name to be overridden.

3.12.2. 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 won't be covered in this example. As with the `ItemReader` example, a `List` will be used in order to keep the example as simple as possible:

```

public class CustomItemWriter implements ItemWriter{

    List output = new ArrayList();

    public void write(Object item) throws Exception {
        output.add(item);
    }

    public void clear() throws ClearFailedException { }

    public void flush() throws FlushFailedException { }
}

```

3.12.2.1. Making the `ItemReader` transactional

The example is extremely simple, but it's worth showing to illustrate an `ItemWriter` that doesn't respond to rollbacks and commits (i.e. `clear` and `flush`). If your potential writer is such that it doesn't need to care about rollback or commit, likely because it's writing to a database, then there is little value to the `ItemWriter` interface in that scenario other than using it to meet another class's requirement for an implementation of the `ItemWriter` interface. In that case, the `ItemWriterAdapter` would be a better solution. However, if it does need to be transactional, then `flush` and `clear` should be implemented to allow for a buffering solution:

```

public class CustomItemWriter implements ItemWriter{

    List output = new ArrayList();
    List buffer = new ArrayList();

    public void write(Object item) throws Exception {
        buffer.add(item);
    }

    public void clear() throws ClearFailedException {

```

```
    buffer.clear();
}

public void flush() throws FlushFailedException {
    for(Iterator it = buffer.iterator(); it.hasNext();){
        output.add(it.next());
        it.remove();
    }
}
```

The `ItemWriter` buffers all output, only writing to the actual output (in this case by added to a list) when the `ItemWriter flush()` method is called. The contents of the buffer are thrown away when `ItemWriter clear()` is called.

3.12.2.2. Making the `ItemWriter` restartable

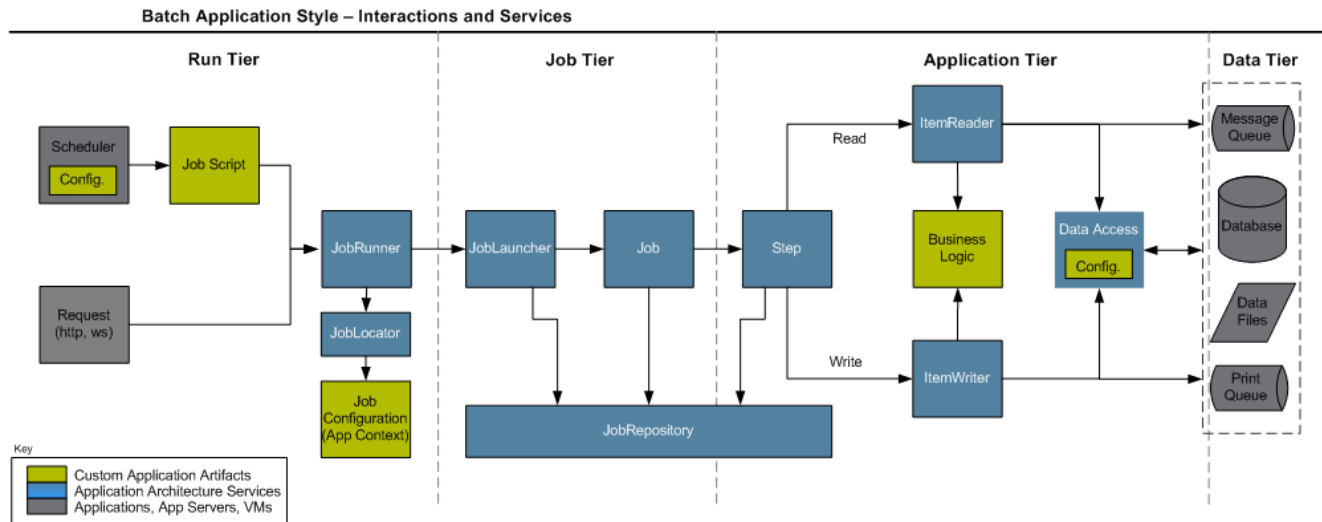
To make the `ItemWriter` restartable we would follow the same process as for the `ItemReader`, adding and implementing the `ItemStream` interface to synchronize the execution context. In the example we might have to count the number of items processed and add that as a footer record. If we needed to 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 (e.g. when writing to a file), or else it writes to a transactional resource so doesn't need to be restartable because it is stateless. When you have a stateful writer you should probably also 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 with a factory bean (e.g. one of the `StepFactoryBean` implementations in Spring Batch Core).

Chapter 4. Configuring and Executing A Job

4.1. Introduction

In Chapter 2, the overall architecture design was discussed, using the following diagram as a guide:



When viewed from left to right, the diagram describes a basic flow for the execution of a batch job:

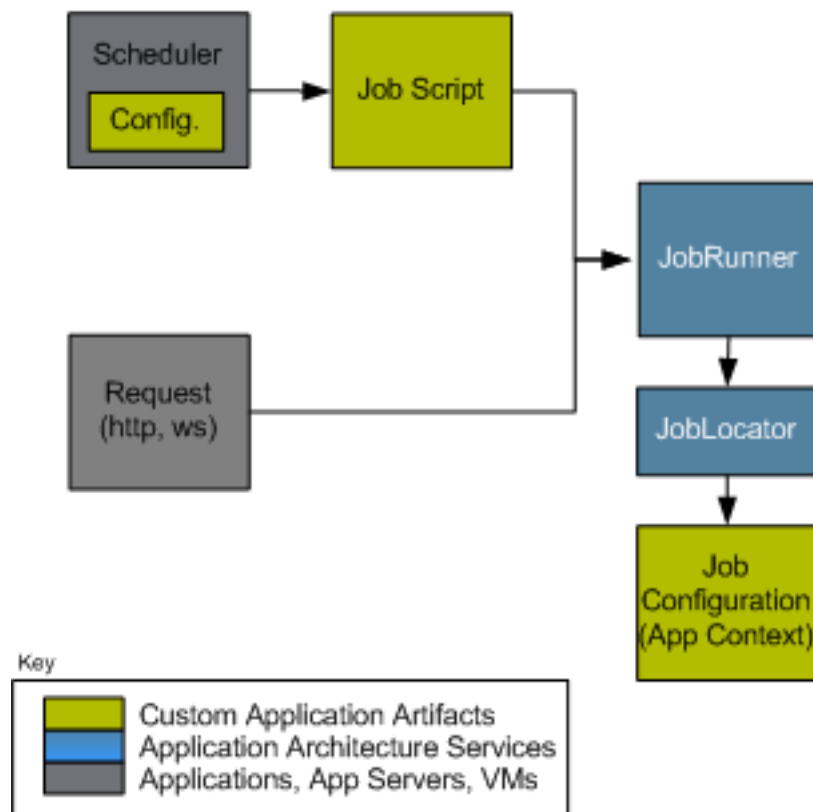
1. A Scheduler kicks off a job script (usually some form of shell script)
2. The script sets up the classpath appropriately, and starts the Java process. In most cases, using `CommandLineJobRunner` as the entry point
3. The `JobRunner` finds the `Job` using the `JobLocator`, pulls together the `JobParameters` and launches the `Job`
4. The `JobLauncher` retrieves a `JobExecution` from the `JobRepository`, and executes the `Job`
5. The `Job` executes each `Step` in sequence.
6. The `Step` calls `read` on the `ItemReader`, handing the resulting item to the `ItemWriter` until null is returned, periodically committing and storing status in the `JobRepository`.
7. When execution is complete, the `Step` returns control back to the `Job`, and if no more steps exist, control is returned back to the original caller, in this case, the scheduler.

This flow is perhaps a bit overly simplified, but describes the complete flow in the most basic terms. From here, each tier will be described in detail, using actual implementations and examples.

4.2. Run Tier

As its name suggests, this tier is entirely concerned with actually running the job. Regardless of whether the originator is a Scheduler or an HTTP request, a `Job` must be obtained, parameters must be parsed, and eventually a `JobLauncher` called:

Run Tier



4.2.1. Running Jobs from the Command Line

For users that want to run their jobs from an enterprise scheduler, the command line is the primary interface. This is because most schedulers (with the exception of Quartz unless using the `NATIVEJOB`) work directly with operating system processes, primarily kicked off with shell scripts. There are many ways to launch a Java process besides a shell script, such as Perl, Ruby, or even 'build tools' such as ant or maven. However, because most people are familiar with shell scripts, this example will focus on them.

4.2.1.1. The CommandLineJobRunner

Because the script launching the job must kick off a Java Virtual Machine, there needs to be a class with a main method to act as the primary entry point. Spring Batch provides an implementation that serves just this purpose: `CommandLineJobRunner`. It's important to note that this is just one way to bootstrap your application, but there are many ways to launch a Java process, and this class should in no way be viewed as definitive. It performs four tasks:

- Loads the appropriate Application Context
- Parses command line arguments into `JobParameters`
- Locates the appropriate job based on arguments
- Uses the `JobLauncher` provided in the application context to launch the job.

All of these tasks are accomplished based completely upon the arguments passed in. The following are required arguments:

Table 4.1. CommandLineJobRunner arguments

jobPath	The location of the XML file that will be used to create an <code>ApplicationContext</code> . This file should contain everything needed to run the complete <code>Job</code>
jobName	The name of the job to be run.

These arguments must be passed in with the path first and the name second. All arguments after these are considered to be `JobParameters` and must be in the format of 'name=value':

```
bash$ java CommandLineJobRunner endOfDayJob.xml endOfDay schedule.date(date)=2008/01/01
```

In most cases you would want to use a manifest to declare your main class in a jar, but for simplicity, the class was used directly. This example is using the same 'EndOfDay' example from Chapter 2. The first argument is 'endOfDayJob.xml', which is the Spring `ApplicationContext` containing the `Job`. The second argument, 'endOfDay' represents the job name. The final argument, 'schedule.date=01-01-2008' will be converted into `JobParameters`. An example of the XML configuration is below:

```
<bean id="endOfDay"
    class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">
    <bean id="step1" parent="simpleStep" />
    <!-- Step details removed for clarity -->
  </property>
</bean>

<!-- Launcher details removed for clarity -->
<bean id="jobLauncher"
    class="org.springframework.batch.core.launch.support.SimpleJobLauncher" />
```

This example is overly simplistic, since there are many more requirements to a run a batch job in Spring Batch in general, but it serves to show the two main requirements of the `CommandLineJobRunner`: `Job` and `JobLauncher`

4.2.1.2. ExitCodes

When launching a batch job from the command-line, it is often from an enterprise scheduler. Most schedulers are fairly dumb, and work only at the process level. Meaning, they only know about some operating system process such as a shell script that they're invoking. In this scenario, the only way to communicate back to the scheduler about the success or failure of a job is through return codes. A number is returned to a scheduler that is told how to interpret the result. In the simple case: 0 is success and 1 is failure. However, there may be scenarios such as: If job A returns 4 kick off job B, if it returns 5 kick off job C. This type of behavior is configured at the scheduler level, but it is important that a processing framework such as Spring Batch provide a way to return a numeric representation of of the 'Exit Code' for a particular batch job. In Spring Batch this is encapsulated within an `ExitStatus`, which is covered in more detail in Chapter 5. For the purposes of discussing exit codes, the only important thing to know is that an `ExitStatus` has an exit code property that is set by the framework (or the developer) and is returned as part of the `JobExecution` returned from the `JobLauncher`. The `CommandLineJobRunner` converts this string value to a number using the `ExitCodeMapper` interface:

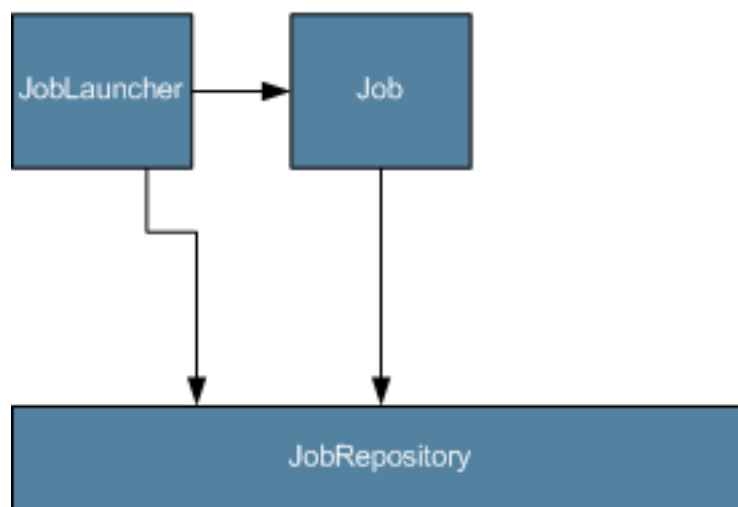
```
public interface ExitCodeMapper {
    public int intValue(String exitCode);
}
```

The essential contract of an `ExitCodeMapper` is that, given a string exit code, a number representation will be returned. The default implementation used by the job runner is the `SimpleJvmExitCodeMapper` that returns 0 for completion, 1 for generic errors, and 2 for any job runner errors such as not being able to find a `Job` in the provided context. If anything more complex than the 3 values above is needed, then a custom implementation of the `ExitCodeMapper` interface must be supplied. Because the `CommandLineJobRunner` is the class that creates an `ApplicationContext`, and thus cannot be 'wired together', any values that need to be overwritten must be autowired. This means that if an implementation of `ExitCodeMapper` is found within the `BeanFactory`, it will be injected into the runner after the context is created. All that needs to be done to provide your own `ExitCodeMapper` is to declare the implementation as a root level bean, and ensure it's part of the `ApplicationContext` that is loaded by the runner.

4.3. Job Tier

The Job Tier is responsible for the overall execution of a batch job. It sequentially executes batch steps, ensuring that all steps are in the correct state and all appropriate policies are enforced:

Job Tier



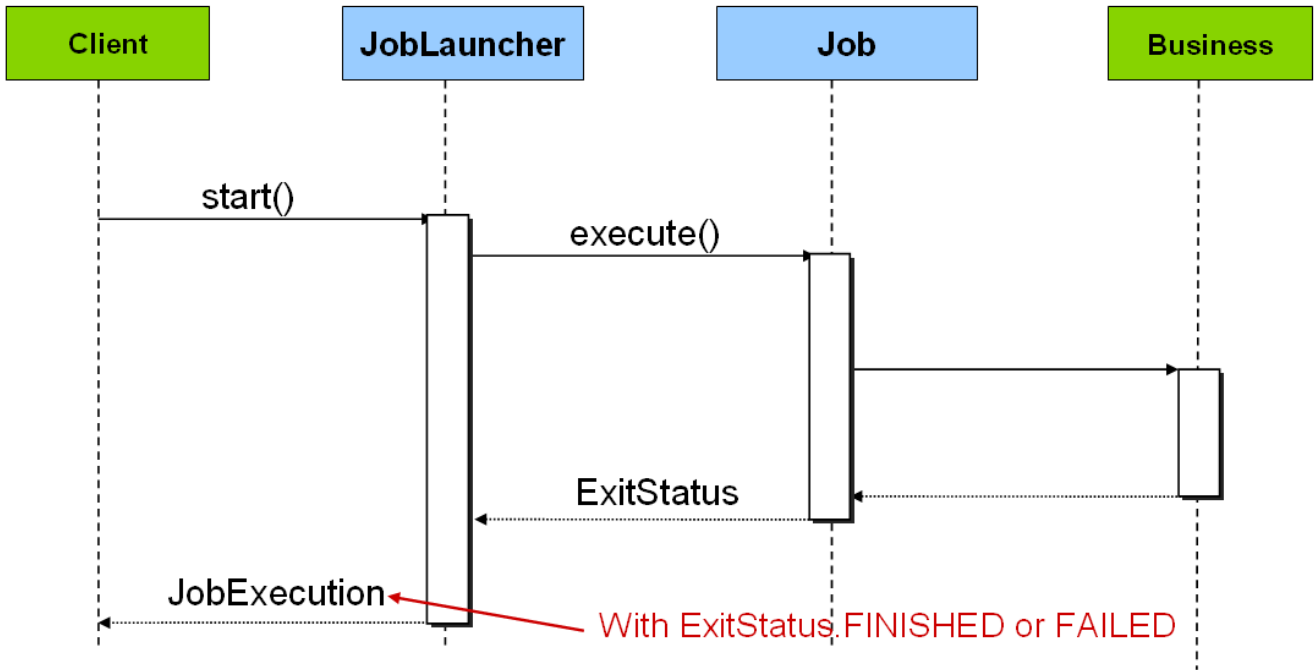
The job tier is entirely concerned with maintaining the three job stereotypes: `Job`, `JobInstance`, and `JobExecution`. The `JobLauncher` interacts with the `JobRepository` in order to create a `JobExecution`, and the `Job` stores the `JobExecution` using the repository.

4.3.1. SimpleJobLauncher

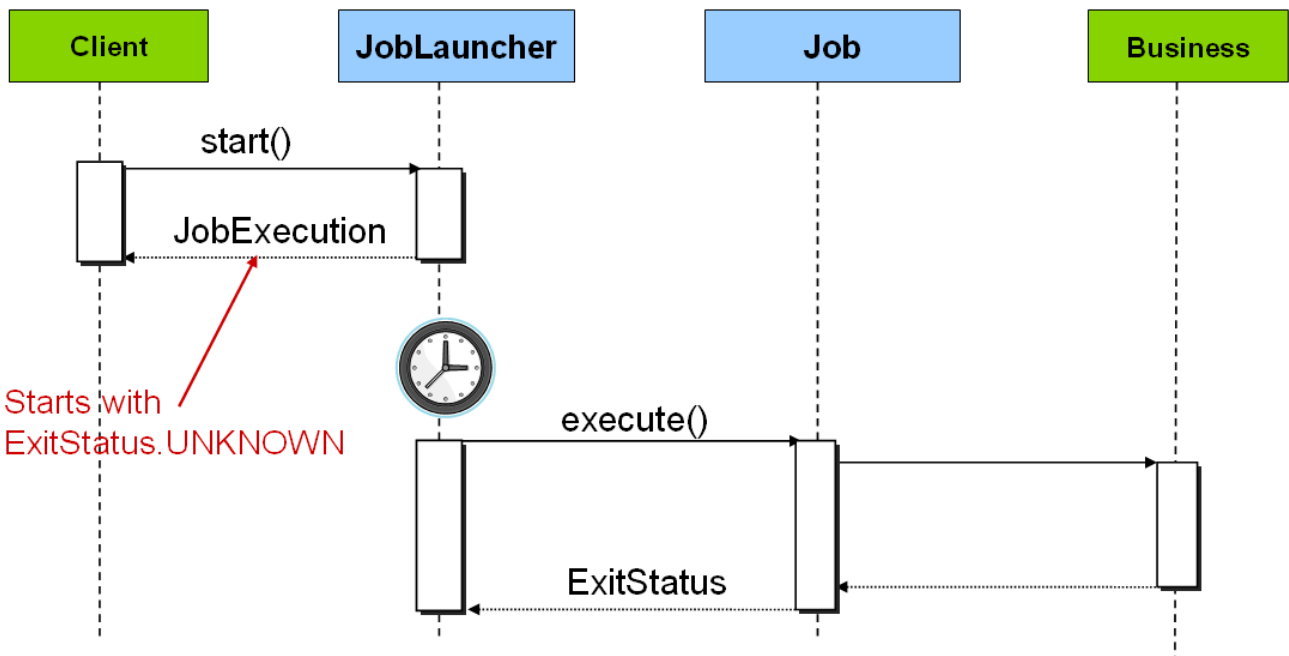
The most basic implementation of the `JobLauncher` interface is the `SimpleJobLauncher`. It's only required dependency is a `JobRepository`, in order to obtain an execution:

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

Once a `JobExecution` is obtained, it is passed to the `execute` method of `Job`, ultimately returning the `JobExecution` to the caller:



The sequence is straightforward, and works well when launched from a scheduler, but causes issues when trying to launch from an HTTP request. In this scenario, the launching needs to be done asynchronously, so that the `SimpleJobLauncher` returns immediately to its caller. This is because it is not good practice to keep an HTTP request open for the amount of time needed by long running processes such as batch. An example sequence is below:



The `SimpleJobLauncher` can easily be configured to allow for this scenario by configuring a `TaskExecutor`:

```

<bean id="jobLauncher"
    class="org.springframework.batch.execution.launch.SimpleJobLauncher">
    <property name="jobRepository" ref="jobRepository" />
    <property name="taskExecutor">
        <bean class="org.springframework.core.task.SimpleAsyncTaskExecutor" />
    </property>
</bean>
    
```

Any implementation of the `spring TaskExecutor` interface can be used to control how jobs are asynchronously executed.

4.3.1.1. Stopping a Job

One of the most common reasons for wanting to launching a job asynchronously is to be able to gracefully stop it. This can be done through the `JobExecution` returned by the `JobLauncher`:

```
JobExecution jobExecution = launcher.run(getJob(), jobParameters);

//give job adequate time to start
Thread.sleep(1000);

assertEquals("BatchStatus.STARTED", jobExecution.getStatus());
assertTrue(jobExecution.isRunning());

jobExecution.stop();

//give job time to stop
Thread.sleep(1000);

assertEquals("BatchStatus.STOPPED", jobExecution.getStatus());
assertFalse(jobExecution.isRunning());
```

The shutdown is not immediate, since there is no way to force immediate shutdown, especially if the execution is currently in developer code that the framework has no control over, such as a business service. What it does mean, is that as soon as control is returned back to the framework, it will set the status of the current `StepExecution` to `BatchStatus.STOPPED`, save it, then do the same for the `JobExecution` before finishing.

4.3.2. SimpleJobRepository

The `SimpleJobRepository` is the only provided implementation of the `JobRepository` interface. It completely manages the various batch domain objects and ensures they are created and persisted correctly. The `SimpleJobRepository` uses three different DAO interfaces for the three major domain types it stores: `JobInstanceDao`, `JobExecutionDao`, and `StepExecutionDao`. The repository delegates to these DAOs to both persist the various domain objects and query for them during initialization. The following configuration shows a `SimpleJobRepository` configured with JDBC DAOs:

```
<bean id="jobRepository" class="org.springframework.batch.core.repository.support.SimpleJobRepository">
  <constructor-arg ref="jobInstanceDao" />
  <constructor-arg ref="jobExecutionDao" />
  <constructor-arg ref="stepExecutionDao" />
</bean>

<bean id="jobInstanceDao" class="org.springframework.batch.core.repository.support.dao.JdbcJobInstanceDao" >
  <property name="jdbcTemplate" ref="jdbcTemplate" />
  <property name="jobIncrementer" ref="jobIncrementer" />
</bean>

<bean id="jobExecutionDao" class="org.springframework.batch.core.repository.support.dao.JdbcJobExecutionDao" >
  <property name="jdbcTemplate" ref="jdbcTemplate" />
  <property name="jobExecutionIncrementer" ref="jobExecutionIncrementer" />
</bean>

<bean id="stepExecutionDao" class="org.springframework.batch.core.repository.support.dao.JdbcStepExecutionDao" >
  <property name="jdbcTemplate" ref="jdbcTemplate" />
  <property name="stepExecutionIncrementer" ref="stepExecutionIncrementer" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate" >
  <property name="dataSource" ref="dataSource" />
</bean>
```

The configuration above isn't quite complete, each DAO implementation makes a reference to a Spring `DataFieldMaxValueIncrementer`. `JobInstance`, `JobExecution`, and `StepExecution` each have unique IDs, and the incrementers are used to create them.

4.3.2.1. JobRepositoryFactoryBean

Including the incrementers, which must be database specific, the configuration above is verbose. In order to make this more manageable, the framework provides a `FactoryBean` for convenience: `JobRepositoryFactoryBean`.

```
<bean id="jobRepository"
      class="org.springframework.batch.execution.repository.JobRepositoryFactoryBean"
      <property name="databaseType" value="hsqldb" />
      <property name="dataSource" value="dataSource" />
/>
```

The `databaseType` property indicates the type of incrementer that must be used. Options include: "db2", "derby", "hsqldb", "mysql", "oracle", and "postgres".

4.3.2.2. In-Memory Repository

There are scenarios in which you may not want to persist your domain objects to the database. One reason may be speed, storing domain objects at each commit point takes extra time. Another reason may be that you just don't need to persist status for a particular job. Spring batch provides a solution:

```
<bean id="jobRepository" class="org.springframework.batch.core.repository.support.SimpleJobRepository">
  <constructor-arg ref="mapJobInstanceDao" />
  <constructor-arg ref="mapJobExecutionDao" />
  <constructor-arg ref="mapStepExecutionDao" />
</bean>

<bean id="mapJobInstanceDao"
      class="org.springframework.batch.execution.repository.dao.MapJobInstanceDao" />

<bean id="mapJobExecutionDao"
      class="org.springframework.batch.execution.repository.dao.MapJobExecutionDao" />

<bean id="mapStepExecutionDao"
      class="org.springframework.batch.execution.repository.dao.MapStepExecutionDao" />
```

The `Map*` DAO implementations store the batch artifacts in a transactional map. So, the repository and DAOs may still be used normally, and are transactionally sound, but their contents will be lost when the class is destroyed.

4.3.2.2.1. Transaction Configuration For the JobRepository

If the JDBC daos are used with the `JobRepository` it is also essential to configure the transactional behaviour of the repository. This is to ensure that the batch meta data, including state that is necessary for restarts after a failure, is persisted correctly. The behaviour of the framework is not well defined if the repository methods are not transactional.

The Spring Batch samples have a `simple-job-launcher-context.xml` configuration file that contains the necessary details. Here is the relevant section:

```
<aop:config>
  <aop:advisor
    pointcut="execution(* org.springframework.batch.core.*Repository+.*(..))"
    <advice-ref="txAdvice" />
  </aop:advisor>
</aop:config>
```

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="create*" propagation="REQUIRES_NEW" isolation="SERIALIZABLE" />
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>
```

This fragment can be used as is, with almost no changes. The isolation level in the `create*` method attributes is specified to ensure that when jobs are launched there if two processes are trying to launch the same job at the same time, only one will succeed. This is quite aggressive, and `READ_COMMITTED` would work just as well; `READ_UNCOMMITTED` would be fine if two processes are not likely to collide in this way. However, since a call to the `create*` method is quite short, it is unlikely that the `SERIALIZED` will cause problems, as long as the database platform supports it.

Remember also to include the appropriate namespace declarations and to make sure `spring-tx` and `spring-aop` (or the whole of `spring`) is on the classpath.

4.3.2.2.2. Recommendations for Indexing Meta Data Tables

Spring Batch provides DDL samples for the meta-data tables in the Core jar file for several common database platforms. Index declarations are not included in that DDL because there are too many variations in how users may want to index depending on their precise platform, local conventions and also the business requirements of how the jobs will be operated. The table below provides some indication as to which columns are going to be used in a `WHERE` clause by the Dao implementations provided by Spring Batch, and how frequently they might be used, so that individual projects can make up their own minds about indexing.

Table 4.2. Where clauses in SQL statements (excluding primary keys) and their approximate frequency of use.

Default Table Name	Where Clause	Frequency
BATCH_JOB_INSTANCE	JOB_NAME = ? and JOB_KEY = ?	Every time a job is launched
BATCH_JOB_EXECUTION	JOB_INSTANCE_ID = ?	Every time a job is restarted
BATCH_STEP_EXECUTION_INSTANCE	STEP_EXECUTION_ID = ? and KEY_NAME = ?	On commit interval, a.k.a. chunk
BATCH_STEP_EXECUTION	VERSION = ?	On commit interval, a.k.a. chunk (and at start and end of step)
BATCH_STEP_EXECUTION	STEP_NAME = ? and JOB_EXECUTION_ID = ?	Before each step execution

4.3.3. SimpleJob

The only current implementation of the `Job` interface is `SimpleJob`. Since a `Job` is just a simple loop through a list of `Steps`, this implementation should be sufficient for the majority of needs. It has only three required dependencies: a name, `JobRepository`, and a list of `Steps`.

```
<bean id="footballJob"
  class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">
```

```

<list>
  <!-- Step Bean details omitted for clarity -->
  <bean id="playerload" parent="simpleStep" />
  <bean id="gameLoad" parent="simpleStep" />
  <bean id="playerSummarization" parent="simpleStep" />
</list>
</property>
<property name="jobRepository" ref="jobRepository" />
</bean>

```

Each `Step` will be executed in sequence until all have completed successfully. Any `Step` that fails will cause the entire job to fail.

4.3.3.1. Restartability

One key concern when execution a batch job, is what happens when a failed job is restarted? A `Job` is considered to have been 'restarted' if the same `JobInstance` has more than one `JobExecution`. Ideally, all jobs should be able to start up where they left off, but there are scenarios where this is not possible. **It is entirely up to the developer to ensure that a new instance is always created in this scenario.** However, Spring Batch does provide some help. If a `Job` should never be restarted, but should always be run as part of a new `JobInstance`, then the `restartable` property may be set to 'false':

```

<bean id="footballJob"
      class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">
    <list>
      <!-- Step Bean details omitted for clarity -->
      <bean id="playerload" parent="simpleStep" />
      <bean id="gameLoad" parent="simpleStep" />
      <bean id="playerSummarization" parent="simpleStep" />
    </list>
  </property>
  <property name="jobRepository" ref="jobRepository" />
  <property name="restartable" value="false" />
</bean>

```

To phrase it another way, setting `restartable` to false means "this `Job` does not support being started again". Restarting a `Job` that is not restartable will cause a `JobRestartException` to be thrown:

```

Job job = new SimpleJob();
job.setRestartable(false);

JobParameters jobParameters = new JobParameters();

JobExecution firstExecution = jobRepository.createJobExecution(job, jobParameters);
jobRepository.saveOrUpdate(firstExecution);

try {
  jobRepository.createJobExecution(job, jobParameters);
  fail();
}
catch (JobRestartException e) {
  // expected
}

```

This snippet of JUnit code shows how attempting to create a `JobExecution` the first time for a non restartable job will cause no issues. However, the second attempt will throw a `JobRestartException`.

4.3.3.2. Intercepting Job execution

During the course of the execution of a `Job`, it may be useful to be notified of various events in its lifecycle so that custom code may be executed. The `SimpleJob` allows for this by calling a `JobListener` at the appropriate time:


```
public interface JobListener {

    void beforeJob(JobExecution jobExecution);

    void afterJob(JobExecution jobExecution);

    void onError(JobExecution jobExecution, Throwable e);

    void onInterrupt(JobExecution jobExecution);
}
```

Listeners can be added to a `SimpleJob` via the `setJobListeners` property:

```
<bean id="footballJob"
      class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">
    <list>
      <!-- Step Bean details omitted for clarity -->
      <bean id="playerload" parent="simpleStep" />
      <bean id="gameLoad" parent="simpleStep" />
      <bean id="playerSummarization" parent="simpleStep" />
    </list>
  </property>
  <property name="jobRepository" ref="jobRepository" />
  <property name="jobListeners">
    <bean class="org.springframework.batch.core.listener.JobListenerSupport" />
  </property>
</bean>
```

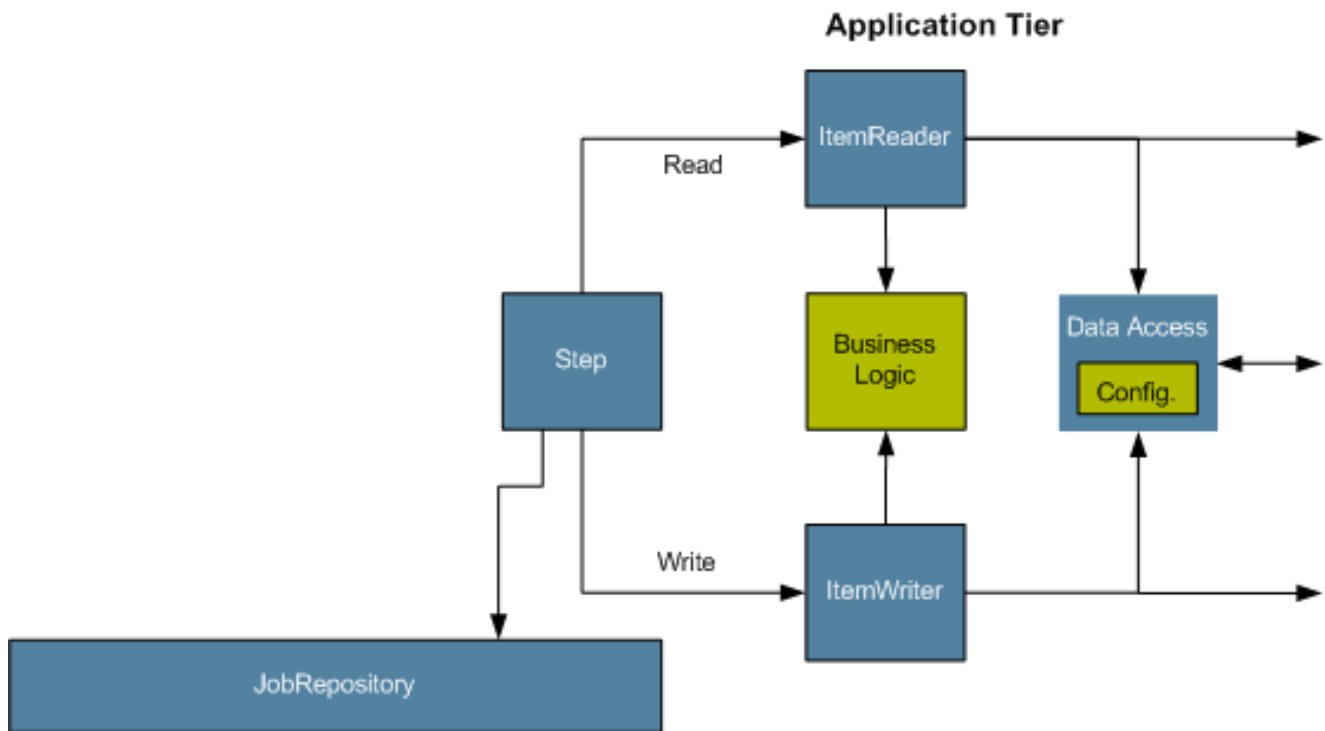
4.3.4. JobFactory and Stateful Components in Steps

Unlike many traditional Spring applications, many of the components of a batch application are stateful, the file readers and writers are obvious examples. The recommended way to deal with this is to create a fresh `ApplicationContext` for each job execution. If the `Job` is launched from the command line with `CommandLineJobRunner` this is trivial. For more complex launching scenarios, where jobs are executed in parallel or serially from the same process, some extra steps have to be taken to ensure that the `ApplicationContext` is refreshed. This is preferable to using prototype scope for the stateful beans because then they would not receive lifecycle callbacks from the container at the end of use. (e.g. through `destroy-method` in XML)

The strategy provided by Spring Batch to deal with this scenario is the `JobFactory`, and the samples provide an example of a specialized implementation that can load an `ApplicationContext` and close it properly when the job is finished. A relevant examples is `ClassPathXmlApplicationContextJobFactory` and its use in the `adhoc-job-launcher-context.xml` and the `quartz-job-launcher-context.xml`, which can be found in the Samples project.

4.4. Application Tier

The Application tier is entirely concerned with the actual processing of input:



4.4.1. ItemOrientedStep

The figure above shows a simple 'item-oriented' execution flow. One item is read in from an `ItemReader`, and then handed to an `ItemWriter`, until there are no more items left. When processing first begins, a transaction is started and periodically committed until the `Step` is complete. Given these basic requirements, the `ItemOrientedStep` requires the following dependencies, at a minimum:

- `ItemReader` - The `ItemReader` that provides items for processing.
- `ItemWriter` - The `ItemWriter` that processes the items provided by the `ItemReader`.
- `PlatformTransactionManager` - Spring transaction manager that will be used to begin and commit transactions during processing.
- `JobRepository` - The `JobRepository` that will be used to periodically store the `StepExecution` and `ExecutionContext` during processing (just before committing).

4.4.1.1. SimpleStepFactoryBean

Despite the relatively short list of required dependencies for an `ItemOrientedStep`, it is an extremely complex class that can potentially contain many collaborators. In order to ease configuration, a `SimpleStepFactoryBean` can be used:

```

<bean id="simpleStep"
      class="org.springframework.batch.core.step.item.SimpleStepFactoryBean" >
  <property name="transactionManager" ref="transactionManager" />
  <property name="jobRepository" ref="jobRepository" />
  <property name="itemReader" ref="itemReader" />
  <property name="itemWriter" ref="itemWriter" />
</bean>

```

The configuration above represents the only required dependencies of the factory bean. Attempting to instantiate a `SimpleStepFactoryBean` without at least those four dependencies will result in an exception being thrown during construction by the Spring container.

4.4.1.2. Configuring a CommitInterval

As mentioned above, the `ItemOrientedStep` reads in and writes out items, periodically committing using the supplied `PlatformTransactionManager`. By default, it will commit after each item has been written. This is less than ideal in many situations, since beginning and committing a transaction is expensive. Ideally, you would like to process as many items as possible in each transaction, which is completely dependant upon the type of data being processed and the resources that are being interacted with. For this reason, the number of items that are processed within a commit can be set as the commit interval:

```
<bean id="simpleStep"
    class="org.springframework.batch.core.step.item.SimpleStepFactoryBean" >
    <property name="transactionManager" ref="transactionManager" />
    <property name="jobRepository" ref="jobRepository" />
    <property name="itemReader" ref="itemReader" />
    <property name="itemWriter" ref="itemWriter" />
    <property name="commitInterval" value="10" />
</bean>
```

In this example, 10 items will be processed within each transaction. At the beginning of processing a transaction is begun, and each time read is called on the `ItemReader`, a counter is incremented. When it reaches 10, the transaction will be committed.

4.4.1.3. Configuring a Step for Restart

Earlier in this chapter, restarting a `Job` was discussed. Restart has numerous impacts on steps, and as such may require some specific configuration.

4.4.1.3.1. Setting a StartLimit

There are many scenarios where you may want to control the number of times a `Step` may be started. An example is a `Step` that may be run only once, usually because it invalidates some resource that must be fixed manually before it can be run again. This is configurable on the step level, since different steps have different requirements. One `Step` that may only be executed once can exist as part of the same `Job` as `Step` that can be run infinitely. Below is an example start limit configuration:

```
<bean id="simpleStep"
    class="org.springframework.batch.core.step.item.SimpleStepFactoryBean" >
    <property name="transactionManager" ref="transactionManager" />
    <property name="jobRepository" ref="jobRepository" />
    <property name="itemReader" ref="itemReader" />
    <property name="itemWriter" ref="itemWriter" />
    <property name="commitInterval" value="10" />
    <property name="startLimit" value="1" />
</bean>
```

The simple step above can be run only once. Attempting to run it again will cause an exception to be thrown. It should be noted that the default value for `startLimit` is `Integer.MAX_VALUE`.

4.4.1.3.2. Restarting a completed step

In the case of a restartable job, there may be one or more steps that should always be run, regardless of whether or not they were successful the first time. An example might be a validation step, or a step that cleans up resources before processing. During normal processing of a restarted job, any step with a status of 'COMPLETED', meaning it has already been completed successfully, will be skipped. Setting `allowStartIfComplete` to true overrides this so that the step will always run:

```
<bean id="simpleStep"
    class="org.springframework.batch.core.step.item.SimpleStepFactoryBean" >
```

```

<property name="transactionManager" ref="transactionManager" />
<property name="jobRepository" ref="jobRepository" />
<property name="itemReader" ref="itemReader" />
<property name="itemWriter" ref="itemWriter" />
<property name="commitInterval" value="10" />
<property name="startLimit" value="1" />
<property name="allowStartIfComplete" value="true" />
</bean>

```

4.4.1.3.3. Step restart configuration example

```

<bean id="footballJob"
  class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">
    <list>
      <!-- Step Bean details omitted for clarity -->
      <bean id="playerload" parent="simpleStep" />
      <bean id="gameLoad" parent="simpleStep" >
        <property name="allowStartIfComplete" value="true" />
      </bean>
      <bean id="playerSummarization" parent="simpleStep" >
        <property name="startLimit" value="2" />
      </bean>
    </list>
  </property>
  <property name="jobRepository" ref="jobRepository" />
  <property name="restartable" value="true" />
</bean>

```

The above example configuration is for a job that loads in information about football games and summarizes them. It contains three steps: `playerLoad`, `gameLoad`, and `playerSummarization`. The `playerLoad` step loads player information from a flat file, while the `gameLoad` step does the same for games. The final step, `playerSummarization`, then summarizes the statistics for each player based upon the provided games. It is assumed that the file loaded by 'playerLoad' must be loaded only once, but that 'gameLoad' will load any games found within a particular directory, deleting them after they have been successfully loaded into the database. As a result, the `playerLoad` step contains no additional configuration. It can be started almost limitlessly, and if complete will be skipped. The 'gameLoad' step, however, needs to be run everytime, in case extra files have been dropped since it last executed, so it has 'allowStartIfComplete' set to 'true' in order to always be started. (It is assumed that the database tables games are loaded into has a process indicator on it, to ensure new games can be properly found by the summarization step) The summarization step, which is the most important in the `Job`, is configured to have a start limit of 3. This is useful in case it continually fails, a new exit code will be returned to the operators that control job execution, and it won't be allowed to start again until manual intervention has taken place.

Note

This job is purely for example purposes and is not the same as the `footballJob` found in the samples project.

Run 1:

1. `playerLoad` is executed and completes successfully, adding 400 players to the 'PLAYERS' table.
2. `gameLoad` is executed and processes 11 files worth of game data, loading their contents into the 'GAMES' table.
3. `playerSummarization` begins processing and fails after 5 minutes.

Run 2:

1. `playerLoad` is not run, since it has already completed successfully, and `allowStartIfComplete` is false (the default).
2. `gameLoad` is executed again and processes another 2 files, loading their contents into the 'GAMES' table as well (with a process indicator indicating they have yet to be processed)
3. `playerSummarization` begins processing of all remaining game data (filtering using the process indicator) and fails again after 30 minutes.

Run 3:

1. `playerLoad` is not run, since it has already completed successfully, and `allowStartIfComplete` is false (the default).
2. `gameLoad` is executed again and processes another 2 files, loading their contents into the 'GAMES' table as well (with a process indicator indicating they have yet to be processed)
3. `playerSummarization` is not start, and the job is immediately killed, since this is the third execution of `playerSummarization`, and its limit is only 2. The limit must either be raised, or the `Job` must be executed as a new `JobInstance`.

4.4.1.4. Configuring Skip Logic

There are many scenarios where errors encountered while processing should not result in `Step` failure, but should be skipped instead. This is usually a decision that must be made by someone who understands the data itself and what meaning it has. Financial data, for example, may not be skippable because it results in money being transferred, which needs to be completely accurate. Loading in a list of vendors, on the other hand, might allow for skips, since a vendor not being loaded because it was formatted incorrectly, or missing necessary information, won't cause issues. Usually these bad records are logged as well, which will be covered later when discussing listeners. Configuring skip handling requires using a new factory bean: `SkipLimitStepFactoryBean`

```
<bean id="skipSample" parent="simpleStep"
      class="org.springframework.batch.core.step.item.SkipLimitStepFactoryBean">
  <property name="skipLimit" value="10" />
  <property name="itemReader" ref="flatFileItemReader" />
  <property name="itemWriter" ref="itemWriter" />
  <property name="skippableExceptionClasses"
            value="org.springframework.batch.item.file.FlatFileParseException">
  </property>
</bean>
```

In this example, a `FlatFileItemReader` is used, and if at any point a `FlatFileParseException` is thrown, it will be skipped and counted against the total skip limit of 10. It should be noted that any failures encountered while reading will not count against the commit interval. In other words, the commit interval is only incremented on writes (regardless of success or failure).

4.4.1.5. Configuring Retry Logic

In most cases you want an `Exception` to cause either a skip or `Step` failure. However, not all exceptions are deterministic. If a `FlatFileParseException` is encountered while reading, it will always be thrown for that record. Resetting the `ItemReader` will not help. However, for other exceptions, such as a `DeadlockLoserDataAccessException`, which indicates that the current process has attempted to update a record that another process holds a lock on, waiting and trying again might result in success. In this case, a `StatefulRetryStepFactoryBean` should be used:

```
<bean id="step1" parent="simpleStep"
```

```

class="org.springframework.batch.core.step.item.StatefulRetryStepFactoryBean">
<property name="itemReader" ref="itemGenerator" />
<property name="itemWriter" ref="itemWriter" />
<property name="retryLimit" value="3" />
<property name="retryableExceptionClasses" value="org.springframework.dao.DeadlockLoserDataAccesssException"
/>bean>

```

The `StatefulRetryStepFactoryBean` requires a limit for the number of times an individual item can be retried, and a list of Exceptions that are 'retryable'.

4.4.1.6. Registering ItemStreams with the Step

The step has to take care of `ItemStream` callbacks at the necessary points in its lifecycle. This is vital if a step fails, and might need to be restarted, because the `ItemStream` interface is where the step gets the information it needs about persistent state between executions. The factory beans that Spring Batch provides for convenient configuration of `Step` instances have features that allow streams to be registered with the step when it is configured.

If the `ItemReader` or `ItemWriter` themselves implement the `ItemStream` interface, then these will be registered automatically. Any other streams need to be registered separately. This is often the case where there are indirect dependencies, like delegates being injected into the reader and writer. To register these they can be injected into the factory beans through the streams property, as illustrated below:

```

<bean id="step1" parent="simpleStep"
class="org.springframework.batch.core.step.item.StatefulRetryStepFactoryBean">
<property name="streams" ref="fileItemReader" />
<property name="itemReader">
<bean
class="org.springframework.batch.item.validator.ValidatingItemReader">
<property name="itemReader" ref="itemReader" />
<property name="validator" ref="fixedValidator" />
</bean>
</property>
...
</bean>

```

In the example above the main item reader is being set up to delegate to a bean called "fileItemReader", which itself is being registered as a stream directly. The step will now be restartable and the state of the reader will be correctly persisted in case of a failure.

4.4.1.7. Intercepting Step Execution

Just as with the `Job`, there are many events during the execution of a `Step` that a user may need notification of. For example, if writing out to a flat file that requires a footer, the `ItemWriter` needs to be notified when the `Step` has been completed, so that it can write the footer. This can be accomplished with one of many `Step` scoped listeners.

4.4.1.7.1. StepExecutionListener

`StepExecutionListener` represents the most generic listener for `Step` execution. It allows for notification before a `Step` is started, after it has completed, and if any errors are encountered during processing:

```

public interface StepExecutionListener extends StepListener {

void beforeStep(StepExecution stepExecution);

ExitStatus onErrorInStep(StepExecution stepExecution, Throwable e);

ExitStatus afterStep(StepExecution stepExecution);

}

```

`ExitStatus` is the return type of `onErrorInStep` and `afterStep` in order to allow listeners the chance to modify the exit code that is returned upon completion of a `Step`. A `StepExecutionListener` can be applied to any step factory bean via the `listeners` property:

```
<bean id="simpleStep"
      class="org.springframework.batch.core.step.item.SimpleStepFactoryBean" >
  <property name="transactionManager" ref="transactionManager" />
  <property name="jobRepository" ref="jobRepository" />
  <property name="itemReader" ref="itemReader" />
  <property name="itemWriter" ref="itemWriter" />
  <property name="commitInterval" value="10" />
  <property name="listeners" ref="stepListener" />
</bean>
```

Because all listeners extend the `StepListener` interface, they all may be applied to factory beans in the same way.

4.4.1.7.2. ChunkListener

A chunk is defined as the items processed within the scope of a transaction. Committing a transaction commits a 'chunk'. It may be useful to be notified before and after a chunk has completed, in which case the `ChunkListener` interface may be used:

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

The `beforeChunk` method is called after the transaction is started, but before `read` is called on the `ItemReader`. Conversely, `afterChunk` is called after the last call to `write` on the `ItemWriter`, but before the chunk has been committed.

4.4.1.7.3. ItemReadListener

When discussing skip logic above, it was mentioned that it may be beneficial to log out skipped records, so that they can be dealt with later. In the case of read errors, this can be done with an `ItemReaderListener`:

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

The `beforeRead` method will be called before each call to `read` on the `ItemReader`. The `afterRead` method will be called after each successful call to `read`, and will be passed the item that was read. If there was an error while reading, the `onReadError` method will be called. The exception encountered will be provided so that it can be logged.

4.4.1.7.4. ItemWriteListener

Just as with the `ItemReaderListener`, the writing of an item can be 'listened' to:

```
public interface ItemWriteListener extends StepListener {
    void beforeWrite(Object item);
}
```

```

void afterWrite(Object item);

void onWriteError(Exception ex, Object item);
}

```

The `beforeWrite` method will be called before `write` on the `ItemWriter`, and is handed the item that will be written. The `afterWrite` method will be called after the item has been successfully written. If there was an error while writing, the `onWriteError` method will be called. The exception encountered and the item that was attempted to be written will be provided, so that they can be logged.

4.4.2. TaskletStep

ItemOriented processing is not the only way to process in a `Step`. What if a `Step` must consist as a simple stored procedure call? You could implement the call as an `ItemReader` and return null after the procedure finishes, but it is a bit unnatural since there would need to be a no-op `ItemWriter` and lots of overhead for transaction handling, listeners, etc. Spring Batch provides an implementation of `Step` for this scenario: `TaskletStep`. As explained in Chapter 2, the `Tasklet` is a simple interface that has one method, `execute`, which will be called once for the whole `Step`. `Tasklet` implementors might call a stored procedure, a script, or a simple SQL update statement. Because there are less concerns, there are only two required dependencies for a `TaskletStep`: a `Tasklet`, and a `JobRepository`:

```

<bean id="taskletStep"
      class="org.springframework.batch.core.step.tasklet.TaskletStep" />
  <property name="tasklet" ref="tasklet" />
  <property name="jobRepository" ref="repository" />
</bean>

```

4.4.2.1. TaskletAdapter

As with other adapters for the `ItemWriter` and `ItemReader` interfaces, the `Tasklet` interface contains an implementation that allows for adapting itself to any pre-existing class: `TaskletAdapter`. An example where this may be useful is an existing DAO that is used to update a flag on a set of records. The `TaskletAdapter` can be used to call this class without having to write an adapter for the `Tasklet` interface:

```

<bean id="deleteFilesInDir" parent="taskletStep">
  <property name="tasklet">
    <bean class="org.springframework.batch.core.step.tasklet.TaskletAdapter">
      <property name="targetObject">
        <bean class="org.mycompany.FooDao">
        </property>
      <property name="targetMethod" value="updateFoo" />
    </bean>
  </property>
</bean>

```

4.4.2.2. Example Tasklet implementation

Many batch jobs contains steps that must be done before the main processing begins in order to set up various resources, or after processing has completed to cleanup those resources. In the case of a job that works heavily with files, it is often necessary to delete certain files locally after they have been uploaded successfully to another location. The example below taken from the Spring Batch samples project, is a `Tasklet` implementation with just such a responsibility:

```

public class FileDeletingTasklet implements Tasklet, InitializingBean {

  private Resource directory;

```



```

public ExitStatus execute() throws Exception {
    File dir = directory.getFile();
    Assert.state(dir.isDirectory());

    File[] files = dir.listFiles();
    for (int i = 0; i < files.length; i++) {
        boolean deleted = files[i].delete();
        if (!deleted) {
            throw new UnexpectedJobExecutionException("Could not delete file " + files[i].getPath());
        }
    }
    return ExitStatus.FINISHED;
}

public void setDirectoryResource(Resource directory) {
    this.directory = directory;
}

public void afterPropertiesSet() throws Exception {
    Assert.notNull(directory, "directory must be set");
}
}

```

The above `Tasklet` implementation will delete all files within a given directory. It should be noted that the `execute` method will only be called once. All that is left is to inject the `Tasklet` into a `TaskletStep`:

```

<bean id="taskletJob" parent="simpleJob">
  <property name="steps">
    <bean id="deleteFilesInDir" parent="taskletStep">
      <property name="tasklet">
        <bean class="org.springframework.batch.sample.tasklet.FileDeletingTasklet">
          <property name="directoryResource" ref="directory" />
        </bean>
      </property>
    </bean>
  </property>
</bean>

<bean id="directory"
      class="org.springframework.core.io.FileSystemResource">
  <constructor-arg value="target/test-outputs/test-dir" />
</bean>

```

4.5. Examples of Customized Business Logic

Some batch jobs can be assembled purely from off-the-shelf components in Spring Batch, mostly the `ItemReader` and `ItemWriter` implementations. Where this is not possible (the majority of cases) the main API entry points for application developers are the `Tasklet`, `ItemReader`, `ItemWriter` and the various listener interfaces. Most simple batch jobs will be able to use off-the-shelf input from a Spring Batch `ItemReader`, but it is very often the case that there are custom concerns in the processing and writing, which normally leads developers to implement an `ItemWriter`, or `ItemTransformer`.

Here we provide a few examples of common patterns in custom business logic, mainly using the listener interfaces. It should be noted that an `ItemReader` or `ItemWriter` can implement the listener interfaces as well if appropriate.

4.5.2. 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. The `ItemOrientedStep` (created from the step factory beans) allows users to implement this use case with a simple `ItemReadListener`, for errors on read, and an `ItemWriteListener`, for errors on write. The below code snippets illustrate a listener that logs both read and

write failures:

```
public class ItemFailureLoggerListener extends ItemListenerSupport {
    private static Log logger = LoggerFactory.getLog("item.error");

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

    public void onError(Exception ex, Object item) {
        logger.error("Encountered error on write", e);
    }
}
```

Having implemented this listener it must be registered with the step:

```
<bean id="simpleStep"
    class="org.springframework.batch.core.step.item.SimpleStepFactoryBean" >
    ...
    <property name="listeners">
        <bean class="org.example...ItemFailureLoggerListener"/>
    </property>
</bean>
```

Remember that if your listener does anything in an `onError()` method, it will 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 the value `REQUIRES_NEW`.

4.5.3. Stopping a Job Manually for Business Reasons

Spring Batch provides a `stop()` method through the `JobLauncher` interface, but this is really aimed at 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 isn't retried indefinitely or skipped), For example, a custom exception type could be used, as in the example below:

```
public class PoisonPillItemWriter extends AbstractItemWriter {
    public void write(Object item) throws Exception {
        if (isPoisonPill(item)) {
            throw new PoisonPillException("Posion pill detected: "+item);
        }
    }
}
```

Another simple way to stop a step from executing is to simply return `null` from the `ItemReader`:

```
public class EarlyCompletionItemReader extends AbstractItemReader {
    private ItemReader delegate;

    public void setDelegate(ItemReader delegate) { ... }

    public Object read() throws Exception {
        Object 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 which 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 `RepeatOperationsStepFactoryBean`:

```

<bean id="simpleStep"
      class="org.springframework.batch.core.step.item.RepeatOperationsStepFactoryBean" >
  ...
  <property name="chunkOperations">
    <bean class="org.springframework.batch.repeat.support.RepeatTemplate">
      <property name="completionPolicy">
        <bean class="org.example...SpecialCompletionPolicy"/>
      </property>
    </bean>
  </property>
</bean>

```

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`. Here is an example of 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);
        }

    }

}

```

The default behaviour here when the flag is set is for the step to throw a `JobInterruptedException`. This can be controlled through the `StepInterruptionPolicy`, but the only choice is to throw or not throw an exception, so this is always an abnormal ending to a job.

4.5.4. Adding a Footer Record

A very common requirement is to aggregate information during the output process and to append a record at the end of a file summarizing the data, or providing a checksum. This can also be achieved with a callbacks in the step, normally as part of a custom `ItemWriter`. In this case, since a job is accumulating state that should not be lost if the job aborts, the `ItemStream` interface should be implemented:

```

public class CustomItemWriter extends AbstractItemWriter implements
    ItemStream, StepListener
{

```

```
private static final String TOTAL_AMOUNT_KEY = "total.amount";

private ItemWriter delegate;

private double totalAmount = 0.0;

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

public ExitStatus afterStep(StepExecution stepExecution) {
    // Add the footer record here...
    delegate.write("Total Amount Processed: " + totalAmount);
}

public void open(ExecutionContext executionContext) {
    if (executionContext.containsKey(TOTAL_AMOUNT_KEY) {
        totalAmount = executionContext.getDouble(TOTAL_AMOUNT_KEY);
    }
}

public void update(ExecutionContext executionContext) {
    executionContext.setDouble(TOTAL_AMOUNT_KEY, totalAmount);
}

public void write(Object item) {

    delegate.write(item);
    totalAmount += ((Trade) item).getAmount();

}

}
```

The custom writer in the example is stateful (it maintains its total in an instance variable `totalAmount`), but the state is stored through the `ItemStream` interface in the `ExecutionContext`. In this way we can be sure that when the `open()` callback is received on a restart. The framework guarantees we always get the last value that was committed. It should be noted that it is not always necessary to implement `ItemStream`. For example, if the `ItemWriter` is re-runnable, in the sense that it maintains its own state in a transactional resource like a database, there is no need to maintain state within the writer itself.

Chapter 5. Repeat

5.1. RepeatTemplate

Batch processing is about repetitive actions - either as a simple optimisation, or as part of a job. To strategise and generalise the repetition, and provide what amounts to an iterator framework, Spring Batch has the `RepeatOperations` interface. The `RepeatOperations` interface looks like this:

```
public interface RepeatOperations {  
    ExitStatus iterate(RepeatCallback callback) throws RepeatException;  
}
```

where the callback is a simple interface that allows you to insert some business logic to be repeated

```
public interface RepeatCallback {  
    ExitStatus doInIteration(RepeatContext context) throws Exception;  
}
```

The callback is executed repeatedly, until the implementation decides that the iteration should end. The return value in these interfaces is a special form of extendable enumeration (not a true enumeration because users are free to create new values). An `ExitStatus` is immutable and conveys information to the caller of the repeat operations about whether there is any more work to do. Generally speaking, implementations of `RepeatOperations` should inspect the `ExitStatus` and use it as part of the decision to end the iteration. Any callback that wishes to signal to the caller that there is no more work to do can return `ExitStatus.FINISHED`.

The simplest general purpose implementation of `RepeatOperations` is `RepeatTemplate`. It could be used like this

```
RepeatTemplate template = new RepeatTemplate();  
template.setCompletionPolicy(new FixedChunkSizeCompletionPolicy(2));  
template.iterate(new RepeatCallback() {  
    public ExitStatus doInIteration(RepeatContext context) {  
        // Do stuff in batch...  
        return ExitStatus.CONTINUABLE;  
    }  
});
```

In the example we return `ExitStatus.CONTINUABLE` to show that there is more work to do. The callback can also return `ExitStatus.FINISHED` if it wants to signal to the caller that there is no more work to do. Some iterations can be terminated by considerations intrinsic to the work being done in the callback, others are effectively infinite loops as far as the callback is concerned, and the completion decision is delegated to an external policy as in the case above.

5.1.1. RepeatContext

The method parameter for the `RepeatCallback` is a `RepeatContext`. Many callbacks will simply ignore the context, but if necessary it can be used as an attribute bag to store transient data for the duration of the iteration.

After the `iterate` method returns, the context will no longer exist.

A `RepeatContext` will have a parent context if there is a nested iteration in progress. The parent context is occasionally useful for storing data that need to be shared between calls to `iterate`. This is the case for instance if you want to count the number of occurrences of an even in the iteration and remember it across subsequent calls.

5.1.2. ExitStatus

`ExitStatus` is used by Spring Batch to indicate whether processing has finished, and if so whether or not it was successful. It is also used to carry textual information about the end state of a batch or iteration, in the form of an exit code and a description of the status in freeform text. These are the properties of an `ExitStatus`:

Table 5.1. ExitStatus properties

Property Name	Type	Description
<code>continuable</code>	boolean	true if there is more work to do
<code>exitCode</code>	String	Short code describing the exit status, e.g. <code>CONTINUABLE</code> , <code>FINISHED</code> , <code>FAILED</code>
<code>exitDescription</code>	String	Long description of the exit status, could be a stack trace for example.

`ExitStatus` values are designed to be flexible, so that they can be created with any code and description the user needs. Spring Batch comes with some standard values out of the box, to support common use cases, but users are free to create their own values, as long as the semantics of the `continuable` property are honoured.

`ExitStatus` values can also be combined with various operators built into the class as methods. You can add an exit code, or description, or combine the `continuable` values with logical AND using methods in `ExitStatus`. You can also combine two `ExitStatus` values with the `and` method taking `ExitStatus` as a parameter. The effect of this is to do a logical AND on the `continuable` flag, concatenate the descriptions and replace the exit code with the new value, as long as the result is `continuable`, or the input is not `continuable`. This has the effect of maintaining the semantics of the `continuable` flag, but not making any "surprising" changes to the exit code (e.g. it never becomes `CONTINUABLE` when it was already `FINISHED`, unless someone does something wilful, like pass in a value that is not `continuable`, but with a code of `CONTINUABLE`).

5.2. Completion Policies

Inside a `RepeatTemplate` the termination of the loop in the `iterate` method is determined by a `CompletionPolicy` which is also a factory for the `RepeatContext`. The `RepeatTemplate` has the responsibility to use the current policy to create a `RepeatContext` and pass that in to the `RepeatCallback` at every stage in the iteration. After a callback completes its `doInIteration` the `RepeatTemplate` has to make a call to the `CompletionPolicy` to ask it to update its state (which will be stored in the `RepeatContext`), then it asks the policy if the iteration is complete.

Spring Batch provides some simple general purpose implementations of `CompletionPolicy`, for example the `SimpleCompletionPolicy` used in the example above. The `SimpleCompletionPolicy` just allows an execution up to a fixed number of times (with `ExitStatus.FINISHED` forcing early completion at any time).

Users might need to implement their own completion policies for more complicated decisions, e.g. a batch processing window that prevents batch jobs from executing once the online systems are in use.

5.3. Exception Handling

If there is an exception thrown inside a `RepeatCallback`, the `RepeatTemplate` consults an `ExceptionHandler` which can decide whether or not to re-throw the exception.

```
public interface ExceptionHandler {  
  
    void handleException(RepeatContext context, Throwable throwable)  
        throws RuntimeException;  
  
}
```

A common use case is to count the number of exceptions of a given type, and fail when a limit is reached. For this purpose Spring Batch provides the `SimpleLimitExceptionHandler` and slightly more flexible `RethrowOnThresholdExceptionHandler`. The `SimpleLimitExceptionHandler` has a `limit` property and an exception type that should be compared with the current exception - all subclasses of the provided type are also counted. Exceptions of the given type are ignored until the limit is reached, and then rethrown. Those of other types are always rethrown.

An important optional property of the `SimpleLimitExceptionHandler` is the boolean flag `useParent`. It is false by default, so the limit is only accounted for in the current `RepeatContext`. When set to true, the limit is kept across sibling contexts in a nested iteration (e.g. a set of chunks inside a step).

5.4. Listeners

Often it is useful to be able to receive additional callbacks for cross cutting concerns across a number of different iterations. For this purpose Spring Batch provides the `RepeatListener` interface. The `RepeatTemplate` allows users to register `RepeatListeners`, and they will be given callbacks with the `RepeatContext` and `ExitStatus` where available during the iteration.

The interface looks like this:

```
public interface RepeatListener {  
    void before(RepeatContext context);  
  
    void after(RepeatContext context, ExitStatus result);  
  
    void open(RepeatContext context);  
  
    void onError(RepeatContext context, Throwable e);  
  
    void close(RepeatContext context);  
  
}
```

The `open` and `close` callbacks come before and after the entire iteration, and `before`, `after` and `onError` apply to the individual `RepeatCallback` calls.

Note that when there is more than one listener, they are in a list, so there is an order. In this case `open` and `before` are called in the same order, and `after`, `onError` and `close` are called in reverse order.

5.5. Parallel Processing

Implementations of `RepeatOperations` are not restricted to executing the callback sequentially. It is quite important that some implementations are able to execute their callbacks in parallel. To this end Spring Batch provides the `TaskExecutorRepeatTemplate`, which uses the Spring `TaskExecutor` strategy to run the `RepeatCallback`. The default is to use a `SynchronousTaskExecutor`, which has the effect of executing the whole iteration in the same thread (the same as a normal `RepeatTemplate`).

5.6. Declarative Iteration

Sometimes there is some business processing that you know you want to repeat every time it happens. The classic example of this is the optimization of a message pipeline - it is more efficient to process a batch of messages, if they are arriving frequently, than to bear the cost of a separate transaction for every message. Spring Batch provides an AOP interceptor that wraps a method call in a `RepeatOperations` for just this purpose. The `RepeatOperationsInterceptor` executes the intercepted method and repeats according to the `CompletionPolicy` in the provided `RepeatTemplate`.

Here is an example of declarative iteration using the Spring AOP namespace to repeat a service call to a method called `processMessage` (for more detail on how to configure AOP interceptors see the Spring User Guide):

```
<aop:config>
  <aop:pointcut id="transactional"
    expression="execution(* com...*Service.processMessage(..))" />
  <aop:advisor pointcut-ref="transactional"
    advice-ref="retryAdvice" order="-1"/>
</aop:config>

<bean id="retryAdvice"
  class="org.springframework.batch.repeat.interceptor.RepeatOperationsInterceptor"/>
```

The example above uses a default `RepeatTemplate` inside the interceptor. To change the policies, listeners etc. you only need to inject an instance of `RepeatTemplate` into the interceptor.

If the intercepted method returns `void` then the interceptor always returns `ExitStatus.CONTINUABLE` (so there is a danger of an infinite loop if the `CompletionPolicy` does not have a finite end point). Otherwise it returns `ExitStatus.CONTINUABLE` until the return value from the intercepted method is `null`, at which point it returns `ExitStatus.FINISHED`. So the business logic inside the target method can signal that there is no more work to do by returning `null`, or by throwing an exception that is re-thrown by the `ExceptionHandler` in the provided `RepeatTemplate`.

Chapter 6. Retry

6.1. RetryTemplate

To make processing more robust and less prone to failure, sometimes it helps to automatically retry a failed operation in case it might succeed on a subsequent attempt. Errors that are susceptible to this kind of treatment are transient in nature, for example a remote call to a web service or RMI service that fails because of a network glitch, or a `DeadLockLoserException` in a database update. To automate the retry of such operations Spring Batch has the `RetryOperations` strategy. The `RetryOperations` interface looks like this:

```
public interface RetryOperations {  
    Object execute(RetryCallback retryCallback) throws Exception;  
}
```

where the callback is a simple interface that allows you to insert some business logic to be retried

```
public interface RetryCallback {  
    Object doWithRetry(RetryContext context) throws Throwable;  
}
```

The callback is executed and if it fails (by throwing an `Exception`), it will be retried until either it is successful, or the implementation decides to abort.

The simplest general purpose implementation of `RetryOperations` is `RetryTemplate`. It could be used like this

```
RetryTemplate template = new RetryTemplate();  
template.setRetryPolicy(new TimeoutRetryPolicy(30000L));  
Object result = template.execute(new RetryCallback() {  
    public Object doWithRetry(RetryContext context) {  
        // Do stuff that might fail, e.g. webservice operation  
        return result;  
    }  
});
```

In the example we execute a web service call and return the result to the user. If that call fails then it is retried until a timeout is reached.

6.1.1. RetryContext

The method parameter for the `RetryCallback` is a `RetryContext`. Many callbacks will simply ignore the context, but if necessary it can be used as an attribute bag to store data for the duration of the iteration.

A `RetryContext` will have a parent context if there is a nested retry in progress in the same thread. The parent context is occasionally useful for storing data that need to be shared between calls to `execute`.

6.2. Retry Policies

Inside a `RetryTemplate` the decision to retry or fail in the `execute` method is determined by a `RetryPolicy` which is also a factory for the `RetryContext`. The `RetryTemplate` has the responsibility to use the current policy to create a `RetryContext` and pass that in to the `RetryCallback` at every attempt. After a callback fails the `RetryTemplate` has to make a call to the `RetryPolicy` to ask it to update its state (which will be stored in the `RetryContext`), and then it asks the policy if another attempt can be made. If another attempt cannot be made (e.g. a limit is reached or a timeout is detected) then the policy is also responsible for handling the exhausted state. Simple implementations will just throw `RetryExhaustedException`, and any enclosing transaction will be rolled back. More sophisticated implementations might attempt to take some recovery action, in which case the transaction can remain intact.

Tip

Failures are inherently either retryable or not - if the same exception is always going to be thrown from the business logic, it doesn't help to retry it. So don't retry on all exception types - try to focus on only those exceptions that you expect to be retryable. It's not usually harmful to the business logic to retry more aggressively, but it's wasteful because if a failure is deterministic there could be a very tight loop retrying something that you know in advance is fatal.

6.2.1. Stateless Retry

In the simplest case a retry is just a while loop - the `RetryTemplate` can just keep trying until it either succeeds or fails. The `RetryContext` contains some state to determine whether to retry or abort, but this state is on the stack and there is no need to store it anywhere globally, so we call this stateless retry. The distinction between stateless and stateful retry is contained in the implementation of the `RetryPolicy` (the `RetryTemplate` can handle both). In a stateless retry, the callback is always executed in the same thread on retry as when it failed.

Spring Batch provides some simple general purpose implementations of stateless `RetryPolicy`, for example a `SimpleRetryPolicy`, and the `TimeoutRetryPolicy` used in the example above.

The `SimpleRetryPolicy` just allows a retry on any of a named list of exception types, up to a fixed number of times. It also has a list of "fatal" exceptions that should never be retried, and this list overrides the retryable list, so it can be used to give finer control over the retry behaviour, e.g.

```
SimpleRetryPolicy policy = new SimpleRetryPolicy(5);
// Retry on all exceptions (this is the default)
policy.setRetryableExceptions(new Class[] {Exception.class});
// ... but never retry IllegalStateException
policy.setFatalExceptions(new Class[] {IllegalStateException.class});

// Use the policy...
RetryTemplate template = new RetryTemplate();
template.setRetryPolicy(policy);
template.execute(new RetryCallback() {
    public Object doWithRetry(RetryContext context) {
        // business logic here
    }
});
```

There is also a more flexible implementation called `ExceptionClassifierRetryPolicy`, which allows the user to configure different retry behaviour for an arbitrary set of exception types through the `ExceptionClassifier` abstraction. The policy works by calling on the classifier to convert an exception into a delegate `RetryPolicy`, so for example, one exception type can be retried more times before failure than another by mapping it to a different policy.

Users might need to implement their own retry policies for more customized decisions, e.g. if there is a

well-known solution-specific classification of exceptions into retryable and not retryable.

6.2.2. Stateful Retry

Where the failure has caused a transactional resource to become invalid there are some special considerations. This does not apply to a simple remote call because there was no transactional resource (usually), but it does sometimes apply to a database update, especially when using Hibernate. In this case it only makes sense to rethrow the exception that called the failure immediately, so that the transaction can roll back, and we can start a new valid one.

In these cases a stateless retry is not good enough because the re-throw and roll back necessarily involve leaving the `RetryOperations.execute()` method and potentially losing the context that was on the stack. To avoid losing it we have to introduce a storage strategy to lift it off the stack and put it (at a minimum) in heap storage. For this purpose Spring Batch provides a storage strategy `RetryContextCache`. The default implementation of the `RetryContextCache` is in memory, using a simple `Map`. Advanced usage with multiple processes in a clustered environment might also consider implementing the `RetryContextCache` with a cluster cache of some sort (even in a clustered environment this might be overkill).

6.2.2.1. Item processing and stateful retry

Part of the responsibility of a stateful retry policy is to recognise the failed operations when they come back in a new transaction. To facilitate this in the commonest case where an object (like a message or message payload) is being processed, Spring Batch provides the `ItemWriterRetryPolicy`. This works in conjunction with a special `RetryCallback` implementation `ItemWriterRetryCallback`, which in turn relies on the user providing an `ItemWriter`. This callback implements the common pattern where it passes the item to a writer.

The way the failed operations are recognised in this implementation is by identifying the item across multiple invocations of the retry. To identify the item the user can provide an `ItemKeyGenerator` strategy, and this is responsible for returning a unique key identifying the item. The identifier is used as a key in the `RetryContextCache`. An `ItemKeyGenerator` can be provided either by injecting it directly into the `ItemWriterRetryCallback`, or by implementing the interface in the `ItemWriter`, or by accepting the default which is to simply use the item itself as a key.

Warning

If you use the default item key generation strategy be very careful with the implementation of `Object.equals()` and `Object.hashCode()` in your item class. In particular, if the `ItemWriter` is going to insert the item into a database and update a primary key field it is not a good idea to use the primary key in the `equals` and `hashCode` implementations, because their values will change before and after the call to the `ItemWriter`. The best advice is to use a business key to identify the items.

When the retry is exhausted, because a stateful retry is always in a fresh transaction, there is also the option to handle the failed item in a different way, instead of calling the `RetryCallback` (which is presumed now to be likely to fail). This option is provided by the `ItemRecoverer` strategy. Like the key generator, it can be directly injected or provided by implementing the interface in the `ItemWriter`.

The decision to retry or not is actually delegated to a regular stateless retry policy, so the usual concerns about limits and timeouts can be injected into the `ItemWriterRetryPolicy` through the `delegate` property.

6.3. Backoff Policies

When retrying after a transient failure it often helps to wait a bit before trying again, because usually the failure is caused by some problem that will only be resolved by waiting. If a `RetryCallback` fails, the `RetryTemplate` can pause execution according to the `BackoffPolicy` in place.

```
public interface BackoffPolicy {
    BackOffContext start(RetryContext context);

    void backOff(BackOffContext backOffContext)
        throws BackOffInterruptedException;
}
```

A `BackoffPolicy` is free to implement the `backOff` in any way it chooses. The policies provided by Spring Batch out of the box all use `Object.wait()`. A common use case is to backoff with an exponentially increasing wait period, to avoid two retries getting into lock step and both failing - this is a lesson learned from the ethernet. For this purpose Spring Batch provides the `ExponentialBackoffPolicy`.

6.4. Listeners

Often it is useful to be able to receive additional callbacks for cross cutting concerns across a number of different retries. For this purpose Spring Batch provides the `RetryListener` interface. The `RetryTemplate` allows users to register `RetryListeners`, and they will be given callbacks with the `RetryContext` and `Throwable` where available during the iteration.

The interface looks like this:

```
public interface RetryListener {
    void open(RetryContext context, RetryCallback callback);

    void onError(RetryContext context, RetryCallback callback, Throwable e);

    void close(RetryContext context, RetryCallback callback, Throwable e);
}
```

The `open` and `close` callbacks come before and after the entire retry in the simplest case, and `onError` applies to the individual `RetryCallback` calls. The `close` method might also receive a `Throwable`, if there has been an error it is the last one thrown by the `RetryCallback`.

Note that when there is more than one listener, they are in a list, so there is an order. In this case `open` is called in the same order, and `onError` and `close` are called in reverse order.

6.5. Declarative Retry

Sometimes there is some business processing that you know you want to retry every time it happens. The classic example of this is the remote service call. Spring Batch provides an AOP interceptor that wraps a method call in a `RetryOperations` for just this purpose. The `RetryOperationsInterceptor` executes the intercepted method and retries on failure according to the `RetryPolicy` in the provided `RepeatTemplate`.

Here is an example of declarative iteration using the Spring AOP namespace to repeat a service call to a method called `remoteCall` (for more detail on how to configure AOP interceptors see the Spring User Guide):

```
<aop:config>
  <aop:pointcut id="transactional"
    expression="execution(* com...*Service.remoteCall(..))" />
  <aop:advisor pointcut-ref="transactional"
    advice-ref="retryAdvice" order="-1"/>
</aop:config>

<bean id="retryAdvice"
  class="org.springframework.batch.retry.interceptor.RetryOperationsInterceptor"/>
```

The example above uses a default `RetryTemplate` inside the interceptor. To change the policies, listeners etc. you only need to inject an instance of `RetryTemplate` into the interceptor.

Chapter 7. Unit Testing

Just as with other application styles, it is extremely important to unit test any code written as part of a batch job as well. The Spring core documentation covers how to unit and integration test with Spring in great detail, so it won't be repeated here. It is important, however, to think about how to 'end to end' test a batch job, which is what this chapter will focus on.

7.1. End To End Testing Batch Jobs

'End To End' testing can be defined as testing the complete run of a batch job from beginning to end. If the job reads from a file, then writes into the database, this type of testing ensures that any preconditions are met (reference data, correct file, etc) and then runs the job, verifying afterwards that all records that should be in the database are present and correct. Below is an example from one of the Spring Batch sample jobs, the 'fixedLengthImportJob'. It reads from a flat file (in fixed length format) and loads the records into the database. The following unit test code assures it processes correctly:

```
//fixed-length file is expected on input
protected void validatePreConditions() throws Exception{
    BufferedReader reader = null;
    reader = new BufferedReader(new FileReader(fileLocator.getFile()));
    String line;
    while ((line = reader.readLine()) != null) {
        assertEquals (LINE_LENGTH, line.length());
    }
}

//Check that records have been correctly written to database
protected void validatePostConditions() throws Exception {

    inputSource.open(new ExecutionContext());

    jdbcTemplate.query("SELECT ID, ISIN, QUANTITY, PRICE, CUSTOMER FROM trade ORDER BY id",
        new RowCallbackHandler() {

            public void processRow(ResultSet rs) throws SQLException {
                Trade trade;
                try {
                    trade = (Trade)inputSource.read();
                }
                catch (Exception e) {
                    throw new IllegalStateException(e.getMessage());
                }
                assertEquals(trade.getIsin(), rs.getString(2));
                assertEquals(trade.getQuantity(),rs.getLong(3));
                assertEquals(trade.getPrice(), rs.getBigDecimal(4));
                assertEquals(trade.getCustomer(), rs.getString(5));
            }
        });

    assertNull(inputSource.read());
}
```

In the first method, `validatePreConditions`, the input file is checked to ensure it is correctly formatted. Because it is common to add extra lines to the file to test additional use cases, this test ensures that the fixed length lines are the length they should be. If they are not, it is much preferred to fail in this phase, rather than the job (correctly) failing during the run and causing needless debugging.

In the second method, `validatePostconditions`, the database is checked to ensure all data has been written correctly. This is arguably the most important part of the test. In this case, it reads one line from the file, and one row from the database, and checks each column one by one for accuracy. It's important to not hard-code the data that should be present in the database into the test class. Instead, use the input file (bypassing the job) to check the output. This allows you to quickly add additional test cases to your file without having to add them to

code. The same would be true for database to database jobs, or database to file jobs. It is preferable to be able to add additional rows to the database input without having to add them to the hard coded list in the test class.

7.2. Extending Unit Test frameworks

Because most unit testing of complete batch jobs will take place in the development environment (i.e. eclipse) it's important to be able to launch these tests in the same way you would launch any unit test. In the following examples JUnit 3.8 will be used, but any testing framework could be substituted. The Spring Batch samples contain many 'sample jobs' that are unit tested using this technique. The most important step is being able to launch the job within a unit test. This requires the use of the `JobLauncher` interface that is discussed in chapters 2 and 4. A `Job` and `JobLauncher` must be obtained from an `ApplicationContext`, and then launched. The following abstract class from Spring Batch Samples illustrates this:

```
public abstract class AbstractBatchLauncherTests extends
    AbstractDependencyInjectionSpringContextTests {

    JobLauncher launcher;
    private Job job;
    private JobParameters jobParameters = new JobParameters();

    public AbstractBatchLauncherTests() {
        setDependencyCheck(false);
    }

    /*
    * @see org.springframework.test.AbstractSingleSpringContextTests#getConfigLocations()
    */
    protected String[] getConfigLocations() {
        return new String[] { ClassUtils.addResourcePathToPackagePath(getClass(),
            ClassUtils.getShortName(getClass()) + "-context.xml" ) };
    }

    public void testLaunchJob() throws Exception {
        launcher.run(job, jobParameters);
    }

    public void setLauncher(JobLauncher bootstrap) {
        this.launcher = bootstrap;
    }

    public void setJob(Job job) {
        this.job = job;
    }
}
```

The Spring Test class `AbstractDependencyInjectionSpringContextTests` is extended to allow for context loading, autowiring, etc. Only two classes are needed: The `Job` to be run, and the `JobLauncher` to run it. Empty `JobParameters` are used in the example above. However, if the job requires specific parameters they could be coded in subclasses with an abstract method, or using a factory bean in the `ApplicationContext` for testing purposes. Because none of the sample jobs require this, an empty `JobParameters` is used. One simple JUnit test case is present in the file, which actually launches the job. If any exceptions are thrown or assertions fail, it will act the same way as any other unit test and display as a failed test due to errors or assertion failure. Because of the best practice for validation mentioned earlier in the chapter, this class is extended further to allow for separate validation before and after the job is run:

```
public abstract class AbstractValidatingBatchLauncherTests extends AbstractBatchLauncherTests {

    public void testLaunchJob() throws Exception {
        validatePreConditions();
        super.testLaunchJob();
        validatePostConditions();
    }

    /**
```

```
* Make sure input data meets expectations
*/
protected void validatePreConditions() throws Exception {}

/**
 * Make sure job did what it was expected to do.
 */
protected abstract void validatePostConditions() throws Exception;
}
```

In the class above, the `testLaunchJob` method is overridden to call the two abstract methods for validation. Before actually running the job, `validatePreConditions` is called (it should be noted that it's not required), and then after the job completes successfully, `validatePostConidtions` is called.

Appendix A. List of ItemReaders

A.1. Item Readers

Table A.1. Available Item Readers

Item Reader	Type of Item Provided	Description
ListItemReader	java.lang.Object	Provides the items from a list, one at a time
ValidatingItemReader	java.lang.Object	A simple extension of DelegatingItemReader that provides for validation before returning input.
AggregateItemReader	java.util.Collection	An ItemReader that delivers a list as its item, storing up objects from the injected ItemReader until they are ready to be packed out as a collection. This ItemReader should mark the beginning and end of records with the constant values in FieldSetMapper AggregateItemReader# BEGIN_RECORD and AggregateItemReader# END_RECORD
DelegatingItemReader	java.lang.Object	Extends AbstractMethodInvokingDelegator, which enables dynamically calling of a custom method of the injected object. Provides a convenient API for dynamic method invocation shielding subclasses from the low-level details and exception handling.
FlatFileItemReader	java.lang.String	Reads from a flat file, includes ItemStream and Skippable functionality. See section on Read from a File
StaxEventItemReader	java.lang.Object	Reads via StAX. See HOWTO - Read from a File
JdbcCursorItemReader	java.lang.Object	Reads from a database cursor via JDBC. See HOWTO - Read from a Database
DrivingQueryItemReader	java.lang.Object	Base class for operations that read from a database based on a single

Item Reader	Type of Item Provided	Description
		driving query. Configured by injecting a KeyGenerator object. See HOWTO - Read from a Database
HibernateCursorItemReader	java.lang.Object	Reads from a cursor based on an HQL query. See section on Reading from a Database
IbatisDrivingQueryItemReader	java.lang.Object	Reads via iBATIS based on a driving query. See HOWTO - Read from a Database
JmsItemReader	javax.jms.Message	Given a Spring JmsOperations object and a JMS Destination or destination name to send errors, provides items received through the injected JmsOperations receive() method

A.2. Item Writers

Table A.2. Available Item Writers

Item Writer	Type of Item Written	Description
CompositeItemWriter	java.lang.Object	Passes an item to the process method of each in an injected List of ItemWriter objects
DelegatingItemWriter	java.lang.Object	Wraps ItemWriter and is BeanAware allowing it to respond to Spring Bean events like afterPropertiesSet().
PropertyExtractingDelegatingItemWriter	java.lang.Object	Extends AbstractMethodInvokingDelegator creating arguments on the fly. Arguments are created by retrieving the values from the fields in the item to be processed (via a SpringBeanWrapper) based on an injected array of field name
ItemTransformerItemWriter	java.lang.Object	Extends ItemWriterItemWriter by defining its doProcess method to call an injected ItemTransformer
FlatFileItemWriter	java.lang.Object	Attempts to convert the item to a String , Collection or array using an injected Converter and then

Item Writer	Type of Item Written	Description
		recurses. See [HOWTO - Write to a File]
HibernateAwareItemWriter	java.lang.Object	This item writer is hibernate session aware and handles some transaction-related work that a non-"hibernate aware" item writer would not need to know about and then delegates to another item writer to do the actual writing. See [HOWTO - Write to a Database]
StaxEventWriterItemWriter	java.lang.Object	Uses an ObjectToXmlSerializer implementation to convert each item to XML and then writes it to an XML file using StAX. See [HOWTO - Write to a File]

Appendix B. Meta-Data Schema

B.1. Overview

The Spring Batch Meta-Data tables very closely match the Domain objects that represent them in Java. For example, `JobInstance`, `JobExecution`, `JobParameters`, `StepExecution`, and `ExecutionContext` map to `BATCH_JOB_INSTANCE`, `BATCH_JOB_EXECUTION`, `BATCH_JOB_PARAMS`, `BATCH_STEP_EXECUTION`, `BATCH_STEP_EXECUTION_CONTEXT`, respectively. The `JobRepository` is responsible for saving and storing each Java object into its correct table. The following appendix describes the meta-data tables in detail, along with many of the design decisions that were made when creating them. When viewing the various table creation statements below, it is important to realize that the data types used are as generic as possible. Spring Batch provides many schemas as examples, which all have varying data types due to variations in individual database vendors' handling of data types. Below is an ERD model of all 5 tables and their relationships to one another:

B.1.1. Version

Many of the database tables discussed in this appendix contain a version column. This column is important because Spring Batch employs an optimistic locking strategy when dealing with updates to the database. This means that each time a record is 'touched' (updated) the value in the version column is incremented by one. When the repository goes back to try and save the value, if the version number has changed it will throw `OptimisticLockingFailureException`, indicating there has been an error with concurrent access. This check is necessary, since even though different batch jobs may be running in different machines, they are all using the same database tables.

B.1.2. Identity

`BATCH_JOB_INSTANCE`, `BATCH_JOB_EXECUTION`, and `BATCH_STEP_EXECUTION` each contain columns ending in `_ID`, which act as primary keys for their respective tables. However, they are not database generated keys, but rather are generated by separate sequences. This is necessary because after inserting one of the domain objects into the database, the key it is given needs to be set on the actual object, so that they can be uniquely identified in Java. Newer database drivers (Jdbc 3.0 and up) support this feature with database generated keys, but rather than requiring it, sequences were used. Each variation of the schema will contain some form of the following:

```
CREATE SEQUENCE BATCH_STEP_EXECUTION_SEQ;  
CREATE SEQUENCE BATCH_JOB_EXECUTION_SEQ;  
CREATE SEQUENCE BATCH_JOB_SEQ;
```

Many database vendors don't support sequences. In these cases, work arounds are used, such as the following for MySQL:

```
CREATE TABLE BATCH_STEP_EXECUTION_SEQ (ID BIGINT NOT NULL) type=MYISAM;  
INSERT INTO BATCH_STEP_EXECUTION_SEQ values(0);  
CREATE TABLE BATCH_JOB_EXECUTION_SEQ (ID BIGINT NOT NULL) type=MYISAM;  
INSERT INTO BATCH_JOB_EXECUTION_SEQ values(0);  
CREATE TABLE BATCH_JOB_SEQ (ID BIGINT NOT NULL) type=MYISAM;  
INSERT INTO BATCH_JOB_SEQ values(0);
```

In the above case, a table is used in place of each sequence. The Spring core class `MySQLMaxValueIncrementer`

will then increment the one column in this sequence in order to give similar functionality.

B.2. BATCH_JOB_INSTANCE

The BATCH_JOB_INSTANCE table holds all information relevant to a `JobInstance`, and serves as the top of the overall hierarchy. The following generic DDL statement is used to create it:

```
CREATE TABLE BATCH_JOB_INSTANCE (
  JOB_INSTANCE_ID BIGINT PRIMARY KEY ,
  VERSION BIGINT,
  JOB_NAME VARCHAR(100) NOT NULL ,
  JOB_KEY VARCHAR(2500)
);
```

Below are descriptions of each column in the table:

- **JOB_INSTANCE_ID:** The unique id that will identify the instance, which is also the primary key. The value of this column should be obtainable by calling the `getId` method on `JobInstance`.
- **VERSION:** See above section.
- **JOB_NAME:** Name of the job obtained from the `Job` object. Because it is required to identify the instance, it must not be null.
- **JOB_KEY:** A serialization of the `JobParameters` that uniquely identifies separate instances of the same job from one another. (`JobInstances` with the same job name

B.3. BATCH_JOB_PARAMS

The BATCH_JOB_PARAMS table holds all information relevant to the `JobParameters` object. It contains 0 or more key/value pairs that together uniquely identify a `JobInstance` and serve as a record of the parameters a job was run with. It should be noted that the table has been denormalized. Rather than creating a separate table for each type, there is one table with a column indicating the type:

```
CREATE TABLE BATCH_JOB_PARAMS (
  JOB_INSTANCE_ID BIGINT NOT NULL ,
  TYPE_CD VARCHAR(6) NOT NULL ,
  KEY_NAME VARCHAR(100) NOT NULL ,
  STRING_VAL VARCHAR(250) ,
  DATE_VAL TIMESTAMP DEFAULT NULL,
  LONG_VAL BIGINT ,
  DOUBLE_VAL DOUBLE PRECISION,
  constraint JOB_INSTANCE_PARAMS_FK foreign key (JOB_INSTANCE_ID)
  references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
);
```

Below are descriptions for each column:

- **JOB_INSTANCE_ID:** Foreign Key from the BATCH_JOB_INSTANCE table that indicates the job instance the parameter entry belongs to. It should be noted that multiple rows (i.e key/value pairs) may exist for each instance.
- **TYPE_CD:** String representation of the type of value stored, which can be either a string, date, long, or double. Because the type must be known, it cannot be null.

- **KEY_NAME**: The parameter key.
- **STRING_VAL**: Parameter value, if the type is string.
- **DATE_VAL**: Parameter value, if the type is date.
- **LONG_VAL**: Parameter value, if the type is a long.
- **DOUBLE_VAL**: Parameter value, if the type is double.

It is worth noting that there is no primary key for this table. This is simply because the framework has no use for one, and thus doesn't require it. If a user so chooses, one may be added with a database generated key, without causing any issues to the framework itself.

B.4. BATCH_JOB_EXECUTION

The `BATCH_JOB_EXECUTION` table holds all information relevant to the `JobExecution` object. Every time a `Job` is run there will always be a new `JobExecution`, and a new row in this table:

```
CREATE TABLE BATCH_JOB_EXECUTION (
  JOB_EXECUTION_ID BIGINT PRIMARY KEY ,
  VERSION BIGINT,
  JOB_INSTANCE_ID BIGINT NOT NULL,
  START_TIME TIMESTAMP DEFAULT NULL,
  END_TIME TIMESTAMP DEFAULT NULL,
  STATUS VARCHAR(10),
  CONTINUABLE CHAR(1),
  EXIT_CODE VARCHAR(20),
  EXIT_MESSAGE VARCHAR(2500),
  constraint JOB_INSTANCE_EXECUTION_FK foreign key (JOB_INSTANCE_ID)
  references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
) ;
```

Below are descriptions for each column:

- **JOB_EXECUTION_ID**: Primary key that uniquely identifies this execution. The value of this column is obtainable by calling the `getId` method of the `JobExecution` object.
- **VERSION**: See above section.
- **JOB_INSTANCE_ID**: Foreign key from the `BATCH_JOB_INSTANCE` table indicating the instance to which this execution belongs. There may be more than one execution per instance.
- **START_TIME**: Timestamp representing the time the execution was started.
- **END_TIME**: Timestamp representing the time the execution was finished, regardless of success or failure. An empty value in this column even though the job is not currently running indicates that there has been some type of error and the framework was unable to perform a last save before failing.
- **STATUS**: Character string representing the status of the execution. This may be `COMPLETED`, `STARTED`, etc. The object representation of this column is the `BatchStatus` enumeration.
- **CONTINUABLE**: Character indicating whether or not the execution is currently able to continue. 'Y' for yes and 'N' for no.
- **EXIT_CODE**: Character string representing the exit code of the execution. In the case of a command line job, this may be converted into a number.

- **EXIT_MESSAGE**: Character string representing a more detailed description of how the job exited. In the case of failure, this might include as much of the stack trace as is possible.

B.5. BATCH_STEP_EXECUTION

The `BATCH_STEP_EXECUTION` table holds all information relevant to the `StepExecution` object. This table is very similar in many ways to the `BATCH_JOB_EXECUTION` table and there will always be at least one entry per `Step` for each `JobExecution` created:

```
CREATE TABLE BATCH_STEP_EXECUTION (
  STEP_EXECUTION_ID BIGINT PRIMARY KEY ,
  VERSION BIGINT NOT NULL,
  STEP_NAME VARCHAR(100) NOT NULL,
  JOB_EXECUTION_ID BIGINT NOT NULL,
  START_TIME TIMESTAMP NOT NULL ,
  END_TIME TIMESTAMP DEFAULT NULL,
  STATUS VARCHAR(10),
  COMMIT_COUNT BIGINT ,
  ITEM_COUNT BIGINT ,
  CONTINUABLE CHAR(1),
  EXIT_CODE VARCHAR(20),
  EXIT_MESSAGE VARCHAR(2500),
  constraint JOB_EXECUTION_STEP_FK foreign key (JOB_EXECUTION_ID)
  references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;
```

Below are descriptions for each column:

- **STEP_EXECUTION_ID**: Primary key that uniquely identifies this execution. The value of this column should be obtainable by calling the `getId` method of the `StepExecution` object.
- **VERSION**: See above section.
- **STEP_NAME**: The name of the step to which this execution belongs.
- **JOB_EXECUTION_ID**: Foreign key from the `BATCH_JOB_EXECUTION` table indicating the `JobExecution` to which this `StepExecution` belongs. There may be only one `StepExecution` for a given `JobExecution` for a given `Step` name.
- **START_TIME**: Timestamp representing the time the execution was started.
- **END_TIME**: Timestamp representing the time the execution was finished, regardless of success or failure. An empty value in this column even though the job is not currently running indicates that there has been some type of error and the framework was unable to perform a last save before failing.
- **STATUS**: Character string representing the status of the execution. This may be `COMPLETED`, `STARTED`, etc. The object representation of this column is the `BatchStatus` enumeration.
- **COMMIT_COUNT**: The number of times in which the step has committed a transaction during this execution.
- **ITEM_COUNT**: The number of items that have been written out during this execution.
- **CONTINUABLE**: Character indicating whether or not the execution is currently able to continue. 'Y' for yes and 'N' for no.
- **EXIT_CODE**: Character string representing the exit code of the execution. In the case of a command line

job, this may be converted into a number.

- **EXIT_MESSAGE:** Character string representing a more detailed description of how the job exited. In the case of failure, this might include as much of the stack trace as is possible.

B.6. BATCH_STEP_EXECUTION_CONTEXT

The `BATCH_STEP_EXECUTION_CONTEXT` table holds all information relevant to an `ExecutionContext`. There is exactly one `ExecutionContext` per `StepExecution`, and it contains all user defined key/value pairs that need to be persisted for a particular job run. This data is typically state that must be retrieved back after a failure so that a `JobInstance` can 'start from where it left off'. As with the `BATCH_JOB_PARAMS` table, this table has been denormalized and uses a column to determine the type:

```
CREATE TABLE BATCH_STEP_EXECUTION_CONTEXT (
  STEP_EXECUTION_ID BIGINT NOT NULL ,
  TYPE_CD VARCHAR(6) NOT NULL ,
  KEY_NAME VARCHAR(1000) NOT NULL ,
  STRING_VAL VARCHAR(1000) ,
  DATE_VAL TIMESTAMP DEFAULT NULL ,
  LONG_VAL VARCHAR(10) ,
  DOUBLE_VAL DOUBLE PRECISION ,
  OBJECT_VAL BLOB,
  constraint STEP_EXECUTION_CONTEXT_FK foreign key (STEP_EXECUTION_ID)
  references BATCH_STEP_EXECUTION(STEP_EXECUTION_ID)
) ;
```

Below are descriptions for each column:

- **STEP_EXECUTION_ID:** Foreign key representing the `StepExecution` to which the context belongs. There may be more than one row associated to a given `StepExecution`.
- **TYPE_CD:** String representation of the type of value stored, which can be either a character string, date, long, or double. Because the type must be known, it cannot be null.
- **KEY_NAME:** The Parameter key.
- **STRING_VAL:** Parameter value, if the type is string.
- **DATE_VAL:** Parameter value, if the type is date.
- **LONG_VAL:** Parameter value, if the type is a long.
- **DOUBLE_VAL:** Parameter value, if the type is double.
- **OBJECT_VAL:** Parameter value, if the type is a blob.

When an `ExecutionContext` is stored, values that are one of the well known types above will be stored as their respective type. Any unknown type will be serialized to a blob and stored in the `OBJECT_VAL` column. As with `BATCH_JOB_PARAMS`, there is no primary key for this table. This is simply because the framework has no use for one, and thus doesn't require it. If a user so chooses, one may be added with a database generated key, without causing any issues to the framework itself.

B.7. Archiving

Because there are entries in multiple tables everytime a batch job is run, it is common to create an archive strategy for the meta-data tables. The tables themselves are designed to show a record of what happened in the past, and generally won't affect the run of any job, with a couple of notable exceptions:

- **Restart:** Because the `ExecutionContext` is persisted, removing any entries from this table of jobs that haven't completed successfully, will prevent them from starting at the correct point if run again. Furthermore, if an entry for a `JobInstance` is removed without having completed successfully, the framework will think that the job is new, rather than a restart.
- **Determining if an instance has been run:** The framework will use the meta-data tables to determine if a particular `JobInstance` has been run before, and if it has an exception will be thrown.

Glossary

Spring Batch Glossary

Batch	An accumulation of business transactions over time.
Batch Application Style	Term used to designate batch as an application style in its own right similar to online, Web or SOA. It has standard elements of input, validation, transformation of information to business model, business processing and output. In addition, it requires monitoring at a macro level.
Batch Processing	The handling of a batch of many business transactions that have accumulated over a period of time (e.g. an hour, day, week, month, or year). It is the application of a process, or set of processes, to many data entities or objects in a repetitive and predictable fashion with either no manual element, or a separate manual element for error processing.
Batch Window	The time frame within which a batch job must complete. This can be constrained by other systems coming online, other dependent jobs needing to execute or other factors specific to the batch environment.
Step	It is the main batch task or unit of work controller. It initializes the business logic, and controls the transaction environment based on commit interval setting, etc.
Tasklet	An application program created by application developer to process the business logic for an entire Step.
Batch Job Type	Job Types describe application of jobs for particular type of processing. Common areas are interface processing (typically flat files), forms processing (either for online pdf generation or print formats), report processing.
Driving Query	A driving query identifies the set of work for a job to do; the job then breaks that work into individual units of work. For instance, identify all financial transactions that have a status of "pending transmission" and send them to our partner system. The driving query returns a set of record IDs to process; each record ID then becomes a unit of work. A driving query may involve a join (if the criteria for selection falls across two or more tables) or it may work with a single table.
Item	An item represents the smallest amount of complete data for processing. In the most simple terms this might mean a line in a file, a row in a database table, or a particular element in an XML file.
Logical Unit of Work (LUW)	A batch job iterates through a driving query (or another input source such as a file) to perform the set of work that the job must accomplish. Each iteration of work performed is a unit of work.
Commit Interval	A set of LUWs constitute a commit interval.
Partitioning	Splitting a job into multiple threads where each thread is responsible for a subset of the overall data to be processed. The threads of execution may be

within the same JVM or they may span JVMs in a clustered environment that supports workload balancing.

Staging Table	A table that holds temporary data while it is being processed.
Restartable	- a job that can be executed again and will assume the same identity as when run initially. In othewords, it is has the same job instance id.
Rerunnable	a job that is restartable and manages it's own state in terms of previous run's record processing. An example of a rerunnable step is one based on a driving query. If the driving query can be formed so that it will limit the processed rows when the job is restarted than it is re-runnable. This is managed by the application logic. Often times a condition is added to the where statement to limit the rows returned by the driving query with something like "and processedFlag != true".
Repeat	One of the most basic units of batch processing, that defines repeatability calling a portion of code until it is finished, and while there is no error. Typically a batch process would be repeatable as long as there is input.
Retry	Simplifies the execution of operations with retry semantics most frequently associated with handling transactional output exceptions. Retry is slightly different from repeat, rather than continually calling a block of code, retry is stateful, and continually calls the same block of code with the same input, until it either succeeds, or some type of retry limit has been exceeded. It is only generally useful if the operation is non-deterministic meaning that a retry on a subsequent invocation might succeed because something in the environment has improved.
Recover	Recover operations handle an exception in such a way that a repeat process is able to continue.
Skip	Skip is a recovery strategy often used on file input sources as the strategy for ignoring bad input records that failed validation.