Testing Batch Processes

Testing: everyone’s favorite part of programming. The funny thing is, like most things in life, once you get good at it, testing actually is fun. It allows you to be more productive. It provides a safety net for you to try new things. Programmatic tests also give you a test bed to try new technologies (most companies don’t mind if you want to try something new in the tests but mind greatly if you try it in code that’s going to production). You’ve spent the previous 10 chapters writing code without the ability to prove that any of it works. This chapter looks at how to exercise your code in a variety of ways so you can not only prove that it works as designed, but also provide a safety net for when you change it.

This chapter covers the following topics:

- **Unit tests with JUnit and Mockito**: You begin with a high-level overview of the JUnit and Mockito frameworks. Although you move past JUnit’s base functionality in the later parts of the chapter, the concepts that Spring has incorporated into its testing apparatus are based in the JUnit conventions, so knowing them helps you understand what is going on in the more advanced tests. The chapter also covers how the mock object framework Mockito can help you unit-test the components you develop for your batch processes.

- **Integration testing with Spring’s test framework**: Spring has done to testing what it’s done to most other harder Java tasks: made it easy. It provides a collection of classes that allow you to easily test interactions with your various resources (databases, files, and so on) with minimal overhead. You learn how to use the Spring testing components to test various aspects of your Spring Batch jobs.

The most fundamental aspect of testing begins with unit testing, so the discussion begins there.

**Unit Tests with JUnit and Mockito**

Probably the easiest to write and perhaps the most valuable, unit tests are the most overlooked type of testing. Although the development done in this book hasn’t taken a test-driven approach for a number of reasons, you’re encouraged to do so in your own development. As a proven way to improve not only the quality of the software you produce but also the overall productivity of any individual developer and a team as a whole, the code encased in these tests is some of the most valuable you can produce. This section looks at how to use JUnit and Mockito to unit-test the components you develop for your batch processes.

What is a unit test? It’s a test of a single, isolated component in a repeatable way. Let’s break down that definition to understand how it applies to what you’re trying to do:

- **A test of a single**: One. Unit tests are intended to test the smallest building blocks of your application. A single method is typically the scope of a unit test.
• *Isolated:* Dependencies can wreak havoc on the testing of a system. Yet all systems have dependencies. The goal of a unit test isn't to test your integration with each of these dependencies, but to instead test how your component works by itself.

• *In a repeatable way:* When you fire up a browser and click through your application, it isn't a repeatable exercise. You may enter different data each time. You may click the buttons in a slightly different order. Unit tests should be able to repeat the exact same scenario time and time again. This allows you to use them to regression-test as you make changes in your system.

The frameworks you use to execute the isolated testing of your components in a repeatable way are JUnit, Mockito, and the Spring framework. The first two are common, multipurpose frameworks that are useful for creating unit tests for your code. The Spring test utilities are helpful for testing more broad concerns including the integration of the different layers and even testing job execution from end to end (from a service or Spring Batch component to the database and back).

**JUnit**

Considered the gold standard for testing frameworks in Java,¹ JUnit is a simple framework that provides the ability to unit-test Java classes in a standard way. Whereas most frameworks you work with require add-ons to things like your IDE and build process, Maven and most Java IDEs have JUnit support built in with no additional configuration required. Entire books have been written on the topic of testing and using frameworks like JUnit, but it's important to quickly review these concepts. This section looks at JUnit and its most commonly used features.

The current version of JUnit as of the writing of this book is JUnit 4.8.2. Although each revision contains marginal improvements and bug fixes, the last major revision of the framework was the move from JUnit 3 to JUnit 4, which introduced annotations to configure test cases. Test cases? Let's step back a minute and go over how JUnit test are structured.

**JUnit Lifecycle**

JUnit tests are broken down into what are called *test cases*. Each test case is intended to test a particular piece of functionality, with the common divisor being at the class level. The common practice is to have at least one test case for each class. A test case is nothing more than a Java class configured with JUnit annotations to be executed by JUnit. In a test case exist both test methods and methods that are executed to set preconditions and clean up post conditions after each test or group of tests. Listing 12-1 shows a very basic JUnit test case.

**Listing 12-1. A Basic JUnit Test Case**

```java
package com.apress.springbatch.chapter12;

import org.junit.Test;
import static org.junit.Assert.*;

public class StringTest {

    // your test methods here
}
```

¹ Or at least it won the Betamax versus VHS wars against frameworks like TestNG and others.