Exercise 9

Test your widget

INTRODUCTION

This exercise will cover in detail how the Backoffice Framework supports testing. As you will see, the framework allows for easy testing of UI components with all the technologies and strategies developers are familiar with. These are typically associated with the following areas:

1. Unit testing: we use the standard jUnit framework with Spring and the Backoffice Testing Framework support
2. Mocking: we mock objects using Mockito but you are free to use your own test mock framework or even stubs
3. Test Last / Test Driven Development (TLD/TDD): we demonstrate these two techniques in the lab steps

While it provides the capability of using the same testing tools as developers usually use for business components, Backoffice goes a step further with its testing framework. It delivers generic base classes that allow for automatic testing of several aspects of our code. For example:

1. Testing boundary cases and variations of boundary values in multi-argument methods
2. Calling all visible API methods
3. Testing the existence and availability of appropriate constructors (for example, a no-argument constructor for an editor)

Since the Backoffice Framework is designed to be extensible, the Backoffice Testing Framework has included easy-to-use extensibility verification for your code base. Extensibility means not only the ease of overriding implementations via inheritance, but also the accessibility of UI components and services through extension. The overused Java keyword *private* enables encapsulation but also very often leads to inextensible code, which is difficult to track. It is a good practice to thoroughly test your API and SPI endpoints, but when unit testing the internals of your code, you should be aware that too much encapsulation has a bad side effect that leads to untestable code.

What you will learn

Unit testing capabilities of the Backoffice Framework, including TLD/TDD of UI components (widgets / actions / editors) using jUnit, Mockito and out-of-the-box abstract unit tests.

What you will need before starting

- A basic understanding of unit testing with jUnit and Mockito
- Having run the preparation of exercise 9

INSTRUCTIONS

As part of the preparation for this exercise, a test class has been created and put in the `backoffice/testsrc` directory inside the `bookstorecustombackoffice` project. The test class is created for testing `BookDetailsController` – it is called `BookDetailsControllerTest`.

1. Go through the code inside `BookDetailsControllerTest` class and try to understand what every method and field is for. More specifically note that:
    - Global cockpit events that `BookDetailsController` works with are defined using proper annotations
    - `BookDetailsControllerTest` extends `AbstractWidgetUnitTest<BookDetailsController>`
    - `@InjectMocks` and `Mockito.spy()` were used to create a mock version of the controller

- `@Spy` and `@Mock` annotations were used to create mock objects that will be injected into the mock controller, once `MockitoAnnotations.initMocks()` is run
- `prepareBookInstance()`, `prepareDetailsConfiguration()`, `prepareBaseConfiguration()` are methods that prepare test values for testing and put them in `testBook`, `detailsConfig`, and `baseConfig` correspondingly
- `getWidgetController()` is overridden. This is necessary to provide access to all the widget controller methods coming from `super`
- The `testReceiveBookBySocketEvent()` method is defined as a *test* method by using jUnit's `@Test` annotation. You'll also see `@Ignore` which tells jUnit to ignore this test method for now
- The `setUp()` method has an annotation called `@Before`. In jUnit, that means that this method is invoked by the test runner immediately before each test method in the class is invoked (i.e., "test methods" are designated as such via the @Test annotation). Note, to eliminate side effects between tests, for each @Before / @Test cycle, the jUnit test runner creates a new instance of your test class.
- `setUp`() sets up some mock behavior to imitate some of the services and objects in the controller

Spy vs. Mock

Mockito provides us with two different mechanisms for mimicking the behavior of a class we're testing: spying and mocking.

A *mock* is a bare-bone instance of a class whose methods do nothing unless you define behaviors for them. I.e., the methods default to do-nothing, and YOU have to define (train) ALL do-*something* methods.

On the other hand, a *spy* is a wrapper around an existing instance that allows the methods of the wrapped instance to behave normally unless you define alternate behaviors for them. I.e., Spies default to normal, and you only have to define (train) do-different methods.

For a more detailed explanation, have a look at Mockito's documentation here.

2. As you've already seen, we have declared some *global* cockpit events for the test class using the `@DeclaredGlobalCockpitEvents` annotation. These are the global cockpit events that are listened to in `BookDetailsController`. Something similar can be done for the inputs of the Book Details widget.

   Using the proper annotations (combination of `@DeclaredInputs` and `@DeclaredInput`), declare the inputs of the Book Details widget for the test class. Refer to the Book Details widget's `definition.xml` for the socket names.

3. Run `BookDetailsControllerTest` as a jUnit test. Right click on the test class' name, go to "Run As", and then choose "JUnit Test".

   At this point, only the tests inherited from `AbstractWidgetUnitTest` will run.

4. The test run should have failed on at least these two tests:
   - `TestNullSafeSocketInputs()`
   - `TestExtensibleFields()`

Revisit the slides to help recall why your controller would fail these tests.

You may see that `testHasAllObligatorySocketsDeclared()` fails, despite you completing step 2 and defining all the input sockets. One probable reason could be that you have not defined the type of `allowPreview` input socket consistently everywhere.

If you chose the type as `java.lang.Boolean` in exercise 5, you should do the same everywhere else like when defining the input sockets using `@DeclaredInput` annotation or in the signature of the `handleAllowPreview()` method.

5. Fix the issues in `BookDetailsController` that cause the tests to fail! You may have to change method and field declarations, or even add new methods.

   You should keep doing this until all the tests pass (keeping the following points in mind) …

   - If you see an error message telling you that the null-safe test failed on the `handleSelectedBook()` method for argument `BookModel(<unsaved>)`, simply ignore this error and pay attention to the additional error messages for causes of the null-safe check's failure. More details about this error is explained in the info box below.
   - **TIP:** be sure to look at this unit test's "console output" (both in Eclipse *and* in your DOS or terminal window) for possible details as to what is being tested (specifically during execution of the `testExtensibleFields()` method).

Despite having *null-safe* socket-input handlers, you may still fail `TestNullSafeSocketInputs()` of `AbstractWidgetUnitTest`. The reason is commonly an inadequate test context that fails to provide resources which are otherwise present in a production environment.

In our case, a missing `LocaleProvider` can cause this test to fail. It is required by the service layer of the SAP Commerce Platform. Unfortunately, the Backoffice Framework doesn't provide a dedicated easy way of creating test context when used with the SAP Commerce Platform.

To address this issue, we have used a simple work-around by overriding the test method such that it ignores ("buries") the exception if it's the only one *and* it is caused by a missing `LocaleProvider`, *and* it is caused by something other than a `null` reference parameter, such as an "empty" (not even a PK) `BookModel(<unsaved>)`. This will keep our exercise from becoming *too* complicated.

This is, however, not the best solution in general. A better approach is to refactor the code to guard against `null` references AND empty `Model` objects in our Backoffice application. This approach would give us the possibility to decouple our tests from any specific data layer.

6. By now we've made sure that our widget controller passes the primary tests set by the `AbstractWidgetUnitTest` class. Let's add one test to it ourselves.

   Write the implementation for `testReceiveBookBySocketEvent()` which is ignored at the moment. To enable it, remove `@Ignore` first.

   It should test if the controller behaves correctly when it receives the `testBook` through the widget's "selectedBook" input socket. In terms of *correct behavior*, it should check for these three criteria:

- Check if `populateImgDiv()` is called with the correct input parameters
- Check if `populateInfoDiv()` is called with the correct input parameters
- Check if `addProperty()` is called the correct number of times, i.e., it should be called exactly 3 times

When you try to call the three methods that you're trying to test their behavior, you'll see that your IDE is complaining about them not being visible. The reason is simple: they are `private` and only visible inside `BookDetailsController`. Change their visibility to either `protected` or package (default) visibility (i.e., the visibility Java uses when no visibility keyword is used) to make them

> ⓘ `Mockito.verify()` is the method that enables you to check for the criteria set above. Look at examples here and here to learn how one can use this method. The mock object to use with the Mockito verify() method can be retrieved from `getWidgetController()`

accessible from the test class.

7. After you're done with the implementation, run this unit test. You can do it by right-clicking on the specific test method and then running it as a jUnit test.

   If the test fails, go back to your implementation and try to figure out why. You should expect the test to pass if you've written the test correctly.

Congratulations, you are now ready to work on the rest of the exercises!