Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

TESTING, DEBUGGING & DEPLOYMENT (WEEKS 19-21)

USING LOGCAT & DEBUGGING TOOLS IN ANDROID DEVELOPMENT

CHAPTER 1: INTRODUCTION TO DEBUGGING IN ANDROID

1.1 What is Debugging?

- **Debugging** is the process of identifying and fixing errors (bugs) in an Android application.
- Android Studio provides **various debugging tools** to inspect, analyze, and resolve issues in code.

✅ **Why is Debugging Important?**

✓ Helps identify **runtime errors** and crashes.

✓ Allows **real-time monitoring** of app logs.

✓ Improves **app stability and performance**.

✓ Enables **testing & troubleshooting network requests, UI interactions, and background processes**.

📌 **Example Use Cases:**

- A **banking app** needs debugging to ensure transactions don't fail.

- A **social media app** checks for crashes when users upload images.

CHAPTER 2: INTRODUCTION TO LOGCAT

2.1 What is Logcat?

🔹 **Logcat** is a debugging tool in **Android Studio** that **logs system messages, errors, and debug information** during app execution.

🔹 It helps developers track app behavior, identify bugs, and optimize performance.

✅ **Key Features of Logcat:**

✓ Displays **real-time logs** from the running app.

✓ Filters logs by **tag, level, or process ID**.

✓ Helps debug **network requests, crashes, and UI interactions**.

✓ Supports **searching and exporting logs** for analysis.

2.2 Opening Logcat in Android Studio

1️⃣ Open **Android Studio**.

2️⃣ Run the app on an **Emulator or Physical Device**.

3️⃣ Go to **View > Tool Windows > Logcat** (or press Alt + 6).

4️⃣ Select the device from the **device dropdown** at the top.

5️⃣ Select **log level (Verbose, Debug, Info, Warn, Error, Assert)**.

📌 **Example:**

- If your app crashes, Logcat will display the **exact error message and stack trace**.

CHAPTER 3: UNDERSTANDING LOGCAT MESSAGES

- Logcat messages are structured into **different priority levels**.

Log Level	Description	Usage Example
Verbose (V)	Shows all log messages (detailed output).	Logging all API responses.
Debug (D)	Used for debugging information.	Checking variable values during execution.
Info (I)	Shows general app behavior messages.	Displaying user interactions.
Warning (W)	Indicates potential issues but not errors.	Slow network warning.
Error (E)	Displays runtime errors and crashes.	NullPointerException handling.
Assert (A)	Used for assertions that should never happen.	Debugging critical failures.

✅ **Example Log Messages:**

Log.v("MyApp", "Verbose log: API response received")

Log.d("MyApp", "Debug log: Button clicked")

Log.i("MyApp", "Info log: User logged in")

Log.w("MyApp", "Warning log: Slow network detected")

Log.e("MyApp", "Error log: NullPointerException occurred")

📌 **Filtering Logs in Logcat:**

- Type tag:MyApp to show only logs from **MyApp**.
- Use log level:Error to display only **error messages**.

Chapter 4: Using Logcat to Debug Crashes

- **Logcat is essential for identifying app crashes.**
- The error message in Logcat provides:

✓ **Exception Type** (e.g., NullPointerException).
✓ **File and Line Number** where the error occurred.
✓ **Stack Trace** showing the method call hierarchy.

✅ **Example Logcat Output for a Crash:**

E/MyApp: java.lang.NullPointerException: Attempt to invoke virtual method 'java.lang.String getName()' on a null object reference

at com.example.myapp.MainActivity.onCreate(MainActivity.kt:24)

✅ **Fixing the Issue in Kotlin:**

```
val user: User? = null

Log.d("MyApp", "User Name: ${user?.name ?: "User is null"}") // Avoids NullPointerException
```

📌 **Best Practices for Crash Debugging:**
✓ **Use try-catch to handle exceptions.**
✓ **Check Logcat for E/ (Error) messages.**
✓ **Use breakpoints to inspect variables.**

Chapter 5: Debugging with Breakpoints in Android Studio

5.1 What is a Breakpoint?

- **Breakpoints** allow developers to **pause code execution** at a specific line to inspect variables, values, and flow.
- They help analyze **logical errors and unexpected behavior**.

✅ **Steps to Use Breakpoints:**

1️⃣ Open **Android Studio**.

2️⃣ Click on the **left margin** next to a code line to set a **breakpoint**.

3️⃣ Run the app in **Debug Mode** (Shift + F9).

4️⃣ The execution **pauses at the breakpoint,** allowing variable inspection.

✅ **Example: Debugging a Login Function**

```
fun loginUser(email: String, password: String) {
    val user = fetchUser(email) // Set a breakpoint here
    Log.d("Login", "User: ${user?.name}")
}
```

- **Pause execution at fetchUser(email) to check its value.**

📌 **Best Practices:**

✓ Set **breakpoints at function calls** to inspect variables.

✓ Use **Step Over (F8)** to execute the next line.

✓ Use **Step Into (F7)** to go inside function calls.

CHAPTER 6: DEBUGGING NETWORK REQUESTS USING OKHTTP & RETROFIT

6.1 Logging API Requests in Logcat

🔹 **Retrofit + OkHttp Logging Interceptor** helps debug network requests.

✅ **Step 1: Add Dependencies in build.gradle**

```
dependencies {
    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
```

```
    implementation 'com.squareup.okhttp3:logging-interceptor:4.9.1'
}
```

✅ **Step 2: Enable Logging in Retrofit**

```
val logging = HttpLoggingInterceptor()
logging.setLevel(HttpLoggingInterceptor.Level.BODY)

val client = OkHttpClient.Builder()
    .addInterceptor(logging)
    .build()

val retrofit = Retrofit.Builder()
    .baseUrl("https://api.example.com")
    .client(client)
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

✅ **Step 3: Check API Logs in Logcat**

```
D/OkHttp: --> GET https://api.example.com/users
D/OkHttp: {"id":1,"name":"John Doe"}
D/OkHttp: <-- 200 OK (345ms)
```

📌 **Benefits of Logging API Requests:**

✓ Easily debug **network failures and errors**.
✓ Check **response data, headers, and latency**.
✓ Validate **request payloads and responses**.

CHAPTER 7: DEBUGGING MEMORY LEAKS WITH ANDROID PROFILER

🔹 **Memory leaks occur when objects are not released, causing excessive memory usage.**

🔹 **Android Profiler** detects leaks and monitors CPU, RAM, and Network usage.

✅ **Steps to Use Android Profiler:**

1️⃣ Open **View > Tool Windows > Profiler**.

2️⃣ Select **Memory Tab** to check memory usage.

3️⃣ Click **Record Heap Dump** to detect memory leaks.

📌 **Example: Preventing Memory Leaks**

```
class MainActivity : AppCompatActivity() {
    private var someListener: SomeListener? = null

    override fun onDestroy() {
        super.onDestroy()
        someListener = null // Prevents memory leak
    }
}
```

📌 **Best Practices to Avoid Memory Leaks:**

✓ Use WeakReference for long-lived objects.

✓ Release listeners in onDestroy().

✓ Use LeakCanary library for detection.

Exercise: Test Your Understanding

- What are the different log levels in Logcat?
- How can breakpoints help debug logical errors?
- What is the use of OkHttp logging interceptor?
- What are memory leaks, and how can you detect them?
- Modify an API request to log detailed response data.

Conclusion

✅ Logcat is a powerful tool for monitoring logs and debugging crashes.
✅ Breakpoints help inspect variables and fix logical errors.
✅ Logging network requests with OkHttp improves API debugging.
✅ Android Profiler helps detect memory leaks and optimize app performance.

PROFILING APPS FOR PERFORMANCE OPTIMIZATION IN ANDROID

CHAPTER 1: INTRODUCTION TO PERFORMANCE OPTIMIZATION

1.1 Why Optimize Android Apps?

🔹 **Performance optimization** ensures Android apps run **smoothly, efficiently, and with minimal resource usage**.

🔹 Poorly optimized apps can lead to:

✓ **Slow UI** – Janky animations and laggy interactions.

✓ **High CPU & Memory Usage** – Increased battery drain.

✓ **Long Load Times** – Slow database and network operations.

✓ **App Freezes (ANR – Application Not Responding Errors)**.

✅ **Benefits of Performance Optimization**

✓ **Smooth UI & Better UX** – Faster screen transitions and smooth scrolling.

✓ **Lower Battery Usage** – Less CPU and memory consumption.

✓ **Reduced Crash Rate** – Fewer ANRs and app freezes.

✓ **Better Retention** – Users prefer fast, responsive apps.

📌 **Example Use Cases:**

- An **e-commerce app** improves checkout speed by optimizing network calls.
- A **social media app** improves scrolling performance by reducing image loading time.

CHAPTER 2: TOOLS FOR PROFILING ANDROID APPS

🔹 **Android Profiler** in **Android Studio** provides real-time monitoring of:

✓ **CPU Usage** – Detects slow and inefficient code.

✓ **Memory Usage** – Identifies memory leaks and high RAM consumption.

✓ **Battery Consumption** – Measures energy usage of the app.

✓ **Network Performance** – Analyzes API calls and data transfer.

✅ **How to Open Android Profiler?**

1️⃣ **Run your app** on an emulator or physical device.

2️⃣ **Go to View → Tool Windows → Profiler**.

3️⃣ Select the **CPU, Memory, or Network tabs** for analysis.

CHAPTER 3: CPU PROFILING – DETECTING PERFORMANCE BOTTLENECKS

3.1 Why Profile CPU Usage?

- 🔹 High CPU usage can **cause lag and battery drain**.
- 🔹 Profiling **identifies inefficient code execution**.

✅ **How to Start CPU Profiling?**

1️⃣ Open **Profiler** in Android Studio.

2️⃣ Click on the **CPU tab** → Select **Record CPU Activity**.

3️⃣ Perform actions in the app and stop recording.

4️⃣ Analyze **Main Thread & Background Threads** execution.

3.2 Optimizing CPU Performance

📌 **Example: Avoiding Heavy Computation on the Main Thread**

// ❌ BAD: Running heavy computation on the UI thread

```
fun heavyTask() {
  for (i in 1..1000000) {
    Log.d("Performance", "Processing $i")
  }
}
```

✅ **Solution: Move Work to Background Thread Using Coroutines**

```
// ✅ GOOD: Perform task in background using Coroutines
GlobalScope.launch(Dispatchers.Default) {
  for (i in 1..1000000) {
    Log.d("Performance", "Processing $i")
  }
}
```

📌 **Other Optimizations:**

✓ **Use background threads** (Coroutines or WorkManager) for heavy tasks.

✓ **Minimize UI rendering work** to reduce CPU load.

✓ **Use hardware acceleration** for better rendering performance.

CHAPTER 4: MEMORY PROFILING – PREVENTING MEMORY LEAKS

4.1 Why Monitor Memory Usage?

- Excessive memory usage can **lead to app crashes** due to OutOfMemory (OOM) errors.
- **Memory leaks** occur when objects **are not properly freed** after use.

✅ How to Start Memory Profiling?

1️⃣ Open **Profiler** in Android Studio.

2️⃣ Click on the **Memory tab** → Select **Record Memory Allocations**.

3️⃣ Perform actions in the app and stop recording.

4️⃣ Identify **memory leaks and unnecessary allocations**.

4.2 Detecting & Fixing Memory Leaks

📌 Example: Memory Leak Due to Unreleased Context

```
// ❌ BAD: Holding an Activity reference in a Singleton
object MemoryLeakExample {
    var context: Context? = null
}
```

✅ Solution: Use Application Context Instead

```
// ✅ GOOD: Avoid memory leak by using application context
object MemorySafeExample {
    lateinit var appContext: Context

    fun init(context: Context) {
        appContext = context.applicationContext
    }
}
```

📌 Other Memory Optimizations:

✓ **Use WeakReference** for long-lived objects.

✓ **Avoid Static Context References** in singletons.

✓ **Use ViewBinding Instead of findViewById** to prevent memory leaks.

✓ **Call recycle() on Bitmaps** to release memory manually.

Chapter 5: Battery Profiling – Reducing Power Consumption

5.1 Why Optimize Battery Usage?

🔹 Apps that frequently **wake up the CPU, use location services, or perform background tasks** consume high battery.

🔹 Battery-heavy apps get **flagged by Android's Battery Manager** and may be restricted.

✅ **How to Start Battery Profiling?**

1️⃣ Enable **Battery Historian** (adb shell dumpsys batterystats).

2️⃣ Monitor **Wake Locks** and excessive network activity.

3️⃣ Optimize background tasks to minimize wakeups.

5.2 Optimizing Battery Usage

📌 **Example: Avoiding Unnecessary Background Work**

```
// ❌ BAD: Running frequent background updates

val handler = Handler()

handler.postDelayed({ updateData() }, 1000) // Every second
```

✅ **Solution: Use WorkManager for Efficient Background Tasks**

```
// ✅ GOOD: Schedule work efficiently using WorkManager
```

```
val workRequest = PeriodicWorkRequestBuilder<MyWorker>(15, TimeUnit.MINUTES).build()

WorkManager.getInstance(context).enqueue(workRequest)
```

📌 **Other Battery Optimizations:**

✓ **Use WorkManager instead of background services.**

✓ **Reduce GPS and location updates to minimize power consumption.**

✓ **Batch network requests to minimize radio wakeups.**

Chapter 6: Network Profiling – Optimizing API Calls

6.1 Why Profile Network Usage?

🔹 **Slow or excessive network requests** can cause:

✓ **Slow app response times.**

✓ **Increased data usage for users.**

✓ **Higher server costs and crashes.**

✅ **How to Start Network Profiling?**

1️⃣ Open **Profiler** in Android Studio.

2️⃣ Click on the **Network tab** → Start recording.

3️⃣ Monitor API requests, response times, and data usage.

6.2 Optimizing Network Calls

📌 **Example: Avoiding Unnecessary API Calls**

```
// ❌ BAD: Fetching data every time the user opens the app

fun fetchData() {

   val url = "https://api.example.com/data"
```

```
    val request = Request.Builder().url(url).build()
    OkHttpClient().newCall(request).execute()
}
```

✅ **Solution: Cache Data Locally Using Room Database**

```
// ✅ GOOD: Store API response in Room Database for offline access
fun fetchData() {
    if (database.hasCachedData()) {
        return database.getCachedData()
    } else {
        val url = "https://api.example.com/data"
        val request = Request.Builder().url(url).build()
        OkHttpClient().newCall(request).execute()
    }
}
```

📌 **Other Network Optimizations:**

✓ **Use Retrofit for efficient API requests.**

✓ **Enable Gzip compression for reducing data size.**

✓ **Use OkHttp caching to avoid repeated requests.**

Exercise: Test Your Understanding

- **How does CPU profiling help improve performance?**
- **What are memory leaks, and how can you detect them?**
- **How can you reduce battery consumption in an Android app?**

- **Write a code snippet to optimize network requests using Retrofit.**
- **How can you prevent ANR (Application Not Responding) errors?**

Conclusion

✅ **Profiling tools help detect performance bottlenecks in Android apps.**
✅ **CPU profiling identifies slow code execution and main thread blocking.**
✅ **Memory profiling detects memory leaks and excessive allocations.**
✅ **Battery profiling helps reduce background tasks and GPS usage.**
✅ **Network profiling optimizes API calls and reduces data usage.**

JUnit & Espresso Testing in Android Development

Chapter 1: Introduction to Testing in Android

1.1 Why Testing is Important?

- Testing in Android ensures that the app functions correctly before deployment.
- It helps in **catching bugs early,** improving the **user experience,** and ensuring **code reliability**.

✅ **Benefits of Testing:**

✓ Prevents bugs and crashes.

✓ Ensures app stability across different devices.

✓ Saves time in debugging and maintenance.

✓ Supports continuous integration and deployment (CI/CD).

📌 **Example:**

- **A banking app requires thorough testing to avoid transaction failures.**

Chapter 2: Understanding JUnit for Unit Testing

2.1 What is JUnit?

- **JUnit** is a Java-based testing framework used for **unit testing** in Android.
- It is designed to **test individual functions, methods, and components** in isolation.

✅ Why Use JUnit?

✓ Tests small units of code independently.

✓ Fast execution and immediate feedback.

✓ Supports mocking and dependency injection.

📌 Example:

- **Testing a function that calculates tax on a given amount.**

2.2 Setting Up JUnit in Android

✅ Add JUnit dependencies in build.gradle (Module: app)

```
dependencies {
    testImplementation 'junit:junit:4.13.2'
}
```

✅ Create a Sample Calculator Class for Testing

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

    fun multiply(a: Int, b: Int): Int {
        return a * b
    }
}
```

✅ Write JUnit Test Cases in CalculatorTest.kt

```
import org.junit.Assert.assertEquals
import org.junit.Test

class CalculatorTest {

    private val calculator = Calculator()

    @Test
    fun testAddition() {
        val result = calculator.add(5, 3)
        assertEquals(8, result)
    }

    @Test
    fun testMultiplication() {
        val result = calculator.multiply(4, 2)
        assertEquals(8, result)
    }
}
```

✅ **Run the test cases:**

- Open **Run > Run 'Tests in CalculatorTest'**.
- Green checkmarks indicate successful tests.

📌 **Example:**

- **A shopping cart function is tested using JUnit to verify correct total calculations.**

CHAPTER 3: INTRODUCTION TO ESPRESSO FOR UI TESTING

3.1 What is Espresso?

🔹 **Espresso** is a UI testing framework provided by **Android Jetpack** to automate UI interactions.

🔹 It allows developers to simulate **user actions like clicks, typing, and scrolling**.

✅ **Why Use Espresso?**

✓ Tests UI components automatically.

✓ Ensures app UI behaves correctly.

✓ Detects UI bugs before release.

📌 **Example:**

- **A login screen test that verifies if the correct error message is shown for invalid credentials.**

3.2 Setting Up Espresso in Android

✅ **Add Espresso dependencies in build.gradle (Module: app)**

androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'

androidTestImplementation 'androidx.test.ext:junit:1.1.3'

androidTestImplementation 'androidx.test:runner:1.4.0'

✅ **Enable Java Instrumentation Tests**

- Modify **defaultConfig** inside build.gradle

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

CHAPTER 4: WRITING AN ESPRESSO TEST CASE

✅ Create a Simple Login Activity (activity_login.xml)

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:padding="16dp">

  <EditText
    android:id="@+id/username"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Username"/>

  <EditText
    android:id="@+id/password"
```

```
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Password"
    android:inputType="textPassword"/>

  <Button
    android:id="@+id/loginButton"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Login"/>
</LinearLayout>
```

✅ **Write Espresso Test Case (LoginActivityTest.kt)**

```
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 org.junit.Rule
import org.junit.Test

class LoginActivityTest {
```

```
@get:Rule
val activityRule = ActivityScenarioRule(LoginActivity::class.java)

@Test
fun testLoginButtonClick() {
    onView(withId(R.id.username)).perform(typeText("user123"), closeSoftKeyboard())
    onView(withId(R.id.password)).perform(typeText("password"), closeSoftKeyboard())
    onView(withId(R.id.loginButton)).perform(click())
}
}
```

✅ **Run the test:**

- Open **Run > Run 'Tests in LoginActivityTest'**.
- The test will simulate **typing username, password, and clicking the login button**.

📌 **Example:**

- **Testing a signup form to verify if the user gets redirected to the home screen after registration.**

CHAPTER 5: ADVANCED ESPRESSO TESTING

5.1 Testing UI Assertions

✅ **Check if a Button is Displayed:**

```
onView(withId(R.id.loginButton)).check(matches(isDisplayed()))
```

✅ **Verify Text in a TextView:**

```
onView(withId(R.id.username)).check(matches(withText("user123")))
```

✅ **Perform Scroll Actions:**

```
onView(withId(R.id.scrollView)).perform(scrollTo())
```

5.2 Handling UI Delays in Espresso

🔹 Use IdlingResource when UI changes take time, like loading API data.

```
IdlingRegistry.getInstance().register(myIdlingResource)
```

📌 **Example:**

- **Waiting for an API response before validating UI elements.**

Exercise: Test Your Understanding

🔹 **What is the difference between JUnit and Espresso?**
🔹 **How do you test UI elements using Espresso?**
🔹 **Modify the login test to verify if an error message is displayed for empty fields.**
🔹 **How can you mock API responses for UI testing?**
🔹 **Write a test case for checking if a RecyclerView scrolls correctly.**

Conclusion

✅ **JUnit is used for unit testing individual functions, while Espresso is used for UI testing in Android.**

✅ **JUnit helps ensure logic correctness, while Espresso verifies UI behavior.**

✅ **Automated tests improve app stability and user experience.**

✅ **Testing is crucial for preventing app crashes and ensuring smooth functionality.**

TEST-DRIVEN DEVELOPMENT (TDD) APPROACH IN ANDROID

CHAPTER 1: INTRODUCTION TO TEST-DRIVEN DEVELOPMENT (TDD)

1.1 What is Test-Driven Development (TDD)?

🔹 **Test-Driven Development (TDD)** is a software development approach where **tests are written before the actual implementation of the code**.

🔹 The goal is to **ensure code correctness, improve design, and reduce bugs**.

✅ **Key Principles of TDD:**

✓ **Write a failing test first** before writing the actual code.

✓ **Write the minimal code** to pass the test.

✓ **Refactor the code** for optimization without breaking the test.

📌 **Example:**

- Before implementing a UserLogin function, we first write a test case that expects a correct username and password to return true.

1.2 The TDD Cycle: Red-Green-Refactor

🔹 TDD follows an **iterative process** called the **Red-Green-Refactor cycle**:

Phase	Description
Red (Write Test)	Write a unit test that fails because the code does not yet exist.

Green (Write Code)	Write the minimum amount of code to make the test pass.
Refactor (Optimize Code)	Refactor the code for efficiency while ensuring tests still pass.

📌 **Example:**

1. Write a test that expects Login() to return true for correct credentials (Red - Test Fails).
2. Implement a basic Login() function that returns true for hardcoded credentials (Green - Test Passes).
3. Refactor the function to handle real user input and improve security (Refactor).

CHAPTER 2: SETTING UP TDD IN AN ANDROID PROJECT

2.1 Types of Testing in Android

✅ **Unit Testing:**

- Tests individual components (e.g., functions, classes).
- Uses **JUnit and Mockito**.

✅ **Integration Testing:**

- Tests interactions between components (e.g., API + Database).
- Uses **Retrofit, Room, and MockWebServer**.

✅ **UI Testing:**

- Tests user interface and user interactions.
- Uses **Espresso for UI automation**.

2.2 Adding Testing Dependencies to build.gradle

Open **build.gradle (Module Level)** and add:

```
dependencies {
    // JUnit for Unit Testing
    testImplementation 'junit:junit:4.13.2'

    // Mockito for Mocking Dependencies
    testImplementation 'org.mockito:mockito-core:3.11.2'

    // AndroidX Test for UI Testing
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

📌 **Explanation:**

- **JUnit** for writing test cases.
- **Mockito** for mocking dependencies.
- **Espresso** for UI testing.

CHAPTER 3: IMPLEMENTING TDD WITH JUNIT AND MOCKITO

3.1 Writing a Unit Test Before Implementing Code

🔹 **Use Case:** Implement a LoginManager class that validates user credentials.

✅ Step 1: Write a Unit Test (Red Phase - Test Fails)

📌 LoginManagerTest.java

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class LoginManagerTest {
    private LoginManager loginManager;

    @Before
    public void setup() {
        loginManager = new LoginManager();
    }

    @Test
    public void
testLogin_WithCorrectCredentials_ShouldReturnTrue() {
        boolean result = loginManager.login("admin", "password123");
        assertTrue(result); // Expected: true
    }

    @Test
```

```
    public void
testLogin_WithIncorrectCredentials_ShouldReturnFalse() {
        boolean result = loginManager.login("admin",
"wrongpassword");
        assertFalse(result); // Expected: false
    }
}
```

📌 **Explanation:**

- setup() initializes LoginManager.
- testLogin_WithCorrectCredentials() expects true for correct credentials.
- testLogin_WithIncorrectCredentials() expects false for wrong credentials.
- Running this test **fails** because LoginManager is not implemented yet.

3.2 Implement the Minimal Code to Pass the Test (Green Phase - Test Passes)

📌 **LoginManager.java**

```
public class LoginManager {
    public boolean login(String username, String password) {
        return username.equals("admin") &&
password.equals("password123");
    }
```

```
}
```

📌 **Explanation:**

- **Hardcoded credentials** make the test pass.
- Running the test now **passes both cases**.

3.3 Refactor the Code (Refactor Phase - Optimize Code)

🔹 Instead of hardcoding credentials, use a **UserRepository** to fetch user data dynamically.

📌 **Refactored LoginManager.java**

```
public class LoginManager {
  private UserRepository userRepository;

  public LoginManager(UserRepository userRepository) {
    this.userRepository = userRepository;
  }

  public boolean login(String username, String password) {
    return userRepository.isValidUser(username, password);
  }
}
```

📌 **Explanation:**

- The class now **depends on a repository** instead of hardcoded values.

- Tests still **pass** without changing the test cases.

CHAPTER 4: USING MOCKITO TO MOCK DEPENDENCIES

4.1 What is Mockito?

🔹 **Mockito** is a **mocking framework** used to create **fake objects** for testing.

🔹 It helps in testing **components that rely on external dependencies** like databases or APIs.

✅ **Why Use Mockito?**

✓ Avoids **real database/API calls** during testing.

✓ Simulates **real-world scenarios** (e.g., failed login attempts).

✓ Improves **test speed and reliability**.

4.2 Mocking UserRepository in LoginManagerTest

📌 **Modified LoginManagerTest.java using Mockito**

```
import org.junit.Before;

import org.junit.Test;

import static org.mockito.Mockito.*;

import static org.junit.Assert.*;


public class LoginManagerTest {

   private LoginManager loginManager;

   private UserRepository mockUserRepository;
```

```
    @Before
    public void setup() {
        mockUserRepository = mock(UserRepository.class);
        loginManager = new LoginManager(mockUserRepository);
    }

    @Test
    public void
testLogin_WithCorrectCredentials_ShouldReturnTrue() {
        when(mockUserRepository.isValidUser("admin",
"password123")).thenReturn(true);
        assertTrue(loginManager.login("admin", "password123"));
    }

    @Test
    public void
testLogin_WithIncorrectCredentials_ShouldReturnFalse() {
        when(mockUserRepository.isValidUser("admin",
"wrongpassword")).thenReturn(false);
        assertFalse(loginManager.login("admin", "wrongpassword"));
    }
}
```

📌 **Explanation:**

- **Mockito's mock(UserRepository.class)** creates a **fake UserRepository**.
- **when().thenReturn()** simulates **different login responses**.
- Tests **pass** without calling a real database.

CHAPTER 5: WRITING UI TESTS USING ESPRESSO

5.1 What is Espresso?

🔹 **Espresso** is an Android UI testing framework for **validating UI interactions**.

🔹 It ensures **buttons, text fields, and UI elements behave as expected**.

✅ **Example: Testing Login Button Click**

```
import androidx.test.ext.junit.runners.AndroidJUnit4;

import androidx.test.rule.ActivityTestRule;

import org.junit.Rule;

import org.junit.Test;

import org.junit.runner.RunWith;

import static androidx.test.espresso.Espresso.onView;

import static androidx.test.espresso.action.ViewActions.*;

import static androidx.test.espresso.matcher.ViewMatchers.*;


@RunWith(AndroidJUnit4.class)

public class LoginActivityTest {

  @Rule
```

```
  public ActivityTestRule<LoginActivity> activityRule = new
ActivityTestRule<>(LoginActivity.class);


  @Test
  public void testLoginButtonClick() {
    onView(withId(R.id.username)).perform(typeText("admin"));

onView(withId(R.id.password)).perform(typeText("password123"));
    onView(withId(R.id.loginButton)).perform(click());
  }
}
```

📌 **Explanation:**

- Uses **Espresso** to **type text and click buttons** in UI tests.

🚀 **Conclusion**

✅ **TDD improves code reliability by writing tests first.**
✅ **JUnit + Mockito help test logic without external dependencies.**
✅ **Espresso ensures UI elements work as expected.**
✅ **Following TDD prevents bugs and speeds up debugging.**

GENERATING APKS & APP BUNDLES IN ANDROID DEVELOPMENT

CHAPTER 1: INTRODUCTION TO APKS & APP BUNDLES

1.1 What is an APK?

🔹 **APK (Android Package Kit)** is the **executable file format** for Android applications.

🔹 It contains **compiled code, resources, assets, and dependencies** required to run an app on an Android device.

✅ **Why Generate an APK?**

✓ Used for **installing apps manually (sideloading)**.

✓ Useful for **testing** apps before release.

✓ Required for **publishing on Google Play Store**.

📌 **Example:**

- A developer creates an APK to test the app on different devices before deployment.

1.2 What is an App Bundle (.AAB)?

🔹 **App Bundle (AAB)** is the modern publishing format used by **Google Play**.

🔹 Instead of one universal APK, AAB contains **optimized APKs** for different devices.

✅ **Benefits of App Bundles:**

✓ **Smaller app size** – Only necessary files are installed.

✓ **Automatic device-specific optimization** – Google Play delivers

the best APK version for each user.

✓ **Efficient updates** – Uses **Google Play Dynamic Delivery**.

📌 **Example:**

- An **AAB file reduces app size** by delivering only the required language, screen size, and CPU architecture.

CHAPTER 2: GENERATING A SIGNED APK FOR RELEASE

2.1 What is a Signed APK?

🔹 **A Signed APK** is an APK that has been **digitally signed** with a developer's **keystore** to verify its authenticity.

🔹 Google Play requires apps to be **signed before publishing**.

✅ **Why Sign an APK?**

✓ Ensures **app integrity** and prevents tampering.

✓ Allows **secure updates** (only the signed app can be updated).

✓ Mandatory for **Play Store submission**.

2.2 Steps to Generate a Signed APK

✅ **Step 1: Open Android Studio**

- Click **Build > Generate Signed Bundle/APK**.
- Choose **APK** and click **Next**.

✅ **Step 2: Create or Use an Existing Keystore**

- Click **Create New** (if you don't have a keystore).
- Enter a **password and key alias**.
- Choose a **storage location** for the keystore file.

✅ **Step 3: Configure APK Signature**

- Select **release** mode.
- Enable **V1 (Jar Signature) and V2 (Full APK Signature)**.

✅ **Step 4: Build the Signed APK**

- Click **Finish,** and Android Studio will generate the **signed APK**.
- The APK will be saved in **app/release/** folder.

📌 **Example:**

- Developers use signed APKs to **distribute apps securely** on the Play Store.

CHAPTER 3: GENERATING AN ANDROID APP BUNDLE (AAB)

3.1 Steps to Generate an App Bundle (.AAB)

✅ **Step 1: Open Android Studio**

- Click **Build > Generate Signed Bundle/APK**.
- Choose **Android App Bundle** and click **Next**.

✅ **Step 2: Select Keystore and Signature**

- Choose **an existing keystore** or **create a new one**.
- Enter the **keystore password**.

✅ **Step 3: Build the App Bundle**

- Click **Finish,** and Android Studio will generate the **.aab file**.
- The AAB file will be stored in the **app/release/** folder.

📌 **Example:**

- Developers upload an **AAB file to the Play Store,** and Google Play generates optimized APKs for different devices.

CHAPTER 4: DEBUG APK VS. RELEASE APK

Feature	Debug APK	Release APK
Used for	Development & testing	Publishing & distribution
Optimized?	No	Yes
Signed?	No (Unsigned)	Yes (Signed with a keystore)
Logging	Includes debugging logs	Logs removed for performance
Play Store Upload?	❌ No	✅ Yes

📌 **Example:**

- **A Debug APK is used by developers** for testing, while a **Release APK is used for publishing**.

CHAPTER 5: OPTIMIZING APK SIZE BEFORE DEPLOYMENT

5.1 Enable ProGuard to Remove Unused Code

✅ **Modify proguard-rules.pro to enable shrinking:**

```
-dontwarn okhttp3.**

-keep class retrofit2.** { *; }
```

✅ **Enable ProGuard in build.gradle (Module: app)**

```
buildTypes {
   release {
      minifyEnabled true
      shrinkResources true
      proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
   }
}
```

🔹 **ProGuard reduces APK size by removing unused methods and classes.**

5.2 Convert Images to WebP for Smaller APK Size

✅ **Convert PNG/JPEG to WebP:**

- Right-click an image in res/drawable.
- Click **Convert to WebP**.
- Select **Lossless Compression**.

🔹 **WebP images reduce APK size without losing quality.**

5.3 Use Vector Drawables Instead of PNGs

✅ **Replace PNG icons with Vector Drawables:**

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
   android:width="24dp"
```

```
    android:height="24dp"
    android:viewportWidth="24"
    android:viewportHeight="24">
    <path android:fillColor="#000000"
        android:pathData="M12,2L15,8H9L12,2Z"/>
</vector>
```

- **Vector Drawables scale without increasing APK size.**

CHAPTER 6: TESTING THE APK BEFORE DEPLOYMENT

6.1 Test on Multiple Devices (Emulators & Real Devices)

✅ **Use different screen sizes and OS versions to check UI compatibility.**

6.2 Run Performance Tests

✅ **Enable Developer Options > Profile GPU Rendering** to analyze UI rendering speed.

6.3 Perform Unit Tests Using JUnit

```
import org.junit.Assert.assertEquals
import org.junit.Test

class MathUtilsTest {
    @Test
    fun testAddition() {
        assertEquals(10, 5 + 5)
```

```
    }
}
```

✅ **Run the test in Android Studio:**

- Open **Run > Run 'Tests in MathUtilsTest'**.

6.4 Perform UI Testing Using Espresso

```
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.matcher.ViewMatchers.*
import androidx.test.espresso.action.ViewActions.*
import org.junit.Rule
import org.junit.Test

class LoginActivityTest {
    @get:Rule
    val activityRule = ActivityScenarioRule(LoginActivity::class.java)

    @Test
    fun testLoginButtonClick() {
        onView(withId(R.id.username)).perform(typeText("user"),
closeSoftKeyboard())
        onView(withId(R.id.password)).perform(typeText("password"),
closeSoftKeyboard())
        onView(withId(R.id.loginButton)).perform(click())
    }
```

}

✅ **Run the UI test:**

- Open **Run > Run 'Tests in LoginActivityTest'**.

📌 **Example:**

- A **fully tested APK ensures better performance and fewer crashes** after deployment.

CHAPTER 7: PUBLISHING THE APP ON GOOGLE PLAY STORE

7.1 Uploading an App Bundle to Google Play Console

✅ **Step 1: Create a Developer Account**

- Visit Google Play Console.
- Pay the **one-time registration fee ($25 USD)**.

✅ **Step 2: Create a New App Listing**

- Click **Create App** and enter the **app name, language, and category**.

✅ **Step 3: Upload the AAB File**

- Navigate to **Release Management > App Releases**.
- Click **Upload**, select the **AAB file**, and submit.

✅ **Step 4: Complete Store Listing**

- Add **screenshots, descriptions, and app permissions**.

✅ **Step 5: Publish the App**

- Click **Review & Publish**.
- Google Play will **review the app (takes a few hours to days).**

📌 Conclusion

✅ APKs are used for manual installation, while AABs are for Play Store publishing.
✅ ProGuard, WebP, and Vector Drawables reduce APK size.
✅ Thorough testing ensures a bug-free, optimized release.
✅ Google Play Console manages app distribution.

GOOGLE PLAY CONSOLE & APP MONETIZATION

CHAPTER 1: INTRODUCTION TO GOOGLE PLAY CONSOLE

1.1 What is Google Play Console?

🔹 **Google Play Console** is a **developer platform** that allows Android developers to **publish, manage, and monitor** their apps on the Google Play Store.

🔹 It provides tools to:

✓ Upload and distribute Android apps.

✓ Monitor app performance, crashes, and ANR (Application Not Responding) errors.

✓ Manage in-app purchases and monetization strategies.

✓ Optimize app visibility with store listings and analytics.

✅ **Why Use Google Play Console?**

✓ **Global Distribution** – Reach millions of users worldwide.

✓ **Performance Analytics** – Track installs, retention, and user engagement.

✓ **Monetization Tools** – Earn revenue through ads, in-app purchases, and subscriptions.

✓ **Security & Policy Compliance** – Ensure apps meet Google's guidelines.

📌 **Example:**

- A **game developer** publishes an Android game and uses Google Play Console to **track downloads and revenue**.

Chapter 2: Setting Up Google Play Console

2.1 Creating a Developer Account

✅ Step 1: Sign Up for a Google Play Developer Account

1. Visit Google Play Console.
2. Sign in with a Google account.
3. Pay the **one-time registration fee of $25**.
4. Fill in developer details (name, contact email, website, etc.).

✅ Step 2: Complete App Store Listing Information

1. **App Name & Description** – Provide an engaging app title and detailed description.
2. **Category & Tags** – Select the right category (e.g., Games, Productivity).
3. **Screenshots & Videos** – Upload high-quality app images and a promotional video.
4. **Privacy Policy** – Link to an online privacy policy page.

Chapter 3: Uploading an Android App to Google Play Store

✅ Step 1: Prepare Your APK or AAB File

- Generate a **signed APK (.apk) or Android App Bundle (.aab)** file in Android Studio.

✅ Step 2: Upload the App to Google Play Console

1. Open Google Play Console.
2. Click **"Create App"** and enter app details.
3. Upload the **signed APK or AAB file** in the "Production" section.
4. Complete **Content Rating** and **Target Audience** settings.
5. Set **Pricing & Distribution** (Free or Paid app).
6. Submit for **Google Play Review** (approval takes 2-3 days).

📌 **Example:**

- A **fitness app developer** uploads a signed .aab file to distribute the app globally.

CHAPTER 4: APP MONETIZATION STRATEGIES

4.1 What is App Monetization?

🔹 **App monetization** refers to earning revenue from mobile applications through various methods such as **ads, in-app purchases, and subscriptions**.

✅ **Why Monetize Your App?**

✓ Earn revenue while providing free content to users.

✓ Sustain app development and maintenance.

✓ Increase profitability through premium features.

4.2 App Monetization Models

🔹 **There are multiple ways to monetize an Android app:**

Monetization Model	Description	Example
Freemium	Free app with optional paid features (in-app purchases).	Spotify (free music with ads, premium version available).
In-App Purchases	Users buy virtual items, subscriptions, or premium content.	PUBG (buy skins, weapons, and in-game currency).

Ad-Based Monetization	Display ads using Google AdMob or other ad networks.	Free games with banner and video ads.
Subscription Model	Users pay a recurring fee (monthly/yearly) for premium access.	Netflix, YouTube Premium.
Paid Apps	Users pay upfront to download the app.	Productivity apps like Tasker.

📌 **Example:**

- A **news app** offers **free articles with ads** and a **premium subscription** for an ad-free experience.

CHAPTER 5: IMPLEMENTING ADS WITH GOOGLE ADMOB

🔹 **Google AdMob** is a mobile ad platform that allows developers to earn revenue by displaying ads in their apps.

✅ Step 1: Sign Up for AdMob

1️⃣ Go to Google AdMob and sign in.

2️⃣ Register your app and get an **App ID**.

✅ Step 2: Add AdMob Dependency to build.gradle

```
dependencies {
    implementation 'com.google.android.gms:play-services-ads:21.0.1'
}
```

✅ Step 3: Initialize AdMob SDK in MainActivity.kt

```
import com.google.android.gms.ads.MobileAds
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    MobileAds.initialize(this) {}
}
```

✅ **Step 4: Display Banner Ads**

📌 **Add AdView to activity_main.xml**

```
<com.google.android.gms.ads.AdView
    android:id="@+id/adView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:adSize="BANNER"
    app:adUnitId="ca-app-pub-xxxxxxxxxxxxxxxx/banner_ad_unit"/>
```

📌 **Load Ads in MainActivity.kt**

```
import com.google.android.gms.ads.AdRequest

val adRequest = AdRequest.Builder().build()
findViewById<AdView>(R.id.adView).loadAd(adRequest)
```

📌 **Example:**

- A **free puzzle game** earns revenue by displaying **banner and rewarded ads**.

CHAPTER 6: IMPLEMENTING IN-APP PURCHASES (IAP)

🔹 **Google Play Billing** allows apps to sell digital content like premium features, subscriptions, and virtual goods.

✅ Step 1: Add Google Play Billing Dependency

```
dependencies {
    implementation "com.android.billingclient:billing-ktx:5.0.0"
}
```

✅ Step 2: Initialize BillingClient in MainActivity.kt

```
import com.android.billingclient.api.*

private lateinit var billingClient: BillingClient

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    billingClient = BillingClient.newBuilder(this)
        .setListener { billingResult, purchases ->
            if (billingResult.responseCode ==
BillingClient.BillingResponseCode.OK && purchases != null) {
```

```
            for (purchase in purchases) {
                handlePurchase(purchase)
            }
        }
    }
    .enablePendingPurchases()
    .build()
}
```

✅ Step 3: Handle User Purchase

```
fun handlePurchase(purchase: Purchase) {
  if (purchase.purchaseState == Purchase.PurchaseState.PURCHASED) {
    Toast.makeText(this, "Purchase Successful!", Toast.LENGTH_SHORT).show()
  }
}
```

📌 Example:

- A **photo editing app** unlocks **premium filters and effects** after an in-app purchase.

Exercise: Test Your Understanding

- **How do you create a Google Play Developer account?**
- **What are the different monetization models for mobile apps?**
- **Write a code snippet to display a banner ad using Google**

AdMob.

- How can you integrate Google Play Billing for in-app purchases?
- What are the advantages of using a subscription model over ad-based monetization?

Conclusion

✅ Google Play Console enables developers to publish, manage, and monetize Android apps.

✅ Monetization strategies include ads, in-app purchases, freemium models, and subscriptions.

✅ Google AdMob is a powerful tool for ad-based revenue, while Google Play Billing enables digital purchases.

✅ Optimizing app monetization improves user experience and increases revenue potential.

ASSIGNMENT:

OPTIMIZE AND TEST AN APP BEFORE DEPLOYMENT

📌 Step-by-Step Assignment Solution: Optimize and Test an Android App Before Deployment

📌 Step 1: Optimize the Code for Performance

1.1 Optimize RecyclerView with ViewHolder Pattern

🔹 Improper use of RecyclerView can lead to memory leaks and poor performance.

✅ **Before Optimization (Inefficient View Binding)**

```
override fun onBindViewHolder(holder: MyViewHolder, position: Int)
{
   holder.itemView.findViewById<TextView>(R.id.textView).text =
dataList[position]
}
```

🔻 **Problem:** Repeated findViewById() calls reduce performance.

✅ **After Optimization (Using ViewHolder Pattern with ViewBinding)**

```
class MyViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
   val textView: TextView = itemView.findViewById(R.id.textView)
}
```

🔹 **ViewHolder reduces findViewById() calls, improving performance.**

1.2 Reduce Unnecessary Layout Nesting

🔹 **Deeply nested layouts** slow down rendering.

✅ **Before Optimization (Nested LinearLayouts)**

```
<LinearLayout>
  <LinearLayout>
    <TextView />
  </LinearLayout>
</LinearLayout>
```

🔻 **Problem:** Overuse of LinearLayouts increases rendering time.

✅ **After Optimization (Using ConstraintLayout)**

```
<ConstraintLayout>
  <TextView app:layout_constraintTop_toTopOf="parent"/>
</ConstraintLayout>
```

🔹 **ConstraintLayout improves UI rendering speed.**

📌 Step 2: Reduce App Size and Optimize Resources

2.1 Enable ProGuard to Shrink Code

✅ **Modify proguard-rules.pro to remove unused code**

```
-dontwarn okhttp3.**
-dontwarn retrofit2.**
-keep class retrofit2.** { *; }
```

✅ **Enable ProGuard in build.gradle (Module: app)**

```
buildTypes {
    release {
        minifyEnabled true
        shrinkResources true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
```

- **ProGuard removes unused code and reduces APK size.**

2.2 Compress Images for Optimization

- Use **WebP** instead of PNG/JPEG for better compression.

✅ **Convert images to WebP format in Android Studio:**

1. Right-click on an image in res/drawable.
2. Click **Convert to WebP**.
3. Select **Lossless Compression** to preserve quality.

- **WebP images reduce APK size without affecting quality.**

📌 Step 3: Identify and Fix Memory Leaks

3.1 Use LeakCanary for Memory Leak Detection

✅ **Add LeakCanary dependency in build.gradle (Module: app)**

```
dependencies {
```

```
    debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.9.1'
}
```

✅ **LeakCanary automatically detects memory leaks and logs them.**

📌 **Example Leak Fix:**

🔻 **Problem: Holding references in an activity**

```
class MyActivity : AppCompatActivity() {
    var myListener: MyListener? = null
}
```

✅ **Fix: Use WeakReference instead of strong references**

```
class MyActivity : AppCompatActivity() {
    var myListener: WeakReference<MyListener>? = null
}
```

🔹 **Avoid memory leaks by removing references when the activity is destroyed.**

📌 **Step 4: Improve App Security**

4.1 Secure API Keys with local.properties

🔹 **Never hardcode API keys in the app.**

✅ **Store API keys in local.properties**

```
API_KEY="your_secret_api_key"
```

✅ **Access API key securely in build.gradle**

```
buildConfigField "String", "API_KEY", project.properties["API_KEY"]
```

- **Prevents API key exposure in version control.**

📌 Step 5: Perform Unit Testing with JUnit

5.1 Setup JUnit for Testing Core Functions

✅ Add JUnit dependency in build.gradle (Module: app)

```
testImplementation 'junit:junit:4.13.2'
```

✅ Create a Sample Math Utility for Testing

```
class MathUtils {
    fun add(a: Int, b: Int): Int = a + b
}
```

✅ Write a JUnit Test in MathUtilsTest.kt

```
import org.junit.Assert.assertEquals
import org.junit.Test

class MathUtilsTest {
    private val mathUtils = MathUtils()

    @Test
    fun testAddition() {
        assertEquals(8, mathUtils.add(5, 3))
    }
```

}

✅ **Run the test:**

- Open **Run > Run 'Tests in MathUtilsTest'**.
- **Green checkmarks indicate successful tests.**

📌 **Step 6: Perform UI Testing with Espresso**

6.1 Setup Espresso for UI Testing

✅ **Add Espresso dependencies in build.gradle (Module: app)**

androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'

androidTestImplementation 'androidx.test.ext:junit:1.1.3'

✅ **Write an Espresso Test for Login Activity**

```
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.action.ViewActions.*
import androidx.test.espresso.matcher.ViewMatchers.*
import androidx.test.ext.junit.rules.ActivityScenarioRule
import org.junit.Rule
import org.junit.Test


class LoginActivityTest {


  @get:Rule
```

```
    val activityRule = ActivityScenarioRule(LoginActivity::class.java)


    @Test
    fun testLoginButtonClick() {
        onView(withId(R.id.username)).perform(typeText("user123"), closeSoftKeyboard())
        onView(withId(R.id.password)).perform(typeText("password"), closeSoftKeyboard())
        onView(withId(R.id.loginButton)).perform(click())
    }
}
```

✅ **Run the test:**

- Open **Run > Run 'Tests in LoginActivityTest'**.
- Espresso will **simulate typing username, password, and clicking login.**

📌 **Step 7: Prepare for Deployment**

7.1 Generate a Signed APK for Release

1. **Go to** Build > Generate Signed Bundle/APK.
2. **Choose APK** and click **Next**.
3. **Create a Keystore** (if not created before).
4. **Sign and build the release APK**.
5. **Upload the APK to Google Play Console**.

📌 Conclusion

✅ Optimized RecyclerView for performance.
✅ Reduced app size using ProGuard and WebP images.
✅ Identified and fixed memory leaks with LeakCanary.
✅ Implemented secure API key storage.
✅ Performed unit testing with JUnit and UI testing with Espresso.
✅ Generated a signed APK for deployment.