ISLAMIC UNIVERSITY OF TECHNOLOGY

Organization of Islamic Cooperation

Board Bazar, Gazipur

Assignment 02

CSE 4643

CSE'18

11th March, 2022

**Distinguish the differences between Internal Storage and External Storage of App-Specific Storage with examples and codes.**

Differences:

Both the internal and external storage have dedicated directories for app-specific data storage. There are several differences between these two storage types, such as:

* Storage Space - The internal storage has a limited amount of space while the external storage has a comparatively large amount of space.
* Access Reliability – The external storage can by physically removed in some devices, meaning it cannot be relied on to always be accessible. The internal storage cannot be removed. Because of this, data essential to the basic functionality of an application should be stored on the internal storage.
* Access Permissions – For Android 4.4 and above, neither storage type requires any permissions to use, but for Android versions below that, permissions are required to use the external storage.
* Data Privacy – The internal storage is preferable for sensitive data since it is hidden from users as well as encrypted on Android 10 and above. Additionally, other applications are unable to access the app-specific storage of the internal storage, whereas they can access the app-specific storage of the external storage with the appropriate permissions.

Examples:

With these points in mind, an example of a use case for internal storage would be storing an API key to access an online service. These keys are provided to the developer and should be kept secret. Since they require a very small amount of space and should be kept secure, the app-specific internal storage is an appropriate place for them. Additionally, it is possible that the API access is essentially to running the application, which would be another point in favor of using the app-specific internal storage.

On the flip side, if the application creates documents of a special format that only the application itself can use, these documents should be kept on the app-specific external storage. This is because the documents may be large, but they are unlikely to be essential to the application or to contain any highly sensitive information.

Code:

For the internal storage, we can use the File API or input and output streams. The code below expands on the example mentioned above, showing how to access a small file containing a single line with an API key using an input stream.

String API\_KEY\_FILE = "mySecretAPIKey";  
String API\_KEY = "";  
  
private void getKey() throws IOException {  
 FileInputStream fis = getApplicationContext().openFileInput(API\_KEY\_FILE);  
 InputStreamReader inputStreamReader = new InputStreamReader(fis);  
 BufferedReader reader = new BufferedReader(inputStreamReader);  
 API\_KEY = reader.readLine();  
}

JAVA

For the external storage, we can use the same methods, but we just need to make a few changes. Firstly, we need to check that the external storage is writable or at least readable. Secondly, a specific directory should be used based on the data type of the content we are trying to read or write. The code below expands on the example mentioned above, showing how to write to a custom document format using the File API.

private Boolean externalStorageWritable() {  
 String state = Environment.*getExternalStorageState*();  
 return state.equals(Environment.*MEDIA\_MOUNTED*);  
}  
  
private void saveFile(String fileName, String content) throws IOException {  
 if (!externalStorageWritable()) return;  
 Context context = getApplicationContext();  
 String directory = Environment.*DIRECTORY\_DOCUMENTS*;  
 File file = new File(context.getExternalFilesDir(directory), fileName);  
 FileOutputStream fos = new FileOutputStream(file);  
 fos.write(content.getBytes());  
}

JAVA

**Write a java code for storing music in** Environment.*DIRECTORY\_MUSIC* **within external storage of your application.**

private Boolean externalStorageWritable() {  
 String state = Environment.*getExternalStorageState*();  
 return state.equals(Environment.*MEDIA\_MOUNTED*);  
}  
  
private void saveFile(File audioFile) throws IOException {  
 if (!externalStorageWritable()) return;  
  
 Context context = getApplicationContext();  
 String directory = Environment.*DIRECTORY\_MUSIC*;  
 String fileName = audioFile.getName();  
 File out = new File(context.getExternalFilesDir(directory), fileName);  
 FileOutputStream fos = new FileOutputStream(out);  
  
 FileInputStream fis = new FileInputStream(audioFile);  
 InputStreamReader inputStreamReader = new InputStreamReader(fis);  
 BufferedReader reader = new BufferedReader(inputStreamReader);  
 String line = reader.readLine();  
 while (line != null) {  
 fos.write(line.getBytes());  
 line = reader.readLine();  
 }  
}

JAVA

**Elucidate the Room Database with its required components and define the relationship with examples & codes.**

The Room library provides an abstraction over the SQLite database, making it easier to use. It is a local database that can even be used offline to store non-trivial amounts of structured data. Other applications are unable to access this data, and the application does not need any additional permissions to use the database.

The Room library has three major components, Databases, Entities and DAOs. The database object contains the database holder and serves as the main access point to the underlying connection to the actual database. Entity objects represent tables from the database. DAOs provide methods to access the database and manipulate its tables.

Suppose we have two tables in our database, one that stores user information and another that stores user authentication information. Thus, we would require two entities.

@Entity  
class UserEntity {  
 @PrimaryKey public int id;  
 public String name;  
 public int age;  
}  
  
@Entity  
class AuthEntity {  
 @PrimaryKey public int id;  
 public String pass;  
}

JAVA

We can use one DAO that has methods used to authenticate users and another DAO that allows retrieval of user information, which we will use after users have been authenticated. The code below only shows one method for each DAO, but practical applications would have multiple methods for each DAO (e.g. a password reset method for the Authentication DAO).

@Dao  
interface *UserDao* {  
 @Query("SELECT \* FROM UserEntity WHERE id = :id")  
 public User retrieveUser(int id);  
}  
  
@Dao  
interface *AuthDao* {  
 @Query("SELECT COUNT(\*) FROM AuthEntity WHERE id = :id AND pass = :pass")  
 public int authenticate(int id, String pass);  
}

JAVA

Notice that the retrieveUser method in *UserDao* returns a User object, not a UserEntity object. UserEntity describes a table. The actual objects returned by the query are records from the table and must be encapsulated by a separate class. This is a class we must create ourselves. Thus, the User class is born.

public class User {  
 @Embedded  
 UserEntity user;  
}

JAVA

Essentially, we created a class and told it to include all the properties of the UserEntity entity.

Finally, we combine this information into a Database.

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

JAVA

Once this is done, we can use the database.

public User getUser(int userId, String password) {  
 // retrieve database  
 AppDatabase db = Room.*databaseBuilder*(  
 getApplicationContext(),  
 AppDatabase.class,  
 "app-database"  
 ).build();  
  
 // verify credentials  
 if (db.authDao().authenticate(userId, password) == 1)  
 return db.userDao().retrieveUser(userId);  
 else return null;  
}

JAVA