Sommario

[Introduction: Mobile apps and the whole ecosystem 3](#_Toc147313010)

[Android Architecture 4](#_Toc147313011)

# Introduction: Mobile apps and the whole ecosystem

The first smartphone to be successfully introduced was the iPhone in 2007, marking a significant shift in the way we interact with technology. Subsequently, Google introduced Android in 2008, and Microsoft entered the market with Windows Phone in 2010.

One of the core aspects of the mobile ecosystem is mobile apps, which are the driving force behind the popularity of smartphones. Mobile apps provide developers with a platform to make their work accessible to a wide audience, but there must be incentives for developers to create and distribute them. Infact, smartphones can run “apps” as functionality, which can be delivered in form of games, apps or whatever functionality the user finds useful in a particular moment. The logic is *user-centric*.

To maintain a vibrant ecosystem of mobile apps, there must be incentives for developers to invest their time and resources in creating and maintaining them.

These incentives can be multifaceted:

* **Monetization:** Developers can earn revenue through various means, including selling apps, offering in-app purchases, displaying ads, or implementing subscription models.
* **Market Reach:** The widespread adoption of smartphones means developers have access to a vast global audience, making it an attractive platform for reaching users.
* **Innovation:** Mobile app development offers opportunities for innovation and creativity, allowing developers to bring new ideas to life.
* **Portfolio Building:** Developing successful mobile apps can enhance a developer's portfolio and career prospects.
* **Community and Recognition:** Building popular apps can lead to recognition and a sense of accomplishment within the developer community.

Incentives vary depending on the platform (iOS, Android, or others) and the business model chosen by the developer.

Let’s talk about Apple and its operating system iOS, the operating system that powers iPhones and iPads, is a closed-source system (source code of iOS is not publicly available or open for modification by anyone outside of Apple). This allows Apple to have full control over the software, allowing them to maintain a high level of security and consistency across all devices, otherwise not possible in a system as vastly spread across multiple brands and devices like Android is.

Apple's ecosystem is often described as a "*walled garden*" because it tightly controls what can be installed and run on its devices. Infact, Apple devices only mount iOS (in case of mobile) or macOS (in case of desktop) as their operating system and apps for iOS can only be distributed through the Apple App Store, and they undergo a rigorous review process to ensure quality, security, and compliance with Apple's guidelines, making the user experience as seamless as possible.

While it is still possible, since the first Apple versions it has become harder and harder the act of *jailbreaking* a device (process of circumventing Apple's restrictions to install unauthorized apps or modify the operating system, usually third-party services, and stuff like that); doing so can void warranties and introduce security risks not handled by the manufacturer.

Apple has a strategy that works: keeping it tightly monitored and secure has become a sort of status symbol, where the Apple devices hold a huge market share, because of the quality of the products themselves.

We then have a different approach, the Google one: an Open ecosystem. Google's involvement in the development of Android began in 2005 when they acquired Android Inc., a company that had been working on the Android operating system since 2003. Google recognized the potential of a mobile operating system and aimed to create a competitive alternative to iOS.

To compete with Apple, Google formed the "Open Handset Alliance" in 2007, a consortium of 84 companies collaborating on the development and promotion of the Android platform. This collaborative approach allowed various hardware manufacturers, carriers, and software developers to contribute to and benefit from the Android ecosystem.

One of the defining features of Android is its open-source nature. The *Android Open-Source Project (AOSP)* provides the source code for the Android operating system, allowing developers to modify and customize it. This open nature makes Android highly adaptable and suitable for a wide range of devices, including those not manufactured by Google (which are the minority, considering the Pixel devices).

Developers working with Android enjoy a high degree of flexibility:

* **App Distribution:** Unlike iOS, Android allows "sideloading" apps, which means users can install apps from sources other than the official Google Play Store (third party services).
* **App Modification:** Android's open nature makes it relatively easy to inspect, modify, and reverse engineer apps. While this offers flexibility, it can also present security challenges.
* **Customization:** Developers can create custom versions of Android, known as custom ROMs, to cater to specific user preferences or needs.

Unlike iOS, the jailbreaking, called *rooting* for Android devices, is much easier and well-defined, allowing the users to gain more control and install custom software is relatively straightforward. While this provides users with more freedom, it also introduces security risks and potential vulnerabilities.

This approach ensured Google’s Android a widespread adoption, making it the most popular mobile operating system globally in terms of market share. Remember also, this diversity in hardware and software options provides users with choices but can also lead to fragmentation issues, where not all devices receive timely updates and security patches.

We also quote Microsoft and its entry into the mobile ecosystem with Windows Phone was marked by a unique strategy, which primarily targeted the enterprise market (having the same UI across all devices, from tablets, smartphones, PCs, called Metro). However, despite offering promising devices, Microsoft faced challenges that eventually led to the fading of Windows Phone.

Windows Phone devices were designed with features and capabilities specifically tailored to meet the needs of businesses and professionals. These features included robust security options, integration with Microsoft Office, and a user interface optimized for productivity. While promising, it wasn’t enough: there were no apps to keep users interested in the system (very few compared to the other OSes), it arrived in a complicated moment and possibly quite late in the smartphone panorama, having very few manufacturers to support it (Nokia, Acer, Toshiba, ZTE) and very few updates.

This below is the recap table on Android vs iOS:

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

# Lesson 2: Android Architecture

Immagine che contiene testo, schermata, software, design

Descrizione generata automaticamenteAndroid's architecture is designed as a layered stack, with each layer responsible for specific functions and interactions within the operating system. Android's architecture consists of the following key layers:

1. **Linux Kernel:**
   * At the foundation of Android is the Linux Kernel. It provides core services such as hardware abstraction, memory management, process management, and drivers for device hardware components. It handles all the applications logic and functionalities.
   * The Linux Kernel acts as an intermediary between the hardware and the upper layers of the Android stack, ensuring that the hardware is utilized efficiently.
2. **Hardware Abstraction Layer (HAL):**
   * Above the Linux Kernel is the Hardware Abstraction Layer (HAL), which provides a standardized interface for Android to interact with various hardware components. It can be considered as the connection between the software and hardware components.
   * The HAL abstracts hardware-specific details, enabling Android to run on a wide range of devices with different hardware configurations.
3. **Android Runtime (ART):**
   * Android's runtime environment changed from the Dalvik Virtual Machine (DVM) to the Android Runtime (ART) in later versions.
   * ART is responsible for executing Android application code written in Java or Kotlin. It uses Ahead-of-Time (AOT) compilation to convert bytecode into native machine code, improving app performance and efficiency.
   * It’s based on the Core Libraries, which provide the main features for executing an application and then the ART (Android Runtime) itself to run the app (this one was introduced in recent Android Versions, because at the birth of Android, there was no such a thing; this bases itself on the Dalvik Virtual Machine, to have it running quickly valuing performance over the Java Virtual Machine)
   * All the processes in Android are completely isolated, unless they communicate with the bridged communication offered by the OS itself.
4. **Native Libraries:**
   * Native libraries consist of pre-compiled libraries written in C and C++ that are essential for Android's core functionalities. This again is a choice based on performance reasons.
   * These libraries include components like the Surface Manager for graphics management, the Media Framework for multimedia support, and SQLite for database operations.
5. **Application Framework Layer (Java API Framework):**
   * The Application Framework layer provides developers with a set of high-level APIs and tools to build Android applications.
   * It includes various managers and services, such as:
     1. Activity Manager:

* Responsible for managing the lifecycle of Android applications, including launching, pausing, and stopping activities (UI components).
* Manages the back stack of activities, ensuring proper navigation and user interaction.
  + 1. Content Provider
* Enables data sharing and data access between different apps on the device.
* Provides a standardized interface to access and modify data from databases, files, or other sources.
* Ensures data security and access control through permissions.
* Other managers include the Intent resolver, allowing to request actions and data, the View System, which manages and renders UI components, the Package Manager, managing the installation and permissions of apps, the Notification one for events and the Resource one.

1. **System Apps and User Apps:**
   * Android includes a set of system apps that come pre-installed on the device (on the system partition), such as the dialer, messaging, and settings apps. They cannot be uninstalled or pause their execution. They are considered more secure being part of the Android stock image. They have a subset of permissions, called previous permissions, which are a superpart given to system apps.
   * User-installed apps are also part of this layer and are built on top of the Android framework.

Android apps are typically composed of several loosely coupled components that work together to provide various functionalities. These components include:

* + Activities: These represent individual screens or user interface elements within an app. Activities are responsible for presenting the app's user interface to the user.
  + Services: Services run in the background and perform tasks that don't require a user interface, such as downloading data, playing music, or handling push notifications.
  + Broadcast Receivers: These components listen for system-wide or app-specific events (broadcasts) and respond to them. For example, an app might have a broadcast receiver that responds to incoming text messages.
  + Content Providers: Content providers facilitate data sharing between apps. They allow one app to access and modify data stored by another app, such as contact information or app-specific settings.

We then specify the following:

1. **Privilege Separation (Sandbox):**
   * Android enforces a security model that separates each app into its own isolated environment, often referred to as a "sandbox." This dedicates a memory specifically for the Android execution of a specific application and none of the other apps can interact with it.
   * This sandboxing ensures that an app's data and code are isolated from other apps, enhancing security, and preventing unauthorized access.
2. **Principle of Least Privilege (Permissions):**
   * Android follows the principle of least privilege, which means that apps should only have access to the resources and capabilities they need to function.
   * To achieve this, Android uses a permissions system. When you install an app, it may request certain permissions (e.g., access to your camera, location, or contacts). These permissions specify what resources or data the app can access.
   * Users have the option to grant or deny these permissions during installation or while using the app. This allows users to control the level of access apps have to their device's resources. The unwritten rule is to have a bar minimum of required permissions.

A very important file is he AndroidManifest.xml file is a crucial component of every Android app. It contains essential information about the app's structure, components, permissions, and other metadata. This file serves as a blueprint for the Android operating system, detailing how the app should be installed, executed, and interact with the device and other apps.

This file is the first that is parsed and allows retrieving all of the app’s info. Here's an overview of the key elements specified in the AndroidManifest file:

1. **Package Name:**
   * The package name is a unique identifier for the app. It is defined using the **package** attribute in the **<manifest>** element, such as **com.example.myapp**. We can’t have then apps with the same package name on Play Store. It can’t be updated afterwards as name. In this case you have to update the whole application.
   * Package names are crucial for distinguishing apps and ensuring they don't conflict with each other on a device. From the signature of the apps themselves, we can distinguish between the versions of apps, whether they are updated or old, given it will be the same package name by the same developer.
   * They often follow a reverse domain naming convention, such as **com.facebook.katana**
2. **Components:**
   * The AndroidManifest file defines the app's components, including:
     + **Activities:** These are the app's user interface screens. Each activity specifies its name, intent filters, and which activity should launch when the app is started.
     + **Services:** Background processes or tasks that run independently of the user interface.
     + **Broadcast Receivers:** Components that listen for system or app-specific events and respond accordingly.
     + **Content Providers:** Components that manage access to app-specific data or shared data with other apps.
3. **Permissions:**
   * The AndroidManifest file lists the permissions the app requires to access specific device resources or perform certain actions.
   * Permissions include access to the camera, location, contacts, internet, and more.
   * When users install an app, they are informed about the requested permissions and can choose to grant or deny them.
4. **Intent Filters:**
   * Intent filters define how an app responds to external events or user actions. They specify which components should handle specific intents (e.g., opening a web link or receiving a text message).
   * Intent filters allow apps to interact with each other and with the system.
5. **App Metadata:**
   * The AndroidManifest file includes metadata about the app, such as its version number, label (display name), icon, and the minimum Android API level required to run the app.
6. **Activities Launch Mode:**
   * The manifest can specify the launch mode for activities, determining how they interact with the task stack and back stack.
7. **App Permissions:**
   * Apps can declare custom permissions in the AndroidManifest file, which can be used to control access to specific app features or data by other apps.
8. **Application Theme and Style:**
   * The file can specify the default theme and style for the app's user interface.

The AndroidManifest file is essential for both the Android operating system and the Google Play Store. It helps the system understand how to manage and execute the app and ensures that the app complies with security and compatibility requirements. It also plays a role in app distribution on the Google Play Store by providing necessary information for listing and categorizing the app.

Unlike traditional console-based programs, Android apps do not have a central "*main*" function. Instead, they consist of various components, and the Android operating system manages their lifecycle. Android apps primarily interact with users through a graphical user interface (GUI). Users interact with elements such as buttons, text fields (EditText), checkboxes, and more.

When you click an app icon in the lanucher, we link the MainActivity of an application, then using its UI and its components. We have no command line for input, only in a debugging phase.

Many aspects of Android app development are event-driven. This means that actions and responses are triggered by events or user interactions. The process often involves two steps:

* **Registering a Listener:** Developers register event listeners (also known as event handlers or callbacks) for specific UI elements or system events. For example, you might register a click listener for a Button.
* **Callback Invocation:** When the associated event occurs (e.g., a button is clicked), the registered callback is invoked, allowing the app to respond to the event. This is where developers write the code to handle the event.

An Activity in Android is a fundamental component that represents a single screen with a user interface. It serves as the entry point for user interaction within an app and encapsulates the user interface and associated logic for a specific task or screen. We can define a main one, but we also do have multiple running at the same time, each with a lifecycle of various states, used to manage the behaviour of said ones during user interaction.

Below we have the Android Activity Lifecycle:

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

To be complete, each one will be briefly described above (not required, but just to give more notions):

1. **onCreate()**
   * This method is called when the activity is first created.
   * You should perform one-time initialization here, such as setting up the user interface and initializing variables.
   * The activity is not visible to the user at this stage.
2. **onStart()**
   * Called when the activity becomes visible to the user but is not yet in the foreground (i.e., not yet interacting with the user).
   * This is a good place to start or resume background threads or services that should be active when the activity is visible.
3. **onResume()**
   * This method is called when the activity comes into the foreground and becomes interactive.
   * It's a suitable place to start animations, acquire resources, and register listeners for user input.
4. **onPause()**
   * Called when the activity is about to lose focus but is still partially visible.
   * You should release resources that are no longer needed, stop animations, and unregister listeners to conserve system resources.
5. **onStop()**
   * Called when the activity is no longer visible to the user.
   * You can release resources and perform cleanup operations that are not needed when the activity is not visible.
6. **onRestart()**
   * This method is called when the activity is restarting after being stopped but not destroyed.
   * It provides an opportunity to reinitialize components that were stopped in **onStop()**.
7. **onDestroy()**
   * Called when the activity is being destroyed or is about to be removed from memory.
   * You should perform final cleanup here, release resources, and unregister any remaining listeners.
8. **onSaveInstanceState(Bundle outState)**
   * Called when the activity is about to be destroyed but might be recreated in the future.
   * You can save critical data to the **outState** bundle to restore the activity's state when it is recreated.
9. **onRestoreInstanceState(Bundle savedInstanceState)**
   * Called after **onCreate()** if the activity is being recreated from a previously saved state.
   * You can retrieve the saved data from the **savedInstanceState** bundle to restore the activity's state.

A Service in Android is a component that performs tasks in the background, independently of the user interface, and often for an extended period. Services are used when you need to execute operations that should continue running, even if the user switches between different activities or exits the app altogether.

Services are typically used for tasks that don't require a user interface, such as downloading files, playing music, handling push notifications, or monitoring sensors. They are suitable for tasks that should run persistently or for a long duration. Also, they do not provide any kind of UI.

Broadcast Receiver is a component in Android that listens for system-wide events or custom broadcast messages. When a relevant event occurs, the system delivers it to the registered broadcast receiver, allowing the app to respond to or process the event. They are designed to respond to events such as incoming SMS messages, battery state changes, screen on/off events, network connectivity changes, and more. They serve as event handlers for system-level and app-specific events.

Content Provider in Android is an object that manages and allows controlled access to a shared set of app data. Content Providers provide a high-level API that allows other apps and services to query, interact with, and potentially modify this data, even if they are from different applications. They allow other apps to share the same data in various formats (e.g., SQLite databases, files, or remote servers) through a consistent and structured interface.

(Approximately at 20:57 mark to see)