March 3, 2016

JUnit Testing using Mockito and PowerMock

03/04/16 by Thomas Jaspers

Von diesem Artikel ist auch eine deutschsprachige Version verfügbar.

3 Comments

There was this nasty thought coming to my mind when starting to write this blog post: Does the world really need yet another article on *JUnit*, *Mockito* and *PowerMock*? Well, in fact there is already quite some information available out there. On the other hand every new article sure is touching slightly different aspects of the topic and is therefore hopefully still useful for some readers. But enough with the philosophical part of this blog post. Let's start with some nice unit testing and (power-)mocking :-).

What are Mocks and why do we need them

This paragraph is intentionally kept short and you can safely skip it in case you already know the concepts behind mocking.

In unit testing we want to test methods of one class in isolation. But classes are not isolated. They are using services and methods from other classes. Those are often referred to as collaborators. This leads to two major problems:

- External services might simply not work in a unit testing environment as they require database access or are using some other external systems.
- Testing should be focused on the implementation of one class. If external classes are used directly their behaviour is influencing those tests. That is usually not wanted.

This is when mocks are entering the stage and thus *Mockito* and *PowerMock*. Both tools are "hiding away" the collaborators in the class under test replacing them with mock objects. The division of work between the two is that *Mockito* is kind of good for all the standard cases while *PowerMock* is needed for the harder cases. That includes for example mocking static and private methods.

More information can be found from the following article which is kind of THE article on the subject matter http://martinfowler.com/articles/mocksArentStubs.html.

Mocking with Mockito

Let's jump into a first example right away. In the dummy application that is used here to have some examples there is a class **ItemService** that is using an **ItemRepository** to fetch data from the database. Obviously that is a good candidate to do some mocking. In the method under test an item is fetched by its id and then some logic is applied. We only want to test that logic.

```
public class ItemServiceTest {

    @Mock
    private ItemRepository itemRepository;

    @InjectMocks
    private ItemService itemService;

    @Before
    public void setUp() throws Exception {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void getItemNameUpperCase() {

        //
        // Given
        //
        Item mockedItem = new Item("it1", "Item 1", "This is item 1", 2000,
true);
        when(itemRepository.findById("it1")).thenReturn(mockedItem);

        //
        // When
        //
        String result = itemService.getItemNameUpperCase("it1");

        //
        // Then
        //
        verify(itemRepository, times(1)).findById("it1");
        assertThat(result, is("ITEM 1"));
    }
}
```

Probably in most projects some kind of dependency injection framework is used. Therefore the example is

based on Spring and we are using the corresponding *Mockito* annotation **@Mock** to create the mock objects. The class under test is then annotated with the **@InjectMocks** annotation. This is quite straightforward together with initializing the mocks in the setup-method of the JUnit-class.

> The complete example project can be found from GitHub here: https://github.com/ThomasJaspers/java-junit-sample. This does not only contain the test samples, but also the dummy application that contains the functionality to be tested. Thus it is really easy to play around with this :-).

The actual mocking happens using the “when-method-call-then-return”-syntax like **when(itemRepository.findById(“it1”)).thenReturn(mockedItem)**. There are other possibilities like throwing an exception for example. Even if the example is a bit artificial it hopefully helps getting used to the syntax and the overall approach. When testing more complex classes/methods later on the amount of mock-objects might be a bit discouraging on first sight. Anyway, this would be of course an additional motivation to keep methods short.

Beside the mocking of method-calls it is also possible to verify that those methods have been really called. This happens in the above example in the line **verify(itemRepository, times(1)).findById(“it1”)** and is especially useful for testing the flow logic of the class under test.

More Mocking with Mockito

The previous paragraph was showing kind of the basics of using *Mockito*. Of course there are more possibilities and one rather important one is changing objects passed to mocked objects as parameters. This can be done using **doAnswer** as shown in the following example which is an excerpt from this JUnit test

```
@Test
public void testPLZAddressCombinationIncludingHostValue() {

    //
    // Given
    //
    Customer customer = new Customer("204", "John Do", "224B Bakerstreet");

    doAnswer(new Answer<Customer>() {
        @Override
        public Customer answer(InvocationOnMock invocation) throws Throwable {
            Object originalArgument = (invocation.getArguments())[0];
            Customer param = (Customer) originalArgument;
            param.setHostValue("TestHostValue");
            return null;
        }
    }).when(hostService).expand(any(Customer.class));

    when(addressService.getPLZForCustomer(customer)).thenReturn(47891);
    doNothing().when(addressService).updateExternalSystems(customer);

    //
    // When
    //
    String address = customerService.getPLZAddressCombinationIncludingHostValue(customer,
true);

    //
    // Then
    //
    Mockito.verify(addressService, times(1)).updateExternalSystems(any(Customer.class));
    assertThat(address, is("47891_224B Bakerstreet_TestHostValue"));
}
```

With the concepts shown so far it should be possible to cover most of the “standard usecases”. But of course the answer to one important question is still missing: What if – for example – static methods from a collaborator are used? Probably by now it is not that hard to guess the answer to this :-).

PowerMock – Mocking the Impossible

With *PowerMockito* it is possible to mock all the hard cases that *Mockito* does not support. Most of the time this means mocking of static methods. But it is also possible to mock private methods and constructor calls. Anyway most of time the use case is mocking static methods calls. *PowerMockito* is using byte code manipulation and thus it comes with its own JUnit runner. Furthermore the list of classes that needs to be mocked must be given using the **@PrepareForTest** annotation. Let’s take a look at an example again.

```
@RunWith(PowerMockRunner.class)
@PrepareForTest({StaticService.class})
public class ItemServiceTest {

    @Mock
    private ItemRepository itemRepository;

    @InjectMocks
    private ItemService itemService;

    @Before
    public void setUp() throws Exception {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void readItemDescriptionWithoutIOException() throws
IOException {

        //
        // Given
        //
        String fileName = "DummyName";

        mockStatic(StaticService.class);
        when(StaticService.readFile(fileName)).thenReturn("Dummy");

        //
        // When
        //
        String value = itemService.readItemDescription(fileName);

        //
        // Then
        //
        verifyStatic(times(1));
        StaticService.readFile(fileName);
        assertThat(value, equalTo("Dummy"));
    }
}
```

It can be nicely seen here that the tests are written in almost the same manner using *PowerMock* as we are used to from *Mockito*. Main reason for this is that *PowerMock* comes with a specific API for *Mockito* (and also for EasyMock). This can be seen from an excerpt of the Maven file where we are not only importing the *PowerMock* JUnit-module but also the *Mockito*-API:

```
...
<dependency>
    <groupId>org.powermock</groupId>
    <artifactId>powermock-module-
junit4</artifactId>
    <version>1.6.4</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.powermock</groupId>
    <artifactId>powermock-api-
mockito</artifactId>
    <version>1.6.4</version>
    <scope>test</scope>
</dependency>
...
```

Classes containing static methods must be mocked using the**mockStatic()**-method. Also verification if a method has actually been called is slightly different. But for the when-then mocking-part the syntax stays the same.

> Of course you can – and probably will – use Mockito and PowerMock in the same JUnit test at some point of time. When doing so a small agreement in the team might be helpful on which methods are statically imported (e.g. *Mockito-when*) and which are then used fully qualified (e.g.*PowerMockito.when*) to avoid confusion.

One feature of *PowerMock*, that is quite handy at times, is the possibility to delegate to another JUnit runner using the **@PowerMockRunnerDelegate** annotation. This is shown in the following code snippet and the complete example can be found here.

```
@RunWith(PowerMockRunner.class)
@PowerMockRunnerDelegate(Parameterized.c
lass)
@PrepareForTest({StaticService.class})
public class RateServiceTest {
...
}
```

The example is delegating to the Parametrized JUnit runner, while using *PowerMock* at the same time. Another use case – that might be quite likely – is delegating to *SpringJUnit4ClassRunner.class*.

Conclusion

Mockito is offering a very readable and easy to use interface for mocking tests in Java. As*PowerMock* is offering a *Mockito*-like API it can be used almost the same way as*Mockito* itself. This is really quite convenient.

Tests using mocks can be written very readable. But as always in unit testing this depends mostly on the classes under test.

Well, other than that there is not much more to say than: Just get started and happy mocking :-).