Module Code & Title

CS6P05 Final Year Project MAD

Food Share - Android App

Artifact – Local database implementation

Student Details

Name: Sita Ram Thing

London Met Id: 22015892

College Id: NP01MA4S220003

Islington College, Kathmandu

24 April 2024

GitHub: git@github.com:FYP-23-24/22015892-SitaRamThing.git

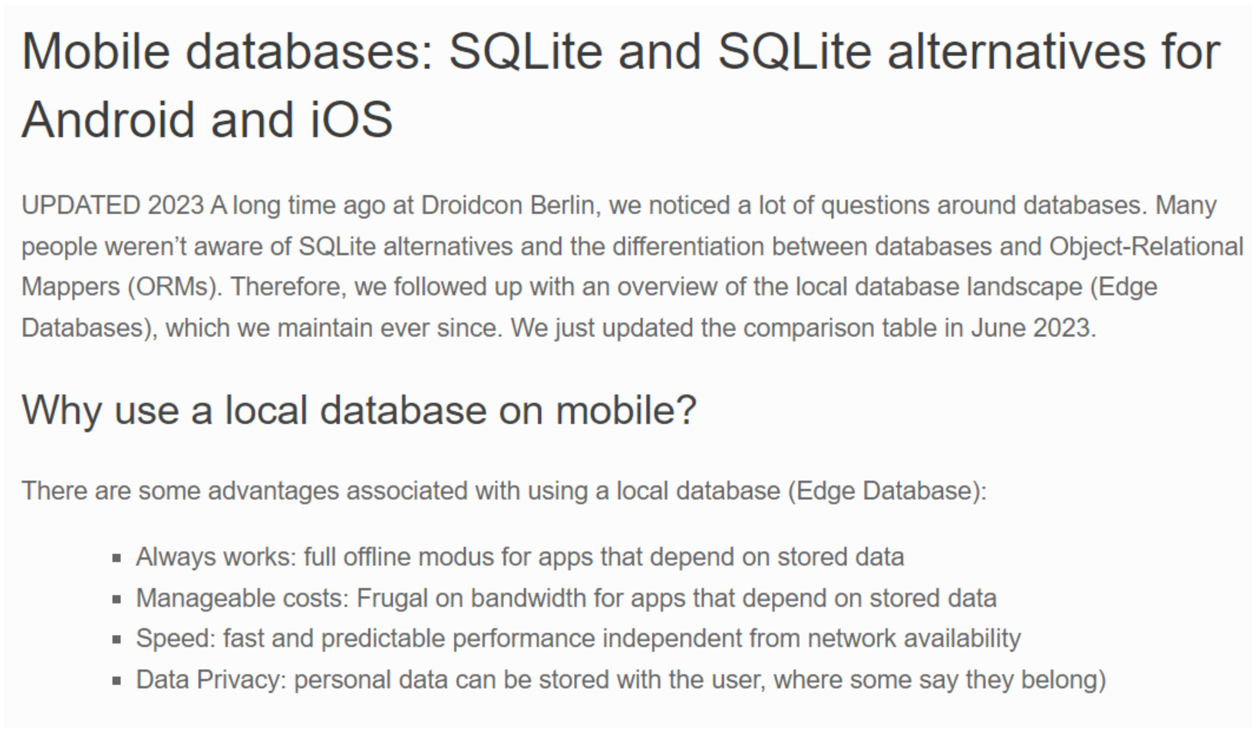
Contents

List of figure

1 Introduction

1.1 SQLite database

Figure 1: Sqlite database alternative.



Mobile databases: SQLite and SQLite alternatives for Android and iOS

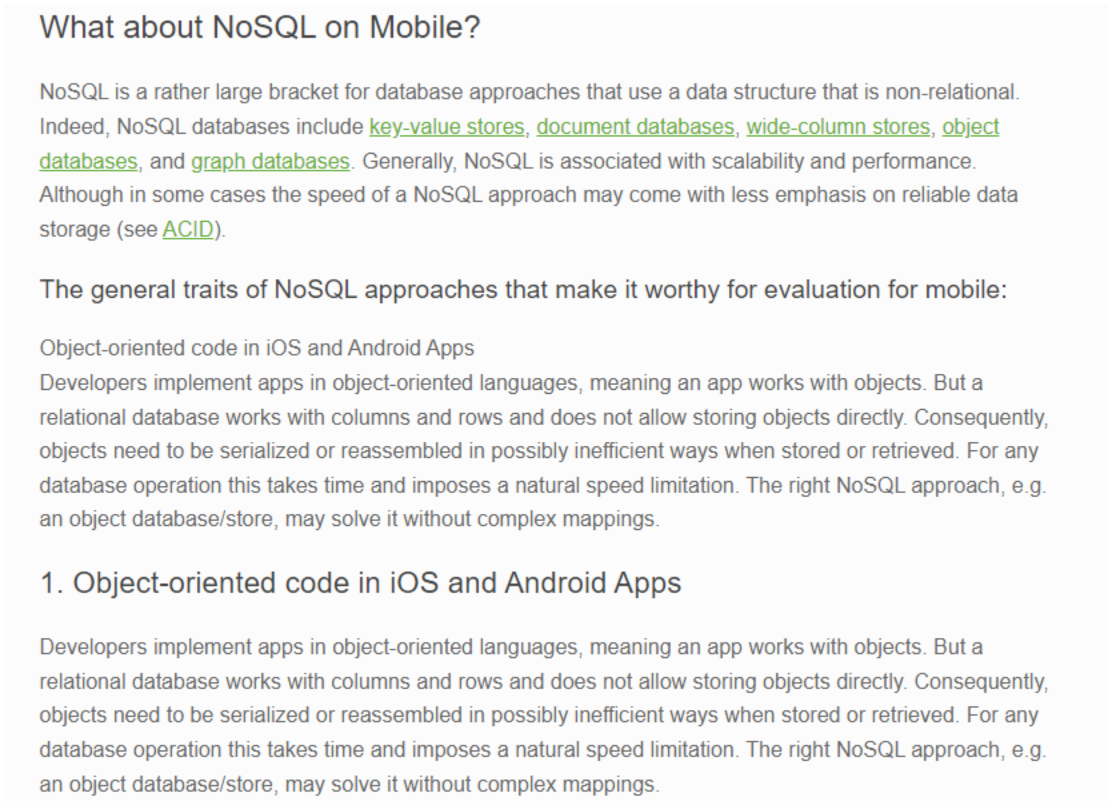
UPDATED 2023 A long time ago at Droidcon Berlin, we noticed a lot of questions around databases. Many people weren't aware of SQLite alternatives and the differentiation between databases and Object-Relational Mappers (ORMs). Therefore, we followed up with an overview of the local database landscape (Edge Databases), which we maintain ever since. We just updated the comparison table in June 2023.

Why use a local database on mobile?

There are some advantages associated with using a local database (Edge Database):

- Always works: full offline modus for apps that depend on stored data
- Manageable costs: Frugal on bandwidth for apps that depend on stored data
- Speed: fast and predictable performance independent from network availability
- Data Privacy: personal data can be stored with the user, where some say they belong)

Figure 2: Mobile database Android and ios



What about NoSQL on Mobile?

NoSQL is a rather large bracket for database approaches that use a data structure that is non-relational. Indeed, NoSQL databases include key-value stores, document databases, wide-column stores, object databases, and graph databases. Generally, NoSQL is associated with scalability and performance. Although in some cases the speed of a NoSQL approach may come with less emphasis on reliable data storage (see ACID).

The general traits of NoSQL approaches that make it worthy for evaluation for mobile:

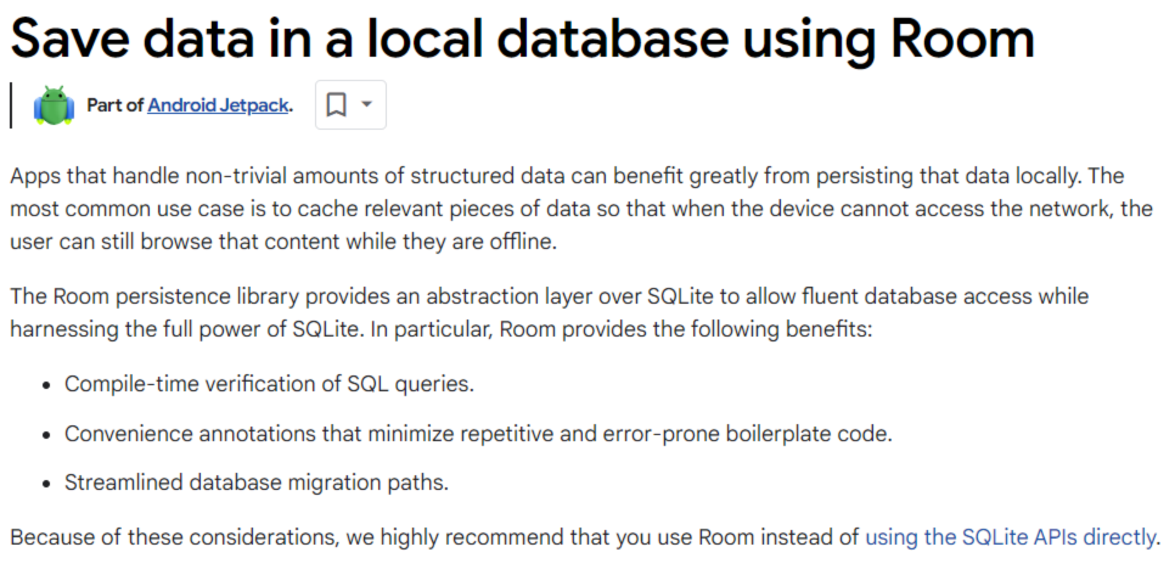
Object-oriented code in iOS and Android Apps
Developers implement apps in object-oriented languages, meaning an app works with objects. But a relational database works with columns and rows and does not allow storing objects directly. Consequently, objects need to be serialized or reassembled in possibly inefficient ways when stored or retrieved. For any database operation this takes time and imposes a natural speed limitation. The right NoSQL approach, e.g. an object database/store, may solve it without complex mappings.

1. Object-oriented code in iOS and Android Apps

Developers implement apps in object-oriented languages, meaning an app works with objects. But a relational database works with columns and rows and does not allow storing objects directly. Consequently, objects need to be serialized or reassembled in possibly inefficient ways when stored or retrieved. For any database operation this takes time and imposes a natural speed limitation. The right NoSQL approach, e.g. an object database/store, may solve it without complex mappings.

Figure 3: Objective-orientated code in Ios and Android

1.2 Room database



Save data in a local database using Room

Part of Android Jetpack.

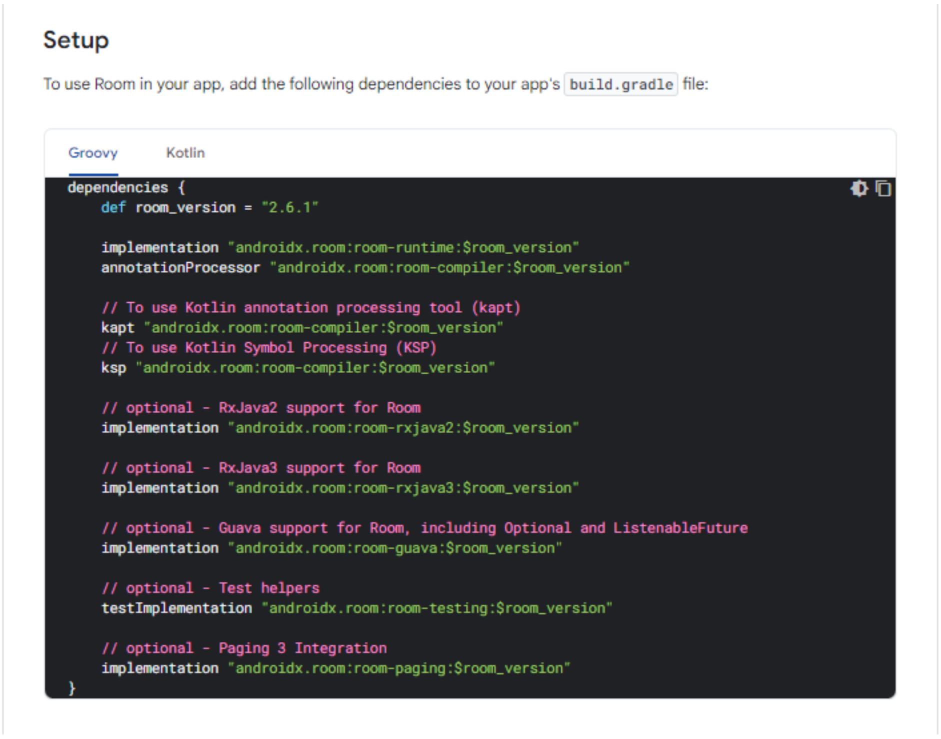
Apps that handle non-trivial amounts of structured data can benefit greatly from persisting that data locally. The most common use case is to cache relevant pieces of data so that when the device cannot access the network, the user can still browse that content while they are offline.

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.

Figure 4: About room database.

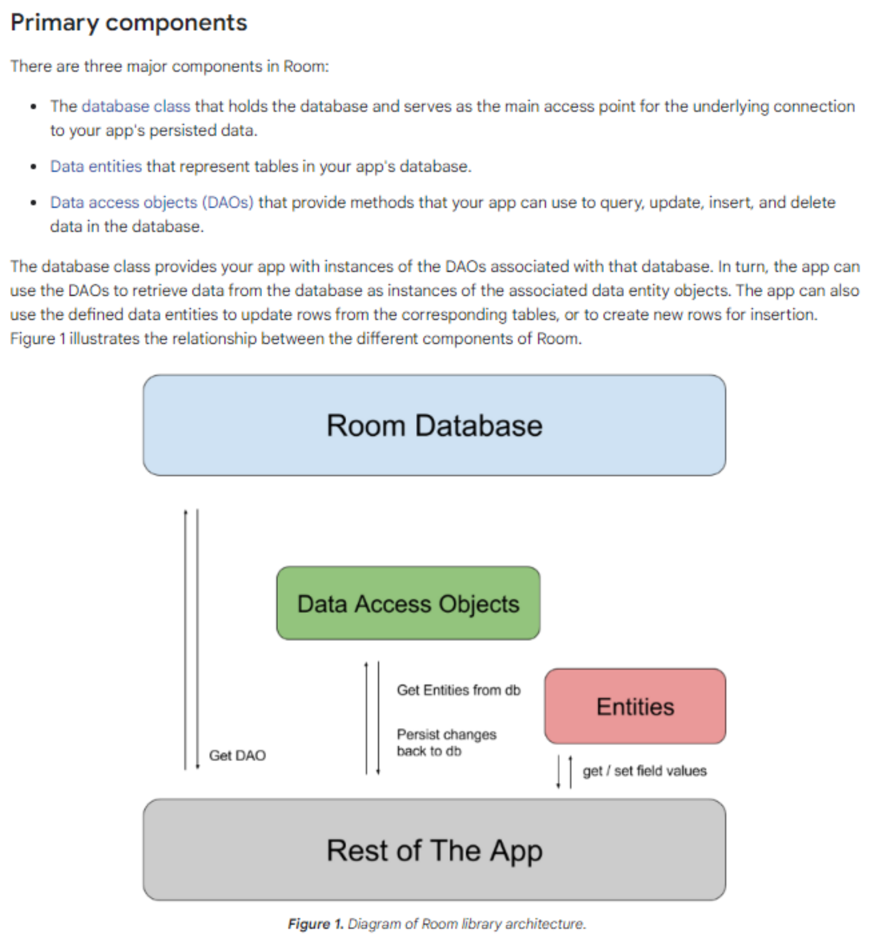


Setup

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

```
dependencies {
    def room_version = "2.6.1"

    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"
}
```

Figure 5: Room database dependency implementations



Primary components

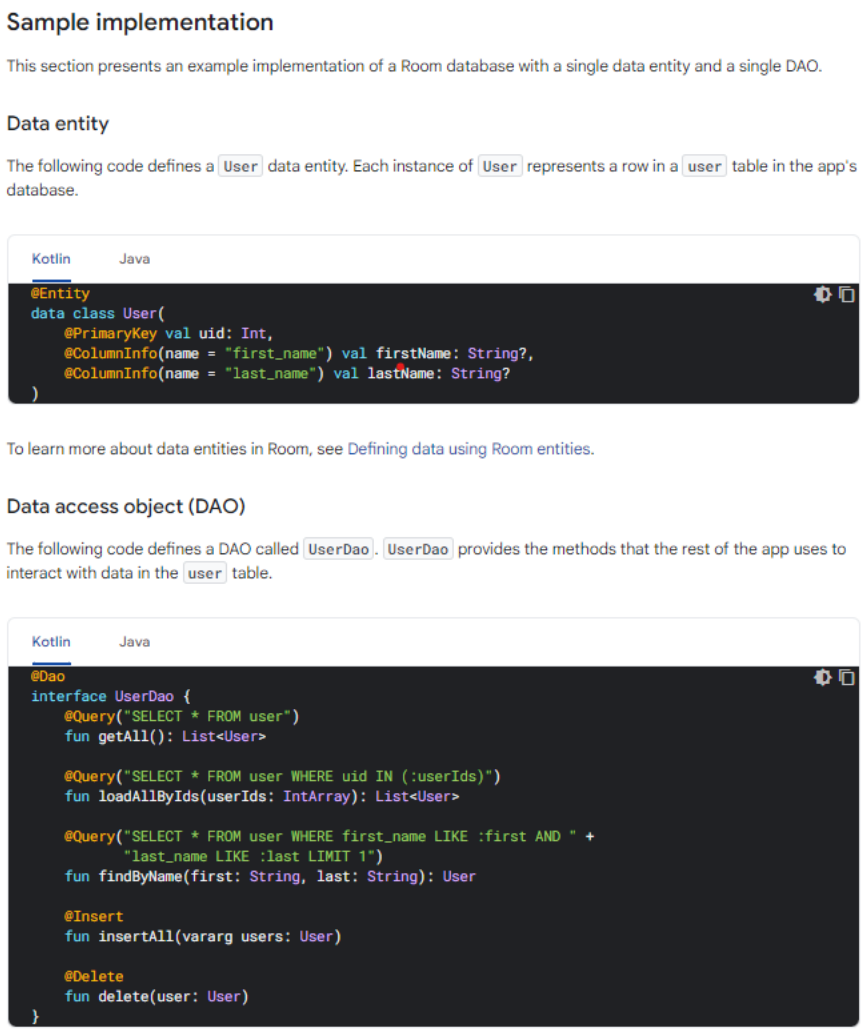
There are three major components in Room:

- 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. Figure 1 illustrates the relationship between the different components of Room.

Figure 1. *Diagram of Room library architecture.*

Figure 6: Structure of room database



Sample implementation

This section presents an example implementation of a Room database with a single data entity and a single DAO.

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.

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

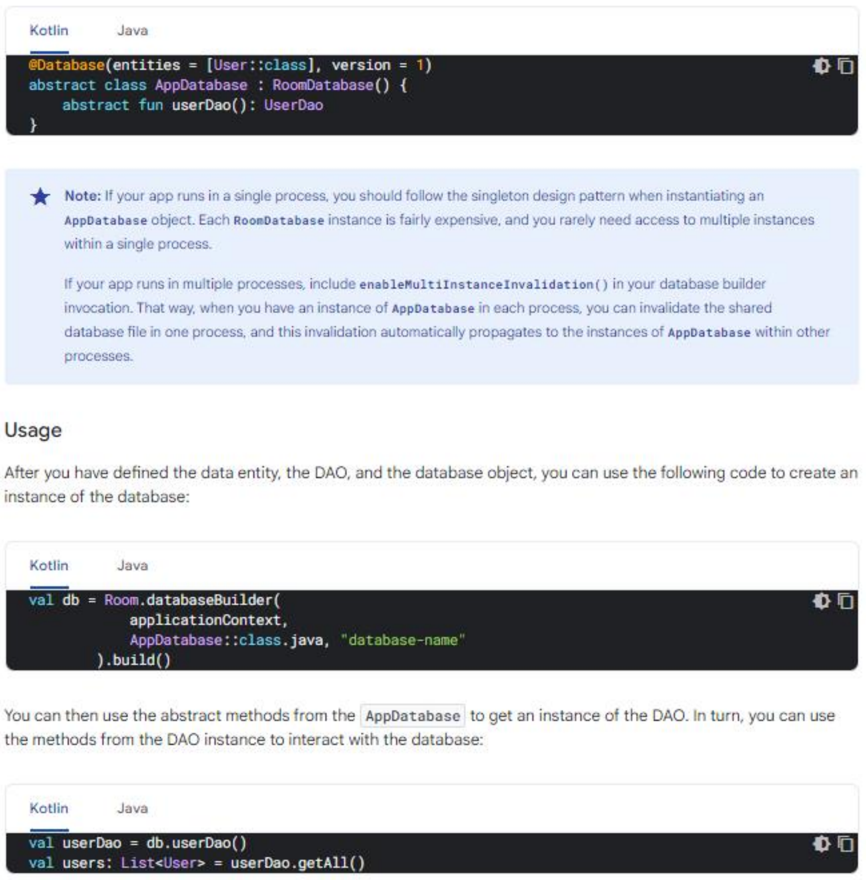
To learn more about data entities in Room, see Defining data using Room entities.

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.

```kotlin
@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)
}
```

Figure 7: Implementation of room


Kotlin Java

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

★ **Note:** If your app runs in a single process, you should follow the singleton design pattern when instantiating an `AppDatabase` object. Each `RoomDatabase` instance is fairly expensive, and you rarely need access to multiple instances within a single process.

If your app runs in multiple processes, include `enableMultiInstanceInvalidation()` in your database builder invocation. That way, when you have an instance of `AppDatabase` in each process, you can invalidate the shared database file in one process, and this invalidation automatically propagates to the instances of `AppDatabase` within other processes.

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:

Kotlin Java

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

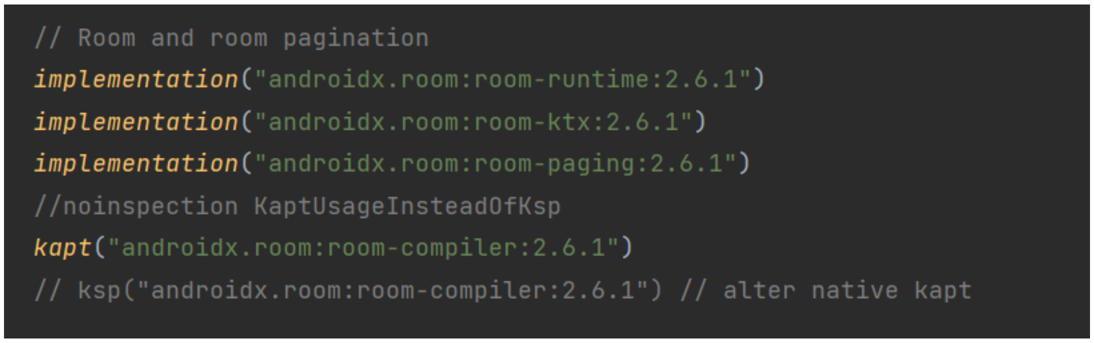
You can then use the abstract methods from the `AppDatabase` to get an instance of the DAO. In turn, you can use the methods from the DAO instance to interact with the database:

Kotlin Java

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

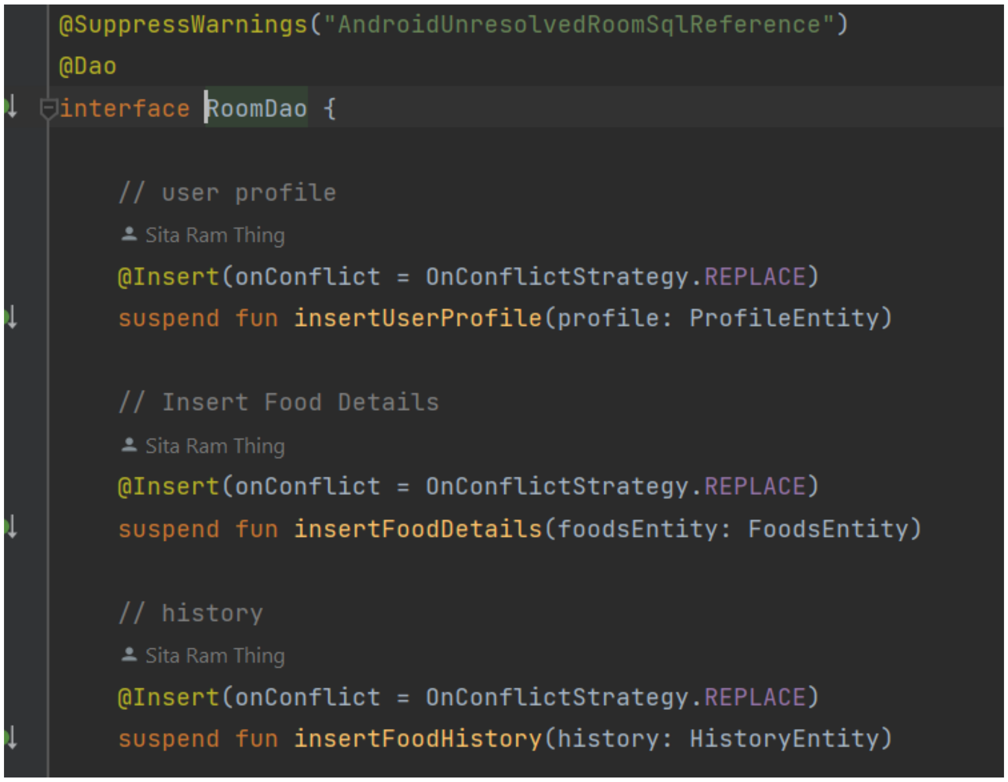
Figure 8: Use the functions

1.3 Implementation of the system



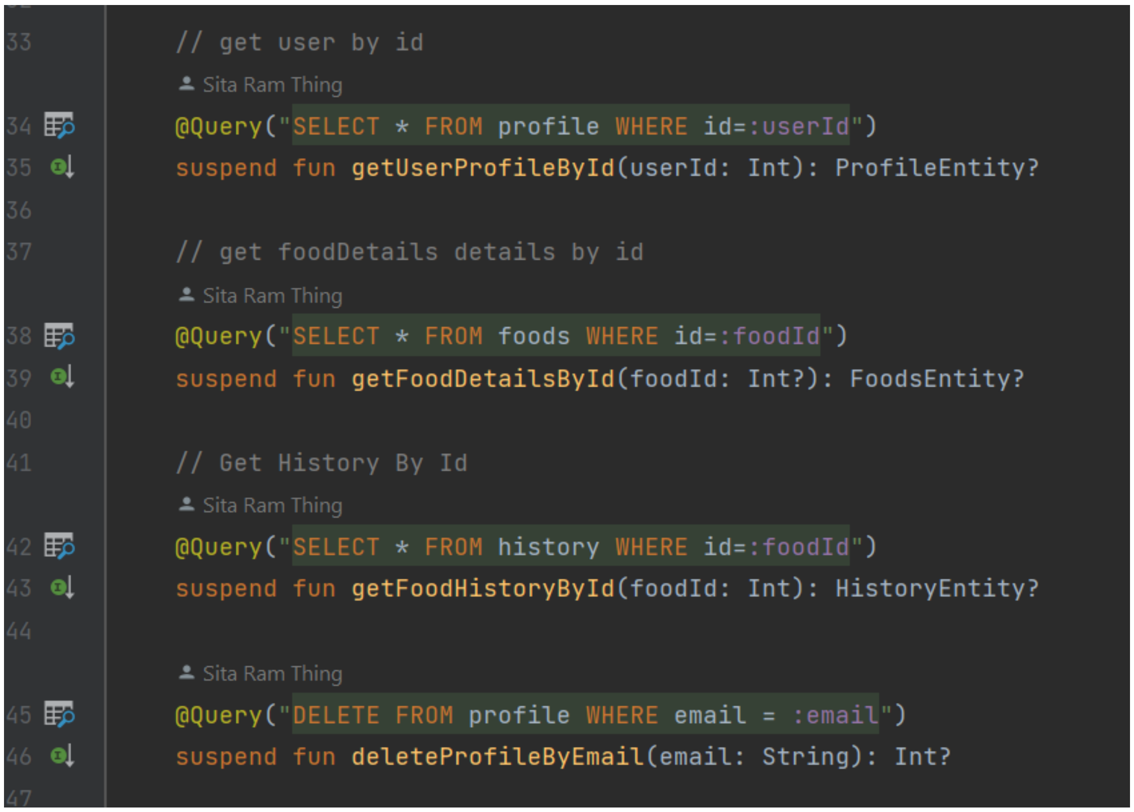
```
// Room and room pagination
implementation("androidx.room:room-runtime:2.6.1")
implementation("androidx.room:room-ktx:2.6.1")
implementation("androidx.room:room-paging:2.6.1")
//noinspection KaptUsageInsteadOfKsp
kapt("androidx.room:room-compiler:2.6.1")
// ksp("androidx.room:room-compiler:2.6.1") // alter native kapt
```

Figure 9: Implementation of the dependency.



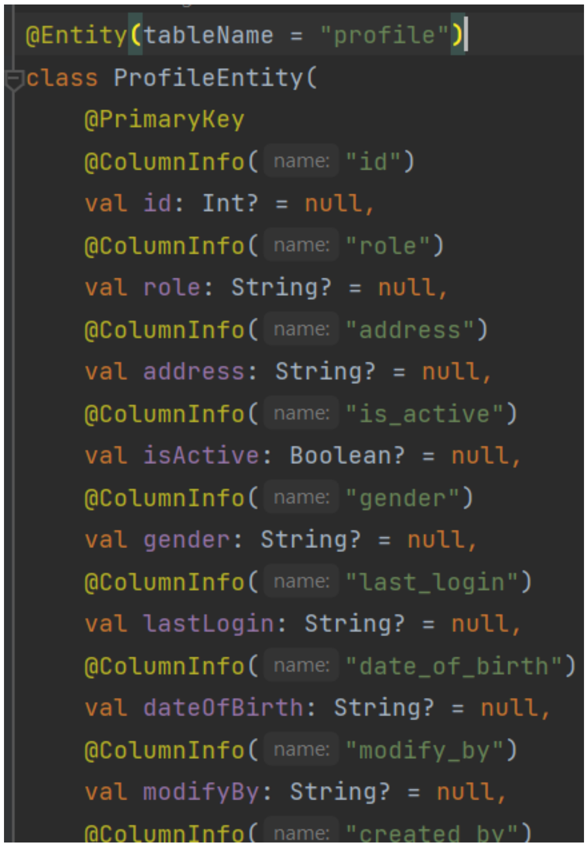
```
@SuppressWarnings("AndroidUnresolvedRoomSqlReference")
@Dao
interface RoomDao {

    // user profile
    Sita Ram Thing
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUserProfile(profile: ProfileEntity)

    // Insert Food Details
    Sita Ram Thing
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertFoodDetails(foodsEntity: FoodsEntity)

    // history
    Sita Ram Thing
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertFoodHistory(history: HistoryEntity)
```

Figure 10: insert data Room database



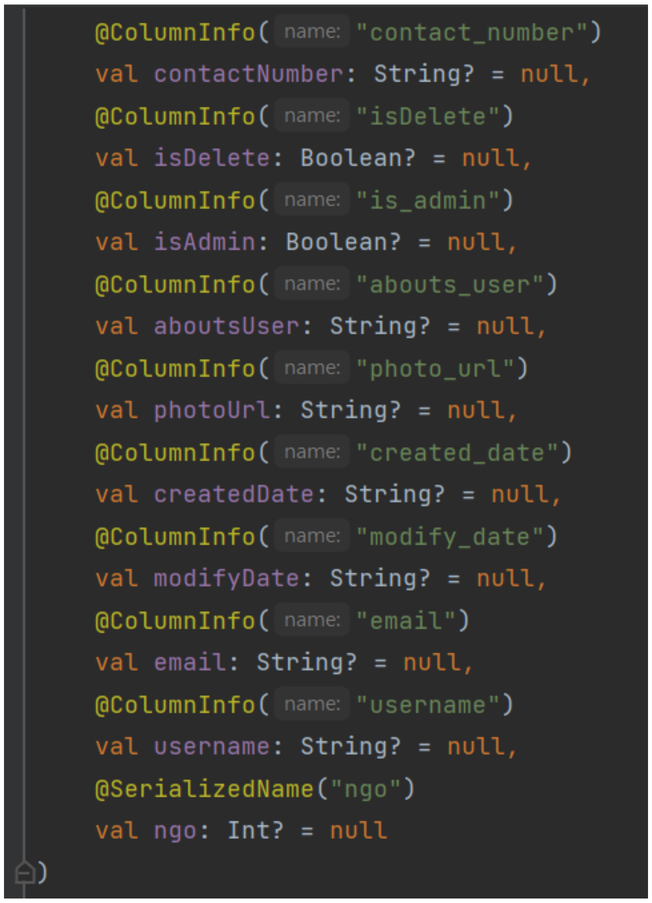
```
// get user by id
Sita Ram Thing
@Query("SELECT * FROM profile WHERE id=:userId")
suspend fun getUserProfileById(userId: Int): ProfileEntity?

// get foodDetails details by id
Sita Ram Thing
@Query("SELECT * FROM foods WHERE id=:foodId")
suspend fun getFoodDetailsById(foodId: Int?): FoodsEntity?

// Get History By Id
Sita Ram Thing
@Query("SELECT * FROM history WHERE id=:foodId")
suspend fun getFoodHistoryById(foodId: Int): HistoryEntity?

Sita Ram Thing
@Query("DELETE FROM profile WHERE email = :email")
suspend fun deleteProfileByEmail(email: String): Int?
```

Figure 11: Phase data from the database



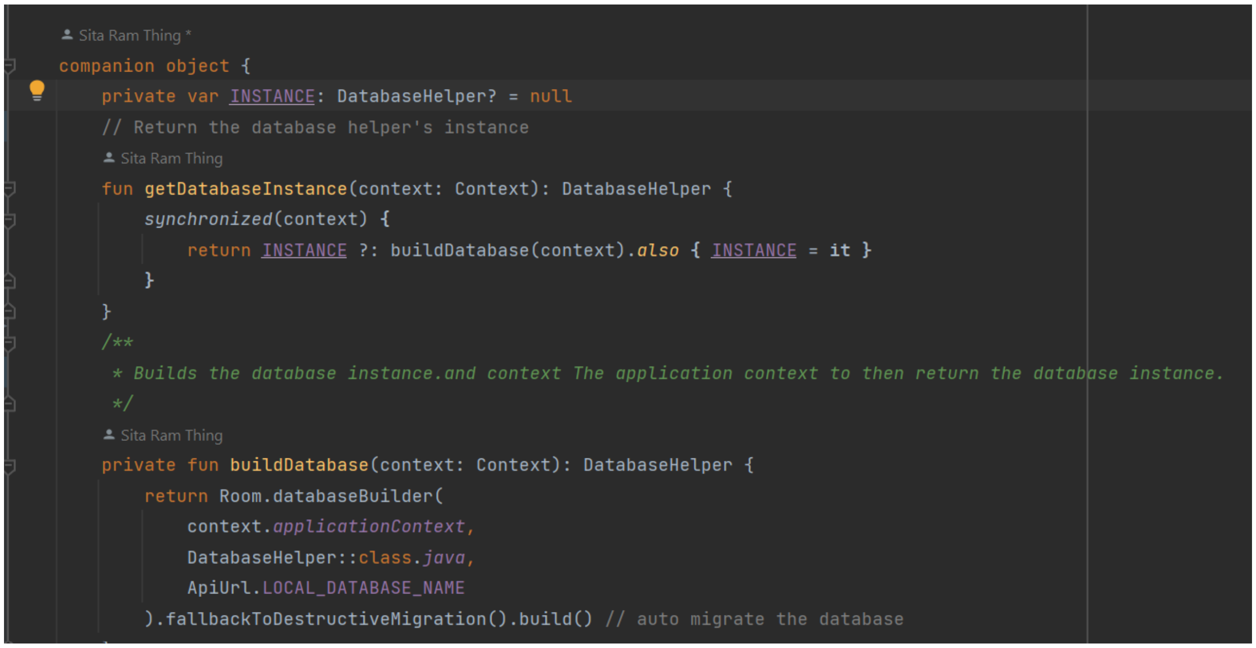
```
@Entity(tableName = "profile")
class ProfileEntity(
    @PrimaryKey
    @ColumnInfo( name: "id")
    val id: Int? = null,
    @ColumnInfo( name: "role")
    val role: String? = null,
    @ColumnInfo( name: "address")
    val address: String? = null,
    @ColumnInfo( name: "is_active")
    val isActive: Boolean? = null,
    @ColumnInfo( name: "gender")
    val gender: String? = null,
    @ColumnInfo( name: "last_login")
    val lastLogin: String? = null,
    @ColumnInfo( name: "date_of_birth")
    val dateOfBirth: String? = null,
    @ColumnInfo( name: "modify_by")
    val modifyBy: String? = null,
    @ColumnInfo( name: "created_by")
```

Figure 12: Initialize the entity for the local room database



```
    @ColumnInfo( name: "contact_number")
    val contactNumber: String? = null,
    @ColumnInfo( name: "isDelete")
    val isDelete: Boolean? = null,
    @ColumnInfo( name: "is_admin")
    val isAdmin: Boolean? = null,
    @ColumnInfo( name: "abouts_user")
    val aboutsUser: String? = null,
    @ColumnInfo( name: "photo_url")
    val photoUrl: String? = null,
    @ColumnInfo( name: "created_date")
    val createdDate: String? = null,
    @ColumnInfo( name: "modify_date")
    val modifyDate: String? = null,
    @ColumnInfo( name: "email")
    val email: String? = null,
    @ColumnInfo( name: "username")
    val username: String? = null,
    @SerializedName("ngo")
    val ngo: Int? = null
)
```

Figure 13: Remaining entity full show.



```
Sita Ram Thing *
companion object {
    private var INSTANCE: DatabaseHelper? = null
    // Return the database helper's instance
    Sita Ram Thing
    fun getDatabaseInstance(context: Context): DatabaseHelper {
        synchronized(context) {
            return INSTANCE ?: buildDatabase(context).also { INSTANCE = it }
        }
    }
    /**
     * Builds the database instance.and context The application context to then return the database instance.
     */
    Sita Ram Thing
    private fun buildDatabase(context: Context): DatabaseHelper {
        return Room.databaseBuilder(
            context.applicationContext,
            DatabaseHelper::class.java,
            ApiUrl.LOCAL_DATABASE_NAME
        ).fallbackToDestructiveMigration().build() // auto migrate the database
```

Figure 14: get the database instance