Android Development with Kotlin

Room

Introduction:

Most of the times we want our app to have some persistent data, this is the data that we want the user to be able to see the next time they start our application. E.g. in our **Task list app** we would like to save the tasks that the user has entered. Similarly a **web browser** would like to **save bookmarks or browsing history**.

There are various ways of storing data in an Android application. Here are a few of those methods:

a. Shared Preferences
b. Internal File System
c. External Storage
d. SQLite Database
e. Network Server

We are not covering Network Server in these notes. Network Server just means that you can store this data in a webserver by sending the data over the network. In these notes we will go through the first four methods of data storage, some of which you have already studied and implemented.

Shared Preferences:

You have studied as well as implemented this concept. **Shared Preferences is a key value storage system. It is generally used to store user preferences and other small amount of data. We don't store a large amount of data in this storage, as we need to remember all the keys that are used to store data.** Along with this restriction we can't really search through the content like the way we will be able to in SQLite Databases.

Internal Storage:

We can use a **device's internal storage to store our data as well. Here internal storage doesn't mean the complete inbuilt storage. Internal storage is small memory,** which is used by Android for saving app data. Along with any SD Card, rest of the inbuilt storage is considered as external storage as well. **So, this was just a brief intro to this storage option, we will be covering this in details** in the upcoming lectures, so stay tuned:)

External Storage:

This includes any SD Card installed by the user as well as all user data like albums, ringtones, music falls into external storage as well.

a. Permissions: In our file explorer you will see that we have a folder called sdcard. Clicking on this folder doesn't show us any data. Try adding android.permission.READ_EXTERNAL_STORAGE permission in your manifest and install the app again. You will be able to see all the missing data in this folder now.

b. Reading: In case you want to read a specific directory from the external storage, you should first check if the external storage is available or not. You can use Environment.getExternalStorageState() function to get the current state and you can read from external storage if the state is equal to Environment.MEDIA_MOUNTED or ENVIRONMENT.MEDIA_MOUNTED_READ_ONLY. After which you can use Environemnt.getExternalStoragePublicDirectory function to get various libraries. Checkout the documentation here for more details.

c. Writing: You can save files like any other files as long you have the right permissions to do so. You would need android.permission.WRITE_EXTERNAL_STORAGE to write to external storage.

SQLite Database:

A lot of times data we want to store has a fixed structure to it. **E.g. our task list app has a bunch of tasks, where each task has a name, duration, date etc.** In these cases we will store the data in a database. Android supports SQLite databases; this is the most popular storage method.

SQLite VS SQL:

Sqlite is very light version of SQL supporting many features of SQL. Basically it has been developed for small devices like mobile phones, tablets etc. SQLite is a third party ,open-sourced and in-process database engine. SQL Server Compact is from Microsoft, and is a stripped-down version of SQL Server.

SQLite Introduction:

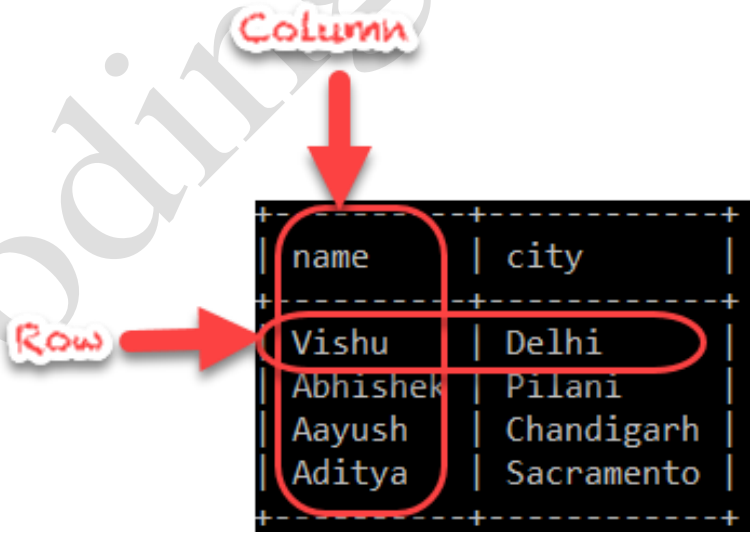
SQLite is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine. It is the most used database engine in the world. It is an in-process library and its code is publicly available. It is free for use for any purpose, commercial or private. It is basically an embedded SQL database engine. Ordinary disk files can be easily read and write by SQLite because it does not have any separate server like SQL. **The SQLite database file format is cross-platform so that anyone can easily copy a database between 32-bit and 64-bit systems. Due to all these features, it is a popular choice as an Application File Format.**

What are tables in a database?

In databases, tables are actually the containers. These containers actually store the data. For the purpose of a definition, we can term tables as a structured format in a database which holds the data. To put in the importance of tables in SQLite in a nutshell, we can say that tables are the heart of SQLite. Without tables, you cannot store any data in a database.

Tables are actually made up of two things:

- **Columns**
- **Rows**



Few SQLite operations examples:

1) **Creating a table "Student":**

```
CREATE TABLE STUDENT(
ID INT PRIMARY KEY NOT NULL,
NAME TEXT NOT NULL,
AGE INT NOT NULL,
ADDRESS CHAR(50),
FEES REAL
);
```

2) **Inserting values into the table:**

```
INSERT INTO STUDENT (ID, NAME, AGE, ADDRESS, FEES)
VALUES (1, 'Sunil', 28, 'Mumbai', 20000.00);
```

3) **Extracting information from the table:**

```
SELECT * FROM STUDENT;
```

4) **Dropping a table:**

```
Drop Table STUDENT;
```

Here is a list of all the basic commands for you. Feel free to explore :)

https://www.tutorialspoint.com/sqlite/sqlite_syntax.htm

https://www.javatpoint.com/dbms-sql-command

https://www.sqlitetutorial.net/

As writing SQLite queries **directly in Android** requires **a lot of boilerplate code**, which the room persistence library saves us from. Thus, in this **lecture our prime focus will be on Room**, which we will be learning next

What is room and why to use it?

Room is a persistence library, part of the Android Architecture Components. It makes it easier to work with SQLite Database objects in your app, decreasing the amount of boilerplate code and verifying SQL queries at compile time.

Room annotations and main components:

- **@Entity**—Define our database tables
- **@DAO**—Provide an API for reading and writing data
- **@Database**—Represent a database holder

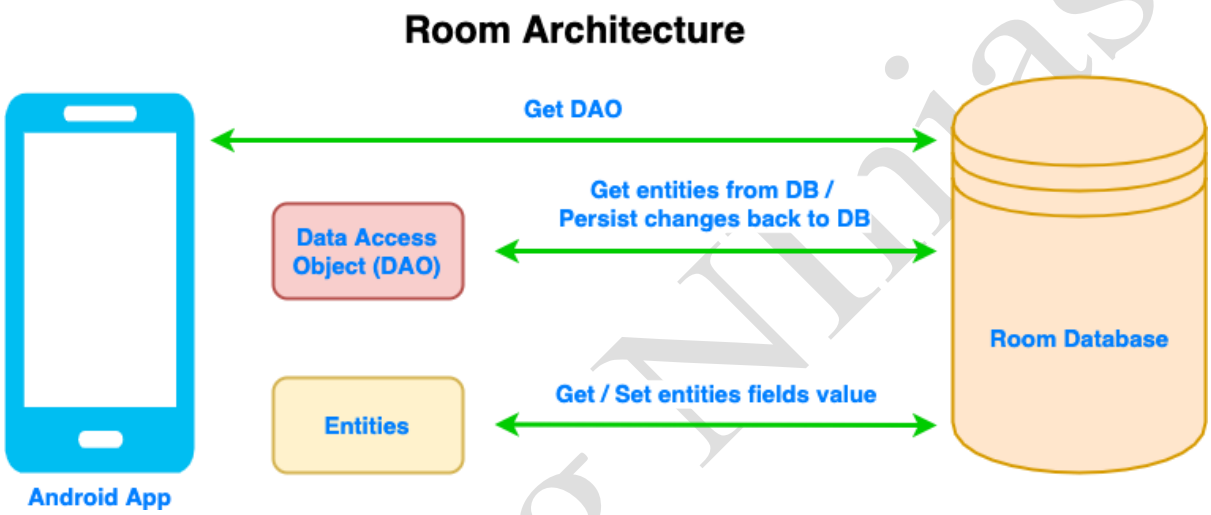


image src: https://gorillalogic.com/wp-content/uploads/2019/12/RoomDB_Transparent.png

Steps for setting up Room Database with an example:

Add the needed dependencies:

Room's dependencies are available via Google's new Maven repository, simply add it to the list of repositories in your main app/build.gradle file:

```
apply plugin: "kotlin-kapt"

dependencies {

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

    implementation "androidx.legacy:legacy-support-v4:1.0.0"

    implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0'

    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.0.0'

    kapt "androidx.room:room-compiler:$room_version"

}
```

Entity:

Room creates a table for each class annotated with **@Entity;** the fields in the class correspond to columns in the table. Therefore, the entity classes tend to be small model classes that don't contain any logic. Our **todo_items class** represents the model for the data in the database.

```
@Entity(tableName = "todo_items")
data class TodoEntity(
        @PrimaryKey(autoGenerate = true)
        var id: Int,

        @ColumnInfo(name = "title") var title: String,
        @ColumnInfo(name = "content") var content: String
)
```

DAO:

DAOs are responsible for defining the methods that access the database. With Room, we don't need all the Cursor related code and can simply define our queries using annotations in the TodoDao interface.

```
@Dao
interface TodoDao {
    @Query("SELECT * FROM todoentity")
    fun getAll(): List<TodoEntity>

    @Query("SELECT * FROM todoentity WHERE title LIKE :title")
    fun findByTitle(title: String): TodoEntity

    @Insert
    suspend fun insertAll(vararg todo: TodoEntity)

    @Delete
    suspend fun delete(todo: TodoEntity)

    @Update
    suspend fun updateTodo(vararg todos: TodoEntity)
}
```

Database:

After that we can start writing the database which contains all your DAOs as abstract methods. Here we define the version of the database and the entity and DAO that we are using.

You can also use more than one entity and define a custom build and invoke methods as you can see in this example.

```
@Database(
    entities = [TodoEntity::class, TaskEntry::class],
    version = 1
)
abstract class AppDatabase : RoomDatabase(){
    abstract fun TodoDao(): TodoDao
    abstract fun TaskDao(): TaskDao

    companion object {
        @Volatile private var instance: AppDatabase? = null
        private val LOCK = Any()

        operator fun invoke(context: Context)= instance ?:
synchronized(LOCK){
            instance ?: buildDatabase(context).also { instance = it}
        }

        private fun buildDatabase(context: Context) =
Room.databaseBuilder(context,
                AppDatabase::class.java, "todo-list.db")
                .build()
    }
}
```

Accessing the database:

After defining the database we can get an instance in our activity using the Room.databaseBuilder() method.

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

We don't need the Room.databaseBuilder() to get an instance of the second database example defined above. We just need to call the invoke() methode and pass the activity context.

```
val db = AppDatabase(this)
```

Now we can start using the database instance to access our *DAO* object.

```
GlobalScope.launch {
            db.todoDao().insertAll(TodoEntry("Title", "Content"))
            data = db.todoDao().getAll()

            data?.forEach {
                println(it)
            }
}
```

Testing your database:

Now we can start testing our database to make sure that the read and write functionality is functioning correctly.

```
@RunWith(AndroidJUnit4::class)
class EnityReadWriteTest {
    private lateinit var todoDao: TodoDao
    private lateinit var db: AppDatabase

    @Before
    fun createDb() {
        val context = InstrumentationRegistry.getContext()
        db = Room.inMemoryDatabaseBuilder(
                context, AppDatabase::class.java).build()
```

```
        todoDao = db.todoDao()
    }

    @After
    @Throws(IOException::class)
    fun closeDb() {
        db.close()
    }

    @Test
    @Throws(Exception::class)
    fun writeUserAndReadInList() {
        val todo: TodoEntry = TodoEntry("title", "detail")
        todoDao.insertAll(todo)
        val todoItem = todoDao.findByTitle(todo.title)
        assertThat(todoItem, equalTo(todo))
    }
}
```

Difference between SQLite and Room Persistence Library:

- In case of SQLite, there is no compile time verification of raw SQLite queries. But in Room there is SQL validation at compile time.
- As your schema changes, you need to update the affected SQL queries manually. Room solves this problem.
- You need to use lots of boilerplate code to convert between SQL queries and Java data objects. But Room maps our database objects to Java Object without boilerplate code.
- Room is built to work with LiveData and RxJava for data observation, while SQLite does not

We have covered all the concepts related Room in the lectures. To explore further please refer to the official documentation below:

https://developer.android.com/topic/libraries/architecture/room