Writing Integration Tests for Spring Powered Repositories

Petri Kainulainen

Table of Contents

About the Author

Petri Kainulainen is a Finnish software developer who is specialized in software development with the Spring Framework. He got his first computer as a Christmas gift when he was eight years old. A few months later he "wrote" his first computer program by copying the code from the user's manual of his Commodore 64.

At the time Petri had no idea what he was doing but the outcome of his work amazed him. He wanted to learn more about this thing called programming. In a way, that moment was the starting point of his career.

Petri has over a decade of experience in software development, and he has participated in the development projects of Finland's leading online marketplaces as a software architect. Currently he is working in a privately owned software company, where he concentrates on adding value to customers and learning new things every day.

He also has a popular blog where he writes about Spring Framework, automated testing, and other topics that are related to software development. Petri is currently recording a new video course that helps you to write automated tests for Spring web applications and save time by writing less test code.

By the way, Petri no longer practices copy and paste programming.

Introduction

Spring Framework has an excellent support for creating repositories that insert information into a relational database or query information from it. We can create repositories by using Hibernate, JDBC, and JPA. We can also use Spring Data JPA that helps us to create JPA repositories without writing any boilerplate code.

Naturally Spring Framework has also a world class support for writing integration tests to repositories that use a relational database. This eBook describes how we can leverage this support for writing integration tests to our Spring powered repositories.

Let's start by getting the required testing dependencies with Maven.

This eBook assumes that you know how you can add integration tests into your build. If this is not the case, you should read the following blog posts:

- Integration Testing With Maven describes how you can create a Maven build that uses different directories for unit and integration tests.
- Getting Started With Gradle: Integration Testing With the TestSets Plugin describes how you can add integration tests into your Gradle build by using the Gradle TestSets plugin.

Getting the Required Testing Dependencies With Maven

We can get the required testing dependencies with Maven by following these steps:

First, we need to a testing framework that we will use for writing our integration tests. Because JUnit is the most popular testing framework for Java applications, we will use it in this eBook. We can get the JUnit dependency by adding the following dependency declaration into the `dependencies` section of our *pom.xml* file:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <scope>test</scope>
    <version>4.11</version>
</dependency>
```

Second, we need to get the Spring Test dependency. The Spring Test framework provides tools that we can use for writing both unit and integration tests for our Spring powered applications. We can get it by adding the following dependency declaration into the `dependencies` section of our *pom.xml* file:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <scope>test</scope>
    <version>4.2.1.RELEASE</version>
</dependency>
```

Third, we need to get the Spring Test DBUnit library because it integrates DbUnit with the Spring Test framework. We can get it by adding the following the dependency declaration into the `dependencies` section of our *pom.xml* file:

```
<dependency>
    <groupId>com.github.springtestdbunit</groupId>
    <artifactId>spring-test-dbunit</artifactId>
    <version>1.2.1</version>
    <scope>test</scope>
</dependency>
```

Fourth, We use DbUnit for initializing our database into a known state before each integration test is invoked. We can get the DbUnit library and exclude the JUnit dependency by adding the following dependency declaration into the `dependencies` section of our *pom.xml* file:

```
<dependency>
    <groupId>org.dbunit</groupId>
    <artifactId>dbunit</artifactId>
    <version>2.5.1</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <artifactId>junit</artifactId>
            <groupId>junit</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

This eBook uses the newest versions of these dependencies. If you use an old Spring version, remember that **the version of the Spring Test dependency must be same as your Spring version**.

After we have declared the required dependencies in our *pom.xml* file, we have to configure our integration tests.

Configuring Our Integration Tests

The first thing that we have to do is to create an integration test class. After we have done it, we can configure our integration tests by following these steps:

First, because our integration tests use the Spring Test framework, we need to run them by using the `SpringJUnit4ClassRunner` class. It is a custom JUnit runner that provides support for the features of the Spring Test framework. We can configure the used runner by annotating our test class with the `@RunWith` annotation and setting the type of used runner as its value.

After we have configured the used runner, the source code of our test class looks as follows:

```
@RunWith(SpringJUnit4ClassRunner.class) ①
public class TodoRepositoryTestIT {
}
```

① Ensure that our integration tests are run by the `SpringJUnit4ClassRunner` class.

Second, we have to specify the application context configuration class or XML configuration file that configures the application context used by our integration tests. We can do this by using the `@ContxtConfiguration` annotation.

If we want to use a configuration class, we have to annotate our test class with the `@ContextConfiguration` annotation and specify the used application context configuration class by setting the value of the `classes` attribute.

After we have specified the application context configuration class that configures the application context used by our integration tests, the source code of our test class looks as follows:

```
@RunWith(SpringJUnit4ClassRunner.class) ①
@ContextConfiguration(classes = PersistenceContext.class) ②
public class TodoRepositoryTestIT {
}
```

① Ensure that our integration tests are run by the `SpringJUnit4ClassRunner` class.

② Specify that the application context used by our integration tests is configured in the `PersistenceContext` class.

If we want to use an XML configuration file, we have to annotate our test class with the `@ContextConfiguration` annotation and specify the used application context configuration class by

setting the value of the `classes` attribute.

After we have specified the XML configuration file that configures the application context used by our integration tests, the source code of our test class looks as follows:

```
@RunWith(SpringJUnit4ClassRunner.class) ①
@ContextConfiguration(locations = "classpath:applicationContext-persistence.xml") ②
public class TodoRepositoryTestIT {
}
```

① Ensure that our integration tests are run by the `SpringJUnit4ClassRunner` class.

② Specify that the application context used by our integration tests is configured in the *applicationContext-persistence.xml* file that is found from the classpath.

If we are writing integration tests for repositories which belong to a web application project, and we need to configure the application context of our integration tests by using the root application context configuration class (or XML configuration file), we have to annotate our test class with the `@WebAppConfiguration` annotation. It ensures that the `ApplicationContext` loaded for our integration tests is a `WebApplicationContext`.

Third, we need to configure the test execution listeners that react to the test execution events that are published by the Spring Test framework. Typically we don't have worry about this because the Spring Test framework registers the default listeners for us.

However, because we need to use Spring Test DbUnit, we have to register a new test execution listener. This means that we have to specify the other registered test execution listeners as well.

We have to register the following the test execution listeners:

- The `DependencyInjectionTestExecutionListener` provides dependency injection for the test instance.
- The `TransactionalTestExecutionListener` adds transaction support (with default rollback semantics) into our integration tests.
- The `DbUnitTestExecutionListener` adds support for the features provided by the Spring Test DbUnit library.

Section 14.5.2 TestExecutionListener configuration of the Spring Framework Reference Documentation provides more information about test execution listeners.

After we have configured the required test execution listeners, the source code of our test class looks as follows:

```
@RunWith(SpringJUnit4ClassRunner.class) ①
@ContextConfiguration(classes = PersistenceContext.class) ②
@TestExecutionListeners({ DependencyInjectionTestExecutionListener.class, ③
        TransactionalTestExecutionListener.class,
        DbUnitTestExecutionListener.class })
public class TodoRepositoryTestIT {
}
```

① Ensure that our integration tests are run by the `SpringJUnit4ClassRunner` class.

② Specify that the application context used by our integration tests is configured in the `PersistenceContext` class.

③ Configure the registered test execution listeners.

If we want to provide additional configuration to the `DbUnitTestExecutionListener`, we have to annotate the test class with the `@DbUnitConfiguration` annotation. Typically we need to use this annotation when we want to:

- Configure the name of the `DataSource` bean that provides database connections to DbUnit. We need to do this if the name of our `DataSource` bean is not *dataSource* or *dbUnitDatabaseConnection*.
- Use a custom `DataSetLoader` that enables column sensing and / or adds support for replacement datasets.

After we have configured our integration tests, we can start writing integration tests for our repository.

Writing Integration Tests for Our Repository

We will now write integration tests for the repository method called `findById(Long id)`. This method finds a `Todo` object from the used database by using its id as a search criteria and selects the returned object by following these rules:

- If a `Todo` object is found, the `findById(Long id)` method returns an `Optional` that contains the found `Todo` object.
- If no `Todo` object is found, the `findById(Long id)` method returns an empty `Optional`.

The source code of the `TodoRepository` interface looks as follows:

```
public interface TodoRepository {

    Optional<Todo> findById(Long id);
}
```

The implementation of the `TodoRepository` interface can use Hibernate, JDBC, or JPA. Also, it can be a Spring Data JPA repository interface. In any case, we can write integration tests for it by using the method described in this section.

We can write integration tests for the `findById(Long id)` method by following these steps:

First, we need to create a DbUnit dataset that initializes our database into a known state before each test method is invoked. We will use the flat XML dataset format because it is less verbose than the original DbUnit dataset format. When we create our dataset, we have to follow these rules:

- Each XML element contains the information of a single table row.
- The name of the XML element identifies the name of the database table in which its information is inserted.
- The attributes of the XML element specifies the values that are inserted into the columns of the database table.

We know that the `TodoRepository` queries information from the `todos` database table that has the following columns: `id`, `created_by_user`, `creation_time`, `description`, `modified_by_user`, `modification_time`, `title`, and `version`. Because we want to insert one row into the *todos* table, we have to create a DbUnit dataset file (*todo-entries.xml*) that looks as follows:

```
<dataset>
    <todos id="1"
           created_by_user="createdByUser"
           creation_time="2014-12-24 11:13:28"
           description="description"
           modified_by_user="modifiedByUser"
           modification_time="2014-12-25 11:13:28"
           title="title"
           version="0"/>
</dataset>
```

The DbUnit documentation provides more information about different DbUnit datasets.

After we have created the DbUnit dataset, we have to ensure that it is used to initialize our database into a known state before our test methods are invoked. We can do this by annotating our test class or its test method with the `@DatabaseSetup` annotation and configuring the used datasets by setting the value of its `value` attribute. The difference of these options is described in the following:

- If we annotate our test class with the `@DatabaseSetup` annotation, the datasets are inserted into the database before each test method is invoked.
- If we annotate a test method with the `@DatabaseSetup` annotated, the datasets are inserted into the database before the annotated test method is invoked.

Because our integration tests use the same dataset, we have to annotate our test class with the `@DatabaseSetup` annotation. After we have done this, the source code of our test class looks as follows:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = PersistenceContext.class)
@TestExecutionListeners({ DependencyInjectionTestExecutionListener.class,
        TransactionalTestExecutionListener.class,
        DbUnitTestExecutionListener.class })
@DatabaseSetup("todo-entries.xml") ①
public class TodoRepositoryTest {

}
```

① Configure the used DbUnit dataset.

If the DbUnit dataset is in the same package than our test class, we can use its file name. On the other hand, if it is not in the same package than our test class, we have to specify its full path. For example, if our DbUnit dataset would be in the `net.petrikainulainen.data` package, we would have to use the string: *"/net/petrikainulainen/data/todo-entries.xml"*.

Second, We need to inject the tested repository into our test class. We can do this by adding a `TodoRepository` field into our test class and annotating that field with the `@Autowired` annotation. After we have done this, the source code of our test class looks as follows:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = PersistenceContext.class)
@TestExecutionListeners({ DependencyInjectionTestExecutionListener.class,
        TransactionalTestExecutionListener.class,
        DbUnitTestExecutionListener.class })
@DatabaseSetup("todo-entries.xml") ①
public class TodoRepositoryTestIT {

    @Autowired ②
    private TodoRepository repository;
}
```

① Configure the used DbUnit dataset.

② Inject the tested repository into our test class.

Third, we need to write the integration tests which ensure that our repository method is working as expected. We can do this by following these steps:

1. Write an integration test which ensures that the tested repository method returns an empty `Optional` if no `Todo` object is found.
2. Write an integration test which ensures that the tested repository method returns an `Optional` that contains the found `Todo`object.

After we have written these integration tests, the source code of our test class looks as follows:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = PersistenceContext.class)
@TestExecutionListeners({ DependencyInjectionTestExecutionListener.class,
        TransactionalTestExecutionListener.class,
        DbUnitTestExecutionListener.class })
@DatabaseSetup("todo-entries.xml") ①
public class TodoRepositoryTestIT {

    @Autowired ②
    private TodoRepository repository;

    @Test ③
    public void findById_ObjectIsNotFound_ShouldReturnAnEmptyOptional() {
        Optional<Todo> found = repository.findById(99L);
        assertThat(found.isPresent(), is(false));
    }

    @Test ④
    public void findById_ObjectIsFound_ShouldReturnAnOptionalThatContainsFoundObject()
{
        Optional<Todo> found = repository.findById(1L);

        Todo todoEntry = found.get();
        assertThat(todoEntry.getId(), is(1L));
    }
}
```

① Configure the used DbUnit dataset.

② Inject the tested repository into our test class.

③ Write an integration test which ensures that the `findById(Long id)` method works correctly when no `Todo` object is found.

④ Write an integration test which ensures that the `findById(Long id)` method works correctly when `Todo` object is found.

Let's summarize what we learned from this eBook.

Summary

This eBook has taught us five things:

- We know what libraries / frameworks we need, and we know how we can get them with Maven.
- If we want to write integration tests for our Spring powered repositories, we have to run them by using the `SpringJUnit4ClassRunner` class.
- We can specify the used application context configuration class or XML configuration file by using the `@ContextConfiguration` annotation.
- If we want to write integration tests that use Spring Test DbUnit, we have to register the `DbUnitTestExecutionListener` (and a few other test execution listeners) by using the `@TestExecutionListeners` annotation.
- We can specify the DbUnit dataset, which initializes our database into a known state before each test method is invoked, by using the `@DatabaseSetup` annotation.

Additional Reading:

- Spring From the Trenches: Using Null Values in DbUnit Datasets describes why we should use null values in our DbUnit datasets and explains how we can use them.
- Spring From the Trenches: Resetting Auto Increment Columns Before Each Test Method describes why we should reset the auto increment columns before each test method and explains how we can do it.
- Writing Tests for Data Access Code describes how we can write clean tests for our data access code.
- Test With Spring course helps you to write automated tests for Spring web applications and save time by writing less test code.

About the Author

Petri Kainulainen is a Finnish software developer who is specialized in software development with the Spring Framework. He got his first computer as a Christmas gift when he was eight years old. A few months later he "wrote" his first computer program by copying the code from the user's manual of his Commodore 64.

At the time Petri had no idea what he was doing but the outcome of his work amazed him. He wanted to learn more about this thing called programming. In a way, that moment was the starting point of his career.

Petri has over a decade of experience in software development, and he has participated in the development projects of Finland's leading online marketplaces as a software architect. Currently he is working in a privately owned software company, where he concentrates on adding value to customers and learning new things every day.

He also has a popular blog where he writes about Spring Framework, automated testing, and other topics that are related to software development. Petri is currently recording a new video course that helps you to write automated tests for Spring web applications and save time by writing less test code.

By the way, Petri no longer practices copy and paste programming.