Table of Contents

Chapter 1. Unit Testing

As with other application styles, it is extremely important to unit test any code written as part of a batch job. The Spring core documentation covers how to unit and integration test with Spring in great detail, so it is not be repeated here. It is important, however, to think about how to 'end to end' test a batch job, which is what this chapter covers. The spring-batch-test project includes classes that facilitate this end-to-end test approach.

1.1. Creating a Unit Test Class

In order for the unit test to run a batch job, the framework must load the job's `ApplicationContext`. Two annotations are used to trigger this behavior:

- `@RunWith(SpringRunner.class)`: Indicates that the class should use Spring's JUnit facilities
- `@ContextConfiguration(…)`: Indicates which resources to configure the `ApplicationContext` with.

Starting from v4.1, it is also possible to inject Spring Batch test utilities like the `JobLauncherTestUtils` and `JobRepositoryTestUtils` in the test context using the `@SpringBatchTest` annotation.

The following example shows the annotations in use:

Using Java Configuration

```
@SpringBatchTest
@RunWith(SpringRunner.class)
@ContextConfiguration(classes=SkipSampleConfiguration.class)
public class SkipSampleFunctionalTests { ... }
```

Using XML Configuration

```
@SpringBatchTest
@RunWith(SpringRunner.class)
@ContextConfiguration(locations = { "/simple-job-launcher-context.xml",
                                    "/jobs/skipSampleJob.xml" })
public class SkipSampleFunctionalTests { ... }
```

1.2. End-To-End Testing of Batch Jobs

'End To End' testing can be defined as testing the complete run of a batch job from beginning to end. This allows for a test that sets up a test condition, executes the job, and verifies the end result.

In the following example, the batch job reads from the database and writes to a flat file. The test method begins by setting up the database with test data. It clears the CUSTOMER table and then inserts 10 new records. The test then launches the `Job` by using the `launchJob()` method. The `launchJob()` method is provided by the `JobLauncherTestUtils` class. The `JobLauncherTestUtils` class also provides the `launchJob(JobParameters)` method, which allows the test to give particular parameters. The `launchJob()` method returns the `JobExecution` object, which is useful for asserting

particular information about the Job run. In the following case, the test verifies that the Job ended with status "COMPLETED":

XML Based Configuration

```
@SpringBatchTest
@RunWith(SpringRunner.class)
@ContextConfiguration(locations = { "/simple-job-launcher-context.xml",
                                    "/jobs/skipSampleJob.xml" })
public class SkipSampleFunctionalTests {

    @Autowired
    private JobLauncherTestUtils jobLauncherTestUtils;

    private SimpleJdbcTemplate simpleJdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.simpleJdbcTemplate = new SimpleJdbcTemplate(dataSource);
    }

    @Test
    public void testJob() throws Exception {
        simpleJdbcTemplate.update("delete from CUSTOMER");
        for (int i = 1; i <= 10; i++) {
            simpleJdbcTemplate.update("insert into CUSTOMER values (?, 0, ?, 100000)",
                                      i, "customer" + i);
        }

        JobExecution jobExecution = jobLauncherTestUtils.launchJob();


        Assert.assertEquals("COMPLETED", jobExecution.getExitStatus().getExitCode());
    }
}
```

Java Based Configuration

```
@SpringBatchTest
@RunWith(SpringRunner.class)
@ContextConfiguration(classes=SkipSampleConfiguration.class)
public class SkipSampleFunctionalTests {

    @Autowired
    private JobLauncherTestUtils jobLauncherTestUtils;

    private SimpleJdbcTemplate simpleJdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.simpleJdbcTemplate = new SimpleJdbcTemplate(dataSource);
    }

    @Test
    public void testJob() throws Exception {
        simpleJdbcTemplate.update("delete from CUSTOMER");
        for (int i = 1; i <= 10; i++) {
            simpleJdbcTemplate.update("insert into CUSTOMER values (?, 0, ?, 100000)",
                                      i, "customer" + i);
        }

        JobExecution jobExecution = jobLauncherTestUtils.launchJob();


        Assert.assertEquals("COMPLETED", jobExecution.getExitStatus().getExitCode());
    }
}
```

1.3. Testing Individual Steps

For complex batch jobs, test cases in the end-to-end testing approach may become unmanageable. It these cases, it may be more useful to have test cases to test individual steps on their own. The `JobLauncherTestUtils` class contains a method called `launchStep`, which takes a step name and runs just that particular `Step`. This approach allows for more targeted tests letting the test set up data for only that step and to validate its results directly. The following example shows how to use the `launchStep` method to load a `Step` by name:

```
JobExecution jobExecution = jobLauncherTestUtils.launchStep("loadFileStep");
```

1.4. Testing Step-Scoped Components

Often, the components that are configured for your steps at runtime use step scope and late binding to inject context from the step or job execution. These are tricky to test as standalone components,

unless you have a way to set the context as if they were in a step execution. That is the goal of two components in Spring Batch: `StepScopeTestExecutionListener` and `StepScopeTestUtils`.

The listener is declared at the class level, and its job is to create a step execution context for each test method, as shown in the following example:

```
@ContextConfiguration
@TestExecutionListeners( { DependencyInjectionTestExecutionListener.class,
    StepScopeTestExecutionListener.class })
@RunWith(SpringRunner.class)
public class StepScopeTestExecutionListenerIntegrationTests {

    // This component is defined step-scoped, so it cannot be injected unless
    // a step is active...
    @Autowired
    private ItemReader<String> reader;

    public StepExecution getStepExecution() {
        StepExecution execution = MetaDataInstanceFactory.createStepExecution();
        execution.getExecutionContext().putString("input.data", "foo,bar,spam");
        return execution;
    }

    @Test
    public void testReader() {
        // The reader is initialized and bound to the input data
        assertNotNull(reader.read());
    }

}
```

There are two `TestExecutionListeners`. One is the regular Spring Test framework, which handles dependency injection from the configured application context to inject the reader. The other is the Spring Batch `StepScopeTestExecutionListener`. It works by looking for a factory method in the test case for a `StepExecution`, using that as the context for the test method, as if that execution were active in a `Step` at runtime. The factory method is detected by its signature (it must return a `StepExecution`). If a factory method is not provided, then a default `StepExecution` is created.

Starting from v4.1, the `StepScopeTestExecutionListener` and `JobScopeTestExecutionListener` are imported as test execution listeners if the test class is annotated with `@SpringBatchTest`. The preceding test example can be configured as follows:

```
@SpringBatchTest
@RunWith(SpringRunner.class)
@ContextConfiguration
public class StepScopeTestExecutionListenerIntegrationTests {

    // This component is defined step-scoped, so it cannot be injected unless
    // a step is active...
    @Autowired
    private ItemReader<String> reader;

    public StepExecution getStepExecution() {
        StepExecution execution = MetaDataInstanceFactory.createStepExecution();
        execution.getExecutionContext().putString("input.data", "foo,bar,spam");
        return execution;
    }

    @Test
    public void testReader() {
        // The reader is initialized and bound to the input data
        assertNotNull(reader.read());
    }

}
```

The listener approach is convenient if you want the duration of the step scope to be the execution of the test method. For a more flexible but more invasive approach, you can use the `StepScopeTestUtils`. The following example counts the number of items available in the reader shown in the previous example:

```
int count = StepScopeTestUtils.doInStepScope(stepExecution,
    new Callable<Integer>() {
      public Integer call() throws Exception {

        int count = 0;

        while (reader.read() != null) {
           count++;
        }
        return count;
    }
});
```

1.5. Validating Output Files

When a batch job writes to the database, it is easy to query the database to verify that the output is as expected. However, if the batch job writes to a file, it is equally important that the output be verified. Spring Batch provides a class called `AssertFile` to facilitate the verification of output files.

The method called `assertFileEquals` takes two `File` objects (or two `Resource` objects) and asserts, line by line, that the two files have the same content. Therefore, it is possible to create a file with the expected output and to compare it to the actual result, as shown in the following example:

```
private static final String EXPECTED_FILE = "src/main/resources/data/input.txt";
private static final String OUTPUT_FILE = "target/test-outputs/output.txt";

AssertFile.assertFileEquals(new FileSystemResource(EXPECTED_FILE),
                            new FileSystemResource(OUTPUT_FILE));
```

1.6. Mocking Domain Objects

Another common issue encountered while writing unit and integration tests for Spring Batch components is how to mock domain objects. A good example is a `StepExecutionListener`, as illustrated in the following code snippet:

```
public class NoWorkFoundStepExecutionListener extends StepExecutionListenerSupport {

    public ExitStatus afterStep(StepExecution stepExecution) {
        if (stepExecution.getReadCount() == 0) {
            return ExitStatus.FAILED;
        }
        return null;
    }
}
```

The preceding listener example is provided by the framework and checks a `StepExecution` for an empty read count, thus signifying that no work was done. While this example is fairly simple, it serves to illustrate the types of problems that may be encountered when attempting to unit test classes that implement interfaces requiring Spring Batch domain objects. Consider the following unit test for the listener's in the preceding example:

```
private NoWorkFoundStepExecutionListener tested = new
NoWorkFoundStepExecutionListener();

@Test
public void noWork() {
    StepExecution stepExecution = new StepExecution("NoProcessingStep",
                new JobExecution(new JobInstance(1L, new JobParameters(),
                                "NoProcessingJob")));

    stepExecution.setExitStatus(ExitStatus.COMPLETED);
    stepExecution.setReadCount(0);

    ExitStatus exitStatus = tested.afterStep(stepExecution);
    assertEquals(ExitStatus.FAILED.getExitCode(), exitStatus.getExitCode());
}
```

Because the Spring Batch domain model follows good object-oriented principles, the `StepExecution` requires a `JobExecution`, which requires a `JobInstance` and `JobParameters`, to create a valid `StepExecution`. While this is good in a solid domain model, it does make creating stub objects for unit testing verbose. To address this issue, the Spring Batch test module includes a factory for creating domain objects: `MetaDataInstanceFactory`. Given this factory, the unit test can be updated to be more concise, as shown in the following example:

```
private NoWorkFoundStepExecutionListener tested = new
NoWorkFoundStepExecutionListener();

@Test
public void testAfterStep() {
    StepExecution stepExecution = MetaDataInstanceFactory.createStepExecution();

    stepExecution.setExitStatus(ExitStatus.COMPLETED);
    stepExecution.setReadCount(0);

    ExitStatus exitStatus = tested.afterStep(stepExecution);
    assertEquals(ExitStatus.FAILED.getExitCode(), exitStatus.getExitCode());
}
```

The preceding method for creating a simple `StepExecution` is just one convenience method available within the factory. A full method listing can be found in its Javadoc.