

Healthcare Application SDL Report

| | |
|----------------------|-----------------------------|
| Name: | Respeto Sofia Amihan Molase |
| Admin Number: | 2300600J |
| Email: | 2300600J@student.tp.edu.sg |
| Deadline: | 18 February 2025 |

Table Of Contents

| | |
|--|-----------|
| 1.0 Topic Overview | 2 |
| 2.0 Research | 3 |
| 2.1 Definitions | 3 |
| 2.2 Context | 4 |
| 2.3 Use Cases | 7 |
| 2.4 Strengths | 7 |
| 2.5 Weaknesses | 7 |
| 2.6 Reflection | 8 |
| 3.0 Implementation | 9 |
| 3.1 Setting Up | 9 |
| 3.2 Entity | 9 |
| 3.3 DAO | 10 |
| 3.4 Database | 10 |
| 3.5 Repository | 11 |
| 3.6 View Model | 11 |
| 3.7 View Model Factory | 12 |
| 4.0 Annotations | 13 |
| 5.0 Application | 15 |
| 5.1 Logic | 15 |
| 5.2 Code | 16 |
| 5.2.1 Entity | 16 |
| 5.2.2 DAO | 16 |
| 5.2.3 Database | 17 |
| 5.2.4 Repository | 17 |
| 5.2.5 View Model | 20 |
| 5.2.6 View Model Factory | 21 |
| 5.2.7 Display | 21 |
| 5.3 Code Analysis | 22 |
| 6.0 Plan-Perform-Monitor-Reflect (PPMR) Journal | 27 |
| 6.1 PLAN | 27 |
| 6.2 PERFORM | 28 |
| 6.3 MONITOR | 29 |
| 6.4 REFLECT | 29 |
| 7.0 References | 31 |

1.0 Topic Overview

| | |
|----------------------------|--|
| Area of Uncertainty | Rooms Persistent Library, Kotlin Internal Database using Jetpack Compose |
| Description | Room is a persistence library provided by Jetpack Compose which utilises an abstraction layer over SQLite to allow fluent database access. Instead of using SQLite directly, Room simplifies the database set-up, configuration, and interaction process. By using Room, we can persist significant amounts of structured data locally. |
| Research Scope | <ol style="list-style-type: none"> 1. Definitions A foundation for understanding the subject is by understanding key components and concepts in order to obtain an overarching view. 2. Context Background information and terminology understanding for the Rooms Database. This is crucial as it influences how information is perceived and interpreted. 3. Use Cases Various use cases are provided to describe the specific situations where Rooms can be applied. 4. Strengths 5. Weaknesses 6. Reflection |
| Purpose | <p>In the Healthcare Application I am building, one of the features of the app is the ability to store user data and allow offline access to their profile, health logs, physical activities, medications, and settings. Room can then be used as a local database layer for my application's functionality as it allows the app to cache the data such that users can access it even when there is no internet connection available. In order to maintain data integrity, any user-initiated content changes will only be allowed once the device re-connects with the internet. Room ensures that user data is persistent, stored in a structured format, and can be queried with minimal performance overhead.</p> <p>Unlike SQLite or shared preferences, Room provides an abstraction layer over raw SQL, reducing the boilerplate code while ensuring performance and security. Furthermore, Rooms can be easily integrated into my MVVM infrastructure as well as the libraries I plan on using.</p> <p>Besides serving my project, by learning Rooms, I will be able to develop my skills in database management by learning how to manage a local database, understand lifecycle management, handle interactions between the local and remote data through a repository, as well as ensure that data persists across application configurations. These skills will be useful when it comes to my WorldSkills Competition.</p> <p>At the end of this report, I aim to develop my own modular boilerplate code that can be easily integrated into other mobile projects whether it be small or large in scale. This will allow me to better master creating dynamic code that can be utilised for different scenarios with minimal refactoring.</p> |

2.0 Research

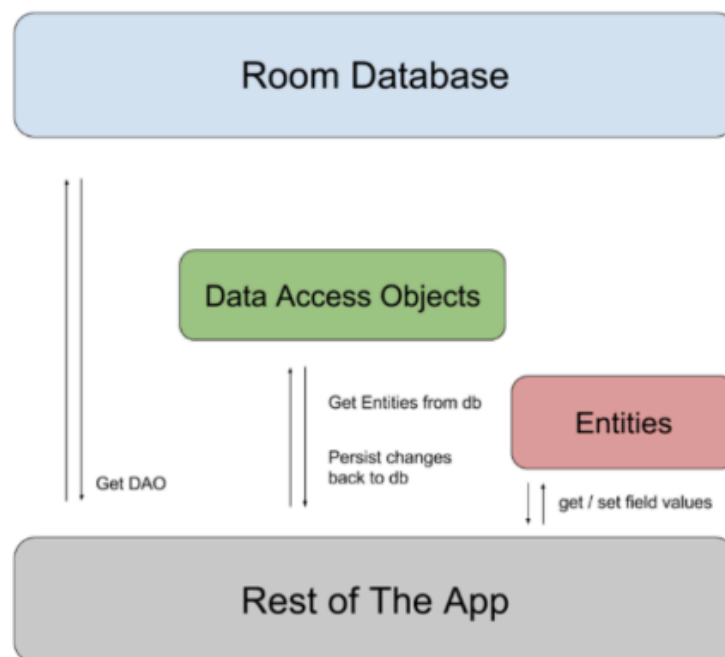
2.1 Definitions

| | |
|---------------------------------|--|
| Rooms | Room is a persistence library in Android that provides an abstraction layer over SQLite, making it easier to handle database operations while ensuring robust data management. It enforces compile-time validation of SQL queries and integrates seamlessly with Kotlin coroutines and LiveData. |
| Entity | An entity represents a table in the Room database. Each instance of an entity class corresponds to a row in the database, and the class fields represent columns. |
| Data Access Object (DAO) | DAOs are interfaces that provide methods that the application uses to retrieve, update, insert, and delete data in the database. DAOs abstract the database interactions and are written using SQL queries. |
| Database | The database class serves as the main access point to the Room database. It provides the application with instances of the DAOs associated with that database. |
| Repository | The repository acts as a mediator, often between data sources such as a Room Database and remote APIs. It serves as the single source of truth, encapsulating the logic for fetching and storing data. It utilises Kotlin Coroutines such that the operations carried out avoids blocking the UI thread. |
| View Model | The View Models are used to store and manage UI-related data in a lifecycle-conscious way. It provides data to the UI by creating an instance of the repository class as LiveData objects. This part of Android's architecture also ensures that the process survives configuration changes. |
| View Model Factory | A class that returns instances of a View Model. This is necessary for parameterised View Model creation such that the correct data is passed through each instance. |
| UI Composables | Jetpack Compose's building blocks to define and render UI elements. |

2.2 Context

Room is a persistent library introduced as part of Jetpack Compose, serving as a tool to manage data efficiently and securely for a seamless user experience. It provides an abstraction layer over SQLite to allow fluent database access for developers to focus on application logic without needing to know about low-level and unnecessary configuration details. Code is made easier to understand as the database operations have been simplified through compile-time query verification and the elimination of excessive boilerplate code.

Room comprises of three primary components: Entities, DAOs, and the Room Database. Once the components have been established, the Room database binds them together, serving as the main access point for the application to interact with the database. These components are then efficiently utilised through the Android's MVVM Architecture through encapsulation and abstraction. By employing Room in this architecture, developers can keep the ViewModel focused solely on UI logic, with data operations abstracted within the Repository.



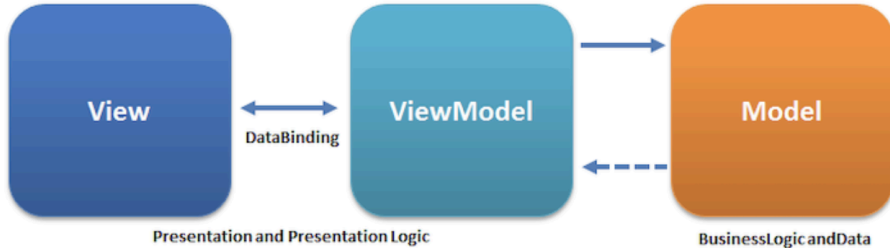
The Repository is then employed to further enhance the abstraction provided to ensure that View Models only access required data without handling the complexities of database operations. By acting as the Single Source Of Truth (SSOT), the Repository mediates between Room and other data sources, such as remote APIs, if applicable. Kotlin's coroutines also work alongside this architecture and allow the asynchronous execution of tasks, allowing multiple tasks to run and thus increasing application efficiency.

In order to further streamline development, Room integrates seamlessly with other Jetpack Components such as LiveData, which allows UI components to observe changes in data. This ensures that when the underlying database data changes, the UI is updated automatically thus resulting in more responsive and engaging applications.

Furthermore, Room is frequently paired with the Singleton Method Design Pattern that enhances database interaction and system performance by providing only one instance of the Room database. This conserves resources and maintains data consistency, especially when it is utilised with data binding.

Overall, Room offers a robust solution for local data storage, efficient database interaction, and seamless UI integration. The Rooms database can be used for small-scale applications requiring simple data persistence as well as complex projects demanding sophisticated data management strategies. By the end of this report, I would be able to create a boilerplate that can be used for either scenario, delivering a scalable and efficient solution.

| | |
|--------------------------|---|
| Data Persistence | Data persistence is the longevity of data after the application that created it has been closed. It involves saving data in a non-volatile storage system so that the data's value can be retrieved reliably later. This system involves a type of memory that can retain information for a long term, even if the application is no longer running. The data remains consistent and nothing is lost between sessions, thus maintaining data integrity. |
| Abstraction | <p>In general, abstraction is one of the four concepts related to Object-Oriented Programming (OOP) that allows developers to focus on the essential aspects of an object rather than its implementation details.</p> <ul style="list-style-type: none"> a. Data Abstraction - An object representing some data but the underlying characteristics or structure of that data is not showcased b. Process Abstraction - The underlying implementation details of a process are hidden <p>By hiding these processes, it makes it easier for the programmer to understand as they need not deal with the unnecessary complexities.</p> |
| SQLite | SQLite is a free and open-source relational database management system. |
| Structured Data | Data that has a standardised format, typically tabular with rows and columns that clearly define data attributes. |
| Unstructured Data | Data with no set data model, or data that has not yet been ordered in a predefined way such as emails, images, video files, etc. These are generally stored in NoSQL databases. |
| Encapsulation | Like abstraction, encapsulation is another fundamental concept related to OOP. This process is the combination of attributes and methods that work with that data into a single unit known as a class. This protective layer around the data maintains its integrity and prevents unauthorised access. |

| | |
|--|---|
| Singleton Method Design Pattern | The Singleton Method Design Pattern ensures a class has only one instance and provides a global access point to it. This is an application of Abstraction and is already implemented when creating a Room Database via the Abstract Class. |
| Boilerplate Code | Boilerplate code is seemingly repetitive code that you can reuse with little or no alteration in several different contexts. |
| MVVM Architecture | <p>MVVM (Model-View-ViewModel) Architecture is a software design pattern that separates the graphical user interface from the business logic of an application.</p>  <pre> graph LR View[View] <--> DataBinding ViewModel[ViewModel] ViewModel --> Model[Model] Model -.-> ViewModel subgraph Presentation_and_Presentation_Logic [Presentation and Presentation Logic] View ViewModel end subgraph BusinessLogic_and_Data [BusinessLogic and Data] Model end </pre> <ul style="list-style-type: none"> - Model: Encapsulates the data and business logic of an application such as the Database and DAOs - View: Responsible for displaying the user interface to the end user. It receives input from the user and presents the data provided by the View Model. - View Model: Acts as the bridge between the Model and the View. It provides data and behavior to the View, allowing it to bind directly to the View Model properties and commands. |
| Coroutines | A coroutine is a piece of code that can be suspended and resumed without blocking the executing thread. It's a way to write asynchronous code in a sequential manner. |
| Data Binding | Data Binding is a way to connect an app's user interface (UI) with data. It keeps the UI updated with the latest data automatically, thus reducing the need for manual updates. |
| Instance | An Instance acts as a copy of an object (made from a class) that you can interact with independently from other instances. This allows you to use the same code to create many objects, each with its own set of data. |
| Superclass | The class from which another class is derived from. For example, our Database will be the subclass of the superclass RoomDatabase(). |
| Concurrency | Concurrency is a concept that enables multiple tasks to execute simultaneously, leading to improved performance and responsiveness. |
| Asynchronous Programming | When program tasks can run simultaneously because they are independent of one another. |

2.3 Use Cases

On a small scale, Room is ideal for local data storage, where it efficiently handles user preferences, app settings, and lightweight data that needs to persist between app sessions, ensuring that the data remains available even when the app is closed or restarted.

In medium-scale applications, Rooms can be used for caching API responses to enable offline functionality, allowing previously fetched information to be retrieved even without an internet connection. This lets users browse content, with any user-initiated changes synced to their server once connectivity is restored, if applicable. By caching API data locally and syncing with remote sources through the Repository, Room ensures a seamless user experience.

On a larger scale, Room can manage complex data relationships and operations involving multiple entities and intricate business logic. Developers can build scalable solutions that handle large amounts of interconnected data through this relational database which has efficient query handling and schema migrations.

2.4 Strengths

1. An abstraction layer is provided over the Room database to get clear access to the database
2. Room maps the database object into a Java object without the boilerplate codes
3. Room provides the annotation to perform any operation in the database instead of writing the raw SQL queries
4. Room provides compile-time query verification, which will reduce general errors and minimise runtime errors, thus enhancing application stability. Compared to traditional SQLite operations, where SQL queries are prone to typographical errors, Room ensures that these queries are pre-validated during compilation.

2.5 Weaknesses

| Weaknesses | Mitigations |
|--|--|
| Barriers for developers not familiar with SQL Syntax | Room's use of annotations as well as Kotlin extensions make it more developer-friendly |
| Potential performance issues with extremely large datasets or high-frequency data access | Using pagination or lazy loading techniques to only load subsets of data when necessary helps solve this issue |
| Learning curve for managing both local and remote servers | Libraries such as Retrofit allow these implementations to be streamlined easily |

2.6 Reflection

Through this Self-Directed Learning, I have learnt a lot about database management through the integration of Room as a local database solution for my Healthcare Application. This report has allowed me to further my understanding of database persistence as well as enhance my proficiency in handling backend systems across mobile applications. The skills that I have adapted here will be valuable as I work on larger-scale applications as well as my competition for WorldSkills.

This experience has also deepened my understanding of Kotlin and Android Development in terms of implementing efficient and asynchronous data management with Kotlin Coroutines. I've improved in the way I navigate complex interactions between local and remote data sources which is crucial for maintainability as I develop as a programmer. Learning how to leverage the Repository pattern for handling data has also been an important part of my learning journey as I had the opportunity to utilise LiveData and ViewModels. Overall, not only did I learn more about the Room Persistent Library, but I was also able to learn more about Kotlin Mobile Application Development in general, a truly enriching experience.

3.0 Implementation

Learning Project: <https://github.com/sofiaamihan/simple-notes-application>

3.1 Setting Up

```
id("com.google.devtools.ksp") version "2.0.21-1.0.27" apply false
```

```
[versions]
roomRuntime = "2.6.1"
runtimeLivedata = "1.7.7"

[libraries]
androidx-room-compiler = { module = "androidx.room:room-compiler", version.ref =
"roomRuntime" }
androidx-room-ktx = { module = "androidx.room:room-ktx", version.ref = "roomRuntime" }
androidx-room-runtime = { module = "androidx.room:room-runtime", version.ref =
"roomRuntime" }
androidx-runtime-livedata = { module = "androidx.compose.runtime:runtime-livedata",
version.ref = "runtimeLivedata" }
```

```
plugins {
    id("com.google.devtools.ksp")
}

dependencies {
    implementation(libs.androidx.room.runtime)
    ksp(libs.androidx.room.compiler)
    implementation(libs.androidx.runtime.livedata)
    implementation(libs.androidx.room.ktx)
}
```

3.2 Entity

```
@Entity(tableName = "notes_table")
data class Note(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val title: String,
    val category: String,
    val description: String
)
```

3.3 DAO

```
@Dao
interface NoteDao {

    @Query("SELECT * FROM notes_table")
    fun getNotes(): LiveData<List<Note>>

    @Upsert
    suspend fun insertNotes(notes: Note)

    @Update
    suspend fun updateNotes(notes: Note)

    @Query("DELETE FROM notes_table WHERE id = :noteId")
    suspend fun clearNotes(noteId: Int)

}
```

3.4 Database

```
@Database(
    entities = [Note::class],
    version = 1,
    exportSchema = false
)
abstract class NotesDatabase : RoomDatabase() {
    abstract val noteDao: NoteDao

    companion object {
        @Volatile
        private var INSTANCE: NotesDatabase? = null

        fun getDatabase(context: Context): NotesDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    NotesDatabase::class.java,
                    "notes_database"
                ).fallbackToDestructiveMigration().build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

3.5 Repository

```
class NotesRepository(private val noteDao: NoteDao) {

    fun getNotes(): LiveData<List<Note>> {
        return noteDao.getNotes()
    }

    suspend fun insertNotes(note: Note){
        noteDao.insertNotes(note)
    }

    suspend fun updateNotes(note: Note) {
        noteDao.updateNotes(note)
    }

    suspend fun deleteNotes(note: Note){
        noteDao.deleteNotes(note.id)
    }
}
```

3.6 View Model

```
class NotesViewModel(application: Application): AndroidViewModel(application) {

    private val repository: NotesRepository

    val noteList: LiveData<List<Note>>

    init {
        val database = NotesDatabase.getDatabase(application)
        val noteDao = database.noteDao
        repository = NotesRepository(noteDao)
        noteList = repository.getNotes()
    }

    fun addNote(note: Note) {
        viewModelScope.launch {
            repository.insertNotes(note)
        }
    }

    fun updateNote(note: Note) {
        viewModelScope.launch {
            repository.updateNotes(note)
        }
    }

    fun clearNote(note: Note) {
        viewModelScope.launch{
            repository.deleteNotes(note)
        }
    }
}
```

3.7 View Model Factory

```
class NotesViewModelFactory(
    private val application: Application
): ViewModelProvider.Factory{

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if(modelClass.isAssignableFrom(NotesViewModel::class.java)) {
            @Suppress("UNCHECKED_CAST")
            return NotesViewModel(application) as T
        }
        throw IllegalArgumentException("Unable to construct ViewModel")
    }
}
```

4.0 Annotations

`@Entity(tableName = "notes_table")` declares a table (entity) in the database - can be multiple

```
data class Note(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val title: String,
    val category: String,
    val description: String
)
```

each val represents one column/feature

`@Insert` (with conflict strategy deletes existing row & replaces it while `@Upsert` updates said row if it exists

```
@Dao
interface NoteDao {

    @Query("SELECT * FROM notes_table")
    fun getNotes(): LiveData<List<Note>>

    @Upsert
    suspend fun insertNotes(notes: Note)

    @Update
    suspend fun updateNotes(notes: Note)

    @Query("DELETE FROM notes_table WHERE id = :noteId")
    suspend fun clearNotes(noteId: Int)
}
```

lifecycle-aware data holder that notifies observers when the data changes

use coroutines for functions that don't need to return anything to increase efficiency by not interrupting the main thread

```
@Database(
    entities = [Note::class],
    version = 1,
    exportSchema = false
)
abstract class NotesDatabase : RoomDatabase() {
    abstract val noteDao: NoteDao
}
```

to prevent constant schema exportation during project generations

returns an instance of the DAO to access the operations

ensures that changes made to the variable are visible across threads immediately :: prevents potential race conditions

encapsulated in this class :: it can only be modified here

only one thread at a time can execute this code :: preventing duplicate databases :: singleton method design pattern

allows the database to be recreated if the schema changes

```
companion object {
    @Volatile
    private var INSTANCE: NotesDatabase? = null

    fun getDatabase(context: Context): NotesDatabase {
        return INSTANCE?.synchronized(this) {
            val instance = Room.databaseBuilder(
                context.applicationContext,
                NotesDatabase::class.java,
                "notes_database"
            ).fallbackToDestructiveMigration().build()
            INSTANCE = instance
            instance
        }
    }
}
```

```

class NotesRepository(private val noteDao: NoteDao) {
    fun getNotes(): LiveData<List<Note>> {
        return noteDao.getNotes()
    }

    suspend fun insertNotes(note: Note){
        noteDao.insertNotes(note)
    }

    suspend fun updateNotes(note: Note) {
        noteDao.updateNotes(note)
    }

    suspend fun deleteNotes(note: Note){
        noteDao.clearNotes(note.id)
    }
}

```

The Repository creates functions that execute the DAO methods by providing the right parameters

- ensures that the UI & business logic are now aware of each other
- allows for isolated testings
- better scalability

```

}
}

```

access database, resource, & other application-level features without needing to pass around an Activity or Fragment :: preventing memory leaks

```

class NotesViewModel(application: Application): AndroidViewModel(application) {
    private val repository: NotesRepository
    val noteList: LiveData<List<Note>>

    init {
        val database = NotesDatabase.getDatabase(application)
        val noteDao = database.noteDao
        repository = NotesRepository(noteDao)
        noteList = repository.getNotes()
    }

    fun addNote(note: Note) {
        viewModelScope.launch {
            repository.insertNotes(note)
        }
    }

    fun updateNote(note: Note) {
        viewModelScope.launch {
            repository.updateNotes(note)
        }
    }

    fun clearNote(note: Note) {
        viewModelScope.launch{
            repository.deleteNotes(note)
        }
    }
}

```

view model's only interaction with data is through the repo. this pattern can be seen throughout the code

val is used instead of var to ensure that the reference itself cannot be reassigned. LiveData values can change over time without requiring the reference itself to change

central location to instantiate view models

```

class NotesViewModelFactory(
    private val application: Application
): ViewModelProvider.Factory{

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if(modelClass.isAssignableFrom(NotesViewModel::class.java)) {
            @Suppress("UNCHECKED_CAST")
            return NotesViewModel(application) as T
        }
        throw IllegalArgumentException("Unable to construct ViewModel")
    }
}

```

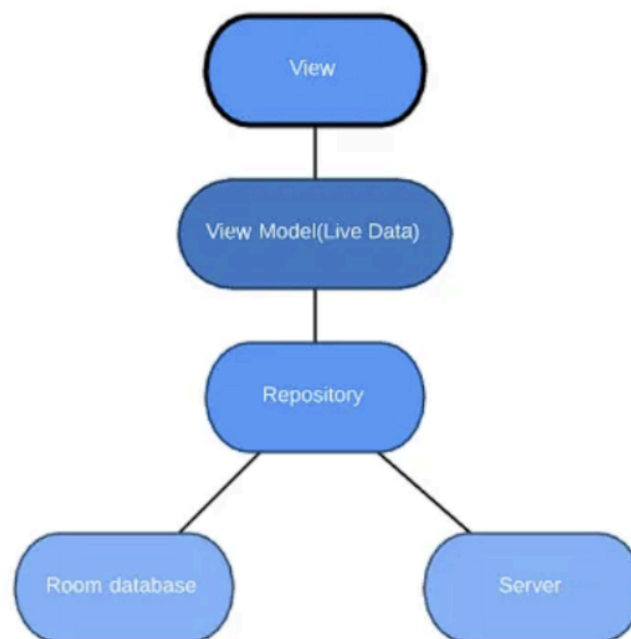
T is a type parameter that can be substituted with any type when the class or function is used

Checks if the model class we received is actually the correct viewmodel

5.0 Application

5.1 Logic

For my official healthcare application, particularly the Health Service Database and Microservice, there will be two sources of data at any given time. One will be the Room database stored inside the device and the other will be the remote server hosted on AWS. By utilising the aforementioned MVVM structure, the Repository Class will serve as the singular place where data is fetched from and interacted with; this class will contain all the mechanics for the Rooms as well as the Remote Database. This system is emphasised by the diagram below:



As for the flow of this architecture, when sending requests, the Remote Server is first updated as it is more likely to fail because of reasons like internet connection, slow internet, high network, etc. Therefore, we need to attend to the AWS Server first to reduce the chances of discrepancies being created between the local and remote database. Once this request is successful, the updated information is then added into the Room Database, where we will ultimately GET all the data to be used and displayed in the application. Rooms will serve as the SSOT in the Repository. This system guarantees that requests will only be successful if both the remote and local database are in sync.

Besides the general flow, for the initial run of the application, the repository will be empty. So until then, we can show the loading dialog to let the user know that the data is being fetched. Only once the API returns the response successfully can we save the response in the Room Database using the Upsert Query. The repository returns the object from the Room database which now has data in it. Finally, the UI is updated and the loading dialogue is dismissed. For subsequent times, the loading dialogue is no longer shown.

5.2 Code

5.2.1 Entity

```
@Entity(tableName = "activity")
data class Activity(
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "user_id")
    val userId: Int,
    @ColumnInfo(name = "category_id")
    val categoryId: Int,
    @ColumnInfo(name = "time_taken")
    val timeTaken: String,
    @ColumnInfo(name = "calories_burnt")
    val caloriesBurnt: Double,
    @ColumnInfo(name = "step_count")
    val stepCount: Double,
    @ColumnInfo(name = "distance")
    val distance: Double,
    @ColumnInfo(name = "walking_speed")
    val walkingSpeed: Double,
    @ColumnInfo(name = "walking_steadiness")
    val walkingSteadiness: Double,
)
```

5.2.2 DAO

```
interface ActivityDao {
    @Query ("SELECT * FROM activity WHERE user_id= :userId AND DATE(time_taken)= :date")
    fun getAllActivities(userId: Int, date: String): List<Activity>

    @Upsert
    suspend fun addActivity(activity: Activity)

    @Query("DELETE FROM activity WHERE id = :id")
    suspend fun deleteActivity(id: Int)
}
```

5.2.3 Database

```
@Database(
    entities = [User::class, Activity::class, Category::class, Medication::class,
    Time::class],
    version = 7,
    exportSchema = false
)
abstract class HealthServiceDatabase : RoomDatabase() {
    abstract val userDao: UserDao
    abstract val activityDao: ActivityDao
    abstract val medicationDao: MedicationDao
    abstract val categoryDao: CategoryDao
    abstract val timeDao: TimeDao

    companion object {
        @Volatile
        private var INSTANCE: HealthServiceDatabase? = null

        fun getDatabase(context: Context): HealthServiceDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    HealthServiceDatabase::class.java,
                    "health_service_database"
                ).fallbackToDestructiveMigration().build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

5.2.4 Repository

```
data class ActivityResponse(
    val id: Int,
    val userId: Int,
    val categoryId: Int,
    val timeTaken: String,
    val caloriesBurnt: Double,
    val stepCount: Double,
    val distance: Double,
    val walkingSpeed: Double,
    val walkingSteadiness: Double
)

class HealthServiceRepository(
    private val tokenDataStore: TokenDataStore,
    private val context: Context
){
    private val database = HealthServiceDatabase.getDatabase(context)
    private val userDao = database.userDao
    private val activityDao = database.activityDao
    private val medicationDao = database.medicationDao
    private val categoryDao = database.categoryDao
}
```

```

private val timeDao = database.timeDao

private val baseUrl = "<https://f5fqqafe6e.execute-api.us-east-1.amazonaws.com>"

suspend fun getActivities(
    userId: Int,
    date: String
): Result<List<ActivityResponse>> {
    return withContext(Dispatchers.IO) {
        try {
            val url = URL("$baseUrl/activity/$userId/$date")
            val connection = url.openConnection() as HttpURLConnection
            connection.requestMethod = "GET"

            val token = tokenDataStore.getToken.first()
            if (token != null) {
                connection.setRequestProperty("Authorization", token)
                Log.d("Token Here", token)
            } else {
                Log.e("Health Service Repository", "Token is null")
                return@withContext Result.Error(Exception("Token is null"))
            }

            val responseCode = connection.responseCode
            if (responseCode == HttpURLConnection.HTTP_OK) {
                val response = connection.inputStream.bufferedReader().use {
                    it.readText() }

                val jsonArray = JSONArray(response)
                val contentList = mutableListOf<ActivityResponse>()

                for (i in 0 until jsonArray.length()) {
                    val jsonResponse = jsonArray.getJSONObject(i)
                    val activities = ActivityResponse(
                        id = jsonResponse.getInt("id"),
                        userId = jsonResponse.getInt("user_id"),
                        categoryId = jsonResponse.getInt("category_id"),
                        timeTaken = jsonResponse.getString("time_taken"),
                        caloriesBurnt = jsonResponse.getDouble("calories_burnt"),
                        stepCount = jsonResponse.getDouble("step_count"),
                        distance = jsonResponse.getDouble("distance"),
                        walkingSpeed = jsonResponse.getDouble("walking_speed"),
                        walkingSteadiness =
                            jsonResponse.getDouble("walking_steadiness")
                    )
                    val activity = Activity(
                        id = activities.id,
                        userId = activities.userId,
                        categoryId = activities.categoryId,
                        timeTaken = activities.timeTaken,
                        caloriesBurnt = activities.caloriesBurnt,
                        stepCount = activities.stepCount,
                        distance = activities.distance,
                        walkingSpeed = activities.walkingSpeed,
                        walkingSteadiness = activities.walkingSteadiness
                    )
                    contentList.add(activity)
                }
            }
        }
    }
}

```

```

        Log.d("Got Activities !!", "$activities")
        activityDao.addActivity(activity)
    }
    val cachedActivities = activityDao.getAllActivities(userId, date)
    val activityList = mutableListOf<ActivityResponse>()
    for (i in cachedActivities) {
        val activity = ActivityResponse(
            id = i.id,
            userId = i.userId,
            categoryId = i.categoryId,
            timeTaken = i.timeTaken,
            caloriesBurnt = i.caloriesBurnt,
            stepCount = i.stepCount,
            distance = i.distance,
            walkingSpeed = i.walkingSpeed,
            walkingSteadiness = i.walkingSteadiness
        )
        activityList.add(activity)
    }

    Log.d("HealthServiceRepo", "Successfully cached: $activityList")
    return@withContext Result.Success(activityList)
} else {
    return@withContext Result.Error(Exception("Activities not found in
cache"))
}

} catch (e: Exception) {
    Log.e(
        "HealthServiceRepository",
        "Error getting categories: ${e.localizedMessage}",
        e
    )
    return@withContext Result.Error(e)
}
}

suspend fun getAllActivities(
    userId: Int,
    date: String
): List<Activity> {
    return withContext(Dispatchers.IO) {
        return@withContext activityDao.getAllActivities(userId, date)
    }
}
}

```

5.2.5 View Model

```
class GetActivitiesViewModel(
    private val healthServiceRepository: HealthServiceRepository,
): ViewModel() {
    var state by mutableStateOf(HealthResultState())
    fun getActivities(
        userId: Int,
        date: String
    ) {
        viewModelScope.launch {
            state = state.copy(loadingState = true)

            when (val response = healthServiceRepository.getActivities(userId, date)) {
                is Result.Success -> {
                    state = state.copy(
                        successState = true,
                        activityList = response.data
                    )
                }
                is Result.Error -> {
                    state = state.copy(
                        errorState = true,
                        errorMessage = "Get Activities failed:
${response.exception.localizedMessage}"
                    )
                }
            }
            state = state.copy(loadingState = false)
        }
    }
}

class GetAllActivitiesViewModel(
    private val healthServiceRepository: HealthServiceRepository
): ViewModel() {
    var state by mutableStateOf(HealthResultState())
    fun getAllActivities(
        userId: Int,
        date: String
    ) {
        viewModelScope.launch {
            state = state.copy(loadingState = true)

            val result = healthServiceRepository.getAllActivities(userId, date)
            if(result != null){
                state = state.copy(successState = true)
                state.cachedActivityList = result
            } else {
                state = state.copy(errorState = true, errorMessage = "Get Activities
Failed")
            }
            state = state.copy(loadingState = false)
        }
    }
}
```

5.2.6 View Model Factory

```
@Suppress("UNCHECKED_CAST")
class HealthServiceViewModelFactory(
    private val healthServiceRepository: HealthServiceRepository
) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return when {
            modelClass.isAssignableFrom(GetActivitiesViewModel::class.java) ->
            GetActivitiesViewModel(
                healthServiceRepository
            ) as T
            modelClass.isAssignableFrom(GetAllActivitiesViewModel::class.java) ->
            GetAllActivitiesViewModel(
                healthServiceRepository
            ) as T
            else -> throw IllegalArgumentException("Unknown ViewModel class")
        }
    }
}
```

5.2.7 Display

```
@Composable
fun SampleScreen(
    healthServiceViewModelFactory: HealthServiceViewModelFactory,
    tokenDataStore: TokenDataStore
){
    val dateKeyFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd")
    val dateKey = LocalDateTime.now().format(dateKeyFormatter)

    val getActivitiesViewModel: GetActivitiesViewModel = viewModel(factory =
    healthServiceViewModelFactory)
    val getAllActivitiesViewModel: GetAllActivitiesViewModel = viewModel(factory =
    healthServiceViewModelFactory)

    val remoteAState = getActivitiesViewModel.state
    val localAState = getAllActivitiesViewModel.state

    LaunchedEffect(Unit){
        getActivitiesViewModel.getActivities(
            tokenDataStore.getId.first()?.toInt() ?: 0,
            dateKey.toString()
        )
        Log.d("Launch", "${remoteAState.activityList}")
    }
    LaunchedEffect(localState){
        getAllActivitiesViewModel.getAllActivities(
            tokenDataStore.getId.first()?.toInt() ?: 0,
            dateKey.toString()
        )
        Log.d("Scope", "${localAState.cachedActivityList}")
    }
}
```

5.3 Code Analysis

```
@Entity(tableName = "activity")
data class Activity(
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "user_id")
    val userId: Int,
    @ColumnInfo(name = "category_id")
    val categoryId: Int,
    @ColumnInfo(name = "time_taken")
    val timeTaken: String,
    @ColumnInfo(name = "calories_burnt")
    val caloriesBurnt: Double,
    @ColumnInfo(name = "step_count")
    val stepCount: Double,
    @ColumnInfo(name = "distance")
    val distance: Double,
    @ColumnInfo(name = "walking_speed")
    val walkingSpeed: Double,
    @ColumnInfo(name = "walking_steadiness")
    val walkingSteadiness: Double,
)
```

```
@Dao
interface ActivityDao {
    @Query("SELECT * FROM activity WHERE user_id= :userId AND DATE(time_taken)= :date")
    fun getAllActivities(userId: Int, date: String): List<Activity>

    @Upsert
    suspend fun addActivity(activity: Activity)

    @Query("DELETE FROM activity WHERE id = :id")
    suspend fun deleteActivity(id: Int)
}
```

→ No longer needs LiveData due to mutable state management below

```
@Database(
    entities = [User::class, Activity::class, Category::class, Medication::class, Time::class],
    version = 7,
    exportSchema = false
)
abstract class HealthServiceDatabase : RoomDatabase() {
    abstract val userDao: UserDao
    abstract val activityDao: ActivityDao
    abstract val medicationDao: MedicationDao
    abstract val categoryDao: CategoryDao
    abstract val timeDao: TimeDao

    companion object {
        @Volatile
        private var INSTANCE: HealthServiceDatabase? = null

        fun getDatabase(context: Context): HealthServiceDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    HealthServiceDatabase::class.java,
                    "health_service_database"
                ).fallbackToDestructiveMigration().build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

tracks the number of times you update the database schema

```
    }
}
}
```

```

data class ActivityResponse {
    val id: Int,
    val userId: Int,
    val categoryId: Int,
    val timeTaken: String,
    val caloriesBurnt: Double,
    val stepCount: Double,
    val distance: Double,
    val walkingSpeed: Double,
    val walkingSteadiness: Double
}

class HealthServiceRepository {
    private val tokenDataStore: TokenDataStore,
    private val context: Context
}{
    private val database = HealthServiceDatabase.getDatabase(context)
    private val userDao = database.userDao
    private val activityDao = database.activityDao
    private val medicationDao = database.medicationDao
    private val categoryDao = database.categoryDao
    private val timeDao = database.timeDao

    private val baseUrl = "<https://f5fqafe6e.execute-api.us-east-1.amazonaws.com>"

    suspend fun getActivities(
        userId: Int,
        date: String
    ): Result<List<ActivityResponse>> {
        return withContext(Dispatchers.IO) {
            try {
                val url = URL("$baseUrl/activity/$userId/$date")
                val connection = url.openConnection() as HttpURLConnection
                connection.requestMethod = "GET"

                val token = tokenDataStore.getToken.first()
                if (token != null) {
                    connection.setRequestProperty("Authorization", token)
                    Log.d("Token Here", token)
                } else {
                    Log.e("Health Service Repository", "Token is null")
                    return@withContext Result.Error(Exception("Token is null"))
                }

                val responseCode = connection.responseCode
                if (responseCode == HttpURLConnection.HTTP_OK) {
                    val response = connection.inputStream.bufferedReader().use { it.readText() }
                    val jsonArray = JSONArray(response)
                    val contentList = mutableListOf<ActivityResponse>()

                    for (i in 0 until jsonArray.length()) {
                        val jsonResponse = jsonArray.getJSONObject(i)
                        val activities = ActivityResponse(
                            id = jsonResponse.getInt("id"),
                            userId = jsonResponse.getInt("user_id"),
                            categoryId = jsonResponse.getInt("category_id"),
                            timeTaken = jsonResponse.getString("time_taken"),
                            caloriesBurnt = jsonResponse.getDouble("calories_burnt"),
                            stepCount = jsonResponse.getDouble("step_count"),
                            distance = jsonResponse.getDouble("distance"),
                            walkingSpeed = jsonResponse.getDouble("walking_speed"),
                            walkingSteadiness = jsonResponse.getDouble("walking_steadiness")
                        )
                        val activity = Activity(
                            id = activities.id,

```

serves as a Data Transfer Object (DTO) that helps separate the data structure used in the remote API response from the Activity entity

- flexibility in differing data formats
- data mapping
- acts as a potential data preprocessor

database is called in the repository instead of the view model, db is abstracted in the repo

this function acts as a refreshing mechanism

Request Requirement: Parameters

Request Requirement: Token

data is returned in the form of [{} , {} , {}]

→ replace LiveData


```

        userId = activities.userId,
        categoryId = activities.categoryId,
        timeTaken = activities.timeTaken,
        caloriesBurnt = activities.caloriesBurnt,
        stepCount = activities.stepCount,
        distance = activities.distance,
        walkingSpeed = activities.walkingSpeed,
        walkingSteadiness = activities.walkingSteadiness
    )
    contentList.add(activities)
    Log.d("Got Activities !!", "$activities")
    activityDao.addActivity(activity)
}

val cachedActivities = activityDao.getAllActivities(userId, date)
val activityList = mutableListOf<ActivityResponse>()
for (i in cachedActivities) {
    val activity = ActivityResponse(
        id = i.id,
        userId = i.userId,
        categoryId = i.categoryId,
        timeTaken = i.timeTaken,
        caloriesBurnt = i.caloriesBurnt,
        stepCount = i.stepCount,
        distance = i.distance,
        walkingSpeed = i.walkingSpeed,
        walkingSteadiness = i.walkingSteadiness
    )
    activityList.add(activity)
}

Log.d("HealthServiceRepo", "Successfully cached: $activityList")
return@withContext Result.Success(activityList)
} else {
    return@withContext Result.Error(Exception("Activities not found in cache"))
}

} catch (e: Exception) {
    Log.e(
        "HealthServiceRepository",
        "Error getting categories: ${e.localizedMessage}",
        e
    )
    return@withContext Result.Error(e)
}
}

suspend fun getAllActivities(
    userId: Int,
    date: String
): List<Activity> {
    return withContext(Dispatchers.IO) {
        return@withContext activityDao.getAllActivities(userId, date)
    }
}
}

```

following the logic system, room operations are carried out only after operations carried out on the remote server are successful

Necessity:

1. Conversion to fit response format
2. Logging purposes during development
3. Evidence that Rooms serves as a single source of truth

used to offload from the main thread, keeping it free for UI updates & enhanced user experience

```

class GetActivitiesViewModel(
    private val healthServiceRepository: HealthServiceRepository,
): ViewModel() {
    var state by mutableStateOf<HealthResultState>()
    fun getActivities(
        userId: Int,
        date: String
    ) {
        viewModelScope.launch {
            state = state.copy(loadingState = true)

            when (val response = healthServiceRepository.getActivities(userId, date)) {

```

mutableState is used instead of LiveData for UI state management

```

is Result.Success -> {
    state = state.copy(
        successState = true,
        activityList = response.data
    )
}
is Result.Error -> {
    state = state.copy(
        errorState = true,
        errorMessage = "Get Activities failed: ${response.exception.localizedMessage}"
    )
}

state = state.copy(loadingState = false)
}
}

class GetAllActivitiesViewModel(
    private val healthServiceRepository: HealthServiceRepository
): ViewModel() {
    var state by mutableStateOf(HealthResultState())
    fun getAllActivities(
        userId: Int,
        date: String
    ) {
        viewModelScope.launch {
            state = state.copy(loadingState = true)

            val result = healthServiceRepository.getAllActivities(userId, date)
            if(result != null){
                state = state.copy(successState = true)
                state.cachedActivityList = result
            } else {
                state = state.copy(errorState = true, errorMessage = "Get Activities Failed")
            }
            state = state.copy(loadingState = false)
        }
    }
}

```

```

sealed class Result<out R>{
    data class Success<out T>(val data:T) : Result<T>()
    data class Error(val exception :Exception): Result<Nothing>()
}

```

NO DATABASE CALL IN VIEW MODELS:

- this adaptation is more applicable towards the MVVM architecture where the VM only interacts with the repository, keeping it clean & unaware of how data is stored
- however, it's okay for the database to be called in the VM for simple read-only operations where you don't need API data (like the notes application I made)

```

@Suppress("UNCHECKED_CAST")
class HealthServiceViewModelFactory(
    private val healthServiceRepository: HealthServiceRepository
): ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return when {
            modelClass.isAssignableFrom(GetActivitiesViewModel::class.java) -> GetActivitiesViewModel(
                healthServiceRepository
            ) as T
            modelClass.isAssignableFrom(GetAllActivitiesViewModel::class.java) -> GetAllActivitiesViewModel(
                healthServiceRepository
            ) as T
            else -> throw IllegalArgumentException("Unknown ViewModel class")
        }
    }
}

```

```

@Composable
fun SampleScreen(
    healthServiceViewModelFactory: HealthServiceViewModelFactory,
    tokenDataStore: TokenDataStore
){

```

```

val dateKeyFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd")
val dateKey = LocalDateTime.now().format(dateKeyFormatter)

val getActivitiesViewModel: GetActivitiesViewModel = viewModel(factory = healthServiceViewModelFactory)
val getAllActivitiesViewModel: GetAllActivitiesViewModel = viewModel(factory =
healthServiceViewModelFactory)

val remoteAState = getActivitiesViewModel.state
val localAState = getAllActivitiesViewModel.state

LaunchedEffect(Unit){
    getActivitiesViewModel.getActivities(
        tokenDataStore.getId.first()?.toInt() ?: 0,
        dateKey.toString()
    )
    Log.d("Launch", "${remoteAState.activityList}")
}
LaunchedEffect(localState){
    getAllActivitiesViewModel.getAllActivities(
        tokenDataStore.getId.first()?.toInt() ?: 0,
        dateKey.toString()
    )
    Log.d("Scope", "${localAState.cachedActivityList}")
}
}

```

→ the coroutine re-runs whenever the dependencies change which can be useful once scaled

6.0 Plan-Perform-Monitor-Reflect (PPMR) Journal

6.1 PLAN

| Idea/Issues (Topic Knowledge) | |
|--|---|
| What I know | What I don't know, or what I need to find out/work on |
| <ul style="list-style-type: none"> • The fundamentals of Mobile Application Development using Kotlin • Basic usage of ViewModels and Coroutines • Techniques for retrieving data from a remote server | <ul style="list-style-type: none"> • The Room Persistent Library and its implementation • How to create and manage a local database • The process of connecting a local database to a remote database • The recommended architectural patterns to achieve what I want |
| Action (Identify Resources/Information) | |
| Get help from | Resources I should use |
| <ul style="list-style-type: none"> • Teacher in charge • Classmate who is also working on a similar project • Trusted online resources | <ul style="list-style-type: none"> • Udemy courses • Medium articles • Official Room Database documentation • Philipp Lackner's Youtube channel - A professional on Kotlin and Jetpack Compose |
| Time Management | |
| By when must I complete, set deadlines for subtasks | |
| <ol style="list-style-type: none"> 1. By November 2024 - Research on Rooms 2. By December 2024 - Setting Up Rooms as a local database 3. By December 2024 - Setting Up Rooms as a local database that caches information from a Remote database 4. By January 2024 - Finishing this SDL Report | |

6.2 PERFORM

| Strategies that I am using to |
|--|
| <p>1. Monitor progress/keep on task (Is the task taking longer than expected?)</p> <p>I broke down my project into smaller milestones and write down these deadlines on my physical planner from Muji in order to track my progress more easily and prevent myself from getting overwhelmed. I also use Notion to divide and plan these tasks, integrating this system with Google Calendar that constantly reminds me of my upcoming deadlines. If a task is taking longer than expected, I re-evaluate the time allocated to it and adjust my deadlines accordingly. If I genuinely cannot meet an important deadline due to many school commitments, I would request for an extension from my teacher.</p> |
| <p>2. Locate resource/get help (where & how - strategies used to find these)</p> <p>My primary source of information would be trusted online platforms like the official documentations, Stack Overflow, GitHub, and Medium articles where I can find specific coding examples and use-cases. Sometimes I refer to Youtube videos for a more simple explanation incase the documentations are too complex for me to understand. If there is an issue I'm taking too long to debug, I would reach out to my teacher or my friends so that another pair of eyes can help solve the issue.</p> |
| <p>3. Check understanding (strategies to confirm understanding)</p> <p>I like fully understanding boilerplate codes and syntax because in-depth learning on the base topics make it easier for me to apply these concepts on a larger scale or on more complex scenarios. I also do documentations on my progression on note-taking platform Obsidian where I can constantly update code and review my understanding. There are also many coding labs online that allow me to test my understanding.</p> |
| <p>4. Motivate myself</p> <p>I like having multiple productivity platforms that allow me to visualise my goals and check my little goals everyday so these little dopamine achievements allow my to stay motivated at small but consistent intervals.</p> |
| Resources/Information I have gathered |
| <p>List the online resources or books reference I have identified</p> <ul style="list-style-type: none">• https://developer.android.com/develop/ui/compose/documentation• https://kotlinlang.org/docs/home.html• https://www.udemy.com/• https://medium.com/• https://www.youtube.com/@PhilippLackner |

6.3 MONITOR

Evaluating Time Management & Potential Information Sources

Tasks Monitoring (Describe how you monitor your own progress while researching the chosen domain)

In the beginning, I already planned out how to execute my project into manageable chunks, ensuring that I spread out my goals and deadlines reasonably. So in order to monitor myself, I always cross check with my plans, calendars, and reminders. Due to my documentations, I am also able to regularly review the progress of my research against the goals and scope outlined. I also ensure to allocate filler timings incase a task takes longer than required or scheduled breaks to ensure that I do not get burnt out.

Resources Monitoring (Are the resources reliable and level appropriate for the assignment?)

My resources are popular and commonly references among programmers. Additionally, these sources have also been provided to me by the school as recommended learning platforms. However, I still make sure to verify that each of these sources are credible through my own cross-referencing and online research.

6.4 REFLECT

Write a reflection on your learning

Articulated Learning

1. Write about some strategies or skills you have acquired while working through the assignment.

I've learnt how to properly utilise Kotlin Coroutines, ViewModels and LiveData through the MVVM architecture which has really aided in efficiently setting up and managing a local database, implement CRUD operations, and handle asynchronous tasks. I've also learnt how to better plan my schedule to accommodate for this large project on top of my other commitments.

2. Do you think you have progressed well? Did you have better time management?

I think I've progressed well considering I have successfully understood and implemented the Rooms Library both syntax-wise and in general. I've also improved my time management skills by organising my schedule and prioritising important aspects of the project such as functionality before documentation.

3. Given another assignment/project, might there be anything you would do differently from what you have done?

I would have implemented the database design and interactions earlier instead of spending as much time as I did researching. I underestimated the amount of times I needed to modify and redesign the database so I would have saved a lot of time.

4. How would you rate yourself as a Self-Directed Learner?

I would rate myself as a strong self-directed learner. I enjoy researching and implementing concepts on my own and believe that I am a rather self-disciplined and motivated person so I was able to carry out my project well.

Evaluation

The PPMR process has helped me challenge myself towards achieving the desired standards I set for my assignment.


☒ Strongly
Agree

☐ Agree

☐ Disagree

☐ Strongly
Disagree

7.0 References

- I. *Persist data with Room* | Android Developers. (2025). Android Developers. <https://developer.android.com/codelabs/basic-android-kotlin-compose-persisting-data-room#0>
- II. Divyanshutw. (2021, January 20). *How to make an offline cache in android using Room database and MVVM architecture?*  Medium. <https://divyanshutw.medium.com/how-to-make-an-offline-cache-in-android-using-room-database-and-mvvm-architecture-6d1b011e819c>
- III. *What Is Data Persistence? | Full Guide | MongoDB*. (2024). MongoDB. <https://www.mongodb.com/resources/basics/databases/data-persistence>
- IV. Janssen, T. (2023, May). *Stackify*. Stackify. <https://stackify.com/oop-concept-abstraction/>
- V. Sayed, R. (2024, August 12). *Understanding Abstraction in Kotlin OOP: A Deep Dive with Practical Examples*. Medium. <https://medium.com/@ramadan123sayed/abstraction-in-kotlin-oop-a-deep-dive-with-practical-examples-ff3a5e7adac1>
- VI. S, R. A. (2021, July 22). *What is SQLite? Everything You Need to Know*. Simplilearn.com; Simplilearn. <https://www.simplilearn.com/tutorials/sql-tutorial/what-is-sqlite>
- VII. *What is Structured Data? - Structured Data Explained - AWS*. (2023). Amazon Web Services, Inc. <https://aws.amazon.com/what-is/structured-data/>
- VIII. Janssen, T. (2023, May). *Stackify*. Stackify. <https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/>
- IX. GeeksforGeeks. (2016, April 20). *Singleton Method Design Pattern*. GeeksforGeeks. <https://www.geeksforgeeks.org/singleton-design-pattern/>
- X. *What is Boilerplate Code? - Boilerplate Code Explained - AWS*. (2023). Amazon Web Services, Inc. <https://aws.amazon.com/what-is/boilerplate-code/#:~:text=Boilerplate%20code%20is%20computer%20language,minimal%20changes%20for%20different%20situations>
- XI. Onur Cem Işık. (2023, June 30). *Introduction to MVVM Architecture - Onur Cem Işık - Medium*. Medium. <https://medium.com/@onurcem.isik/introduction-to-mvvm-architecture-5c5558c3679>
- XII. *What Is MVVM (Model-View-ViewModel)? | Built In*. (2023). Built In. <https://builtin.com/software-engineering-perspectives/mvvm-architecture>
- XIII. Singh, H. P. (2023, May 22). *Mastering Kotlin Coroutines with Practical Examples*. Medium; hprog99. <https://medium.com/hprog99/mastering-kotlin-coroutines-with-practical-examples-1544e0bdbd64>
- XIV. *Data binding definition – Glossary*. (2022, July). NordVPN. <https://nordvpn.com/cybersecurity/glossary/data-binding/>
- XV. *What is an Instance? Get the Answer Here | Lenovo Singapore*. (2021). Lenovo.com. <https://www.lenovo.com/sg/en/glossary/what-is-instance/>
- XVI. *Inheritance (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance)*. (2024). Oracle.com. <https://docs.oracle.com/javase/tutorial/java/landl/subclasses.html#:~:text=Definitions%3A%20A%20class%20that%20is,class%20or%20a%20parent%20class>

- XVII. Sahu, R. (2023). *Concurrency in Kotlin*. C-Sharpcorner.com.
<https://www.c-sharpcorner.com/article/concurrency-in-kotlin/#:~:text=Concurrency%20is%20a%20fundamental%20concept,using%20multithreading%20and%20synchronization%20techniques>
- XVIII. *Explained: Asynchronous vs. Synchronous Programming*. (2024, October 14). Mendix.
<https://www.mendix.com/blog/asynchronous-vs-synchronous-programming/#:~:text=Asynchronous%20is%20a%20non%20Dblocking%20architecture%2C%20so%20the%20execution%20of,on%20to%20the%20next%20iteration>
- XIX. *Android Room Persistence Library: Complete Guide*. (2024). Daily.dev.
<https://daily.dev/blog/android-room-persistence-library-complete-guide>
- XX. Sayed, R. (2024, August 12). *Kotlin Generics: The Ultimate Guide with Practical Examples*. Medium.
<https://medium.com/@ramadan123sayed/kotlin-generics-the-ultimate-guide-with-practical-examples-ca3f5ca557e7#:~:text=The%20basic%20syntax%20for%20defining,class%20or%20function%20is%20used>.
- XXI. Chavez, A. (2023, August 11). *How to create one-to-many relationship between objects in Android Room database*. Medium.
[https://medium.com/@fofito.1295/how-to-create-one-to-many-relationship-between-objects-in-android-room-database-697470534c4d#:~:text=Replace\)%20or%20employing%20%40Upsert%20,updates%20the%20pre%20existing%20value](https://medium.com/@fofito.1295/how-to-create-one-to-many-relationship-between-objects-in-android-room-database-697470534c4d#:~:text=Replace)%20or%20employing%20%40Upsert%20,updates%20the%20pre%20existing%20value).
- XXII. necessary. (2023, March 16). *Is it necessary to define the functions inside the DAO as suspend?* Stack Overflow.
<https://stackoverflow.com/questions/75755509/is-it-necessary-to-define-the-functions-inside-the-dao-as-suspend>
- XXIII. *Database* | Android Developers. (2025). Android Developers.
[https://developer.android.com/reference/androidx/room/Database#exportSchema\(\)](https://developer.android.com/reference/androidx/room/Database#exportSchema())
- XXIV. val. (2019, December 5). *why we use val for LiveData in getter instead of var*. Stack Overflow.
<https://stackoverflow.com/questions/59192225/why-we-use-val-for-livedata-in-getter-instead-of-var>
- XXV. Sujatha Mudadla. (2023, July 26). *LaunchedEffect in Jetpack Compose: Your Comprehensive Guide*. Medium.
<https://medium.com/@sujathamudadla1213/what-is-launchedeffect-coroutine-api-android-jetpack-compose-76d568b79e63>
- XXVI. Garg, A. (2024, July 17). *Dispatchers.IO Explained: Supercharge Your Android App's Performance*. Medium; DevJoint.
[https://medium.com/devjoint/dispatchers-io-explained-supercharge-your-android-apps-performance-14761b915e6b#:~:text=withContext\(Dispatchers.IO\)%20%7B%20...%20%7D%20%3A,thread%20to%20update%20the%20UI](https://medium.com/devjoint/dispatchers-io-explained-supercharge-your-android-apps-performance-14761b915e6b#:~:text=withContext(Dispatchers.IO)%20%7B%20...%20%7D%20%3A,thread%20to%20update%20the%20UI).