Job Repository and Metadata

When you look into writing a batch process, the ability to execute processes without a UI in a stand-alone manner isn’t that hard. When you dig into Spring Batch, the execution of a job amounts to nothing more than using an implementation of Spring’s TaskExecutor to run a separate task. You don’t need Spring Batch to do that.

Where things get interesting, however, is when things go wrong. If your batch job is running and an error occurs, how do you recover? How does your job know where it was in processing when the error occurred, and what should happen when the job is restarted? State management is an important part of processing large volumes of data. This is one of the key features that Spring Batch brings to the table. Spring Batch, as discussed previously in this book, maintains the state of a job as it executes in a job repository. It then uses this information when a job is restarted or an item is retried to determine how to continue. The power of this feature can’t be overstated.

Another aspect of batch processing in which the job repository is helpful is monitoring. The ability to see how far a job is in its processing as well as trend elements such as how long operations take or how many items were retried due to errors is vital in the enterprise environment. The fact that Spring Batch does the number gathering for you makes this type of trending much easier.

This chapter covers job repositories in detail. It goes over ways to configure a job repository for most environments by using either a database or an in-memory repository. You also look briefly at performance impacts on the configuration of the job repository. After you have the job repository configured, you learn how to put the job information stored by the job repository to use using the JobExplorer and the JobOperator.

Configuring the Job Repository

In order for Spring Batch to be able to maintain state, the job repository needs to be available. Spring offers two options by default: an in-memory repository and a persisted repository in a database. This section looks at how to configure each of those options as well as the performance impacts of both options. Let’s start with more simpler option, the in-memory job repository.

Using an In-Memory Job Repository

The opening paragraphs of this chapter laid out a list of benefits for the job repository, such as the ability to maintain state from execution to execution and trend run statistics from run to run. However, you’ll almost never use an in-memory repository for those reasons. That’s because when the process ends, all of that data is lost. So, why would you use an in-memory repository at all?

The answer is that sometimes you don’t need to persist the data. For example, in development, it’s common to run jobs with an in-memory repository so that you don’t have to worry about maintaining the job schema in a database. This also allows you to execute the same job multiple times with the same
parameters, which is a must-have in development. You might also run a job using the in-memory repository for performance reasons. There is a cost to maintaining job state in a database that may not be needed. Say, for instance, that you’re using Spring Batch to do a data migration, moving data from one database table to another; the destination table is empty to start, and you have a small amount of data to migrate. In a case like this, the overhead of setting up a Spring Batch schema and using it may not make sense. Situations that don’t need Spring Batch to manage restarts and so on can use the in-memory option.

The JobRepository you’ve been using so far is configured in the launch-context.xml file. In the previous examples, you’ve configured the job repository using MySQL. To configure your job to use an in-memory repository, you use the org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean, as shown in Listing 5-1. Notice that a transaction manager is still required. This is because the data the JobRepository stores is still dependent on transaction semantics (rollback, and so on), and business logic may depend on transactional stores as well. The transaction manager configured in the listing, org.springframework.batch.support.transaction.ResourcelessTransactionManager, actually doesn’t do anything with transactions; it’s a dummy transaction manager that provides a dummy transaction.

Listing 5-1. Configuring an In-Memory Job Repository

```xml
<?xml version="1.0" encoding="UTF-8"?>
<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-3.0.xsd
                                   http://www.springframework.org/schema/batch
                                   http://www.springframework.org/schema/batch/spring-batch-2.1.xsd">
  <beans:bean id="transactionManager"
              class="org.springframework.batch.support.transaction.
                     ResourcelessTransactionManager"/>

  <beans:bean id="jobRepository"
              class="org.springframework.batch.core.repository.support.
                     MapJobRepositoryFactoryBean"
              p:transactionManager-ref="transactionManager" />

...</beans:beans>
```

If you take the HelloWorld example from Chapter 2 and configure it to use the in-memory elements of Listing 5-1, you see that you can run the job over and over without Spring Batch throwing an exception for running the same job with the same parameters multiple times.

You should keep in mind a couple of limitations when using an in-memory job repository. First, as already stated, because the storage of data is in memory, once a JVM is restarted the data is lost. Second, because synchronization occurs in the memory space of a particular JVM, there is no guarantee that a given job won’t be executed with the same given parameters by executing the same job in two JVMs. Finally, if your job is using any of the multithreading options (multithreaded step, parallel flows, and so on) provided by Spring Batch, this option won’t work.

That’s it for the in-memory option. By making a small configuration tweak, you can prevent yourself from having to deal with setting up a database to run your batch jobs. However, given the limitations of this approach and the features that a persistent job repository provides, most of the time you’ll use a database to back your job repository. With that in mind, let’s look at configuring the job repository in a database.