Unit Testing Tools for Android

A unit test verifies in isolation the functionality of a certain component. For example, assume a button in an Android activity is used to start another activity. A unit test would determine, if the corresponding intent was issued, not if the second activity was started

A unit tests for an Android application can be:

- local unit test - which runs on the JVM
- Android unit test - which runs on the Android runtime

Advantage and Disadvantages of Unit Testing Tools:

JUnit

JUnit the popular unit-testing framework for Java, provides simple and convenient APIs to write tests and perform common testing operations such as setup, teardown, and assertions with ease.

This is the first testing tool you should learn for Android. Whatever tests you're writing (unit tests, integration tests, etc.), you can use the JUnit framework as the base. You should spend some time learning about the testing lifecycle and the various annotations, such as @Test, @Before, @After, *and* @Rule.

You should always remember that each JUnit test is isolated and should not be dependent on other tests in any way. You should also disregard the order in which these tests run. That's not really important.

Advantages of JUnit testing

- Ability to reuse the older test cases as well as the test data for making a new test case.
- Quick and easy generation of test cases and test data.

- Generate test cases which stick on to their previous test values.
- Promote TDD, i.e. Test Driven Development.
- Enhance productivity and reduce production cost.
- Excellent and highly comprehensive reporting technique.
- Easy contrast between expected output and the output displayed on the console.
- Logical aggregation of test cases.

Disadvantages of JUnit testing

- Cannot do dependency testing.
- Not suitable for higher level testing.

Mockito

The core idea of unit tests is to test a unit of code in isolation from any of its dependencies. When you are unit testing a piece of code, you really don’t care if its dependencies work properly or not, since you will need to test them separately anyway.

But sometimes you can't test code without data from another part of the application. Therefore, to make unit testing possible, you need to "mock" out the dependencies. This is where Mockito comes into play. This amazing framework provides a very simple and fluent API to help you create mock objects with ease. You can stub the state of a particular method or create mock objects to verify interactions. (You should understand the difference between mocks and stubs, since developers often confuse the two.)

You should definitely check out the mock(), *when()*, and thenReturn() APIs to stub method states, and also the verify() API to check mock interactions. You should also consider checking out the ArgumentCaptor API, which can help you capture arguments passed to the mocks so you can act on them later.

Advantages of Mockito

- We can Mock any class or interface as per our need.
- It supports Test Spies, not just Mocks, thus it can be used for partial mocking as well.

Disadvantages of Mockito

- *It takes sooooooooo long, even more if you are a test-coverage addict.*
- The mocking doesn't always work as you expect. This is what motivates this article.
- Mockito cannot test static classes. So, if you're using Mockito, I would recommend to change static classes for Singletons.
- You have to create methods otherwise you wouldn't need. This is caused by the mocking again, getters & setters are needed to allow the mocking.
- Why all the methods have to be public? This annoys me a lot. Yes, you cannot test a private method.

Espresso

Espresso is an excellent framework developed by Google to help you easily and quickly automate UI tests that simulate real user behavior. These tests will run on a real device or emulator.

The Espresso APIs are very simple to understand and use. You can find any View using the onView() method. Once you have the view, you can either perform() any action (such as click(), swipe(), etc.) on it and check() that certain properties of the view match the expected results.

The great thing about Espresso is that it can automatically handle UI synchronization, so you can get rid of those smelly Thread.sleep() methods. The APIs are also quite extensible, and they give you rich error information on failures.

1. **Espresso Workflow Is Simple to Use**

The way Espresso works is by allowing developers to build a test suite as a stand-alone APK that can be installed on the target devices alongside the application under test and be executed very quickly.

2. **Fast and Reliable Feedback to Developers**

As developers are trying to accelerate deployment, Espresso gives them fast feedback on their code changes so they can move on to the next feature or defect fix; having a robust and fast test framework plays a key role.

Espresso does not require any server (selenium remote web driver) to communicate with; instead it runs side-by-side with the app and delivers very fast (minutes) test results to the developer.

3. **Less Mobile Testing Flakiness**

Because Espresso offers a synchronized method of execution, the stability of the test cycle is very high. There's a built-in mechanism in Espresso that, prior to moving to the next steps in the test, validates that the Element or Object is actually displayed on the screen. This eliminates test execution from breaking when confronted with "objects not detected" and other errors.

4. **Developing Espresso Test Automation Isn't Hard**

Developing Espresso test automation is quite easy. It is based on Java and Junit, which is a core skill set for any Android app developer. Because Espresso works seamlessly within the Android Studio IDE, there's no setup or ramping up and no "excuses" – to actually shift quality in the in-cycle stage of the app SDLC.

In addition to the above, there is, of course, the large community powered by Google that pushes the Espresso test automation framework and allow easy and fast ramp up for newcomers.

Espresso Drawbacks:

- There is a risk to get used to the in-built test synchronization and UI – then it might be hard to work with WebDriver.
- It requires access to the application's source code. Without the source code, you won't be able to do anything.
- Narrow focus. If UI tests are required for both Android and iOS, it will be necessary to write twice, for two different systems. If tests require to work with Android outside the application (for example, open a received notification with a text message), you'll have to use additional tools, such as UIAutomator.
- It is desirable to have at least minimal experience of building and launching Android applications on emulators.

Robolectric

Espresso tests are great, but the only problem is that they are slow because they cannot run on the local JVM—they need to run on a real device or emulator. This makes testing Android APIs less convenient than running JUnit tests.

You should run your tests as frequently as possible so that you can catch bugs early in your development cycle. But if your tests take a lot of time to run, you will not want to run them as often, which sometimes defeats the purpose of testing early and often.

Robolectricmakes it possible to run Android tests right on your local machine by providing a dummy implementation of the Android SDK using shadow classes.

But there are some downsides to using Robolectric. It is always one step behind the latest Android SDK available. And though Robolectric usually works well, when it doesn’t, you might spend days or even weeks fixing the issue. So you might want to use it only when you really need to.

Advantages of Robolectric

- The most significant advantage of running tests on the JVM is the higher speed of execution compared to instrumented tests. As a result of not having to launch an emulator or a device, Espresso works recognisably slower than Robolectric.

- Robolectric tests can easily be executed on a continuous integration server. We experienced that it’s somewhat complicated and laborious to set up a working environment for instrumented tests on build servers.
- The unit tests are executed by referring to a third party implementation of the Android API. Developers therefore tend to adopt a sceptical attitude towards frameworks like Robolectric arguing that the tests are not based on the actual Android API but on a reflection of it. As we worked along with Robolectric, we didn’t notice any limitation due to missing Android API features. Almostevery Android class is available.
- Robolectric also stands out when it comes to multi threaded code (e.g. background work with AsyncTask).

Disadvantages of Robolectric

- Like already mentioned, Robolectric covers almost every feature of the Android API but not all of them. Every so often Google publishes new Android versions including new features and Java classes. As a result, we need to keep in mind that functionalities of newer API versions might not be available at the very same time they appear.

- Though Robolectric facilitates most of the unit tests we are not getting around performing UI and integration tests (e.g. testing multi touch,…). In fact, it would be grossly negligent just to rely on Robolectric tests.

UIAutomator

Espresso is really good at what it does, but it has limitations. You will not be able to write UI tests that involve interactions across multiple apps or the system UI with Espresso.

There could be several scenarios where you need to write UI tests that can span multiple apps. For example, you might have an app in which users can tap a button to open the image viewer, pick a particular image, and then return to the original app to display it. Espresso simply won't work (without making some sort of compromise) if you need to test these kinds of scenarios.

Advantage of UIAutomator

UI Automatordoes have the ability to interact with views that are beyond your app's boundary. You won't always need to use this framework, but it's useful to pull it out when building apps that interact a lot with third-party apps and system UI components.

Disadvantage of UIAutomator

The downside of the UI Automator is that it doesn't support WebView, upon which hybrid Android apps are built. Therefore, UI Automator supports only native Android apps.