Mobile development

Assignment 4: Working with Databases in Kotlin Android and Using Retrofit in Kotlin Android

Bitanov Assanali (23MD0392)

01.12.2024

Table of Contents

Introduction

In Assignment 4, I built a mobile app to manage user data and fetch posts from an API. The project focused on using Room for local database storage and Retrofit for connecting to a remote API. This helped me learn how to manage data both locally and from the internet in an Android application.

First, I used Room to store user data locally. I created data models, a DAO interface to handle database queries, and a repository to make working with the database easier. The app lets users add, view, and delete their data. I made sure all database tasks run in the background to keep the app responsive.

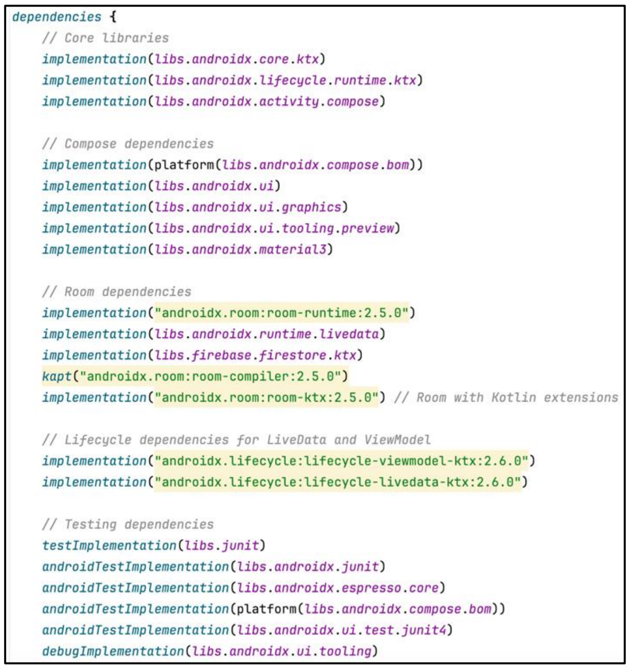
Next, I used Retrofit to fetch posts from a remote API. I set up Retrofit with a base URL, defined the API endpoints, and created data models to match the API responses. I used a repository to handle the API calls, making the app easy to maintain. The posts fetched from the API are displayed in the app's user interface.

I designed the app's UI using Jetpack Compose, which made it easier to build modern and dynamic layouts. I used `LazyColumn` to display lists of users and posts and added buttons to perform actions like adding or deleting users. The UI updates automatically when the data changes, providing a smooth user experience.

This project taught me how to use Room for local storage, Retrofit for working with APIs, and Jetpack Compose for building user interfaces. It also gave me hands-on experience with managing both local and remote data in Android apps.

Working with Databases in Kotlin Android

First, I created a new directory for the project. Assignment4 is the name of the main directory. In the assignment4 folder I created data and network folders. The data folder will contain User.kr, UserDao.kt, AppDatabase.kt, and UserRepository.kr files. The network folder will contain Post.kt, PostRepository.kt, RetrofitInstance.kt, and ApiService.kt files. First I added the necessary dependencies into the gradle file. The file is shown in Figure 1.

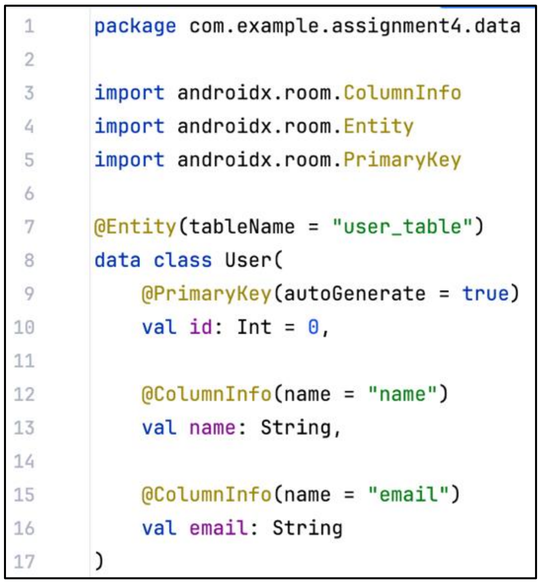


```
dependencies {
    // Core libraries
    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
    implementation(libs.androidx.activity.compose)

    // Compose dependencies
    implementation(platform(libs.androidx.compose.bom))
    implementation(libs.androidx.ui)
    implementation(libs.androidx.ui.graphics)
    implementation(libs.androidx.ui.tooling.preview)
    implementation(libs.androidx.material3)

    // Room dependencies
    implementation("androidx.room:room-runtime:2.5.0")
    implementation(libs.androidx.runtime.livedata)
    implementation(libs.firebase.firestore.ktx)
    kapt("androidx.room:room-compiler:2.5.0")
    implementation("androidx.room:room-ktx:2.5.0") // Room with Kotlin extensions

    // Lifecycle dependencies for LiveData and ViewModel
    implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.6.0")
    implementation("androidx.lifecycle:lifecycle-livedata-ktx:2.6.0")

    // Testing dependencies
    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)
    androidTestImplementation(platform(libs.androidx.compose.bom))
    androidTestImplementation(libs.androidx.ui.test.junit4)
    debugImplementation(libs.androidx.ui.tooling)
```

Figure 1. The dependencies in the gradle file

Here I added core, compose, room, retrofit and other dependencies. The Room persistence library provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite [1]. Retrofit is a library that makes it easier to work with network requests in Android applications. For example, you can use this tool to configure the weather application. The developer sets the parameters of the requests that should be sent to the server and collect temperature data in different cities [2].

Then I created the User data class in the User.kt file. The file is shown in Figure 2.

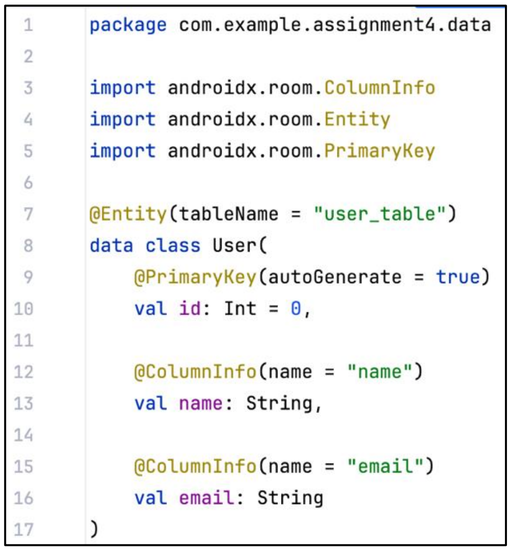


```kotlin
package com.example.assignment4.data

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "user_table")
data class User(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,

    @ColumnInfo(name = "name")
    val name: String,

    @ColumnInfo(name = "email")
    val email: String
)
```

Figure 2. The User class in the User.kt file

The @Entity annotation marks this class as a Room database table. The id property is marked with @PrimaryKey, which makes sure each user in the table has a unique ID. The name and email properties are marked with @ColumnInfo, which sets their column names in the table to "name" and "email". The User class is the main part of the app's local data management. It defines how user information is saved and retrieved, making it easy to add, update, or get user records from the database.

Then I created the Data Access Object (DAO) interface in the UserDao.kt file. When we use the Room persistence library to store our app's data, we interact with the stored data by defining data access objects, or DAOs. Each DAO includes methods that offer abstract access to your app's database [3]. The file is shown in Figure 3.

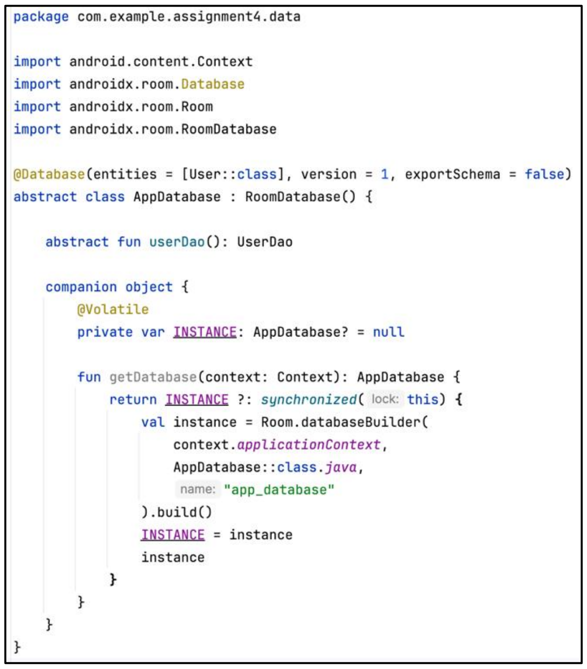


```kotlin
package com.example.assignment4.data

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "user_table")
data class User(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,

    @ColumnInfo(name = "name")
    val name: String,

    @ColumnInfo(name = "email")
    val email: String
)
```

Figure 3. The Data Access Object (DAO) interface in the UserDao.kt file

The UserDao interface defines how the app interacts with the User table in the database. It provides methods to add, delete, and get users, and Room takes care of writing the database code. The insertAll method is used to add one or more users to the database. The delete method removes a specific user from the database. The getAll method gets a list of all users from the database.

Then I created a Room database class in the AppDatabase.kt file. It tells Room how to set up and manage the app's database. This class ensures that the app can store and retrieve data using Room. The AppDatabase.kt file is shown in Figure 4.

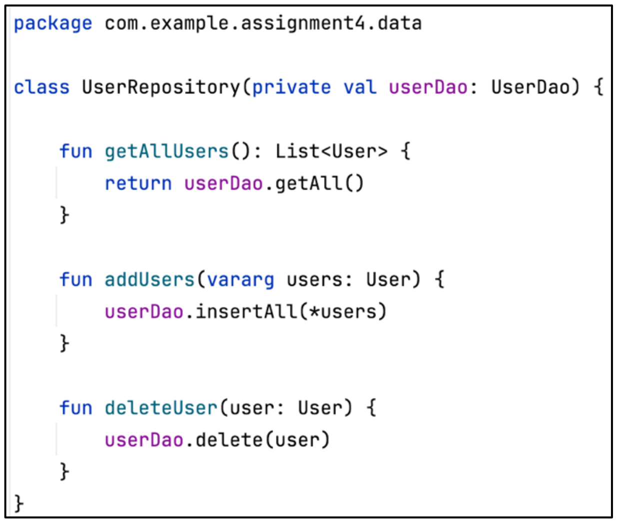


```kotlin
package com.example.assignment4.data

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [User::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {

    abstract fun userDao(): UserDao

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

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                     name: "app_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

Figure 4. The AppDatabase.kt file

The @Database annotation specifies the User class as the entity (table) for the database. The abstract fun userDao() links the UserDao to the database, allowing the app to perform operations like adding or deleting users. The AppDatabase class is the central place for managing the app's database.

Then I created a repository class in the UserRepository.kt file. The file is shown in Figure 5.

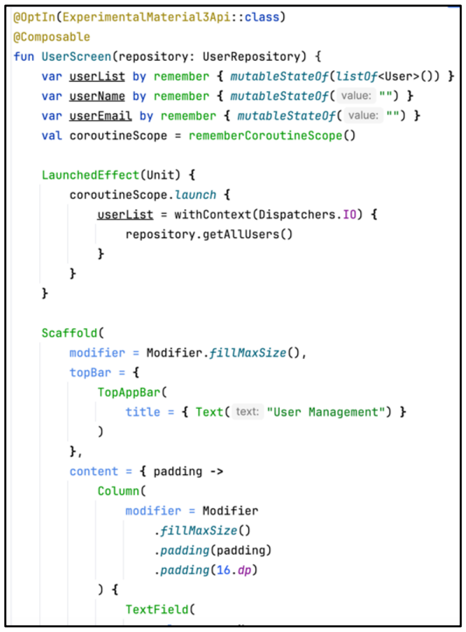


```kotlin
package com.example.assignment4.data

class UserRepository(private val userDao: UserDao) {

    fun getAllUsers(): List<User> {
        return userDao.getAll()
    }

    fun addUsers(vararg users: User) {
        userDao.insertAll(*users)
    }

    fun deleteUser(user: User) {
        userDao.delete(user)
    }
}
```

Figure 5. The UserRepository.kt file

The UserRepository organizes database operations and ensures that the app's UI can interact with the database without directly accessing UserDao. The UserRepository class takes a UserDao as a parameter. It uses UserDao methods to interact with the User table in the database. getAllUsers(): Fetches all users from the database. addUsers(): Adds one or more users to the database using insertAll. deleteUser(): Removes a specific user from the database using delete.

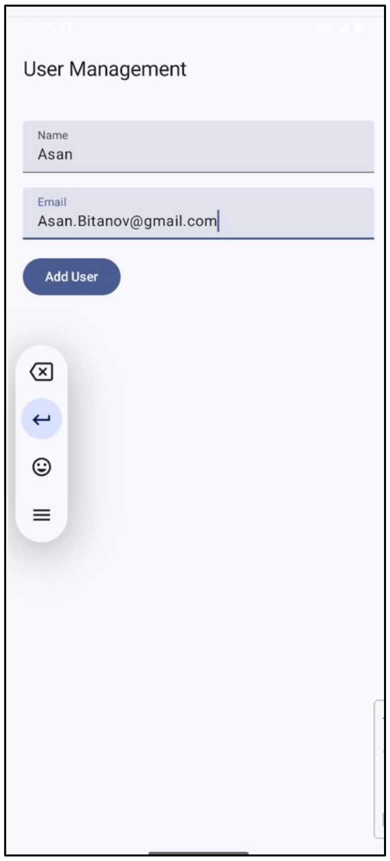
Then I defined the UserScreen function in the MainActivity.kt file. The file is shown in Figure 6.



```kotlin
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun UserScreen(repository: UserRepository) {
    var userList by remember { mutableStateOf(listOf<User>()) }
    var userName by remember { mutableStateOf( value: "") }
    var userEmail by remember { mutableStateOf( value: "") }
    val coroutineScope = rememberCoroutineScope()

    LaunchedEffect(Unit) {
        coroutineScope.launch {
            userList = withContext(Dispatchers.IO) {
                repository.getAllUsers()
            }
        }
    }

    Scaffold(
        modifier = Modifier.fillMaxSize(),
        topBar = {
            TopAppBar(
                title = { Text( text: "User Management") }
            )
        },
        content = { padding ->
            Column(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(padding)
                    .padding(16.dp)
            ) {
                TextField(
```

Figure 6. The UserScreen function in the MainActivity.kt file

It is a composable that handles the user interface for managing a list of users. It allows users to add new entries, view existing ones, and delete users, all while keeping the UI updated dynamically. The LazyColumn displays the list of users from the database. Each user is shown with their name, email, and a delete button. Two TextField components let users enter their name and email. A Button saves the new user to the database via the repository.addUsers() method and refreshes the user list. The UI is shown in Figure 7.


User Management

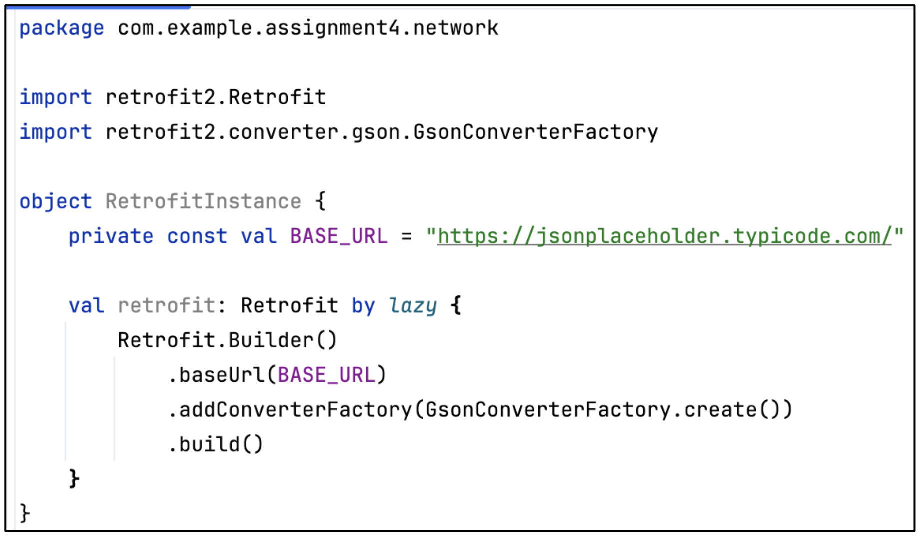
Name
Asan

Email
Asan.Bitanov@gmail.com

Figure 7. The current version of the UI

This UI allows users to input their data easily and submit it to the app. It provides a simple way to manage user information while maintaining a visually appealing design.

Then I created a singleton object for Retrofit initialization in the RetrofitInstance.kt file. The file is shown in Figure 8.

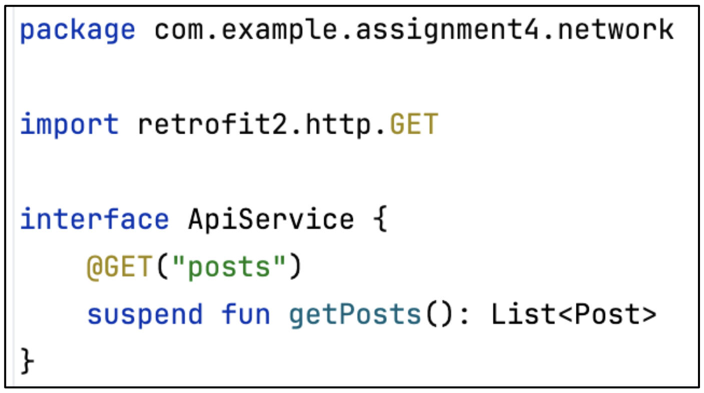


```kotlin
package com.example.assignment4.network

import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

object RetrofitInstance {
    private const val BASE_URL = "https://jsonplaceholder.typicode.com/"

    val retrofit: Retrofit by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }
}
```

Figure 8. The RetrofitInstance.kt file.

The RetrofitInstance centralizes the setup of Retrofit, making it easy to manage the API configuration in one place. This ensures consistent API calls across the app. It provides a ready-to-use Retrofit instance for making network requests. Also, it simplifies API setup by defining the base URL and JSON parsing in a single location.

Then I created an interface that defines the endpoints of the API in the ApiService.kt file. The file is shown in Figure 9.

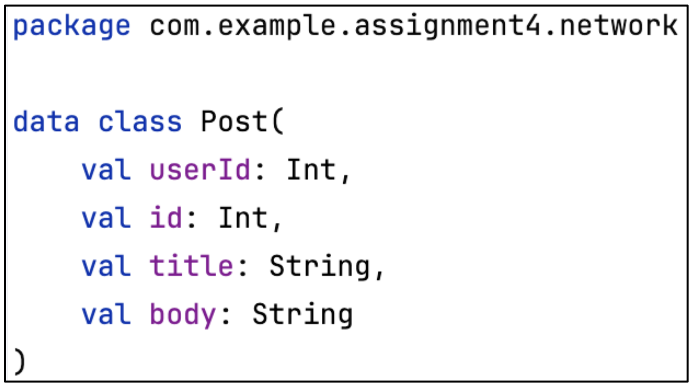


```kotlin
package com.example.assignment4.network

import retrofit2.http.GET

interface ApiService {
    @GET("posts")
    suspend fun getPosts(): List<Post>
}
```

Figure 9. The ApiService.kt file

It uses Retrofit annotations to describe HTTP methods and endpoints. The @GET("posts") annotation defines an HTTP GET request. The getPosts function is marked with suspend, making it compatible with Kotlin coroutines. The function returns List<Post>, where Post is a data class that represents the structure of the API response.

Then I created a data class that represents the JSON response structure of the API in the Post.kt file. The file is shown in Figure 10.

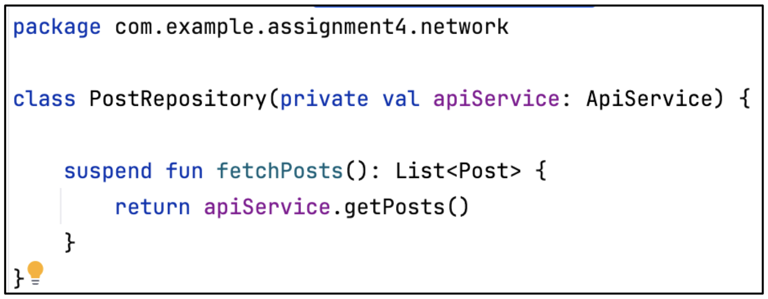


```kotlin
package com.example.assignment4.network

data class Post(
    val userId: Int,
    val id: Int,
    val title: String,
    val body: String
)
```

Figure 10. The Post.kt file

userId: Represents the ID of the user who created the post. id: The unique ID of the post. title: The title of the post. body: The content or body of the post. The use of a data class provides built-in functions like toString, equals, and hashCode, making it easier to work with post objects in the app. The Post class provides a structured way to handle data from the API. Each Post object represents a single post retrieved from the server.

Then I created the PostRepository.kt file. The file is shown in Figure 11.

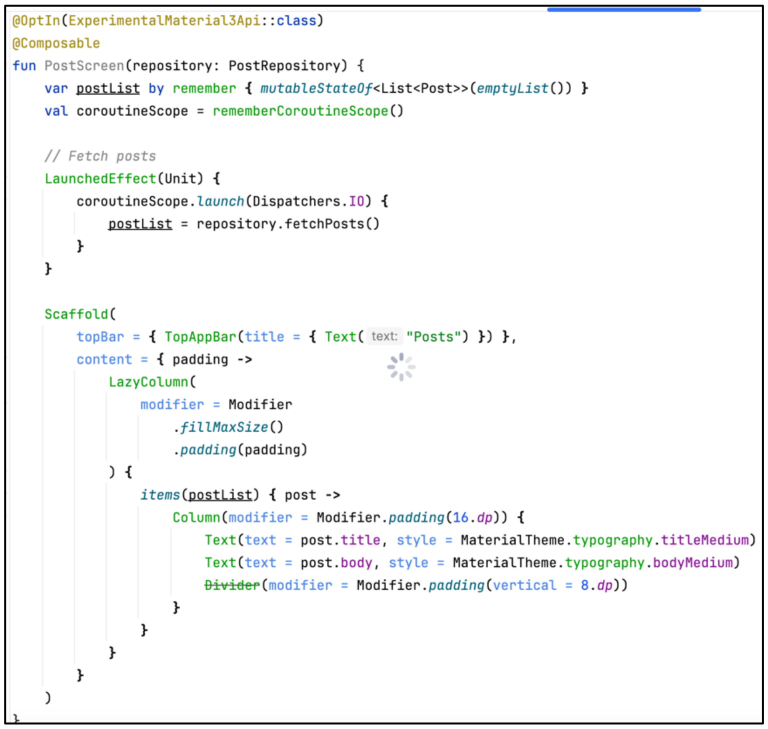


```kotlin
package com.example.assignment4.network

class PostRepository(private val apiService: ApiService) {

    suspend fun fetchPosts(): List<Post> {
        return apiService.getPosts()
    }
}
```

Figure 11. The PostRepository.kt file

The PostRepository class acts as a middle layer between the ApiService and the rest of the app. The PostRepository takes an instance of ApiService as a parameter. This allows it to use the API methods defined in ApiService.

Then, in order to update the UI I created the PostScreen.kt file. The file is shown in Figure 12.



```kotlin
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun PostScreen(repository: PostRepository) {
    var postList by remember { mutableStateOf<List<Post>>(emptyList()) }
    val coroutineScope = rememberCoroutineScope()

    // Fetch posts
    LaunchedEffect(Unit) {
        coroutineScope.launch(Dispatchers.IO) {
            postList = repository.fetchPosts()
        }
    }

    Scaffold(
        topBar = { TopAppBar(title = { Text( text: "Posts") }) },
        content = { padding ->
            LazyColumn(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(padding)
            ) {
                items(postList) { post ->
                    Column(modifier = Modifier.padding(16.dp)) {
                        Text(text = post.title, style = MaterialTheme.typography.titleMedium)
                        Text(text = post.body, style = MaterialTheme.typography.bodyMedium)
                        Divider(modifier = Modifier.padding(vertical = 8.dp))
                    }
                }
            }
        }
    )
}
```

Figure 12. The PostScren.kt file

The PostScreen composable shows a simple and clear interface for viewing posts. It fetches data smoothly and updates the list automatically, making it easy for users to see content from the API.

Conclusion

In conclusion, this project helped me learn how to build a complete Android app by combining different tools. With Room, I learned how to save and manage data locally so that users can access their data even without the internet. I created models, set up a DAO to handle database operations, and used a repository to make things easier and more organized. This showed me how to store and retrieve data effectively.

Using Retrofit, I connected the app to a remote API to fetch data like posts. Even though I didn’t work directly with JSON, Retrofit handled it for me, which made everything simpler. I learned how to set up API endpoints and use a repository to manage API calls. This showed me how to get data from the internet and display it in the app, which is a key feature for many apps.

Jetpack Compose made designing the user interface easier and more modern. I used tools like LazyColumn to display lists of users and posts and added buttons and text fields for tasks like creating and deleting users. The app’s interface updates automatically when the data changes, making it easy to use.

This assignment taught me how to manage both local and remote data in an Android app and how to design a simple and interactive interface. It also helped me get better at using Room, Retrofit, Jetpack Compose, repositories, and coroutines. Overall, it gave me great experience and prepared me to work on more advanced Android projects in the future.

References

1. Save data in a local database using Room [Developers] https://developer.android.com/training/data-storage/room
2. How the Retrofit 2 library solves network translation difficulties [Yandex Practicum] https://practicum.yandex.ru/blog/retrofit-na-android/
3. Accessing data using Room DAOs [Developers] https://developer.android.com/training/data-storage/room/accessing-data