COURSEWARE

Unit testing with Mockito

Contents

Overview

Unit testing involves testing the *smallest possible* section of an application - in Java this usually consists of testing a *single method*.

In more complex applications, a problem arises: how do we test a single method in a class *without its dependencies*?

For example, take this class:

▶ PersonService

`PersonService` *requires* an instance of `PersonRepo` to function - so, how can we test the `PersonService` *without* relying on the functionality of the `PersonRepo`?

The answer is by **mocking** components which we are not focused on testing.

By mocking the `PersonRepo`, we can create a dummy object, with the same methods as the original - but, instead of containing actual functionality, it will instead return a set value *which we can specify*.

Mocking

Creating a mock dependency in Spring done by using the `@MockBean` annotation.

By using this annotation, Spring will create a mocked version of the object when the `ApplicationContext` loads - this mock will then be used for any dependency injection instead of the original.

For example, if we're unit testing the `PersonService`, we'd mock the `PersonRepo`, which is then autowired into the `PersonService`:

▶ PersonServiceUnitTest

The `@SpringBootTest` annotation tells Spring Boot to load the application context for the application when the test is run.

An instance of `PersonService` will be created (as it is a component), but a **mock** instance of `PersonRepo` will be created because of the `@MockBean` annotation.

This **mock** instance will then be injected into the `PersonService` instead of a real one.

This will allow us to mock the `PersonRepo`'s methods using our mocking library.

In this case, we use **Mockito** to mock the responses given by the `PersonRepo`.

Mockito follows the **Given-When-Then** system - given some testing data, when we test a particular method, then check to see if what comes out of that method matches our testing data.

Mockito even has its own When (`when()`) and Then (e.g. `thenReturn()`) methods:

```
Mockito.when(mock.methodName(parameter)).thenReturn(value);
```

Assertions

Assertions were previously covered here.

Spring Boot comes bundled with JUnit5, which uses the `assertEquals(expected, actual)` format.

However, there is a more powerful assertion library available which we recommend: **AssertJ**.

AssertJ supports very readable assertions, so something like `assertEquals(expected, actual)` would instead read like this:

```
assertThat(expected).isEqualTo(actual);
```

This separation of `expected` and `actual` lends itself better to behaviour-driven development (BDD).

Verify

Mockito's `verify()` method is used to check the *structure* of the method being tested. It checks how many times a method was called:

```
Mockito.verify(mock, Mockito.times(number of times)).methodName(parameters);
```

This would get called after completing the Given-When-Then structure.

Tutorial

*(note: this tutorial uses the `PersonService.java` class, as shown in the **Overview** section.)*

Below is the `Person` entity for reference:

▶ Person

We'll start by writing the test class, wiring in any dependencies, and creating the mocks:

```
@SpringBootTest
public class PersonServiceUnitTest {

    @Autowired
    private PersonService service;

    @MockBean
    private PersonRepo repo;

    @Test
    void testCreate() {
        // GIVEN

        // WHEN

        // THEN

        // verify
    }

}
```

Testing the create method

Let's look at the `PersonService`'s `create` method:

```
    public Person create(Person person) {
        return this.repo.save(person);
    }
```

All it does is call the `save` method from `PersonRepo`.

Now, as this is a unit test, we don't want to test the `PersonRepo` functionality - just the service layer - so, instead, we'll **mock** the `PersonRepo` and make it return a dummy value.

Next, we need to set up the mock - this will tell Mockito what it should do when the `save` method is called.

We're expecting here that when we save a `Person` to the database, it is assigned an `id`:

```
    @Test
    void testCreate() {
        Mockito.when(this.repo.save(new Person(null, 26, "JH"))).thenReturn(new
Person(1L, 26, "JH"));
    }
```

Next, use the `Assertions` library to check that the object returned by the service layer's `create` method is the same as the one returned by the repo's `save` method:

```
        Assertions.assertThat(this.service.create(new Person(null, 26,
"JH"))).isEqualTo(new Person(1L, 26, "JH"));
```

Finally, we want to `verify` that the methods we're mocking have been called the correct amount of times:

```
        Mockito.verify(this.repo, Mockito.times(1)).save(new Person(null, 26,
"JH"));
```

Bringing it all together, a full test of the `create()` method might look like this:

▶ Details

Exercises

Fork and clone down this repository of a full, but untested, Spring application.

(*note: this repository makes use of Lombok.*)

Given this empty test class, write a method to test the service layer's `create()` method:

▶ DogServiceUnitTest.java

(*extension: try to write testing methods for the other methods in the service layer.*)

Once you're done, check the `unit-testing` branch for the solution.