Running a Job

Normally, you don’t have to think about how to run an application in Java. If you have a web application, you run it in some form of container. To run the application, you start the container, which starts the application. If you want to run a stand-alone Java program, you either create an executable jar file or call the class directly. In either case you might write a shell script to launch the process.

However, running a batch job is different. This is partially because a batch job can be run either as a thread within an existing process (as it has been up to now) or within the main execution thread. It can be run within a container or as a stand-alone process. You can start a JVM with each execution, or you can have a JVM loaded and call into it via something like JMX to launch the job (as you did in Chapter 5).

You also have the consideration of what should happen when things go wrong and your job stops. Does the entire job need to be rerun, or can you start at the step where it left off? If the step is processing a million rows, do they all need to be reprocessed, or can you restart at the chunk where the error occurred?

With all this to think about when running a batch job, this chapter covers how to start a job in a variety of environments. It discusses the different job runners provided with the Spring Batch framework, as well as integrating the starting and running of jobs with a container like Tomcat and a scheduler like Quartz.

Running a job isn’t all you learn about here. You also see how to programmatically stop a job once it has begun in a way that allows it to be restarted. Finally, you finish this chapter by seeing what it takes to be able to restart a job.

Starting a Job

In the chapters up to now, you’ve almost exclusively run a job each time you start a JVM. However, when you execute a job as you have been with SimpleJobLauncher, things are a little more complex than meets the eye. This section looks at what happens when you launch a job via the SimpleJobLauncher. You then take a detailed look at all the job runners and launchers Spring Batch provides. You see how to execute jobs in a variety of environments, including from within a servlet container, using the Spring Batch Admin administration application and via the open source scheduler Quartz.

Job Execution

When you think about launching a batch job in Spring Batch, you may think that what is happening is Spring Batch executing the job as part of the main execution thread. When it finishes, the process ends. However, it isn’t that simple. The org.springframework.batch.core.launch.JobLauncher interface, which is responsible for the work of starting a job, can be implemented in a number of ways, exposing any number of execution options (web, JMX, command line, and so on).
Because the JobLauncher interface doesn’t guarantee whether a job is run synchronously or asynchronously, SimpleJobLauncher (the only JobLauncher implementation provided by Spring Batch) leaves it up to the developer by running the job in Spring’s TaskExecutor. By default, SimpleJobLauncher uses Spring’s SyncTaskExecutor, which executes the job in the current thread. Although this is ok in many instances, this option is a limiting factor for the number of jobs you run within a single JVM.

Let’s look at how SimpleJobLauncher is configured. Listing 6-1 shows its configuration with the optional taskExecutor property set. This property allows you, as said previously, to specify the algorithm used for launching jobs. In this case, you’re using Spring’s SimpleAsyncTaskExecutor to launch the job in a new thread. However, you can easily configure this to use ThreadPoolTaskExecutor to control the number of threads available.

Listing 6-1. SimpleJobLauncher Configured with a Task Executor

```xml
<bean id="taskExecutor"
     class="org.springframework.core.task.SimpleAsyncTaskExecutor"/>

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

Although the JobLauncher kicks off the job and defines how it’s run (synchronously, asynchronously, in a thread pool, and so on), it’s the job runner that you interact with when you want to launch a job (as you did in the last chapter’s JMX job runner). Next you look at the two job runners that Spring Batch provides out of the box: CommandLineJobRunner and JobRegistryBackgroundJobRunner.

Spring Batch Job Runners

When you look at the Spring Batch API, although theoretically there are many ways to run a job (launch it via a servlet, the command line, JMX, and so on), the framework provides only the two runners org.springframework.batch.core.launch.support.CommandLineJobRunner and org.springframework.batch.core.launch.support.JobRegistryBackgroundJobRunner. All other options—servlet, JMX, and so forth—must be custom developed. Let’s look at how CommandLineJobRunner and JobRegistryBackgroundJobRunner are used and why, starting with CommandLineJobRunner.

**CommandLineJobRunner**

CommandLineJobRunner is the runner you’ve been using up to now. It serves as an interface for your batch processes via the command line. It’s useful for calling jobs from either a terminal or, more commonly, shell scripts. It provides four features:

- Loads the appropriate ApplicationContext based on the passed parameter
- Parses command-line arguments into a JobParameters object
- Locates the job requested based on the parameters passed
- Uses the configured JobLauncher to execute the requested job