Kazakh British Technical University

Mobile Programming

Assignment 4

by Moldir Polat

December 1st 2024

Table of contents

Introduction

Mobile applications have become an integral part of our daily lives, offering a wide range of functionalities that make our tasks easier and more efficient. Behind every feature-rich app lies a solid foundation of data management and easy communication with online services. Databases and RESTful API integration play a crucial role in ensuring these apps function effectively.

Local databases allow mobile applications to store, retrieve, and manage data efficiently on the user's device. This ensures that the app can provide a smooth user experience even without an internet connection. For instance, apps that store user profiles, shopping lists, or offline messages rely on robust local database solutions to manage their data. In Android development, the Room persistence library simplifies the process of setting up and managing local databases while offering features like compile-time verification of SQL queries, easy integration with LiveData, and support for Kotlin coroutines.

On the other hand, RESTful APIs enable mobile applications to interact with remote servers, facilitating tasks such as fetching real-time weather updates, retrieving user data from cloud storage, or posting updates on social media. The Retrofit library is a popular choice for implementing API calls in Android applications because it simplifies HTTP communication, supports JSON parsing, and integrates well with Kotlin coroutines for asynchronous operations.

The purpose of this report is to explore the practical implementation of Room and Retrofit in Android applications. It aims to demonstrate how these tools can be used to build a reliable local database and integrate RESTful APIs for remote data communication. By understanding the significance of these technologies, developers can create apps that are both user-friendly and feature-rich, ensuring a good experience for their users. This report covers the step-by-step tasks for working with Room and Retrofit, showcasing their importance in modern mobile programming.

Working with Databases in Kotlin Android

Exercise 1

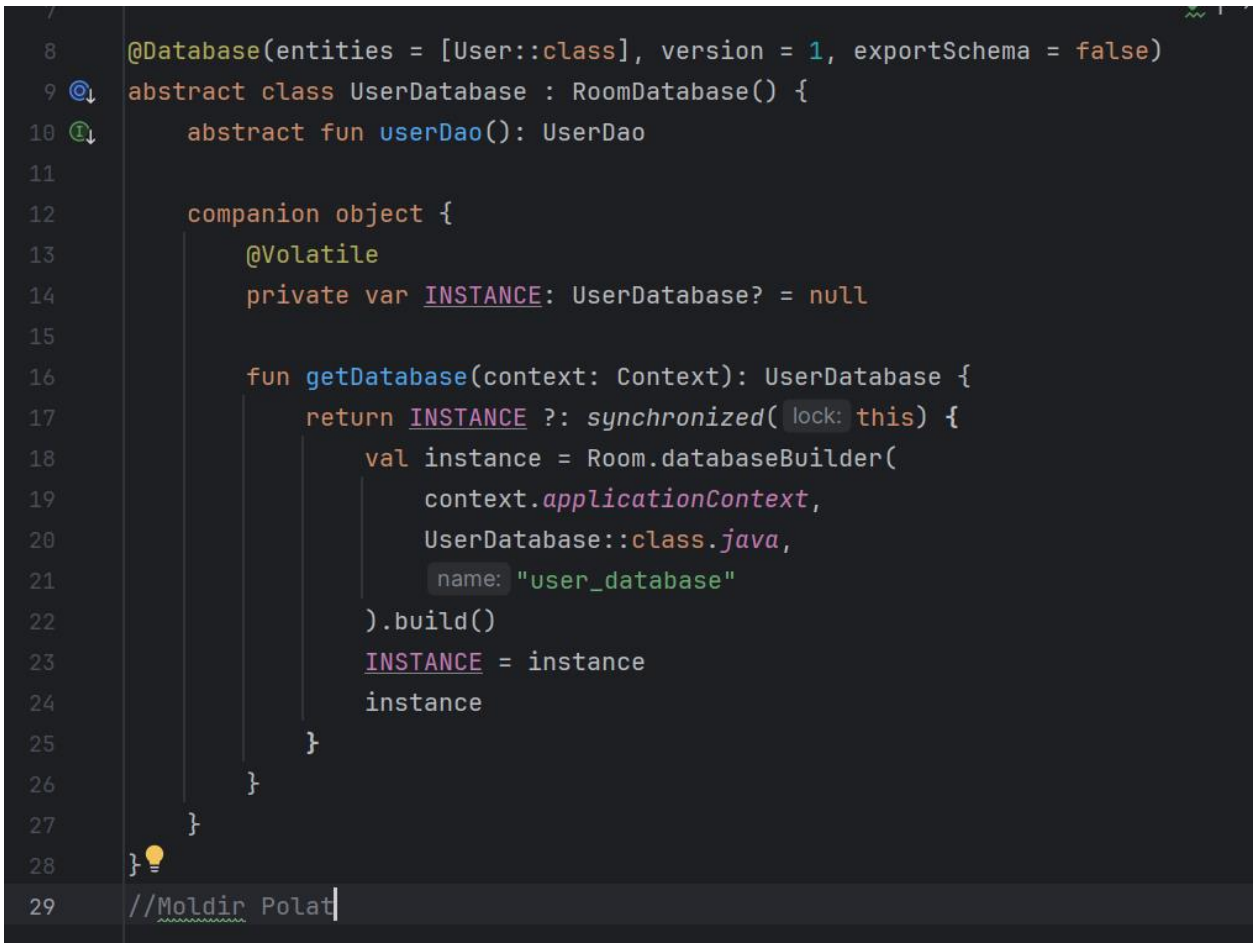
Room is an abstraction layer over SQLite designed to simplify database interactions in Android applications. It provides compile-time verification of SQL queries, integration with LiveData, and support for Kotlin coroutines, making it an excellent choice for modern Android development. Unlike SQLite, Room reduces boilerplate code, improves readability, and enhances reliability by enforcing type safety. In this project, I used Room to implement a database for managing user data in a Kotlin Android application, complemented by Jetpack Compose for the user interface.

To begin with, I defined the data model and data access object (DAO). The data model, User, represents the schema of the database table. It is annotated with @Entity, with the id field marked as the primary key using @PrimaryKey. Additionally, the name and email fields define the structure of the user data stored in the database. The DAO, UserDao, contains methods for database operations. These methods are annotated with @Insert and @Query to define insertion and retrieval of user data, respectively. Room processes these annotations at compile time, generating the necessary database implementation.

```
4   //Moldir Polat
5   @Entity(tableName = "user_table")
6   data class User(
7       @PrimaryKey(autoGenerate = true) val id: Int = 0,
8       val name: String,
9       val email: String
10  )
```

```
7   //Moldir Polat
8   @Dao
9   interface UserDao {
10      @Insert
11      suspend fun insert(user: User)
12
13      @Query("SELECT * FROM user_table")
14      fun getAllUsers(): LiveData<List<User>>
15  }
16
```

After defining the data model and DAO, I set up the Room database. The UserDatabase class extends RoomDatabase and is annotated with @Database. It specifies the entities used in the database and its version. To ensure that only one instance of the database is used throughout the app, I implemented the singleton pattern. The Room.databaseBuilder method initializes the database, linking it to the application context and specifying its name. This setup prevents resource leaks and ensures consistency in database operations.



```
@Database(entities = [User::class], version = 1, exportSchema = false)
abstract class UserDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao

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

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

To maintain clean architecture, I introduced a repository pattern. The UserRepository class abstracts the data access logic from the rest of the app. It holds a reference to the DAO and exposes LiveData for observing user data. The repository encapsulates database operations, ensuring that the ViewModel and UI layers interact with the database through a consistent and testable API.

```
//Moldir Polat
class UserRepository(private val userDao: UserDao) {
    val allUsers: LiveData<List<User>> = userDao.getAllUsers()

    suspend fun insert(user: User) {
        userDao.insert(user)
    }
}
```

I used Jetpack Compose to create a form with OutlinedTextField for name and email input and a Button to submit data. The UserForm composable triggers the onSubmit callback with the entered data.

```
@Composable
fun UserForm(onSubmit: (String, String) -> Unit) {
    // Mutable states for name and email input
    var name by remember { mutableStateOf( value: "") }
    var email by remember { mutableStateOf( value: "") }

    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(16.dp)
    ) {...}
}
//Moldir Polat
```

I used Compose's LazyColumn to display the list of users retrieved from the database. Each user is displayed in a UserItem composable showing their name and email. I designed a UserForm composable to collect user inputs and trigger the insertion of new users into the database. Then used LazyColumn to dynamically display a list of users stored in the database.

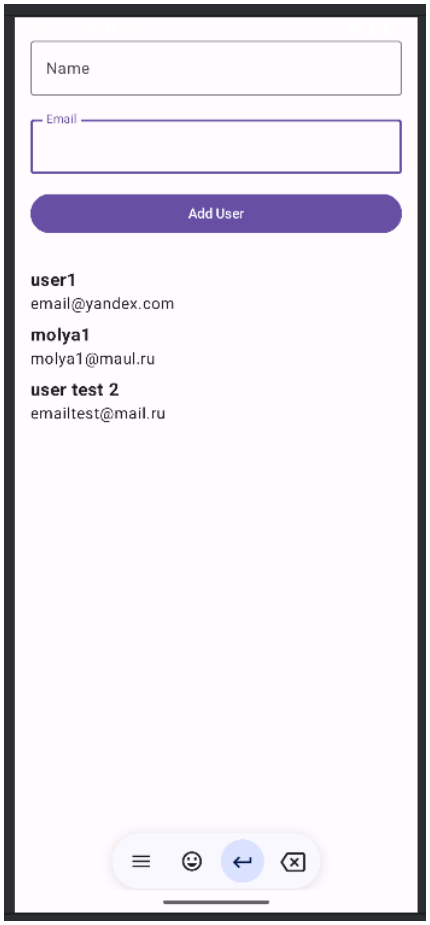
```
//Moldir Polat
@Composable
fun UserList(users: List<User>) {
    LazyColumn(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp)
    ) {
        items(users) { user ->
            UserItem(user = user)
            Spacer(modifier = Modifier.height(8.dp))
        }
    }
}
```

```
//Moldir Polat
@Composable
fun UserItem(user: User) {
    Column(modifier = Modifier.fillMaxWidth()) {
        Text(
            text = user.name,
            fontSize = 18.sp,
            fontWeight = FontWeight.Bold
        )
        Text(
            text = user.email,
            fontSize = 16.sp
        )
    }
}
```

I observed changes in the database with LiveData and integrated it with Compose using observeAsState. I implemented UserViewModel to handle business logic and provide data to the UI. The viewModelScope was used to perform database operations on a background thread.

```
8   //Moldir Polat
9   class UserViewModel(application: Application) : AndroidViewModel(application) {
10      private val repository: UserRepository
11      val allUsers: LiveData<List<User>>
12
13      init {
14          val userDao = UserDatabase.getDatabase(application).userDao()
15          repository = UserRepository(userDao)
16          allUsers = repository.allUsers
17      }
18
19      fun insertUser(name: String, email: String) {
20          viewModelScope.launch {
21              val newUser = User(name = name, email = email)
22              repository.insert(newUser)
23          }
24      }
25  }
```

Simple screen with form for adding new user and the list of all users was created and displayed below. After restarting the application all data remains in the screen meaning that the exercise was accomplished.



Using Retrofit in Kotlin Android

Exercise 2

Overview of Retrofit. Retrofit is a popular type-safe HTTP client for Android and Java developed by Square. It simplifies API calls by providing an easy-to-use, annotation-based approach for defining RESTful services. Retrofit integrates easily with popular serialization libraries like Gson, enabling effortless parsing of JSON responses into Kotlin data classes. One of its key benefits is the ability to handle both synchronous and asynchronous requests with minimal boilerplate code. Additionally, Retrofit supports features like dynamic URL endpoints, request headers, and query parameters, making it highly flexible for complex API interactions.

To set up Retrofit, I first added the necessary dependencies for Retrofit and Gson to the project. This ensured that the project could serialize and deserialize JSON responses easily. I then created a singleton object to initialize Retrofit. This singleton serves as the centralized client for all API requests, ensuring efficient and consistent configuration across the app. I configured the BASE_URL and enabled logging for debugging API requests and responses. The OkHttpClient was set up with a logging interceptor to monitor HTTP traffic during development.

```
object RetrofitClient {

    private val loggingInterceptor = HttpLoggingInterceptor().apply {
        level = HttpLoggingInterceptor.Level.BODY
    }

    private val okHttpClient = OkHttpClient.Builder()
        .addInterceptor(loggingInterceptor)
        .build()

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

API Service Definition. The next step was to define the API service interface. This interface outlines the endpoints of the API and the expected HTTP methods. I used Retrofit's annotations, such as @GET and @Path, to specify the endpoints and request parameters.

```kotlin
//Moldir Polat
interface ApiService {
    @GET("users")
    suspend fun getUsers(): Response<List<User>>

    @GET("users/{id}")
    suspend fun getUserDetails(@Path("id") id: Int): Response<User>
}

sealed class ApiResponse<out T> {
    data class Success<out T>(val data: T) : ApiResponse<T>()
    data class Error(val message: String) : ApiResponse<Nothing>()
    object Loading : ApiResponse<Nothing>()
}
```

The getUsers method fetches a list of users, while getUserDetails retrieves details for a specific user identified by their id. By leveraging Retrofit's type-safe API definitions, I ensured that the Kotlin compiler would catch any discrepancies between the API contract and the client code.

Data Models. For handling API responses, I reused the User data model from the Room database implementation. This integration demonstrated how the same data model could be used both for persistent storage and for representing API responses. The User class was annotated with @SerializedName to map JSON keys to Kotlin properties when necessary.

```kotlin
@Entity(tableName = "user_table")
data class User(
    @PrimaryKey(autoGenerate = true)
    @SerializedName("id") val id: Int = 0,
    @SerializedName("name") val name: String,
    @SerializedName("email") val email: String
)
//Moldir Polat
```

API Calls and Response Handling. To manage API calls and process responses, I implemented a repository and a ViewModel. The repository handles the interaction with the ApiService, while the ViewModel exposes the processed data to the UI layer.

In the repository, I defined methods to fetch data from the API. For instance, the getUsersFromApi method calls the getUsers endpoint and returns the API response. The repository acts as a mediator, ensuring the API details are abstracted from the ViewModel.

```kotlin
//Moldir Polat

class UserRepository(private val apiService: ApiService) {

    suspend fun getUsersFromApi(): Response<List<User>> {
        return apiService.getUsers()
    }

    suspend fun getUserDetailsFromApi(id: Int): Response<User> {
        return apiService.getUserDetails(id)
    }
}
```

In the ViewModel, I used viewModelScope to make API calls asynchronously and handle the responses. I also incorporated LiveData to observe changes in the data and update the UI dynamically. This code ensures that the API calls do not block the main thread, maintaining a responsive user interface. Errors are handled gracefully and communicated to the UI via LiveData.

```
//Moldir Polat
class UserViewModel(private val repository: UserRepository) : ViewModel() {

    private val _users = MutableLiveData<List<User>>()
    val users: LiveData<List<User>> get() = _users

    private val _error = MutableLiveData<String>()
    val error: LiveData<String> get() = _error

    fun fetchUsers() {
        viewModelScope.launch {
            try {
                val response = repository.getUsersFromApi()
                if (response.isSuccessful) {
                    _users.postValue(response.body())
                } else {
                    _error.postValue( value: "Error: ${response.code()}")
                }
            } catch (e: Exception) {
                _error.postValue( value: "Exception: ${e.message}")
            }
        }
    }
```

Caching Responses. To enhance the app's performance and provide offline functionality, I implemented caching using Room. The API responses were stored in the Room database, and the app would fetch data from the database if the network was unavailable.

In the repository, I added a method to save API responses to the Room database.

```
//Moldir Polat

class UserRepository(
    private val apiService: ApiService,
    private val userDao: UserDao
) {
    val cachedUsers: LiveData<List<User>> = userDao.getAllUsers()

    suspend fun fetchAndCacheUsers() {
        val response = apiService.getUsers()
        if (response.isSuccessful) {
            response.body()?.let { users ->
                userDao.insertAll(users)
            }
        }
    }
}
```

The DAO was updated to support bulk insertion of user data with conflict resolution.

With this caching mechanism, the app provided a smooth experience by displaying cached data when offline and updating the database when new data was fetched from the API.

Recommendations

We can improve database handling by using migrations to handle changes to the database structure when the app is updated. This will ensure that the app works smoothly even after updates. To make the database faster, we can add indexing for fields that are often searched. Room's type converters can also help us store custom data types more easily.

To make API integration more reliable, we can add a retry mechanism for network requests. This will make the app work better during internet issues. Better error handling is also important, so we can show clear messages to users if something goes wrong. Using API versioning will help the app stay compatible with future changes to the server.

Finally, we can combine caching strategies to make the app work well offline. For example, we can use Room for long-term data storage and SharedPreferences for simple settings or preferences. Adding automatic synchronization between the app and the server will ensure that data stays updated without the user needing to do anything. These changes will make the app faster and easier to use.

Conclusion

In this project, I explored two important aspects of modern Android application development: managing local databases with Room and integrating APIs using Retrofit. Both exercises provided a comprehensive understanding of how to efficiently handle data in Kotlin Android applications, ensuring responsiveness, scalability, and a good user experience.

The first exercise focused on database management using Room. Room simplified working with SQLite by providing an abstraction layer that was easy to implement and less prone to errors. By defining entities and data access objects (DAOs), I could efficiently store, retrieve, and manage data. Integrating Room with LiveData and ViewModel ensured that the UI dynamically updated whenever the database changed. This lifecycle-aware design ensured the application remained responsive and adhered to modern best practices. Additionally, leveraging coroutines made asynchronous database operations straightforward, avoiding UI thread blocking.

The second exercise revolved around API integration using Retrofit. Retrofit proved to be a powerful and flexible library for making network requests. Its annotation-based approach made defining endpoints intuitive and minimized boilerplate code. By using Gson, I could easily map JSON responses to Kotlin data models, enabling easy communication with the API. The integration of Retrofit with ViewModel and LiveData ensured that the application maintained a clean architecture. Additionally, implementing error handling and caching enhanced the app's reliability and user experience, even in offline scenarios. Storing API responses in Room allowed for consistent data representation and ensured the app performed well without constant network connectivity.

Together, these exercises demonstrated the importance of combining robust local database management with efficient API integration in Android development. Room provided a solid foundation for data persistence, while Retrofit enabled smooth interaction with external services. By following best practices like using repositories, LiveData, and coroutines, the application design remained clean and maintainable. These tools and approaches form the backbone of any modern, user-friendly Android application and are crucial for building scalable and reliable apps in the future.

References

https://developer.android.com/training/data-storage/room

https://developer.android.com/topic/libraries/architecture/viewmodel

https://square.github.io/retrofit/

https://square.github.io/okhttp/

https://developer.android.com/jetpack/compose

https://kotlinlang.org/docs/coroutines-overview.html

https://developer.android.com/topic/libraries/architecture/livedata

https://chatgpt.com/ (used for issue resolution)

https://stackoverflow.com/