Metalmann International Interview Round 3

Assignment **: Basic Mobile App Development**

Name : Adarsh Subhas Nayak
College : PES University
Email : adarshsnayak2002@gmail.com
Phone No : +91 7795702175

I chose to build the basic **To-Do mobile application using Kotlin for Android Platform.**
Link to the github repo : Metalmann-to-do-application.

Tech stack : Jetpack Compose, Kotlin, Room Database
Architecture used - MVVM (Model View ViewModel) architecture

Features of the application:

1. Add a task with a title and an optional description for the same.
2. One can delete the task and a snack bar displays a message saying the task has been deleted.
3. One can undo the delete operation.
4. Check and uncheck tasks.
5. Data is persistent even after the app gets refreshed or closed and reopened.

What is MVVM and why I chose MVVM architecture:

1. Model represents the data source which is a repository that collects data.
2. View is the visible part of the app. It gets updated once it receives events from ViewModel. It should not contain business logic.
3. ViewModel contains the business logic. It has direct communication with the Model to get access to data.

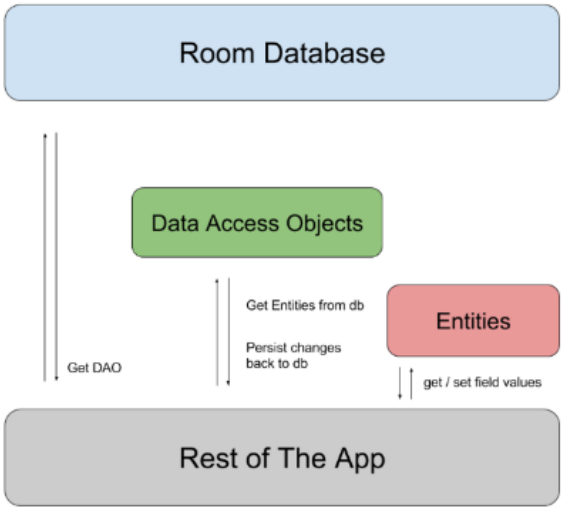
I have chosen MVVM architecture to have a better structure with separate view, model and viewmodel. This will give us a clear understanding of functionality of each part. Code is readable and easy to maintain.

Dependencies :

1. Compose dependencies: viewmodel, navigation
2. Dagger Hilt- dependency injection.

Database : - Room database

We want to make the data persistent, so that the data is sustained even when the application is refreshed or restarted. That's why we make use of the Room Database.



There are three major components in our Room database:

1. The database class that holds the database and serves as the main access point for the underlying connection to your app's persisted data.

```
import androidx.room.Database
import androidx.room.RoomDatabase

@Database(
    entities = [Todo::class],
    version = 1
)
abstract class TodoDatabase: RoomDatabase() {

    abstract val dao: TodoDao
}
```

Creation of a class called 'ToDoDatabase' extending 'RoomDatabase'..

2. Data entities that represent tables in your app's database.

```kotlin
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity
data class Todo(
    val title: String,
    val description: String?,
    val isDone: Boolean,
    @PrimaryKey val id: Int? = null
)
```

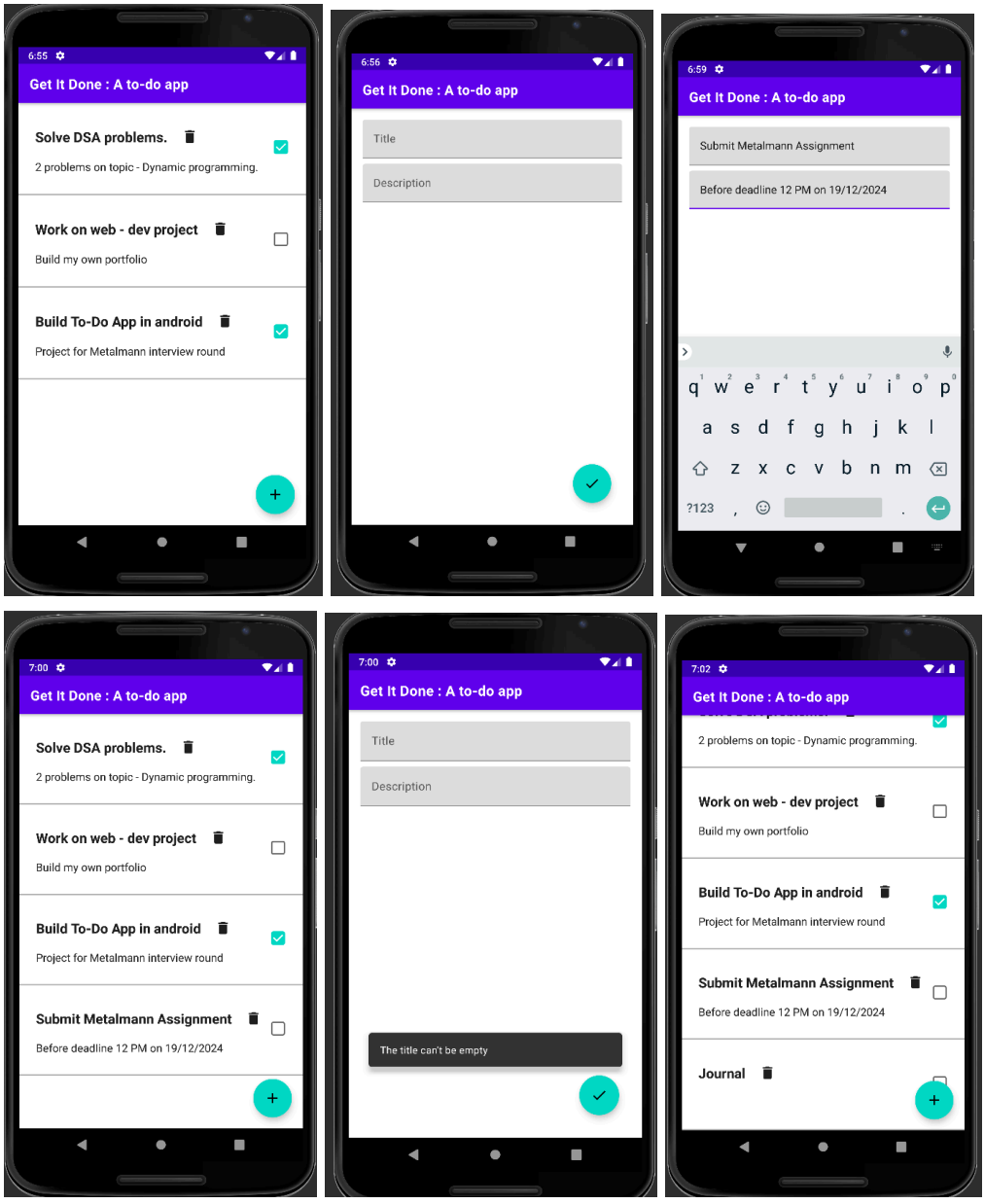
Defining an entity class named ‘ToDo’ with ‘id’ as the primary key, ‘title’, ‘description’ which is optional, and ‘isCompleted’ fields. Here, description is made nullable because the user can choose to keep the description to be null but he can’t keep the title empty. id if not given is assigned by the room database itself, so it is also nullable.

3. Data access objects(DOAs) that provide methods that your app can use to query, update, insert, and delete data in the database.

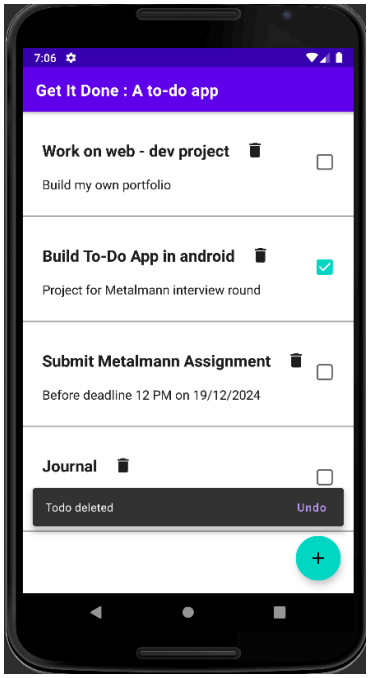
```kotlin
import androidx.room.*
import kotlinx.coroutines.flow.Flow

@Dao
interface TodoDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertTodo(todo: Todo)

    @Delete
    suspend fun deleteTodo(todo: Todo)

    @Query( value: "SELECT * FROM todo WHERE id = :id")
    suspend fun getTodoById(id: Int): Todo?

    @Query( value: "SELECT * FROM todo")
    fun getTodos(): Flow<List<Todo>>
}
```

Define a `ToDoDao` interface with methods for CRUD operations on To-Do items

Now let’s see how the app looks and I’ll later explain how I’ve implemented each part.



Img 1 : Initial Home window.
Img 2 : Page to write the title of the task and description.
Img 3 : Page after writing the task and its description.
Img 4 : Can see the task getting added after we press the Floating tick mark button.
Img 5 : Trying to add an empty task and it shows it can’t be empty.
Img 6 : Adding a task without description, which is possible.
We can mark any task as done by checking the checkbox.



If in case the user deletes the task from the list I have implemented an option to undo the delete. Because the user may delete it by mistake. So in order to retrieve the task in that case this feature has been implemented.

Implementation.

Business Logic Layer

ToDoRepository:

1. Implement the ‘ToDoRepository’ interface as a single source of truth for To-Do data.

```kotlin
import kotlinx.coroutines.flow.Flow

class TodoRepositoryImpl(
    private val dao: TodoDao
): TodoRepository {

    override suspend fun insertTodo(todo: Todo) {
        dao.insertTodo(todo)
    }

    override suspend fun deleteTodo(todo: Todo) {
        dao.deleteTodo(todo)
    }

    override suspend fun getTodoById(id: Int): Todo? {
        return dao.getTodoById(id)
    }

    override fun getTodos(): Flow<List<Todo>> {
        return dao.getTodos()
    }
}
```

2. Utilize ‘ToDoDao’ methods to perform CRUD operations.
3. Wrap database calls with Kotlin coroutines for asynchronous data access and background processing.

UI Layer :

1. To-Do List Screen.
    a. Define a composable function for the To-Do list screen.
    b. Use ‘Column’ or ‘LazyColumn’ to display a list of ‘ToDoItem’ composables.
    c. Each ‘ToDoItem’ composable:
        i. Displays the title and completion status of the To-Do item.
        ii. Uses ‘Checkbox’ for toggling completion status.
        iii. On click, triggers events to the view model.
2. Add/Edit Screen.
    a. Define a composable function for the Add/Edit screen.
    b. Use `TextField` to capture the title and optional description.
    c. Depending on the mode (Add or Edit), pre-populate fields with existing data.
    d. Provide a save button to trigger events to the view model.

MVVM interactions:

1. ViewModel.
    a. Expose observable `LiveData` for the list of To-Do items.
    b. Define methods for adding, updating, and deleting To-Do items.
    c. Observe events sent from the UI (clicks, text changes).
    d. Trigger business logic functions in the repository based on events.
    e. Update the `LiveData` based on data changes from the repository.
2. Dependency injection with Dagger Hilt
    a. Annotations: Annotated the `ToDoRepository` and `ToDoDao` classes with `@Inject`.
    b. Hilt Module: Created a Hilt module that provides dependencies for repositories and DAOs.
    c. ViewModel Injection: Inject the repository into the ViewModel constructor using Hilt.

Understanding the workflow of the application - I have in detail explained how events are triggered and the dataflows:

Use Case 1: Adding a New To-Do Item

- User Action: On the To-Do List screen, tap the "Add To-Do" button (+ icon).
- UI Event: Addition of a new To-Do item is flagged to the ViewModel.
- ViewModel Communication: The ViewModel retrieves an empty ToDo object.
- UI Interaction: The user navigates to the Add/Edit screen and enters a title (optional description).
- User Input: Text entered in the title field is captured by the UI.
- Data Binding (Optional): If data binding is implemented, the title updates the ViewModel's ToDo object automatically.

- Manual Update (Optional): If no data binding, the user taps a "Save" button. This triggers an event to update the ViewModel's ToDo object with the entered title.
- ViewModel and Business Logic: The ViewModel calls the ToDoRepository's insertToDo method with the updated ToDo object.
- Room Database Insertion: The repository uses Room database calls to insert the new ToDo item into the database.
- Data Update and LiveData: The repository updates the LiveData containing the list of To-Do items.
- UI Re-rendering: The ViewModel receives the updated LiveData and triggers a re-rendering of the To-Do List screen.
- Outcome: The newly added To-Do item appears on the list with the entered title and an incomplete status.

Use Case 2: Marking a To-Do Item as Completed

- User Action: On the To-Do List screen, tap the checkbox beside a specific item.
- UI Event: Click event is sent to the ViewModel, identifying the selected To-Do item.
- ViewModel and Business Logic: The ViewModel updates the isCompleted flag of the corresponding ToDo object.
- Room Database Update: The ToDoRepository uses Room database calls to update the isCompleted flag of the item in the database.
- Data Update and LiveData: The repository updates the LiveData containing the list of To-Do items.
- UI Re-rendering: The ViewModel receives the updated LiveData and triggers a re-rendering of the specific To-Do item on the list.
- Outcome: The checkbox of the selected To-Do item is unchecked, and the visual style may change to indicate completion.

Use Case 3: Editing an Existing To-Do Item

- User Action: On the To-Do List screen, tap an existing item to open the Edit screen.
- UI Navigation: The user navigates to the Add/Edit screen pre-populated with the item's existing title and description (if available).
- Data Binding (Optional): If data binding is implemented, changes made in the text fields update the ViewModel's ToDo object automatically.
- Manual Update (Optional): If no data binding, the user taps a "Save" button after modifying the title or description. This triggers an event to update the ViewModel's ToDo object.
- ViewModel and Business Logic: The ViewModel calls the ToDo Repositories updateToDo method with the modified ToDo object.
- Room Database Update: The repository uses Room database calls to update the existing ToDo item in the database with the revised title and description.

- Data Update and LiveData: The repository updates the LiveData containing the list of To-Do items.
- UI Re-rendering: The ViewModel receives the updated LiveData and triggers a re-rendering of the specific To-Do item on the list.
- Outcome: The To-Do item on the list reflects the updated title and description (if modified).