Simple Login Screen

1. What is XML?

- XML (Extensible Markup Language) is a structured text format used to store and transport data.
- It relies on tags (similar to HTML) but is designed to describe data rather than display it.
- XML is both human-readable and machine-readable, making it ideal for defining structured information.

2. Role in Android Development

XML role in Android development is primarily in defining user interfaces and resources:

- UI Layouts:
  - Android apps use XML files to define the structure of screens (e.g., buttons, text fields, images).
  - Example: A simple login screen is created using XML before adding functionality with Java/Kotlin.
- Separation of Concerns:
  - XML handles the presentation layer (how the app looks).
  - Java/Kotlin handles the logic layer (how the app behaves).
  - This separation makes apps easier to design, maintain, and scale.
- Project Structure & Resources:
  - XML is used in resource files (colors, strings, dimensions, styles).
  - It ensures consistency across the app and allows easy localization (different languages, screen sizes).
- Integration with Android Studio:
  - Developers visually design layouts in Android Studio, which generates XML under the hood.
  - XML files are stored in the res/layout directory of the project.

Hands-on: Create Simple Login Screen using XML

1. XML Layout (UI Definition) - This is stored in res/layout/activity_main.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="24dp"
   android:gravity="center">

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

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Username"
        android:inputType="textPersonName"/>

    <EditText
        android:id="@+id/passwordInput"
        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>
```

This defines the look and structure of the login screen: two input fields and a button.

2. Java/Kotlin Code (Logic Layer)

This goes in MainActivity.java or MainActivity.kt:

Java Example

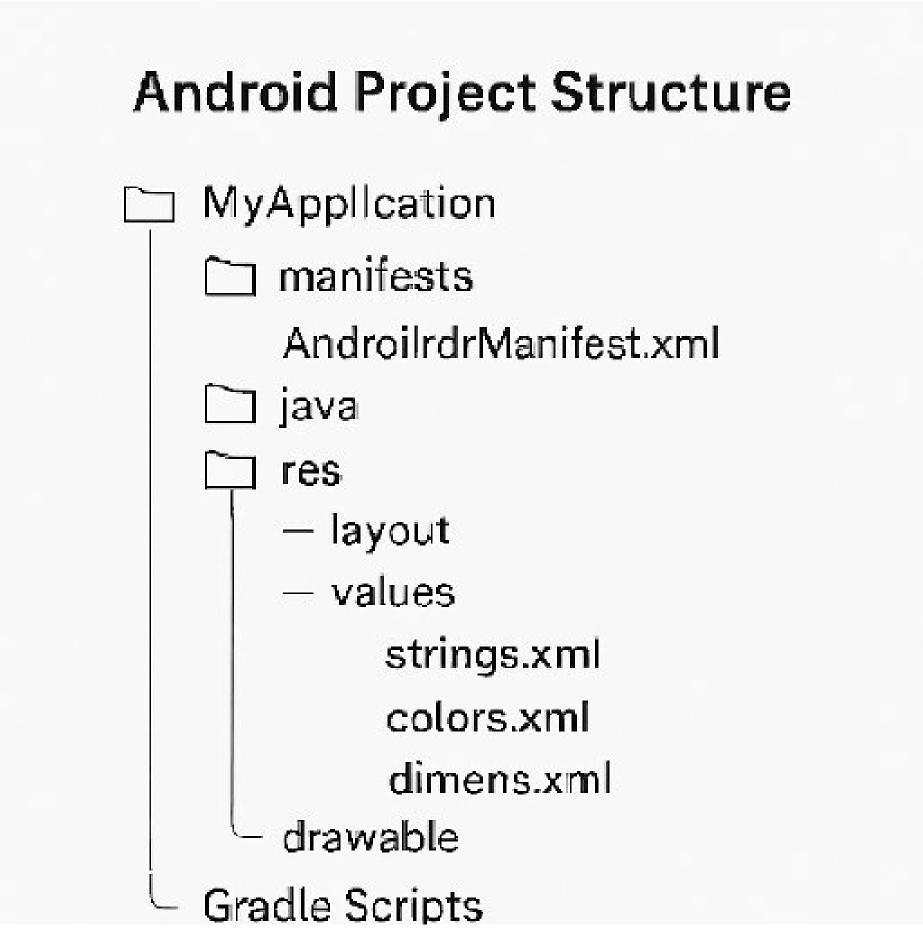
```
package com.example.loginapp;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        EditText username = findViewById(R.id.usernameInput);
        EditText password = findViewById(R.id.passwordInput);
        Button loginButton = findViewById(R.id.loginButton);

        loginButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String user = username.getText().toString();
                String pass = password.getText().toString();
            if(user.equals("admin") && pass.equals("1234")) {
                Toast.makeText(MainActivity.this, "Login Successful!",
Toast.LENGTH_SHORT).show();
            } else {
                Toast.makeText(MainActivity.this, "Invalid Credentials",
Toast.LENGTH_SHORT).show();
            }
        }
    });
  }
}
```

3. How They Work Together

- XML defines the UI elements (username field, password field, button).
- Java/Kotlin retrieves those elements using findViewById() and adds behavior (what happens when the button is clicked).
- Together, they create a functional login screen: XML handles the design, Java/Kotlin handles the logic.

In summary: XML is the blueprint for the app’s interface, while Java/Kotlin brings it to life with functionality.

Anatomy: Project Structure in Android



In Android Studio, every app project follows a standard structure. The training material highlights the key folders and files:

- manifests/
    - Contains the AndroidManifest.xml file.
    - Declares essential information about the app: package name, permissions, activities, services, and app entry point.
- java/
    - Holds all the Java/Kotlin source code.
    - Each Activity, Fragment, or helper class lives here.
    - Organized into packages (e.g., com.ltimindtree.myapp).
- res/ (Resources)
    - Stores non-code assets that define the look and feel of the app.
    - Subfolders include:

  - layout/ XML files for UI screens.
  - values/ XML files for strings, colors, dimensions, styles.
  - drawable/ Images and graphics.
  - mipmap/ App icons.
- Gradle Scripts
  - Build configuration files (build.gradle) that manage dependencies, SDK versions, and compilation settings.

While the AndroidManifest.xml acts as the blueprint for your app's identity, the build.gradle files serve as the "recipe" for how your app is actually cooked (compiled and packaged).

In modern Android development (using Android Studio), the build system is powered by Gradle. Most projects have two distinct build.gradle files to keep things organized.

While the AndroidManifest.xml acts as the blueprint for your app's identity, the build.gradle files serve as the "recipe" for how your app is actually cooked (compiled and packaged).

In modern Android development (using Android Studio), the build system is powered by Gradle. Most projects have two distinct build.gradle files to keep things organized.

1. The Two-File Structure

A. Project-level build.gradle (The Manager)

Located in your project's root directory, this file defines the build configurations that apply to every module in your project. It's where you define where Gradle should look for libraries (repositories) and which version of the Android Gradle Plugin to use.

B. Module-level build.gradle (The Specialist)

Located inside the /app folder, this is where the real action happens. This file dictates how a specific module is built. If you have a project with an "App" module and a "Library" module, each will have its own version of this file.

2. Key Sections of the Module-level File

This is the file you will spend 95% of your time in. Here is the breakdown:

Section	Purpose
plugins	Tells Gradle which tools to use (e.g., Android Application, Kotlin, or Google Services).
android { ... }	The main configuration block for Android-specific settings like SDK versions.
defaultConfig	Sets your App ID (package name), minimum Android version, and target version.
buildTypes	Defines different versions of your app (e.g., debug for testing and release for the Play Store).
dependencies	A list of external libraries your app needs (e.g., Retrofit for networking or Glide for images).

3. Why build.gradle is Important

- Dependency Management: Instead of manually downloading .jar files, you just type a line of code, and Gradle fetches the library from the cloud automatically.

- Version Control: It manages the compileSdk (the newest features your app can use) and minSdk (the oldest phone that can install your app).
- Automation: It handles the complex process of turning your code, images, and resources into a single .apk or .aab file that can be installed on a phone.
- Flavoring: It allows you to create different versions of the same app (e.g., a "Free" version with ads and a "Paid" version without ads) from the same codebase.

4. A Quick Look at the Code

Here is what a standard build.gradle (module-level) looks like in the modern Kotlin DSL (.gradle.kts) format:

Kotlin

```kotlin
plugins {
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
}

android {
    namespace = "com.example.myapp"
    compileSdk = 34 // The version used to compile your app

    defaultConfig {
        applicationId = "com.example.myapp"
        minSdk = 24    // The oldest Android version supported
        targetSdk = 34 // The version the app is optimized for
        versionCode = 1
        versionName = "1.0"
    }

    buildTypes {
        getByName("release") {
            isMinifyEnabled = true // Shrinks your code to make it smaller
            proguardFiles(getDefaultProguardFile("proguard-android.txt"), "proguard-rules.pro")
        }
    }
}

dependencies {
    implementation("androidx.core:core-ktx:1.12.0")
    implementation("com.squareup.retrofit2:retrofit:2.9.0") // Adding a library
}
```

1. compileSdk (The Teacher)

This is the version of the Android API that Android Studio uses to compile your code. It determines which features and APIs are available for you to write in your Java code.

- Rule: This should always be set to the latest version of Android available.
- Analogy: This is the textbook you are studying. If you use a textbook from 2015, you won’t know about any of the new rules of the road (features) introduced in 2024.
- Key Detail: Changing this does not change how your app runs on a user's phone; it only changes what tools you can use while writing code.

2. targetSdk (The Behavior)

This tells the Android system: "I designed and tested this app for version X."

- How it works: Android uses this to maintain "forward compatibility." If a new version of Android (e.g., Android 14) changes how permissions work, but your targetSdk is set to 13, the phone will run your app in a compatibility mode to ensure it doesn't break.
- Rule: You should keep this as high as possible (usually matching compileSdk) to ensure your app feels modern and secure.
- Analogy: This is the year you took your driving test. The police (Android OS) might give you a little leeway if the rules changed yesterday, but they expect you to update your knowledge eventually.

3. minSdk (The Floor)

This is the absolute oldest version of Android that can run your app. Devices running anything older will simply see a "Not Compatible" message in the Play Store.

- The Trade-off:
    - Lower minSdk: More potential users, but more work for you (you have to

write extra code to handle old bugs or missing features).

- Higher minSdk: Fewer users, but you can use all the cool new Java features without worrying about old phones.

- Analogy: The "You must be this tall to ride" sign at a roller coaster.

Quick Comparison Table

Property	What it controls	Where it matters
compileSdk	Available code/APIs	During Development
targetSdk	App behavior/Security	During Runtime
minSdk	Device compatibility	During Installation

A Practical Example

Imagine you want to use the Fingerprint Sensor API (introduced in API 23):

1. Your compileSdk must be 23+ or you can't even type the code for it.
2. If your minSdk is 21, you must write an if statement to check the user's version, or the app will crash on older phones: