Mockito and PowerMockito

Table of Contents

1.1 Mockito

1.1.1 Introduction to Mockito

1.1.2 Mocking JPA Repositry with Mockito

1.1.3 Mocking RestTemplate

1.1.4 Kafka Mocking

1.1.5 Limitations of Mockito

1.2 PowerMockito

1.2.1 Introduction

1.2.2 Mocking final methods

1.2.3 Mocking static methods

1.3 Differences between Mockito and PowerMockito

MOCKITO AND POWERMOCK

1.1 MOCKITO

1.1.1 Introduction

Mockito is an open source testing framework for Java .The framework allows the creation of test double objects (mock objects) in automated unit tests for the purpose of test-driven development (TDD) or behavior-driven development (BDD).

Mockito is a mocking framework, **JAVA**-based library that is used for effective unit testing of **JAVA** applications. Mockito is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing.

There are some annotations used here:

- `@Mock` will create a mock implementation for the `Dao` class
- `@InjectMocks` will inject the mocks marked with `@Mock` to this instance when it is created.
- These instances would be created at the start of every test method of this test class

1.1.2 Mocking JPA Repositry with Mockito:

```
1.  @Test
2.      public void updateBooksTest() {
3.          Books booksInventoryExpected = new Books();
4.          booksInventoryExpected.setAuthor("author");
5.          booksInventoryExpected.setAuthor2("author2");
6.          booksInventoryExpected.setBookId(1);
7.          booksInventoryExpected.setBookName("bookName");
8.          booksInventoryExpected.setGenre("genre");
9.          booksInventoryExpected.setPublisher("publisher");
10.         booksInventoryExpected.setYearOfPublisher("Year");
11.
12.     when(libraryRepository.getOne(Mockito.anyInt())).thenReturn(booksInventoryExpected);
13. Books BooksInventoryActual = libraryService.updateBooks(booksInventoryExpected,booksInventoryExpe
    cted.getBookId());
14.
15.         assertEquals(booksInventoryExpected, BooksInventoryActual);
16.
17.     }
```

Since Mockito any(Class) and anyInt family matchers perform a type check, thus they won't match null arguments. Instead use the isNull matcher.

1.1.3 Mocking RestTemplate:

```
1.  @Service
2.  public class LibService {
3.
4.      @Autowired(required=false)
5.      private RestTemplate restTemplate;
6.
7.      public Books getBooksByid(int bookid) {
8.      ResponseEntity<Books> resp =
9.            restTemplate.getForEntity("http://localhost:9003/getbooks/" + bookid, Books.class
    );
10.
11.     return resp.getStatusCode() == HttpStatus.OK ? resp.getBody() : null;
12.
13.     }
14. }
```

In service class we will mock the instance of RestTemplate and we have the method getForEntity() which accepts two parameters,one is any string and other is any class.

```
1.  @Test
2.    public void getBooksByid() {
3.        Books book = new Books();
4.        book.setBookId(5);
5.        when(restTemplate.getForEntity(Mockito.anyString(),Mockito.<Class<Books>>any(
    )
6.        .thenReturn(new ResponseEntity<Books>(book, HttpStatus.OK));
7.        Books books = libService.getBooksByid(5);
8.        Assert.assertEquals(book, books);
```

Here in test class we are setting some bookId and it will be compared with mocked instance of books .Then if both are equal it will return the status as ok and returns the object or else null.

1.1.4 Kafka Mocking:

First step is to add the dependency in the POM file :

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
  </dependency>
 <dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka-test</artifactId>
    <scope>test</scope>
  </dependency>
```

- Here we will have two classes Producer and Consumer .Producer is the one which is used to send the messages and the Consumer receives the messages.
- In the SpringKafkaSenderTest test case, we will be testing the Sender by sending a message to a 'sender.t' topic. We will verify whether the sending works by setting up a *test-listener* on the topic.
- The second SpringKafkaReceiverTest test class focuses on the Receiver which listens to a 'receiver.t' topic as defined in the applications.yml properties file. To check the correct working, we use a *test-template* to send a message to this topic.
- We need to ensure that the Receiver is initialized before sending the test message. For this we use the waitForAssignment()method of ContainerTestUtils. The link to the message listener container is acquired by auto-wiring the KafkaListenerEndpointRegistry which manages the lifecycle of the listener containers that are not created manually.
- In the test, we send a greeting and check that the message was received by asserting that the latch of the Receiver

```
@Test
 public void testReceive() throws Exception {
    // send the message
String greeting = "Hello Spring Kafka Receiver!";
template.sendDefault(greeting);
    LOGGER.debug("test-sender sent message='{}'", greeting);
    receiver.getLatch().await(10000, TimeUnit.MILLISECONDS);
    // check that the message was received
    assertThat(receiver.getLatch().getCount()).isEqualTo(0);
 }
```

Reference for Kafka mocking:
https://codenotfound.com/spring-kafka-embedded-unit-test-example.html

1.1.5 Limitations of Mockito

Mockito creates a dynamic (proxy-based) subclass of the class you pass in. This means that, like a subclass you would write yourself, you won't have access or control over private fields and methods, static methods, and local variables. Here PowerMock comes into picture.
PowerMock is a framework that extends other mock libraries giving them more powerful capabilities. PowerMock uses a custom classloader and bytecode manipulation to enable mocking of static methods, constructors, final classes and methods, private methods, removal of static initializers and more.

1.2 POWERMOCKITO

1.2.1 Introduction

PowerMockito is a PowerMock's extension API to support Mockito. It provides capabilities to work with the Java Reflection API in a simple way to overcome the problems of Mockito, such as the lack of ability to mock final, static or private methods.

The first step to integrate PowerMock support for Mockito is to include the following two dependencies in the Maven POM file:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.0.5-beta</version>
</dependency>
<dependency>
    <groupId>org.powermock</groupId>
```

```
        <artifactId>powermock-api-mockito</artifactId>
        <version>1.6.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.powermock</groupId>
        <artifactId>powermock-module-junit4</artifactId>
        <version>1.6.2</version>
        <scope>test</scope>
    </dependency>
```

❖ Hence the Mockito and PowerMockito dependency must have the compatible versions

Supported versions

Mockito	PowerMock
1.10.8+	1.6.2+
1.9.5-rc1 - 1.9.5	1.5.0 - 1.5.6
1.9.0-rc1 & 1.9.0	1.4.10 - 1.4.12
1.8.5	1.3.9 - 1.4.9
1.8.4	1.3.7 & 1.3.8
1.8.3	1.3.6
1.8.1 & 1.8.2	1.3.5
1.8	1.3
1.7	1.2.5

Next, we need to prepare our test cases for working with PowerMockito by applying the following two annotations:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(fullyQualifiedNames = "com.baeldung.powermockito.introduction.*")
```

The *fullyQualifiedNames* element in the *@PrepareForTest* annotation represents an array of fully qualified names of types we want to mock.

1.2.2 Mocking Final Methods

We will demonstrate the ways to get a mock instance instead of a real one when instantiating a class with the *new* operator, and then use that object to mock a final method. The collaborating class, whose constructors and final methods will be mocked, is defined as follows:

```
1.  @Test
2.     public void testClassWithFinalMethods_printMessage_finalMethod() throws Excepti
    on{
3.         String message = "Hello PowerMockito";
4.
5.      ClassWithFinalMethods mockObject = PowerMockito.mock(ClassWithFinalMethods.
    class)
6.         PowerMockito.whenNew(ClassWithFinalMethods.class).withNoArguments().thenR
    eturn(mockObject);
7.         ClassWithFinalMethods object = new ClassWithFinalMethods();
8.         PowerMockito.verifyNew(ClassWithFinalMethods.class).withNoArguments();
9.
10.        PowerMockito.when(object.printMessage(message)).thenReturn(message);
11.        String helloPowerMockito = object.printMessage(message);
12.        Mockito.verify(object).printMessage(message);
13.        Assert.assertEquals(message, helloPowerMockito);
14.    }
```

1. We defined a generic String message which we will be using as a parameter and expectation.
2. We mock an instance of the system under test, *ClassWithFinalMethods*.
3. *whenNew()* method makes sure that whenever **an instance of this class is made using the new keyword by invoking a no argument constructor**, this mock instance is returned instead of the real object.
4. We invoke the no argument constructor to make an instance of the system under test.
5. We verify that the no argument constructor was actually involved during the last step.
6. We set an expected String when the final method is called, using the String we defined in Step 1.
7. The final method *printMessage(…)* is invoked.
8. We verify that the final method was actually called.
9. Finally, we assert our expectations to the actual String returned to us.

1.2.3 Mocking Static Methods

Suppose that we want to mock static methods of a class .

```
1.  @Test
2.      public void testClassWithStaticMethod_printMessage_staticMethod() {
3.          String message = "Hello PowerMockito";
4.          String expectation = "Expectation";
5.          PowerMockito.mockStatic(ClassWithStaticMethod.class);
6.          PowerMockito.when(ClassWithStaticMethod.printMessage(message)).thenReturn(e
    xpectation);
7.          String actual = ClassWithStaticMethod.printMessage(message);
8.          Assert.assertEquals(expectation, actual);
9.      }
```

1. We defined a generic String message which we will be using as a parameter.
2. Another generic String message, to be used as an expectation.
3. Prepare *ClassWithStaticMethod* for static method test.
4. Preparing expectations when the static method will be invoked.
5. Invoking the static method.
6. Verifying the expected and actual result.

References for PowerMockito:

https://java2blog.com/powermock-tutorial/

https://examples.javacodegeeks.com/core-java/mockito/powermock-mockito-integration-example/

1.3 Differences between Mockito and PowerMockito

Mockito is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing.

PowerMockito is the extended part of Mockito framework which is used to mock the static and final methods, variables and constructors.