Table of Contents

Chapter 1. The Domain Language of Batch

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 `ItemReader` and `ItemWriter`. 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 allow for quick adoption and ease of use out-of-the-box.
- Significantly enhanced extensibility.

The following diagram is a simplified version of the batch reference architecture that has been used for decades. It provides an overview of the components that make up the domain language of batch processing. 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 the robust, maintainable systems that are used to address the creation of simple to complex batch applications, with the infrastructure and extensions to address very complex processing needs.

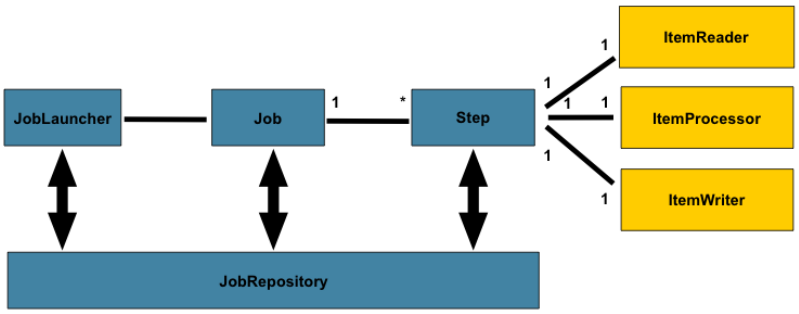

Figure 1. Batch Stereotypes

The preceding diagram highlights the key concepts that make up the domain language of Spring Batch. A Job has one to many steps, each of which has exactly one `ItemReader`, one `ItemProcessor`, and one `ItemWriter`. A job needs to be launched (with `JobLauncher`), and metadata about the currently running process needs to be stored (in `JobRepository`).

1.1. Job

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` is wired together with either an XML configuration file or Java-based configuration. This configuration may be referred to as the "job configuration". However, `Job` is just the top of an overall hierarchy, as shown in the following diagram:

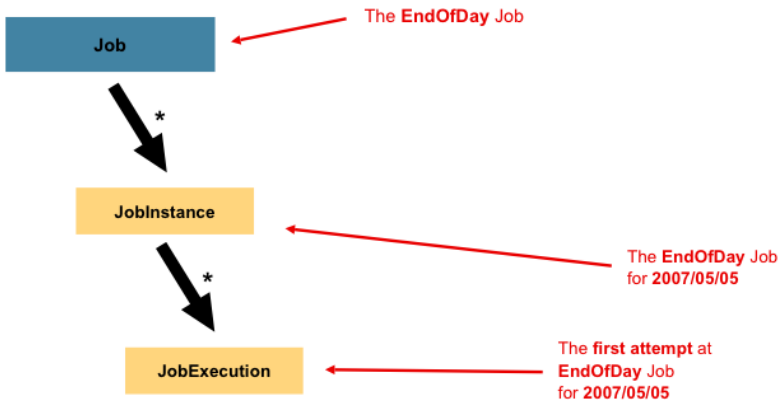

Figure 2. Job Hierarchy

In Spring Batch, a `Job` is simply a container for `Step` instances. It combines multiple steps that belong logically together in a flow and allows for configuration of properties global to all steps, such as restartability. The job configuration contains:

- The simple name of the job.
- Definition and ordering of `Step` instances.
- 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`. When using java based configuration, a collection of builders are made available for the instantiation of a `Job`, as shown in the following example:

```
@Bean
public Job footballJob() {
    return this.jobBuilderFactory.get("footballJob")
                     .start(playerLoad())
                     .next(gameLoad())
                     .next(playerSummarization())
                     .end()
                     .build();
}
```

However, when using XML configuration, the batch namespace abstracts away the need to instantiate it directly. Instead, the `<job>` tag can be used as shown in the following example:

```
<job id="footballJob">
    <step id="playerload" next="gameLoad"/>
    <step id="gameLoad" next="playerSummarization"/>
    <step id="playerSummarization"/>
</job>
```

1.1.1. JobInstance

A `JobInstance` refers to the concept of a logical job run. Consider a batch job that should be run once at the end of the day, such as the 'EndOfDay' `Job` from the preceding diagram. There is one 'EndOfDay' job, but each individual run of the `Job` must be tracked separately. In the case of this job, there is one logical `JobInstance` per day. For example, there is a January 1st run, a January 2nd run, and so on. If the January 1st run fails the first time and is run again the next day, it is still the January 1st run. (Usually, this corresponds with the data it is processing as well, meaning the January 1st run processes data for January 1st). Therefore, each `JobInstance` can have multiple executions (`JobExecution` is discussed in more detail later in this chapter), and only one `JobInstance` corresponding to a particular `Job` and identifying `JobParameters` can run at a given time.

The definition of a `JobInstance` has absolutely no bearing on the data the to be loaded. It is entirely up to the `ItemReader` implementation to determine how data is 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 load only data from the 1st, and the January 2nd run would use only data from the 2nd. Because this determination is likely to be a business decision, it is left up to the `ItemReader` to decide. However, using the same `JobInstance` determines whether or not the 'state' (that is, the `ExecutionContext`, which is discussed later in this chapter) from previous executions is used. Using a new `JobInstance` means 'start from the beginning', and using an existing instance generally means 'start from where you left off'.

1.1.2. 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`. A `JobParameters` object holds a set of parameters used to start a batch job. They can be used for identification or even as reference data during the run, as shown in the following image:

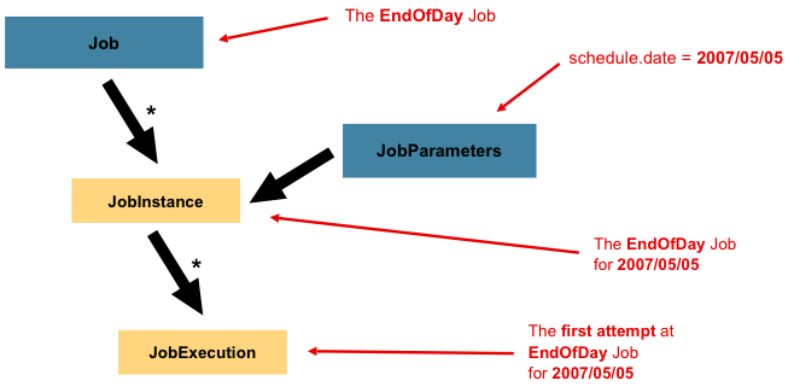

Figure 3. Job Parameters

In the preceding example, where there are two instances, one for January 1st, and another for January 2nd, there is really only one `Job`, but it has two `JobParameter` objects: one that was started with a job parameter of 01-01-2017 and another that was started with a parameter of 01-02-2017. Thus, the contract can be defined as: `JobInstance` = `Job` + identifying `JobParameters`. This allows a developer to effectively control how a `JobInstance` is defined, since they control what parameters are passed in.

Not all job parameters are required to contribute to the identification of a `JobInstance`. By default, they do so. However, the framework also allows the submission of a `Job` with parameters that do not contribute to the identity of a `JobInstance`.

1.1.3. 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 is not considered to be complete unless the execution completes successfully. Using the EndOfDay `Job` described previously as an example, consider a `JobInstance` for 01-01-2017 that failed the first time it was run. If it is run again with the same identifying job parameters as the first run (01-01-2017), a new `JobExecution` is created. However, there is still only one `JobInstance`.

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

Table 1. JobExecution Properties

Property	Definition
Status	A `BatchStatus` object that indicates the status of the execution. While running, it is `BatchStatus#STARTED`. If it fails, it is `BatchStatus#FAILED`. If it finishes successfully, it is `BatchStatus#COMPLETED`
startTime	A `java.util.Date` representing the current system time when the execution was started. This field is empty if the job has yet to start.
endTime	A `java.util.Date` representing the current system time when the execution finished, regardless of whether or not it was successful. The field is empty if the job has yet to finish.
exitStatus	The `ExitStatus`, indicating the result of the run. It is most important, because it contains an exit code that is returned to the caller. See chapter 5 for more details. The field is empty if the job has yet to finish.
createTime	A `java.util.Date` representing the current system time when the `JobExecution` was first persisted. The job may not have been started yet (and thus has no start time), but it always has a createTime, which is required by the framework for managing job level `ExecutionContexts`.
lastUpdated	A `java.util.Date` representing the last time a `JobExecution` was persisted. This field is empty if the job has yet to start.

executionContext	The "property bag" containing any user data that needs to be persisted between executions.
failureExceptions	The list of exceptions encountered during the execution of a `Job`. These can be useful if more than one exception is encountered during the failure of a `Job`.

These properties are important because they are 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 are made in the batch metadata tables:

Table 2. BATCH_JOB_INSTANCE

JOB_INST_ID	JOB_NAME
1	EndOfDayJob

Table 3. BATCH_JOB_EXECUTION_PARAMS

JOB_EXECUTION_ID	TYPE_CD	KEY_NAME	DATE_VAL	IDENTIFYING
1	DATE	schedule.Date	2017-01-01	TRUE

Table 4. BATCH_JOB_EXECUTION

JOB_EXEC_ID	JOB_INST_ID	START_TIME	END_TIME	STATUS
1	1	2017-01-01 21:00	2017-01-01 21:30	FAILED

Column names may have been abbreviated or removed for the sake of clarity and formatting.

Now that the job has failed, assume that it took the entire night for the problem to be determined, so that the 'batch window' is now closed. Further assuming that the window starts at 9:00 PM, the job is kicked off again for 01-01, starting where it left off and completing successfully at 9:30. Because it is now the next day, the 01-02 job must be run as well, and it 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 a `Job` should be run. Since they are separate `JobInstances`, Spring Batch makes no attempt to stop them from being run concurrently. (Attempting to run the same `JobInstance` while another is already running results 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, as shown in the following tables:

Table 5. BATCH_JOB_INSTANCE

JOB_INST_ID	JOB_NAME
1	EndOfDayJob
2	EndOfDayJob

Table 6. BATCH_JOB_EXECUTION_PARAMS

JOB_EXECUTION_ID	TYPE_CD	KEY_NAME	DATE_VAL	IDENTIFYING
1	DATE	schedule.Date	2017-01-01 00:00:00	TRUE
2	DATE	schedule.Date	2017-01-01 00:00:00	TRUE
3	DATE	schedule.Date	2017-01-02 00:00:00	TRUE

Table 7. BATCH_JOB_EXECUTION

JOB_EXEC_ID	JOB_INST_ID	START_TIME	END_TIME	STATUS
1	1	2017-01-01 21:00	2017-01-01 21:30	FAILED
2	1	2017-01-02 21:00	2017-01-02 21:30	COMPLETED
3	2	2017-01-02 21:31	2017-01-02 22:29	COMPLETED

Column names may have been abbreviated or removed for the sake of clarity and formatting.

1.2. Step

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` 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. As with a `Job`, a `Step` has an individual `StepExecution` that correlates with a unique `JobExecution`, as shown in the following image:

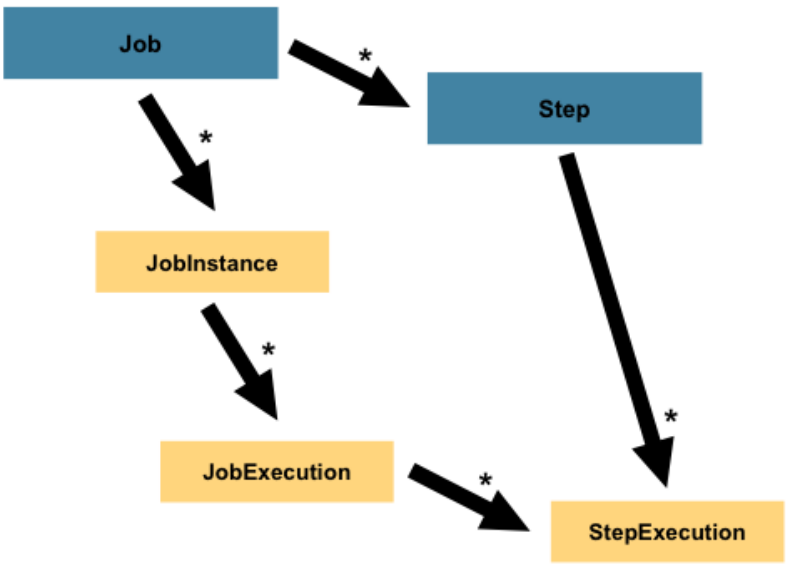

Figure 4. Job Hierarchy With Steps

1.2.1. StepExecution

A `StepExecution` represents a single attempt to execute a `Step`. A new `StepExecution` is created each time a `Step` is run, similar to `JobExecution`. However, if a step fails to execute because the step before it fails, no execution is persisted for it. A `StepExecution` is created only when its `Step` is actually started.

`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 counts and start and end times. Additionally, each step execution contains an `ExecutionContext`, which contains any data a developer needs to have persisted across batch runs, such as statistics or state information needed to restart. The following table lists the properties for `StepExecution`:

Table 8. StepExecution Properties

Property	Definition
Status	A `BatchStatus` object that indicates the status of the execution. While running, the status is `BatchStatus.STARTED`. If it fails, the status is `BatchStatus.FAILED`. If it finishes successfully, the status is `BatchStatus.COMPLETED`.
startTime	A `java.util.Date` representing the current system time when the execution was started. This field is empty if the step has yet to start.
endTime	A `java.util.Date` representing the current system time when the execution finished, regardless of whether or not it was successful. This field is empty if the step has yet to exit.
exitStatus	The `ExitStatus` indicating the result of the execution. It is most important, because it contains an exit code that is returned to the caller. See chapter 5 for more details. This field is empty if the job has yet to exit.
executionContext	The "property bag" containing any user data that needs to be persisted between executions.
readCount	The number of items that have been successfully read.
writeCount	The number of items that have been successfully written.
commitCount	The number of transactions that have been committed for this execution.
rollbackCount	The number of times the business transaction controlled by the `Step` has been rolled back.
readSkipCount	The number of times `read` has failed, resulting in a skipped item.

processSkipCount	The number of times `process` has failed, resulting in a skipped item.
filterCount	The number of items that have been 'filtered' by the `ItemProcessor`.
writeSkipCount	The number of times `write` has failed, resulting in a skipped item.

1.3. 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` object or a `JobExecution` object. For those familiar with Quartz, it is very similar to JobDataMap. The best usage example is to facilitate restart. Using flat file input as an example, while processing individual lines, the framework periodically persists the `ExecutionContext` at commit points. Doing so 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, as shown in the following example, and the framework will do the rest:

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

Using the EndOfDay example from the `Job` Stereotypes section as an example, assume there is one step, 'loadData', that loads a file into the database. After the first failed run, the metadata tables would look like the following example:

Table 9. BATCH_JOB_INSTANCE

JOB_INST_ID	JOB_NAME
1	EndOfDayJob

Table 10. BATCH_JOB_EXECUTION_PARAMS

JOB_INST_ID	TYPE_CD	KEY_NAME	DATE_VAL
1	DATE	schedule.Date	2017-01-01

Table 11. BATCH_JOB_EXECUTION

JOB_EXEC_ID	JOB_INST_ID	START_TIME	END_TIME	STATUS
1	1	2017-01-01 21:00	2017-01-01 21:30	FAILED

Table 12. BATCH_STEP_EXECUTION

STEP_EXEC_ID	JOB_EXEC_ID	STEP_NAME	START_TIME	END_TIME	STATUS
1	1	loadData	2017-01-01 21:00	2017-01-01 21:30	FAILED

Table 13. BATCH_STEP_EXECUTION_CONTEXT

STEP_EXEC_ID	SHORT_CONTEXT

1	{piece.count=40321}

In the preceding case, the `Step` ran for 30 minutes and processed 40,321 'pieces', which would represent lines in a file in this scenario. This value is updated just before each commit by the framework and can contain multiple rows corresponding to entries within the `ExecutionContext`. Being notified before a commit requires one of the various `StepListener` implementations (or an `ItemStream`), which are discussed in more detail later in this guide. As with the previous example, it is assumed that the `Job` is restarted the next day. When it is restarted, the values from the `ExecutionContext` of the last run are reconstituted from the database. When the `ItemReader` is opened, it can check to see if it has any stored state in the context and initialize itself from there, as shown in the following example:

```
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();
    }
}
```

In this case, after the above code runs, the current line is 40,322, allowing the `Step` to start again from where it left off. 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 the number of lines read), so that an email can be sent at the end of the `Step` with the total number of 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. As a result, 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.

It is also important to note that there is at least one `ExecutionContext` per `JobExecution` and one for every `StepExecution`. For example, consider the following code snippet:

```
ExecutionContext ecStep = stepExecution.getExecutionContext();
ExecutionContext ecJob = jobExecution.getExecutionContext();
//ecStep does not equal ecJob
```

As noted in the comment, `ecStep` does not equal `ecJob`. They are two different `ExecutionContexts`. The one scoped to the `Step` is saved at every commit point in the `Step`, whereas the one scoped to the Job is saved in between every `Step` execution.

1.4. JobRepository

`JobRepository` is the persistence mechanism for all of the Stereotypes mentioned above. It provides CRUD operations for `JobLauncher`, `Job`, and `Step` implementations. When a `Job` is first launched, a `JobExecution` is obtained from the repository, and, during the course of execution, `StepExecution` and `JobExecution` implementations are persisted by passing them to the repository.

The batch namespace provides support for configuring a `JobRepository` instance with the `<job-repository>` tag, as shown in the following example:

```
<job-repository id="jobRepository"/>
```

When using java configuration, `@EnableBatchProcessing` annotation provides a `JobRepository` as one of the components automatically configured out of the box.

1.5. JobLauncher

`JobLauncher` represents a simple interface for launching a `Job` with a given set of `JobParameters`, as shown in the following example:

```
public interface JobLauncher {

public JobExecution run(Job job, JobParameters jobParameters)
            throws JobExecutionAlreadyRunningException, JobRestartException,
                   JobInstanceAlreadyCompleteException, JobParametersInvalidException;
}
```

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

1.6. 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 indicates this by returning `null`. More details about the `ItemReader` interface and its various implementations can be found in Readers And Writers.

1.7. Item Writer

`ItemWriter` is an abstraction that represents the output of a `Step`, one batch or chunk of items at a time. Generally, an `ItemWriter` has no knowledge of the input it should receive next and knows only the item that was passed in its current invocation. More details about the `ItemWriter` interface and its various implementations can be found in Readers And Writers.

1.8. Item Processor

`ItemProcessor` is an abstraction that represents the business processing of an item. While the `ItemReader` reads one item, and the `ItemWriter` writes them, the `ItemProcessor` provides an access point to transform or apply other business processing. If, while processing the item, it is determined that the item is not valid, returning `null` indicates that the item should not be written out. More details about the `ItemProcessor` interface can be found in Readers And Writers.

1.9. Batch Namespace

Many of the domain concepts listed previously need to be configured in a Spring `ApplicationContext`. While there are implementations of the interfaces above that can be used in a standard bean definition, a namespace has been provided for ease of configuration, as shown in the following example:

```
<beans:beans xmlns="http://www.springframework.org/schema/batch"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/batch
   http://www.springframework.org/schema/batch/spring-batch.xsd">

<job id="ioSampleJob">
    <step id="step1">
        <tasklet>
            <chunk reader="itemReader" writer="itemWriter" commit-interval="2"/>
        </tasklet>
    </step>
</job>

</beans:beans>
```

As long as the batch namespace has been declared, any of its elements can be used. More information on configuring a Job can be found in Configuring and Running a Job. More information on configuring a `Step` can be found in Configuring a Step.