**­­CMPS 312 Mobile App Development**

**Lab 10 – Data Layer**

**Objective**

In this Lab, you will **build a Todo app that persists data in a local SQLite database.** You will use Room library and coroutines and practice the following skills:

* Create Entity classes.
* Create Data Access Objects (DAO) to map DAO methods to SQL queries.
* Perform database CRUD operations.
* Create and interact with a SQLite database using Room library.
* Handle database relations such as one to many relationships.
* Create cascade delete and enforce integrity checks using foreign keys.
* Use Database Inspector to interact with the SQLite database.

Figure 1 illustrates the Data Layer you will implement in this Lab as part of MVVM architecture.

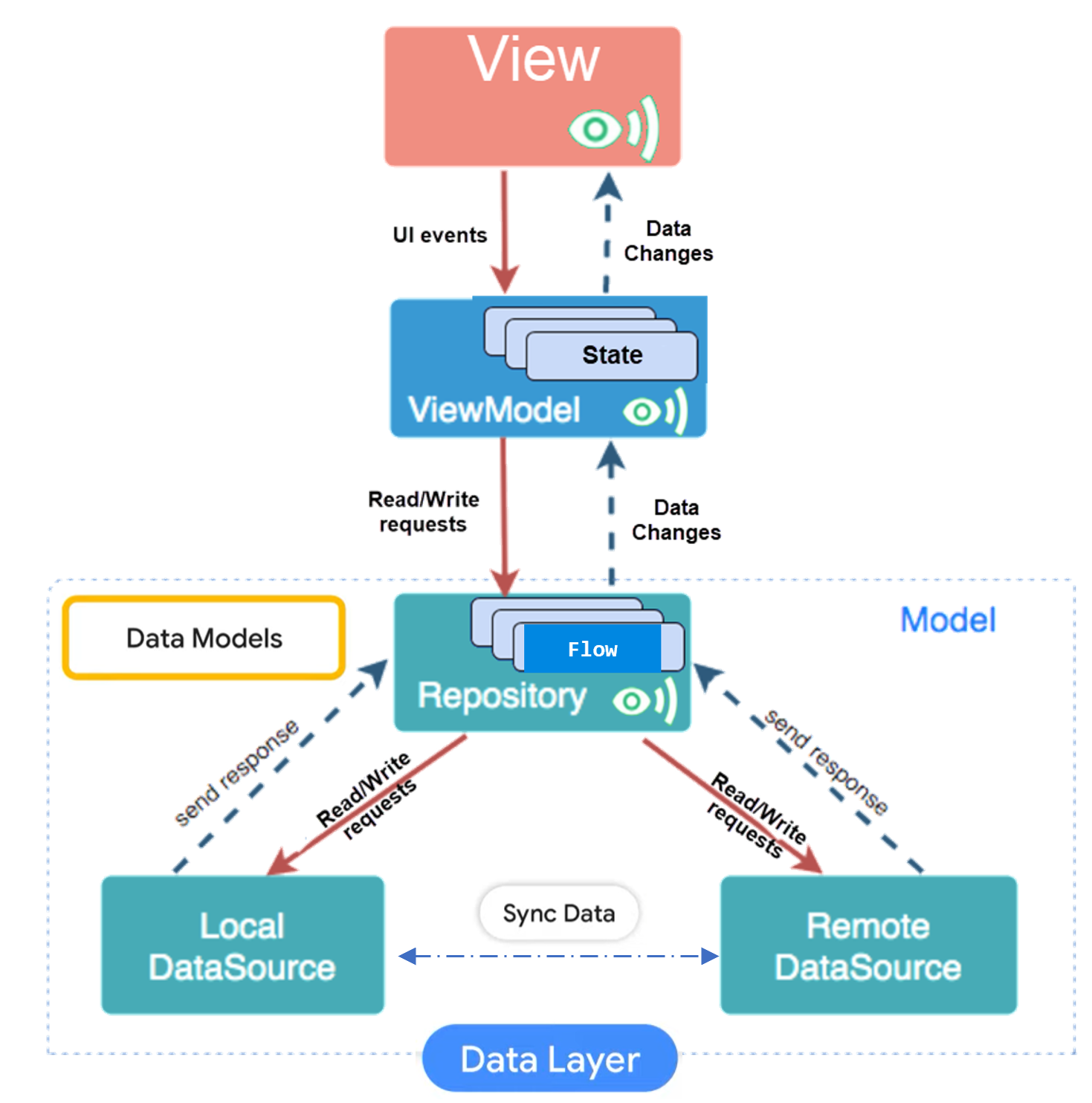


Figure 1. MVVM architecture

Figure 2 illustrates the relationship between ToDo repository, Project and ToDo models, ProjectDoa and TodoDao data sources, and ToDo Database

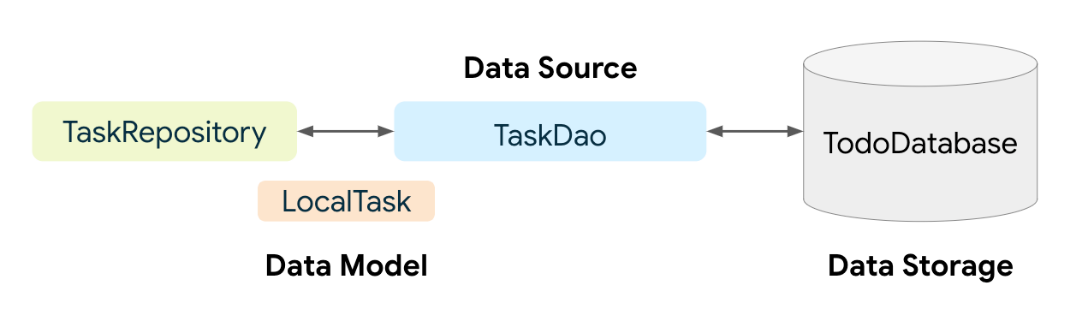


Figure 2. Relationship between ToDo repository, Project and ToDo models, ProjectDoa and TodoDao data sources, and ToDo Database

**Preparation**

1. Sync the Lab GitHub repo and copy the **Lab 10-****Data Layer** folder into your repository.
2. Add the following inside plugins of app’s build.gradle

plugins **{**  
 id ("com.google.devtools.ksp") *version* "1.9.10-1.0.13"  
**}**

Then add the following room dependencies

val roomVersion = "2.6.0"  
*implementation*("androidx.room:room-runtime:$roomVersion")  
*annotationProcessor*("androidx.room:room-compiler:$roomVersion")  
*// Kotlin Symbol Processing (KSP) - for processing annotations  
ksp*("androidx.room:room-compiler:$roomVersion")  
*// Kotlin Extensions and Coroutines support for Room  
implementation*("androidx.room:room-ktx:$roomVersion")  
  
*// Kotlin Datetime  
implementation*("org.jetbrains.kotlinx:kotlinx-datetime:0.4.1")  
  
*// Required to be able to use collectAsStateWithLifecycle  
implementation*("androidx.lifecycle:lifecycle-runtime-compose:2.6.2")

**PART A: Implementing the Todo App**

**­­­**

**A picture containing background pattern

Description automatically generated**Implement a Todo app to allows users to track todo tasks per projects. The user can add a project and subtasks under each project. The user can also update and delete both projects and todos. If the user deletes a project, then all associated todos should also be deleted.

Graphical user interface

Description automatically generated**A screen shot of a computer

Description automatically generatedGraphical user interface, application

Description automatically generatedGraphical user interface, text, application, chat or text message

Description automatically generatedGraphical user interface, application

Description automatically generated**

**A screenshot of a computer screen

Description automatically generatedGraphical user interface, application, Teams

Description automatically generated**

Figure 3: ToDo app UI design

1. **Create the entities**

Diagram

Description automatically generated

Figure 4. Todo App Entity Relations (ER) diagram

1. Open the entity package entity and create two data classes as shown in the Entity Relations (ER) diagram presented in Figure 4.
2. Annotate the Project class with **@**Entityannotation.

Annotate the id parameter of the Project as a primary key **@PrimaryKey(autoGenerate = true)**

1. Annotate the Todo entity with the following annotation to create a **one to many** relationship with the Project entity.

@Entity(  
 foreignKeys = [  
 ForeignKey(  
 entity = Project::class,  
 parentColumns = ["id"],  
 childColumns = ["pid"],  
 onDelete = ForeignKey.*CASCADE*,  
 onUpdate = ForeignKey.*CASCADE* )  
 ]  
)

* Annotate the id with **@PrimaryKey(autoGenerate = true)**
* Annotate the pid (project id) with @ColumnInfo(index = true)

1. **Create the data sources as DAO Interface**



1. Create **ProjectDao** interface under the **datasource** package and annotate with @Dao. Then add the following methods. Annotate the methods with the appropriate @Query, @Delete , @Insert , @Upsert annotations. E.g.,

suspend fun addProject(project: Project)

suspend fun deleteProject(project: Project)

1. Create **TodoDao** interface under the **datasource** package and annotate with @Dao. Then add the following methods. Annotate the methods with the appropriate @Query, @Delete , @Insert , @Upsert annotations.

fun observeTodos(pid : Int): Flow<List<Todo>>  
suspend fun getTodo(id: Int): Todo  
suspend fun upsertTodo(todo: Todo) : Long  
suspend fun deleteTodo(todo: Todo): Int

1. **Create the Room Database class**

Create a Room database abstract class that extends RoomDatabase and annotated with @Database.

It should provide a singleton dbInstance object created using Room.databaseBuilder() to create (if does not exit) and connect to the database. Also, it serves as the main access point to get the DAOs to interact with the database.

1. Create a public abstract class named TodoDB that extends RoomDatabase. The class is abstract because Room will generate the implementation.

Annotate the class with **@Database** and pass as arguments: list the app entities and the version number.

@Database(entities = [Todo::class, Project::class], version = 1)

1. Inside the class define 2 abstract methods to return the return the DAOs created earlier. Room will generate the implementation body.

abstract fun projectDao(): ProjectDao  
abstract fun todoDao(): TodoDao

1. Create a companion object that returns an instance of TodoDB. Only one instance (a singleton) of the database object is needed for the whole app.

Use Room.databaseBuilder() to create the database only if it doesn't already exist. Otherwise, return the existing instance.

The complete code for the companion object is shown below.

companion object {  
private var db: TodoDB? = null  
  
 fun getDatabase(context: Context): TodoDB {  
 if (db== null) {  
 db= Room.databaseBuilder(  
 context.*appContext*,  
 **TodoDB**::class.*java*,  
 "todoDB"  
 ).fallbackToDestructiveMigration().build()  
 }  
 return db as TodoDB  
 }  
}

1. **Create the Repository**
2. Implement TodoRepository class that call the methods on ProjectDao and TodoDao to read/write data from the database.

Create an instance of the todoDao by instantiating the database object and getting the dao instance

private val projectDao by *lazy* **{** TodoDB.getDatabase(context).projectDao()  
**}**

private val todoDao by *lazy* **{** TodoDB.getDatabase(context).todoDao()  
**}**

1. Implement the repository functions by calling the corresponding ProjectDao and TodoDao functions. repository
2. Run and test your implementation.
3. **Query One to Many relationship**

An important feature of a relational database is the ability to query data from multiple tables. Using [@Relation](https://developer.android.com/reference/android/arch/persistence/room/Relation) annotation you can easily get a project and its associated todos in one query.

A picture containing graphical user interface

Description automatically generated

1. Create a new **ProjectWithTodos** data class
2. Under the class create two properties. A project property of type Project and another list property of type Todo named todos.
3. Annotate the project property with @Embedded val project: Project
4. Annotate the todos property with @Relation(parentColumn = "id", entityColumn = "pid")
5. Add getProjectWithTodos to TodoDAO class to get all the projects with their to-dos. Make sure you add @Transaction as this method will run multiple SQL statements. @Transaction will ensure that all of them are executed as one unit of work.

@Transaction  
@Query ("SELECT \* FROM Project")  
suspend fun getProjectWithTodos(): List<ProjectWithTodos>

1. **Test the database queries using Database Inspector**

Graphical user interface, text, application

Description automatically generatedTest the app queries using Android Studio *Database Inspector*. This helps you write and test your queries before using them in the DAOs. Try to run all the queries used the DAOs interface. Try other queries that we did not implement.

Graphical user interface, application

Description automatically generated

**PART B: Implement the Banking Apps Database**

**­­­**

Extend the Banking App introduced in the previous week's implementation, which interacted with the Bank Web API. Enhance the app by incorporating data caching in the database. Consequently, only the QuBankService class in the repository should handle data updates in the database, avoiding direct communication with the view model. The ViewModel should retrieve data directly from the cached database.