Developers

Application Fundamentals

https://developer.android.com/guide/components/fundamentals

Android apps can be written using Kotlin, 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 behaviour 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. The following sections describe the four types of app components.

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.

You implement an activity as a subclass of the **`Activity`** class. For more information about the `Activity` class, see Introduction to activities.

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**.

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.

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. Further, if process A is something the user cares about, then it knows to treat process B as something the user also cares about.

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.

A service is implemented as a subclass of **`Service`**. For more information about the `Service` class, see the Services overview.

> ➢ **Note:** If your app targets Android 5.0 (API level 21) or higher, use the **`JobScheduler`** class to schedule actions. JobScheduler has the advantage of conserving battery by optimally scheduling jobs to reduce power consumption and by working with the Doze API. For more information about using this class, see the **`JobScheduler`** reference documentation.

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.

For instance, a broadcast receiver might schedule a **`JobService`** to perform some work based on an event using **`JobScheduler`**. Broadcast receivers often involve apps interacting with each other, so it's important to be aware of the security implications when setting them up.

A broadcast receiver is implemented as a subclass of **`BroadcastReceiver`**, and each broadcast is delivered as an **`Intent`** object. For more information, see the **`BroadcastReceiver`** class.

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.

To the system, a content provider is an entry point into an app for publishing named data items, identified by a URI scheme. Thus, an app can decide how it wants to map the data it contains to a URI namespace, handing out those URIs to other entities which can in turn use them to access the data. There are a few particular things this lets the system do in managing an app:

- Assigning a URI doesn't require that the app remain running, so URIs can persist after their owning apps exit. The system only needs to make sure that an owning app is still running when it retrieves the app's data from the corresponding URI.

- These URIs also provide an important fine-grained security model. For example, an app can place the URI for an image it has on the clipboard, but leave its content provider locked up so that other apps cannot freely access it. When a second app attempts to access that URI on the clipboard, the system can let that app access the data using a temporary *URI permission grant* so that it accesses the data only behind that URI, and nothing else in the second app.

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

A content provider is implemented as a subclass of **`ContentProvider`** and must implement a standard set of APIs that enable other apps to perform transactions. For more information, see the Content providers developer guide.

A unique aspect of the Android system design is that any app can start another app's component. For example, if you want the user to capture a photo with the device camera, there's probably another app that does that—and your app can use it instead of developing an activity to capture a photo yourself. You don't need to incorporate or even link to the code from the camera app. Instead, you can start the activity in the camera app that captures a photo. When complete, the photo is even returned to your app so you can use it. To the user, it seems as if the camera is actually a part of your app.

When the system starts a component, it starts the process for that app, if it's not already running, and instantiates the classes needed for the component. For example, if your app starts the activity in the camera app that captures a photo, that activity runs in the process that belongs to the camera app, not in your app's process. Therefore, unlike apps on most other systems, Android apps don't have a single entry point: there's no `main()` function.

Because the system runs each app in a separate process with file permissions that restrict access to other apps, your app can't directly activate a component from another app. However, the Android system can. To activate a component in another app, you deliver a message to the system that specifies your *intent* to start a particular component. The system then activates the component for you.

Activate components

An asynchronous message called an *intent* activates three of the four component types: activities, services, and broadcast receivers. Intents bind individual components to each other at runtime. You can think of them as the messengers that request an action from other components, whether the component belongs to your app or another.

An intent is created with an **`Intent`** object, which defines a message to activate either a specific component (an *explicit* intent) or a specific type of component (an *implicit* intent).

For activities and services, an intent defines the action to perform, such as to *view* or *send* something, and might specify the URI of the data to act on, among other things that the component being started might need to know.

For example, an intent might convey a request for an activity to show an image or to open a web page. In some cases, you can start an activity to receive a result, in which case the activity also returns the result in an `Intent`. You can also issue an intent to let the user pick a personal contact and have it returned to you. The return intent includes a URI pointing to the chosen contact.

For broadcast receivers, the intent defines the broadcast announcement. For example, a broadcast to indicate that the device battery is low includes only a known action string that indicates *battery is low*.

Unlike activities, services, and broadcast receivers, content providers are activated when targeted by a request from a **`ContentResolver`**. The content resolver handles all direct transactions with the content provider, and the component performing transactions with the provider calls methods on the `ContentResolver` object. This leaves a layer of abstraction for security reasons between the content provider and the component requesting information.

There are separate methods for activating each type of component:

- You can start an activity or give it something new to do by passing an `Intent` to **`startActivity()`** or, when you want the activity to return a result, **`startActivityForResult()`**.
- On Android 5.0 (API level 21) and higher, you can use the **`JobScheduler`** class to schedule actions. For earlier Android versions, you can start a service or give new instructions to an ongoing service by passing an `Intent` to **`startService()`**. You can bind to the service by passing an `Intent` to **`bindService()`**.
- You can initiate a broadcast by passing an `Intent` to methods such as **`sendBroadcast()`** or **`sendOrderedBroadcast()`**.
- You can perform a query to a content provider by calling **`query()`** on a `ContentResolver`.

For more information about using intents, see the Intents and Intent Filters document. The following documents provide more information about activating specific components: Introduction to activities, Services overview, `BroadcastReceiver`, and Content providers.

The manifest file

Before the Android system can start an app component, the system must know that the component exists by reading the app's *manifest file*, `AndroidManifest.xml`. Your app declares all its components in this file, which is at the root of the app project directory.

The manifest does a number of things in addition to declaring the app's components, such as the following:

- Identifies any user permissions the app requires, such as internet access or read-access to the user's contacts.
- Declares the minimum API level required by the app, based on which APIs the app uses.
- Declares hardware and software features used or required by the app, such as a camera, Bluetooth services, or a multitouch screen.
- Declares API libraries the app needs to be linked against (other than the Android framework APIs), such as the Google Maps library.

Declare components

The primary task of the manifest is to inform the system about the app's components. For example, a manifest file can declare an activity as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application android:icon="@drawable/app_icon.png" ... >
        <activity android:name="com.example.project.ExampleActivity"
                  android:label="@string/example_label" ... >
        </activity>
        ...
    </application>
</manifest>
```

In the `<application>` element, the `android:icon` attribute points to resources for an icon that identifies the app.

In the `<activity>` element, the `android:name` attribute specifies the fully qualified class name of the `Activity` subclass, and the `android:label` attribute specifies a string to use as the user-visible label for the activity.

You must declare all app components using the following elements:

- `<activity>` elements for activities
- `<service>` elements for services
- `<receiver>` elements for broadcast receivers
- `<provider>` elements for content providers

Activities, services, and content providers that you include in your source but don't declare in the manifest aren't visible to the system and, consequently, can never run. However, broadcast receivers can either be declared in the manifest or created dynamically in code as `BroadcastReceiver` objects and registered with the system by calling `registerReceiver()` .

For more about how to structure the manifest file for your app, see the App manifest overview.

Declare component capabilities

As discussed in the Activate components section, you can use an `Intent` to start activities, services, and broadcast receivers. You do this by explicitly naming the target component, using the component class name, in the intent. You can also use an implicit intent, which describes the type of action to perform and, optionally, the data you want to perform the action on. An implicit intent lets the system find a component on the device that can perform the action and start it. If there are multiple components that can perform the action described by the intent, the user selects which one to use.

> ! **Caution:** If you use an intent to start a `Service`, make sure that your app is secure by using an explicit intent. Using an implicit intent to start a service is a security hazard, because you can't be certain what service responds to the intent and the user can't see which service starts. Beginning with Android 5.0 (API level 21), the system throws an exception if you call `bindService()` with an implicit intent. Don't declare intent filters for your services.

The system identifies the components that can respond to an intent by comparing the intent received to the *intent filters* provided in the manifest file of other apps on the device.

When you declare an activity in your app's manifest, you can optionally include intent filters that declare the capabilities of the activity so it can respond to intents from other apps. You do this by adding an `<intent-filter>` element as a child of the component's declaration element.

For example, if you build an email app with an activity for composing a new email, you can declare an intent filter to respond to "send" intents to send a new email, as shown in the following example:

```
<manifest ... >
    ...
    <application ... >
        <activity android:name="com.example.project.ComposeEmailActivity">
            <intent-filter>
                <action android:name="android.intent.action.SEND" />
                <data android:type="*/*" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

If another app creates an intent with the `ACTION_SEND` action and passes it to `startActivity()`, the system might start your activity so the user can draft and send an email.

For more about creating intent filters, see the Intents and Intent Filters document.

Declare app requirements

There are a variety of devices powered by Android, and not all of them provide the same features and capabilities. To prevent your app from being installed on devices that lack features needed by your app, it's important that you clearly define a profile for the types of devices your app supports by declaring device and software requirements in your manifest file.

Most of these declarations are informational only. The system doesn't read them, but external services such as Google Play do read them to provide filtering for users when they search for apps from their device.

For example, suppose your app requires a camera and uses APIs introduced in Android 8.0 (API level 26). You must declare these requirements. The values for `minSdkVersion` and `targetSdkVersion` are set in your app module's `build.gradle` file:

```
android {
  ...
  defaultConfig {
    ...
    minSdkVersion 26
    targetSdkVersion 29
  }
}
```

> **Note:** Don't set `minSdkVersion` and `targetSdkVersion` directly in the manifest file, since they are overwritten by Gradle during the build process. For more information, see Specify API level requirements.

You declare the camera feature in your app's manifest file:

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera.any"
                  android:required="true" />
    ...
</manifest>
```

With the declarations shown in these examples, devices that do *not* have a camera or have an Android version *lower* than 8.0 can't install your app from Google Play. However, you can also declare that your app uses the camera, but does not *require* it. To do so, you set the **`required`** attribute to `false`, check at runtime whether the device has a camera, and disable any camera features as needed.

More information about how you can manage your app's compatibility with different devices is provided in the Device compatibility overview.

App resources

An Android app is composed of more than just code. It requires resources that are separate from the source code, such as images, audio files, and anything relating to the visual presentation of the app. For example, you can define animations, menus, styles, colors, and the layout of activity user interfaces with XML files.

Using app resources makes it easy to update various characteristics of your app without modifying code. Providing sets of alternative resources lets you optimize your app for a variety of device configurations, such as different languages and screen sizes.

For every resource that you include in your Android project, the SDK build tools define a unique integer ID, which you can use to reference the resource from your app code or from other resources defined in XML. For example, if your app contains an image file named `logo.png` (saved in the `res/drawable/` directory), the SDK tools generate a resource ID named `R.drawable.logo`. This ID maps to an app-specific integer, which you can use to reference the image and insert it in your user interface.

One of the most important aspects of providing resources separate from your source code is the ability to provide alternative resources for different device configurations.

For example, by defining UI strings in XML, you can translate the strings into other languages and save those strings in separate files. Then Android applies the appropriate language strings to your UI based on a language *qualifier* that you append to the resource directory's name, such as `res/values-fr/` for French string values, and the user's language setting.

Android supports many qualifiers for your alternative resources. The qualifier is a short string that you include in the name of your resource directories to define the device configuration those resources are used for.

For example, you can create different layouts for your activities depending on the device's screen orientation and size. When the device screen is in portrait (tall) orientation, you might want a layout with buttons arranged vertically, but when the screen is in landscape (wide) orientation, you might want the buttons aligned horizontally. To change the layout depending on the orientation, you can define two layouts and apply the appropriate qualifier to each layout's directory name. Then, the system automatically applies the appropriate layout depending on the current device orientation.

For more information about the different kinds of resources you can include in your application and how to create alternative resources for different device configurations, read the App resources overview. To learn more about best practices and designing robust, production-quality apps, see the Guide to app architecture.