Department of Computer Science
FC College University

Department of Data Science
University of the Punjab

HO# 2.12 Android App Pen-Testing - I

A Hello World on Android App Development using Android Studio

History of Android Operating System

History of Android goes back to **2003**, when four friends Andy Rubin, Rich Miner, Nick Sears and Chris White planned to create an operating system for digital cameras. Later, they realized that the mobile phone industry had more potential, so they shifted their focus to developing an open-source OS for mobile devices. In **2005**, Google saw potential in Android's vision and acquired Android Inc. for around $50 million and as part of the deal, Andy Rubin and his team joined Google. Today, Android dominates over 70% of the global smartphone market. It powers a vast range of devices, including smartphones, tablets, wearables, and even some smart TVs and automotive systems. Here's a summary of some important versions of Android versions with their newly introduced features and the Linux Kernel used in each. Remember, the actual Kernel version may vary based on the device manufacturer and to ensure optimal compatibility and performance with the specific hardware requirements:

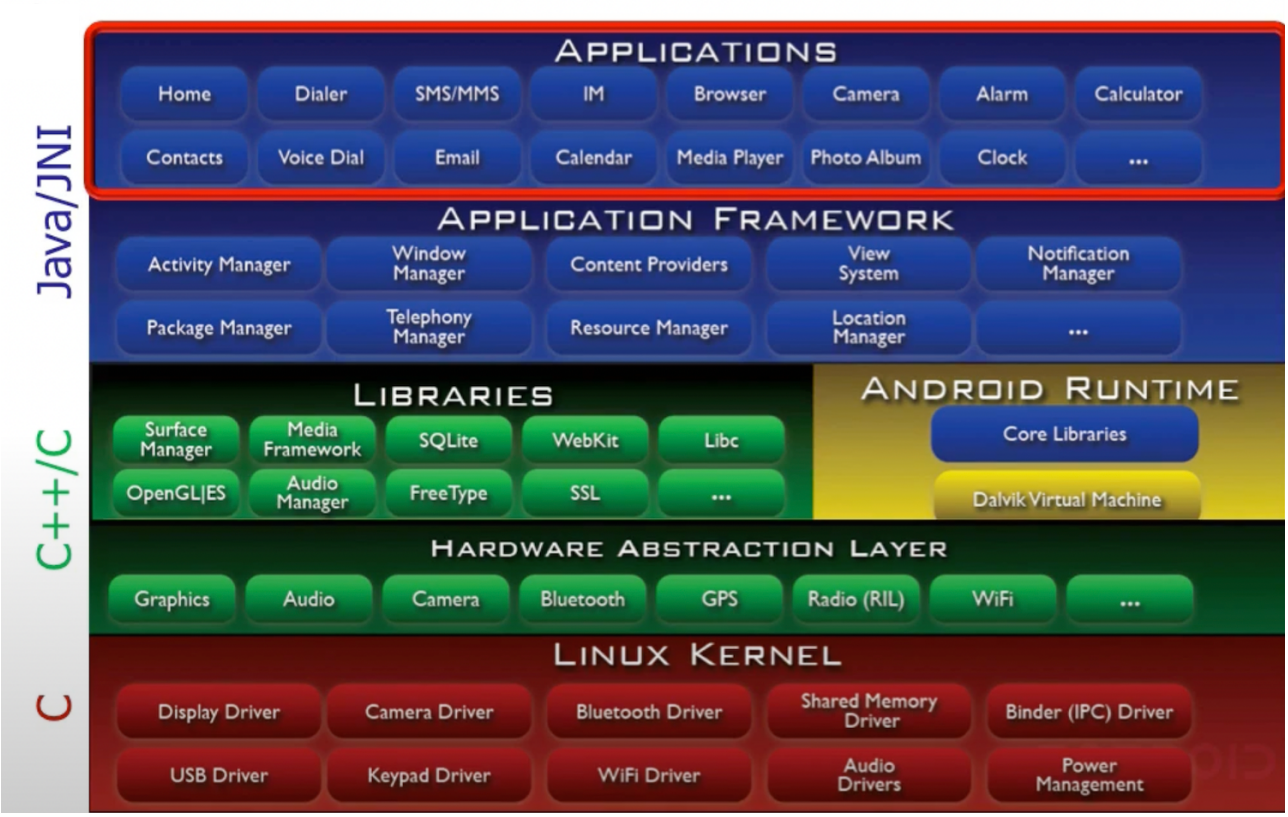
- **Android 1.0 (2008)**: The first-ever Android phone, HTC Dream (T-Mobile G1) was launched that ran Android 1.0. It featured a basic set of applications including a web browser, email client, and Google's core services. (*Linux Kernel version: 2,6.25)*
- **Android 1.5 (Cupcake, 2009)**: Introduced several important features including an on-screen keyboard, support for third-party apps, and video recording.
- **Android 2.0 - 2.1 (Eclair, 2009-2010)**: Included features like support for multiple accounts, better camera controls, and an updated user interface. The introduction of Google Maps Navigation was a notable addition.
- **Android 2.3 (Gingerbread, 2010)**: Introduced support for front-facing cameras and video calls.
- **Android 3.0 - 3.2 (Honeycomb, 2011)**: Featured a new user interface designed for larger screens (tablets).
- **Android 4.0 (Ice Cream Sandwich, 2011, API 16)**: Unified the phone and tablet interfaces into a single operating system. It introduced the Holo theme, improved multitasking, and better facial recognition.
- **Android 4.4 (KitKat, 2013, API 19)**: Focused on performance optimizations for low-end devices and introduced a new user interface with translucent status and navigation bars.
- **Android 5.0 - 5.1 (Lollipop, 2014-2015, API 21)**: Introduced Material Design, improved battery life and performance enhancements.
- **Android 6.0 (Marshmallow, 2015, API 23)**: Introduced features like Doze mode for better battery life, and app permissions management.
- **Android 7.0 - 7.1 (Nougat, 2016, API 24)**: Brought features like split-screen multitasking, quick settings, and improved notifications.
- **Android 8.0 - 8.1 (Oreo, 2017)**: Focused on performance improvements, battery optimization (Adaptive Battery), and introduced picture-in-picture mode.
- **Android 9.0 (Pie, 2018, API 26, API 28)**: Added gesture navigation, improved AI-based features, and focused on digital wellbeing with tools like Dashboard and App Timer. *(Linux Kernel version: 4.4 - 4.14)*
- **Android 10 (Quince Tart, 2019, API 29)**: Introduced system-wide dark mode, and gesture navigation. *(Linux Kernel version:4.9-4.19)*
- **Android 11 (Red Velvet Cake, 2020, API 30)**: Focused on user control over notifications, chat bubbles, and privacy settings. Introduced improvements for 5G and foldable devices. *(Linux Kernel version: 4.14 - 5.4)*
- **Android 12 (Snow cone, 2021, API 31)**: Introduced the "Material You" design language for personalized themes, improved privacy controls, and enhanced performance. *(Linux Kernel version: 4.19 - 5.10)*
- **Android 13 (Tiramisu, 2022, API 33)**: Continued the Material You design with more customization options, improved support for large-screen devices, and enhanced privacy features. *(Linux Kernel version: 5.10 - 5.15)*
- **Android 14 (Upside down cake, 2023, API 34)**: Focused on further customization, performance optimizations, and new features for large-screen devices and foldables. *(Linux Kernel version: 5.15 - 6.1)*
- **Android 15 (Vanilla Ice Cream, 2024, API 35)**: Focused on further customization, security aspects, and performance optimizations. *(Linux Kernel version: 5.15, 6.1 - 6.6)*
- **Android 16 (Baklava, 2025)**: As of this writing Android 16 is in the pipeline, featuring platform stability milestone and privacy sandbox improvements. Based on Android Common Kernel (ACK), which is a modified version of 6.12 *Linux LTS Kernel.*

Android Device Manufacturers and Processors

- **Samsung:** Galaxy S, Galaxy Z, Galaxy Note, Galaxy A, Galaxy M ***[Exynos, Snapdragon]***
- **Google:** Pixel 8, Pixel Fold **[Tensor]**
- **Xiaomi:** Mi, Redmi, Poco, Black Shark **[Snapdragon, MediaTek Dimensity]**
- **OnePlus:** OnePlus Number series (11, 12), Nord **[Snapdragon, MediaTek Dimensity]**
- **Oppo:** Find X, Reno, A-Series **[Snapdragon, MediaTek Dimensity]**
- **Vivo:** X-Series, V-Series **[Snapdragon, MediaTek Dimensity]**
- **Realme:** GT, Narzo, Realme ***[Snapdragon, MediaTek Dimensity]***
- **Motorola:** Edge, Moto G, Moto E, Razr ***[Snapdragon, MediaTek Dimensity]***
- ***Nokia:*** *Android One devices* ***[Snapdragon, MediaTek Dimensity]***

Key Layers in Android Software Architecture

Android is a popular open-source smart phone Operating System, based on Linux Kernel, developed by Google. Being open-source, you can download the source code of Android OS from the Open-Source Android Platform (OSAP) by visit this link: https://source.android.com/docs/setup/download. In order to have a deep understanding of the security aspects both offensive and defensive of Android devices, it is very important to have a crystal-clear understanding of the Android software architecture stack. Android provides a layered stack for mobile devices, where each layer does a set of things in a standardized way and also provides a level of abstraction. This is shown in the diagram:



1. **Linux Kernel:** Positioned at the bottom of the Android software stack, the Linux Kernel provides a level of abstraction between the device hardware and the upper layers of the Android software stack. The kernel provides low-level core system services such as virtual memory, process/thread, power management, the four layered TCP/IP network stack and device driver framework (function entry points for character/block-oriented h/w devices). The Linux kernel used by Android is not the same as the vanilla kernel you would find at https://linux.org/. Android provides Android-specific kernel enhancements that include various patches and modifications to work better with mobile hardware, touchscreen, sensors, power management, Binder IPC, and system optimizations specific to mobile devices. Moreover, Google introduces many security enhancements specific to Android (apart from the Linux kernel) like verified boot, SELinux, app sandboxing, KASLR, Spectre & Meltdown patches, etc.
2. **Hardware Abstraction Layer:** The HAL is a user-space C/C++ library layer for accessing h/w devices. This seems odd because usually device drivers live in kernel space. One reason is lot of inconsistent h/w devices are there in modern smart phones which are not there in normal Linux Desktop/Laptop environment. For example, in your laptop running Linux, there might be a Wi-Fi but it doesn't have a cellular radio interface, because you do not make phone calls with it. Likewise, you might not have a very high-resolution cameras in your laptop that exists inside mobile devices these days. Second reason is, if they have put all these drivers inside OS kernel (which being open-source), then everybody who was a manufacturer of Android devices would have had to release all source code

of their touchscreen drivers, accelerometer and so on under GPL license. So HAL component implementations are not opensource.

3. **Android Runtime:** The android runtime provides a key component known as ***Dalvik VM*** which is a kind of JVM (Java Virtual Machine) specially designed and optimized for executing Android applications. The developer of Dalvik (Dan Bornstein) named it after his home town in Iceland. Dalvik VM enables every Android app to run in its own process, with its own instance of the Dalvik VM. So, it is created when a process is started and destroyed when it exits. Dalvik is being phased out by Android Runtime (ART) that uses ahead-of-time compiler instead of JIT compiler and has better garbage collection. Android runtime also provides a set of ***Core Libraries*** (written in java/kotlin) which consist of standard Java libraries and additional Android-specific APIs that facilitate mobile app development. Application developers mostly interact with the core libraries inside Android runtime, rather than the native C/C++ libraries. For example, `android.app` is one of the core libraries (others are `android.content`, `android.os`, `android.view`, `android.widget`, `java.io`, `java.net`) that contains components for UI and background tasks like Activities, Services, Content Providers and Broadcast Receivers. You will come across methods like `onCreate()`, `onStart()`, `onResume()`, `onDestroy()` in Activity and Services classes that are essential in managing lifecycle of components in an Android app.

4. **Native Libraries:** These are collection of C/C++ libraries that provides essential low-level functionalities and directly interact with Linux Kernel to enable system operations. Although Android apps are written using Java APIs, the implementation of these APIs are often written using native C/C++ via the Java Native Interface. JNI provides an interface using which a developer can call the native methods of non-Java programming languages (such as C and C++) from within Java code. This is how the code inside all these layers interact/communicate with each other. One last thing is Native Development Kit (NDK) is a toolset that allows developers to implement part of Android app using native code languages like C/C++. This may optimize performance by minimizing latency, maximizing throughput and conserving resources. The most famous native library written in C/C++ is the system C library `libc` that provides C interfaces to many UNIXs system services like processes, threads and memory management. Other is Media Framework which does audio/video streaming, which is computationally extensive processing so C/C++ is the appropriate choice for that.

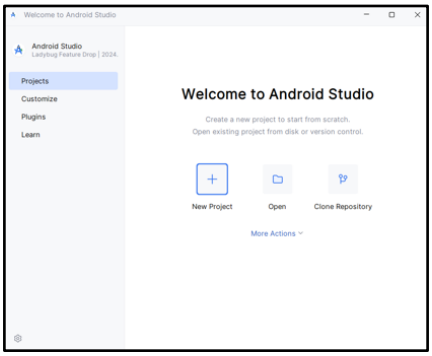
5. **Application Framework:** The layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications. Some key services that Application Framework provides are:
    - Window Manager: Manages top level windows look and behavior.
    - View System: An extensible set of views used to create application user interfaces.
    - Activity Manager: Controls all aspects of the application lifecycle and activity stack.
    - Content Provider: Allows applications to publish and share data with other applications.
    - Notifications Manager: Allows applications to display alerts and notifications to the user.
    - Package Manager: Allows you to install new apps and find info about already installed apps.
    - Telephony Manager: Keeps track of the telephone and radio allowing a user to make calls.
    - Location Manager: Provides access to the location services allowing an application to receive updates about location changes.

6. **Applications:** The android packaged applications are in the top layer of the architecture. They are mainly written in Java/Kotlin, although internally there are lots and lots of calls to JNI taking place under the hood. In order to distribute the Android applications, all the necessary components (DEX files, Manifest file, Resources and Libraries) are compressed into an archive called Android Package Kit (APK) or Android App Bundle (AAB). Android apps can be divided into two main categories:
    - Pre-installed Apps: Android is shipped with over two dozen open-source applications like dialer, contacts, messaging, camera, photos, calendar, calculator, clock, setting, file manager, chrome, maps and so on.
    - User-installed Apps: WhatsApp, Facebook, Instagram, Snapchat, Twitter/X, Telegram, Discord, Browsers (Mozilla Firefox, MS Edge, Brave, Opera), Netflix, Spotify, MS office, Evernote, Adobe Reader, CamScanner, OpenVPN, and so on.

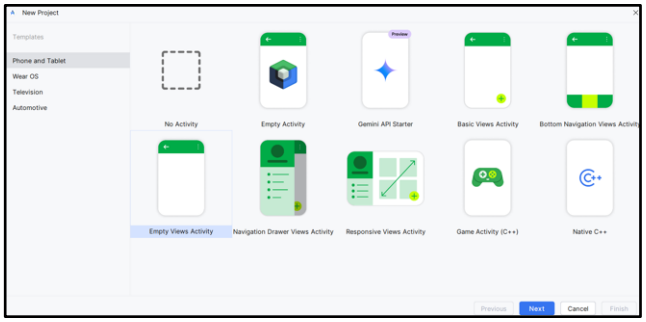
Designing, Building and Running a Hello World Android App

- Dear students, although we are more concerned with the security aspects of Android devices, and soon we will perform static/dynamic analysis of android apps by reverse engineering them using tools like `apktool`, `jadx`, `dex2jar` and `MobSF`. But for a crystal-clear understanding, it is a must to know as to how an Android application is designed, coded, build and distributed.

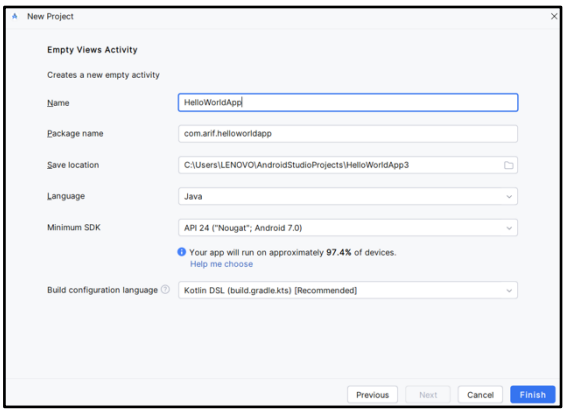
- **Download and Install Android Studio** https://developer.android.com/studio
  - Android Studio is the official Integrated Development Environment(IDE) for Android native app development using Java or Kotlin programming languages.
  - Download and install Android Studio on your machine (Windows, MacOS or Linux). I have installed Android Studio ***Ladybug*** on my host Windows11 machine, you can install ***Meerkat*** what is the latest one that came in March 2025.



- **Creating a New Android Project**
  - When you start Android Studio, you will come across a Welcome screen (as shown) from where you can either start a new project or open an existing one. Moreover, you have the option to customize your Android Studio environment as well.
  - When you start a new project for a Phone or Tablet, you need to select a project template out of the available options. To start with our hello world project, you will select **Empty Views Activity**. This template is ideal if you want to build the UI entirely yourself without having any layout components already added for you.

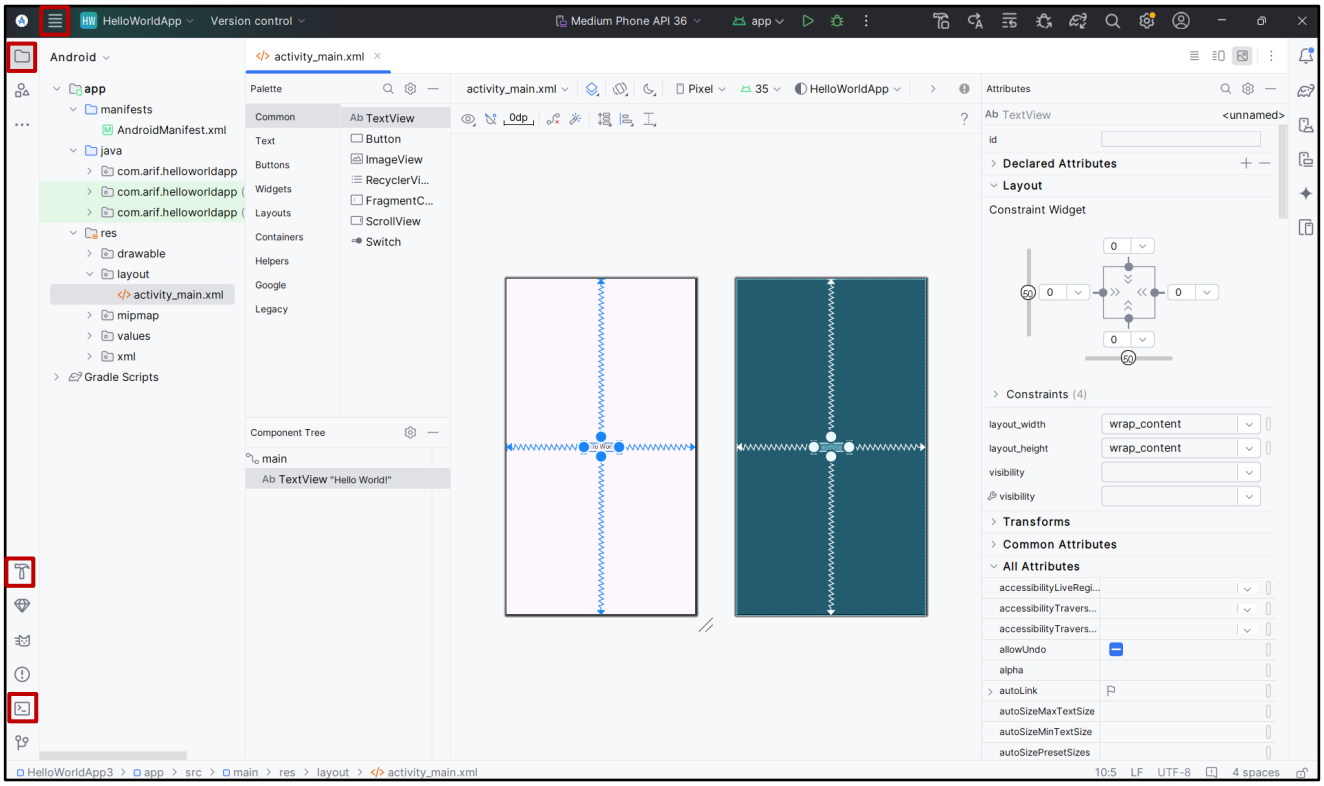


  - Then in the New Project screen, you will give the project name, package name, select programming language (Java/Kotlin), Minimum SDK API version and Build Configuration language (for your `build.gradle` scripts).
  - Every Android version has an associated SDK (API level), that brings in new features, capabilities, bug fixes, and changes in behavior. The **minimum SDK** specifies the lowest Android version (API level) your app can run on. You may come across **Target SDK** (version of Android your app is optimized for) and **Maximum SDK** (specifying the highest Android version your app supports, though it's rarely set since Android is forward-compatible).

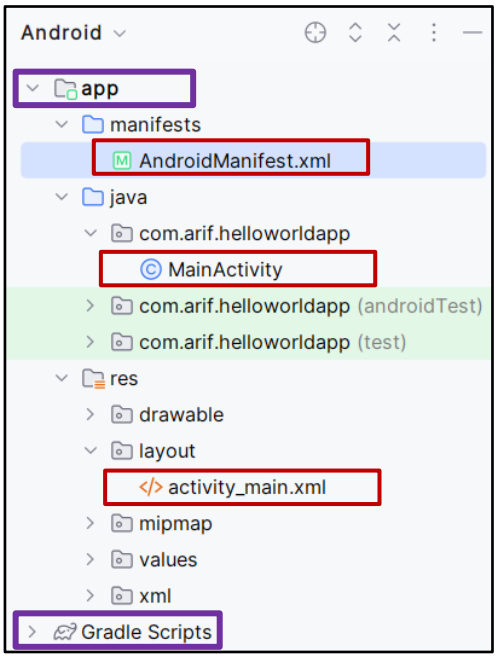


Android Studio IDE Description

Once you run Android studio and start a project, you will come across the GUI of Android studio, as shown in the screenshot below having a **Project Window**, **Palette**, **Component Tree**, **Design Editor**, and **Attributes Window** as shown in the screenshot below:

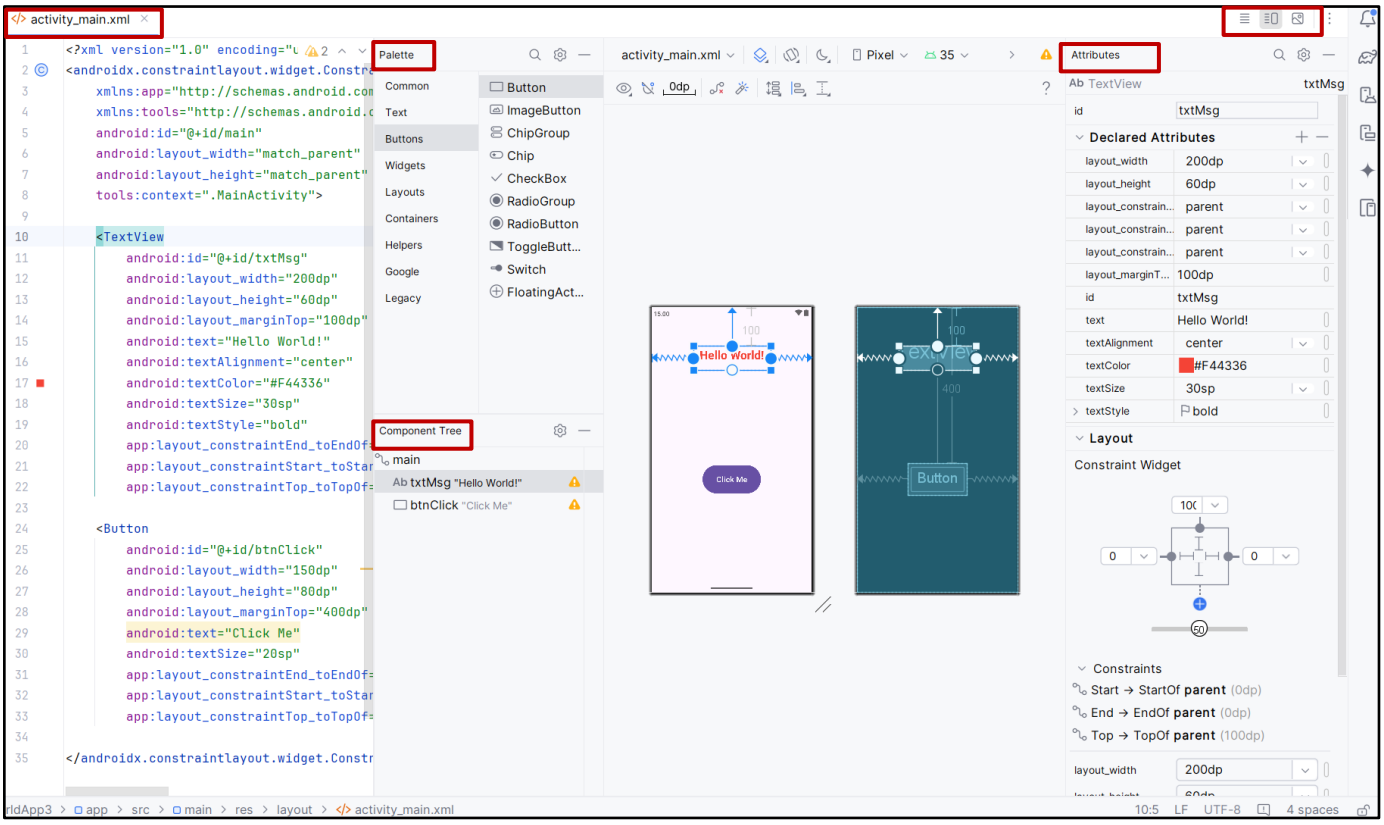


Project Window: It shows the structure of your Android project and its files, having `app` and `Gradle Scripts` folders. The `app` folder further contains `manifests`, `java` and `res` folders.



- The **`manifests` folder** contains the most famous file *`AndroidManifest.xml`*, that describes the essential information required by the Android build tools, Android OS and Google Play. This file is used to declare app components, permissions, minimum SDK version, h/w features and the API libraries the app needs to be linked against.
- The **`java` folder** contains the Java/Kotlin source code for all your activities (screens) in your app. Each activity has its own Java file to define how it behaves. The file of our interest right now is the *`MainActivity.java`* that is usually the entry point for your app and contains the logic for the first screen/activity.
- The **`res` folder** contains additional files containing static content that your code uses, drawable (contains image files used in your app like icons and backgrounds), layout (contains XML files that defines the UI layout of your app screens, e.g., `activity_main.xml`), mipmap (contains app icons in different resolutions optimized for different screen densities), values (contains xml files like colors, strings, and themes).
- Finally, the Gradle is an automated build system that handles compiling, linking and packaging the app (contains scripts for these tasks).
- Note: For every activity we have two files a `.xml` for designing the screen UI and `.java` for writing the business logic.

UI Design: For our Hello World app, since we selected the template as an Empty Views Activity, therefore it contains an empty layout file (`activity_main.xml`) that doesn't come with any pre-configured UI components, giving you the flexibility to add your own Views (like Buttons, TextViews). The following screenshot shows how we are going to design the main activity of our HelloWorld app.



Above screenshot has following important windows related to UI design:

- **Palette** is a set of pre-designed UI components that you can drag and drop into your layout. I have drag-dropped a TextView and a Button component on the design view of this activity.
- **Component Tree** shows the view hierarchy of all the views and layout components that are currently in your layout file. It allows you to view how your UI elements are nested and organized.
- **Design Editor** displays a Design view and a Blueprint view to give you a visual representation of your layout.
- **Attributes Panel** allows you to customize the properties of a selected UI component. All you need to do is to select a component in the design view of Component tree and change properties like id, text, textStyle, textColor, textAlignment, textSize (in sp) and layout height/width (in dp). Density-independent pixels (dp) is used for defining sizes of UI elements (buttons, images, etc.), ensuring consistent appearance across devices having different screen densities (different screens have different pixels per inch (PPI)). Similarly, scale-independent pixels (sp) is similar to dp, but is primarily used for defining text size ensuring that the text size scales appropriately based on user's system-wide font size preferences. Another important property that we must set for every component is the layout constraint, that defines the position and size of a UI component relative to other components or the parent container. Out of four, we must set three at least, the left, right and either the top and the bottom constraint.

Finally, the XML code inside **`activity_main.xml`** file represents your layout and defines the structure and properties of the UI components. When you add or modify components in the design view, Android Studio automatically generates the corresponding XML code ☺

Program Logic: Now we are done creating our UI, it is time to write some Java code, inside the `MainActivity.java` that is associated with `activity_main.xml`. We want that when a user clicks the button (`btnClick`), the text of the textView (`txtMsg`) should be changed to **"Hello Android!"**. Following is the screenshot that shows the complete code, in which most of the code is self-generated by Android Studio. We just have to write few lines of codes that I have highlighted inside two red rectangles:

- The `findViewById()` is a method that is used to reference a UI element from our XML layout file and associate it with a Java variable so that we can manipulate it in our code. The argument passed to this method is `R.id.btnClick`.
  - `R` is the Root class that contains references to all the resources in our project (layouts, strings, images etc.).
  - `id` refers to the component ID defined in our XML layout file.
  - `btnClick` is the ID that we assigned to the button component.
- The code inside the second red rectangle is setting an OnClickListener on a Button object, so that when the button is clicked, it performs a specific action:
  - The `btn.setOnClickListener()` is a method that specifies as to what should happen when the `btn` is clicked. It expects an instance of the `OnClickListener` interface.
  - The `new View.OnClickListener()` creates an anonymous class that implements the `OnClickListener` interface
  - Then we override its `onClick(View v)` method, which is passed an argument of type View (btn in this case) that was clicked.
  - The single line of code that we actually wrote simply use the `setText()` method of the `txt` object to set its text to **"Welcome Android!"**.

MainActivity.java | AndroidManifest.xml | activity_main.xml

```
1   package com.arif.helloworldapp;
2
3   import ...
13
14  public class MainActivity extends AppCompatActivity {
15      @Override
16      protected void onCreate(Bundle savedInstanceState) {
17          super.onCreate(savedInstanceState);
18          EdgeToEdge.enable( $this$enableEdgeToEdge: this);
19          setContentView(R.layout.activity_main);
20          ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main), (v, insets) -> {
21              Insets systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars());
22              v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom);
23              return insets;
24          });
25          Button btn = findViewById(R.id.btnClick);
26          TextView txt = findViewById(R.id.txtMsg);
27          btn.setOnClickListener(new View.OnClickListener() {
28              @Override
29              public void onClick(View v) {
30                  txt.setText("Welcome Android!");
31              }
32          });
33
34      }
35  }
```

`AndroidManifest.xml`: Every Android app has an `AndroidManifest.xml` file, that describes the essential information required by the Android build tools, Android OS and Google Play. This file is used to:

- Declare the four components of an Android app (activities, services, broadcast receivers, and content providers).
  - **Activities:** An activity represents a single screen that implements one user facing operation like login screen, setting up the alarm, reading/composing an email message, viewing a contact, browsing the Internet etc.
  - **Services:** Components that run in the background to perform long running operations.
  - **Content Providers:** Components that manages persistent data in structured storage and supplies data from one app to the other on request, e.g., sharing a file from file explorer to WhatsApp.
  - **Broadcast Receivers:** Components that listen and respond to broadcast messages from other apps or from the system, like when the device is charging or when the Wi-Fi status changes.
- Declares any user permissions the app requires, such as Internet access or read-access to the user's contacts.
- Declares the minimum API Level or minimum SDLK version required by the app.
- Declares hardware and software features used or required by the app, such as a camera, Bluetooth services, or a multi-touch screen.
- Declares API libraries the app needs to be linked against (other than the Android framework APIs), such as the Google Maps library.

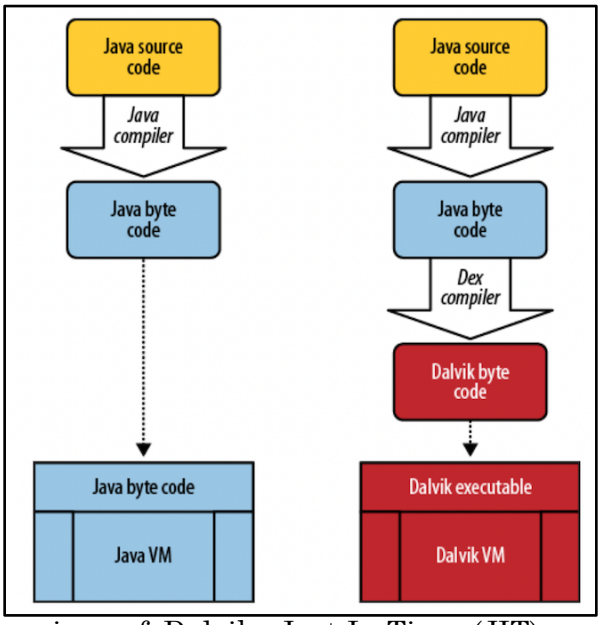
Let me describe the four app components using a basic Alarm application:

- Suppose you open the Alarm application, a UI will be displayed and using that interface you will set the alarm. So, the interface that is opened using which you can set the alarm is nothing but the **Activity**.
- When you set the alarm, the time will either be stored in a file or may be in some database. The component which is responsible for storing the alarm time inside file/database is called **Content Provider**.
- Once the alarm is stored inside a file/database, there is a service which continuously run in the background and check the system clock time and the set alarm time. The process that continuously run in the back ground and compare the two times is a **Service**.
- Once the service finds that the current time is equal to the alarm time, then it will start the alarm event, which will be handled by the **broadcast receiver**.

The following snippet of manifest file does not exactly relate to our HelloWorld app.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http"//schemas.android.com/tools">

<!-- Permissions your app needs like accessing the internet, using the camera, reading contacts, etc -->
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_READ_CONTACTS " />

<!-- The application tag contains the four components tags (activity, service, receiver, and provider -->
    <application
        android:name=".myapp"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="HelloWorldApp"
        tools:targetApi="31">

<!-- The activity component(s) represent a single screen with a UI -->
        <activity android:name=".MainActivity"
                  Android:exported="true">
<!--The intent-filter specifies which actions (e.g MAIN) and categories (e.g LAUNCHER) this activity should handle-->
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

<!-- The service component(s) run in the background to perform long running operations. -->
        <service android:name=".MyService"
                 android:enabled="true"
                 android:exported="false" />

<!--Broadcast Receivers are components that listen and respond to broadcast messages from other apps or from the system,
(like when the device is charging or when the Wi-Fi status changes. It is declared using the <receiver tag-->
        <receiver android:name=".MyReceiver"
                  android:enabled="true">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
            </intent-filter>
        </receiver>

<!-- The Content Provider component(s) supplies data from one app to other on request, e.g., sharing a file from file
explorer to WhatsApp. It is declared using the <provider> tag -->
        <provider android:name=".MyContentProvider"
                  android:authorities="com.example.myapp.provider"
                  android:exported="true" />

    </application>
</manifest>
```

Android Application Build Process

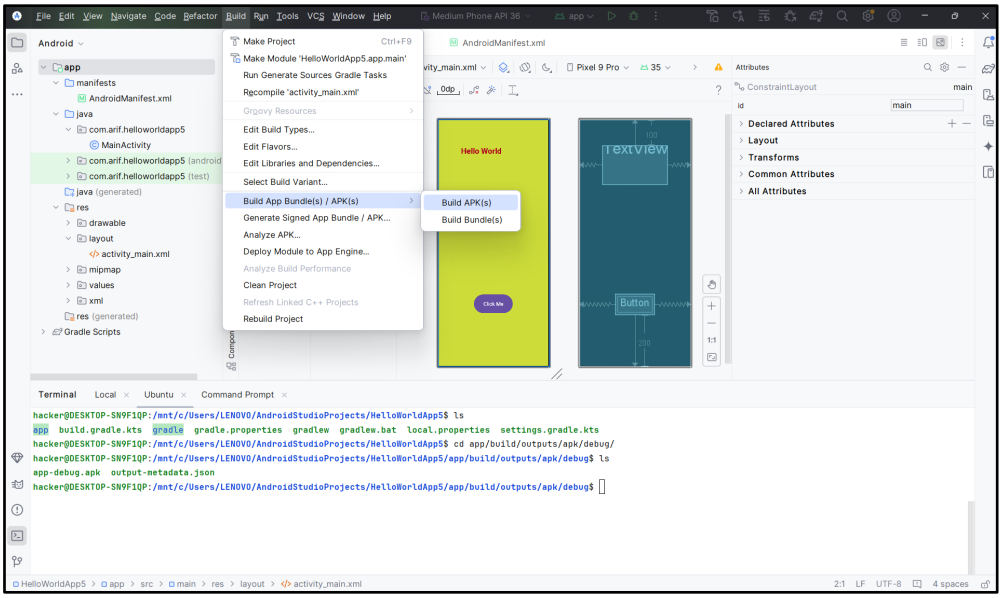


- In a normal Desktop environment, you compile your Java source file (`.java`) into Java byte code (`.class`) using the Java compiler (`javac`), and then run this byte code inside the Java Virtual Machine. The JVM was designed to be a one-size-fits-all solution (left image).
- Android applications are written in Java or Kotlin (`.java` or `kt`), and the source file is compiled using `javac` or `kotlinc` to Java byte code (`.class`). The additional step is that all the `.class` files are given to Dalvik compiler (**`d8`**) that is responsible for converting the `.class` files into **D**alvik **EX**ecutable `.dex` file(s), which executes on Dalvik Virtual Machine (a purpose-built virtual machine designed specifically for Android).
- **Execution before Android 5.0:** It used Dalvik VM, which used an interpreted approach to execute the bytecode instructions inside the `.dex` file(s). In later versions of Dalvik, Just-In-Time (JIT) compiler was introduced to improve performance by dynamically compiling frequently used code into native machine code during execution.
- **Execution after Android 5.0:** The new Android devices use Android Run Time (ART) instead of Dalvik VM. ART introduced Ahead-of-Time (AOT) compilation, which means that at the time of app installation the `.dex` file(s) are converted into native machine code. This results in faster app startup times. Even with AOT compilation, ART still uses JIT compilation to compile frequently used code at tun time to achieve performance gains.
- The **Android Application Build Process** is a sequence of steps that compiles your Android app into an APK ready for installation on devices or AAB for publishing on Google Play store. The build process is managed by *Gradle*, an automated build system that handles compiling, linking, and packaging the app. The steps performed during the build process are mentioned below:

  1. ***Resource Processing:*** The build process compiles non-code resources like xml files (layouts, values, drawables), assets (images or fonts), and resources (strings, colors, styles). The `res/` directory and other asset files are packaged into a `resources.arsc` file ready for inclusion in the APK.
  2. ***Source Code Compilation:*** The Java/Kotlin source code (`.java` or `.kt` files) is compiled into Java bytecode (`.class` files) using the `javac` compiler (for Java) or `kotlinc` (for Kotlin).
  3. ***Conversion to DEX Format using D8/R8:*** The compiled `.class` files are then converted into Dalvik Executable (`.dex`) files using `D8` for debug builds or using `R8` for release builds. Other than `dexing`, `R8` performs `code` *shrinking, optimization,* and *obfuscation*.
  4. ***APK Packaging:*** Once all the code is compiled and resources are processed, `Gradle` packages all the `.dex` files, processed resources, and native libraries into an Android Package Kit (`APK`) or Android App Bundle (`AAB`). Remember APK files are installed directly on devices or distributed via third-party sources, while AAB files are uploaded on Google Play store.
  5. ***App Signing:*** Android requires that all apps must be digitally signed with a certificate before they are installed on a device. App signing is a process that ensures the security, integrity and authenticity of an Android app. When you build an Android app, you need to "sign" it with a digital certificate before it can be installed on a device or emulator (self-signing) or published to the Google Play Store (CA).
  6. ***Running app on a Device/Emulator:*** The final signed APK is ready for deployment on an emulator, physical device, or to be uploaded the Google Play Store. Android Studio installs the APK on the connected device or emulator and then you can test it.

Building/Singing Android App (Generating `apk` file)

- To build your Android app, click main menu icon at the top left, which will open the menu bar, click Build and then click Build APK(s).
- The Gradle build system compiles your Java/Kotlin source code into bytecode (DEX format), processes your XML resources (layouts, strings, etc.), and merges them into the final output. It also applies any code shrinking, obfuscation, or optimization rules (like with ProGuard or R8), and signs the APK/AAB with either a debug or release key depending on the build type. App signing is a process that ensures the security, integrity and authenticity of an Android app.



Types of App Signing:

- **Debug Signing:** If you want to run your app on Android emulator or on a physical device by manually allowing installation from unknown sources, you can sign the `apk` using the debug key. When you Build a project, Android Studio automatically generate the key from the default *debug keystore*. It uses the default key as `androiddebugkey` and default password as `android`, so no manual action needed. Once the Build process finishes, the `apk` file signed using debug key can be found at the following location:
  `C:\Users\<username>\AndroidStudioProjects\<proj-folder>/app/build/outputs/apk/debug/app-debug.apk`
- **Release Signing:** If you want to distribute your app to users (through Google Play Store or other app markets), you need to sign your Android app with a private key. The signature proves that the app comes from the developer and in case of an update the app must be signed with the same key. For this you need to generate a private key of your own either using the `keytool` or Android Studio APK Wizard and it is saved in the *release keystore*. Self-signed keys are fully accepted by Android OS (for installation and updates), Google Play Store (for app publishing), as well as by third party app stores. So you do not need CA-Certificates for signing an APK.
  - Build → Generate Signed App Bundle / APK. Choose APK and click next.
  - Choose Keystore: If you don't have a keystore yet, create one by clicking "Create new…", if you already have generated a keystore, then you may need to give following information:
    ⇒ Give the keystore path and select your `.jks` file
    ⇒ Enter the password you set when creating the keystore
    ⇒ Give your key alias name
    ⇒ Specify password for the alias
    ⇒ Configure the keystore
  - Select the Build type as release and click Finish. The build process will start and once completed the `apk` file signed using release key can be found at the following location:
    `C:\Users\<username>\AndroidStudioProjects\<proj-folder>/app/release/app-release.apk`
- To publish your app on Google Play, create Google Play Developer Account at this link: https://play.google.com/console/signup. To upload/publish the Android App Bundle (AAB, not an APK) to Google Play watch this video: https://www.youtube.com/watch?v=jvEI-um-P7Q

Why is App Signing Important?

- **Security and Integrity**: By signing your app, you ensure that the app has not been altered after it was compiled. If the app is tampered with, the signature will not match, and Android will not allow the installation of the app.
- **Authentication**: Signing your app helps establish the identity of the developer. It ensures that the app you've developed and published is indeed the same app, without any malicious modifications.
- **Updates**: When you release updates for your app on the Google Play Store, the updates must be signed with the same key as the original app. This ensures that users can trust the updates as coming from the original developer.

App Signing Schemes

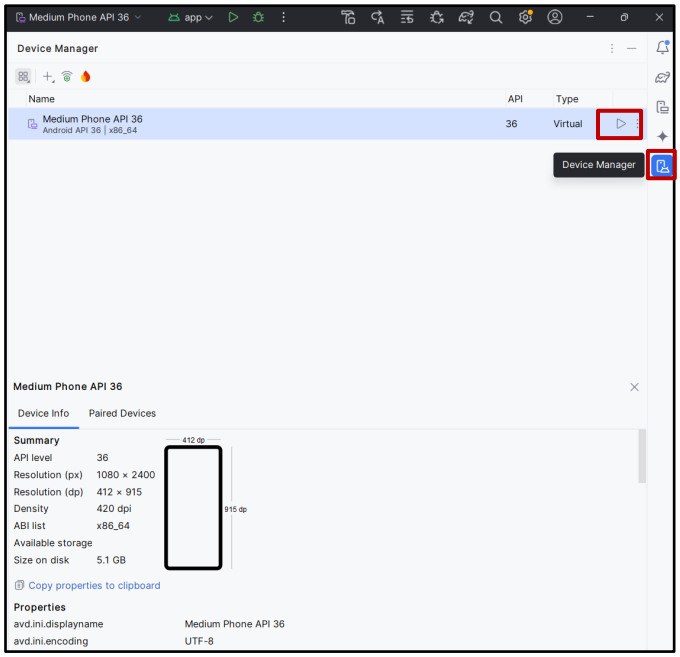
- **JAR Signature Scheme v1:** This scheme was introduced in Android 7.0 (Nougat, API level 24). It signs the APK as a whole, ensuring basic integrity but with limited security.
- **APK Signature Scheme v2:** This scheme was also introduced in Android 7.0 (Nougat, API level 28) alongside v1. It provides full-file signing, i.e., sign DEX files, resources and manifest.
- **APK Signature Scheme v3:** This scheme was introduced in Android 9.0 (Pie, API level 28). It adds timestamping allowing for long-term signature validity even after the signing certificate expires.
- **APK Signature Scheme v4:** This scheme was introduced in Android 12.0 (Snow cone, API level 31). It is optimized for signing split APKs, offers faster verification and improved security.

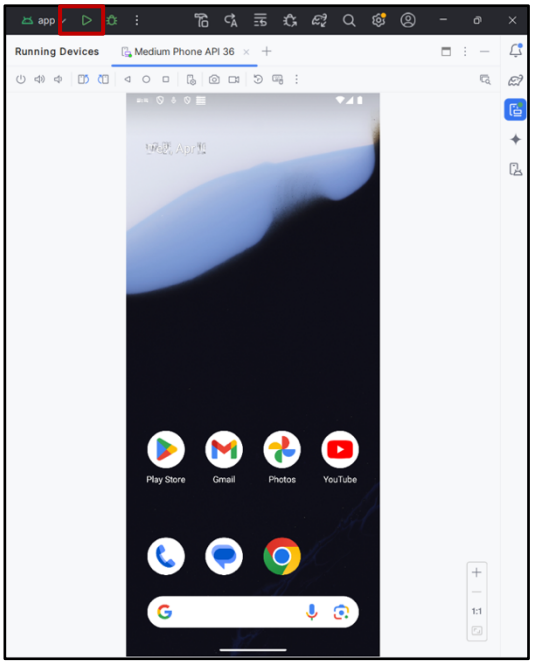
The APK Signature Scheme v2 suffers with ***Janus*** vulnerability allowing an attacker to inject malicious code into an APK without invalidating the signature, making it appear legitimate. To mitigate this, developers should use the V3 or V4 signing schemes.

Running an Android App

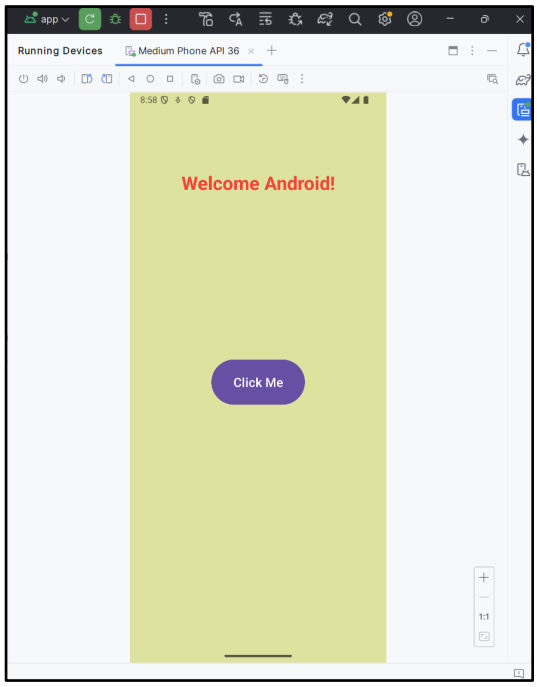
Once your APK is build, you can test it by installing and running it on a **physical** or a **virtual** Android device.

Option 1: (Use AVD inside Android Studio): The *Android Studio Device Manager* has a built-in *Android Emulator* that allows you to emulate different Android Virtual Devices (AVDs), each having different Android OS versions, screen sizes and so on for a controlled testing environment. This allows developers and security researchers to test apps or perform penetration testing without needing a physical Android device. Visit https://developer.android.com/studio/run/managing-avds to understand as to how you can download and install and manage different Android Virtual Devices inside Android Studio.





- In the **top left image**, click the Device Manager icon, to display all the connected AVDs. Click the Play button highlighted with a red square and it will start the appropriate AVD.
- The running AVD is shown in the **top right image**.
- Play with the device by running the pre-installed apps, use Chrome to visit http://arifbutt.me
- Now in order to install and run our HelloWorld app, we just need to click the run app icon at the top on the title bar of Android Studio (shown in the top right image in a red square).
- Once you do that your app will appear on the AVD as shown in the right bottom screenshot. Click the "Click Me" button and it will change the text from "Hello World!" to "Welcome Android!" ☺



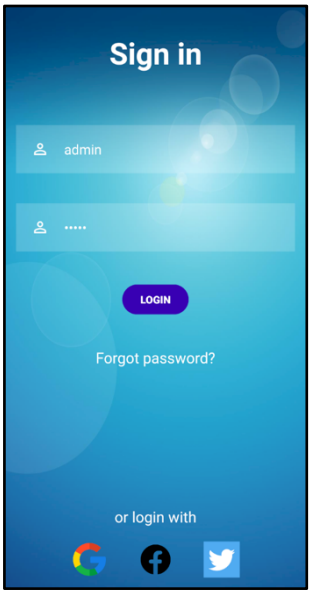
Option 2: (Running on a Physical Android Device): For this you need to ensure that your development machine and Android device are on the same Wi-Fi network, and there are no firewall rules blocking port 5555 on either your phone or your Windows machine. Now on your physical device

- **Method 1 (Manual APK Installation):** Copy the APK file from your Windows machine to your virtual Kali Linux machine inside `/var/www/html/` directory. Start `apache2` service on Kali. Now from your physical Android device open a browser and give the IP address of Kali and download/install the APK file.
- **Method 2 (USB Connection):** Connect your device with your Windows machine using a USB cable and your device will appear inside Android Studio's Device Manager and will run directly on your device. But for this you have to enable USB debugging on your device by performing following steps:
  - ⇒ Enable Developer options: Go to Settings > About phone, tap OS ver 7 times to enable developer options.
  - ⇒ Enable USB Debugging: Go to Settings > Additional Settings > Developer options >Turn on USB debugging. It will warn you that this will enable installing apps w/o your permission, monitor and record screen content, access and edit files via computer and so on. Click OK
  - ⇒ Enable Wireless Debugging: Go to Settings > Additional Settings > Developer options > Wireless debugging > Turn on Wireless debugging.
  - ⇒ Check IP of your device: Go to Settings > Wi-Fi > Your NW.
- **Method 3 (Use ADB):** If you are running your lab environment using Android Debug Bridge, having your physical or virtual Android device connected. You can simply use the `adb install app-debug.apk` to install it on your device. More on this later ☺

Simulator	Emulator
Mimic the behaviour of target device at a higher-level w/o replicating the underlying h/w or s/w.	Mimic not only the behaviour of target device, but also its underlying h/w like CPU, GPU, touch screen, camera, mic etc.
UI/UX testing, basic app functionality.	Full system behavior, hardware interaction.
Simulators are less accurate but faster.	Emulators are more accurate but slower.
Less accurate for hardware-specific testing	More accurate for hardware-specific testing
Apple's Xcode includes an iOS simulator, which mimics the iPhone/iPAd interface and runs apps, but it does not simulate the actual h/w components of iPhone.	Google's Android studio includes an Android Emulator, which can emulate various Android device configurations, such as different screen sizes, resolutions and OS versions. Genymotion is an Android emulator, that rely on QEMU (Quick Emulator) to emulate h/w and OS interactions at a low level.

To Do:

Build a basic Login app having a single screen as shown in this screenshot. While designing the interface, the background image, and the three social media icons need to be copied inside the `res/drawable/` folder inside the project Window of your Android Studio. Inside the `onClick()` method, you need to compare the credentials, and can display appropriate message of Success/Failure using the `Toast.makeTest()` method. Finally, other than testing your app inside Android Studio on AVD, you should also install it on your physical Android device, and later publish it on Play Store as well ☺



Disclaimer

The series of handouts distributed with this course are only for educational purposes. Any actions and or activities related to the material contained within this handout is solely your responsibility. The misuse of the information in this handout can result in criminal charges brought against the persons in question. The authors will not be held responsible in the event any criminal charges be brought against any individuals misusing the information in this handout to break the law.