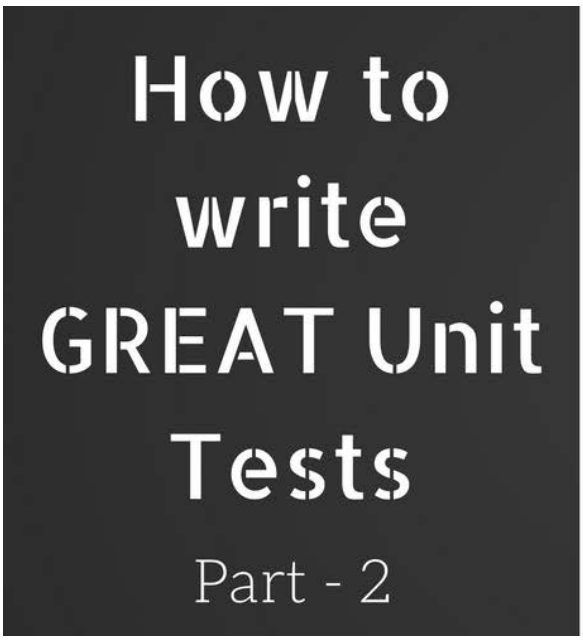
How to mock with Mockito (A comprehensive guide with examples)



Learn to write elegant unit tests and mock dependencies with Mockito. Mockito is a mocking framework for Java which is extremely easy to use, so this post will discuss all the cool features you need to know about mockito with simple and easy examples.

WHY MOCK?

Most of the classes we come across have dependencies. and often times methods delegates some of the work to

other methods in other classes, and we call these classes dependencies. When unit testing such methods, if we only used JUnit, our tests will also depend on those methods as well. We want the unit tests to be independent of all other dependencies.

- eg: we want to test the method `addCustomer` in `CustomerService` class, and within this `addCustomer` method, the save method of the `CustomerDao` class is invoked. We don't want to call the real implementation of the `CustomerDao` `save()` method for a few reasons:
  - We only want to test the logic inside the `addCustomer()` in isolation.
  - We may not yet have implemented it.
  - We don't want the unit test of the `addCustomer()` fail if there is a defect in `save()` method in the `CustomerDao`.
- So we should some how mock the behavior of the dependencies. This is where mocking frameworks comes in to play.
- Mockito framework is what I use for just this and in this post we'll see how to use mockito effectively to mock those dependencies.

If you are new to unit testing with JUnit, please check out the previous post on How to write great unit tests with JUnit

What is mockito?

> *Mockito is a mocking framework that tastes really good. It lets you write beautiful tests with a clean & simple API. Mockito doesn't give you hangover because the tests are very readable and they produce clean verification errors.*
>
> *— "Mockito." Mockito Framework Site. N.p., n.d. Web. 28 Apr. 2017.*

HOW TO INJECT MOCKS

So going back to the example above, how do we mock out the dependency using Mockito? *Well, we could inject a mock to the class under test instead of the real implementation while we run our tests!*

Let's look at an example of a class under test which has a dependency on `CustomerDao`

```
public class CustomerService {

    @Inject
    private CustomerDao customerDao;

    public boolean addCustomer(Customer customer){

    if(customerDao.exists(customer.getPhone())){
         return false;
    }

         return customerDao.save(customer);
    }
    public CustomerDao getCustomerDao() {
         return customerDao;
    }

    public void setCustomerDao(CustomerDao customerDao) {
         this.customerDao = customerDao;
    }

}
```

Following is the test which mocks the dependency using Mockito

```
public class CustomerServiceTest {

    @Mock
    private CustomerDao daoMock;

    @InjectMocks
    private CustomerService service;

    @Before
    public void setUp() throws Exception {

         MockitoAnnotations.initMocks(this);
    }

    @Test
    public void test() {

         //assertion here
    }
```

```
}
```

Let's look at the role of the annotations in the above example.

- `@Mock` will create a mock implementation for the `CustomerDao`
- `@InjectMocks` will inject the mocks marked with `@Mock` to this instance when it is created.
- So **when** or **where** are these instances created? *Well, it is done by this line which reside in the setUp method.*

```
MockitoAnnotations.initMocks(this);
```

- So these instances would be created at the start of every test method of this test class.

HOW TO **MOCK** METHODS WITH MOCKITO

Great! now we have successfully created and injected the mock, and now we should tell the mock how to behave when certain methods are called on it.

The `when` `then` pattern

- We do this in each of the test methods, the following line of code tells the Mockito framework that we want the `save()` method of the mock dao instance to return true when passed in a certain customer instance.

```
when(dao.save(customer)).thenReturn(true);
```

- `when` is a static method of the Mockito class and it returns an `OngoingStubbing<T>` (`T` is the return type of the method that we are mocking, in this case it is `boolean`)
- So if we just extract that out to get hold of the stub, it looks like this:

```
OngoingStubbing<Boolean> stub = when(dao.save(customer));
```

- Following are some of the methods that we can call on this `stub`
  - `thenReturn(returnValue)`
  - `thenThrow(exception)`
  - `thenCallRealMethod()`

`thenAnswer()` - this could be used to set up smarter stubs and also mock behavior of void methods as well (see How to mock void method behavior).

- Simply putting this all in one line again.

```
when(dao.save(customer)).thenReturn(true);
```

- Do we really need to pass in an actual customer object to the save method here? *No, we could use matchers like the following:*

```
when(dao.save(any(Customer.class))).thenReturn(true);
```

- However, when there are multiple parameters to a method, we cannot mix matchers and actual objects, for example we *cannot* do the following:

```
Mockito.when(mapper.map(any(), "test")).thenReturn(new Something());
```

This would compile without a complaint but would fail during runtime with an error saying: `matchers can't be mixed with actual values in the list of arguments to a single method.`

- We either have to use matchers for all parameters or should pass in real values or objects.

Mock behavior of dependencies using `Mockito.when` - Example

```
package com.tdd;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;
import static org.mockito.Mockito.*;

import org.junit.Before;
import org.junit.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

public class CustomerServiceTest {

    @Mock
    private CustomerDao daoMock;

    @InjectMocks
    private CustomerService service;

    @Before
    public void setUp() throws Exception {
```

```
        MockitoAnnotations.initMocks(this);
    }

     @Test
        public void testAddCustomer_returnsNewCustomer() {

            when(daoMock.save(any(Customer.class))).thenReturn(new Customer());

            Customer customer = new Customer();

            assertThat(service.addCustomer(customer), is(notNullValue()));

        }


    //Using Answer to set an id to the customer which is passed in as a parameter to
 the mock method.
    @Test
    public void testAddCustomer_returnsNewCustomerWithId() {

        when(daoMock.save(any(Customer.class))).thenAnswer(new Answer<Customer>() {

            @Override
            public Customer answer(InvocationOnMock invocation) throws Throwable {

                Object[] arguments = invocation.getArguments();

                if (arguments != null && arguments.length > 0 && arguments[0] != nul
l){

                    Customer customer = (Customer) arguments[0];
                    customer.setId(1);

                    return customer;
                }

                return null;
            }
        });

        Customer customer = new Customer();

        assertThat(service.addCustomer(customer), is(notNullValue()));

    }
```

```
    //Throwing an exception from the mocked method
     @Test(expected = RuntimeException.class)
        public void testAddCustomer_throwsException() {

            when(daoMock.save(any(Customer.class))).thenThrow(RuntimeException.class
);

            Customer customer = new Customer();

            service.addCustomer(customer);//

        }
}
```

HOW TO **MOCK VOID** METHOS WITH MOCKITO

---

1. `doAnswer` - If we want our mocked void method to do something (mock the behavior despite being void).
2. `doThrow` - Then there is `Mockito.doThrow()` if you want to throw an exception from the mocked void method.

Following is an example of how to use it (not an ideal usecase but just wanted to illustrate the basic usage).

```
    @Test
    public void testUpdate() {

        doAnswer(new Answer<Void>() {

            @Override
            public Void answer(InvocationOnMock invocation) throws Throwable {
                Object[] arguments = invocation.getArguments();
                if (arguments != null && arguments.length > 1 && arguments[0] != nul
l && arguments[1] != null) {

                    Customer customer = (Customer) arguments[0];
```

```
                    String email = (String) arguments[1];
                    customer.setEmail(email);

                }
                return null;
            }
        }).when(daoMock).updateEmail(any(Customer.class), any(String.class));

        // calling the method under test
        Customer customer = service.changeEmail("[email protected]", "[email protect
ed]");

        //some asserts
        assertThat(customer, is(notNullValue()));
        assertThat(customer.getEmail(), is(equalTo("[email protected]")));

    }

    @Test(expected = RuntimeException.class)
    public void testUpdate_throwsException() {

        doThrow(RuntimeException.class).when(daoMock).updateEmail(any(Customer.class
), any(String.class));

        // calling the method under test
        Customer customer = service.changeEmail("[email protected]", "[email protect
ed]");

    }
}
```

HOW TO **TEST VOID** METHODS WITH MOCKITO - TWO WAYS

Methods with return values can be tested by asserting the returned value, but how to test void methods? The void method that you want to test could either be calling other methods to get things done or processing the input parameters or maybe generating some values or all of it. With Mockito, you can test all of the above scenarios.

1 | Verify with Mockito

- A great thing about mocking is that we can verify that certain methods have been called on those mock objects during test execution in addition to assertions or in place of assertions when the method under test is void.
- There are two overloaded verify methods.
  - one which accepts only the mock object - we can use this if the method is supposed to be invoked only once.
  - the other accepts the mock and a `VerificationMode` - there are quite a few methods in the Mockito class which provides some useful verificationModes
    - `times(int wantedNumberOfInvocations)`
    - `atLeast( int wantedNumberOfInvocations )`
    - `atMost( int wantedNumberOfInvocations )`
    - `calls( int wantedNumberOfInvocations )`
    - `only( int wantedNumberOfInvocations )`
    - `atLeastOnce()`
    - `never()`

`Mockito.verify` - Example

```
package com.tdd;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;
import static org.mockito.Mockito.*;

import org.junit.Before;
import org.junit.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

public class CustomerServiceTest {

    @Mock
    private CustomerDao daoMock;

    @InjectMocks
    private CustomerService service;

    @Before
    public void setUp() throws Exception {
```

```
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void test() {

        when(daoMock.save(any(Customer.class))).thenReturn(true);

        Customer customer=new Customer();
        assertThat(service.addCustomer(customer), is(true));

        //verify that the save method has been invoked
        verify(daoMock).save(any(Customer.class));
        //the above is similar to :  verify(daoMock, times(1)).save(any(Customer.cla
ss));

        //verify that the exists method is invoked one time
        verify(daoMock, times(1)).exists(anyString());

        //verify that the delete method has never been  invoked

        verify(daoMock, never()).delete(any(Customer.class));
    }

}
```

2 | Capture Arguments

Another cool feature is the `ArgumentCaptor` which allows us to capture any arguments passed in to the mocked or spied methods.

`Mockito.ArgumentCaptor` - Example

```
package com.service;

import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.assertThat;
import static org.mockito.Mockito.verify;

import org.junit.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
```

```
import com.dao.CustomerDao;
import com.entity.Customer;

public class CustomerServiceTest {

    @Mock
    private CustomerDao doaMock;

    @InjectMocks
    private CustomerService service;

    @Captor
    private ArgumentCaptor<Customer> customerArgument;

    public CustomerServiceTest() {

        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testRegister() {

        //Requirement: we want to register a new customer. Every new customer should
 be assigned a random token before saving in the database.

        service.register(new Customer());

        //captures the argument which was passed in to save method.
        verify(doaMock).save(customerArgument.capture());

        //make sure a token is assigned by the register method before saving.
        assertThat(customerArgument.getValue().getToken(), is(notNullValue()));
    }
}
```

I SPY - MOCKITO SPY

Why spy?

- Sometimes we do need to call real methods of a dependency but still want to verify or track interactions with that dependency, this is where we would use a spy.
- When a field is annotated with `@Spy`, Mockito will create a wrapper around an actual instance of that object and therefore we can call real implementation and also verify interactions at the same time.
- Some of the behavior of a spy could be mocked if neened.
- in the example below, the dependency behavior is not mocked but still it's interactions are verified.

```
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;
import static org.mockito.Mockito.*;

import org.junit.Before;
import org.junit.Test;
import org.mockito.InjectMocks;
import org.mockito.MockitoAnnotations;
import org.mockito.Spy;

public class CustomerServiceTestV2 {

    @Spy
    private CustomerDaoImpl daoSpy;

    @InjectMocks
    private CustomerService service;

    @Before
    public void setUp() throws Exception {

        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void test() {

        Customer customer = new Customer();

        assertThat(service.addCustomer(customer), is(false));

        verify(daoSpy).save(any(Customer.class));

        verify(daoSpy, times(1)).exists(anyString());
```

```
        verify(daoSpy, never()).delete(any(Customer.class));
    }

}
```

Click Here to get the example source code given in this tutorial.

That's it on this post, please check out the below websites for more cool features, best practices and guidelines on Mockito.

Official Mockito website

Mockito Main reference documentation

Wiki page

Btw here's the part-1 How to write great unit tests with JUnit

Please let me know how you liked this tutorial, and what you would want me to write about in future posts in the comments below.

Please share your thoughts...