**ROOM DB**

Room Database is a part of the Android Architecture components which provides an abstraction layer over SQLite which allows for more robust database access while still providing the full power of SQLite.

**Why use Room Database?**

* Compile-time verification of SQL queries. each @Query and @Entity is checked at the compile time.
* Using Annotations to reduce the boilerplate code.
* Easily integrated with other Architecture components like LiveData, and RxJava.

**What is the difference between the room and SQLite database?**

* In the 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.

**There Are Basically 3 Major Components In Room.**

1. **@Entity:** 
   1. The “@Entity” class represents an entity in a table.
   2. [Data entities](https://developer.android.com/training/data-storage/room/defining-data) that represent tables in your app's database.

@Entity(tableName = "user")  
data class Users(  
 @PrimaryKey(autoGenerate = true)  
 var userId: Int? = null,  
 val userName: String,  
 var location: String,  
 val email: String  
)

1. **@Dao — Data Access Object:** 
   1. [Data access objects (DAOs)](https://developer.android.com/training/data-storage/room/accessing-data) that provide methods that your app can use to query, update, insert, and delete data in the database.
   2. It is An Interface class where we put all our SQL queries.
   3. We don’t require to write whole queries now; we need to make a method and annotate with specific annotations like
   * @**Insert**: Used to insert a record into the Room database.
   * **@Delete**: Used to delete record from Room database.
   * **@Update**: Used to update record in Room Database.
   * **@Query:**  Used to enter the Query like (SELECT FROM\*)

@Dao  
interface UserDao {  
  
 @Insert  
 fun insertUser(users: Users)  
  
  
 @Insert(onConflict = OnConflictStrategy.REPLACE)  
 fun insertUsers(vararg users: User)  
  
  
 @Insert  
 fun insertBothUsers(user1: User, user2: User)  
  
  
 @Insert  
 fun insertUsersAndFriends(user: User, friends: List<User>)  
  
  
 @Query("Select \* from user")  
 fun gelAllUsers(): List<Users>  
  
 @Query("SELECT \* FROM user WHERE userId IN (:userIds)")  
 fun loadAllByIds(userIds: IntArray): List<Users>  
  
  
 @Query("SELECT \* FROM user WHERE userName LIKE :namex AND location LIKE :locx LIMIT 1”)  
 fun findByName(namex: String, locx: String): Users  
  
  
  
 @Query("SELECT \* FROM user WHERE first\_name LIKE :search " +  
 "OR last\_name LIKE :search")  
 fun findUserWithName(search: String): List<User>  
  
  
 @Query(  
 "SELECT \* FROM book " +  
 "INNER JOIN loan ON loan.book\_id = book.id " +  
 "INNER JOIN user ON user.id = loan.user\_id " +  
 "WHERE user.name LIKE :userName"  
 )  
 fun findBooksBorrowedByNameSync(userName: String): List<Book>  
  
  
  
  
 @Query("SELECT first\_name, last\_name FROM user")  
 fun loadFullName(): List<NameTuple>  
  
  
 @Update  
 fun updateUser(users: Users)  
  
 @Delete  
 fun deleteUser(users: Users)  
  
}

1. **@Database:** 
   1. The [database class](https://developer.android.com/reference/kotlin/androidx/room/Database) that holds the database and serves as the main access point for the underlying connection to your app's persisted data.
   2. This is an abstract class that extends **RoomDatabase**, this is where you define the entities (tables)and the version number of your database. It contains the database holder and serves as the main access point for the underlying connection.

@Database(entities = [Users::class], version = 1, exportSchema = false)  
@TypeConverters(Converters::class)  
abstract class AppDatabase : RoomDatabase() {  
  
 abstract fun userDao() : UserDao  
  
 companion object {  
 private var INSTANCE: AppDatabase? = null  
  
 fun getInstance(context: Context): AppDatabase? {  
 if (INSTANCE == null) {  
 synchronized(AppDatabase::class) {  
 INSTANCE = Room.databaseBuilder(context.applicationContext,  
 AppDatabase::class.java, "user.db").allowMainThreadQueries()  
 .build()  
 }  
 }  
 return INSTANCE  
 }  
  
 fun destroyInstance() {  
 INSTANCE = null  
 }  
 }  
}

**Things to notice here:**

* This is an abstract class that has to extend from RoomDatabase.
* It has to be annotated with @Database, it receives a list of entities with all the classes that compose the database (all these classes have to be annotated with @Entity). We also have to provide a database version.
* We have to declare an abstract function for each of the entities included in the @Database annotation, this function has to return the correspondentDAO (A class annotated with @Dao).
* Finally, we declare a companion object to get static access to the method getAppDataBase which gives us a singleton instance of the database.

**Type Converters**

Type Converters are used when we declare a property that Room and SQL don’t know how to serialize. Let’s see an example of how to serialize the List<String> data type.

class Converters {  
  
 @TypeConverter  
 fun fromString(value: String): List<String> {  
 val listType = object : TypeToken<List<String>>() {  
  
 }.type  
 return Gson().fromJson(value, listType)  
 }  
  
 @TypeConverter  
 fun fromArrayList(list: List<String>): String {  
 val gson = Gson()  
 return gson.toJson(list)  
 }  
}

**UserRepository.kt(Optional)**

class UserRepository(context: Context) {  
  
 var db: UserDao = AppDatabase.getInstance(context)?.userDao()!!  
  
  
 //Fetch All the Users  
 fun getAllUsers(): List<Users> {  
 return db.gelAllUsers()  
 }  
  
 // Insert new user  
 fun insertUser(users: Users) {  
 db..insertUser(users)  
//insertAsyncTask(db).execute(users)  
 }  
  
 // update user  
 fun updateUser(users: Users) {  
 db.updateUser(users)  
 }  
  
 // Delete user  
 fun deleteUser(users: Users) {  
 db.deleteUser(users)  
 }  
}

**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()

**Some other Point:**

* **autoGenerate**: If you need Room to assign automatic IDs to entity instances, set the [autoGenerate](https://developer.android.com/reference/kotlin/androidx/room/PrimaryKey#autogenerate) property of **@PrimaryKey** to **true**.
* @**PrimaryKey:** Each Room entity must define a [primary key](https://en.wikipedia.org/wiki/Primary_key) that uniquely identifies each row in the corresponding database table.
* **@Ignore:** By default, Room creates a column for each field that's defined in the entity. If an entity has fields that you don't want to persist, you can annotate them using [@Ignore](https://developer.android.com/reference/androidx/room/Ignore)

**Create a view**

To create a view, add the [@DatabaseView](https://developer.android.com/reference/androidx/room/DatabaseView) annotation to a class. Set the annotation's value to the query that the class should represent.

@DatabaseView("SELECT user.id, user.name, user.departmentId," +  
 "department.name AS departmentName FROM user " +  
 "INNER JOIN department ON user.departmentId = department.id")  
data class UserDetail(  
 val id: Long,  
 val name: String?,  
 val departmentId: Long,  
 val departmentName: String?  
)

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

**Write asynchrones DAO query**

1. **Write asynchronous one-shot queries using coroutine**

@Dao  
interface UserDao {  
 @Insert(onConflict = OnConflictStrategy.REPLACE)  
 suspend fun insertUsers(vararg users: User)  
  
 @Update  
 suspend fun updateUsers(vararg users: User)  
  
 @Delete  
 suspend fun deleteUsers(vararg users: User)  
  
 @Query("SELECT \* FROM user WHERE id = :id")  
 suspend fun loadUserById(id: Int): User  
  
 @Query("SELECT \* from user WHERE region IN (:regions)")  
 suspend fun loadUsersByRegion(regions: List<String>): List<User>  
}

1. **Write observable query using coroutine Flow**

@Dao  
interfaceUserDao {  
 @Query("SELECT \* FROM user WHERE id = :id")  
 fun loadUserById(id: Int): Flow<User>  
  
 @Query("SELECT \* from user WHERE region IN (:regions)")  
 fun loadUsersByRegion(regions: List<String>): Flow<List<User>>  
}

**Pre-Populate your Room database**

Sometimes, you might want your app to start with a database that is already loaded with a specific set of data. This is called *prepopulating* a database. In Room 2.2.0 and higher, you can use API methods to prepopulate a Room database at initialization with contents from a prepackaged database file in the device's file system.

1. **Prepopulate from an app asset:** To prepopulate a Room database from a prepackaged database file that is located anywhere in your app's assets/ directory, call the [createFromAsset()](https://developer.android.com/reference/kotlin/androidx/room/RoomDatabase.Builder#createfromasset) method from your RoomDatabase.Builder object before calling [build()](https://developer.android.com/reference/kotlin/androidx/room/RoomDatabase.Builder#build):

Room.databaseBuilder(appContext, AppDatabase.class, "Sample.db")

    .createFromAsset("database/myapp.db")

    .build()

**Note**: The createFromAsset() method accepts a string argument that contains a relative path from the assets/ directory to the prepackaged database file.

1. **Pre-populate from the file system:** To prepopulate a Room database from a prepackaged database file that is located anywhere in the device's file system *except* your app's assets/ directory, call the [createFromFile()](https://developer.android.com/reference/kotlin/androidx/room/RoomDatabase.Builder#createfromfile) method from your RoomDatabase.Builder object before calling [build()](https://developer.android.com/reference/kotlin/androidx/room/RoomDatabase.Builder#build):

**Migrating Room databases**

1. **Automated Migrations:** To declare an automated migration between two database versions, add an [@AutoMigration](https://developer.android.com/reference/kotlin/androidx/room/AutoMigration) annotation to the [autoMigrations](https://developer.android.com/reference/kotlin/androidx/room/Database#automigrations) property in [@Database](https://developer.android.com/reference/kotlin/androidx/room/Database):

@Database(version = 2,  
 entities = [User::class, UserDetails::class],  
 autoMigrations = [AutoMigration (from = 1, to = 2, spec = AppDatabase.MyAutoMigration::class)]  
 )  
abstract class AppDatabase : RoomDatabase() {  
   
 @RenameTable(fromTableName = "User", toTableName = "AppUser")  
 class MyAutoMigration : AutoMigrationSpec  
   
}

2. Manual Migration: In cases where a migration involves complex schema changes, Room might not be able to generate an appropriate migration path automatically. For example, if you decide to split the data in a table into two tables, Room is unable to tell how this split should be performed. In cases like these, you must manually define a migration path by implementing a [Migration](https://developer.android.com/reference/kotlin/androidx/room/migration/Migration) class.

val MIGRATION\_1\_2 = object : Migration(1, 2) {  
 override fun migrate(database: SupportSQLiteDatabase) {  
 database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, `name` TEXT, " + "PRIMARY KEY(`id`))")  
 }  
}  
  
val MIGRATION\_2\_3 = object : Migration(2, 3) {  
 override fun migrate(database: SupportSQLiteDatabase) {  
 database.execSQL("ALTER TABLE Book ADD COLUMN pub\_year INTEGER")  
 }  
}  
  
Room.databaseBuilder(applicationContext, MyDb::class.java, "database-name").addMigrations(MIGRATION\_1\_2, MIGRATION\_2\_3).build()