KAZAKH BRITISH TECHNICAL UNIVERSITY

Assignment 4

Working with Databases and Retrofit in Kotlin Android.

Prepared by Azimkhanov Bauyrzhan

Almaty, 30.11.2024

Exexcutive summary

This report explains two exercises on mobile development using Kotlin for Android. The first exercise is about working with databases using Room. We set up a new Android project, added Room dependencies, and created data models with Room annotations. Then, we made a Data Access Object (DAO) with methods to handle database operations. We set up a Room database class and included the entities and DAO. We also used the Repository Pattern to manage data access and integrated the user interface with RecyclerView to display data. Finally, we used LiveData to update the UI based on changes in the database and ensured operations run in the background.

The second exercise focuses on using Retrofit to make API calls. We set up Retrofit, defined API endpoints, and created data models to match the API response structure. We made API calls in a ViewModel or Repository class, handled responses, and updated the UI. We also implemented caching to improve user experience and wrote unit tests to ensure the reliability of our API service.

These exercises demonstrate key skills in mobile development, including database management and API integration.

Table of contents

Introduction

Databases and RESTful API integration are very important in mobile applications. Databases help store and manage data locally on the device. This ensures that users can access their data even without an internet connection. RESTful APIs allow mobile apps to communicate with remote servers to fetch and send data. This makes sure that the app can provide real-time updates and access to online services.

The purpose of this report is to explore the role of databases and API integration in mobile development. The report will discuss how to set up and use Room for local database management in an Android application. It will also cover how to use Retrofit for making API calls in a Kotlin Android app. By completing these exercises, we aim to understand the key concepts and best practices in mobile development.

Working with databases in Kotlin Android

Overview of Room database

Room is a library provided by Google for Android development. It helps developers work with databases more easily compared to using SQLite directly. Room provides a simple way to create and manage databases in an Android application.

One of the main advantages of Room over SQLite is its use of annotations. This makes the code easier to read and write. For example, with Room, you can use @Entity, @PrimaryKey, and @ColumnInfo annotations to define your data models. This reduces boilerplate code and helps prevent errors.

Another advantage is that Room integrates with LiveData and Flow. This allows the app to automatically update the user interface when the data changes. This is very useful for keeping the app's data and UI in sync without much extra work.

Room also provides compile-time checks for SQL queries. This means that any errors in your SQL queries are detected when you build your project, instead of at runtime. This helps catch and fix errors early in the development process.

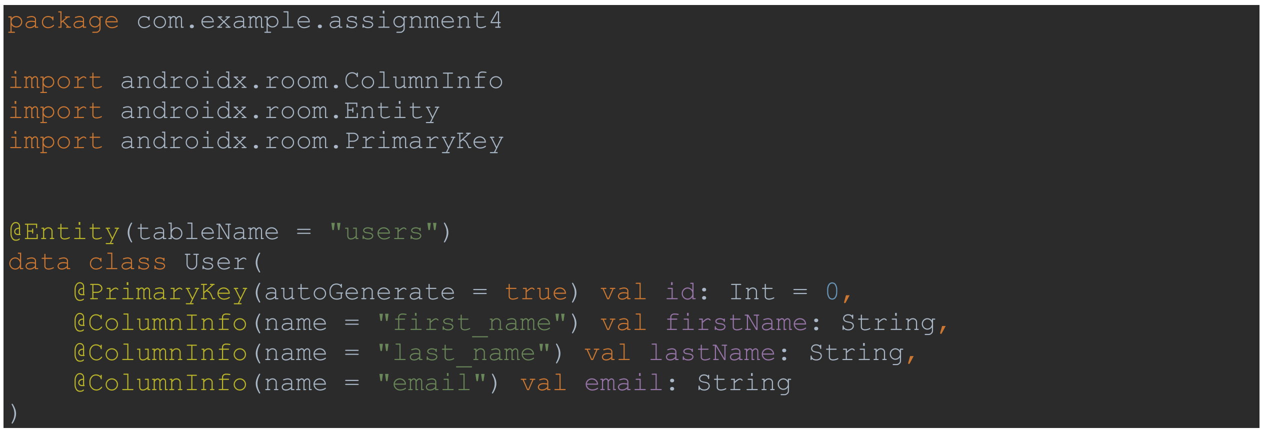
Data models and DAO

In our project, we created data models to represent the entities in our database. For example, we created a data class named User with fields like userId, name, and email. We used Room annotations such as @Entity, @PrimaryKey, and @ColumnInfo to define these fields and specify the database table structure.

To interact with the database, we defined a Data Access Object (DAO). The DAO is an interface that contains methods for performing various database operations. For instance, we created methods like insertUser, updateUser, deleteUser, and getAllUsers. These methods are annotated with Room annotations such as @Insert, @Update, @Delete, and @Query.

The @Insert annotation is used to insert a new user into the database. The @Update annotation updates an existing user's information, while the @Delete annotation removes a user from the database. The @Query annotation is used to retrieve data from the database, such as fetching a list of all users.

By using data models and DAO, we made the database operations more organized and easier to manage in our Kotlin Android application.



```kotlin
package com.example.assignment4

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


@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    @ColumnInfo(name = "first_name") val firstName: String,
    @ColumnInfo(name = "last_name") val lastName: String,
    @ColumnInfo(name = "email") val email: String
)
```

Figure 1. UsersDataClass.kt fragment

Database setup and repository pattern

To set up the Room database, we first created a Room database class that extends RoomDatabase. This class includes the entities and DAO we defined earlier. We annotated the class with @Database and listed the entities and database version. This setup allows Room to manage our database effectively.

Next, we implemented the repository pattern. The repository class provides a clean API for data access to the rest of the application. It abstracts the database operations and allows us to handle data in a more organized way. In our repository, we defined methods like insertUser, updateUser, deleteUser, and getAllUsers. These methods internally call the corresponding DAO methods.

By using the repository pattern, we made the data operations more manageable and separated the data logic from the rest of the app. This approach helps in maintaining clean and modular code, making our application easier to test and maintain.

User interface integration

To display data from the database, we used a RecyclerView in our Android application. RecyclerView is a flexible and efficient component that displays a list of items. We created an adapter class to bind the data from our database to the RecyclerView. The adapter takes the list of data, like users, and creates view holders for each item.

We also set up a click listener to handle user interactions, like selecting an item to update or delete. To add or update data, we used dialogs or separate screens. These screens allow users to enter new data or modify existing data. When users make changes, the app updates the database through the repository, and the changes are reflected in the UI.

By using RecyclerView and connecting it with the database, we ensured that the user interface remains up-to-date with the latest data from the database. This integration makes the app interactive and user-friendly.

Lifecycle awareness

In our project, we used LiveData and coroutines to manage database operations efficiently. LiveData is an observable data holder class that respects the lifecycle of app components, such as activities and fragments. By using LiveData, we ensured that the UI only updates when the data changes and the app is in an active state. This helps avoid memory leaks and unnecessary updates.

We also used coroutines to perform database operations in the background. Coroutines are a lightweight way to handle asynchronous tasks in Kotlin. By using coroutines, we made sure that time-consuming operations like database queries do not block the main thread, keeping the app responsive. For example, we launched coroutines in the ViewModel to fetch or update data from the database.

Combining LiveData and coroutines, we achieved a smooth and efficient data flow in our application. The UI stays updated with the latest data while ensuring the app remains responsive and lifecycle-aware.

Using Retrofit in Kotlin Android

Overview of Retrofit

Retrofit is a popular library for making API calls in Android applications. It simplifies the process of communicating with web services by converting API responses into Java or Kotlin objects. With Retrofit, you can define API endpoints using an interface and annotate methods to specify HTTP requests like GET, POST, PUT, and DELETE.

One of the main benefits of Retrofit is its ease of use. It handles the network calls in the background and automatically converts JSON responses to data models, which saves time and reduces boilerplate code. Retrofit also integrates well with other libraries like Gson for parsing JSON and OkHttp for handling network operations.

Retrofit makes error handling straightforward. You can define error responses and handle them gracefully in your application. Additionally, Retrofit supports synchronous and asynchronous requests, allowing you to choose the best approach for your app's needs.

Overall, Retrofit is a powerful tool for making efficient and reliable API calls in Kotlin Android applications. It simplifies network communication and helps manage API responses effectively.

API service definition

In our project, we defined an API service interface to communicate with the web service. This interface contains methods that describe the different endpoints of the API. For example, we used Retrofit annotations like @GET, @POST, @PUT, and @DELETE to specify the HTTP methods for each endpoint.

Each method in the interface corresponds to a specific API call. For instance, we defined a method to fetch a list of users with the @GET annotation. Similarly, we created methods for adding a new user with @POST, updating user details with @PUT, and deleting a user with @DELETE. We also used annotations like @Path and @Query to pass parameters to the API calls.

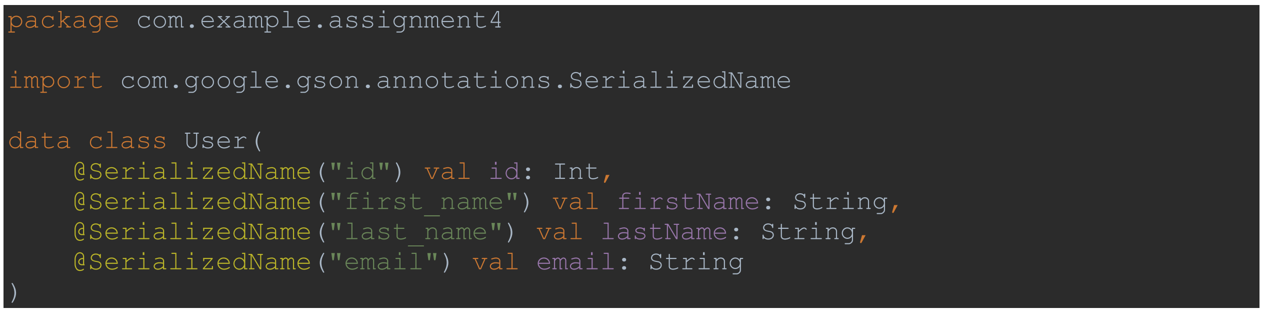
This API service interface helps us organize the different API calls in one place and makes it easier to manage and maintain the network communication in our application.

Data models

In our project, we created data models to represent the structure of the API responses. These data models are Kotlin data classes that match the JSON format returned by the API. For example, if the API returns user information, we created a User data class with fields like id, name, email, and phone. These fields correspond to the keys in the JSON response.

We used Gson annotations, such as @SerializedName, to map the JSON keys to the Kotlin properties when necessary. This ensures that the data is correctly parsed and assigned to the appropriate fields. The data models make it easier to work with the API responses in our application, as we can directly access and manipulate the data using these classes.

By defining these data models, we simplified the process of handling API responses and ensured that our app can easily work with the data provided by the web service.



```kotlin
package com.example.assignment4

import com.google.gson.annotations.SerializedName

data class User(
    @SerializedName("id") val id: Int,
    @SerializedName("first_name") val firstName: String,
    @SerializedName("last_name") val lastName: String,
    @SerializedName("email") val email: String
)
```

Figure 2. UsersDataClassSerializer.kt fragment

API calls and response handling

In our project, we made API calls using the Retrofit library. We defined the API endpoints in the service interface and created methods for each API call. To make an API call, we used the Retrofit instance to create an implementation of the API service interface.

For example, to fetch data from the server, we called the appropriate method in the API service. We used coroutines to handle these API calls asynchronously, ensuring that the main thread remains responsive. The API call returns a Response object, which contains the result of the network request.

We processed the API responses by checking if the call was successful. If successful, we accessed the response body and converted it into our data model. If the call failed, we handled the error gracefully by displaying an error message to the user.

By using Retrofit and coroutines, we efficiently made API calls and processed responses in our Kotlin Android application. This ensured that our app could fetch and display data from the server smoothly.

Caching responses

In our project, we implemented a caching mechanism to enhance the user experience and reduce network calls. We used Room to store API responses locally on the device. When the app makes an API call, it first checks the local cache to see if the data is already available. If the data is found in the cache, it is displayed to the user immediately. This helps to provide quick access to data without waiting for a network response.

If the data is not in the cache or has become outdated, the app makes an API call to fetch fresh data from the server. The new data is then saved in the local database for future use. This approach ensures that the app can work smoothly even when the network connection is slow or unavailable.

By using this caching mechanism, we reduced the number of network calls and improved the overall performance of the application. It also helped in saving bandwidth and provided a better user experience by making the app more responsive.

Conclusion

In this project, we explored important aspects of mobile development using Kotlin for Android. We learned how to effectively manage local databases using Room, which provides advantages over SQLite by simplifying database operations and ensuring data consistency. The use of annotations and LiveData made our database management more efficient and our user interface more responsive. We also integrated RESTful API calls using Retrofit, which streamlined our network communication and data handling. Retrofit's ease of use and seamless integration with other libraries helped us make reliable API calls and manage responses efficiently. By implementing caching mechanisms, we improved the app's performance and user experience.

Overall, this project demonstrated the significance of databases and API integration in building robust and efficient mobile applications. We gained valuable insights into best practices for managing data locally and fetching it from remote servers.

Recommendations

To further improve database handling in Kotlin Android applications, it is recommended to use advanced features of Room, such as migration strategies for updating database schemas and defining complex queries. Additionally, incorporating encryption for sensitive data stored in the database can enhance security. For API integration, consider implementing robust error handling and retry mechanisms to make the app more resilient to network issues. Leveraging third-party libraries for efficient JSON parsing and optimizing network calls can also enhance performance. Adopting best practices such as modularizing the codebase and using dependency injection frameworks like Dagger or Hilt will make the application more maintainable and scalable.

References

1. https://www.geeksforgeeks.org/android-image-picker-from-gallery-using-activityresultcontracts-in-kotlin/
2. https://www.geeksforgeeks.org/bottom-navigation-bar-in-android/
3. https://medium.com/@everydayprogrammer/implement-android-photo-picker-in-android-studio-3562a85c85f1
4. https://medium.com/@prasanth968/kotlin-image-picker-from-gallery-using-activityresultcontracts-9b5aa32e42d0
5. https://developer.android.com/training/data-storage/shared/photopicker#kotlin
6. https://proandroiddev.com/implementing-photo-picker-on-android-kotlin-jetpack-compose-326e33e83b85
7. https://medium.com/geekculture/searchable-recyclerview-e316289edc25
8. https://www.geeksforgeeks.org/gridview-in-android-with-example/
9. https://www.tutorialspoint.com/how-to-create-gridview-layout-in-an-android-app-using-kotlin
10. https://www.geeksforgeeks.org/android-gridview-in-kotlin/
11. https://stackoverflow.com/questions/20191914/how-to-add-gridview-setonitemclicklistener
12. https://developer.android.com/reference/kotlin/androidx/gridlayout/widget/GridLayout
13. https://abhiandroid.com/ui/gridview#gsc.tab=0
14. https://www.geeksforgeeks.org/gridview-in-android-with-example/
15. https://www.tutorialspoint.com/how-to-create-gridview-layout-in-an-android-app-using-kotlin
16. https://www.geeksforgeeks.org/android-gridview-in-kotlin/
17. https://www.geeksforgeeks.org/how-to-implement-android-searchview-with-example/
18. https://developer.android.com/reference/
19. https://kotlinlang.org/docs/home.html

Link to my GitHub repository (screenshots included):

- **https://github.com/BauyrzhanAzimkhanov/Mobile-programming-MSc.git**