Android Application Fundamentals

Android apps can be written using Kotlin and the Java programming language, and C++ languages. The Android SDK tools compile your code along with any data and resource files into an APK or an Android App Bundle.
An Android package, which is an archive file with an .apk suffix, contains the contents of an Android app required at runtime, and it is the file that Android-powered devices use to install the app.
An Android App Bundle, which is an archive file with an .aab suffix, contains the contents of an Android app project, including some additional metadata that isn't required at runtime. An AAB is a publishing format and can't be installed on Android devices. It defers APK generation and signing to a later stage.
When distributing your app through Google Play, for example, Google Play's servers generate optimized APKs that contain only the resources and code that are required by the particular device requesting installation of the app.

Each Android app lives in its own security sandbox, protected by the following Android security features:

- **The Android operating system is a multi-user Linux system in which each app is a different user.**
- **By default, the system assigns each app a unique Linux user ID, which is used only by the system and is unknown to the app. The system sets permissions for all the files in an app so that only the user ID assigned to that app can access them.**
- **Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps.**
- **By default, every app runs in its own Linux process. The Android system starts the process when any of the app's components need to be executed, and then shuts down the process when it's no longer needed or when the system must recover memory for other apps.**

The Android system implements the principle of least privilege. That is, each app, by default, has access only to the components that it requires to do its work and no more. This creates a very secure environment in which an app can't access parts of the system it is not given permission for.

However, there are ways for an app to share data with other apps and for an app to access system services:

- **It's possible to arrange for two apps to share the same Linux user ID, in which case they are able to access each other's files. To conserve system resources, apps with the same user ID can also arrange to run in the same Linux process and share the same VM. The apps must also be signed with the same certificate.**
- **An app can request permission to access device data such as the device's location, camera, and Bluetooth connection. The user has to explicitly grant these permissions. For more information about permissions, see Permissions on Android.**

The rest of this document introduces the following concepts:

- **The core framework components that define your app.**
- **The manifest file in which you declare the components and the required device features for your app.**
- **Resources that are separate from the app code and that let your app gracefully optimize its behavior for a variety of device configurations.**

App components

App components are the essential building blocks of an Android app. Each component is an entry point through which the system or a user can enter your app. Some components depend on others.

There are four types of app components:

- **Activities**
- **Services**
- **Broadcast receivers**
- **Content providers**

Each type serves a distinct purpose and has a distinct lifecycle that defines how a component is created and destroyed.

Activities

An activity is the entry point for interacting with the user. It represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others.

A different app can start any one of these activities if the email app allows it. For example, a camera app might start the activity in the email app for composing a new email to let the user share a picture.

An activity facilitates the following key interactions between system and app:

- **Keeping track of what the user currently cares about—what is on-screen—so that the system keeps running the process that is hosting the activity.**
- **Knowing which previously used processes contain stopped activities the user might return to and prioritizing those processes more highly to keep them available.**
- **Helping the app handle having its process killed so the user can return to activities with their previous state restored.**
- **Providing a way for apps to implement user flows between each other, and for the system to coordinate these flows. The primary example of this is sharing.**

Services

A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface.

For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it to interact with it.

There are two types of services that tell the system how to manage an app: started services and bound services.

1. Started services tell the system to keep them running until their work is completed. This might be to sync some data in the background or play music even after the user leaves the app. Syncing data in the background or playing music represent different types of started services, which the system handles differently:

- Music playback is something the user is directly aware of, and the app communicates this to the system by indicating that it wants to be in the foreground, with a notification to tell the user that it is running. In this case, the system prioritizes keeping that service's process running, because the user has a bad experience if it goes away.
- A regular background service is not something the user is directly aware of, so the system has more freedom in managing its process. It might let it be killed, restarting the service sometime later, if it needs RAM for things that are of more immediate concern to the user.

2. Bound services run because some other app (or the system) has said that it wants to make use of the service. A bound service provides an API to another process, and the system knows there is a dependency between these processes. So if process A is bound to a service in process B, the system knows that it needs to keep process B and its service running for A.

Because of their flexibility, services are useful building blocks for all kinds of higher-level system concepts. Live wallpapers, notification listeners, screen savers, input methods, accessibility services, and many other core system features are all built as services that applications implement and the system binds to when they run.

Broadcast receivers

A broadcast receiver is a component that lets the system deliver events to the app outside of a regular user flow so the app can respond to system-wide broadcast announcements. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running.

So, for example, an app can schedule an alarm to post a notification to tell the user about an upcoming event. Because the alarm is delivered to a BroadcastReceiver in the app, there is no need for the app to remain running until the alarm goes off.

Many broadcasts originate from the system, like a broadcast announcing that the screen is turned off, the battery is low, or a picture is captured. Apps can also initiate broadcasts, such as to let other apps know that some data is downloaded to the device and is available for them to use.

Although broadcast receivers don't display a user interface, they can create a status bar notification to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a *gateway* to other components and is intended to do a very minimal amount of work.

Content providers

A content provider manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. Through the content provider, other apps can query or modify the data, if the content provider permits it.

For example, the Android system provides a content provider that manages the user's contact information. Any app with the proper permissions can query the content provider, such as using ContactsContract.Data, to read and write information about a particular person.

It is tempting to think of a content provider as an abstraction on a database, because there is a lot of API and support built in to them for that common case. However, they have a different core purpose from a system-design perspective.

Content providers are also useful for reading and writing data that is private to your app and not shared.

Android API

The Android API (Application Programming Interface) is a set of rules, protocols, and tools for building software applications on the Android platform. It provides a way for developers to interact with the underlying Android operating system and access various functionalities, such as hardware sensors, network services, and user interface components.

Android versions are identified by a version number and a codename. Each version of Android comes with a specific API level, which represents the version of the Android API framework that is available to developers. API levels are essential for developers to ensure compatibility and target specific features and functionalities based on the Android version running on a device.

Android versions and their corresponding API levels:

1. Android 1.0: API Level 1

2. Android 1.1: API Level 2

3. Android 1.5 Cupcake: API Level 3

4. Android 1.6 Donut: API Level 4

5. Android 2.0/2.1 Eclair: API Levels 5-7

6. Android 2.2 Froyo: API Level 8

7. Android 2.3 Gingerbread: API Levels 9-10

8. Android 3.0 Honeycomb: API Level 11

9. Android 4.0 Ice Cream Sandwich: API Levels 14-15

10. Android 4.1-4.3 Jelly Bean: API Levels 16-18

11. Android 4.4 KitKat: API Level 19

12. Android 5.0-5.1 Lollipop: API Levels 21-22

13. Android 6.0 Marshmallow: API Level 23

14. Android 7.0-7.1 Nougat: API Levels 24-25

15. Android 8.0-8.1 Oreo: API Levels 26-27

16. Android 9 Pie: API Level 28

17. Android 10: API Level 29

18. Android 11: API Level 30

19. Android 12: API Level 31,32

20. Android 13: API Level 33

21. Android 14: API Level 34

Each API level introduces new features, improvements, and changes to the Android platform, and developers can target specific API levels based on their application's requirements and the features they want to utilize. This helps ensure that apps remain compatible with a wide range of Android devices while taking advantage of the latest capabilities offered by the platform. https://developer.android.com/tools/releases/platforms