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.

This page introduces the different data storage options available on Android:

- Internal file storage (#filesInternal): Store app-private files on the device file system.
- External file storage (#filesExternal): Store files on the shared external file system. This is usually for sharared user files, such as photos.
- Shared preferences (#pref): Store private primitive data in key-value pairs.
- Databases (#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 independenly 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's not a 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, ints, 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, such as the user's high score. 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 `PreferenceActivity` (https://developer.android.com/reference/android/preference/PreferenceActivity.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 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#othershellcommands).

Zuletzt aktualisiert: April 24, 2018

Twitter
Follow @AndroidDev on Twitter

Google+
Follow Android Developers on Google+

YouTube
Check out Android Developers on YouTube