Android Development with Kotlin

Testing

Testing

Software testing is a process, to evaluate the functionality of a software application with an intent to find whether the developed software met the specified requirements or not and to identify the defects to ensure that the product is defect-free in order to produce the quality product. As the code becomes more complex, the probability of successful execution decreases substantially. ***Rigorous testing can hence ensure that the code meets its standards and all its features are error and bug free.***

Any modifications in an app module in the later stage can affect the previously implemented features, in this case, testing the entire system post implementation becomes even more important.

Testing can be performed both manually and using automated techniques. Automated testing environments are faster, error free and hence more reliable, whereas manual testing environment are prone to human errors. Therefore, ***Automated testing is preferred to ensure that the application meets the highest standards.***

Test Driven Development

Test-driven development starts with developing test for each one of the features. The test might fail as the tests are developed even before the development. Development team then develops and refactors the code to pass the test.

Test-Driven Development starts with designing and developing tests for every small functionality of an application. TDD instructs developers to write new code only if an automated test has failed. This avoids duplication of code. The simple concept of TDD is to write and correct the failed tests before writing new code (before development). This helps to avoid duplication of code as we write a small amount of code at a time in order to pass tests. (Tests are nothing but requirement conditions that we need to test to fulfil them).

Test-Driven development is a process of developing and running automated test before actual development of the application. Hence, TDD sometimes also called as Test First Development.

Functional and Non-functional Requirements

In software engineering, ***a functional requirement defines a system or its component***. It describes the functions a software must perform. A function is nothing but inputs, its behavior, and outputs. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform. ***Functional software requirements help you to capture the intended behavior of the system. This behavior may be expressed as functions, services or tasks or which system is required to perform.***

A non-functional requirement defines the quality attribute of a software system. They represent a set of standards used to judge the specific operation of a system. Example, how fast does the website load? A non-functional requirement is essential to ensure the usability and effectiveness of the entire software system. Failing to meet non-functional requirements can result in systems that fail to satisfy user needs. Non-functional Requirements allows you to impose constraints or restrictions on the design of the system across the various agile backlogs. Example, the site should load in 3 seconds when the number of simultaneous users

are > 10000. Description of non-functional requirements is just as critical as a functional requirement.

Testing is performed to improve the functional properties of the system, and non-functional requirements are improved be refactoring the code to improve its performance and make code more optimal.

To read more about refactoring visit: https://bit.ly/3hBW7O3

Regression Testing

Regression Testing is a type of testing that is done to verify that a code change in the software does not impact the existing functionality of the product. This is to make sure the product works fine with new functionality, bug fixes or any change in the existing feature. Previously executed test cases are re-executed in order to verify the impact of change.

Regression Testing is a Software Testing type in which ***test cases are re-executed in order to check whether the previous functionality of the application is working fine and the new changes have not introduced any new bugs.*** This test can be performed on a new build when there is a significant change in the original functionality that too even in a single bug fix.

Regression means retesting the unchanged parts of the application.

Unit Testing

Unit testing, a testing technique using which individual modules are tested to determine if there are any issues by the developer himself. It is concerned with functional correctness of the standalone modules. The main aim is to isolate each unit of the system to identify, analyze and fix the defects. Advantages of Unit Testing are as follows:

- Reduces Defects in the Newly developed features or reduces bugs when changing the existing functionality.
- Reduces Cost of Testing as defects are captured in very early phase.
- Improves design and allows better refactoring of code.
- Unit Tests, when integrated with build gives the quality of the build as well.

Android has two types of unit-testing environments, Local Unit Testing and Instrumented Testing. Local Unit Testing uses JUnit to run tests without the Android framework, whereas instrumented tests are run on an emulator or an Android device using the Android framework.

Local Unit Testing

You can evaluate your app's logic using local unit tests when you need to run tests more quickly and don't need the fidelity and confidence associated with running tests on a real device. With this approach, you normally fulfill your dependency relationships using either Robolectric or a mocking framework, such as Mockito. Usually, the types of dependencies associated with your tests determine which tool you use:

- If you have dependencies on the Android framework, particularly those that create complex interactions with the framework, it's better to include framework dependencies using **Robolectric**.
- If your tests have minimal dependencies on the Android framework, or if the tests depend only on your own objects, it's fine to include mock dependencies using a mocking framework like **Mockito**.

To add unit tests to your Android Application use the following steps:

1. Set up the dependencies

```
dependencies {
    // Required -- JUnit 4 framework
    testImplementation 'junit:junit:4.12'
    // Optional -- Robolectric environment
    testImplementation 'androidx.test:core:1.0.0'
    // Optional -- Mockito framework
    testImplementation 'org.mockito:mockito-core:1.10.19'
}
```

2. Create a Local Unit Test case within a class in the "test" sub-directory within the "src" directory

```
import com.google.common.truth.Truth.assertThat
import org.junit.Test
class EmailValidatorTest {
    @Test
    fun emailValidator_CorrectEmailSimple_ReturnsTrue() {
        assertThat(EmailValidator.isValidEmail("name@email.com"))
            .isTrue()
    }
}
```

Note: The assertThat function checks if the returned value is equal to the expected value

3. **Using the Robolectric Framework**

   If your tests interact with several Android framework dependencies, or interact with those dependencies in a complex way, use the Robolectric artifacts that AndroidX Test provides. Robolectric executes real Android framework code and fakes of native framework code on your local JVM or on a real device.

   The following example shows how you might create a unit test that uses Robolectric:

   1. Add dependencies to the build.gradle
   2. Allow the testing environment to access the resource files

```
android {
    // ...
    testOptions {
        unitTests.includeAndroidResources = true
    }
}
```

3. Access the simulated context within your Test Class

```
import android.content.Context
import androidx.test.core.app.ApplicationProvider
import com.google.common.truth.Truth.assertThat
import org.junit.Test
private const val FAKE_STRING = "HELLO_WORLD"
class UnitTestSample {
    val context =
       ApplicationProvider.getApplicationContext<Context>()
    @Test fun readStringFromContext_LocalizedString() {
        // Given a Context object retrieved from Robolectric...
        val myObjectUnderTest = ClassUnderTest(context)
          // ...when the string is returned from the object under
test...
                                        val    result:    String    =
myObjectUnderTest.getHelloWorldString()
        // ...then the result should be the expected one.
        assertThat(result).isEqualTo(FAKE_STRING)
    }
}
```

4. **Using the Mockito Library**

   If you have minimal Android dependencies and need to test specific interactions between a component and its dependency within your app, use a mocking framework to stub out external dependencies in your code. That way, you can easily test that your component interacts with the dependency in an expected way. By substituting Android dependencies with mock objects, you can isolate your unit test from the rest of the Android system while verifying that the correct methods in those dependencies are called. The Mockito mocking framework for Java (version 1.9.5 and higher) offers compatibility with Android unit testing. With Mockito, you can configure mock objects to return some specific value when invoked.

   To add a mock object to your local unit test using this framework, follow this programming model:

   1. Include the Mockito library dependency in your build.gradle file, as described in Set up your testing environment.
   2. At the beginning of your unit test class definition, add the @RunWith(MockitoJUnitRunner.class) annotation. This annotation tells the Mockito test runner to validate that your usage of the framework is correct and simplifies the initialization of your mock objects.
   3. To create a mock object for an Android dependency, add the @Mock annotation before the field declaration.
   4. To stub the behavior of the dependency, you can specify a condition and return value when the condition is met by using the when() and thenReturn() methods.

      A sample code for the same has been added below:

```
import android.content.Context
import com.google.common.truth.Truth.assertThat
```

```
import org.junit.Test
import org.junit.runner.RunWith
import org.mockito.Mock
import org.mockito.Mockito.`when`
import org.mockito.junit.MockitoJUnitRunner
private const val FAKE_STRING = "HELLO WORLD"
@RunWith(MockitoJUnitRunner::class)
class UnitTestSample {
    @Mock
    private lateinit var mockContext: Context
    @Test
    fun readStringFromContext_LocalizedString() {
         // Given a mocked Context injected into the object under
test...
        `when`(mockContext.getString(R.string.hello_word))
                .thenReturn(FAKE_STRING)
        val myObjectUnderTest = ClassUnderTest(mockContext)
         // ...when the string is returned from the object under
test...
                                        val    result:    String    =
myObjectUnderTest.getHelloWorldString()
        // ...then the result should be the expected one.
        assertThat(result, `is`(FAKE_STRING))
    }
}
```

To read more about Local Unit Test visit: https://developer.android.com/training/testing/unit-testing/local-unit-tests

Instrumented Testing

Instrumented unit tests are tests that run on physical devices and emulators, and they can take advantage of the Android framework APIs and supporting APIs, such as AndroidX Test. Instrumented tests provide more fidelity than local unit tests, but they run much more slowly. Therefore, we recommend using instrumented unit tests only in cases where you must test against the behavior of a real device. AndroidX Test provides several libraries that make it easier to write instrumented unit tests when necessary. For example, Android Builder classes make it easier to create Android data objects that would otherwise be difficult to build.

To create Instrumented Test, follow the following steps:

1. Add dependencies to the build.,gradle

```
dependencies {
    androidTestImplementation 'androidx.test:runner:1.1.0'
    androidTestImplementation 'androidx.test:rules:1.1.0'
    // Optional -- UI testing with Espresso
    androidTestImplementation
            'androidx.test.espresso:espresso-core:3.1.0'
}
```

2. Specify AndroidJUnitRunner as the default test instrumentation runner in your project by including the following setting in your app's module-level build.gradle file

```
android {
    defaultConfig {
        testInstrumentationRunner
                "androidx.test.runner.AndroidJUnitRunner"
    }
}
```

3. Create a Unit Test for your classes. The following example creates a test for a ViewModel to ensure that the live data returns the correct data.

```
@RunWith(AndroidJUnit4::class)
class LocationViewModelTest{
    @get:Rule
    var instantExecutorRule = InstantTaskExecutorRule()
    private val location = LatLong(28.6139, 77.2090)
    @Test
    fun setLocation_ReturnsUserLocation(){
        val locationViewModel =
            LocationViewModel(ApplicationProvider.getApplicationConte
            xt())
        val observer = Observer<String> {}
        try {
            locationViewModel.userLocation.observeForever(observer)
            locationViewModel.setLocation(location)
            val value = locationViewModel.userLocation.value
            assertEquals(value, "${location.lat} ${location.long}")
        } finally {
             locationViewModel.userLocation.removeObserver(observer)
        }
    }
}
```

Note: The InstantExecutorRule ensures that the tasks are run in the same thread for better testing.