WHAT IS SQLITE Database?

SQLite is **a lightweight , compact and open-source relational database that does not require any kind of server to run.**
There are many libraries and classes available on Android to perform any kind of database queries on SQLite.

What is Room ?
The Room persistence library provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.
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.
- As your schema changes, you need to update the affected SQL queries manually. Room solves this problem.
- Room is built to work with LiveData and RxJava for data observation, while SQLite does not.

Few kotlin annotation processing tool library and other room dependencies are need to be added in build.gradle

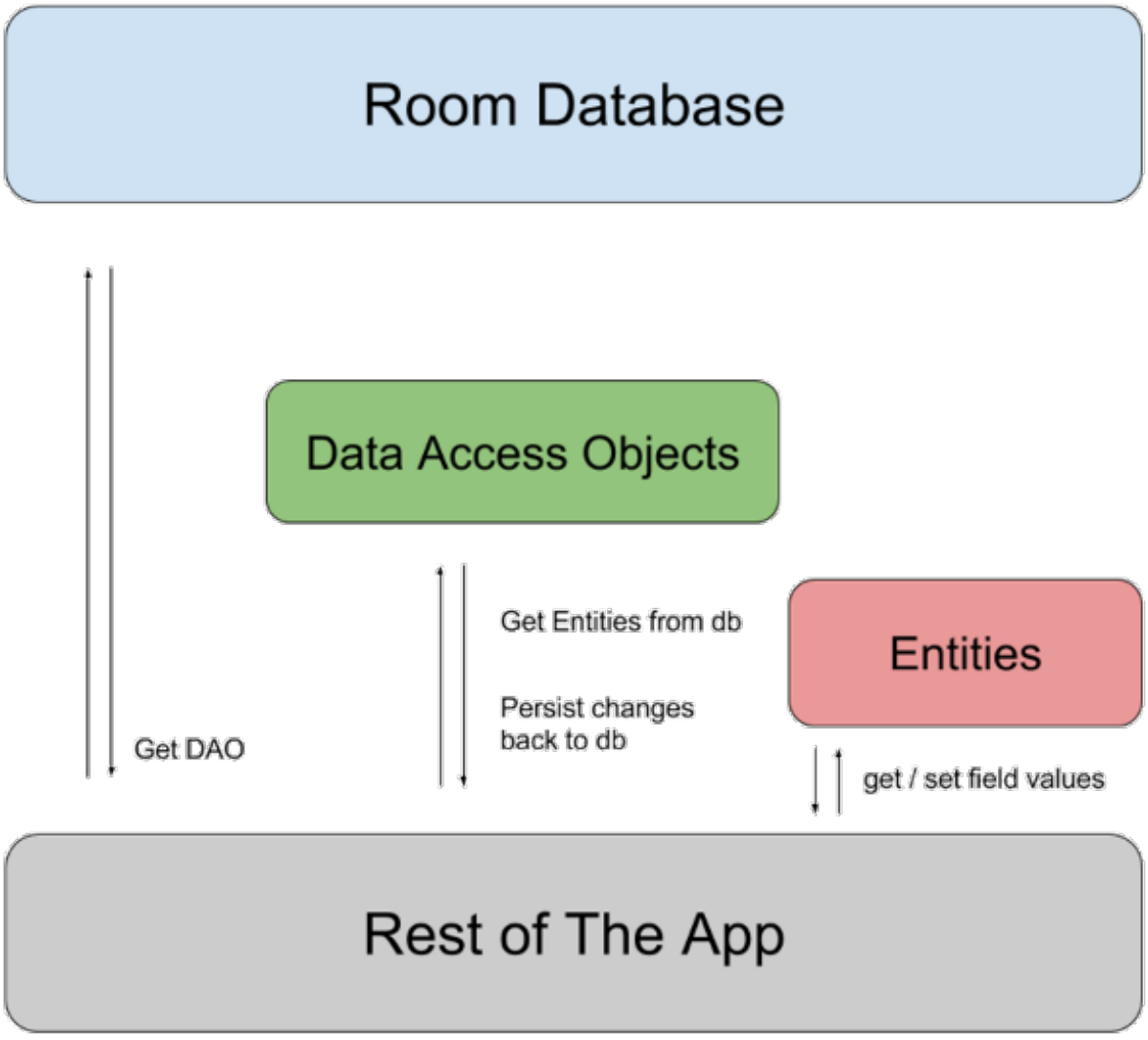
Primary components

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.



Example implementation of a Room database with a single data entity and a single DAO.

Data entity /Tables

```
@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
}
```

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.

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()
```

JAVA CODE:

```
data class Book(
    val isbn: String,
    val title: String,
    val author: String,
    val publisher: String,
) : Parcelable {
    override fun describeContents(): Int {
        TODO("Pending implemented")
    }

    override fun writeToParcel(dest: Parcel?, flags: Int) {
        TODO("Pending implemented")
    }
}
```

//KOTLIN
This Parcelizeannotation tells the Kotlin compiler to generate the writeToParcel(), and describeContents() methods

```
@Parcelize
data class Book(
    val isbn: String,
    val title: String,
```

```
    val author: String,
    val publisher: String,
) : Parcelable
```