



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

🚀 Day 27

Testing Strategies

(Part 1 of 2)

Testing (Unit, Integration, UI) - Fundamentals for
Reliable Apps 🚀

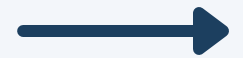


Esmaeeil Moradi

@esmaeeil-moradi700



Day 27



of #30DayAndroidChallenge:

Comprehensive Testing Strategies: Testing (Unit, Integration, UI) - Fundamentals for Reliable Apps 🚀

Hey Android Architects, ambitious developers, and anyone who values the stability and quality of their code!

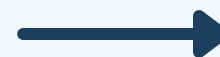
Are you worried about hidden failures in your code? Do you know how damaging a small bug can be in a financial system? It's time to gain absolute confidence in your code through testing, building quality from the ground up! Join me to learn the fundamentals of testing from the perspective of an Android Architect with over 10 years of experience and deep knowledge of AOSP.

In today's world, building an Android application without a comprehensive testing strategy is like constructing a skyscraper without a solid foundation. Problems such as Crashes, ANRs (Application Not Responding), Incorrect Calculations, and Data Loss all stem from insufficient testing. These weaknesses not only severely degrade the user experience but can also lead to skyrocketing maintenance costs and significant damage to a company's reputation.



Esmaeel Moradi

@esmaeel-moradi700



Especially in sensitive domains like digital financial systems or cryptocurrency wallets (Web3), where a single bug can lead to millions in losses, testing is not just a feature, but a moral and technical imperative. For us, who are building critical platforms, this is a fundamental principle. Testing is the solution to ensure the quality, reliability, and scalability of your code.



Esmaeeil Moradi
@esmaeeil-moradi700

Unit Testing: Take Control of the Details!

Description: Unit Testing involves testing the smallest logical unit of your code (such as a class, a method, or a function) in an isolated and independent manner. The goal is for each component to demonstrate its correct functionality on its own, without relying on other parts.

Why is Unit Testing crucial?

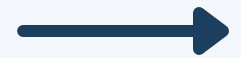
- **Rapid Bug Identification:** Bugs are discovered in the earliest stages of development, within the smallest code unit, making them much cheaper and faster to fix.
- **Improved Code Design (Testable Code):** The need for testability forces you to write more modular, less dependent, and better-designed code.
- **Ensuring Component Correctness:** Every time you make changes to your code, unit tests assure you that existing parts continue to work correctly.
- **Facilitating Refactoring:** With strong unit tests, you can refactor your code with confidence, knowing that the tests will alert you if functionality is broken.



Esmaeel Moradi

@esmaeel-moradi700

Tools:



- **JUnit:** A standard framework in Java/Kotlin for writing and executing unit tests.
- **Mockito:** A tool for Mocking (simulating) the dependencies of a code unit. This allows you to test **the component in complete isolation**.

Structure of a Unit Test:

We typically follow the Given-When-Then (or Arrange-Act-Assert) pattern:

- **Given/Arrange:** Prepare the prerequisites (e.g., initialize objects).
- **When/Act:** Execute the desired operation (call the method or function).
- **Then/Assert:** Verify the expected outcome or behavior.

Code Example:

Testing a Calculator Class

Suppose we have a simple class for performing arithmetic operations:

```
// Example: Calculator.kt - Your production code
class Calculator {
    fun add(a: Int, b: Int) = a + b
    fun subtract(a: Int, b: Int) = a - b
    fun multiply(a: Int, b: Int) = a * b
}
```



Esmaeel Moradi

@esmaeel-moradi700

Now, the unit tests for it:



```
import org.junit.Before
import org.junit.Test

class CalculatorTest {
    private lateinit var calculator: Calculator

    // This method runs before each test method
    @Before
    fun setup() {
        calculator = Calculator()
    }

    @Test
    fun `add returns correct sum for positive numbers`() {
        val result = calculator.add(2, 3)
        assertEquals(5, result) // Assert that the expected result is 5
    }

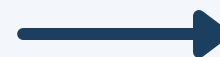
    @Test
    fun `subtract returns correct difference`() {
        val result = calculator.subtract(5, 2)
        assertEquals(3, result)
    }

    @Test
    fun `multiply returns correct product`() {
        val result = calculator.multiply(4, 5)
        assertEquals(20, result)
    }

    @Test
    fun `add with negative numbers works correctly`() {
        val result = calculator.add(-2, 5)
        assertEquals(3, result)
    }
}
```



Esmaeel Moradi
@esmaeel-moradi700



In systems where the slightest error can lead to financial losses or data loss (such as financial systems or cryptocurrency wallets), even a small mistake in business logic (e.g., balance calculation or transaction amount) can be catastrophic. Unit Tests ensure that each logic unit works correctly, even before it's connected to other parts. This precision is the foundation of trust in any critical system.



Esmaeail Moradi

@esmaeail-moradi700

Integration Testing: Verify the Connections!



Description: Integration Testing, instead of isolating units, focuses on the interaction between multiple units or components. These tests verify that when different parts of the application (e.g., a ViewModel and a Repository, or a Fragment with its ViewModel) work together, they behave as expected.

Why is Integration Testing important?

- **Identifying Interaction Bugs:** Problems arising from mismatches or misunderstandings in communication protocols between components are identified at this level.
- **Data Flow Validation:** Ensuring that data flows correctly between different layers (e.g., from the data layer to the business logic layer, and then to the UI).
- **Broader Coverage:** These tests cover more layers of the application compared to unit tests.

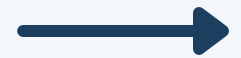
Tools:

- **AndroidX Test:** A set of tools and Rules for testing Android components.
- **Espresso:** In some cases, Espresso can be used for light integration tests that also involve simple UI interactions.
- **Mockito (again):** For Mocking heavier dependencies (e.g., network APIs or databases) while testing the interaction between two units.



Esmaeel Moradi

@esmaeel-moradi700



- **Code Example:**

Testing ViewModel Interaction with Repository

Suppose we have a ViewModel that fetches data from a Repository:

```
// Example: MyRepository (interface or simple class) - Production code
interface MyRepository {
    fun getData(): String
    fun saveData(data: String)
}

// Example: MyViewModel - Production code
class MyViewModel(private val repository: MyRepository) {
    fun fetchData(): String {
        return "Fetched from VM: " + repository.getData()
    }

    fun storeData(data: String) {
        repository.saveData(data)
    }
}
```



Esmaeel Moradi
@esmaeel-moradi700

Now, the integration test (mocking the repository to focus on ViewModel logic):

```
import org.junit.Assert.assertEquals
import org.junit.Before
import org.junit.Test
import org.mockito.Mockito.mock
import org.mockito.Mockito.`when` // Use backticks for 'when' keyword
import org.mockito.Mockito.verify // For verifying method calls

class MyViewModelTest {
    private lateinit var mockRepository: MyRepository
    private lateinit var viewModel: MyViewModel

    @Before
    fun setup() {
        mockRepository = mock(MyRepository::class.java) // Create a mock object for repository
        viewModel = MyViewModel(mockRepository)
    }

    @Test
    fun `fetchData returns data prefixed from repository`() {
        // Given: The mock repository returns "Test Data" when getData() is called
        `when`(mockRepository.getData()).thenReturn("Test Data")

        // When: fetchData is called on the ViewModel
        val result = viewModel.fetchData()

        // Then: The result should be as expected, and getData() should have been called
        assertEquals("Fetched from VM: Test Data", result)
        verify(mockRepository).getData() // Verify that getData was indeed called
    }

    @Test
    fun `storeData calls saveData on repository`() {
        // Given: Some data
        val dataToStore = "Hello World"

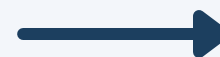
        // When: storeData is called on the ViewModel
        viewModel.storeData(dataToStore)

        // Then: verify that saveData was called on the mock repository with the correct data
        verify(mockRepository).saveData(dataToStore)
    }
}
```



Esmaeel Moradi

@esmaeel-moradi700



In a complex system where different layers (UI, Logic, Data) interact, Integration Tests ensure that these layers work flawlessly together and data flows correctly. These tests are especially crucial in guaranteeing the integrity of data throughout sensitive and complex processes (such as blockchain transactions).



Esmaeeil Moradi
@esmaeeil-moradi700

UI Testing / End-to-End Testing: Verify the User Experience!



Description: UI Testing (sometimes called End-to-End Testing) simulates and tests how a real user interacts with the application's user interface and how the application responds to these interactions. These tests run on an Android emulator or a real device and verify the entire user flow from beginning to end.

Why is UI Testing essential?

- **Ensuring Correct User Experience:** Verifies that all UI elements are visible and interactive, and that the application behaves as the user expects.
- **Validating Core User Flows:** Tests whether critical scenarios (e.g., login, making a transaction, searching) work correctly and without flaws.
- **Identifying UI and State-Related Issues:** Bugs that might occur in complex UI interactions or application state changes are discovered at this level.

Tools:

- **Espresso:** Google's primary UI testing framework for Android applications based on Views (XML Layout).
- **Compose Test:** The UI testing framework for applications built with Jetpack Compose.
- **UI Automator:** A testing framework for testing user interactions with applications at a system level, even across multiple applications.



Esmaeel Moradi

@esmaeel-moradi700



- **Code Example: A Simple Espresso Test**

Suppose you have an Activity that changes the text of a TextView when a button is clicked.

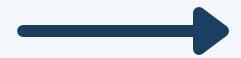
```
// Example: MyActivity.kt - Production code (Simplified)
// Assuming you have a layout with:
// <Button android:id="@+id/my_button" ... />
// <TextView android:id="@+id/my_text_view" ... />

// class MyActivity : AppCompatActivity() {
//     override fun onCreate(savedInstanceState: Bundle?) {
//         super.onCreate(savedInstanceState)
//         setContentView(R.layout.activity_main)
//         findViewById<Button>(R.id.my_button).setOnClickListener {
//             findViewById<TextView>(R.id.my_text_view).text = "New Text!"
//         }
//     }
// }
```



Esmaeel Moradi

@esmaeeil-moradi700



Now, the UI test with Espresso:

```
// Example: MyActivityTest (UI Test with Espresso) - In src/androidTest/java or sr
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.action.ViewActions.click
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.withId
import androidx.test.espresso.matcher.ViewMatchers.withText
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

// The test needs to run with AndroidJUnit4 test runner
@RunWith(AndroidJUnit4::class)
class MyActivityTest {

    // This rule launches the specified activity before each test method
    @get:Rule
    val activityRule = ActivityScenarioRule(MyActivity::class.java) // Replace MyA

    @Test
    fun `button click changes text`() {
        // 1. Perform a click on the button with ID 'my_button'
        onView(withId(R.id.my_button)).perform(click()) // Replace R.id.my_button

        // 2. Verify that the TextView with ID 'my_text_view' now displays the tex
        onView(withId(R.id.my_text_view)).check(matches(withText("New Text!"))) //

    }

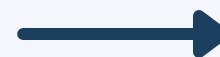
    @Test
    fun `initial text is correct`() {
        // Verify that the TextView initially displays "Hello World!" (or whatever
        onView(withId(R.id.my_text_view)).check(matches(withText("Hello World!")))

    }
}
```



Esmaeel Moradi

@esmaeeil-moradi700



In applications that deal with valuable user assets, such as cryptocurrency wallets or financial platforms, the user experience must be flawless and unambiguous. Even a small error in displaying a balance, a wallet address, or a transaction confirmation can erode user trust. UI Tests ensure that critical processes are performed exactly as the user expects, with full confidence, and that the user interface accurately reflects the system's state.



Esmaeail Moradi

@esmaeail-moradi700

The Test Pyramid: Test Coverage Strategy

Description:

The Test Pyramid is a strategic concept in software testing that suggests dividing tests into three layers based on speed, cost, and abstraction level:

- 1. Bottom (Most): Unit Tests:** The fastest, cheapest, and most numerous tests belong to this layer.
- 2. Middle: Integration Tests:** Fewer in number than Unit Tests, but slightly slower and more expensive.
- 3. Top (Least): UI/End-to-End Tests:** The slowest, most expensive, and fewest tests are in this layer.

Why the Test Pyramid? The goal of the Test Pyramid is to achieve an optimal balance between test coverage, execution speed, and maintenance cost. Focusing on fast and reliable unit tests allows you to identify bugs in the early stages and get quick feedback. As we move up to UI Tests, complexity and slowness increase, so their number should be fewer and used only to cover critical user flows.



Esmaeel Moradi

@esmaeel-moradi700

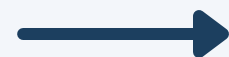


For building a scalable and stable system like an advanced Web3 system, adhering to the Test Pyramid is a fundamental principle. This approach ensures that we achieve the broadest coverage with the lowest cost and in the shortest possible time, guaranteeing the robustness of every layer of the application; from the core logic to the user interface that the end-user interacts with.



Esmaeeil Moradi

@esmaeeil-moradi700



Conclusion: Trust Your Code with Testing, and Build the Future!

Testing not only finds bugs but also helps improve code design, increase scalability, and reduce maintenance costs. It's an investment in your application's future that guarantees its stability and reliability.

By implementing these testing fundamentals, your application will not only improve in quality but also gain the trust of users and business partners. As an Android Architect with years of experience, I emphasize that testing is a vital tool for building critical and sensitive projects in the Web3 domain. This is just the beginning of the testing journey; follow me to dive into Advanced Testing Techniques on Day 28 and explore deep testing strategies for large-scale systems!



Esmaeeil Moradi
@esmaeeil-moradi700

Looking forward to your
insights and experiences!
Let's elevate Android
development together.



Esmaeel Moradi

@esmaeeil-moradi700