Test Driven Development (TDD) using JUnit for Android

Test-Driven Development (TDD) in Android using Kotlin with JUnit involves writing unit tests before implementing the actual functionality. The goal is to ensure that the code meets the desired behavior as defined by the tests. Here's how you can follow TDD with JUnit in Android using Kotlin:

**Steps for TDD in Android with Kotlin:**

1. **Write a Test (Red Phase)**: Write a failing test that defines a function or class behavior. The test will fail since the code for it doesn't exist yet.
2. **Write Code (Green Phase)**: Write the minimal code needed to make the test pass.
3. **Refactor (Refactor Phase)**: Clean up the code while ensuring the tests still pass.

**Example: TDD in Android with a Simple Calculator**

Let's implement a simple calculator that can add two numbers using TDD.

**1. Setup JUnit in Android (Gradle)**

To use JUnit in an Android project, add the following dependencies to your build.gradle:

gradle

Copy code

dependencies {

// JUnit for unit tests

testImplementation 'junit:junit:4.13.2'

}

**2. Write a Test (Red Phase)**

First, create a test class. You will create a failing test for a function that should add two numbers.

kotlin

Copy code

import org.junit.Assert.\*

import org.junit.Test

class CalculatorTest {

@Test

fun addition\_twoNumbers\_returnsSum() {

// Arrange

val calculator = Calculator()

// Act

val result = calculator.add(3, 4)

// Assert

assertEquals(7, result)

}

}

At this point, the test will fail because the Calculator class and the add function do not exist yet.

**3. Write Code (Green Phase)**

Next, implement the Calculator class and the add function to make the test pass.

kotlin

Copy code

class Calculator {

fun add(a: Int, b: Int): Int {

return a + b

}

}

Now, run the test again, and it should pass.

**4. Refactor (Refactor Phase)**

After making the test pass, you can look at your code to clean it up. In this case, there may not be much to refactor since the function is already simple, but in more complex cases, you would improve the design.

**5. Continue with More Tests**

You can now add more tests, for example, for subtraction, multiplication, or division, using the same TDD process.

**Example: Testing a Subtraction Function**

1. **Write the failing test**:

kotlin

Copy code

@Test

fun subtraction\_twoNumbers\_returnsDifference() {

val calculator = Calculator()

val result = calculator.subtract(10, 4)

assertEquals(6, result)

}

1. **Write the code to make the test pass**:

kotlin

Copy code

class Calculator {

fun add(a: Int, b: Int): Int {

return a + b

}

fun subtract(a: Int, b: Int): Int {

return a - b

}

}

**Running the Tests**

In Android Studio, you can run unit tests by navigating to **Run > Run 'Tests in ...'** or by right-clicking the test file and selecting **Run 'CalculatorTest'**.

**Key Points to Remember**

* **Small iterations**: Write small tests for each feature, implement them, and refactor.
* **Isolated tests**: Unit tests should not depend on Android SDK components like Activity or Context. If necessary, use mock objects.
* **JUnit annotations**: Utilize annotations like @Before to set up any common objects, and @After to clean up resources.

By practicing TDD, you'll have better test coverage, more reliable code, and easier refactoring opportunities in your Android projects using Kotlin and JUnit.

Writing and running JUnit tests

Top of Form

Let's go through another complete example of writing and running JUnit tests in Android using Kotlin, but this time we will build and test a **UserValidator** class that checks if a user's email and password are valid. This example will demonstrate TDD principles, focusing on user input validation.

**Example: Validating User Email and Password**

**Step 1: Setting Up Dependencies**

Make sure you have the necessary dependencies in your build.gradle file:

gradle

Copy code

dependencies {

// JUnit for unit tests

testImplementation 'junit:junit:4.13.2'

// Kotlin test libraries

testImplementation "org.jetbrains.kotlin:kotlin-test:1.8.0"

testImplementation "org.jetbrains.kotlin:kotlin-test-junit:1.8.0"

}

**Step 2: Writing a Test (Red Phase)**

In TDD, we first write a failing test. Let's test if the UserValidator class can validate email and password fields correctly.

Create a test class UserValidatorTest.kt under src/test/java.

kotlin

Copy code

import org.junit.Assert.\*

import org.junit.Test

class UserValidatorTest {

@Test

fun email\_withValidEmail\_returnsTrue() {

// Arrange

val validator = UserValidator()

// Act

val result = validator.isEmailValid("test@example.com")

// Assert

assertTrue(result)

}

@Test

fun email\_withInvalidEmail\_returnsFalse() {

val validator = UserValidator()

val result = validator.isEmailValid("invalid-email")

assertFalse(result)

}

@Test

fun password\_withValidPassword\_returnsTrue() {

val validator = UserValidator()

val result = validator.isPasswordValid("Password123!")

assertTrue(result)

}

@Test

fun password\_withInvalidPassword\_returnsFalse() {

val validator = UserValidator()

val result = validator.isPasswordValid("123")

assertFalse(result)

}

}

At this point, all tests will fail because the UserValidator class and the isEmailValid and isPasswordValid methods don't exist yet.

**Step 3: Implementing the Code (Green Phase)**

Now, we will implement the minimal code necessary to make these tests pass. Create the UserValidator class with the methods we need.

kotlin

Copy code

class UserValidator {

// Email should have basic validation for "name@domain"

fun isEmailValid(email: String): Boolean {

return email.contains("@") && email.contains(".")

}

// Password should be at least 8 characters long and contain letters and digits

fun isPasswordValid(password: String): Boolean {

return password.length >= 8 && password.any { it.isDigit() } && password.any { it.isLetter() }

}

}

This code should now pass all the tests you wrote in the UserValidatorTest class. The isEmailValid method checks for the presence of @ and . in the email, and the isPasswordValid method ensures that the password is at least 8 characters long and contains both letters and digits.

**Step 4: Running the Tests**

In Android Studio, you can run the unit tests by doing the following:

1. Right-click the test class UserValidatorTest.kt.
2. Select **Run 'UserValidatorTest'**.

Alternatively, go to the **Run** menu and select **Run Tests in ...**. If the tests pass, you're on the right track!

**Step 5: Refactoring (Optional Refactor Phase)**

Once your tests are passing, you can refactor the code to make it more readable or efficient if needed. In this case, the code is quite straightforward, but you might choose to make the email validation stricter or adjust password requirements.

For example, let's make the email validation stricter using a regular expression:

kotlin

Copy code

class UserValidator {

fun isEmailValid(email: String): Boolean {

// Basic email pattern

val emailRegex = "^[A-Za-z](.\*)([@])(.+)(\\.)(.+)"

return email.matches(Regex(emailRegex))

}

fun isPasswordValid(password: String): Boolean {

return password.length >= 8 && password.any { it.isDigit() } && password.any { it.isLetter() }

}

}

**Step 6: Add More Tests (Optional)**

If you want to follow TDD rigorously, keep adding more tests. For example:

* Test for special characters in the email.
* Test for specific password requirements like special characters or uppercase letters.

**Full Example in Context**

**UserValidator.kt**

kotlin

Copy code

class UserValidator {

// Validates if email is in the correct format.

fun isEmailValid(email: String): Boolean {

val emailRegex = "^[A-Za-z](.\*)([@])(.+)(\\.)(.+)"

return email.matches(Regex(emailRegex))

}

// Validates if the password is at least 8 characters, contains a letter and a digit.

fun isPasswordValid(password: String): Boolean {

return password.length >= 8 && password.any { it.isDigit() } && password.any { it.isLetter() }

}

}

**UserValidatorTest.kt**

kotlin

Copy code

import org.junit.Assert.\*

import org.junit.Test

class UserValidatorTest {

@Test

fun email\_withValidEmail\_returnsTrue() {

val validator = UserValidator()

val result = validator.isEmailValid("test@example.com")

assertTrue(result)

}

@Test

fun email\_withInvalidEmail\_returnsFalse() {

val validator = UserValidator()

val result = validator.isEmailValid("invalid-email")

assertFalse(result)

}

@Test

fun password\_withValidPassword\_returnsTrue() {

val validator = UserValidator()

val result = validator.isPasswordValid("Password123!")

assertTrue(result)

}

@Test

fun password\_withInvalidPassword\_returnsFalse() {

val validator = UserValidator()

val result = validator.isPasswordValid("123")

assertFalse(result)

}

}

**Conclusion**

In this example, we followed the TDD process to develop a simple UserValidator class that validates an email and a password. We wrote tests first, implemented the minimal code to pass the tests, and then ran the tests in Android Studio.

By following this process:

1. **Red**: You write failing tests.
2. **Green**: You implement code to pass the tests.
3. **Refactor**: You improve the code while keeping the tests passing.

This process ensures that your code is always tested, robust, and easy to maintain.

Test suites and assertions, Test-driven development workflow

In Android with Kotlin, Test-Driven Development (TDD) follows a structured approach with **test suites** and **assertions** to ensure that your code behaves as expected. Here’s a detailed overview of how to work with **test suites** and **assertions** within the context of the TDD workflow.

**1. Test-Driven Development (TDD) Workflow in Android with Kotlin**

TDD follows a repeating cycle of writing tests, implementing code, and then refactoring:

1. **Write a Test (Red Phase)**: Write a failing test that defines a desired behavior.
2. **Write Code (Green Phase)**: Write the minimum amount of code necessary to make the test pass.
3. **Refactor (Refactor Phase)**: Refactor the code for efficiency and readability while ensuring all tests pass.

**TDD Workflow Steps**

* **Start with a test**: You begin by writing a unit test for a new feature or functionality that doesn't yet exist.
* **Run the test**: The test should fail initially because the feature hasn’t been implemented.
* **Write the simplest code**: Write the minimal code required to pass the test.
* **Run the test again**: Ensure that the test passes with the newly written code.
* **Refactor**: Clean up and optimize the code while keeping all tests green (passing).

The goal is to ensure that every piece of functionality is verified by a corresponding test. Now, let's dive deeper into **test suites** and **assertions** in Kotlin and how they fit into the TDD workflow.

**2. Test Suites in Kotlin**

A **test suite** is a collection of related tests that are grouped together. In Android with Kotlin, test suites can help organize and run multiple tests efficiently. You can use JUnit to create test suites that group related test classes.

**Example: Test Suite for User Validation**

Suppose you have different validation methods to test (email, password, and username). You can organize these into a test suite.

**Setting Up a Test Suite**

To create a test suite in Kotlin, you can use JUnit's @RunWith and @Suite annotations.

kotlin

Copy code

import org.junit.runner.RunWith

import org.junit.runners.Suite

// Define a test suite that includes all related test classes

@RunWith(Suite::class)

@Suite.SuiteClasses(

UserValidatorTest::class,

EmailValidatorTest::class,

PasswordValidatorTest::class

)

class ValidationTestSuite

In this example:

* UserValidatorTest, EmailValidatorTest, and PasswordValidatorTest are test classes.
* ValidationTestSuite is the test suite that groups all of these related tests together.

You can then run the entire test suite to execute all the tests in one go, making sure that all validations (email, password, username) are tested collectively.

**3. Assertions in Kotlin**

**Assertions** are methods used in unit testing to verify that a specific condition is true. If the condition is false, the test fails.

JUnit provides several assertion methods, and Kotlin makes them concise. Common assertions in Kotlin include:

* assertTrue(condition) – Verifies that the condition is true.
* assertFalse(condition) – Verifies that the condition is false.
* assertEquals(expected, actual) – Verifies that two values are equal.
* assertNotEquals(unexpected, actual) – Verifies that two values are not equal.
* assertNull(value) – Verifies that a value is null.
* assertNotNull(value) – Verifies that a value is not null.
* assertThrows(exception, code) – Verifies that a specific exception is thrown.

**Example: Assertions in a User Validator**

Here's an example of how to use assertions in a test class:

kotlin

Copy code

import org.junit.Assert.\*

import org.junit.Test

class UserValidatorTest {

@Test

fun email\_withValidEmail\_returnsTrue() {

val validator = UserValidator()

val result = validator.isEmailValid("test@example.com")

// Assertion to verify result is true

assertTrue(result)

}

@Test

fun email\_withInvalidEmail\_returnsFalse() {

val validator = UserValidator()

val result = validator.isEmailValid("invalid-email")

// Assertion to verify result is false

assertFalse(result)

}

@Test

fun password\_withValidPassword\_returnsTrue() {

val validator = UserValidator()

val result = validator.isPasswordValid("Password123!")

// Assertion to verify result is true

assertTrue(result)

}

@Test

fun password\_withInvalidPassword\_returnsFalse() {

val validator = UserValidator()

val result = validator.isPasswordValid("123")

// Assertion to verify result is false

assertFalse(result)

}

@Test

fun password\_withShortPassword\_throwsException() {

val validator = UserValidator()

// Assertion to verify that an exception is thrown for a short password

assertThrows(IllegalArgumentException::class.java) {

validator.isPasswordValid("123")

}

}

}

**4. Test-Driven Development Example with Assertions and Test Suites**

Let’s build a more complete example using TDD for a **User Registration** use case, where a User class must be validated for:

* A valid email.
* A valid password.
* A non-empty username.

**4.1. Writing a Failing Test (Red Phase)**

kotlin

Copy code

import org.junit.Assert.\*

import org.junit.Test

class UserTest {

@Test

fun user\_withValidDetails\_returnsTrue() {

val user = User("test@example.com", "Password123!", "johnDoe")

assertTrue(user.isValid())

}

@Test

fun user\_withInvalidEmail\_returnsFalse() {

val user = User("invalid-email", "Password123!", "johnDoe")

assertFalse(user.isValid())

}

@Test

fun user\_withEmptyUsername\_returnsFalse() {

val user = User("test@example.com", "Password123!", "")

assertFalse(user.isValid())

}

@Test

fun user\_withInvalidPassword\_returnsFalse() {

val user = User("test@example.com", "123", "johnDoe")

assertFalse(user.isValid())

}

}

This test will initially fail because the User class and its isValid() method don’t exist yet.

**4.2. Writing the Minimum Code (Green Phase)**

kotlin

Copy code

class User(private val email: String, private val password: String, private val username: String) {

fun isValid(): Boolean {

return isEmailValid() && isPasswordValid() && isUsernameValid()

}

private fun isEmailValid(): Boolean {

val emailRegex = "^[A-Za-z](.\*)([@])(.+)(\\.)(.+)"

return email.matches(Regex(emailRegex))

}

private fun isPasswordValid(): Boolean {

return password.length >= 8 && password.any { it.isDigit() } && password.any { it.isLetter() }

}

private fun isUsernameValid(): Boolean {

return username.isNotEmpty()

}

}

This implementation should pass all the tests in the UserTest class.

**4.3. Refactoring (Refactor Phase)**

After the tests are passing, you can refactor the User class to improve code quality, such as moving validation logic to separate utility classes if needed.

**5. Running Tests in Android Studio**

You can run your tests and test suites in Android Studio as follows:

* **Right-click on the test class or method** and choose **Run 'Test'**.
* Use the **Run menu** and select **Run Tests in ...** to run all the tests.
* If you've set up a **test suite**, you can run the suite to execute multiple tests at once.

**6. Summary**

* **Test Suites**: Organize multiple related test classes.
* **Assertions**: Verify test conditions and ensure expected behavior.
* **TDD Workflow**: Follow the Red-Green-Refactor cycle to iteratively write tests, implement code, and refactor for optimal quality.
* **Running Tests**: Run unit tests and test suites in Android Studio to validate the functionality.

By adhering to the TDD workflow, using test suites, and employing assertions effectively, you can ensure the robustness and reliability of your Android apps written in Kotlin.

Let's integrate an **EmailValidatorTest** and a **PasswordValidatorTest** into the **UserValidatorTest suite** using JUnit in Kotlin. These separate validator classes will handle specific logic for email and password validation, and we will combine them into a test suite to run all tests collectively.

**1. UserValidator with Email and Password Validators**

We'll first create two validators: one for emails and one for passwords. The UserValidator class will combine these two validators to validate the user's email and password together.

**EmailValidator.kt**

kotlin

Copy code

class EmailValidator {

// Basic email validation using a regex

fun isEmailValid(email: String): Boolean {

val emailRegex = "^[A-Za-z](.\*)([@])(.+)(\\.)(.+)"

return email.matches(Regex(emailRegex))

}

}

**PasswordValidator.kt**

kotlin

Copy code

class PasswordValidator {

// Password must be at least 8 characters, contain at least one letter and one digit

fun isPasswordValid(password: String): Boolean {

return password.length >= 8 && password.any { it.isDigit() } && password.any { it.isLetter() }

}

}

**UserValidator.kt**

Now, the UserValidator class will use both EmailValidator and PasswordValidator:

kotlin

Copy code

class UserValidator(

private val emailValidator: EmailValidator = EmailValidator(),

private val passwordValidator: PasswordValidator = PasswordValidator()

) {

fun isUserValid(email: String, password: String): Boolean {

return emailValidator.isEmailValid(email) && passwordValidator.isPasswordValid(password)

}

}

**2. Writing Unit Tests for Each Validator**

**EmailValidatorTest.kt**

This test class will verify the behavior of EmailValidator.

kotlin

Copy code

import org.junit.Assert.\*

import org.junit.Test

class EmailValidatorTest {

@Test

fun email\_withValidEmail\_returnsTrue() {

val validator = EmailValidator()

val result = validator.isEmailValid("test@example.com")

assertTrue(result)

}

@Test

fun email\_withInvalidEmail\_returnsFalse() {

val validator = EmailValidator()

val result = validator.isEmailValid("invalid-email")

assertFalse(result)

}

@Test

fun email\_withMissingAtSymbol\_returnsFalse() {

val validator = EmailValidator()

val result = validator.isEmailValid("testexample.com")

assertFalse(result)

}

@Test

fun email\_withMissingDomain\_returnsFalse() {

val validator = EmailValidator()

val result = validator.isEmailValid("test@.com")

assertFalse(result)

}

}

**PasswordValidatorTest.kt**

This test class will verify the behavior of PasswordValidator.

kotlin

Copy code

import org.junit.Assert.\*

import org.junit.Test

class PasswordValidatorTest {

@Test

fun password\_withValidPassword\_returnsTrue() {

val validator = PasswordValidator()

val result = validator.isPasswordValid("Password123")

assertTrue(result)

}

@Test

fun password\_withShortPassword\_returnsFalse() {

val validator = PasswordValidator()

val result = validator.isPasswordValid("Pass1")

assertFalse(result)

}

@Test

fun password\_withNoDigits\_returnsFalse() {

val validator = PasswordValidator()

val result = validator.isPasswordValid("Password")

assertFalse(result)

}

@Test

fun password\_withNoLetters\_returnsFalse() {

val validator = PasswordValidator()

val result = validator.isPasswordValid("12345678")

assertFalse(result)

}

}

**3. Test Suite for User Validation**

We will now combine the EmailValidatorTest and PasswordValidatorTest into a **UserValidatorTestSuite** to run them together.

**UserValidatorTestSuite.kt**

kotlin

Copy code

import org.junit.runner.RunWith

import org.junit.runners.Suite

// Define a test suite that includes all validator test classes

@RunWith(Suite::class)

@Suite.SuiteClasses(

EmailValidatorTest::class,

PasswordValidatorTest::class,

UserValidatorTest::class

)

class UserValidatorTestSuite

**4. UserValidatorTest**

Finally, we will add tests for UserValidator, which uses both email and password validation logic.

kotlin

Copy code

import org.junit.Assert.\*

import org.junit.Test

class UserValidatorTest {

@Test

fun user\_withValidEmailAndPassword\_returnsTrue() {

val userValidator = UserValidator()

val result = userValidator.isUserValid("test@example.com", "Password123")

assertTrue(result)

}

@Test

fun user\_withInvalidEmail\_returnsFalse() {

val userValidator = UserValidator()

val result = userValidator.isUserValid("invalid-email", "Password123")

assertFalse(result)

}

@Test

fun user\_withInvalidPassword\_returnsFalse() {

val userValidator = UserValidator()

val result = userValidator.isUserValid("test@example.com", "123")

assertFalse(result)

}

@Test

fun user\_withInvalidEmailAndPassword\_returnsFalse() {

val userValidator = UserValidator()

val result = userValidator.isUserValid("invalid-email", "123")

assertFalse(result)

}

}

**5. Running the Test Suite**

You can run the entire test suite in Android Studio by:

1. Right-clicking on the UserValidatorTestSuite.kt file.
2. Selecting **Run 'UserValidatorTestSuite'**.

Alternatively, go to the **Run** menu and select **Run Tests in ...** to execute all the test cases in the suite.

**6. Summary of Key Components**

* **EmailValidator**: Validates if the email is correctly formatted.
* **PasswordValidator**: Validates if the password meets specific criteria (length, digits, letters).
* **UserValidator**: Combines both email and password validators to validate a user.
* **Test Suite**: The UserValidatorTestSuite groups the EmailValidatorTest, PasswordValidatorTest, and UserValidatorTest to run them together.

This structure ensures that individual validations are tested thoroughly and independently, while UserValidator brings them together for user validation.

Bottom of Form