**Creating UnitTests for SpringBoot application**

In this workshop we are going to look at how to create Unit tests for a SpringBoot application.

The code is located at <https://github.com/AMIS-Services/sig-springboot-1/>.

All code that is created should preferably be tested.

The main classes are located src/main/java/.

The unit tests are located at src/test/java/.

The unit test class should reflect the package structure of the class being tested.

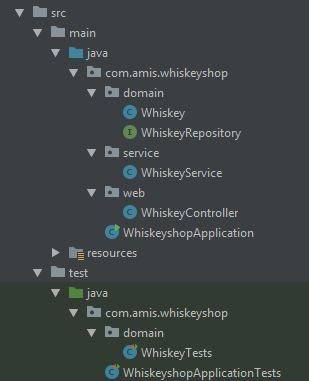
**Create Test class for Whiskey.class**

Let’s start with the Whiskey.class.

The Whiskey.class is located in the package com.amis.whiskeyshop.domain.

We create a test class called WhiskeyTests.class in the same package com.amis.whiskeyshop.domain.

The file structure will look as follow:



After we create the class we add the **@RunWith(SpringRunner.class)** annotation above the class to be able to run the tests.

Let’s create a test for the constructor of the Whiskey.class. We do this by simply creating a method with a descriptive like **testConstructorIdName** () and annotate this method with **@Test**. We want to test when we create a new Whiskey object, the parameters we give actually form the fields of the instance.

You always divide a method into three section for clarity. A prepare, an execute and an verify section. Or alternatively a given, when and then section.

In the prepare section we put all the code related to preparing the parameters needed for execution. In the execute section we put the code which executes the tested function. In the verify section we put the code which verifies whether the result of the executed function matches the expectation.

Dividing the code up like this will create more clarity for users, to see what function each part of the code has. For longer tested methods the code can stack up quickly.

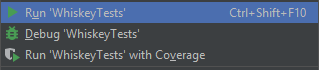
So, the test for Whiskey constructor will look like:

@Test  
public void testConstructorIdName() throws IllegalAccessException {  
 //Prepare  
 String id = "1";  
 String name = "Jack Daniels";  
  
 //Execute  
 Whiskey result = new Whiskey(id, name);  
  
 //Verify  
 *assertEquals*(id, FieldUtils.*readField*(result, "id", true));  
 *assertEquals*(name, FieldUtils.*readField*(result, "name", true));  
}

In this test we need an id and a name which we make in the preparation section. We create the Whiskey object via the constructor in the execute section.

The Whiskey class makes use of private fields. To be able to access these fields we make use of the org.apache.commons **FieldUtils** which can force access for us. We verify whether the prepared items equal the fields of the created object in the verify section. This method is able to throw an IllegalAccessException in case the FieldUtils.readField method accesses a non-existent field.

Once we made a test, we should run the test and see the result.





We can also run the test with coverage, which actually show the percentage of lines being covered the tested class. In general you want a high code coverage. In our small application we should be be able to easily reach a 100% code coverage.

We also have an empty constructor. Try to figure out to create a test method for an empty constructor and see if the test succeeds. Also run the test with coverage and see if the coverage has been increased.

**Create Test class for WhiskeyController.class**

Let’s keep on going and create a test class for WhiskeyController.class. Also annotate the class with **@RunWith(SpringRunner.class).**

Our WhiskeyController makes use of the WhiskeyService class by autowiring the class. And our WhiskeyController methods make use of this autowired WhiskeyService class. We want to make sure that in each method the right method of the WhiskeyService class is called. The scope of testing the WhiskeyController method does not go beyond this. We do not need to actually test whatever happens in the WhiskeyService class, just what happens in the WhiskeyContoller class. Therefore we need to mock the behavior of the WhiskeyService class.

Let’s start by testing the constructor of the class again. And since we always need to create an instance of WhiskeyController before we test all methods of WhiskeyController, let’s set up this test class a bit different.

Our test subject is going to be WhiskeyController:

private WhiskeyController testSubject;

It has a field whiskeyService, which we need to mock:

@Mock  
private WhiskeyService whiskeyService;

And create a setup method for creating a WhiskeyController object, which we can later use is other test methods. Methods annotated with the @Before annotation are executed before each test. This is useful when we want to execute some common code before running a test.

@Before  
public void setUp() {  
 testSubject = new WhiskeyController(whiskeyService);  
}

Our constructor test will look very short like:

@Test  
public void testConstructor() throws IllegalAccessException {  
 *assertEquals*(whiskeyService, FieldUtils.*readField*(testSubject, "whiskeyService", true));  
}

Alright, lets get started with creating a test method for the **WhiskeyContoller.getWhiskey** method. First up we create the test method testGetWhiskey(), annotate with **@Test** and create the three section: Prepare, Execute, Verify.

The getWhiskey takes a String parameter as input. So we setup a String object in the preparation section. getWhiskey also calls the whiskeyService.getWhiskey(id) method, which we have to Mock. We also do this in the preparation section.

As a mocked behavior we want to return a Whiskey-object when the whiskeyService.getWhiskey() method gets called with the id parameter. The Whiskey-object which we want to return we also need to setup in the preparation section. All our preparation is now done and looks like:

//Prepare  
String inputId = "10";  
Whiskey expectedWhiskey = new Whiskey();  
  
*when*(whiskeyService.getWhiskey(inputId)).thenReturn(expectedWhiskey);

In our execute section we want to call the getWhiskey method of our test subject and capture the result in a variable.

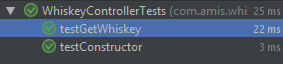
//Execute  
Whiskey result = testSubject.getWhiskey(inputId);

Finally we want to verify whether the result matches the expectation. In this case the returned Whiskey-object from the WhiskeyService and the result of the getWhiskey() method itself. We also want to verify that the WhiskeyService method was only invoked once, and there was no more interaction with the WhiskeyService afterwards.

Our test method will look like:

@Test  
public void testGetWhiskey() {  
 //Prepare  
 String inputId = "10";  
 Whiskey expectedWhiskey = new Whiskey();  
  
 *when*(whiskeyService.getWhiskey(inputId)).thenReturn(expectedWhiskey);  
  
 //Execute  
 Whiskey result = testSubject.getWhiskey(inputId);  
  
 //Verify  
 *assertEquals*(expectedWhiskey, result);  
 *verify*(whiskeyService, *times*(1)).getWhiskey(inputId);  
 *verifyNoMoreInteractions*(whiskeyService);  
}

Let’s run the tests and see if it succeeds.



And that are the basics of unit testing. Longer methods will also require longer preparation and thus test methods will be longer. If a method is able to take multiple paths (if-else), you want to create multiple test methods for the same being tested method. That is why you want to keep your methods small and otherwise refactor them in smaller more general methods, otherwise writing tests will also become longer, less readable and less maintainable afterwards.

**Tip:**

Take a look at the Project Lombok <https://projectlombok.org/features/all>.

When you include the Lombok dependency in your project you can use shorthand annotations to make constructors, getters and setters for you.

You do not need to create unit tests for code you did not write. Thus saving you some time and code.

You probably need an extra plugin for the IDE you are using, so that it understands that you used an annotation for the constructor/getter/setter.

@NoArgsConstructor

@AllArgsConstructor

@Getter and @Setter

Now create some tests for the other methods and classes and see if you can reach a 100% code coverage. You can do it!

**Appendix - files**

WhiskeyTests.class

package com.amis.whiskeyshop.domain;  
  
import org.apache.commons.lang3.reflect.FieldUtils;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.springframework.test.context.junit4.SpringRunner;  
  
import static org.junit.Assert.*assertEquals*;  
  
@RunWith(SpringRunner.class)  
public class WhiskeyTests {  
  
 @Test  
 public void testConstructorIdName() throws IllegalAccessException {  
 //Prepare  
 String id = "1";  
 String name = "Jack Daniels";  
  
 //Execute  
 Whiskey result = new Whiskey(id, name);  
  
 //Verify  
 *assertEquals*(id, FieldUtils.*readField*(result, "id", true));  
 *assertEquals*(name, FieldUtils.*readField*(result, "name", true));  
 }  
   
}

WhiskeyControllerTests.class

package com.amis.whiskeyshop.web;  
  
import com.amis.whiskeyshop.domain.Whiskey;  
import com.amis.whiskeyshop.service.WhiskeyService;  
import org.apache.commons.lang3.reflect.FieldUtils;  
import org.junit.Before;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.mockito.Mock;  
import org.springframework.test.context.junit4.SpringRunner;  
  
import static org.junit.Assert.*assertEquals*;  
import static org.mockito.Mockito.\*;  
  
@RunWith(SpringRunner.class)  
public class WhiskeyControllerTests {  
  
 private WhiskeyController testSubject;  
  
 @Mock  
 private WhiskeyService whiskeyService;  
  
 @Before  
 public void setUp() {  
 testSubject = new WhiskeyController(whiskeyService);  
 }  
  
 @Test  
 public void testConstructor() throws IllegalAccessException {  
 *assertEquals*(whiskeyService, FieldUtils.*readField*(testSubject, "whiskeyService", true));  
 }  
  
 @Test  
 public void testGetWhiskey() {  
 //Prepare  
 String inputId = "10";  
 Whiskey expectedWhiskey = new Whiskey();  
  
 *when*(whiskeyService.getWhiskey(inputId)).thenReturn(expectedWhiskey);  
  
 //Execute  
 Whiskey result = testSubject.getWhiskey(inputId);  
  
 //Verify  
 *assertEquals*(expectedWhiskey, result);  
 *verify*(whiskeyService, *times*(1)).getWhiskey(inputId);  
 *verifyNoMoreInteractions*(whiskeyService);  
 }  
  
  
}