Beginners Guide to Mocking in Scala

We all know that unit test cases are one of the most important parts of an application. No? Then, I must tell you that unit testing is one of the earliest tests to be performed on the unit of code, and the earlier the defects are detected, the easier it is to fix. It reduces the difficulties of discovering errors contained in more complex pieces of the application.

So where does mocking come into the picture? Why do we need it? And how do we understand what we should mock while writing unit test cases? Answers to these questions are right below in this blog.

You may also like: A Guide to Mocking With Mockito

A Look at Mocking

There are programs that do not have any dependencies on external classes. Consider a program that has a method that prints “Hello, World.” Now, this program does not depend on any external classes. However, in real-world applications, classes have dependencies. Those dependencies may be some services and those services may have dependencies on some database objects and the list goes on. The main point of writing unit test cases is to test whether our piece of code is working without interaction with the external environment, i.e. if any method of another class has been called, then that method should be mocked.

Consider this example:

Scala

```
class UserAction(loginService: LoginService) {

def displayActionsPerformed(val userId: String) {

    val result = loginService.getListOfActionsPerformed(userId)

   if(result.size > 0){

         return SUCCESS

   }
```

```
    else {

          return FAILURE

    }

  }

}
```

It’s clear from the code that the class UserAction has a dependency on LoginService. Here, if we start writing unit test cases for this class, we will start by testing the displayActionPerformed method. To do that, we should be able to write test cases for this method without testing getListOfActionsPerformed. We have to assume that getListOfActionsPerformed is tested to work as it is designed. The question is how do we test displayActionsPerformed without executing getListOfActionsPerformed?

That is when mocking comes into the picture. The idea behind mocking is that we will create a mock object so that it can replace the real object and will act the same as the real object. The mock object expects certain methods to be called, and when that happens, it will return some expected result.

In this example, we were having trouble to write unit test cases because we did not really want to execute getListOfActionsPerformed. So now, with the help of mocking, we can mock the LoginService class and create a mock object.

This mock object will call the getListOfActionsPerformed method with userId parameter and it will return say, a list of actions. Now we can easily test the displayActionsPerformed method as we have the result of its dependencies.

Example of Mocking in Scala Using Mockito

Now that you all have understood the concept behind the mocking, let’s look into the Mockito library in ScalaTest.

ScalaTest’s MockitoSugar provides basic syntax sugar for Mockito. There are more choices available that we can use for mocking, such as ScalaMock, EasyMock, and JMock. In this blog, we will be discussing the Mockito framework.

First, we need to add these two library dependencies in our build.sbt file:

Java

```
libraryDependencies ++= Seq ( "org.scalatest" %% "scalatest" % "3.0.1" % "test",
"org.mockito" % "mockito-core" % "2.8.47" % "test" )
```

Now, considering our previous example, let’s write test cases for it using Mockito.

Scala

```
Class UserActionSpec extends WordSpec with Matchers with MockitoSugar {
 val mockedLoginService = mock[LoginService]
 val userAction = new UserAction(mockedLoginService)
“UserAction#displalyActionsPerformed” should {
 “ return SUCCESS” in {
 when(mockedLoginService.getListOfActionsPerformed(any[String])) thenReturn List(“user
logged in”, “user updated profile”)
 val result = userAction.displayActionsPerformed(randomUserId)
 result shouldBe SUCCESS
  }
 }
}
```

Here:

We mocked our LoginService class upon which UserAction depends.

While creating an object for UserAction class we passed on the mocked parameter.

We can see how easily we can stub methods using Mockito functions: when and thenReturn

Mockito also allows us to match against any argument value (as seen in the above example, how we have used any[String] as a matcher for userId parameter:

when(mockedLoginService.getListOfActionsPerformed(any[String])). It also comes with regex matchers and allows us to write custom matchers.

To throw exceptions with Mockito, we simply need to use the thenThrow(….) function. Here is how it can be done,

Scala

```
Class UserActionSpec extends WordSpec with Matchers with MockitoSugar {

val mockedLoginService = mock[LoginService]

val userAction = new UserAction(mockedLoginService)

“UserAction#dispalyActionsPerformed” should {

 “ return error” in {

 when(mockedLoginService.getListOfActionsPerformed(any[String])) thenThrow(

   new RuntimeException())

 intercept[RuntimeException]{userAction.displayActionsPerformed(randomUserId)

   }

  }

 }

}
```

This was all about mocking. For further details on the Mockito framework, you can read it more about it on Mockito User Guide.