# Access app-specific files

In many cases, your app creates files that other apps don't need to access, or shouldn't access. The system provides the following locations for storing such *app-specific* files:

- **Internal storage directories:** These directories include both a dedicated location for storing persistent files, and another location for storing cache data. The system prevents other apps from accessing these locations, and on Android 10 (API level 29) and higher, these locations are encrypted. These characteristics make these locations a good place to store sensitive data that only your app itself can access.
- **External storage directories:** These directories include both a dedicated location for storing persistent files, and another location for storing cache data. Although it's possible for another app to access these directories if that app has the proper permissions, the files stored in these directories are meant for use only by your app. If you specifically intend to create files that other apps should be able to access, your app should store these files in the shared storage (/training/data-storage/shared) part of external storage instead.

When the user uninstalls your app, the files saved in app-specific storage are removed. Because of this behavior, you shouldn't use this storage to save anything that 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 use shared storage to save those types of files to the appropriate media collection (/training/data-storage/shared/media).

**Note:** To further protect app-specific files, use the Security library (/topic/security/data) that's part of Android Jetpack (/jetpack) to encrypt these files at rest. The encryption key is specific to your app.

The following sections describe how to store and access files within app-specific directories.

## Access from internal storage

For each app, the system provides directories within internal storage where an app can organize its files. One directory is designed for your app's persistent files (#internal-access-files), and another contains your app's cached files (#internal-create-cache). Your app doesn't require any system permissions to read and write to files in these directories.

Other apps cannot access files stored within internal storage. This makes internal storage a good place for app data that other apps shouldn't access.

Keep in mind, however, that these directories tend to be small. Before writing app-specific files to internal storage, your app should query the free space (#query-free-space) on the device.

### Access persistent files

Your app's ordinary, persistent files reside in a directory that you can access using the `filesDir` (/reference/android/content/Context#getFilesDir()) property of a context object. The framework provides several methods to help you access and store files in this directory.

#### Access and store files

You can use the `File` (/reference/java/io/File) API to access and store files.

To help maintain your app's performance, don't open and close the same file multiple times.

The following code snippet demonstrates how to use the `File` API:

Kotlin (#kotlin)Java (#java)

```
File file = new File(context.getFilesDir(), filename);
```

### Store a file using a stream

As an alternative to using the `File` API, you can call `openFileOutput()` (/reference/android/content/Context#openFileOutput(java.lang.String,%20int)) to get a `FileOutputStream` (/reference/java/io/FileOutputStream) that writes to a file within the `filesDir` directory.

The following code snippet shows how to write some text to a file:

Kotlin (#kotlin)Java (#java)

```
String filename = "myfile";
String fileContents = "Hello world!";
try (FileOutputStream fos = context.openFileOutput(filename, Context.MOD
    fos.write(fileContents.toByteArray());
}
```

**Caution:** On devices that run Android 7.0 (API level 24) or higher, unless you pass the **`Context.MODE_PRIVATE`** file mode into **`openFileOutput()`**, a **`SecurityException`** (/reference/java/lang/SecurityException) occurs.

To allow other apps to access files (/training/secure-file-sharing) stored in this directory within internal storage, use a `FileProvider` (/reference/androidx/core/content/FileProvider) with the `FLAG_GRANT_READ_URI_PERMISSION` (/reference/android/content/Intent#FLAG_GRANT_READ_URI_PERMISSION) attribute.

### Access a file using a stream

To read a file as a stream, use `openFileInput()` (/reference/android/content/Context#openFileInput(java.lang.String)):

Kotlin (#kotlin)Java (#java)

```
FileInputStream fis = context.openFileInput(filename);
InputStreamReader inputStreamReader =
        new InputStreamReader(fis, StandardCharsets.UTF_8);
StringBuilder stringBuilder = new StringBuilder();
try (BufferedReader reader = new BufferedReader(inputStreamReader)) {
    String line = reader.readLine();
    while (line != null) {
        stringBuilder.append(line).append('\n');
        line = reader.readLine();
    }
} catch (IOException e) {
    // Error occurred when opening raw file for reading.
} finally {
    String contents = stringBuilder.toString();
}
```

**Note:** If you need to access a file as a stream at install time, save the file in your project's **`/res/raw`** directory. You can open these files with **`openRawResource()`** (/reference/android/content/res/Resources#openRawResource(int)), passing in the filename prefixed with **`R.raw`** as the resource ID. This method returns an **`InputStream`** (/reference/java/io/InputStream) that you can use to read the file. You cannot write to the original file.

## View list of files

You can get an array containing the names of all files within the `filesDir` directory by calling `fileList()` (/reference/android/content/Context#fileList()), as shown in the following code snippet:

Kotlin (#kotlin)Java (#java)

```
Array<String> files = context.fileList();
```

### Create nested directories

You can also create nested directories, or open an inner directory, by calling `getDir()` (/reference/kotlin/android/content/Context#getdir) in Kotlin-based code or by passing the root directory and a new directory name into a `File` constructor in Java-based code:

Kotlin (#kotlin)Java (#java)

```
File directory = context.getFilesDir();
File file = new File(directory, filename);
```

**Note:** `filesDir` is always an ancestor directory of this new directory.

## Create cache files

If you need to store sensitive data only temporarily, you should use the app's designated cache directory within internal storage to save the data. As is the case for all app-specific storage, the files stored in this directory are removed when the user uninstalls your app, although the files in this directory might be removed sooner (#internal-remove-cache).

**Note:** This cache directory is designed to store a small amount of your app's sensitive data. To determine how much cache space is currently available for your app, call **`getCacheQuotaBytes()`** (/reference/android/os/storage/StorageManager#getCacheQuotaBytes(java.util.UUID)) .

To create a cached file, call `File.createTempFile()` (/reference/java/io/File#createTempFile(java.lang.String,%20java.lang.String)) :

Kotlin (#kotlin)Java (#java)

```
File.createTempFile(filename, null, context.getCacheDir());
```

Your app accesses a file in this directory using the `cacheDir` (/reference/android/content/Context#getCacheDir()) property of a context object and the `File` (/reference/java/io/File) API:

Kotlin (#kotlin)Java (#java)

```
File cacheFile = new File(context.getCacheDir(), filename);
```

**Caution:** When the device is low on internal storage space, Android may delete these cache files to recover space. So check for the existence of your cache files before reading them.

## Remove cache files

Even though Android sometimes deletes cache files on its own, you shouldn't rely on the system to clean up these files for you. You should always maintain your app's cache files within internal storage.

To remove a file from the cache directory within internal storage, use one of the following methods:

- The `delete()` (/reference/java/io/File#delete()) method on a `File` object that represents the file:

  Kotlin (#kotlin)Java (#java)

  ```
  cacheFile.delete();
  ```

- The `deleteFile()` (/reference/android/content/Context#deleteFile(java.lang.String)) method of the app's context, passing in the name of the file:

Kotlin (#kotlin)Java (#java)

```
context.deleteFile(cacheFileName);
```

## Access from external storage

If internal storage doesn't provide enough space to store app-specific files, consider using external storage instead. The system provides directories within external storage where an app can organize files that provide value to the user only within your app. One directory is designed for your app's persistent files (#external-access-files), and another contains your app's cached files (#external-cache-create).

On Android 4.4 (API level 19) or higher, your app doesn't need to request any storage-related permissions to access app-specific directories within external storage. The files stored in these directories are removed when your app is uninstalled.

**Caution:** The files in these directories aren't guaranteed to be accessible, such as when a removable SD card is taken out of the device. If your app's functionality depends on these files, you should instead store the files within internal storage (#internal).

On devices that run Android 9 (API level 28) or lower, your app can access the app-specific files that belong to other apps, provided that your app has the appropriate storage permissions. To give users more control over their files and to limit file clutter, apps that target Android 10 (API level 29) and higher are given scoped access into external storage, or scoped storage (/training/data-storage#scoped-storage), by default. When scoped storage is enabled, apps cannot access the app-specific directories that belong to other apps.

## Verify that storage is available

Because external storage resides on a physical volume that the user might be able to remove, verify that the volume is accessible before trying to read app-specific data from, or write app-specific data to, external storage.

You can query the volume's state by calling `Environment.getExternalStorageState()` (/reference/android/os/Environment#getExternalStorageState()). If the returned state is `MEDIA_MOUNTED` (/reference/android/os/Environment#MEDIA_MOUNTED), then you can read and write app-specific files within external storage. If it's `MEDIA_MOUNTED_READ_ONLY` (/reference/android/os/Environment#MEDIA_MOUNTED_READ_ONLY), you can only read these files.

For example, the following methods are useful to determine the storage availability:

Kotlin (#kotlin) Java (#java)

```
// Checks if a volume containing external storage is available
// for read and write.
private boolean isExternalStorageWritable() {
    return Environment.getExternalStorageState().equals(Environment.MEDI
}

// Checks if a volume containing external storage is available to at lea
private boolean isExternalStorageReadable() {
     return Environment.getExternalStorageState().equals(Environment.MEI
            Environment.getExternalStorageState().equals(Environment.MEI
}
```

On devices without removable external storage, use the following command to enable a virtual volume for testing your external storage availability logic:

```
$ adb shell sm set-virtual-disk true
```

## Select a physical storage location

Sometimes, a device that allocates a partition of its internal memory as external storage also provides an SD card slot. This means that the device has multiple physical volumes that could contain external storage, so you need to select which one to use for your app-specific storage.

To access the different locations, call `ContextCompat.getExternalFilesDirs()` (/reference/androidx/core/content/ContextCompat#getExternalFilesDirs(android.content.Context,%20java.lang.String)). As shown in the code snippet, the first element in the returned array is considered the primary external storage volume. Use this volume unless it's full or unavailable.

Kotlin (#kotlin) Java (#java)

```
File[] externalStorageVolumes =
        ContextCompat.getExternalFilesDirs(getApplicationContext(), nul]
File primaryExternalStorage = externalStorageVolumes[0];
```

**Note:** If your app is used on a device that runs Android 4.3 (API level 18) or lower, then the array contains just one element, which represents the primary external storage volume.

## Access persistent files

To access app-specific files from external storage, call `getExternalFilesDir()` (/reference/android/content/Context#getExternalFilesDir(java.lang.String)).

To help maintain your app's performance, don't open and close the same file multiple times.

The following code snippet demonstrates how to call `getExternalFilesDir()`:

Kotlin (#kotlin)Java (#java)

```
File appSpecificExternalDir = new File(context.getExternalFilesDir(null)
```

**Note:** On Android 11 (API level 30) and higher, apps cannot create their own app-specific directory on external storage.

## Create cache files

To add an app-specific file to the cache within external storage, get a reference to the `externalCacheDir` (/reference/android/content/Context#getExternalCacheDir()):

Kotlin (#kotlin)Java (#java)

```
File externalCacheFile = new File(context.getExternalCacheDir(), filenam
```

## Remove cache files

To remove a file from the external cache directory, use the `delete()` (/reference/java/io/File#delete()) method on a `File` object that represents the file:

Kotlin (#kotlin)Java (#java)

```
externalCacheFile.delete();
```

## Media content

If your app works with media files that provide value to the user only within your app, it's best to store them in app-specific

directories within external storage, as demonstrated in the following code snippet:

Kotlin (#kotlin)Java (#java)

```
@Nullable
File getAppSpecificAlbumStorageDir(Context context, String albumName) {
    // Get the pictures directory that's inside the app-specific directo
    // external storage.
    File file = new File(context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES), albumName);
    if (file == null || !file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

It's important that you use directory names provided by API constants like `DIRECTORY_PICTURES` (/reference/android/os/Environment#DIRECTORY_PICTURES) . These directory names ensure that the files are treated properly by the system. If none of the pre-defined sub-directory names (/reference/android/os/Environment#fields) suit your files, you can instead pass `null` into `getExternalFilesDir()`. This returns the root app-specific directory within external storage.

## Query free space

Many users don't have much storage space available on their devices, so your app should consume space thoughtfully.

If you know ahead of time how much data you're storing, you can find out how much space the device can provide your app by calling `getAllocatableBytes()` (/reference/android/os/storage/StorageManager#getAllocatableBytes(java.util.UUID)) . The return value of `getAllocatableBytes()` might be larger than the current amount of free space on the device. This is because the system has identified files that it can remove from other apps' cache directories.

If there's enough space to save your app's data, call `allocateBytes()` (/reference/android/os/storage/StorageManager#allocateBytes(java.io.FileDescriptor,%20long)). Otherwise, your app can request the user to remove some files (#user-remove-device-files) from the device or remove all cache files (#user-remove-all-cache-files) from the device.

The following code snippet shows an example of how your app can query free space on the device:

Kotlin (#kotlin) Java (#java)

```
// App needs 10 MB within internal storage.
private static final long NUM_BYTES_NEEDED_FOR_MY_APP = 1024 * 1024 * 10

StorageManager storageManager =
        getApplicationContext().getSystemService(StorageManager.class);
UUID appSpecificInternalDirUuid = storageManager.getUuidForPath(getFiles
long availableBytes =
        storageManager.getAllocatableBytes(appSpecificInternalDirUuid);
if (availableBytes >= NUM_BYTES_NEEDED_FOR_MY_APP) {
    storageManager.allocateBytes(
            appSpecificInternalDirUuid, NUM_BYTES_NEEDED_FOR_MY_APP);
} else {
    // To request that the user remove all app cache files instead, set
    // "action" to ACTION_CLEAR_APP_CACHE.
    Intent storageIntent = new Intent();
    storageIntent.setAction(ACTION_MANAGE_STORAGE);
}
```

**Note:** You aren't required to check the amount of available space before you save your file. You can instead try writing the file right away, then catch an **`IOException`** (/reference/java/io/IOException) if one occurs. You may need to do this if you don't know exactly how much space you need. For example, if you change the file's encoding before you save it by converting a PNG image to JPEG, you don't know the file's size beforehand.

## Create a storage management activity

Your app can declare and create a custom activity that, when launched, allows the user to manage the data that your app has stored on the user's device. You declare this custom "manage space" activity using the `android:manageSpaceActivity` (/guide/topics/manifest/application-element#space) attribute in the manifest file. File manager apps can invoke this activity (/training/data-storage/manage-all-files#invoke-storage-management-activity) even when your app doesn't export the activity; that is, when your activity sets `android:exported` (/guide/topics/manifest/activity-element#exported) to `false`.

## Ask user to remove some device files

To request that the user choose files on the device to remove, invoke an intent that includes the `ACTION_MANAGE_STORAGE` (/reference/android/os/storage/StorageManager#ACTION_MANAGE_STORAGE) action. This intent displays a prompt to the user. If desired, this prompt can show the amount of free space available on the device. To show this user-friendly information, use the result of the following calculation:

```
StorageStatsManager.getFreeBytes() / StorageStatsManager.getTotalBytes()
```

## Ask user to remove all cache files

Alternatively, you can request that the user clear the cache files from **all** apps on the device. To do so, invoke an intent that includes the `ACTION_CLEAR_APP_CACHE` (/reference/android/os/storage/StorageManager#ACTION_CLEAR_APP_CACHE) intent action.

**Caution:** The **`ACTION_CLEAR_APP_CACHE`** intent action can substantially affect device battery life and might remove a large number of files from the device.

# Additional resources

For more information about saving files to the device's storage, consult the following resources.

## Videos

- Preparing for Scoped Storage (Android Dev Summit '19) (https://www.youtube.com/watch?v=UnJ3amzJM94)



Last updated 2022-12-02 UTC.