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 Floor library with Future and Stream, 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 Floor library.
- Handle database relations such as one to many relationships.
- Use Database views to encapsulate complex queries and present them as virtual tables.
- Use Database Inspector to interact with the SQLite database.

Figure 1 illustrates the Data Layer you will implement in this Lab as part of Current MVVM Architecture we are following.

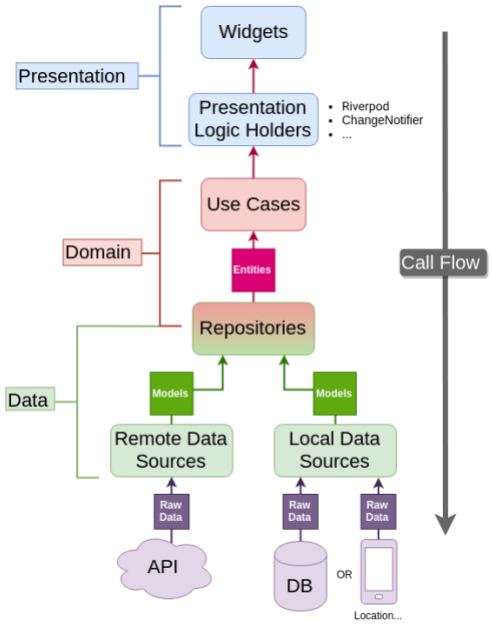


Figure 1. MVVM architecture

The diagram below illustrates the ToDo application's data architecture. The ToDoRepository serves as a central access point, interacting with ProjectDao and ToDoDao to manage data from the ToDoDatabase, which provides persistent storage for projects and tasks.

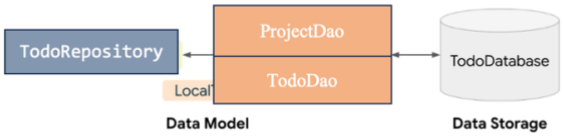


Figure 2. Relationship between ToDo repository, Project and ToDo models, ProjectDoa and TodoDao data sources, and ToDo Database

Preparation

1. Sync the Lab GitHub repo and copy the **Lab 10-Data Layer** folder into your repository.
2. Then add the following dependencies

```
dependencies:
  floor: ^1.5.0
  sqflite: ^2.4.0

dev_dependencies:
  floor_generator: ^1.4.2
  build_runner: ^2.1.2
```

PART A: Implementing the Todo App

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.

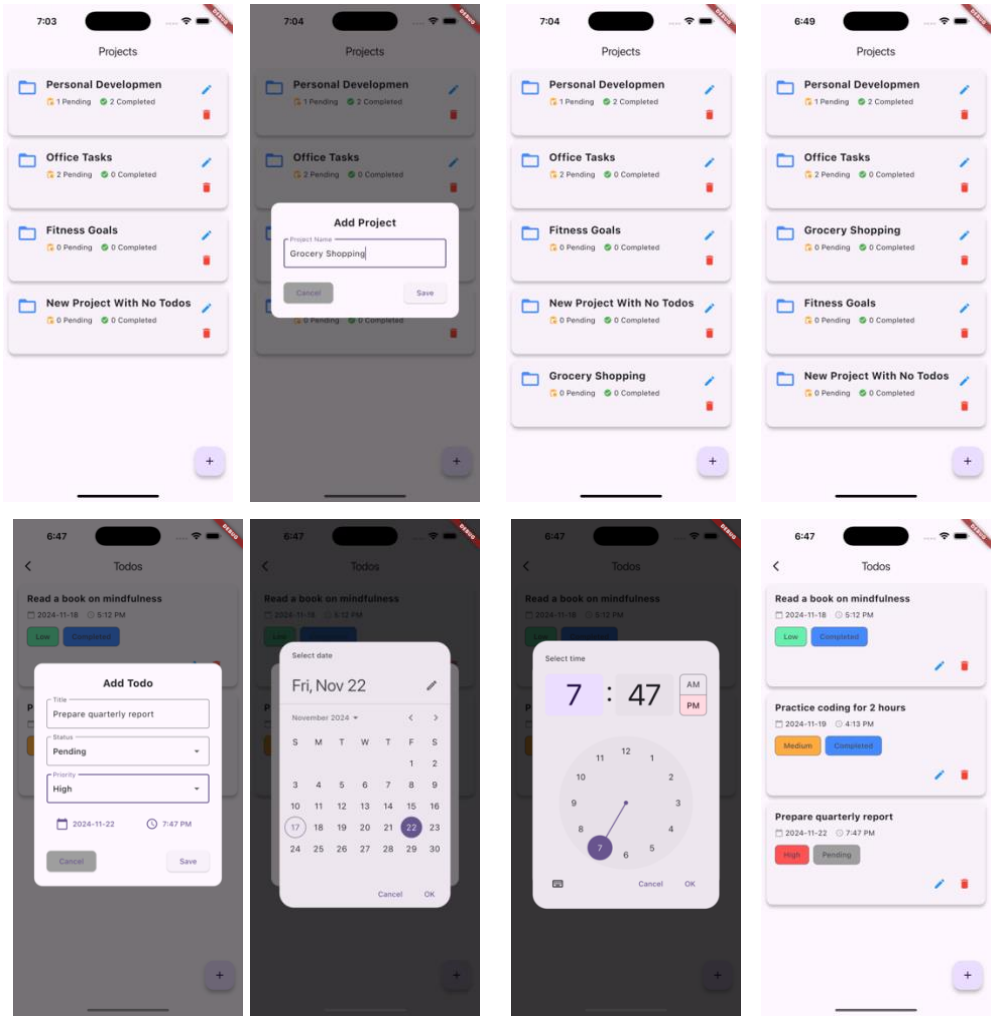


Figure 3: ToDo app UI design

Create the entities

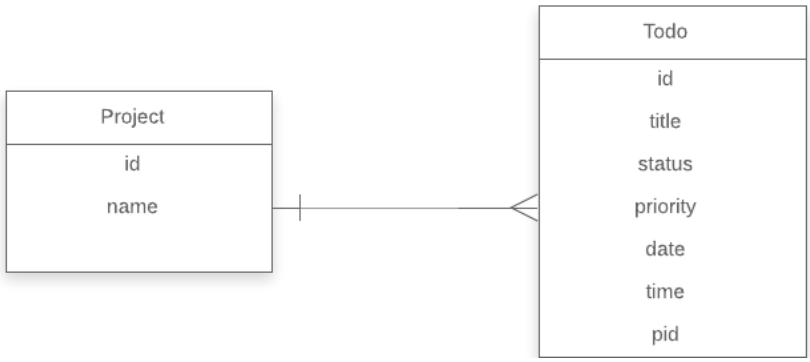


Figure 4. Todo App Entity Relations (ER) diagram

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 @Entity(tableName:'projects') annotation.
   Annotate the id parameter of the Project as a primary key **@PrimaryKey(autoGenerate = true)**
3. Annotate the Todo entity with the following annotation to create a **one to many** relationship with the Project entity.

```
@Entity(
  tableName: 'todos',
  foreignKeys: [
    ForeignKey(
      childColumns: ['pid'],
      parentColumns: ['id'],
      entity: Project,
    )
  ],
)
```

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

I. Create the data sources as DAO Interface

1. Create **ProjectDao** abstract class 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.,

```
@Query("SELECT * from Project")
observeProjects(): Stream<List<Project>>
addProject(project: Project) : Future<void>
deleteProject(project: Project) : Future<void>
```

2. 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 , @update annotations.

```
observeTodos(pid : Int): Stream<List<Todo>>
getTodo(id: Int): Stream<Todo?>
updateTodo(todo: Todo) : Future<void>
deleteTodo(todo: Todo): Future<void>
```

II. Create the Floor Database class

Create a Floor database abstract class that extends FloorDatabase and annotated with `@Database`. It serves as the main access point to get the DAOs to interact with the database.

1. Create a public abstract class named `AppDatabase` that extends `FloorDatabase`. 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.

```
//add this on the top of the database class
part 'app_database.g.dart';

@Database(
  version: 2,
  entities: [Todo, Project],
)
abstract class AppDatabase extends FloorDatabase {
  TodoDao get todoDao;
  ProjectDao get projectDao;
}
```

III. Generate the Database Code

Run the command below to generate the necessary database code using the Floor generator.

```
flutter packages pub run build_runner build --delete-
conflicting-outputs
```

IV. Create the Repository

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

```
class TodoListRepo {
  final ProjectDao projectDao;
  final TodoDao todoDao;

  TodoListRepo({required this.projectDao, required this.todoDao});
```

2. Implement the repository functions by calling the corresponding `projectDao` and `todoDao` functions. For the repository should allow the following:

- **Add** a new project
- **Update** an existing project
- **Get** all projects
- **Delete** a project **and** all associated **Todos**
- **Add** a new todo
- **Update** an existing todo
- **Get** all todos for a project id
- **Delete** a specific todo.

3. Create a TodoList Repository Provider. This provider should instantiate the database and then return an instance of the `TodoListRepo`.

```
final todoListRepoProvider = FutureProvider<TodoListRepo>((ref) async {
  final db =
      await $FloorAppDatabase.databaseBuilder('todo_database.db').build();
  return TodoListRepo(projectDao: db.projectDao, todoDao: db.todoDao);
});
```

4. Update your UI, to use the stream data, by using **StreamBuilder**
5. Run and test your implementation.

V. Creating Complex Quires using Database Views

An important feature of a relational database is the ability to query data from multiple tables. Using @Databaseview annotation you can aggregate the count of pending and completed todos for each project as shown in figure 5.

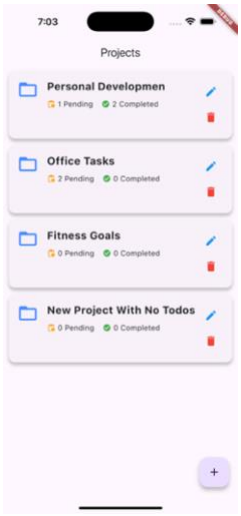


Figure 5 : Projects with Status Count

1. Create a new class called **ProjectTodoStatusCounts**
2. Under the class create three properties. A project property of type **int id**, **int pendingCount** and **int completedCount**
3. Annotate the class with
4. Annotate the id with @primaryKey
5. Make sure you add **ProjectTodoStatusCounts** to the @Database annotation as a view.

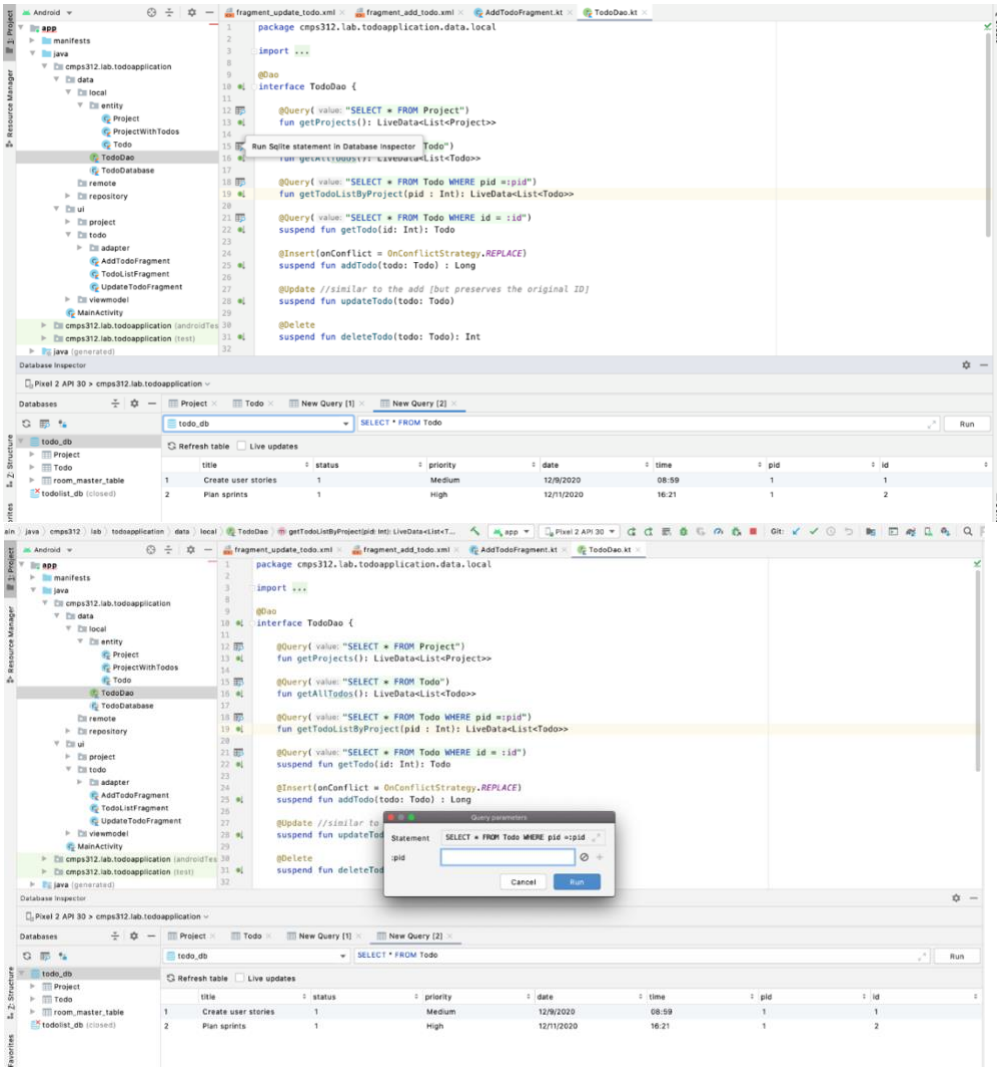
```
@Database(
  version: 2,
  entities: [Todo, Project],
  views: [ProjectTodoStatusCounts],
)
abstract class AppDatabase extends FloorDatabase
```

6. Add the following method to the ProjectDao class

```
@Query('SELECT * FROM ProjectTodoStatusCounts WHERE id = :pid')
Stream<ProjectTodoStatusCounts?> getProjectTodosStatusCounts(int pid);
```

VI. Test the database queries using Database Inspector

Test 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



PART B: Implement the Banking Apps Database

Using the techniques covered in Part A, your task is to enhance the Banking App from previous week's implementation by integrating a local database. Previously, the app interacted directly with the Bank Web API for all its data needs. The goal is to modify the app so that it leverages a local

database for improved performance and offline capability while maintaining the same functionality as before.

Your tasks are as follows:

1. **Cache Data Locally:** Retrieve data from the Web API as before, but store it in the local database. This ensures the data remains available even if the app goes offline or the server is temporarily unreachable.
2. **Update Repositories and Providers:**
    a. Modify the application's repositories and providers so they interact with the local database to fetch and save data.
    b. The Web API should only be called for synchronization or updates, ensuring minimal server dependency.
3. **Preserve App Behaviour:** Ensure the app functions identically to its previous implementation in terms of UI and user experience.

By the end of this task, the app should seamlessly utilize the local database, providing consistent performance while maintaining synchronization with the Web API.

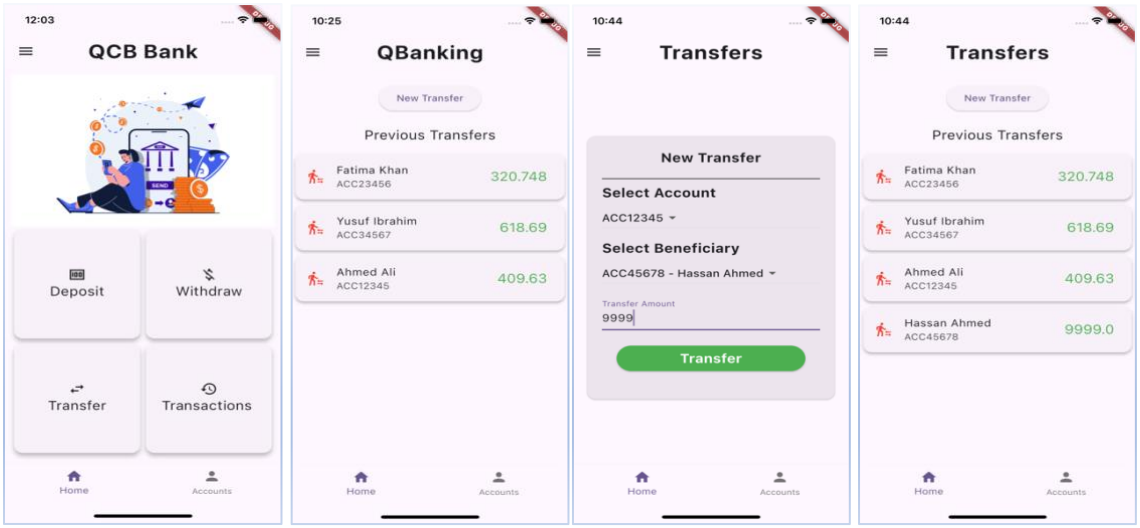


Figure 6 Banking App