

ANDROID Data Storage & Access

Part 1

Overview

- Several options to store app data.
- **App-specific storage:** Store files that are meant for your app's use only, either in dedicated directories within an internal storage volume or different dedicated directories within external storage. Use the directories within internal storage to save sensitive information that other apps shouldn't access.
- **Shared storage:** Store files that your app intends to share with other apps, including media, documents, and other files.
- **Preferences:** Store private, primitive data in key-value pairs.
- **Databases:** Store structured data in a private database using the Room persistence library/ SQLite
- **Network Storage:** Store and sync data across multiple clients using Firebase (Real time Database).

	Type of content	Access method	Permissions needed	Can other apps access?	Files removed on app uninstall?
App-specific files	Files meant for your app's use only	From internal storage, <code>getFilesDir()</code> or <code>getCacheDir()</code> From external storage, <code>getExternalFilesDir()</code> or <code>getExternalCacheDir()</code>	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	No	Yes
Media	Shareable media files (images, audio files, videos)	MediaStore API	<code>READ_EXTERNAL_STORAGE</code> when accessing other apps' files on Android 11 (API level 30) or higher <code>READ_EXTERNAL_STORAGE</code> or <code>WRITE_EXTERNAL_STORAGE</code> 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	Yes, though the other app needs the <code>READ_EXTERNAL_STORAGE</code> permission	No
Documents and other files	Other types of shareable content, including downloaded files	Storage Access Framework	None	Yes, through the system file picker	No
App preferences	Key-value pairs	Jetpack Preferences library	None	No	Yes
Database	Structured data	Room persistence library	None	No	Yes

Selecting Storage Type

- **How much space does your data require?**

- Internal storage has limited space for app-specific data. Use other types of storage if you need to save a substantial amount of data.

- **How reliable does data access need to be?**

- If your app's basic functionality requires certain data, such as when your app is starting up, place the data within internal storage directory or a database. App-specific files that are stored in external storage aren't always accessible because some devices allow users to remove a physical device that corresponds to external storage.

- **What kind of data do you need to store?**

- If you have data that's only meaningful for your app, use app-specific storage. For shareable media content, use shared storage so that other apps can access the content. For structured data, use either preferences (for key-value data) or a database (for data that contains more than 2 columns).

- **Should the data be private to your app?**

- When storing sensitive data—data that shouldn't be accessible from any other app—use internal storage, preferences, or a database. Internal storage has the added benefit of the data being hidden from users.

App Specific Storage

- There are **Two types** of App Specific Storage
- **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 part of external storage instead.

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
- Another contains your app's cached files.
- Your app doesn't require any system permissions to read and write to files in these directories.
- You can use the **File API** to access and store files.
- Your app's ordinary, persistent files reside in a directory that you can access using the **FilesDir** property of a context object.

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

Internal Storage (Using Stream)

- As an alternative to using the **File API** for **storing data**, you can call **openFileOutput()** to get a **FileOutputStream** that writes to a file within the FilesDir directory.

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

- To allow other apps to access files stored in this directory within internal storage, use a FileProvider with the FLAG_GRANT_READ_URI_PERMISSION attribute.

Internal Storage (Using Stream)

- To read a file as a stream, use `openFileInput()` to get a `FileInputStream` :

```
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();
}
```


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
- Another contains your app's cached files.
- 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.
- 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

External Storage (Storage Availability)

- 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.

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

// Checks if a volume containing external storage is available to at least read.
private boolean isExternalStorageReadable() {
    return Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED) ||
        Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED_READ_ONLY);
}
```

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

External Storage

- To access app-specific files from external storage, call `getExternalFilesDir()`.

```
File appSpecificExternalDir = new File(context.getExternalFilesDir(), filename);
```

- 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:

```
@Nullable
File getAppSpecificAlbumStorageDir(Context context, String albumName) {
    // Get the pictures directory that's inside the app-specific directory on
    // 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;
}
```

Public Directories

The Environment class has constants for common public directories:

- `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 notifications that the user can select
- `Environment.DIRECTORY_PICTURES`
- `Environment.DIRECTORY_PODCASTS`
- `Environment.DIRECTORY_RINGTONES`

Shared Preferences

- If you have a relatively small collection of key-values that you'd like to save, you should use the SharedPreferences APIs.
 - A SharedPreferences object points to a file containing key-value pairs and provides simple methods to read and write them.
 - You can create a new shared preference file or access an existing one by calling one of these methods:
1. **getSharedPreferences()** — Use this if you need multiple shared preference files identified by name, which you specify with the first parameter. You can call this from any Context in your app.

```
Context context = getActivity();  
SharedPreferences sharedPref = context.getSharedPreferences(  
    getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

- When naming your shared preference files, you should use a name that's uniquely identifiable to your app. An easy way to do this is prefix the file name with your application ID. For example:
"com.example.myapp.PREFERENCE_FILE_KEY"

Shared Preferences

- 2. **getPreferences()** — Use this from an Activity if you need to use only one shared preference file for the activity. Because this retrieves a default shared preference file that belongs to the activity, you don't need to supply a name.

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

- To write to a shared preferences file, create a **SharedPreferences.Editor** by calling `edit()` on your `SharedPreferences`.

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);  
SharedPreferences.Editor editor = sharedPref.edit();  
editor.putInt(getString(R.string.saved_high_score_key), newHighScore);  
editor.apply();
```



Key

Value

ANDROID Data Storage & Access






Part 2

Shared Storage

- Apps allow users to contribute and access media that's available on an external storage volume
- The MediaStore API provides an optimized index into media collections that allows for retrieving and updating these media files more easily.
- The system provides standard public directories for these kinds of files, so the user has a common location for all their photos, another common location for all their music and audio files, and so on.
- Even after your app is uninstalled, these files remain on the user's device.
- 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.






Shared Storage

- To interact with the media store abstraction, use a **ContentResolver** object that you retrieve from your app's context:
- The system automatically scans an external storage volume and adds media files to the following well-defined collections:
- Images, including photographs and screenshots, which are stored in the DCIM/ and Pictures/ directories. The system adds these files to the **MediaStore.Images** table.
- Videos, which are stored in the DCIM/, Movies/, and Pictures/ directories. The system adds these files to the **MediaStore.Video** table.

```
String[] projection = new String[] {  
    media-database-columns-to-retrieve   
};  
String selection = sql-where-clause-with-placeholder-variables   
String[] selectionArgs = new String[] {  
    values-of-placeholder-variables   
};  
String sortOrder = sql-order-by-clause   
  
Cursor cursor = getApplicationContext().getContentResolver().query(  
    MediaStore.media-type , 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 item itself.  
}
```


Shared Storage

- Audio files, which are stored in the Alarms/, Audiobooks/, Music/, Notifications/, Podcasts/, and Ringtones/ directories, as well as audio playlists that are in the Music/ or Movies/ directories. The system adds these files to the **MediaStore.Audio** table.
- Downloaded files, which are stored in the Download/ directory. On devices that run Android 10 (API level 29) and higher, these files are stored in the **MediaStore.Downloads** table. This table isn't available on Android 9 (API level 28) and lower.

```
String[] projection = new String[] {  
    media-database-columns-to-retrieve   
};  
String selection = sql-where-clause-with-placeholder-variables   
String[] selectionArgs = new String[] {  
    values-of-placeholder-variables   
};  
String sortOrder = sql-order-by-clause   
  
Cursor cursor = getApplicationContext().getContentResolver().query(  
    MediaStore.media-type , 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 item itself.  
}
```

Database (Room Library)

- Room provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.
- Apps that handle non-trivial amounts of structured data can benefit greatly from persisting that data locally.
- When the device cannot access the network, the user can still browse that content while they are offline.
- Any user-initiated content changes are then synced to the server after the device is back online.
- SQLite is fairly low-level and require a great deal of time and effort to use.
- Room takes care of these concerns so it is **highly recommended** to use Room instead of SQLite.
- No permissions are required for storing and accessing data.
- Other apps can not access data that are stored in these local databases.

Room Dependencies

- To use Room in your app, add the following dependencies to your app's build.gradle file:

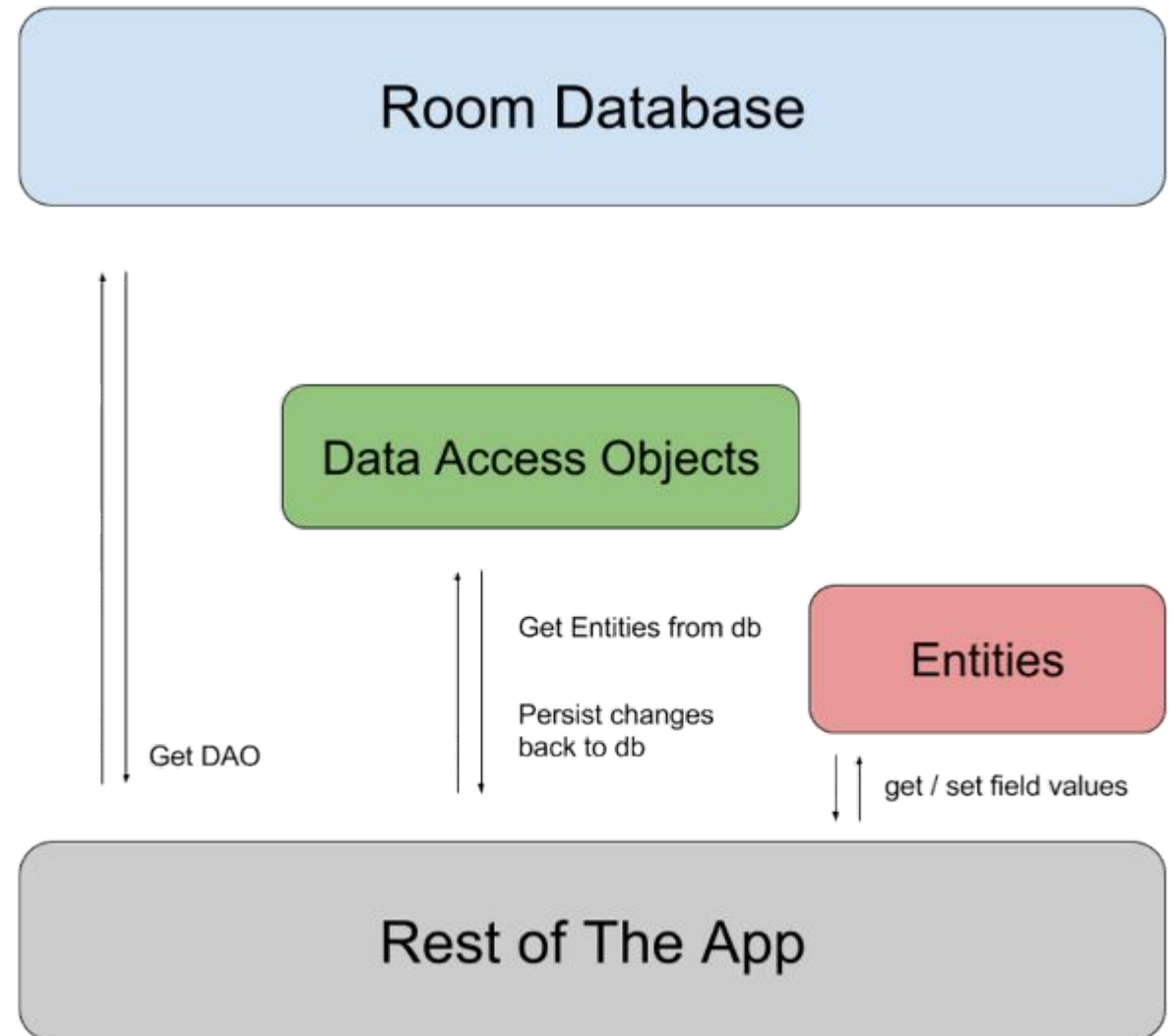
```
dependencies {  
    def room_version = "2.2.5"  
  
    implementation "androidx.room:room-runtime:$room_version"  
    annotationProcessor "androidx.room:room-compiler:$room_version"  
  
    // optional - RxJava support for Room  
    implementation "androidx.room:room-rxjava2:$room_version"  
  
    // optional - Guava support for Room, including Optional and ListenableFuture  
    implementation "androidx.room:room-guava:$room_version"  
  
    // optional - Test helpers  
    testImplementation "androidx.room:room-testing:$room_version"  
}
```

Room Components

- There are 3 major components in Room:
 1. **Database**: Contains the database holder and serves as the main access point for the underlying connection to your app's persisted, relational data.
 2. **Entity**: Represents a table within the database.
 3. **DAO**: Contains the methods used for accessing the database.

Room Components

- The app uses the Room database to get the data access objects, or DAOs, associated with that database.
- The app then uses each DAO to get entities from the database and save any changes to those entities back to the database.
- Finally, the app uses an entity to get and set values that correspond to table columns within the database.:



Entity

- Represents a table within the database.
- **@Entity**
- Primary key
- Column Info
- Table name

```
@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;
}

@Entity
public class Library {
    @PrimaryKey public long libraryId;
    public long userOwnerId;
}
```


Data Access Objects (Dao)

Instance 1

```
@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);
}
```

Instance 2

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
    public User[] loadAllUsersBetweenAges(int minAge, int maxAge);

    @Query("SELECT * FROM user WHERE first_name LIKE :search " +
        "OR last_name LIKE :search")
    public List<User> findUserWithName(String search);
}
```

Instance 3

```
@Dao
public interface MyDao {
    @Query("SELECT user.name AS userName, pet.name AS petName " +
        "FROM user, pet " +
        "WHERE user.id = pet.user_id")
    public LiveData<List<UserPet>> loadUserAndPetNames();
}
```

Dao contains the methods used for accessing the database.

Database

- The class that's annotated with **@Database** should satisfy the following conditions:
 - ❑ Be an abstract class that extends **RoomDatabase**.
 - ❑ Include the list of entities associated with the database within the annotation.
 - ❑ Contain an abstract method that has 0 arguments and returns the class that is annotated with **@Dao**.
 - ❑ At runtime, you can acquire an instance of Database by calling `Room.databaseBuilder()` or `Room.inMemoryDatabaseBuilder()`.

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

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


THANK YOU