Java, JUnit, Testing

MOCKING IN UNIT TESTS WITH MOCKITO

Standard | August 21, 2015 | by jt | 17 Comments

Unit tests should be small tests (atomic), lightweight, and fast. However, an object under test might have dependencies on other objects. It might need to interact with a database, communicate with a mail server, or talk to a web service or a message queue. All these services might not be available during unit testing. Even if they are available, unit testing the *object under test* along with its dependencies can take unacceptable amount of time. What if?

- *The web service is not reachable.*
- *The database is down for maintenance.*
- *The message queue is heavy and slow.*

These all defeat the whole purpose of unit tests being atomic, lightweight, and fast. We want unit tests to execute in a few milliseconds. If the unit tests are slow, your builds become slow, which affects the productivity of your development team. The solution is to use mocking, a way to provide test doubles for your classes being tested.

If you've been following the SOLID Principles of Object Oriented Programming, and using the Spring Framework for Dependency Injection, mocking becomes a natural solution for unit testing. You don't really need a database connection. You just need an object that returns the expected result. If you've written tightly coupled code, you will have a difficult time using mocks. I've seen plenty of legacy code which could not be unit tested because it was so tightly coupled to other dependent objects. This untestable code did not follow the SOLID Principles of Object Oriented Programming, nor did it utilize Dependency Injection.

Mock Objects: Introduction

In unit test, a test double is a replacement of a dependent component (collaborator) of the object under test. A test double provides the same interface as of the collaborator. It may not be the complete interface, but for the functionality required for the test. Also, the test double does not have to behave exactly as the collaborator. The purpose is to mimic the collaborator to make the object under test think that it is actually using the collaborator.

Based on the role played during testing, there can be different types of test doubles, and mock object is one of them. Some other types are dummy object, fake object, and stub.

What makes a mock object different from the others is that it uses behavior verification. It means that the mock object verifies that *it (the mock object) is being used correctly by the object under test*. If the verification succeeds, it can be considered that the object under test will correctly use the real collaborator.

The Test Scenario

For the test scenario, consider a product ordering service. A client interacts with a DAO to fulfill a product ordering process.

We will start with the `Product` domain object and the DAO interface, `ProductDao`.

Product.java

```java
package guru.springframework.unittest.mockito;

public class Product {

}
```

ProductDao.java

```java
package guru.springframework.unittest.mockito;

public interface ProductDao {
 int getAvailableProducts(Product product);
 int orderProduct(Product product, int orderedQuantity);
}
```

For the purpose of the example, I kept the `Product` class empty. But in real applications, it will typically be an entity with states having corresponding getter and setter methods, along with any implemented behaviors.

In the `ProductDao` interface, we declared two methods:

- The `getAvailableProducts()` method returns the number of available quantity of a `Product` passed to it.
- The `orderProduct()` places an order for a product.

The `ProductService` class that we will write next is what we are interested on – *the object under*

test.

ProductService.java

```java
package guru.springframework.unittest.mockito;

public class ProductService {
 private ProductDao productDao;
 public void setProductDao(ProductDao productDao) {
 this.productDao = productDao;
 }
 public boolean buy(Product product, int orderedQuantity) throws
InsufficientProductsException {
 boolean transactionStatus=false;
 int availableQuantity = productDao.getAvailableProducts(product);
 if (orderedQuantity > availableQuantity) {
 throw new InsufficientProductsException();
 }
 productDao.orderProduct(product, orderedQuantity);
 transactionStatus=true;
 return transactionStatus;
 }

}
```

The `ProductService` class above is composed of `ProductDao`, which is initialized through a setter method. In the `buy()` method, we called `getAvailableProducts()` of `ProductDao` to check if sufficient quantity of the specified product is available. If not, an exception of type `InsufficientProductsException` is thrown. If sufficient quantity is available, we called the `orderProduct()` method of `ProductDao`.

What we now need is to unit test **ProductService**. But as you can see, **ProductService** is composed of **ProductDao**, whose implementations we don't have yet. It can be a Spring Data JPA implementation retrieving data from a remote database, or an implementation that communicates with a Web service hosting a cloud-based repository – We don't know. Even if we have an implementation, we will use it later during integration testing, one of the software testing type I wrote earlier on. But now, we are **not interested on any external implementations** in this unit test.

In unit tests, we should not to be bothered what the implementation is doing. What we want is to test that our **ProductService** is behaving as expected and that it is able to correctly use its collaborators. For that, we will mock **ProductDao** and **Product** using Mockito.

The **ProductService** class also throws a custom exception, **InsufficientProductsException**. The code

of the exception class is this.

InsufficientProductsException.java

```java
package guru.springframework.unittest.mockito;

public class InsufficientProductsException extends Exception {
 private static final long serialVersionUID = 1L;
 private String message = null;
 public InsufficientProductsException() { super(); }
 public InsufficientProductsException(String message) {
 super(message);
 this.message = message;
 }
 public InsufficientProductsException(Throwable cause)
 {
 super(cause);
 }
 @Override
 public String toString() {
 return message;
 }
}
```

Using Mockito

Mockito is a mocking framework for unit tests written in Java. It is an open source framework available at github. You can use Mockito with JUnit to create and use mock objects during unit testing. To start using Mockito, download the JAR file and place it in your project class. If you are using Maven, you need to add its dependency in the pom.xml file, as shown below.

pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>guru.springframework.unittest.quickstart</groupId>
  <artifactId>unittest</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>unittest</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
```

```
14      <groupId>junit</groupId>
15      <artifactId>junit</artifactId>
16      <version>4.12</version>
17      <scope>test</scope>
18    </dependency>
19      <dependency>
20          <groupId>org.hamcrest</groupId>
21          <artifactId>hamcrest-library</artifactId>
22          <version>1.3</version>
23          <scope>test</scope>
24      </dependency>
25      <dependency>
26          <groupId>org.mockito</groupId>
27          <artifactId>mockito-all</artifactId>
28          <version>1.9.5</version>
29      </dependency>
  </dependencies>
</project>
```

Once you have set up the required dependencies, you can start using Mockito. But, before we start any unit tests with mocks, let's have a quick overview of the key mocking concepts.

Mock Object Creation

For our example, it's apparent that we need to mock **ProductDao** and **Product**. The simplest way is through calls to the `mock()` method of the `Mockito` class. The nice thing about Mockito is that it allows creating mock objects of both interfaces and classes without forcing any explicit declarations.

MockCreationTest.java

```java
package guru.springframework.unittest.mockito;

import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;
public class MockCreationTest {
    private ProductDao productDao;
    private Product product;
    @Before
    public void setupMock() {
        product = mock(Product.class);
        productDao = mock(ProductDao.class);
    }
    @Test
    public void testMockCreation(){
        assertNotNull(product);
        assertNotNull(productDao);
    }
}
```

An alternative way is to use the `@Mock` annotation. When you use it, you will need to initialize the

mocks with a call to `MockitoAnnotations.initMocks(this)` or specify **MockitoJUnitRunner** as the JUnit test runner as `@RunWith(MockitoJUnitRunner.class)`.

MockCreationAnnotationTest.java

```java
package guru.springframework.unittest.mockito;

import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

public class MockCreationAnnotationTest {
    @Mock
    private ProductDao productDao;
    @Mock
    private Product product;
    @Before
    public void setupMock() {
       MockitoAnnotations.initMocks(this);
    }
    @Test
    public void testMockCreation(){
        assertNotNull(product);
        assertNotNull(productDao);
    }
}
```

Stubbing

Stubbing means simulating the behavior of a mock object's method. We can stub a method on a mock object by setting up an expectation on the method invocation. For example, we can stub the `getAvailableProducts()` method of the `ProductDao` mock to return a specific value when the method is called.

```
. . .
@Test
public void testBuy() throws InsufficientProductsException {
    when(productDao.getAvailableProducts(product)).thenReturn(30);
    assertEquals(30,productDao.getAvailableProducts(product));
}
. . .
```

In **Line 4** of the code above, we are stubbing `getAvailableProducts(product)` of `ProductDao` to return `30`. The `when()` method represents the trigger to start the stubbing and `thenReturn()`

`when(productDao.getAvailableProducts(product))` represents the action of the trigger – which in the example code is to return the value `30`. In **Line 5** with an assertion, we confirmed that the stubbing performed as expected.

Verifying

Our objective is to test **ProductService**, and unitl now we only mocked **Product** and **ProductDao** and stubbed **getAvailableProducts()** of **ProductDao**.

We now want to verify the behavior of the `buy()` method of `ProductService`. First, we want to verify whether it's calling the `orderProduct()` of `ProductDao` with the required set of parameters.

```
1 . . .
2 @Test
3 public void testBuy() throws InsufficientProductsException {
4     when(productDao.getAvailableProducts(product)).thenReturn(30);
5     assertEquals(30,productDao.getAvailableProducts(product));
6     productService.buy(product, 5);
7     verify(productDao).orderProduct(product, 5);
8 }
9 . . .
```

In **Line 6** we called the `buy()` method of `ProductService` that is under test. In **Line 7**, we verified that the `orderProduct()` method of the `ProductDao` mock get's invoked with the expected set of parameters (that we passed to `buy()`).

Our test passed. But, not complete yet. We also want to verify:

- **Number of invocations done on a method**: The **buy()** method invokes **getAvailableProduct()** at least once.
- **Sequence of Invocation**: The **buy()** method first invokes **getAvailableProduct()**, and then **orderProduct()**.
- **Exception verification**: The **buy()** method fails with **InsufficientProductsException** if order quantity passed to it is more than the available quantity returned by **getAvailableProduct()**.
- **Behavior during exception**: The **buy()** method doesn't invokes **orderProduct()** when **InsufficientProductsException** is thrown.

Here is the complete test code.

ProductServiceTest.java

```java
package guru.springframework.unittest.mockito;


import org.junit.Before;
import org.junit.Test;
import org.mockito.InOrder;
import static org.mockito.Mockito.*;
import org.mockito.Mock;

public class ProductServiceTest {
    private ProductService productService;
    private ProductDao productDao;
    private Product product;
    private int purchaseQuantity = 15;

    @Before
    public void setupMock() {
        productService = new ProductService();
        product = mock(Product.class);
        productDao = mock(ProductDao.class);
        productService.setProductDao(productDao);
    }

    @Test
    public void testBuy() throws InsufficientProductsException {
        int availableQuantity = 30;
        System.out.println("Stubbing getAvailableProducts(product) to return " +
availableQuantity);
        when(productDao.getAvailableProducts(product)).thenReturn(availableQuantity);
        System.out.println("Calling ProductService.buy(product," + purchaseQuantity + ")");
        productService.buy(product, purchaseQuantity);
        System.out.println("Verifying ProductDao(product, " + purchaseQuantity + ") is
called");
        verify(productDao).orderProduct(product, purchaseQuantity);
        System.out.println("Verifying getAvailableProducts(product) is called at least
once");
        verify(productDao, atLeastOnce()).getAvailableProducts(product);
        System.out.println("Verifying order of method calls on ProductDao: First call
getAvailableProducts() followed by orderProduct()");
        InOrder order = inOrder(productDao);
        order.verify(productDao).getAvailableProducts(product);
        order.verify(productDao).orderProduct(product, purchaseQuantity);



    }

    @Test(expected = InsufficientProductsException.class)
    public void purchaseWithInsufficientAvailableQuantity() throws
InsufficientProductsException {
        int availableQuantity = 3;
        System.out.println("Stubbing getAvailableProducts(product) to return " +
availableQuantity);
        when(productDao.getAvailableProducts(product)).thenReturn(availableQuantity);
        try {
            System.out.println("productService.buy(product" + purchaseQuantity + ") should
```

```
57  throw InsufficientProductsException");
58              productService.buy(product, purchaseQuantity);
59          } catch (InsufficientProductsException e) {
60              System.out.println("InsufficientProductsException has been thrown");
                verify(productDao, times(0)).orderProduct(product, purchaseQuantity);
                System.out.println("Verified orderProduct(product, " + purchaseQuantity + ") is
not called");
                throw e;
            }
        }

    }
```

I have already explained the initial code of the test class above. So we will start with **Line 36 – Line 38** where we used the `inOrder()` method to verify the order of method invocation that the `buy()` method makes on `ProductDao`.

Then we wrote a `purchaseWithInsufficientAvailableQuantity()` test method to check whether an **InsufficientProductsException** gets thrown, as expected, when an order with quantity more than the available quantity is made. We also verified in **Line 54** that if `InsufficientProductsException` gets thrown, the `orderProduct()` method is not invoked.

The output of the test is this.

```
1   -------------------------------------------------------
2    T E S T S
3   -------------------------------------------------------
4
5   Running guru.springframework.unittest.mockito.ProductServiceTest
6   Stubbing getAvailableProducts(product) to return 30
7   Calling ProductService.buy(product,15)
8   Verifying ProductDao(product, 15) is called
9   Verifying getAvailableProducts(product) is called at least once
10  Verifying order of method calls on ProductDao: First call getAvailableProducts() followed by
11  orderProduct()
12  Stubbing getAvailableProducts(product) to return 3
13  productService.buy(product15) should throw InsufficientProductsException
14  InsufficientProductsException has been thrown
15  Verified orderProduct(product, 15) is not called
    Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.077 sec
```

Mockito Mocks vs Mockito Spies

In testing Spring Boot applications sometimes you need to access the real component. This is where Mockito Spies come into the picture. If you'd like to learn more about using Mockito Spies, check out this post.

Summary

Mocking in unit testing is extensively used in Enterprise Application Development with Spring. By using Mockito, you can replace the **@Autowired** components in the class you want to test with mock objects. You will be unit testing controllers by injecting mock services. You will also be setting up services to use mock DAOs to unit test the service layer. To unit test the DAO layer, you will mock the database APIs. The list is endless – It depends on the type of application you are working on and the object under test. If you're following the Dependency Inversion Principle and using Dependency Injection, mocking becomes easy.

The Mockito library is a very large and mature mocking library. It is very popular to use for mocking objects in unit tests. Mockito is popular because it is easy to use, and very versatile. I wrote this post as just an introduction to mocking and Mockito. Checkout the official Mockito documentation to learn about all the capabilities of Mockito.

mockito unit testing

You May Also Like

Java

Using JAXB for XML with Java

Standard | January 24, 2018 | by jt | 0 Comments

Java

Java 8 forEach

Standard | January 16, 2018 | by jt | 1 Comment

Java

Jackson Mix-in Annotation

Standard | January 11, 2018 | by jt | 0 Comments

Java, Spring, Spring Boot, Spring MVC

Spring Component Scan

Standard | November 30, 2017 | by jt | 0 Comments

Java

Random Number Generation in Java

Standard | November 29, 2017 | by jt | 0 Comments

Java

Jackson Annotations for JSON

Standard | November 29, 2017 | by jt | 3 Comments

17 comments on “ Mocking in Unit Tests with Mockito ”

Abdullah September 20, 2015 at 2:24 am # Reply

Brilliant article. Very helpful to me, Thanks!

sowmya October 7, 2015 at 8:52 pm # Reply

Awesome Article.. very well explained. thank you!

Mansoor March 14, 2016 at 2:29 pm # Reply

Very well explained article. I learnt a lot from it. Thanks!

Ram April 20, 2016 at 4:26 pm # Reply

nice article. well explained. it helps me to understand the basic mockito with good example. Thank you very much for providing nice insight.

routhu kishore June 22, 2016 at 5:55 am # Reply

It is Brilliant article … ! very helpful to me to begin

sathya December 14, 2016 at 3:42 pm # Reply

Well explained. thank you

sfefefef February 9, 2017 at 1:40 am # Reply

Thanks!

nitesh February 19, 2017 at 4:00 am # Reply

where is the code copy ??

nitesh February 19, 2017 at 4:01 am # Reply

how to setup this copy into eclipse ? if would have provide project copy it could be useful rather than copy paste each file from here

Ven March 8, 2017 at 2:47 pm # Reply

This is the best article that I have seen so far on writing JUnit tests using mock objects. I plan to read your other posts as well. Keep up the excellent work, and Thank You!

jt March 8, 2017 at 2:52 pm # Reply

Thanks!

Tahir March 11, 2017 at 3:13 am # Reply

Best article that i have read so far on mockito. Thanks a lot

guest March 18, 2017 at 5:40 am # Reply

nice post, but unprintable

Frank May 16, 2017 at 2:02 pm # Reply

GREAT!! Thank you so much! Best article that i have read so far on mockito also!

lalit kachhwah July 12, 2017 at 3:02 am # Reply

Nice artical

Nk February 13, 2018 at 2:25 pm # Reply

Superb article very helpful

BN March 10, 2018 at 11:30 am # Reply

Always the springframework.org articles are really good, this article also not exceptional. Always they tell the core of the concept or topic, keep up good work and it is very help full.

Leave a Reply

Email (required) (Address never made public)

Name (required)

Website

Notify me of new comments via email.

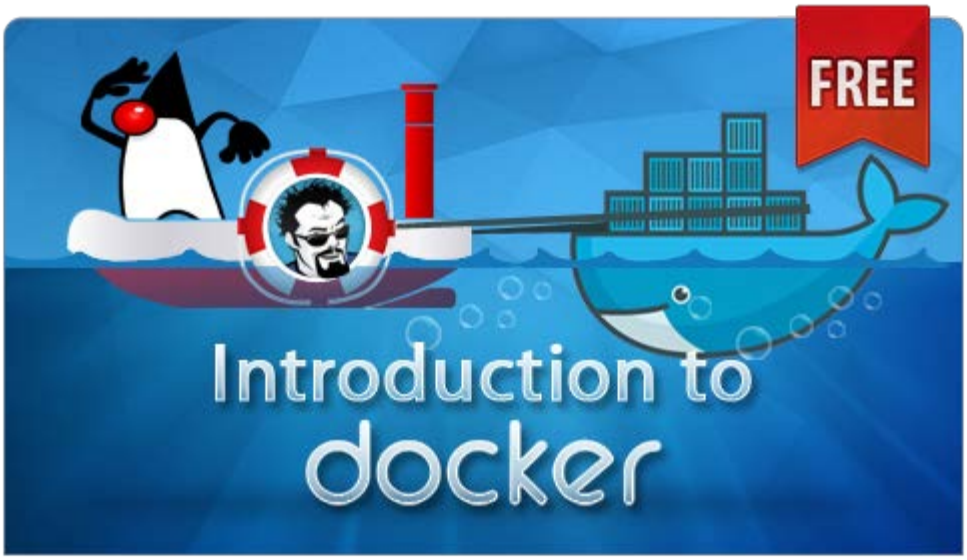
Notify me of new posts via email.

Post Comment

Clear

FREE SPRING FRAMEWORK TUTORIAL

FREE INTRO TO DOCKER COURSE



SPRING CORE ULTIMATE COURSE

SPRING FRAMEWORK GURU

> Spring Framework Guru

RECENT POSTS

Using JAXB for XML with Java

January 24, 2018 | 0 Comments

Java 8 forEach

January 16, 2018 | 1 Comment

Jackson Mix-in Annotation

January 11, 2018 | 0 Comments

Spring Component Scan

November 30, 2017 | 0 Comments

Random Number Generation in Java

November 29, 2017 | 0 Comments

RECENT POSTS

Using JAXB for XML with Java

January 24, 2018 | 0 Comments

Java 8 forEach

January 16, 2018 | 1 Comment

Jackson Mix-in Annotation

January 11, 2018 | 0 Comments

Spring Component Scan

November 30, 2017 | 0 Comments

Random Number Generation in Java

November 29, 2017 | 0 Comments

POPULAR POSTS

Jt on Spring Boot Web Application – Part 4 – Spring MVC

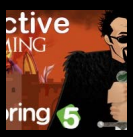
♡ 46

Jt on Using Logback with Spring Boot

♡ 39

Jt on Spring Boot Web Application – Part 2 – Using ThymeLeaf

♡ 36



Jt on What are Reactive Streams in Java?

♡ 36

Jt on Using the H2 Database Console in Spring Boot with Spring Security

♡ 36

RECENT COMMENTS

Simanta Sarma on Spring Boot RESTful API Documentation with Swagger 2

Felix on

Using the H2 Database Console in Spring Boot with Spring Security

someone on Spring Boot RESTful API Documentation with Swagger 2

Jochen on Single Responsibility Principle

Jochen on Gang of Four Design Patterns