Testing and Publishing Kotlin App

Tushar B. Kute,
http://tusharkute.com

App Testing

- App testing, or more broadly, software application testing, is a systematic process of evaluating a mobile, web, or desktop application to ensure it functions correctly, is user-friendly, and is free of defects.
- The primary goal is to verify that the application meets its specified requirements and provides a high-quality user experience before it is released to the public.

Testing

- Unit testing
  - focuses on the smallest, isolated components of code, like a single function or a method.
- UI testing
  - focuses on the user interface and how a user interacts with the application as a whole.

Unit Testing

- Unit testing is a low-level testing approach performed by developers.
- The goal is to verify that a specific "unit" of code performs its function correctly and as expected.
- These tests are fast, automated, and run frequently, often as part of the development cycle.
- They're designed to be isolated from external factors like databases or APIs, so developers use "mocks" or "stubs" to simulate these dependencies.
- This isolation makes it easy to pinpoint exactly where a bug exists.

Unit Testing : Key characteristics

- Scope:
  - A single, isolated unit of code (e.g., a function, class, or method).
- Who performs it: Developers.
- Purpose:
  - To validate the internal logic and correctness of the code itself.
- Tools:
  - Unit testing frameworks built into programming languages (e.g., JUnit for Java, pytest for Python, Jest for JavaScript).

UI Testing

- UI (User Interface) testing, also known as GUI (Graphical User Interface) testing, is a high-level testing approach that simulates the actions of a real user.
- The purpose is to ensure that all the visual elements (buttons, forms, menus, etc.) and the user's workflow function correctly and consistently across different devices and browsers.
- This type of testing is often automated and runs less frequently than unit tests because it's much slower and more complex to set up.

UI Testing: Key characteristics

- Scope:
  - The entire application from the user's perspective, including the graphical elements and the navigation flow.
- Who performs it: QA engineers or testers, and sometimes developers.
- Purpose: To validate the user experience and ensure the application is functional, usable, and visually correct.
- Tools: Automation frameworks that interact with the browser or device (e.g., Selenium, Cypress, Playwright).

JUnit

- JUnit is a widely-used, open-source testing framework for the Java programming language.
- It is primarily used for unit testing, which involves testing individual, isolated components of an application, such as a single method or class.

JUnit

- JUnit helps developers to:
  - Write and run automated, repeatable tests for their code.
  - Catch bugs early in the development cycle, reducing the time and effort needed for debugging later.
  - Ensure code quality and reliability, making it safer to add new features or refactor existing code.
  - Facilitate test-driven development (TDD), an approach where you write tests before writing the actual code.

JUnit

- JUnit helps developers to:
  - Write and run automated, repeatable tests for their code.
  - Catch bugs early in the development cycle, reducing the time and effort needed for debugging later.
  - Ensure code quality and reliability, making it safer to add new features or refactor existing code.
  - Facilitate test-driven development (TDD), an approach where you write tests before writing the actual code.

JUnit: How is works?

- JUnit uses a simple structure based on annotations and assertions.
  - Annotations: Special tags like @Test, @BeforeEach, and @AfterEach are used to define the purpose of a method. For example, @Test marks a method as a test case, while @BeforeEach and @AfterEach are used to set up and clean up resources before and after each test.
  - Assertions: These are methods (e.g., assertEquals(), assertTrue()) that verify if the actual result of a test matches the expected outcome. If an assertion fails, JUnit marks the test as a failure and provides a report, which helps developers quickly identify what went wrong.

JUnit: with Kotlin

- While JUnit was originally built for Java, it's also fully compatible with Kotlin. Kotlin's interoperability with Java makes using JUnit a seamless experience.
- Why Use JUnit with Kotlin?
  - Established and Robust: JUnit is a mature and widely used framework.
  - Easy to Learn: If you're familiar with unit testing, JUnit's syntax is straightforward.
  - Seamless Integration: It works perfectly with all major build tools like Gradle and Maven, which are commonly used in Kotlin projects.

JUnit: with Kotlin

- Example: Testing a simple Kotlin class with JUnit
- Let's test a simple Calculator class.
- 1. The Kotlin Class to be Tested

  First, here's the Calculator class we want to test. Save this in a file named Calculator.kt.

```
class Calculator {
    fun add(a: Int, b: Int): Int {
        return a + b
    }
}
```

JUnit: with Kotlin

- 2. Dependencies

  To use JUnit 5 (the latest version), you need to add the following dependencies to your build.gradle.kts file.

```
dependencies {
    testImplementation(platform("org.junit:junitbom:5.10.1"))
    testImplementation("org.junit.jupiter:junit-jupiter")
}
tasks.test {
    useJUnitPlatform()
}
```

JUnit: with Kotlin

- 3. The JUnit Test Class
- Now, create a test class in your src/test/kotlin directory. Let's call it CalculatorTest.kt.

  This test class uses the @Test annotation from JUnit to mark a function as a test case. We'll use the assertEquals assertion to check if the add method returns the correct result.

```
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.Assertions.assertEquals
class CalculatorTest {
    @Test
    fun `test add function returns correct sum`() {
        // Arrange (Setup the test data)
        val calculator = Calculator()
        val num1 = 5
        val num2 = 3
        val expectedSum = 8
        // Act (Call the method being tested)
        val result = calculator.add(num1, num2)
        // Assert (Verify the result)
        assertEquals(expectedSum, result, "The add function should return the correct sum.")
    }
}
```

Finally: Running the test

- You can run this test directly from your IDE (like IntelliJ IDEA) by right-clicking the test file and selecting "Run". If you are using a build tool, you can run the test from the terminal using a command like ./gradlew test.
- If the add function works as expected, the test will pass. If it returns an incorrect value, the assertion will fail, and JUnit will provide a detailed report showing the expected and actual values.
- This simple example shows how JUnit's familiar annotations and assertions make it easy to write robust unit tests for your Kotlin code.

Espresso

- Espresso is an open-source testing framework developed by Google for UI (User Interface) testing of Android applications.
- It's a key part of the AndroidX Test library and is widely used for writing automated tests that simulate user interactions, like clicks, swipes, and text entry, on an Android device.

How Espresso works?

- Espresso works by synchronizing with the Android application's UI thread. It intelligently waits for the app to be idle before performing an action or an assertion. This "idle waiting" is a core feature that makes Espresso tests highly reliable and eliminates a common problem in UI testing: flaky tests caused by timing issues.
- Espresso tests follow a simple, readable structure that can be summarized as:
  - Find the View: Use a ViewMatcher to locate a specific UI component (e.g., a button with the text "Submit" or an EditText with a certain ID).
  - Perform an Action: Use a ViewAction to simulate a user action on that view (e.g., click(), typeText()).
  - Check the Result: Use a ViewAssertion to verify that the view is in the expected state after the action (e.g., matches(isDisplayed()), hasText("Success")).
- This three-step process is often represented by the following syntax:
  - onView(ViewMatcher).perform(ViewAction).check(ViewAssertion)

Espresso : Benefits

- Reliability: Espresso's synchronization with the UI thread makes tests less prone to flakiness.
- Speed: It is fast because it directly interacts with the application's UI components without needing to wait for a full UI render like a human would.
- Readability: The API is concise and readable, making tests easier to write and maintain.
- Isolation: Espresso tests are designed to be isolated, meaning each test case should only depend on its own data and actions.
- Espresso is a powerful tool for ensuring the stability and correctness of an Android app's user interface, especially as the app grows in complexity.

Espresso : with Kotlin

- Espresso works seamlessly with Kotlin. The framework's core syntax and APIs are fully compatible, allowing developers to write concise and expressive UI tests for Android applications using the Kotlin programming language.

Setting Up a Test Environment

- To use Espresso in a Kotlin project, you need to add the necessary dependencies to your app's build.gradle.kts file. This includes the Espresso core library and the JUnit 4 rule for test execution.

```
dependencies {
    // Other dependencies
    // Espresso
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
    // JUnit for testing rules
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    // The test runner and orchestrator
    androidTestImplementation("androidx.test:runner:1.5.2")
    androidTestImplementation("androidx.test:rules:1.5.0")
}
```

- The androidTestImplementation configuration ensures these libraries are only available for your instrumented tests.

Example: Testing a Login Screen

- Let's imagine a simple login screen with two EditText fields (for email and password) and a "Login" button.

  1. The Layout (activity_main.xml)

```
<EditText
    android:id="@+id/email_edit_text"
    android:hint="Email" />
<EditText
    android:id="@+id/password_edit_text"
    android:hint="Password" />
<Button
    android:id="@+id/login_button"
    android:text="Login" />
```

The Espresso Test in Kotlin

```
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.action.ViewActions.*
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.*
import androidx.test.ext.junit.rules.ActivityScenarioRule
import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith
@RunWith(AndroidJUnit4::class)
class LoginActivityTest {
  // Rule to launch the activity under test before each test method
  @get:Rule
  val activityRule = ActivityScenarioRule(LoginActivity::class.java)
  @Test
  fun loginWithValidCredentials_displaysSuccessMessage() {
    // 1. Find the email EditText and type text
    onView(withId(R.id.email_edit_text))
      .perform(typeText("test@example.com"), closeSoftKeyboard())
    // 2. Find the password EditText and type text
    onView(withId(R.id.password_edit_text))
      .perform(typeText("password123"), closeSoftKeyboard())
    // 3. Find the login button and click it
    onView(withId(R.id.login_button))
      .perform(click())
    // 4. Check if the success message is displayed
    onView(withText("Login successful!"))
      .check(matches(isDisplayed()))
  }
}
```

Key Components

- @RunWith(AndroidJUnit4::class): This annotation specifies the test runner that will execute the tests.
- @get:Rule val activityRule = ...: The ActivityScenarioRule is a JUnit 4 rule that automatically launches the specified activity before each test.
- onView(): This is the core Espresso method used to find a UI element. It takes a ViewMatcher (like withId()) as a parameter.
- .perform(): This method executes a ViewAction (like typeText() or click()) on the matched view.
- .check(): This method verifies the state of the view using a ViewAssertion (like matches(isDisplayed())).
- Using Espresso with Kotlin provides the benefit of both powerful UI testing capabilities and the concise, readable syntax that Kotlin is known for.

Prepare and Publish the App

- Preparing and publishing a Kotlin app to the Google Play Store is a multi-step process that involves preparing your app for a public release, creating your store listing, and finally, uploading and submitting your app for review.
- It's a structured process that ensures your app meets Google's quality and policy standards.

Preparation in Android Studio

- Before you can upload your app, you need to prepare a production-ready build. This process is the same whether your app is written in Kotlin, Java, or any other language that compiles to Android.
  - Finalize Your Code: Ensure your app is stable, bug-free, and has been thoroughly tested. Run a final set of tests, including unit and UI tests, to catch any last-minute issues.
  - Update App Information: In your build.gradle (or build.gradle.kts) file, update the versionCode and versionName. The versionCode must be a unique, incrementing integer for every new release, and the versionName is the public-facing version number (e.g., "1.0.0").

Preparation in Android Studio

- App Signing: Android requires every app to be digitally signed with a certificate. For publishing, you'll need to create a release keystore. You can do this in Android Studio by going to Build > Generate Signed Bundle / APK.... Android Studio will walk you through creating a new keystore file (.jks or .keystore) and a key alias.
- Generate an Android App Bundle (AAB): Google's preferred format for app distribution is the Android App Bundle. This format allows Google Play to generate and serve optimized APKs to users based on their device configurations, which results in smaller app downloads. Select "Android App Bundle" in the "Generate Signed Bundle / APK" dialog.
- Proguard/R8: To shrink and obfuscate your code, enabling Proguard or R8 is a best practice for production builds. This removes unused classes and members, making your app smaller and more difficult to reverse-engineer. You enable it in your build.gradle file by setting minifyEnabled true and adding a proguard-rules.pro file.

Prepare Your Google Play Console Listing

- This is the public-facing marketing material for your app. A good listing is crucial for attracting users.
  - Create a Developer Account: If you haven't already, you must create a Google Play Developer account and pay the one-time registration fee.
  - App Name and Descriptions: You'll need to provide your app's name, a short description (up to 80 characters), and a full description (up to 4000 characters). These should accurately and compellingly describe your app's features.

Prepare Your Google Play Console Listing

- Graphic Assets: Prepare high-quality visual assets.
  - App Icon: A high-resolution icon (512x512 pixels).
  - Feature Graphic: A promotional banner (1024x500 pixels) that appears at the top of your store listing.
  - Screenshots: At least two screenshots that showcase your app's key features on a phone, with additional screenshots recommended for tablets and other form factors.
- Content Rating and Privacy Policy: You'll complete a questionnaire to get a content rating for your app. You must also provide a URL to a valid privacy policy, especially if your app collects any user data.

Upload and Publish

- The final stage is to upload the generated AAB to the Google Play Console and release it.
  - Upload Your App Bundle: In the Play Console, create a new app and go to the "Production" track. Upload the .aab file you generated in Android Studio.
  - Set Up Your Release: Create a new release and add the uploaded app bundle. You'll be asked to name the release and provide release notes (what's new in this version).

Summary

- Testing and Preparation: Before publishing, ensure the app is stable and bug-free by performing both unit testing (e.g., with JUnit) for code correctness and UI testing (e.g., with Espresso) to verify user interactions.
- App Bundling: The final release build must be a signed Android App Bundle (AAB), which is the preferred format for the Google Play Store to optimize the app for different devices.
- Google Play Console: The app is then uploaded to the Google Play Console, where you'll create a store listing with descriptions and screenshots before submitting it to Google for review and final publication.

Thank you

@mitu_skillologies

@mITuSkillologies

@mitu_group

@mitu-skillologies

@MITUSkillologies

kaggle

@mituskillologies

Web Resources
https://mitu.co.in
http://tusharkute.com

@mituskillologies

contact@mitu.co.in

tushar@tusharkute.com