KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Prepared by: Serzhan Yerasil

Assignment 4

Mobile Programming

30.11.2024

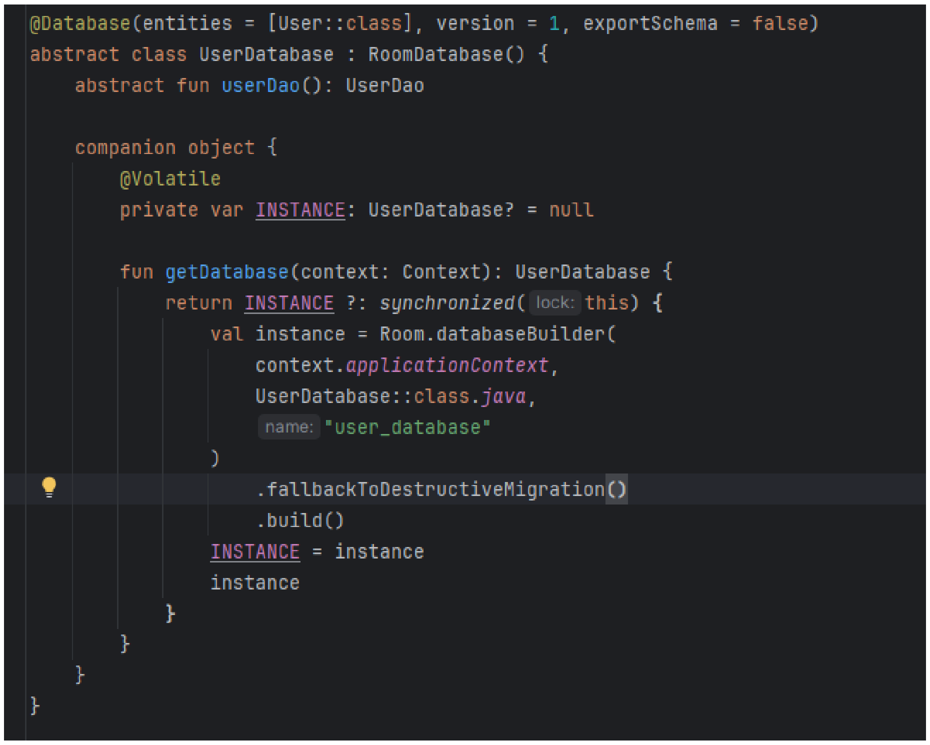
Table of Contents

Introduction

This report focuses on two key technologies used in Kotlin Android development: Room for local databases and Retrofit for API integration. The purpose of this report is to explore how these technologies can be used together to build Android applications. Local databases allow apps to store and manage user data efficiently, while APIs enable communication with external servers, providing access to dynamic content and services.

Overview of Room Database

Room is an SQLite abstraction library for Android that simplifies database management by providing an object-oriented interface to interact with local databases. Unlike traditional SQLite, Room uses annotations and data classes to define entities and queries, making it easier to manage complex database operations. It eliminates the need for boilerplate code while ensuring compatibility with SQLite. Room also provides compile-time verification of SQL queries, improving safety and reducing runtime errors.

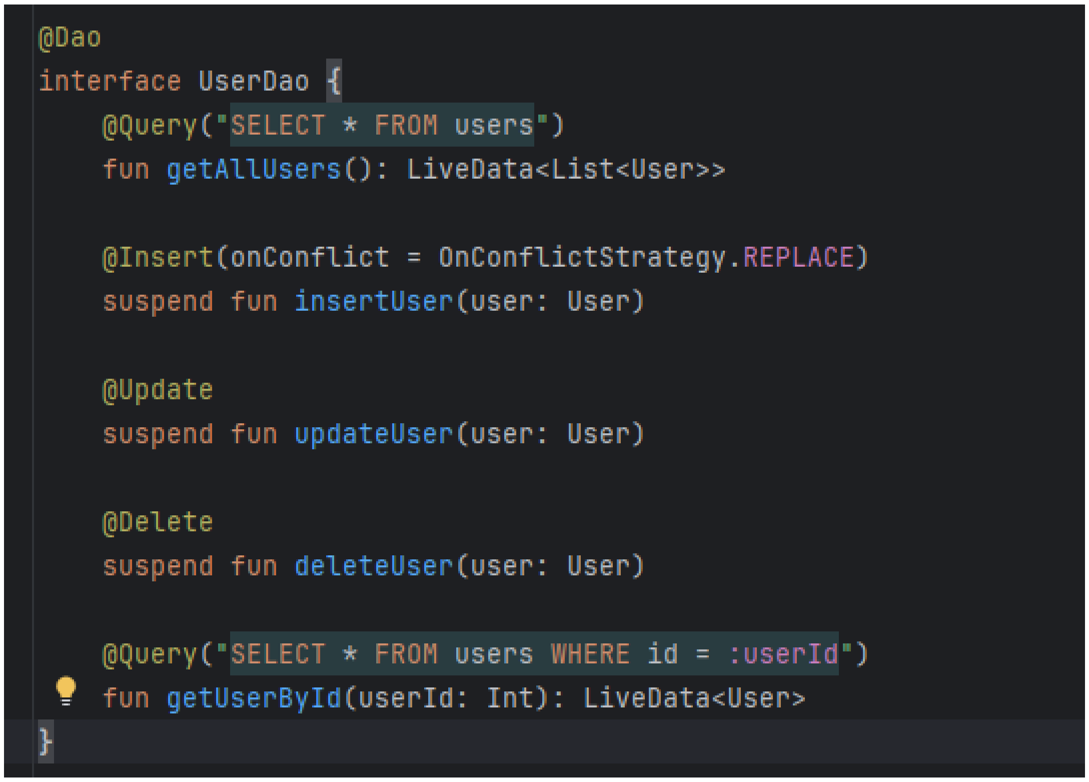


```kotlin
@Database(entities = [User::class], version = 1, exportSchema = false)
abstract class UserDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao

    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"
                )
                    .fallbackToDestructiveMigration()
                    .build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

This code creates a database using Room to store user data. The UserDatabase class inherits from RoomDatabase and contains an abstract method userDao() that provides access to database operations through the DAO interface. Inside the companion, the singleton pattern is implemented to ensure a single instance of the database in the application. The getDatabase method creates the database if it has not yet been created and returns an instance of it. Using fallbackToDestructiveMigration() avoids errors when changing the database schema during development by automatically deleting the old database during migration.

Data Models and DAO

In Room, data models are represented by Kotlin data classes annotated with Entity, which define the structure of the database table. Each class typically has properties such as PrimaryKey and ColumnInfo to define the table's primary key and column names.A DAO is an interface where you define methods to interact with the database. Methods such as Insert, Update, Delete, and Query allow for common database operations

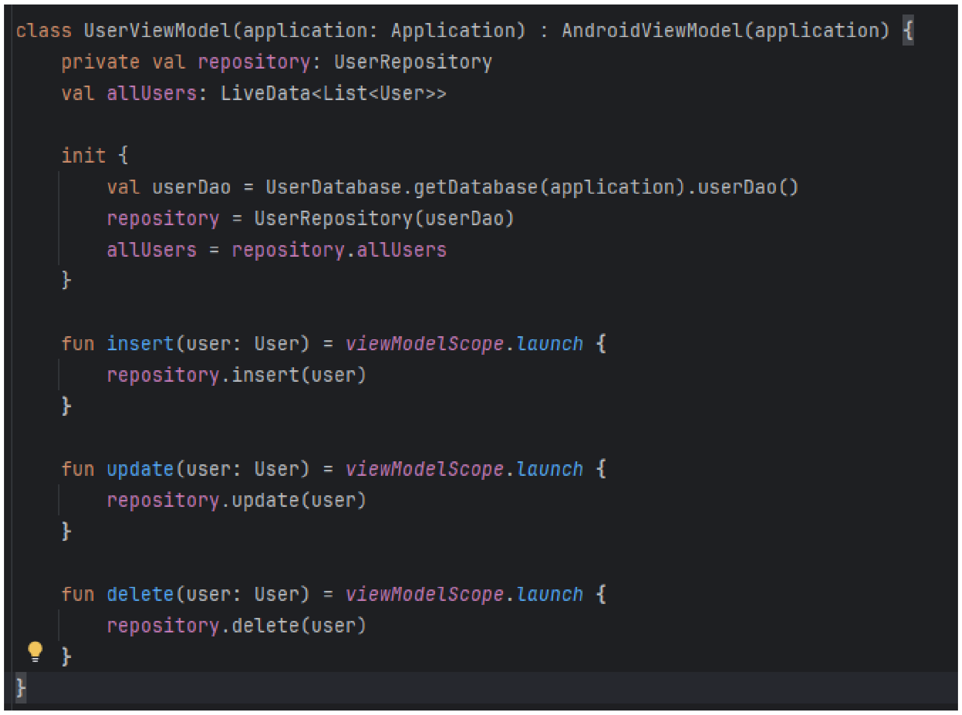


```kotlin
@Dao
interface UserDao {
    @Query("SELECT * FROM users")
    fun getAllUsers(): LiveData<List<User>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUser(user: User)

    @Update
    suspend fun updateUser(user: User)

    @Delete
    suspend fun deleteUser(user: User)

    @Query("SELECT * FROM users WHERE id = :userId")
    fun getUserById(userId: Int): LiveData<User>
}
```

This UserDao interface defines methods for interacting with the users table in the database. The getAllUsers() method returns all users as LiveData, allowing you to monitor data changes in real time. To add, update, and delete users, methods with @Insert, @Update, and @Delete annotations are used, which are executed asynchronously using suspend functions. The getUserById() method allows you to get a user by their identifier and also returns the result as LiveData, allowing you to monitor changes to a specific user's data.

User Interface Integration

To display data in the UI, a UserViewModel is used, which efficiently handles large datasets. A LiveData object is tied to the database query, allowing the UI to automatically update when the data changes. The ViewModel holds the LiveData, ensuring that UI components are lifecycle-aware and do not access the database directly.

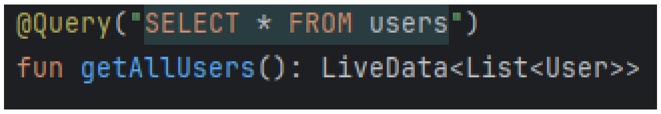


```kotlin
class UserViewModel(application: Application) : AndroidViewModel(application) {
    private val repository: UserRepository
    val allUsers: LiveData<List<User>>

    init {
        val userDao = UserDatabase.getDatabase(application).userDao()
        repository = UserRepository(userDao)
        allUsers = repository.allUsers
    }

    fun insert(user: User) = viewModelScope.launch {
        repository.insert(user)
    }

    fun update(user: User) = viewModelScope.launch {
        repository.update(user)
    }

    fun delete(user: User) = viewModelScope.launch {
        repository.delete(user)
    }
}
```

This UserViewModel class manages user data and acts as a link between the UI and the repository. It initializes the repository, which accesses data through the UserDao obtained from the database. The allUsers variable stores LiveData, which tracks all changes to the user list through the repository. The insert, update, and delete methods perform data operations asynchronously using coroutines, which allows updating the database without blocking the main thread. This approach helps keep data current and improves the performance of the application.

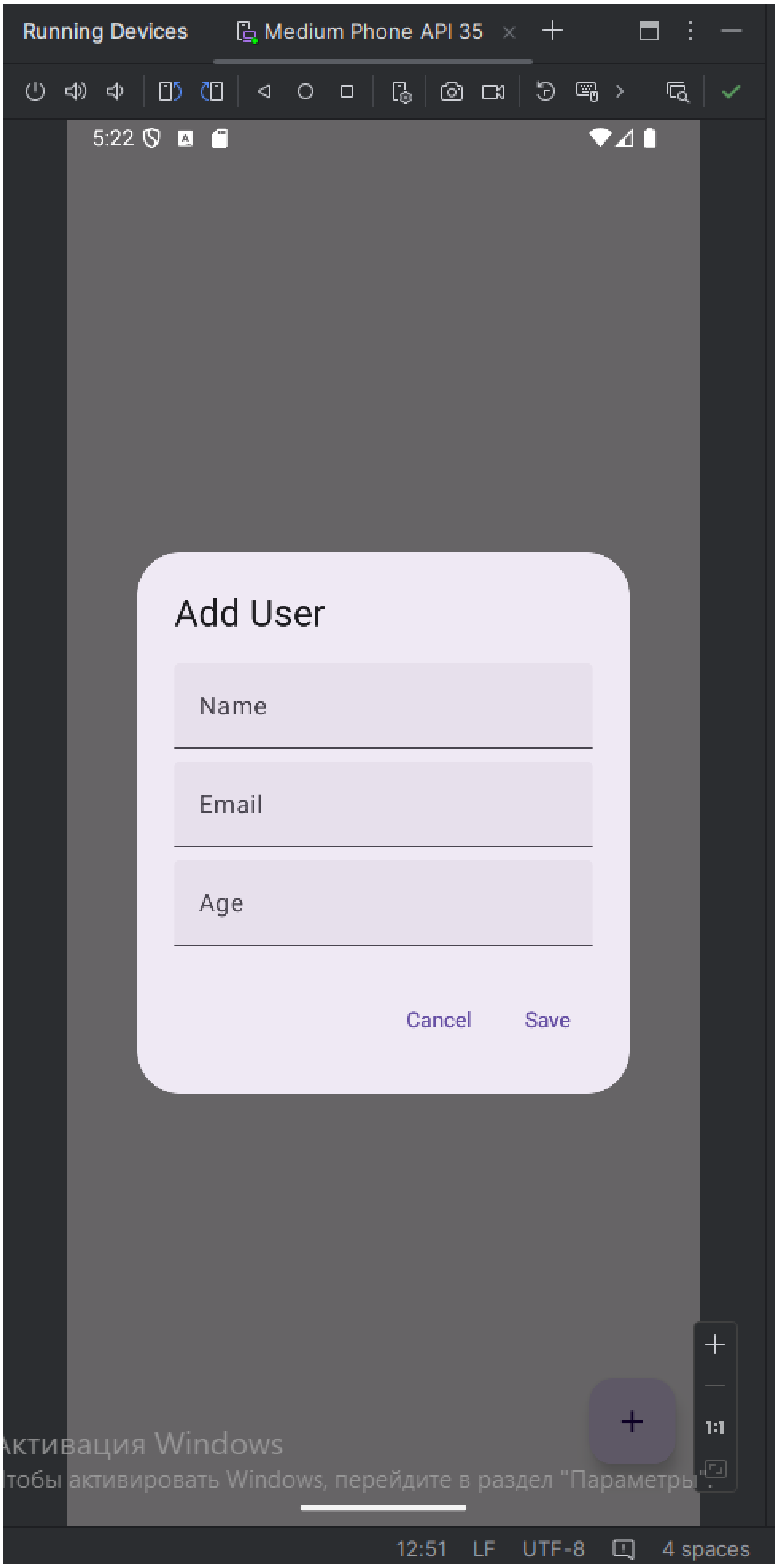
Lifecycle Awareness

LiveData is a key component for lifecycle aware data handling in Android. It ensures that the UI only updates when the activity or fragment is in an active state, preventing memory leaks. Coroutines are used to handle database operations asynchronously, which provides long-running operations which is not block main thread.

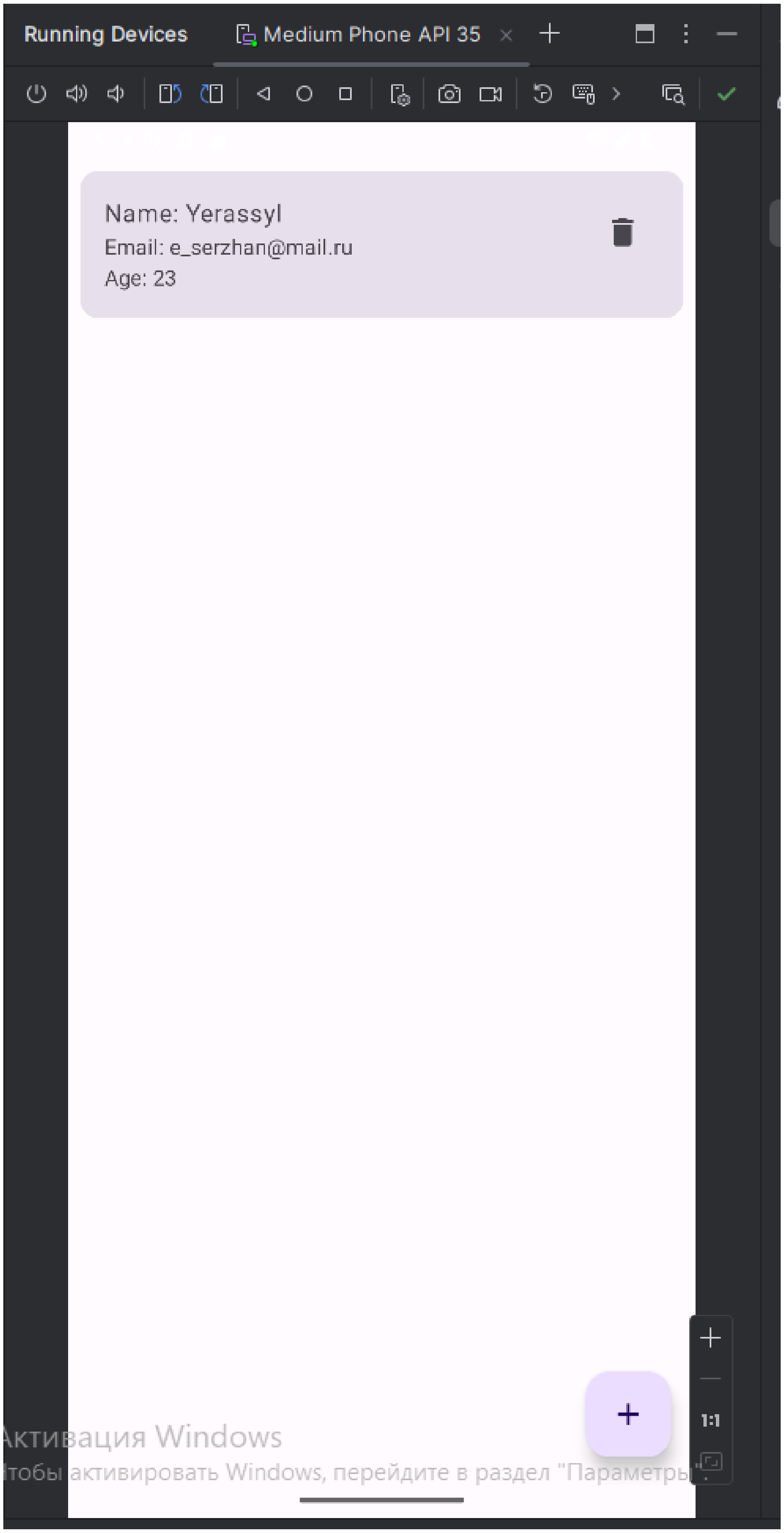


```
@Query("SELECT * FROM users")
fun getAllUsers(): LiveData<List<User>>
```

The return value is a LiveData<List<User>>, which allows you to monitor changes in the database and automatically update the UI when the data changes. Using LiveData ensures that the data is only updated when the corresponding UI component, such as an Activity or Fragment, is active, thus providing efficient and reactive work with data.

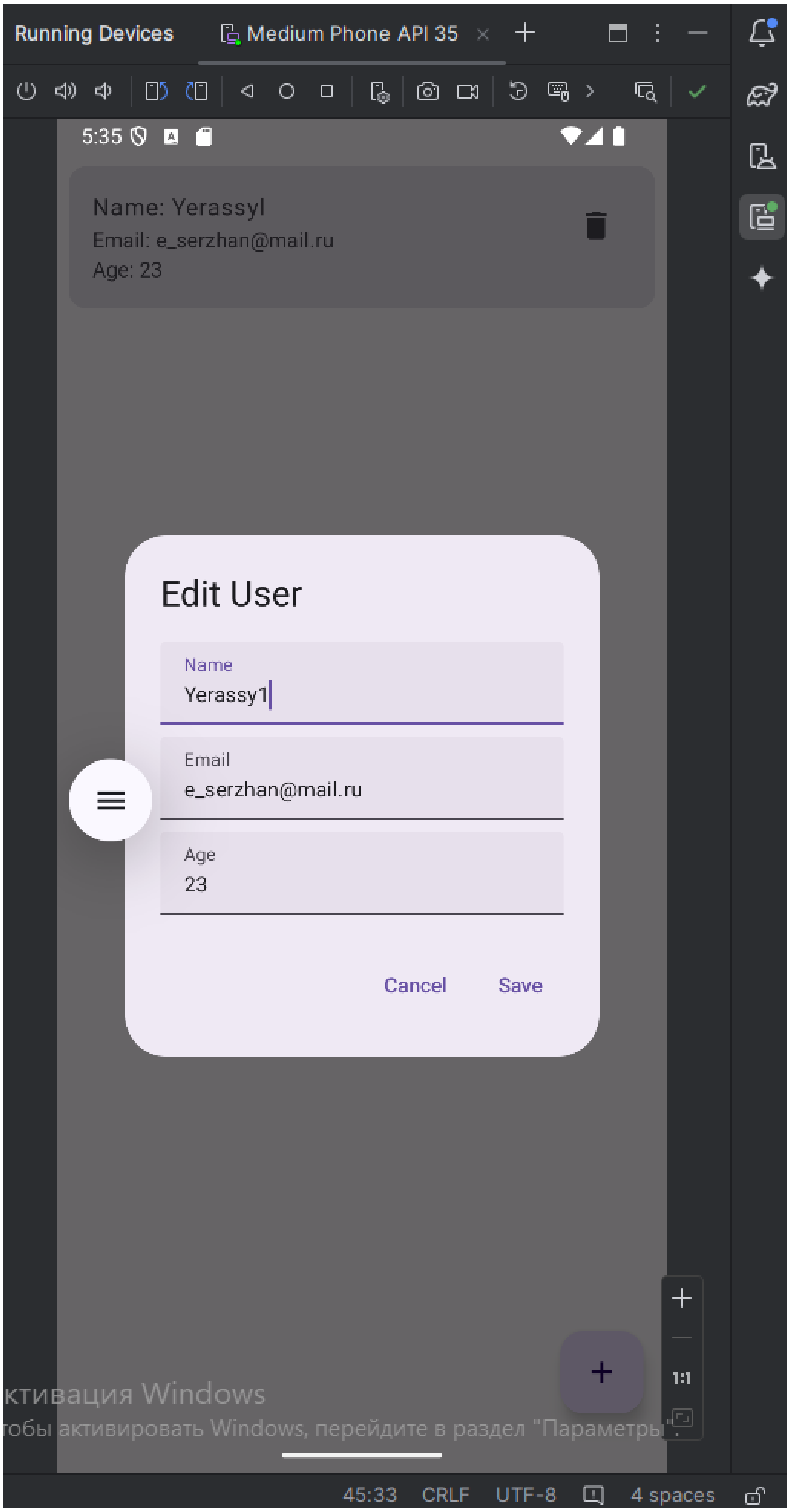


Adding new user



Name: Yerassyl
Email: e_serzhan@mail.ru
Age: 23

New user added



Name: Yerassyl
Email: e_serzhan@mail.ru
Age: 23

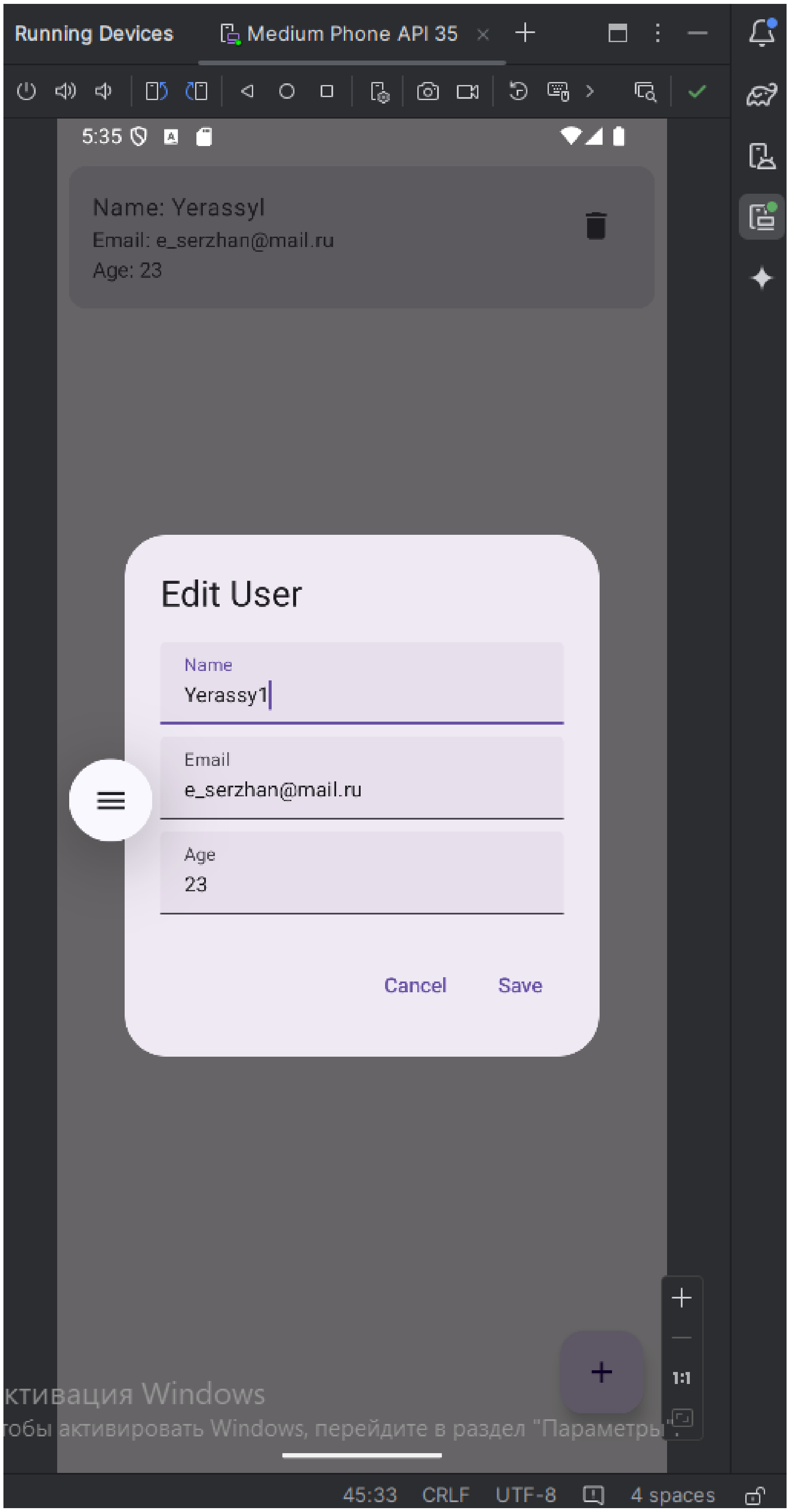
Edit User

Name
Yerassy1

Email
e_serzhan@mail.ru

Age
23

Update existing user



Name: Yerassyl
Email: e_serzhan@mail.ru
Age: 23

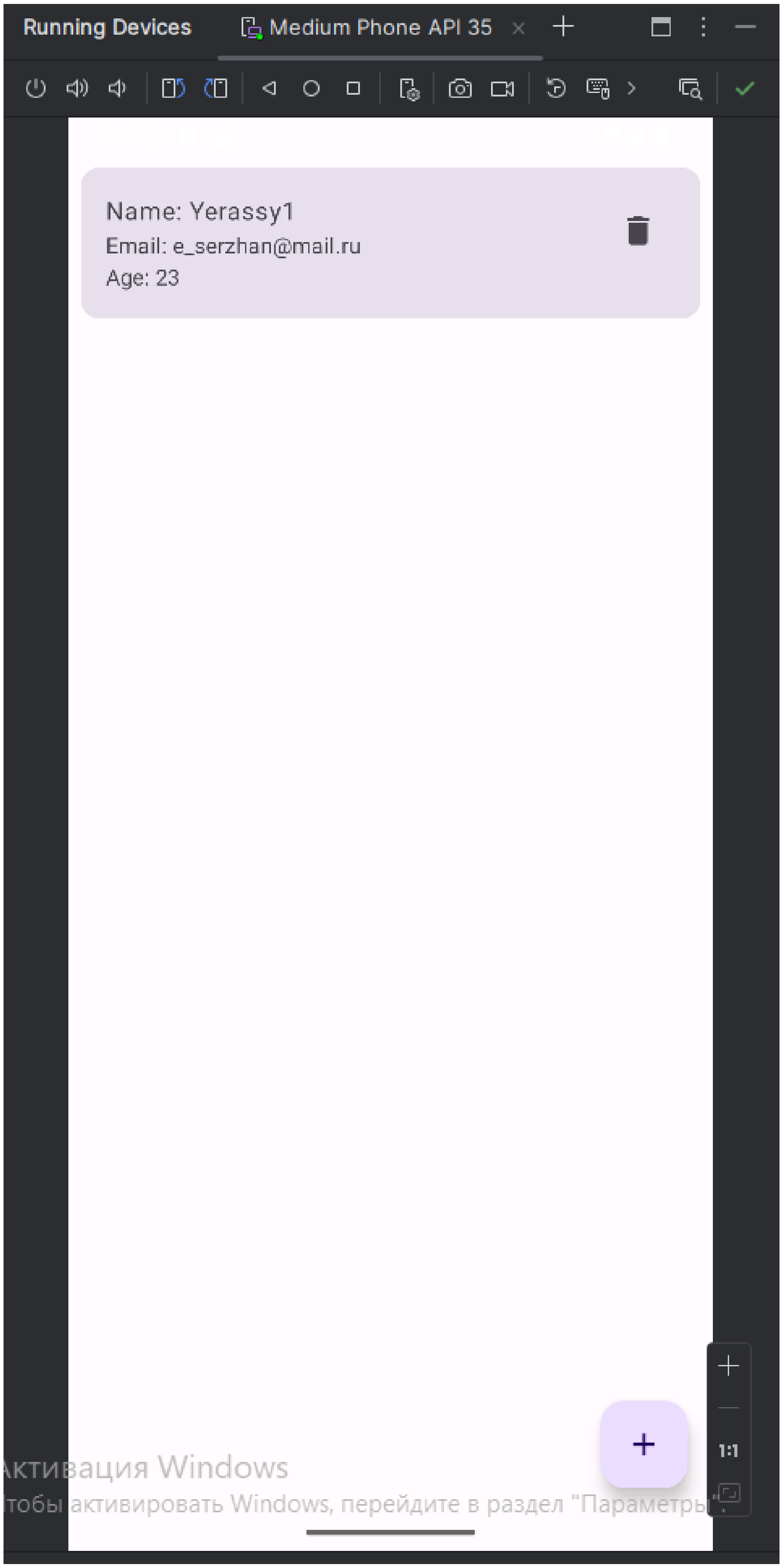
Edit User

Name
Yerassy1

Email
e_serzhan@mail.ru

Age
23

Update existing user



Name: Yerassy1
Email: e_serzhan@mail.ru
Age: 23

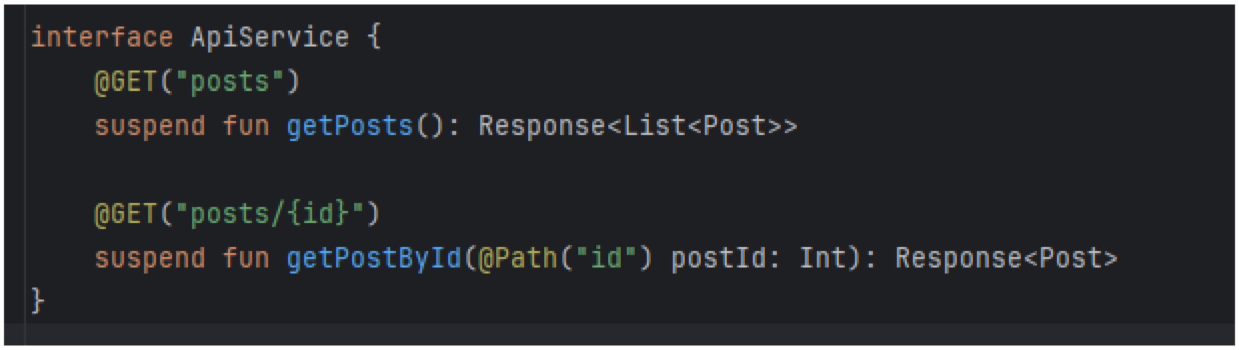
Deleting user

Overview of Retrofit

Retrofit is a type-safe HTTP client for Android, used for making API requests to rest web services. It abstracts away the complexities of making network calls and converting responses into usable data formats. Retrofit supports synchronous and asynchronous operations and integrates well with Gson for JSON parsing. This makes it ideal for integrating APIs into Android apps with minimal setup.

API Service Definition

In Retrofit, an API service interface defines the API endpoints and their corresponding HTTP methods using annotations such as GET, POST, PUT, and DELETE. Each endpoint is mapped to a method in the interface, and parameters are passed using annotations like Query or Body.

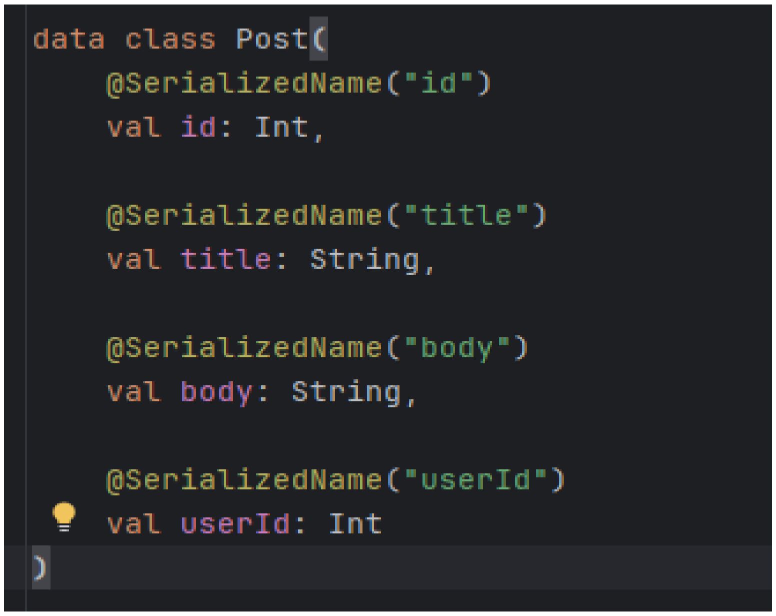


```kotlin
interface ApiService {
    @GET("posts")
    suspend fun getPosts(): Response<List<Post>>

    @GET("posts/{id}")
    suspend fun getPostById(@Path("id") postId: Int): Response<Post>
}
```

This ApiService interface defines two methods for interacting with the REST API using Retrofit. The getPosts() method makes an HTTP GET request to get a list of all posts, returning the result as a Response<List<Post>>. The getPostById() method makes a GET request to get a post by its id, where @Path("id") specifies that the postId parameter will be substituted into the URL instead of {id}. Both methods are marked as suspend, meaning they are executed in an asynchronous context, using coroutines for non-blocking calls.

Data Models

Data models are Kotlin data classes that represent the structure of the JSON response from the API. Each field in the data model corresponds to a key in the JSON response, and Gson annotations are used to handle any discrepancies between the model and the JSON structure.

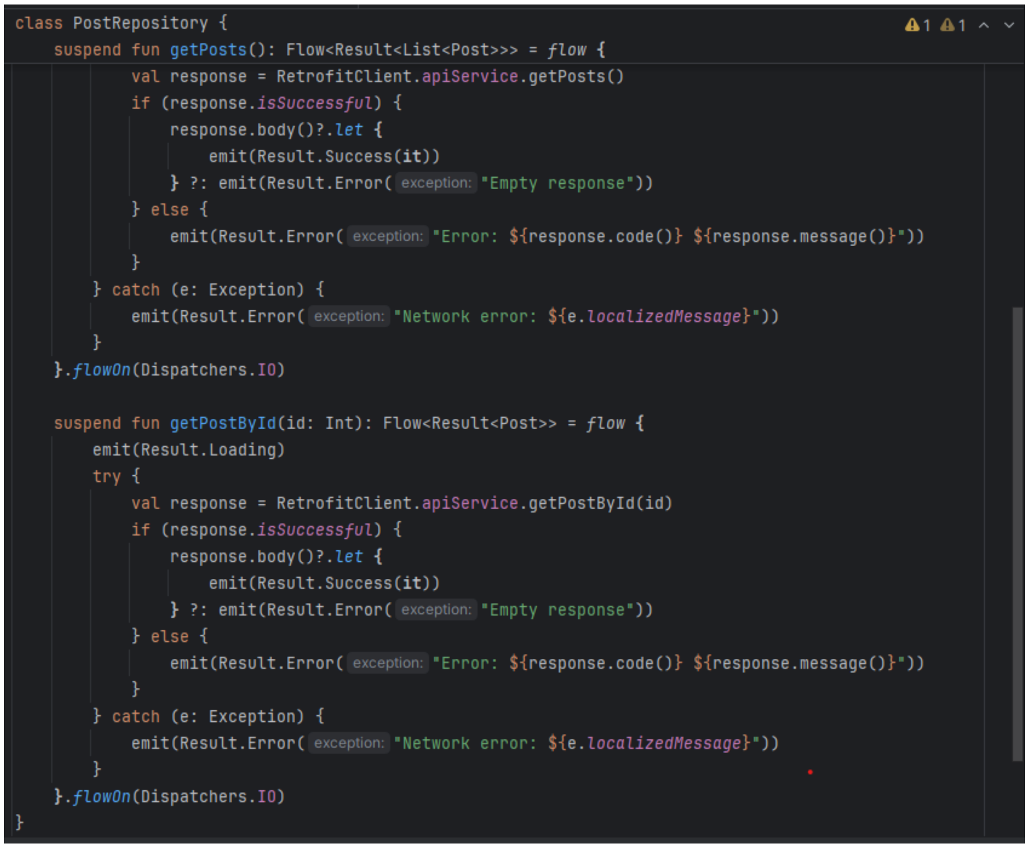


```kotlin
data class Post(
    @SerializedName("id")
    val id: Int,

    @SerializedName("title")
    val title: String,

    @SerializedName("body")
    val body: String,

    @SerializedName("userId")
    val userId: Int
)
```

This Post class represents the data retrieved from the API and is used to serialize/deserialize JSON responses using the Gson library. Each class parameter is annotated with @SerializedName to specify how the JSON fields map to the Kotlin class properties. For example, the "id" field in the JSON will map to the id field in the class, and "title", "body", and "userId" will map to the corresponding properties of the Post class. This allows Retrofit and Gson to automatically transform JSON responses into Post objects and vice versa.

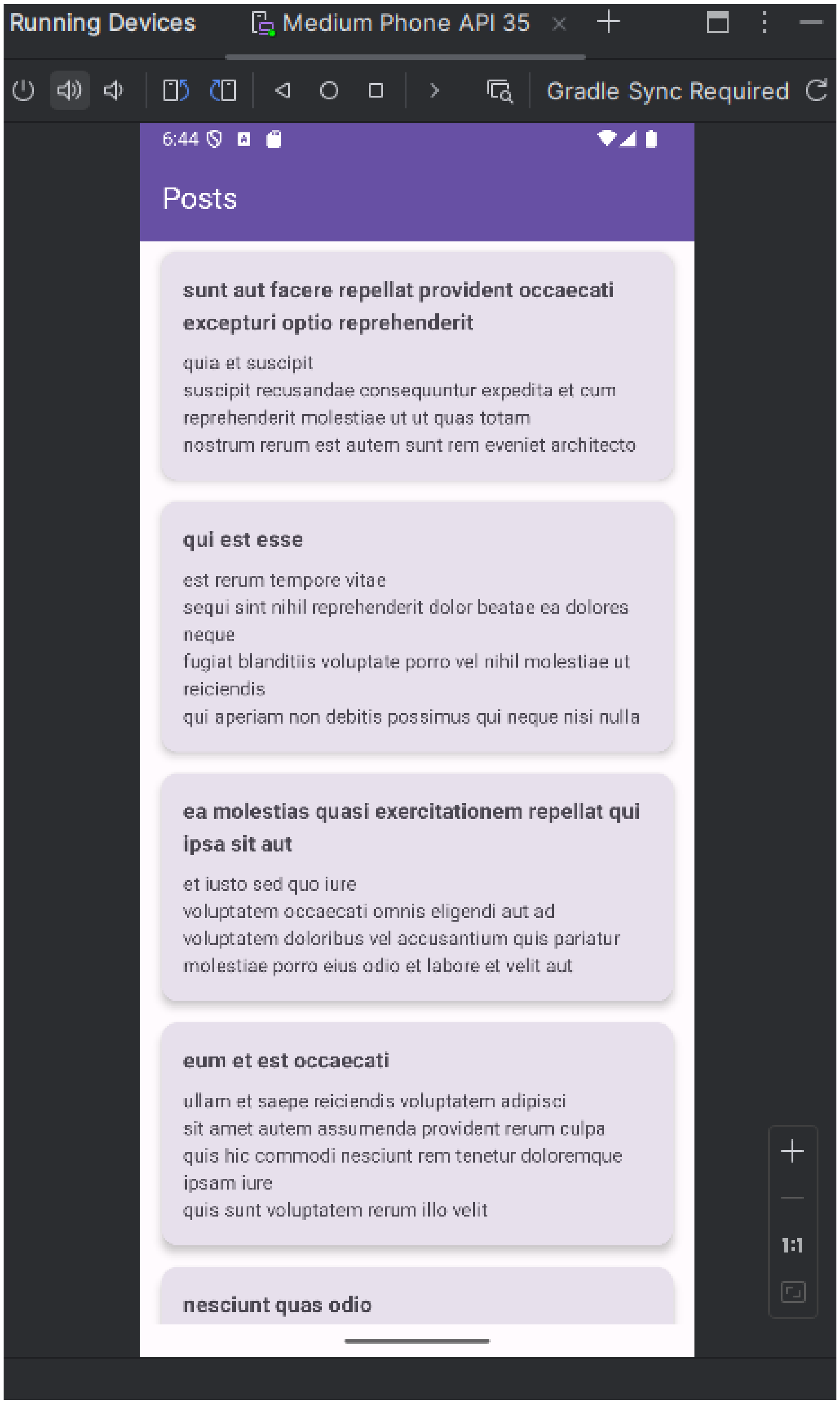
API Calls and Response Handling

API calls are made using Retrofit's Call object or coroutines for asynchronous operations. The response is processed by checking whether the request was successful or if there were any errors. Responses are typically wrapped in Result or sealed classes to handle both success and failure scenarios effectively.



```
class PostRepository {
    suspend fun getPosts(): Flow<Result<List<Post>>> = flow {
            val response = RetrofitClient.apiService.getPosts()
            if (response.isSuccessful) {
                response.body()?.let {
                    emit(Result.Success(it))
                } ?: emit(Result.Error( exception: "Empty response"))
            } else {
                emit(Result.Error( exception: "Error: ${response.code()} ${response.message()}"))
            }
        } catch (e: Exception) {
            emit(Result.Error( exception: "Network error: ${e.localizedMessage}"))
        }
    }.flowOn(Dispatchers.IO)

    suspend fun getPostById(id: Int): Flow<Result<Post>> = flow {
        emit(Result.Loading)
        try {
            val response = RetrofitClient.apiService.getPostById(id)
            if (response.isSuccessful) {
                response.body()?.let {
                    emit(Result.Success(it))
                } ?: emit(Result.Error( exception: "Empty response"))
            } else {
                emit(Result.Error( exception: "Error: ${response.code()} ${response.message()}"))
            }
        } catch (e: Exception) {
            emit(Result.Error( exception: "Network error: ${e.localizedMessage}"))
        }
    }.flowOn(Dispatchers.IO)
}
```

This PostRepository class manages the retrieval of post data and handles network requests via Retrofit. It contains two methods: getPosts() and getPostById(id), which perform asynchronous operations to retrieve a list of posts and a single post by ID, respectively. Both methods use Flow to return results, allowing for handling of asynchronous requests and passing data to the ViewModel.The methods use flow to create a data flow that first emits a loading state (Result.Loading), then performs the request via Retrofit. If the response is successful, the result is processed and emitted as Result.Success, otherwise as Result.Error, with an appropriate error message. In case of a network error, Result.Error will also be emitted. The operations are performed in a background thread via flowOn(Dispatchers.IO), which prevents the main application thread from being blocked.



Posts

sunt aut facere repellat provident occaecati excepturi optio reprehenderit

quia et suscipit
suscipit recusandae consequuntur expedita et cum
reprehenderit molestiae ut ut quas totam
nostrum rerum est autem sunt rem eveniet architecto

qui est esse

est rerum tempore vitae
sequi sint nihil reprehenderit dolor beatae ea dolores neque
fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis
qui aperiam non debitis possimus qui neque nisi nulla

ea molestias quasi exercitationem repellat qui ipsa sit aut

et iusto sed quo iure
voluptatem occaecati omnis eligendi aut ad
voluptatem doloribus vel accusantium quis pariatur
molestiae porro eius odio et labore et velit aut

eum et est occaecati

ullam et saepe reiciendis voluptatem adipisci
sit amet autem assumenda provident rerum culpa
quis hic commodi nesciunt rem tenetur doloremque ipsam iure
quis sunt voluptatem rerum illo velit

nesciunt quas odio

Fetching posts by http request

Conclusion

In this report, we explored the key concepts and implementation strategies for working with local databases and API integration in Kotlin Android applications. while maintaining a clean architecture through DAO and repository patterns. Similarly, Retrofit provides a simple and powerful solution for making API calls, handling responses, and managing network operations asynchronously. Adopting these technologies and best practices will lead to better performance, a smoother user experience, and a more maintainable codebase in Android development.