BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Mobile development and security

Session 7

Karim Karimov
Lecturer

Data and file storage

Android uses a file system that's similar to disk-based file systems on other platforms. The system provides several options for you to save your 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.

Data and file storage

	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, `getFilesDir()` or `getCacheDir()` From external storage, `getExternalFilesDir()` or `getExternalCacheDir()`	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	`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	Yes, though the other app needs the `READ_EXTERNAL_STORAGE` permission	No

Data and file storage

	Type of content	Access method	Permissions needed	Can other apps access?	Files removed on app uninstall?
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

Data and file storage

The solution you choose depends on your specific needs:

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.

Data and file 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.

Categories of storage locations

Android provides two types of physical storage locations: ***internal storage*** and ***external storage***. On most devices, internal storage is smaller than external storage. However, internal storage is always available on all devices, making it a more reliable place to put data on which your app depends.
Removable volumes, such as an SD card, appear in the file system as part of external storage. Android represents these devices using a path, such as **/sdcard**.

Permissions and access to external storage

Android defines the following storage-related permissions: **READ_EXTERNAL_STORAGE**, **WRITE_EXTERNAL_STORAGE**, and **MANAGE_EXTERNAL_STORAGE**.
On earlier versions of Android, apps needed to declare the **READ_EXTERNAL_STORAGE** permission to access any file outside the app-specific directories on external storage. Also, apps needed to declare the **WRITE_EXTERNAL_STORAGE** permission to write to any file outside the app-specific directory.

Permissions and access to external storage

More recent versions of Android rely more on a file's purpose than its location for determining an app's ability to access, and write to, a given file. In particular, if your app targets Android 11 (API level 30) or higher, the **WRITE_EXTERNAL_STORAGE** permission doesn't have any effect on your app's access to storage. This purpose-based storage model improves user privacy because apps are given access only to the areas of the device's file system that they actually use.

Android 11 introduces the **MANAGE_EXTERNAL_STORAGE** permission, which provides write access to files outside the app-specific directory and MediaStore. To learn more about this permission, and why most apps don't need to declare it to fulfill their use cases, see the guide on how to manage all files on a storage device.

Save key-value data

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. Each SharedPreferences file is managed by the framework and can be private or shared.

You can create a new shared preference file or access an existing one by calling one of these methods:

- **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.
- **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.

Save key-value data

For example, the following code accesses the shared preferences file that's identified by the resource string **R.string.preference_file_key** and opens it using the private mode so the file is accessible by only your app:

```
val sharedPref = activity?.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"** Alternatively, if you need just one shared preference file for your activity, you can use the **getPreferences()** method:

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE)
```

Write to shared preferences

To write to a shared preferences file, create a SharedPreferences.Editor by calling **edit()** on your SharedPreferences. Pass the keys and values you want to write with methods such as **putInt()** and **putString()**. Then call **apply()** or **commit()** to save the changes. For example:

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
with (sharedPref.edit()) {
    putInt(getString(R.string.saved_high_score_key), newHighScore)
    apply()
}
```

apply() changes the in-memory SharedPreferences object immediately but writes the updates to disk asynchronously. Alternatively, you can use **commit()** to write the data to disk synchronously. But because **commit()** is synchronous, you should avoid calling it from your main thread because it could pause your UI rendering.

Read from shared preferences

To retrieve values from a shared preferences file, call methods such as **getInt()** and **getString()**, providing the key for the value you want, and optionally a default value to return if the key isn't present. For example:

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
val defaultValue = resources.getInteger(R.integer.saved_high_score_default_key)
val highScore = sharedPref.getInt(getString(R.string.saved_high_score_key), defaultValue)
```

Save data in a local database using Room

Apps that handle non-trivial amounts of structured data can benefit greatly from persisting that data locally. The most common use case is to cache relevant pieces of data so that when the device cannot access the network, the user can still browse that content while they are offline.

The Room persistence library provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite. In particular, Room provides the following benefits:

- Compile-time verification of SQL queries.
- Convenience annotations that minimize repetitive and error-prone boilerplate code.
- Streamlined database migration paths.

Because of these considerations, we highly recommend that you use Room instead of using the SQLite APIs directly.

Setup Room

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

```
dependencies {
    def room_version = "2.5.0"

    implementation "androidx.room:room-runtime:$room_version"
    annotationProcessor "androidx.room:room-compiler:$room_version"

    // To use Kotlin annotation processing tool (kapt)
    kapt "androidx.room:room-compiler:$room_version"
    // To use Kotlin Symbol Processing (KSP)
    ksp "androidx.room:room-compiler:$room_version"

    // optional - RxJava2 support for Room
    implementation "androidx.room:room-rxjava2:$room_version"

    // optional - RxJava3 support for Room
    implementation "androidx.room:room-rxjava3:$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"

    // optional - Paging 3 Integration
    implementation "androidx.room:room-paging:$room_version"
}
```

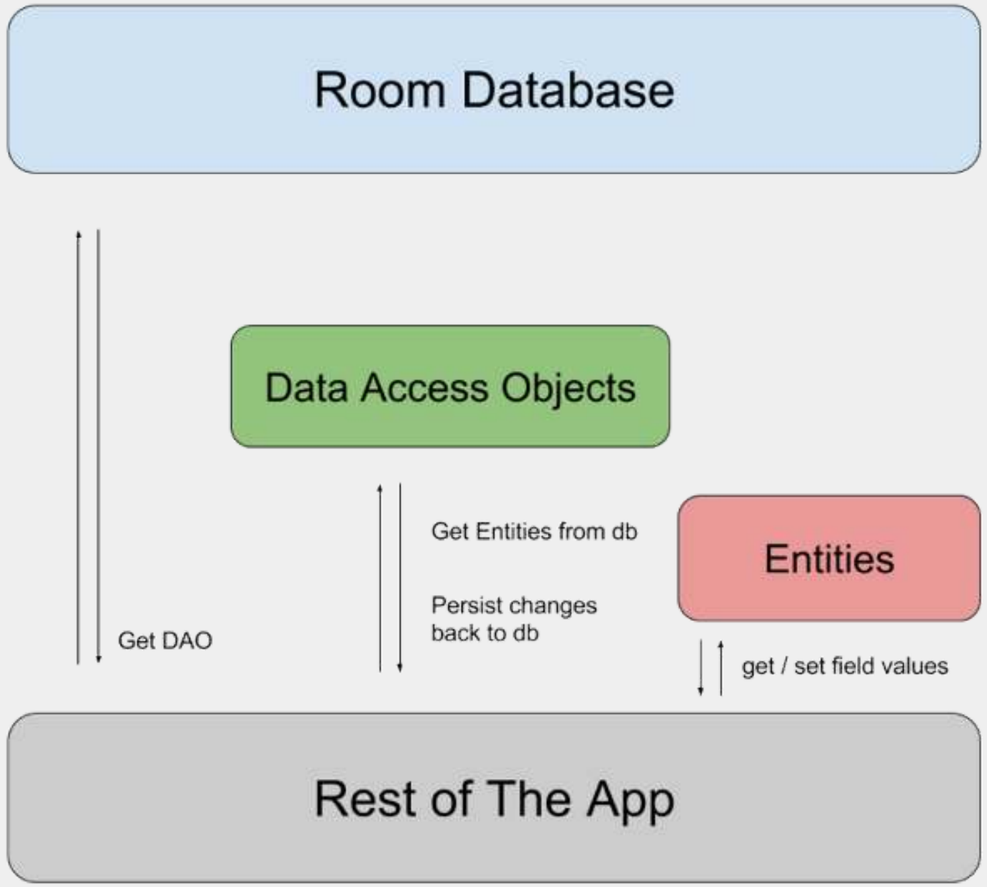
Primary components

There are three major components in Room:

- The **database** class that holds the database and serves as the main access point for the underlying connection to your app's persisted data.
- **Data entities** that represent tables in your app's database.
- **Data access objects (DAOs)** that provide methods that your app can use to query, update, insert, and delete data in the database.

The database class provides your app with instances of the DAOs associated with that database. In turn, the app can use the DAOs to retrieve data from the database as instances of the associated data entity objects. The app can also use the defined data entities to update rows from the corresponding tables, or to create new rows for insertion.

Primary components



Data entity

The following code defines a User data entity. Each instance of User represents a row in a user table in the app's database.

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

Data access object (DAO)

The following code defines a DAO called UserDao. UserDao provides the methods that the rest of the app uses to interact with data in the user table.

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first
AND " +
           "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

Database

The following code defines an AppDatabase class to hold the database. AppDatabase defines the database configuration and serves as the app's main access point to the persisted data. The database class must satisfy the following conditions:

- The class must be annotated with a **@Database** annotation that includes an entities array that lists all of the data entities associated with the database.
- The class must be an abstract class that **extends RoomDatabase**.
- For each **DAO class** that is associated with the database, the database class must define an abstract method that has zero arguments and returns an instance of the DAO class.

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Usage

After you have defined the data entity, the DAO, and the database object, you can use the following code to create an instance of the database:

```
val db = Room.databaseBuilder(
        applicationContext,
        AppDatabase::class.java, "database-name"
    ).build()
```

You can then use the abstract methods from the AppDatabase to get an instance of the DAO. In turn, you can use the methods from the DAO instance to interact with the database:

```
val userDao = db.userDao()
val users: List<User> = userDao.getAll()
```

Anatomy of an entity

You define each Room entity as a class that is annotated with **@Entity**. A Room entity includes fields for each column in the corresponding table in the database, including one or more columns that comprise the **primary key**.
The following code is an example of a simple entity that defines a User table with columns for **ID**, **first name**, and **last name**:

```
@Entity
data class User(
    @PrimaryKey val id: Int,

    val firstName: String?,
    val lastName: String?
)
```

Table name

By default, Room uses the class name as the database table name. If you want the table to have a different name, set the tableName property of the **@Entity** annotation. Similarly, Room uses the field names as column names in the database by default. If you want a column to have a different name, add the **@ColumnInfo** annotation to the field and set the name property. The following example demonstrates custom names for tables and columns:

```
@Entity(tableName = "users")
data class User (
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

Composite primary key

If you need instances of an entity to be uniquely identified by a combination of multiple columns, you can define a composite primary key by listing those columns in the **primaryKeys** property of **@Entity**:

```
@Entity(primaryKeys = ["firstName", "lastName"])
data class User(
    val firstName: String?,
    val lastName: String?
)
```

Anatomy of a DAO

You can define each DAO as either an interface or an abstract class. For basic use cases, you should usually use an interface. In either case, you must always annotate your DAOs with **@Dao**. DAOs don't have properties, but they do define one or more methods for interacting with the data in your app's database.
The following code is an example of a simple DAO that defines methods for inserting, deleting, and selecting User objects in a Room database:

```
@Dao
interface UserDao {
    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)

    @Query("SELECT * FROM user")
    fun getAll(): List<User>
}
```

Insert

The **@Insert** annotation allows you to define methods that insert their parameters into the appropriate table in the database. The following code shows examples of valid **@Insert** methods that insert one or more User objects into the database. Each parameter for an @Insert method must be either an instance of a Room data entity class annotated with **@Entity** or a collection of data entity class instances.

If the **@Insert** method receives a single parameter, it can return a long value, which is the new rowId for the inserted item. If the parameter is an array or a collection, then the method should return an array or a collection of long values instead, with each value as the rowId for one of the inserted items.

```
@Dao
interface UserDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertUsers(vararg users: User)

    @Insert
    fun insertBothUsers(user1: User, user2: User)

    @Insert
    fun insertUsersAndFriends(user: User,
friends: List<User>)
}
```

Queries

The following code defines a method that uses a simple SELECT query to return all of the User objects in the database:

```
@Query("SELECT * FROM user")
fun loadAllUsers(): Array<User>
```

Most of the time, you only need to return a subset of the columns from the table that you are querying. For example, your UI might display just the first and last name for a user instead of every detail about that user. In order to save resources and streamline your query's execution, you should only query the fields that you need.

```
@Query("SELECT first_name, last_name FROM user")
fun loadFullName(): List<NameTuple>
```

Queries

Most of the time, your DAO methods need to accept parameters so that they can perform filtering operations. Room supports using method parameters as bind parameters in your queries.
For example, the following code defines a method that returns all of the users above a certain age:

```
@Query("SELECT * FROM user WHERE age > :minAge")
fun loadAllUsersOlderThan(minAge: Int): Array<User>

@Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
fun loadAllUsersBetweenAges(minAge: Int, maxAge: Int): Array<User>

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

@Query("SELECT * FROM user WHERE region IN (:regions)")
fun loadUsersFromRegions(regions: List<String>): List<User>
```

Async queries

To prevent queries from blocking the UI, Room does not allow database access on the main thread. This restriction means that you must make your DAO queries asynchronous. The Room library includes integrations with several different frameworks to provide asynchronous query execution.

DAO queries fall into three categories:

- **One-shot write** queries that insert, update, or delete data in the database.
- **One-shot read queries** that read data from your database only once and return a result with the snapshot of the database at that time.
- **Observable read queries** that read data from your database every time the underlying database tables change and emit new values to reflect those changes.

Async queries

Query type	Kotlin language features	RxJava	Guava	Jetpack Lifecycle
One-shot write	Coroutines (`suspend`)	`Single<T>`, `Maybe<T>`, `Completable`	`ListenableFuture<T>`	N/A
One-shot read	Coroutines (`suspend`)	`Single<T>`, `Maybe<T>`	`ListenableFuture<T>`	N/A
Observable read	`Flow<T>`	`Flowable<T>`, `Publisher<T>`, `Observable<T>`	N/A	`LiveData<T>`

References

- **Storage overview** - https://developer.android.com/training/data-storage
- **Key-value store (Shared preferences)** - https://developer.android.com/training/data-storage/shared-preferences
- **RoomDB overview** - https://developer.android.com/training/data-storage/room
- **Room entities** - https://developer.android.com/training/data-storage/room/defining-data
- **Room DAO** - https://developer.android.com/training/data-storage/room/accessing-data
- **Async queries** - https://developer.android.com/training/data-storage/room/async-queries

BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

End of the session