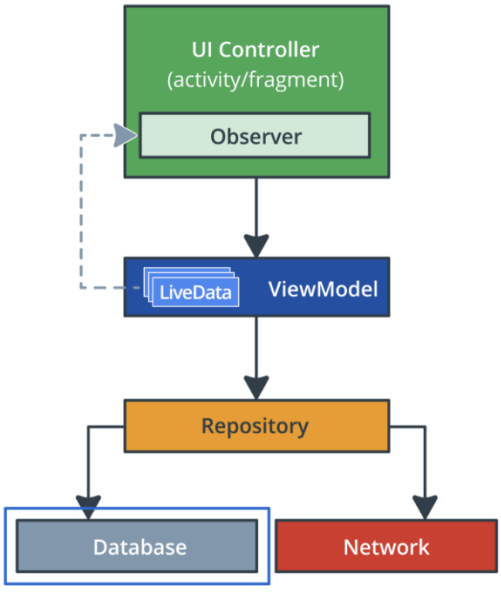
CMPS 312 Mobile App Development
Lab 10 – Data Management

Objective

In this Lab, you will **build a Todo app that persists data offline.** You will use Room library and coroutines to get, add, update, and delete to-dos in SQLite Database.
In this Lab you will practice:

- Create and interact with a SQLite database using Room library.
- Create Entity classes.
- Create Data Access Objects (DAO) to map Kotlin functions to SQL queries.
- Perform database CRUD operations
- 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

The image below shows how the Room database fits in with the overall architecture recommended in our Lab.



Preparation

1. Sync the Lab GitHub repo and copy the **Lab 10-Data Management** folder into your repository.
2. Add the following room dependencies

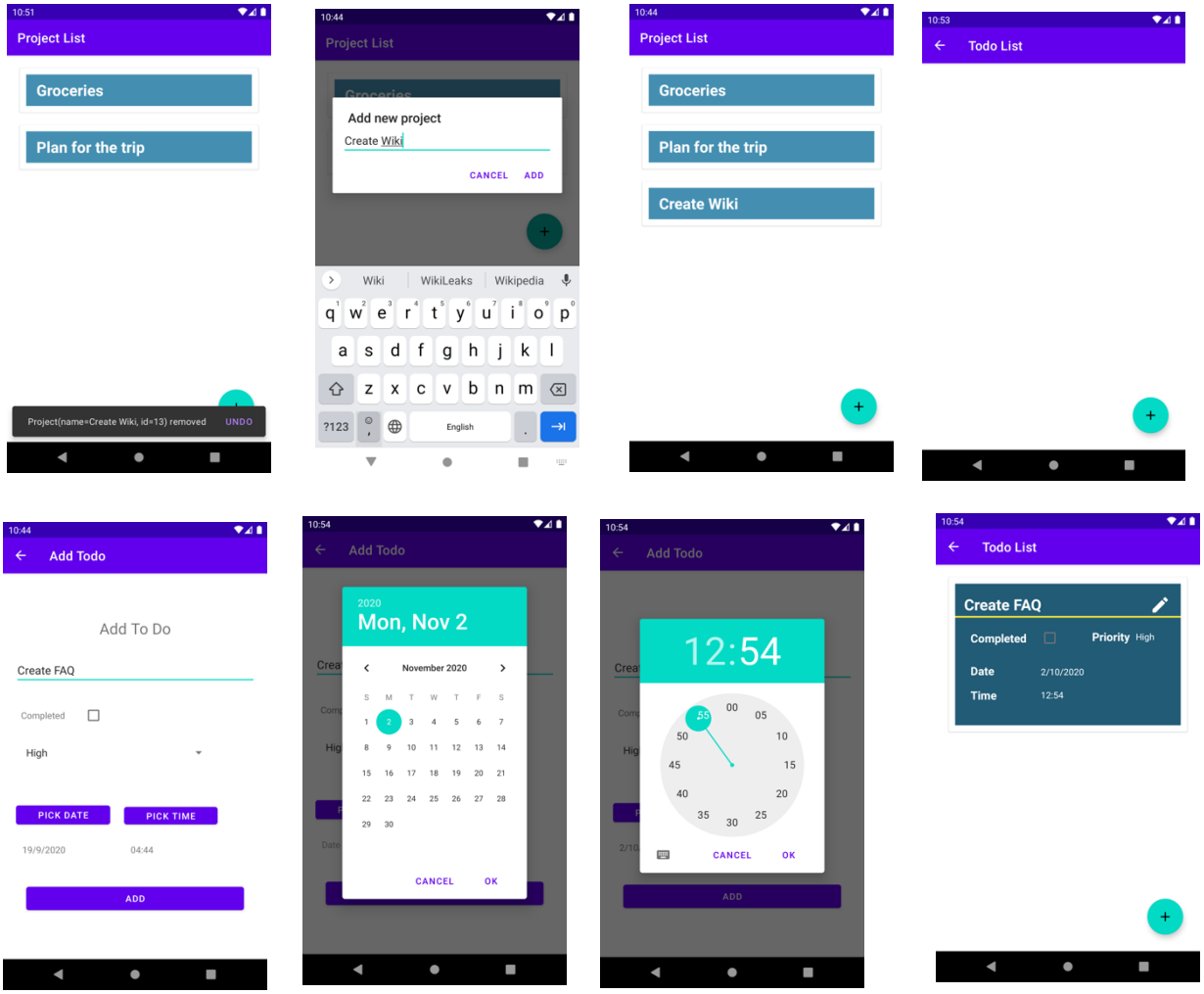
```
//     Room components
    def room_version = "2.2.5"

    implementation "androidx.room:room-runtime:$room_version"
    kapt "androidx.room:room-compiler:$room_version"

    // optional - Kotlin Extensions and Coroutines support for Room
    implementation "androidx.room:room-ktx:$room_version
```

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 also can update and delete both projects and todos. If the user deletes a project, then all associated todos should also be deleted.



Task 1: Creating the entities

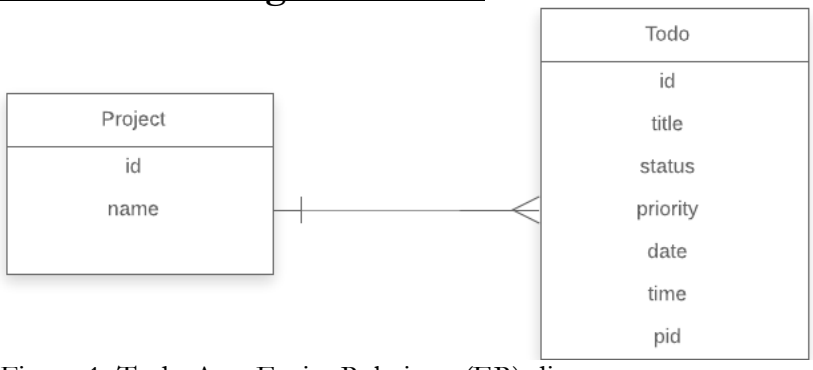


Figure 1. Todo App Entity Relations (ER) diagram

1. Open the entity package inside the data/entity and create two data classes. The classes should have the properties shown in the Entity Relations (ER) diagram (see Figure 1).
2. Annotate the Project class with @Entity annotation.
   Annotate the id parameter of the Project as a primary key **@PrimaryKey(autoGenerate = true)**
3. Annotate the Todo class with the following annotation to create a one to many relationship with the Project class.

```
@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)

Task 2: Creating the DAO Interface

For the todo app 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.

1. Create TodoDao interface under the data package and annotate with @Dao
   TodoDao interface should have the following methods to deliver the use cases above:

```
fun getProjects(): LiveData<List<Project>>
fun getTodos(pid : Int): LiveData<List<Todo>>

suspend fun addProject(project: Project)
suspend fun deleteProject(project: Project)
suspend fun getTodo(id: Int): Todo
suspend fun addTodo(todo: Todo) : Long
suspend fun updateTodo(todo: Todo)
suspend fun deleteTodo(todo: Todo): Int
```

2. Use the appropriate @Query, @Update, @Delete and @Insert to annotate methods of the TodoDao interface.

e.g. To get all projects you should annotate the getProjects function as follows.

```
@Query("select * from Project")
fun getProjects(): LiveData<List<Project>>
```

Task 3: Creating the Room Database

In this task, you create a Room database that uses the Entity and DAO that you created in the previous task. This class creates (if does not exit) and connects to the database. It serves as the main access point to get DAOs to interact with DB .

1. Create a public abstract class named `TodoDatabse` that extends `RoomDatabase`. This class serves as the main access point to interact with our todo list database. The class is abstract because Room will generate the implementation.
2. 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,
   exportSchema = false)
   ```
3. Inside the class define an abstract method that returns a **TodoDao**. Room will generate the implementation body.

   ```
   abstract fun todoDao(): TodoDao
   ```
4. Create a companion object that will return the instance of the todo list database. You only need one instance of the Room database for the whole app, so make the RoomDatabase a singleton.
5. Use Room's database builder to create the database only if the database doesn't exist. Otherwise, return the existing database.

The complete code for the companion object is shown below.

```
companion object {
    @Volatile // Meaning that writes to this field are immediately made visible to other threads
    private var database: TodoDatabase? = null

    /* Protected from concurrent execution by multiple threads */
    @Synchronized
    fun getDatabase(context: Context): TodoDatabase {
        if (database == null) {
            database = Room.databaseBuilder(
                context.appContext,
                TodoDatabase::class.java,
                "todo_db"
            ).fallbackToDestructiveMigration().build()
        }
        return database as TodoDatabase
    }
}
```

Task 4: Creating the Repository

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

Tip: Since TodoRepository has similar functions as TodoDao interface. You can generate the skeleton of repository methods by extending TodoDao interface, ask Android Studio to generate the method interfaces then remove the implements TodoDao.

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

```
private val todoDao by lazy {
    TodoDatabase.getDatabase(context).todoDao()
}
```

2. Implement the repository functions by calling the corresponding TodoDao function.
3. Run and Test your implementation.

Task 5: Query one to Many Relationship

An important part of designing a relational database the ability to query data from multiple tables. Using @Relation annotation you can easily get a project and its associated todos in one query.

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>
```

Task 7: Testing the database queries using Database Inspector

Test the app 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.