Data and file storage overview

Android provides several options for you to save your app data. The solution you choose depends on your specific needs, such as how much space your data requires, what kind of data you need to store, and whether the data should be private to your app or accessible to other apps and the user.

Following are the different data storage options available on Android:

* [Internal file storage](https://developer.android.com/guide/topics/data/data-storage#filesInternal): Store app-private files on the device file system.
* [External file storage](https://developer.android.com/guide/topics/data/data-storage#filesExternal): Store files on the shared external file system. This is usually for shared user files, such as photos.
* [Shared preferences](https://developer.android.com/guide/topics/data/data-storage#pref): Store private primitive data in key-value pairs.
* [Databases](https://developer.android.com/guide/topics/data/data-storage#db): Store structured data in a private database.

Except for some types of files on external storage, all these options are intended for app-private data—the data is not naturally accessible to other apps. If you want to share files with other apps, you should use the [FileProvider](https://developer.android.com/reference/android/support/v4/content/FileProvider.html) API. To learn more, read [Sharing Files](https://developer.android.com/training/secure-file-sharing/index.html).

If you want to expose your app's data to other apps, you can use a [ContentProvider](https://developer.android.com/reference/android/content/ContentProvider.html). Content providers give you full control of what read/write access is available to other apps, regardless of the storage medium you've chosen for the data (though it's usually a database). For more information, read [Content Providers](https://developer.android.com/guide/topics/providers/content-providers.html).

Internal storage

By default, files saved to the internal storage are private to your app, and other apps cannot access them (nor can the user, unless they have root access). This makes internal storage a good place for internal app data that the user doesn't need to directly access. The system provides a private directory on the file system for each app where you can organize any files your app needs.

When the user uninstalls your app, the files saved on the internal storage are removed. Because of this behavior, you should not use internal storage to save anything the user expects to persist independently of your app. For example, if your app allows users to capture photos, the user would expect that they can access those photos even after they uninstall your app. So you should instead save those types of files to the public external storage.

To learn more, read how to [save a file on internal storage](https://developer.android.com/training/data-storage/files.html#WriteInternalStorage).

Internal cache files

If you'd like to keep some data temporarily, rather than store it persistently, you should use the special cache directory to save the data. Each app has a private cache directory specifically for these kinds of files. When the device is low on internal storage space, Android may delete these cache files to recover space. However, you should not rely on the system to clean up these files for you. You should always maintain the cache files yourself and stay within a reasonable limit of space consumed, such as 1MB. When the user uninstalls your app, these files are removed.

For more information, see how to [write a cache file](https://developer.android.com/training/data-storage/files.html#WriteCacheFileInternal).

External storage

Every Android device supports a shared "external storage" space that you can use to save files. This space is called external because it is not guaranteed to be accessible—it is a storage space that users can mount to a computer as an external storage device, and it might even be physically removable (such as an SD card). Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer.

So before you attempt to access a file in external storage in your app, you should check for the availability of the external storage directories as well as the files you are trying to access.

Most often, you should use external storage for user data that should be accessible to other apps and saved even if the user uninstalls your app, such as captured photos or downloaded files. The system provides standard public directories for these kinds of files, so the user has one location for all their photos, ringtones, music, and such.

You can also save files to the external storage in an app-specific directory that the system deletes when the user uninstalls your app. This might be a useful alternative to internal storage if you need more space, but the files here aren't guaranteed to be accessible because the user might remove the storage SD card. And the files are still world readable; they're just saved to a location that's not shared with other apps.

For more information, read how to [save a file on external storage](https://developer.android.com/training/data-storage/files.html#WriteExternalStorage).

Shared preferences

If you don't need to store a lot of data and it doesn't require structure, you should use [SharedPreferences](https://developer.android.com/reference/android/content/SharedPreferences.html). The[SharedPreferences](https://developer.android.com/reference/android/content/SharedPreferences.html) APIs allow you to read and write persistent key-value pairs of primitive data types: booleans, floats, int, longs, and strings.

The key-value pairs are written to XML files that persist across user sessions, even if your app is killed. You can manually specify a name for the file or use per-activity files to save your data.

The API name "shared preferences" is a bit misleading because the API is not strictly for saving "user preferences," such as what ringtone a user has chosen. You can use [SharedPreferences](https://developer.android.com/reference/android/content/SharedPreferences.html) to save any kind of simple data. However, if you *do* want to save user preferences for your app, then you should read how to [create a settings UI](https://developer.android.com/guide/topics/ui/settings.html), which uses the AndroidX [Preference Library](https://developer.android.com/reference/androidx/preference/package-summary.html) to build a settings screen and automatically persist the user's settings.

To learn how to store any type of key-value data, read [Save Key-Value Data with SharedPreferences](https://developer.android.com/training/data-storage/shared-preferences.html).

Databases

Android provides full support for SQLite databases. Any database you create is accessible only by your app. However, instead of using SQLite APIs directly, we recommend that you create and interact with your databases with the [Room persistence library](https://developer.android.com/training/data-storage/room/index.html).

The Room library provides an object-mapping abstraction layer that allows fluent database access while harnessing the full power of SQLite.

Although you can still [save data directly with SQLite](https://developer.android.com/training/data-storage/sqlite.html), the SQLite APIs are fairly low-level and require a great deal of time and effort to use. For example:

* There is no compile-time verification of raw SQL queries.
* As your schema changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
* You need to write lots of boilerplate code to convert between SQL queries and Java data objects.

The [Room persistence library](https://developer.android.com/training/data-storage/room/index.html) takes care of these concerns for you while providing an abstraction layer over SQLite.

For sample apps that demonstrate how to use Room, see the following on GitHub:

* [Android Architecture Components Basic Sample](https://github.com/googlesamples/android-architecture-components/tree/master/BasicSample)
* [Room & RxJava Sample](https://github.com/googlesamples/android-architecture-components/tree/master/BasicRxJavaSample)
* [Room Migration Sample](https://github.com/googlesamples/android-architecture-components/tree/master/PersistenceMigrationsSample)

Database debugging

The Android SDK includes a sqlite3 database tool that allows you to browse table contents, run SQL commands, and perform other useful functions on SQLite databases. For more information, see the [adb documentation](https://developer.android.com/studio/command-line/adb.html" \l "othershellcommands).

FileProvider

public class FileProvider   
extends [ContentProvider](https://developer.android.com/reference/android/content/ContentProvider.html)

|  |  |  |
| --- | --- | --- |
| java.lang.Object | | |
| ↳ | [android.content.ContentProvider](https://developer.android.com/reference/android/content/ContentProvider.html) | |
|  | ↳ | android.support.v4.content.FileProvider |

FileProvider is a special subclass of [ContentProvider](https://developer.android.com/reference/android/content/ContentProvider.html) that facilitates secure sharing of files associated with an app by creating a content:// [Uri](https://developer.android.com/reference/android/net/Uri.html) for a file instead of a file:/// [Uri](https://developer.android.com/reference/android/net/Uri.html).

A content URI allows you to grant read and write access using temporary access permissions. When you create an [Intent](https://developer.android.com/reference/android/content/Intent.html) containing a content URI, in order to send the content URI to a client app, you can also call [Intent.setFlags()](https://developer.android.com/reference/android/content/Intent.html" \l "setFlags(int)) to add permissions. These permissions are available to the client app for as long as the stack for a receiving [Activity](https://developer.android.com/reference/android/app/Activity.html) is active. For an [Intent](https://developer.android.com/reference/android/content/Intent.html) going to a [Service](https://developer.android.com/reference/android/app/Service.html), the permissions are available as long as the [Service](https://developer.android.com/reference/android/app/Service.html) is running.

In comparison, to control access to a file:/// [Uri](https://developer.android.com/reference/android/net/Uri.html) you have to modify the file system permissions of the underlying file. The permissions you provide become available to *any* app, and remain in effect until you change them. This level of access is fundamentally insecure.

The increased level of file access security offered by a content URI makes FileProvider a key part of Android's security infrastructure.

This overview of FileProvider includes the following topics:

1. [Defining a FileProvider](https://developer.android.com/reference/android/support/v4/content/FileProvider.html#ProviderDefinition)
2. [Specifying Available Files](https://developer.android.com/reference/android/support/v4/content/FileProvider.html#SpecifyFiles)
3. [Retrieving the Content URI for a File](https://developer.android.com/reference/android/support/v4/content/FileProvider.html#GetUri)
4. [Granting Temporary Permissions to a URI](https://developer.android.com/reference/android/support/v4/content/FileProvider.html#Permissions)
5. [Serving a Content URI to Another App](https://developer.android.com/reference/android/support/v4/content/FileProvider.html#ServeUri)

Defining a FileProvider

Since the default functionality of FileProvider includes content URI generation for files, you don't need to define a subclass in code. Instead, you can include a FileProvider in your app by specifying it entirely in XML. To specify the FileProvider component itself, add a [<provider>](https://developer.android.com/guide/topics/manifest/provider-element.html) element to your app manifest. Set the android:nameattribute to android.support.v4.content.FileProvider. Set the android:authorities attribute to a URI authority based on a domain you control; for example, if you control the domain mydomain.com you should use the authority com.mydomain.fileprovider. Set the android:exported attribute to false; the FileProvider does not need to be public. Set the [android:grantUriPermissions](https://developer.android.com/guide/topics/manifest/provider-element.html" \l "gprmsn) attribute to true, to allow you to grant temporary access to files. For example:

<manifest>  
    ...  
    <application>  
        ...  
        <provider  
            android:name="android.support.v4.content.FileProvider"  
            android:authorities="com.mydomain.fileprovider"  
            android:exported="false"  
            android:grantUriPermissions="true">  
            ...  
        </provider>  
        ...  
    </application>  
</manifest>

If you want to override any of the default behavior of FileProvider methods, extend the FileProvider class and use the fully-qualified class name in the android:name attribute of the <provider> element.

Specifying Available Files

A FileProvider can only generate a content URI for files in directories that you specify beforehand. To specify a directory, specify the its storage area and path in XML, using child elements of the <paths> element. For example, the following paths element tells FileProvider that you intend to request content URIs for the images/subdirectory of your private file area.

<paths xmlns:android="http://schemas.android.com/apk/res/android">  
    <files-path name="my\_images" path="images/"/>  
    ...  
</paths>

The <paths> element must contain one or more of the following child elements:

**<files-path name="*name*" path="*path*" />**

Represents files in the files/ subdirectory of your app's internal storage area. This subdirectory is the same as the value returned by [Context.getFilesDir()](https://developer.android.com/reference/android/content/Context.html" \l "getFilesDir()).

**<cache-path name="*name*" path="*path*" />**

Represents files in the cache subdirectory of your app's internal storage area. The root path of this subdirectory is the same as the value returned by [getCacheDir()](https://developer.android.com/reference/android/content/Context.html" \l "getCacheDir()).

**<external-path name="*name*" path="*path*" />**

Represents files in the root of the external storage area. The root path of this subdirectory is the same as the value returned by [Environment.getExternalStorageDirectory()](https://developer.android.com/reference/android/os/Environment.html" \l "getExternalStorageDirectory()).

**<external-files-path name="*name*" path="*path*" />**

Represents files in the root of your app's external storage area. The root path of this subdirectory is the same as the value returned by Context#getExternalFilesDir(String) Context.getExternalFilesDir(null).

**<external-cache-path name="*name*" path="*path*" />**

Represents files in the root of your app's external cache area. The root path of this subdirectory is the same as the value returned by [Context.getExternalCacheDir()](https://developer.android.com/reference/android/content/Context.html" \l "getExternalCacheDir()).

**<external-media-path name="*name*" path="*path*" />**

Represents files in the root of your app's external media area. The root path of this subdirectory is the same as the value returned by the first result of [Context.getExternalMediaDirs()](https://developer.android.com/reference/android/content/Context.html" \l "getExternalMediaDirs()).

**Note:** this directory is only available on API 21+ devices.

These child elements all use the same attributes:

name="*name*"

A URI path segment. To enforce security, this value hides the name of the subdirectory you're sharing. The subdirectory name for this value is contained in the path attribute.

path="*path*"

The subdirectory you're sharing. While the name attribute is a URI path segment, the path value is an actual subdirectory name. Notice that the value refers to a **subdirectory**, not an individual file or files. You can't share a single file by its file name, nor can you specify a subset of files using wildcards.

You must specify a child element of <paths> for each directory that contains files for which you want content URIs. For example, these XML elements specify two directories:

<paths xmlns:android="http://schemas.android.com/apk/res/android">  
    <files-path name="my\_images" path="images/"/>  
    <files-path name="my\_docs" path="docs/"/>  
</paths>

Put the <paths> element and its children in an XML file in your project. For example, you can add them to a new file called res/xml/file\_paths.xml. To link this file to the FileProvider, add a [<meta-data>](https://developer.android.com/guide/topics/manifest/meta-data-element.html) element as a child of the <provider> element that defines the FileProvider. Set the <meta-data> element's "android:name" attribute to android.support.FILE\_PROVIDER\_PATHS. Set the element's "android:resource" attribute to @xml/file\_paths (notice that you don't specify the .xml extension). For example:

<provider  
    android:name="android.support.v4.content.FileProvider"  
    android:authorities="com.mydomain.fileprovider"  
    android:exported="false"  
    android:grantUriPermissions="true">  
    <meta-data  
        android:name="android.support.FILE\_PROVIDER\_PATHS"  
        android:resource="@xml/file\_paths" />  
</provider>

Generating the Content URI for a File

To share a file with another app using a content URI, your app has to generate the content URI. To generate the content URI, create a new [File](https://developer.android.com/reference/java/io/File.html) for the file, then pass the [File](https://developer.android.com/reference/java/io/File.html) to [getUriForFile()](https://developer.android.com/reference/android/support/v4/content/FileProvider.html" \l "getUriForFile(android.content.Context,%20java.lang.String,%20java.io.File)). You can send the content URI returned by [getUriForFile()](https://developer.android.com/reference/android/support/v4/content/FileProvider.html" \l "getUriForFile(android.content.Context,%20java.lang.String,%20java.io.File)) to another app in an [Intent](https://developer.android.com/reference/android/content/Intent.html). The client app that receives the content URI can open the file and access its contents by calling [ContentResolver.openFileDescriptor](https://developer.android.com/reference/android/content/ContentResolver.html" \l "openFileDescriptor(android.net.Uri,%20java.lang.String)) to get a [ParcelFileDescriptor](https://developer.android.com/reference/android/os/ParcelFileDescriptor.html).

For example, suppose your app is offering files to other apps with a FileProvider that has the authority com.mydomain.fileprovider. To get a content URI for the file default\_image.jpg in the images/subdirectory of your internal storage add the following code:

File imagePath = new File(Context.getFilesDir(), "images");  
File newFile = new File(imagePath, "default\_image.jpg");  
Uri contentUri = getUriForFile(getContext(), "com.mydomain.fileprovider", newFile);

As a result of the previous snippet, [getUriForFile()](https://developer.android.com/reference/android/support/v4/content/FileProvider.html" \l "getUriForFile(android.content.Context,%20java.lang.String,%20java.io.File)) returns the content URIcontent://com.mydomain.fileprovider/my\_images/default\_image.jpg.

Granting Temporary Permissions to a URI

To grant an access permission to a content URI returned from [getUriForFile()](https://developer.android.com/reference/android/support/v4/content/FileProvider.html" \l "getUriForFile(android.content.Context,%20java.lang.String,%20java.io.File)), do one of the following:

* Call the method [Context.grantUriPermission(package, Uri, mode\_flags)](https://developer.android.com/reference/android/content/Context.html" \l "grantUriPermission(java.lang.String,%20android.net.Uri,%20int)) for the content:// [Uri](https://developer.android.com/reference/android/net/Uri.html), using the desired mode flags. This grants temporary access permission for the content URI to the specified package, according to the value of the the mode\_flags parameter, which you can set to[FLAG\_GRANT\_READ\_URI\_PERMISSION](https://developer.android.com/reference/android/content/Intent.html#FLAG_GRANT_READ_URI_PERMISSION), [FLAG\_GRANT\_WRITE\_URI\_PERMISSION](https://developer.android.com/reference/android/content/Intent.html#FLAG_GRANT_WRITE_URI_PERMISSION) or both. The permission remains in effect until you revoke it by calling [revokeUriPermission()](https://developer.android.com/reference/android/content/Context.html" \l "revokeUriPermission(android.net.Uri,%20int)) or until the device reboots.
* Put the content URI in an [Intent](https://developer.android.com/reference/android/content/Intent.html) by calling [setData()](https://developer.android.com/reference/android/content/Intent.html" \l "setData(android.net.Uri)).
* Next, call the method [Intent.setFlags()](https://developer.android.com/reference/android/content/Intent.html" \l "setFlags(int)) with either [FLAG\_GRANT\_READ\_URI\_PERMISSION](https://developer.android.com/reference/android/content/Intent.html#FLAG_GRANT_READ_URI_PERMISSION) or[FLAG\_GRANT\_WRITE\_URI\_PERMISSION](https://developer.android.com/reference/android/content/Intent.html#FLAG_GRANT_WRITE_URI_PERMISSION) or both.
* Finally, send the [Intent](https://developer.android.com/reference/android/content/Intent.html) to another app. Most often, you do this by calling [setResult()](https://developer.android.com/reference/android/app/Activity.html" \l "setResult(int,%20android.content.Intent)).

Permissions granted in an [Intent](https://developer.android.com/reference/android/content/Intent.html) remain in effect while the stack of the receiving [Activity](https://developer.android.com/reference/android/app/Activity.html) is active. When the stack finishes, the permissions are automatically removed. Permissions granted to one [Activity](https://developer.android.com/reference/android/app/Activity.html) in a client app are automatically extended to other components of that app.

Serving a Content URI to Another App

There are a variety of ways to serve the content URI for a file to a client app. One common way is for the client app to start your app by calling [startActivityResult()](https://developer.android.com/reference/android/app/Activity.html" \l "startActivityForResult(android.content.Intent,%20int,%20android.os.Bundle)), which sends an [Intent](https://developer.android.com/reference/android/content/Intent.html) to your app to start an [Activity](https://developer.android.com/reference/android/app/Activity.html) in your app. In response, your app can immediately return a content URI to the client app or present a user interface that allows the user to pick a file. In the latter case, once the user picks the file your app can return its content URI. In both cases, your app returns the content URI in an [Intent](https://developer.android.com/reference/android/content/Intent.html) sent via [setResult()](https://developer.android.com/reference/android/app/Activity.html" \l "setResult(int,%20android.content.Intent)).

You can also put the content URI in a [ClipData](https://developer.android.com/reference/android/content/ClipData.html) object and then add the object to an [Intent](https://developer.android.com/reference/android/content/Intent.html) you send to a client app. To do this, call [Intent.setClipData()](https://developer.android.com/reference/android/content/Intent.html" \l "setClipData(android.content.ClipData)). When you use this approach, you can add multiple [ClipData](https://developer.android.com/reference/android/content/ClipData.html) objects to the [Intent](https://developer.android.com/reference/android/content/Intent.html), each with its own content URI. When you call [Intent.setFlags()](https://developer.android.com/reference/android/content/Intent.html" \l "setFlags(int)) on the [Intent](https://developer.android.com/reference/android/content/Intent.html) to set temporary access permissions, the same permissions are applied to all of the content URIs.

Sharing files

Apps often have a need to offer one or more of their files to another app. For example, an image gallery may want to offer files to image editors, or a file management app may want to allow users to copy and paste files between areas in external storage. One way a sending app can share a file is to respond to a request from the receiving app.

In all cases, the only secure way to offer a file from your app to another app is to send the receiving app the file's content URI and grant temporary access permissions to that URI. Content URIs with temporary URI access permissions are secure because they apply only to the app that receives the URI, and they expire automatically. The Android [FileProvider](https://developer.android.com/reference/android/support/v4/content/FileProvider.html) component provides the method [getUriForFile()](https://developer.android.com/reference/android/support/v4/content/FileProvider.html" \l "getUriForFile(android.content.Context,%20java.lang.String,%20java.io.File)) for generating a file's content URI.

If you want to share small amounts of text or numeric data between apps, you should send an [Intent](https://developer.android.com/reference/android/content/Intent.html) that contains the data. To learn how to send simple data with an [Intent](https://developer.android.com/reference/android/content/Intent.html), see the training class [Sharing simple data](https://developer.android.com/training/sharing/index.html).

This class explains how to securely share files from your app to another app using content URIs generated by the Android [FileProvider](https://developer.android.com/reference/android/support/v4/content/FileProvider.html) component and temporary permissions that you grant to the receiving app for the content URI.

Lessons

[**Setting up file sharing**](https://developer.android.com/training/secure-file-sharing/setup-sharing.html)

Learn how to set up your app to share files.

[**Sharing a file**](https://developer.android.com/training/secure-file-sharing/share-file.html)

Learn how to offer a file to another app by generating a content URI for the file, granting access permissions to the URI, and sending the URI to the app.

[**Requesting a shared file**](https://developer.android.com/training/secure-file-sharing/request-file.html)

Learn how to request a file shared by another app, receive the content URI for the file, and use the content URI to open the file.

[**Retrieving file information**](https://developer.android.com/training/secure-file-sharing/retrieve-info.html)

Learn how an app can use a content URI generated by a [FileProvider](https://developer.android.com/reference/android/support/v4/content/FileProvider.html) to retrieve file information including MIME type and file size.

For additional related information, refer to:

* [Storage Options](https://developer.android.com/guide/topics/data/data-storage.html)
* [Saving Files](https://developer.android.com/training/basics/data-storage/files.html)
* [Sharing Simple Data](https://developer.android.com/training/sharing/index.html)