Application Fundamentals

Android apps can be written using Kotlin, Java, and C++ languages. The Android SDK tools compile your code along with any data and resource files into an APK, an *Android package*, which is an archive file with an `.apk` suffix. One APK file contains all the contents of an Android app and is the file that Android-powered devices use to install 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 (the ID 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 cannot access parts of the system for which it is not given permission. 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 user's contacts, SMS messages, the mountable storage (SD card), camera, and Bluetooth. The user has to explicitly grant these permissions. For more information, see Working with System Permissions (https://developer.android.com/training/permissions/index.html).

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 allow your app to 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 different types of app components:

- Activities
- Services
- Broadcast receivers
- Content providers

Each type serves a distinct purpose and has a distinct lifecycle that defines how the 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. As such, a different app can start any one of these activities if the email app allows it. For example, a camera app can start the activity in the email app that composes new mail to allow the user to 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) to ensure that the system keeps running the process that is hosting the activity.
> - Knowing that previously used processes contain things the user may return to (stopped activities), and thus more highly prioritize keeping those processes around.

- 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 most classic example here being share.)

You implement an activity as a subclass of the `Activity` (https://developer.android.com/reference/android/app/Activity.html) class. For more information about the `Activity` (https://developer.android.com/reference/android/app/Activity.html) class, see the Activities (https://developer.android.com/guide/components/activities.html) developer guide.

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 in order to interact with it. There are actually two very distinct semantics services tell the system about how to manage an app: Started services tell the system to keep them running until their work is completed. This could 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 also represent two different types of started services that modify how the system handles them:

- Music playback is something the user is directly aware of, so the app tells the system this by saying it wants to be foreground with a notification to tell the user about it; in this case the system knows that it should try really hard to keep that service's process running, because the user will be unhappy if it goes away.
- A regular background service is not something the user is directly aware as running, so the system has more freedom in managing its process. It may allow it to be killed (and then 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. This is basically the service providing an API to another process. The system thus knows there is a dependency between these processes, so if process A is bound to a service in process B, it 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 also knows to treat process B as something the user also cares about. Because of their flexibility (for better or worse), services have turned out to be a really useful building block 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 should be running.

A service is implemented as a subclass of `Service` (https://developer.android.com/reference/android/app/Service.html). For more information about the `Service` (https://developer.android.com/reference/android/app/Service.html) class, see the Services (https://developer.android.com/guide/components/services.html) developer guide.

★ **Note:** If your app targets Android 5.0 (API level 21) or later, use the `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html) 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 (https://developer.android.com/training/monitoring-device-state/doze-standby.html) API. For more information about using this class, see the `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html) reference documentation.

Broadcast receivers

A *broadcast receiver* is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to 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... and by delivering that alarm to a BroadcastReceiver of the app, there is no need for the app to remain running until the alarm goes off. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Apps can also initiate broadcasts—for example, to let other apps know that some data has been downloaded to the device and is available for them to use. Although broadcast receivers don't display a user interface, they may create a status bar notification (https://developer.android.com/guide/topics/ui/notifiers/notifications.html) 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, it might schedule a `JobService` (https://developer.android.com/reference/android/app/job/JobService.html) to perform some

work based on the event with `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html)

A broadcast receiver is implemented as a subclass of `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) and each broadcast is delivered as an `Intent` (https://developer.android.com/reference/android/content/Intent.html) object. For more information, see the `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) 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 allows it. For example, the Android system provides a content provider that manages the user's contact information. As such, any app with the proper permissions can query the content provider, such as `ContactsContract.Data` (https://developer.android.com/reference/android/provider/ContactsContract.Data.html), 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 allows the system to do in managing an app:

- Assigning a URI doesn't require that the app remain running, so URIs can persist after their owning apps have exited. The system only needs to make sure that an owning app is still running when it has to retrieve 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 allow that app to access the data via a temporary *URI permission grant* so that it is allowed to access the data only behind that URI, but 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. For example, the Note Pad (https://developer.android.com/resources/samples/NotePad/index.html) sample app uses a content provider to save notes.

A content provider is implemented as a subclass of `ContentProvider` (https://developer.android.com/reference/android/content/ContentProvider.html) and must implement a standard set of APIs that enable other apps to perform transactions. For more information, see the Content Providers (https://developer.android.com/guide/topics/providers/content-providers.html) 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 simply 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 cannot directly activate a component from another app. However, the Android system can. To activate a component in another app, deliver a message to the system that specifies your *intent* to start a particular component. The system then activates the component for you.

Activating components

Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an *intent*. 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` (https://developer.android.com/reference/android/content/Intent.html) object, which defines a

message to activate either a specific component (explicit intent) or a specific *type* of component (implicit intent).

For activities and services, an intent defines the action to perform (for example, to *view* or *send* something) and may 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` (https://developer.android.com/reference/android/content/Intent.html). For example, you can 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 simply defines the announcement being broadcast. For example, a broadcast to indicate 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 not activated by intents. Rather, they are activated when targeted by a request from a `ContentResolver` (https://developer.android.com/reference/android/content/ContentResolver.html). The content resolver handles all direct transactions with the content provider so that the component that's performing transactions with the provider doesn't need to and instead calls methods on the `ContentResolver` (https://developer.android.com/reference/android/content/ContentResolver.html) object. This leaves a layer of abstraction between the content provider and the component requesting information (for security).

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` (https://developer.android.com/reference/android/content/Intent.html) to `startActivity()` (https://developer.android.com/reference/android/content/Context.html#startActivity(android.content.Intent))
  or `startActivityForResult()` (https://developer.android.com/reference/android/app/Activity.html#startActivityForResult(android.content.Intent, int))
  (when you want the activity to return a result).
- With Android 5.0 (API level 21) and later, you can use the `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html) 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` (https://developer.android.com/reference/android/content/Intent.html) to `startService()`

(https://developer.android.com/reference/android/content/Context.html#startService(android.content.Intent))
. You can bind to the service by passing an `Intent` (https://developer.android.com/reference/android/content/Intent.html) to `bindService()` (https://developer.android.com/reference/android/content/Context.html#bindService(android.content.Intent, android.content.ServiceConnection, int))
.

- You can initiate a broadcast by passing an `Intent` (https://developer.android.com/reference/android/content/Intent.html) to methods such as `sendBroadcast()` (https://developer.android.com/reference/android/content/Context.html#sendBroadcast(android.content.Intent))
, `sendOrderedBroadcast()` (https://developer.android.com/reference/android/content/Context.html#sendOrderedBroadcast(android.content.Intent, java.lang.String))
, or `sendStickyBroadcast()` (https://developer.android.com/reference/android/content/Context.html#sendStickyBroadcast(android.content.Intent))
.
- You can perform a query to a content provider by calling `query()` (https://developer.android.com/reference/android/content/ContentProvider.html#query(android.net.Uri, java.lang.String[], android.os.Bundle, android.os.CancellationSignal))
on a `ContentResolver` (https://developer.android.com/reference/android/content/ContentResolver.html).

For more information about using intents, see the Intents and Intent Filters (https://developer.android.com/guide/components/intents-filters.html) document. The following documents provide more information about activating specific components: Activities (https://developer.android.com/guide/components/activities.html), Services (https://developer.android.com/guide/components/services.html)`BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html), 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 must declare all its components in this file, which must be 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 (https://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels) 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 (http://code.google.com/android/add-ons/google-apis/maps-overview.html).

Declaring 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>`** (https://developer.android.com/guide/topics/manifest/application-element.html) element, the **`android:icon`** attribute points to resources for an icon that identifies the app.

In the **`<activity>`** (https://developer.android.com/guide/topics/manifest/activity-element.html) element, the **`android:name`** attribute specifies the fully qualified class name of the **`Activity`** (https://developer.android.com/reference/android/app/Activity.html) 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>`** (https://developer.android.com/guide/topics/manifest/activity-element.html) elements for activities.
- **`<service>`** (https://developer.android.com/guide/topics/manifest/service-element.html) elements for services.

- `<receiver>` (https://developer.android.com/guide/topics/manifest/receiver-element.html) elements for broadcast receivers.
- `<provider>` (https://developer.android.com/guide/topics/manifest/provider-element.html) elements for content providers.

Activities, services, and content providers that you include in your source but do not declare in the manifest are not visible to the system and, consequently, can never run. However, broadcast receivers can be either declared in the manifest or created dynamically in code as `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) objects and registered with the system by calling `registerReceiver()` (https://developer.android.com/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter))
.

For more about how to structure the manifest file for your app, see The AndroidManifest.xml File (https://developer.android.com/guide/topics/manifest/manifest-intro.html) documentation.

Declaring component capabilities

As discussed above, in Activating components (#ActivatingComponents), you can use an `Intent` (https://developer.android.com/reference/android/content/Intent.html) to start activities, services, and broadcast receivers. You can use an `Intent` (https://developer.android.com/reference/android/content/Intent.html) 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 upon which you'd like to perform the action. The implicit intent allows the system to 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` (https://developer.android.com/reference/android/app/Service.html), ensure that your app is secure by using an explicit (https://developer.android.com/guide/components/intents-filters.html#Types) intent. Using an implicit intent to start a service is a security hazard because you cannot be certain what service will respond to the intent, and the user cannot see which service starts. Beginning with Android 5.0 (API level 21), the system throws an exception if you call `bindService()` (https://developer.android.com/reference/android/content/Context.html#bindService(android.content.Intent, android.content.ServiceConnection, int))
with an implicit intent. Do not 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 can declare an intent filter for your component by adding an **`<intent-filter>`** (https://developer.android.com/guide/topics/manifest/intent-filter-element.html) 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 (in order 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`** (https://developer.android.com/reference/android/content/Intent.html#ACTION_SEND) action and passes it to **`startActivity()`** (https://developer.android.com/reference/android/app/Activity.html#startActivity(android.content.Intent))
, the system may start your activity so the user can draft and send an email.

For more about creating intent filters, see the Intents and Intent Filters (https://developer.android.com/guide/components/intents-filters.html) document.

Declaring 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 and the system does not read them, but external services such as Google Play do read them in order to provide filtering for users when they search for apps from their device.

For example, if your app requires a camera and uses APIs introduced in Android 2.1 (API Level (https://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels) 7), you must declare these as requirements in your manifest file as shown in the following example:

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera.any"
                  android:required="true" />
    <uses-sdk android:minSdkVersion="7" android:targetSdkVersion="19" />
    ...
</manifest>
```

With the declarations shown in the example, devices that do *not* have a camera or have an Android version *lower* than 2.1 cannot install your app from Google Play. However, you can declare that your app uses the camera, but does not *require* it. In that case, your app must set the **`required`** (https://developer.android.com/guide/topics/manifest/uses-feature-element.html#required) attribute to **`false`** and check at runtime whether the device has a camera and disable any camera features as appropriate.

More information about how you can manage your app's compatibility with different devices is provided in the Device Compatibility (https://developer.android.com/guide/practices/compatibility.html) document.

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 enables you to 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 different *qualifiers* for your alternative resources. The qualifier is a short string that you include in the name of your resource directories in order to define the device configuration for which those resources should be used. For example, you should create different layouts for your activities, depending on the device's screen orientation and size. When the device screen is in portrait orientation (tall), you might want a layout with buttons to be vertical, but when the screen is in landscape orientation (wide), the buttons could be aligned horizontally. To change the layout depending on the orientation, you can define two different 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 about the different kinds of resources you can include in your application and how to create alternative resources for different device configurations, read Providing Resources (https://developer.android.com/guide/topics/resources/providing-resources.html). To learn more about best practices and designing robust, production-quality apps, see Guide to App Architecture (https://developer.android.com/topic/libraries/architecture/guide.html).

Continue reading about:

Intents and Intent Filters
(https://developer.android.com/guide/components/intents-filters.html)

How to use the `Intent` (https://developer.android.com/reference/android/content/Intent.html)
APIs to activate app components, such as activities and services, and how to make your app components available for use by other apps.

You might also be interested in:

Device Compatibility
(https://developer.android.com/guide/practices/compatibility.html)

How Android works on different types of devices and an introduction to how you can optimize your app for each device or restrict your app's availability to different devices.

Activities
(https://developer.android.com/guide/components/activities.html)

How to create an instance of the `Activity` (https://developer.android.com/reference/android/app/Activity.html) class, which provides a distinct screen in your application with a user interface.

Providing Resources
(https://developer.android.com/guide/topics/resources/providing-resources.html)

How Android apps are structured to separate app resources from the app code, including how you can provide alternative resources for specific device configurations.

System Permissions
(https://developer.android.com/guide/topics/permissions/)

How Android restricts app access to certain APIs with a permission system that requires the user's consent for your app to use those APIs.

Last updated April 30, 2018.

Twitter
Follow @AndroidDev on Twitter

Google+
Follow Android Developers on Google+

YouTube
Check out Android Developers on YouTube