Android Data Storage and Access

Table of Contents

[Data Storage Options 2](#_Toc100927207)

[Selecting Storage Type 4](#_Toc100927208)

[App-Specific Storage 5](#_Toc100927209)

[Internal Storage 5](#_Toc100927210)

[External Storage 7](#_Toc100927211)

[Shared Preferences 9](#_Toc100927212)

[Shared Storage 10](#_Toc100927213)

[Database 11](#_Toc100927214)

[Entities 13](#_Toc100927215)

[DAOs 14](#_Toc100927216)

[Databases 15](#_Toc100927217)

[Relationships 15](#_Toc100927218)

## Data Storage Options

* **App-Specific Storage** – This is used to store files meant to be used by the app alone. They are stored in dedicated directories within the internal or external storage volumes. The internal storage volume should be used to store sensitive information that other apps should not access.
* **Shared Storage** – This is used to store files meant to be shared with other apps, such as media, documents, etc.
* **Preferences** – This is used to store private, primitive data in key-value pairs.
* **Databases** – This is used to store structured data in a private database using the Room persistence library or SQLite.
* **Network Storage** – This is used to store and sync data across multiple clients using Firebase.

More detailed information about the different storage types can be found in the table below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **App-specific files** | **Media** | **Documents and other files** | **App preferences** | **Database** |
| **Type of content** | Files meant for your app's use only | Shareable media files (images, audio files, videos) | Other types of shareable content, including downloaded files | Key-value pairs | Structured data |
| **Access method** | From internal storage, getFilesDir() or getCacheDir()  From external storage, getExternalFilesDir() or getExternalCacheDir() | MediaStore API | Storage Access Framework | Jetpack Preferences library | Room persistence library |
| **Permissions needed** | Never needed for internal storage  Not needed for external storage when your app is used on devices that run Android 4.4 (API level 19) or higher | READ\_EXTERNAL\_STORAGE when accessing other apps' files on Android 11 (API level 30) or higher  READ\_EXTERNAL\_STORAGE or WRITE\_EXTERNAL\_STORAGE when accessing other apps' files on Android 10 (API level 29)   Permissions are required for all files on Android 9 (API level 28) or lower | None | None | None |
| **Can other apps access?** | No | Yes, though the other app needs the READ\_EXTERNAL\_STORAGE permission | Yes, through the system file picker | No | No |
| **Files removed on app uninstall?** | Yes | No | No | Yes | Yes |

## Selecting Storage Type

To select which type of storage we should use, we need to consider several factors:

* **Space Requirements** – Internal storage has a limited amount of space for app-specific data. If we have large amounts of data, we should consider other options.
* **Access Reliability** – If the app’s basic functionality requires some data, such as when starting the app, this data should be placed in the internal storage or in a database. Data stored in external storage will not always be accessible, since the external storage can be removed for some devices.
* **Data Type** – For data that is only meaningful to the app, use the internal storage. For shareable media content, used shared storage. For structured data, use either preferences (for key-value pairs) or a database (for multi-column data).
* **Data Privacy** – For sensitive data, we should use internal storage, preferences or a database. Internal storage has the added benefit of being hidden from users as well.

## App-Specific Storage

App-specific storage is either on internal or external storage.

* **Internal Storage** – These directories have two parts, one for storing persistent files and another for storing cache data. The system stops other apps from accessing these directories, and from Android 10 onwards, the directories are also encrypted. This means the internal storage is good for sensitive data that is to be used by the app alone.
* **External Storage** – These directories also have two parts, one for storing persistent data and the other for storing cache data. Although other apps can access these directories if they are given the proper permissions, that is not the intended use case. This data is supposed to be for our app alone. If the data is meant to be accessed by other apps, then it should be placed in the shared storage part of the external storage instead.

## Internal Storage

Using **internal storage** does not require any permissions. They can be accessed and used using the **File API**.

File file = new File(context.getFilesDir(), someFileName);

JAVA

The directory in which the app’s ordinary persistent files are stored can be accessed using the FilesDir property of a Context object.

Alternatively, the openFileOutput method of a Context object can be used to retrieved a FileOutputStream object, which can be used to write to a file in the FilesDir directory.

String fileName = "myFile";  
String fileContents = "Hello World!";  
try (FileOutputStream fos =  
 context.openFileOutput(fileName, Context.*MODE\_PRIVATE*)) {  
 fos.write(fileContents.toByteArray());  
}

JAVA

To read a file instead, we can use the openFileInput method of a Context object to retrieve a FileInputStream object.

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) { // reading line by line  
 stringBuilder.append(line).append("\n");  
 line = reader.readLine();  
 }  
} catch(IOException e) {  
 // error occurred when opening raw file for reading  
} finally {  
 String contents = stringBuilder.toString();  
}

JAVA

To allow other apps to access files stored in the FilesDir directory, we must use a FileProvider object with the FLAG\_GRANT\_READ\_URI\_PERMISSION attribute.

## External Storage

The **external storage** should be used when the internal storage does not provide enough space. On Android 4.4 and higher, no special permissions are required to access the app-specific storage on the external storage. For Android 9 and lower, other apps can access this data given they have the appropriate permissions.

Since the external storage **can be removed**, we first need to check if it is even connected.

// check if external storage is readable and writable  
private boolean isExternalStorageWritable() {  
 String writable = Environment.*MEDIA\_MOUNTED*;  
 return Environment.*getExternalStorageState*().equals(writable);  
}  
  
// check if external storage is at least readable  
private boolean isExternalStorageReadable() {  
 String writable = Environment.*MEDIA\_MOUNTED*;  
 String readable = Environment.*MEDIA\_MOUNTED\_READ\_ONLY*;  
 return Environment.*getExternalStorageState*().equals(writable) ||  
 Environment.*getExternalStorageState*().equals(readable);  
}

JAVA

To access the external storage, we can use the getExternalFilesDir method of a Context object.

File file = new File(context.getExternalFilesDir(), fileName);

JAVA

If we want to store media files that we do not want other apps to access, we can use the app-specific directory in the external storage.

@Nullable  
File getAppSpecificAlbumStorageDir(Context context, String albumName) {  
 // Get the pictures directory inside the app-specific directory  
 File file = new File(context.getExternalFilesDir(  
 Environment.*DIRECTORY\_PICTURES*), albumName);  
 if (file == null || !file.mkdir()) {  
 Log.*e*(*LOG\_TAG*, "Directory not created.");  
 }  
 return file;  
}

JAVA

Common public directories include:

* Environment.*DIRECTORY\_ALARMS* - Audio files that should be in the list of alarms that the user can select
* Environment.*DIRECTORY\_DCIM* - Traditional location for pictures and videos when mounting the device as a camera
* Environment.*DIRECTORY\_DOWNLOADS*
* Environment.*DIRECTORY\_MOVIES*
* Environment.*DIRECTORY\_MUSIC*
* Environment.*DIRECTORY\_NOTIFICATIONS* - Audio files that should be in the list of notification sounds that the user can select
* Environment.*DIRECTORY\_PICTURES*
* Environment.*DIRECTORY\_PODCASTS*
* Environment.*DIRECTORY\_RINGTONES*

## Shared Preferences

The **Shared Preferences** API should be used if we have a small collection of key-value pairs. A *SharedPreferences* object points to a single file with the key-value pairs and provides simple methods to read and write them.

If we need access to **multiple** shared preferences files, we can retrieve them by name.

Context context = getActivity();  
*SharedPreferences* sharedPref = context.getSharedPreferences(  
 preference\_file\_key, Context.*MODE\_PRIVATE*);

JAVA

The name should be uniquely identifiable. An easy way to ensure this is to use the application ID in the file name, e.g. com.example.myapp.PREFERENCE\_FILE\_KEY.

If we need access to a **single** shared preference file, we can use the default one that belongs to the activity. This does not require any name to be provided.

*SharedPreferences* sharedPref = context.getPreferences(Context.*MODE\_PRIVATE*);

We can write to a shared preferences file like this:

*SharedPreferences*.*Editor* editor = sharedPref.edit();  
editor.putInt(someKey, someValue);  
editor.apply(); // save

JAVA

## Shared Storage

The **MediaStore API** provides an optimized index into media collections on the shared external storage that allows us to retrieve and update the media files easily. These media files are stored on **standard public directories** accessible to anyone. The files remain on the device even if the app is uninstalled.

To access these files, we run a query on a specific table. Possible tables include MediaStore.Images, MediaStore.Video, MediaStore.Audio and MediaStore.Downloads, the last of which is only available from Android 10 onwards.

String[] projection = new String[] {}; // list of column names to get  
String selection = ""; // SQL WHERE clause with placeholder variables  
String[] selectionArgs = new String[] {}; // placeholder values  
String sortOrder = ""; // SQL ORDER BY clause  
  
*Cursor* cursor = getApplicationContext().getContentResolver().query(  
 MediaStore.Images.Media.*EXTERNAL\_CONTENT\_URI*,  
 projection,  
 selection,  
 selectionArgs,  
 sortOrder  
);  
  
while (cursor.moveToNext()) {  
 */\* Use an ID column from the projection to get a URI representing  
 the media itself. \*/*}

JAVA

## Database

To use a database, we use the **Room library**. This library provides an abstraction over SQLite. SQLite itself is quite low-level and difficult to use. The Room library makes this easier for us.

This database is **local**, so it can be accessed **offline**. Any changes can be synchronized if required when the user goes back online. The database should be used for non-trivial amounts of structured data. No permissions are required. Other apps cannot access this data.

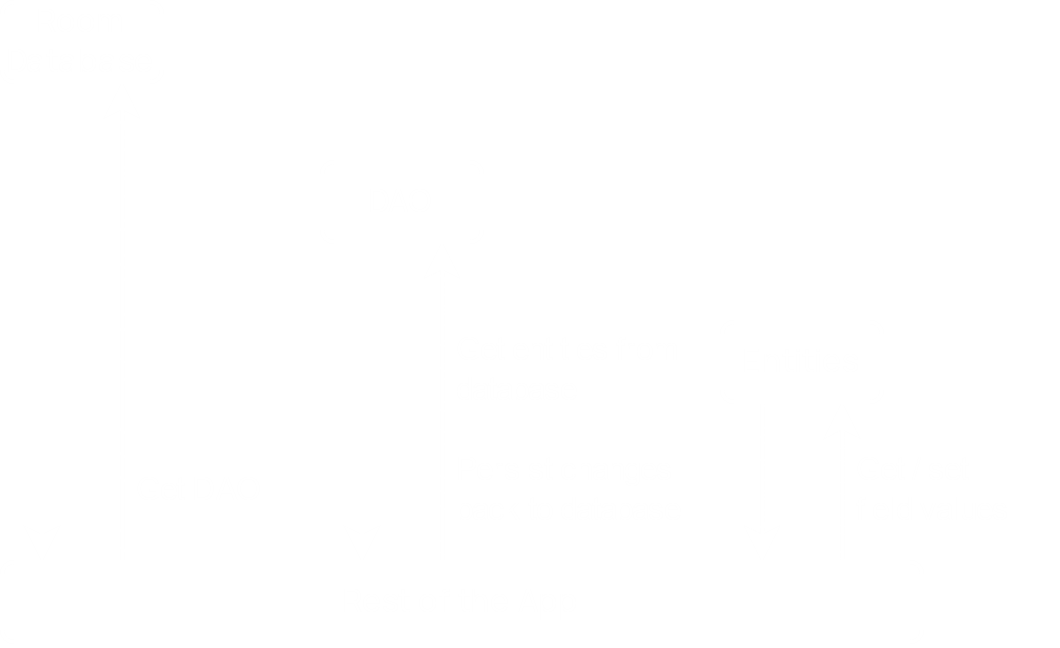
To use the Room library, we first need to add some **dependencies** to the application level build.gradle file.

dependencies {  
 def room\_version = "2.2.5"  
  
 implementation "androidx.room:room-runtime:$room\_version"  
 annotationProcessor "androidx.room:room-compiler:$room\_version"  
  
 // optional - RxJava support for Room  
 implementation "androidx.room:room-rxjava2:$room\_version"  
  
 // optional - Guava support for Room, with Optional and ListenableFuture  
 implementation "androidx.room:room-guava:$room\_version"  
  
 // optional = Text helpers  
 implementation "androidx.room:room-testing:$room\_version"  
}

The Room library has three major components:

* **Database** – This contains the database holder and serves as the main access point for the underlying connection to the application’s persistent relational database.
* **Entity** – This represents a table in the database.
* **DAO** – This contains the methods used to access the database.

The application uses the Room database to get the **Data Access Objects** (DAOs) associated with that database. Each of the DAOs are then used to access a separate entity and save changes to those entities back to the database. Each entity is used to get and set values that correspond to the table columns.



### Entities

We can create **entities** as below. Two examples are provided to highlight the fact that some things (like names) are optional:

@Entity (tableName = "users")  
public class User {  
 @PrimaryKey  
 public int id;  
  
 @ColumnInfo(name = "first\_name")  
 public String firstName;  
  
 @ColumnInfo(name = "last\_name")  
 public String lastName;  
}  
  
@Entity  
public class User {  
 @PrimaryKey public long userId;  
 public String name;  
 public int age;  
}

JAVA

### DAOs

We can use **DAOs** to define methods to use with the entities. This is where the usefulness of the Room library should start to become obvious. It is modularizing complicated SQL queries into simple methods we can use elsewhere.

@Dao  
public interface *UserDao* {  
 @Query("SELECT \* FROM user")  
 *List*<User> getAll();  
  
 @Query("SELECT \* FROM user WHERE uid IN (:userIds)")  
 *List*<User> loadAllByIds(int[] userIds);  
  
 @Query("SELECT \* FROM user WHERE first\_name LIKE :first AND " +  
 "last\_name LIKE :last LIMIT 1")  
 User findByName(String first, String last);  
  
 @Insert  
 void insertAll(User... users);  
  
 @Delete  
 void delete(User user);  
}

JAVA

### Databases

The **Database class** must satisfy some requirements.

* It has to be an abstract class that extends RoomDatabase.
* It must include a list of entities that are associated with the database.
* It must contain an abstract method with no arguments which returns the DAO class.

@Database(entities = {User.class}, version = 1)  
public abstract class AppDatabase extends RoomDatabase {  
 public abstract *UserDao* userDao();  
}

JAVA

An instance of the database can be retrieved using Room.*databaseBuilder* or Room.*inMemoryDatabaseBuilder*.

AppDatabase db = Room.*databaseBuilder*(getApplicationContext(),  
 AppDatabase.class, "database-name").build();

JAVA

### Relationships

A detailed overview of how relationships between tables work with the Room library can be found [here](https://developer.android.com/training/data-storage/room/relationships).