Title: “Working with Databases and Retrofit in Kotlin Android Applications”

Issayev Abzal

01.12.2024

KBTU

GitHub: https://github.com/Karisbala/MobDev2024/tree/master/Assignment4

Executive Summary

This report presents the development of a Kotlin Android application focusing on database management and API integration. The project demonstrates the implementation of a local database using Room and the integration of external APIs using Retrofit, all within the MVVM (Model-View-ViewModel) architectural pattern. Jetpack Compose was employed to build the user interface, providing a modern and responsive design.

A local database was established using Room. Data models were defined with appropriate annotations to represent database entities. Data Access Objects (DAOs) were created to handle database operations such as insertion, updating, deletion, and querying. A Room database class was implemented to manage the database instance efficiently [1].

For API integration, Retrofit was utilized. An API client singleton was set up to manage network requests. An interface was defined to represent the API endpoints, using Retrofit annotations to specify HTTP methods and request parameters. Data models were created to match the JSON response structure, facilitating accurate data parsing [2].

The application adheres to the MVVM architecture, promoting a clear separation of concerns. A repository pattern was implemented to provide a clean API for data access, abstracting data sources. ViewModels were used to prepare and manage data for the UI components, ensuring lifecycle awareness and efficient resource management.

Jetpack Compose was used to build the user interface with composable functions, offering a declarative approach to UI development. Navigation between screens was implemented to enhance usability [3].

Caching mechanisms were incorporated using Room to improve performance and provide offline access. Error handling strategies were implemented to manage exceptions gracefully.

An issue with database migration was encountered and resolved by adding the fallbackToDestructiveMigration() method, allowing the application to handle schema changes without crashing during development.

Unit testing was conducted using MockWebServer to ensure the reliability of network operations. Tests were written for the API service and repository methods, verifying data retrieval and caching functionalities.

Table of Contents

List all sections and subsections with corresponding page numbers.

Introduction

Databases and RESTful API integration are essential components in modern mobile applications. They enable apps to store and manage data efficiently while communicating with external services to provide dynamic content. Effective database management ensures data persistence, offline capabilities, and quick data retrieval within the application. Integrating RESTful APIs allows mobile apps to access real-time data from remote servers, enhancing user experience by providing up-to-date information.

The purpose of this report is to discuss the development of a Kotlin Android application that combines local database handling with external API integration. The project demonstrates how to implement a local database using Room and how to integrate APIs using Retrofit, all within the Model-View-ViewModel (MVVM) architectural pattern. The scope of the report includes the setup and configuration of the database, the implementation of API calls, the use of Jetpack Compose for the user interface, and the strategies employed for data caching and error handling.

By exploring these aspects, the report aims to provide insights into building robust and efficient mobile applications that offer a seamless user experience. The techniques and practices outlined serve as a guide for developers seeking to incorporate similar functionalities in their own projects.

Working with Databases in Kotlin Android

Overview of Room Database

Room is a persistence library provided by Android as part of the Jetpack suite. It serves as an abstraction layer over SQLite, simplifying the process of database management in Kotlin Android applications. By leveraging Room, developers can interact with the database using high-level data access objects (DAOs) and Kotlin data classes, reducing the need for boilerplate code and raw SQL queries. Room integrates seamlessly with other Android architecture components like ViewModel and LiveData, supporting reactive programming patterns and ensuring that the user interface remains updated when the underlying data changes.

While SQLite is a powerful relational database engine included in Android, working with it directly can be complex and error-prone. Developers often have to manage database connections, handle cursor objects, and write extensive SQL queries, which increases the likelihood of runtime errors and hinders maintainability [4].

Room addresses these challenges by providing a higher-level abstraction over SQLite. It allows developers to define database schemas using Kotlin classes annotated with specific Room annotations, such as @Entity for tables, @PrimaryKey for primary keys, and @Dao for data access objects. This approach simplifies data mapping and makes the code more readable and maintainable.

One of the significant advantages of Room over direct SQLite usage is compile-time verification of SQL queries. Room checks the SQL queries during compilation, catching errors early in the

development process. This reduces runtime crashes caused by malformed queries and improves overall code reliability.
Room also supports asynchronous database operations through integration with Kotlin coroutines. This ensures that database queries do not block the main thread, providing a smoother user experience. Additionally, Room works well with LiveData and Flow, enabling reactive data streams where the user interface can automatically update in response to data changes without additional code.
Room is designed to be lifecycle-aware, working seamlessly with ViewModel and other architecture components. This helps prevent memory leaks and ensures that data persists through configuration changes like screen rotations. Room also provides tools for handling database migrations when the schema changes, helping maintain data integrity and simplifying updates in future app versions.

Data Models and DAO

Explanation of the data models created and the methods defined in the DAO.
A data model representing a user was created to manage user information within the application. The User data class was defined using Kotlin and annotated with Room annotations to map it to a database table. The class included fields for the user's first name, last name, email, and an auto-generated primary key id. The use of @Entity and @PrimaryKey annotations allowed Room to recognize this class as a database entity and manage it accordingly:

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

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

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

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

In this model, the id field serves as the primary key and is automatically generated by the database. The firstName, lastName, and email fields store the user's personal information.
To interact with the User data model and perform database operations, a Data Access Object (DAO) interface named UserDao was defined. The UserDao interface contains methods annotated with Room annotations to specify database operations such as insert, update, delete, and query. These methods enable the application to manipulate user data efficiently:

```
@Dao
interface UserDao {

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

    @Update
    suspend fun updateUser(user: User)

    @Delete
    suspend fun deleteUser(user: User)

    @Query("SELECT * FROM user_table ORDER BY first_name ASC")
    fun getAllUsers(): Flow<List<User>>

    @Query("SELECT * FROM user_table WHERE user_id = :id")
    suspend fun getUserById(id: Int): User?
}
```

The methods defined in the UserDao are as follows:

- insertUser(user: User): Inserts a new user into the database. If a conflict occurs (for example, a user with the same primary key exists), the existing entry is replaced due to the OnConflictStrategy.REPLACE strategy.
- updateUser(user: User): Updates an existing user's information in the database.
- deleteUser(user: User): Deletes a user from the database.
- getAllUsers(): Fetches all users from the database, ordering them by lastName in ascending order. This method returns a Flow<List<User>>, allowing the application to observe changes in the user data and update the user interface reactively.
- getUserById(id: Int): Retrieves a user by their unique id. This method returns a User object if found or null if no user with the specified id exists.

The UserDao interface utilizes suspending functions and Kotlin's Flow to handle asynchronous operations, ensuring that database interactions do not block the main thread.

Database Setup and Repository Pattern

Overview of the Room database setup and the repository pattern implementation.

The Room database was configured to manage user data efficiently within the application. The AppDatabase class was created by extending RoomDatabase, serving as the main access point to the persisted data. This class included an abstract method userDao() to provide an instance of UserDao, allowing the application to perform database operations defined in the Data Access Object (DAO).

To ensure a single instance of the database throughout the application, the singleton pattern was implemented using a companion object within the AppDatabase class. The getDatabase method checked if an instance already existed; if not, it synchronized the creation of a new

instance using Room.databaseBuilder. This approach prevented the creation of multiple instances in different parts of the application, which could lead to data inconsistencies.
The database was annotated with @Database, specifying the entities included—in this case, the User entity—and the version number. The version number is crucial for handling database migrations in future updates. By defining the database schema in this way, Room could generate the necessary code to manage the database at compile time:

```
@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(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "app_database"
                )
                .fallbackToDestructiveMigration()
                .build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

The repository pattern was implemented through the UserRepository class, acting as an intermediary between the data sources and the rest of the application. The repository provided a clean API for data access, abstracting the underlying data operations and allowing other components to interact with the data without needing to know the details of data storage or retrieval.
The UserRepository included methods corresponding to the operations defined in the UserDao, such as inserting, updating, and deleting users. It also provided a property to access all users as a Flow<List<User>>, enabling the application to observe changes in the data and update the user interface reactively.
By implementing the repository pattern, the application achieved a clear separation of concerns. The ViewModel interacted with the repository to request data, while the repository managed data operations and handled interactions with the Room database. This structure made the

code more maintainable and testable, as changes to data handling logic could be made within the repository without affecting other parts of the application [5]:

```
class UserRepository(private val userDao: UserDao) {

    val allUsers: Flow<List<User>> = userDao.getAllUsers()

    suspend fun insertUser(user: User) {
        userDao.insertUser(user)
    }

    suspend fun updateUser(user: User) {
        userDao.updateUser(user)
    }

    suspend fun deleteUser(user: User) {
        userDao.deleteUser(user)
    }

    suspend fun getUserById(id: Int): User? {
        return userDao.getUserById(id)
    }
}
```

User Interface Integration

The user interface was developed using Jetpack Compose, which offers a modern and declarative approach to building UI components in Kotlin Android applications. Instead of utilizing traditional XML layouts and RecyclerView, Jetpack Compose allowed for the creation of composable functions that directly interact with data provided by the ViewModel:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun UserListScreen(userViewModel: UserViewModel) {
    val userList by userViewModel.allUsers.collectAsState(initial =
emptyList())

    var isDialogOpen by remember { mutableStateOf(false) }
    var userToEdit by remember { mutableStateOf<User?>(null) }

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text("User List") }
```

```
            )
        },
        floatingActionButton = {
            FloatingActionButton(onClick = {
                userToEdit = null
                isDialogOpen = true
            }) {
                Text("+")
            }
        }
    ) { paddingValues ->
        LazyColumn(
            modifier = Modifier
                .fillMaxSize()
                .padding(paddingValues)
        ) {
            items(userList) { user ->
                UserItem(
                    user = user,
                    onEdit = {
                        userToEdit = it
                        isDialogOpen = true
                    },
                    onDelete = {
                        userViewModel.deleteUser(it)
                    }
                )
            }
        }
    }

    if (isDialogOpen) {
        AddEditUserDialog(
            user = userToEdit,
            onDismiss = { isDialogOpen = false },
            onSave = { user ->
                if (user.id == 0) {
                    userViewModel.insertUser(user)
                } else {
                    userViewModel.updateUser(user)
                }
                isDialogOpen = false
            }
        )
    }
```

```
}
```

The main UI component is the UserListScreen, a composable function that displays a list of users retrieved from the Room database. This screen employs a LazyColumn, which efficiently renders a scrollable list of items on the screen. Each user in the list is represented by a UserItem composable, displaying the user's first name, last name, and email address:

```
@Composable
fun AddEditUserDialog(
    user: User?,
    onDismiss: () -> Unit,
    onSave: (User) -> Unit
) {
     var firstName by remember { mutableStateOf(user?.firstName ?: "")
}
    var lastName by remember { mutableStateOf(user?.lastName ?: "") }
    var email by remember { mutableStateOf(user?.email ?: "") }

    AlertDialog(
        onDismissRequest = onDismiss,
         title = { Text(text = if (user == null) "Add User" else "Edit
User") },
        text = {
            Column {
                OutlinedTextField(
                    value = firstName,
                    onValueChange = { firstName = it },
                    label = { Text("First Name") },
                    modifier = Modifier.fillMaxWidth()
                )
                OutlinedTextField(
                    value = lastName,
                    onValueChange = { lastName = it },
                    label = { Text("Last Name") },
                    modifier = Modifier.fillMaxWidth()
                )
                OutlinedTextField(
                    value = email,
                    onValueChange = { email = it },
                    label = { Text("Email") },
                    modifier = Modifier.fillMaxWidth()
                )
            }
        },
        confirmButton = {
```

```
            Button(onClick = {
                if (firstName.isNotBlank() && lastName.isNotBlank() &&
email.isNotBlank()) {
                    val newUser = User(
                        id = user?.id ?: 0,
                        firstName = firstName,
                        lastName = lastName,
                        email = email
                    )
                    onSave(newUser)
                }
            }) {
                Text("Save")
            }
        },
        dismissButton = {
            TextButton(onClick = onDismiss) {
                Text("Cancel")
            }
        }
    )
}
```

Data synchronization between the UI and the database is achieved through the use of Kotlin coroutines and Flow. The UserListScreen observes the Flow of user data from the UserViewModel by collecting it as a state using the collectAsState() function. This mechanism ensures that any changes in the database are immediately reflected in the UI, providing real-time updates to the user interface [6]:

```
@Composable
fun UserItem(
    user: User,
    onEdit: (User) -> Unit,
    onDelete: (User) -> Unit
) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(8.dp)
            .clickable { onEdit(user) },
            elevation = CardDefaults.cardElevation(defaultElevation =
4.dp)
    ) {
        Row(
            modifier = Modifier.padding(16.dp),
```

```
            horizontalArrangement = Arrangement.SpaceBetween
        ) {
            Column {
                    Text(text = "${user.firstName} ${user.lastName}",
style = MaterialTheme.typography.titleMedium)
                                Text(text = user.email, style =
MaterialTheme.typography.bodyMedium)
            }
            IconButton(onClick = { onDelete(user) }) {
                Icon(
                    imageVector = Icons.Default.Delete,
                    contentDescription = "Delete User"
                )
            }
        }
    }
}
```

Interactions with the database are facilitated through user actions within the UI. A floating action button (FAB) is present on the UserListScreen, allowing users to add new entries. When the FAB is clicked, an AddEditUserDialog composable is displayed. This dialog enables users to input or modify user details. Upon saving, the dialog invokes the appropriate method in the UserViewModel to insert or update the user in the database.

Deletion of users is handled within the UserItem composable. Each user item includes an icon button that, when pressed, calls the deleteUser method in the UserViewModel, resulting in the removal of that user from the database. This action also triggers an update in the UI due to the observed data flow [7].

By employing Jetpack Compose and adhering to the MVVM architecture, the UI components remain decoupled from the data sources, promoting a clear separation of concerns. The composable functions interact with the UserViewModel to receive data and execute actions, while the ViewModel communicates with the repository and the Room database to manage data operations. This integration ensures a responsive and efficient user interface that seamlessly reacts to data changes and user interactions:

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

    val allUsers: Flow<List<User>> = repository.allUsers

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

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

```
    }

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

    suspend fun getUserById(id: Int): User? {
        return repository.getUserById(id)
    }
}

@Suppress("UNCHECKED_CAST")
class UserViewModelFactory(private val repository: UserRepository) :
ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(UserViewModel::class.java)) {
            return UserViewModel(repository) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

Lifecycle Awareness

Lifecycle awareness was a critical aspect in managing database operations within the application. By leveraging Kotlin coroutines and adopting lifecycle-aware components, the application ensured that database interactions were efficient, safe, and responsive to the user's actions [8].

Initially, LiveData was considered for observing data changes and updating the user interface reactively. LiveData is lifecycle-aware, meaning it respects the lifecycle state of app components such as activities and fragments. It ensures that the UI components observe data only when they are in an active state, preventing memory leaks and unnecessary computations.

However, in this project, Kotlin's Flow was utilized instead of LiveData for handling asynchronous data streams from the database. Flow offers several advantages, including better integration with coroutines and the ability to handle streams of data asynchronously. It provides a cold stream of data, emitting values only when collected, which aligns well with the reactive programming model used in the application.

The UserViewModel class played a pivotal role in managing data and lifecycle awareness. It used the viewModelScope, a coroutine scope tied to the ViewModel's lifecycle. This scope automatically cancels coroutines when the ViewModel is cleared, such as when the associated UI component is destroyed. This behavior prevents memory leaks and ensures that resources are released appropriately.

Database operations, such as inserting, updating, or deleting users, were performed within coroutines launched in the viewModelScope. For example:

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

By launching these operations within viewModelScope, the application ensured that long-running tasks did not block the main thread, enhancing performance and responsiveness. It also guaranteed that these operations were canceled if the user navigated away from the screen or if the ViewModel was otherwise disposed of.
In the user interface layer, composable functions collected data from the ViewModel using collectAsState(), which is lifecycle-aware and cancels the collection when the composable leaves the composition. This method allowed the UI to react to data changes efficiently without risking memory leaks or unnecessary updates when the UI was not visible.
For instance, in the UserListScreen composable:

```
val userList by userViewModel.allUsers.collectAsState(initial =
emptyList())
```

Here, allUsers is a Flow<List<User>> from the ViewModel. By collecting it as state, the UI automatically recomposes when the data changes, ensuring that the displayed list of users is always up-to-date.
The combination of Flow and coroutines provided a robust mechanism for asynchronous programming while respecting the lifecycle of UI components. This approach allowed the application to handle data streams and database operations without manual lifecycle management, reducing complexity and potential errors.

Using Retrofit in Kotlin Android

Overview of Retrofit

Retrofit is a type-safe HTTP client for Android and Java, developed by Square. It simplifies the process of making network requests to RESTful web services by converting HTTP APIs into Java or Kotlin interfaces. Retrofit abstracts the complexity of network operations, allowing developers to focus on application logic rather than the intricacies of HTTP communication. By using annotations to define API endpoints, request methods, and parameters, Retrofit enables clean and concise code. It integrates seamlessly with popular JSON parsing libraries like Gson, making it straightforward to convert JSON responses into Kotlin data classes. Retrofit supports both synchronous and asynchronous network calls, providing flexibility in handling network operations.
Retrofit offers significant benefits for making API calls in Kotlin Android applications. It simplifies network communication by reducing boilerplate code and allowing developers to define API

endpoints using interfaces and annotations. This approach results in a cleaner and more maintainable codebase. Retrofit ensures type safety by using strongly typed method signatures, which reduces runtime errors related to incorrect data handling.

Integration with Gson allows Retrofit to automatically serialize and deserialize JSON data, streamlining the process of handling API responses and requests. This eliminates the need for manual parsing and reduces the likelihood of errors. Retrofit supports asynchronous calls with coroutines, enabling network operations without blocking the main thread. This leads to smoother user experiences and better application performance.

Error handling is facilitated by Retrofit's built-in mechanisms to manage HTTP errors and exceptions gracefully. Developers can define how to respond to different HTTP status codes, ensuring robust error management. Customizable interceptors can be used with Retrofit through integration with OkHttp, allowing the addition of headers, logging, or modification of requests and responses. This is useful for adding authentication tokens, logging network activity, or implementing caching strategies.

Retrofit was used to integrate external RESTful APIs into the Kotlin Android application. An API client singleton was established using Retrofit to manage network requests efficiently. An interface was defined to represent the API endpoints, utilizing Retrofit annotations such as @GET, @POST, and @Query to specify HTTP methods and parameters. Data models were created to match the JSON response structures of the APIs, and Gson was used as the converter to automatically parse JSON responses into Kotlin data classes:

By using Retrofit, the application benefited from efficient network operations, clean and maintainable code, robust error handling, and scalability. Asynchronous API calls were made using coroutines, ensuring that network operations did not block the main thread and the user interface remained responsive. The use of interfaces and annotations made the codebase organized and easy to understand. Error handling mechanisms provided by Retrofit enhanced the user experience by handling different HTTP responses and exceptions effectively. Adding new API endpoints or modifying existing ones was straightforward due to Retrofit's flexible structure.

API Service Definition

The API service interface was defined to specify how the application communicates with the external RESTful API using Retrofit. The interface serves as a contract between the application and the API, outlining the available endpoints and the expected request and response formats.

An interface named ApiService was created to represent the API endpoints. This interface uses Retrofit annotations to define the HTTP methods and request parameters. For example, to fetch a list of posts from the JSONPlaceholder API, a method was declared with the @GET annotation [9]:

```
interface ApiService {

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

In this method, @GET("posts") indicates that a GET request is made to the posts endpoint. The method getPosts() is a suspending function, allowing it to be called within a coroutine for asynchronous execution. It returns a Response<List<Post>>, where Post is a data class representing the structure of a post as defined by the API.
By defining the method in the ApiService interface, the application can make network calls to the API endpoints in a type-safe and organized manner. Retrofit generates the implementation of these methods at runtime, handling the network requests and responses.
The use of Retrofit annotations simplifies the process of specifying request methods, parameters, headers, and other configurations. It abstracts the complexity of building HTTP requests and parsing responses, allowing developers to focus on application logic.

Data Models

Data models were created to represent the structure of the API responses, allowing the application to parse and utilize the data effectively. These data models are Kotlin data classes that mirror the JSON objects returned by the API endpoints, facilitating seamless integration with Retrofit and Gson for JSON serialization and deserialization.

For the JSONPlaceholder API, a primary data model named Post was defined to represent individual posts retrieved from the API. The Post data class corresponds directly to the JSON structure provided by the API, ensuring that each field in the JSON response is accurately mapped to a property in the Kotlin class.

```
@Entity(tableName = "posts")
data class Post(
   val userId: Int,
   @PrimaryKey val id: Int,
   val title: String,
   @SerializedName("body") val body: String
)
```

In this model:

- userId represents the identifier of the user who created the post.
- id is the unique identifier of the post.
- title holds the title of the post.
- body contains the main content of the post.

By defining the data class with these properties, the application can automatically deserialize the JSON response into Kotlin objects using Gson, which is integrated with Retrofit. This automatic parsing eliminates the need for manual handling of the JSON data, reducing the potential for errors and simplifying the codebase.
If the JSON keys in the API response differ from the property names in the Kotlin class, the @SerializedName annotation from Gson [10] can be used to map the JSON keys to the correct

properties. However, in this case, the property names match the JSON keys, so additional annotations were not necessary.

API Calls and Response Handling

API calls were made using Retrofit to communicate with an external RESTful API. The process involved defining the API endpoints, creating a Retrofit service, and handling the responses in a way that integrates smoothly with the application's architecture.
To make API calls, an instance of the Retrofit service was created using a singleton pattern. The ApiClient object was responsible for initializing Retrofit with the base URL of the API and setting up any necessary configurations such as converters and interceptors. This ensured that network operations were managed efficiently and consistently throughout the application.

```
object ApiClient {

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

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

   fun <Api> createService(apiClass: Class<Api>): Api {
       return retrofit.create(apiClass)
   }

   private fun provideOkHttpClient(): OkHttpClient {
       val loggingInterceptor = HttpLoggingInterceptor()
       loggingInterceptor.level = HttpLoggingInterceptor.Level.BODY

       return OkHttpClient.Builder()
           .addInterceptor(loggingInterceptor)
           .build()
   }
}
```

With the Retrofit service established, API calls were made by invoking the methods defined in the ApiService interface. For example, to fetch a list of posts, the getPosts() method was called. This method was a suspending function, allowing it to be executed within a coroutine for asynchronous operation.

In the PostRepository, the API call was encapsulated in a function that handled the network request and processed the response:

```
class PostRepository(
    private val apiService: ApiService,
    private val postDao: PostDao
) {

    suspend fun fetchPosts(): Result<List<Post>> {
        return withContext(Dispatchers.IO) {
            try {
                val response = apiService.getPosts()
                if (response.isSuccessful) {
                    response.body()?.let { posts ->
                        postDao.insertPosts(posts)
                        Result.success(posts)
                    } ?: Result.failure(Exception("No data"))
                } else {
                    Result.failure(Exception("Error
${response.code()}"))
                }
            } catch (e: Exception) {
                Result.failure(e)
            }
        }
    }

    suspend fun getCachedPosts(): List<Post> {
        return withContext(Dispatchers.IO) {
            postDao.getAllPosts()
        }
    }
}
```

In this function, the API call was executed, and the response was checked for success. If the response was successful, the body was retrieved and wrapped in a Result.success. If there was an error, a Result.failure was returned with an appropriate exception.

Responses were processed by parsing the JSON data into Kotlin data classes using Gson, which was integrated with Retrofit through the GsonConverterFactory. This allowed the application to work with strongly typed data, simplifying data manipulation and reducing the likelihood of errors.

Error handling was an important aspect of response processing. The application checked the HTTP status codes and handled exceptions such as network failures or unexpected responses. By using the Result class, the success or failure of the API call could be communicated back to the ViewModel, which then updated the user interface accordingly.

In the ViewModel, the data fetched from the repository was exposed to the UI through observable properties. The ViewModel launched coroutines to call the repository functions and handled the results:

```
class PostViewModel(private val repository: PostRepository) :
ViewModel() {

    private val _posts = MutableStateFlow<List<Post>>(emptyList())
    val posts: StateFlow<List<Post>> get() = _posts

    private val _error = MutableStateFlow<String?>(null)
    val error: StateFlow<String?> get() = _error

    fun loadPosts() {
        viewModelScope.launch {
            val result = repository.fetchPosts()
            if (result.isSuccess) {
                _posts.value = result.getOrDefault(emptyList())
            } else {
                _error.value = result.exceptionOrNull()?.message
                // Load cached posts
                _posts.value = repository.getCachedPosts()
            }
        }
    }
}
```

In this example, the loadPosts() function in the ViewModel called the fetchPosts() method from the repository. Based on the result, it updated the _posts LiveData with the list of posts or set the _error LiveData with an error message.
The user interface observed these LiveData properties and updated accordingly. When posts were loaded successfully, they were displayed in a list. If there was an error, an appropriate message was shown to the user.

Caching Responses

Caching was implemented to enhance the application's performance and provide a better user experience by reducing dependency on network availability. The caching mechanism involved storing API responses locally using Room, allowing the application to access data even when offline or when network connectivity is poor.
When the application fetched data from the external API using Retrofit, the responses were saved in the local database. Specifically, when the fetchPosts() method in the PostRepository retrieved posts from the API, it not only returned the data to the calling function but also inserted the posts into the local database using the PostDao.

```
suspend fun fetchPosts(): Result<List<Post>> {
    return withContext(Dispatchers.IO) {
        try {
            val response = apiService.getPosts()
            if (response.isSuccessful) {
                response.body()?.let { posts ->
                    postDao.insertPosts(posts)
                    Result.success(posts)
                } ?: Result.failure(Exception("No data"))
            } else {
                Result.failure(Exception("Error ${response.code()}"))
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}
```

In this function, the application first attempted to fetch posts from the API. If successful, it inserted the posts into the local database using the insertPosts() method of the PostDao. This ensured that the latest data from the API was cached locally.
To retrieve cached data, the getCachedPosts() method was used:

```
suspend fun getCachedPosts(): List<Post> {
    return withContext(Dispatchers.IO) {
        postDao.getAllPosts()
    }
}
```

In the PostViewModel, when an error occurred during the API call (such as network failure), the application fell back to using the cached data:

```
fun loadPosts() {
    viewModelScope.launch {
        val result = repository.fetchPosts()
        if (result.isSuccess) {
            _posts.value = result.getOrDefault(emptyList())
        } else {
            _error.value = result.exceptionOrNull()?.message
            // Load cached posts
            _posts.value = repository.getCachedPosts()
        }
    }
}
```

By updating _posts.value with the cached data, the application ensured that users could still view previously fetched posts even when the API call failed.

Conclusion

The project successfully integrated database management and API communication within a Kotlin Android application. By using Room for local data persistence, the application efficiently handled offline data storage and retrieval. Room simplified database interactions by providing an abstraction over SQLite, allowing for easier data modeling and querying without the need for extensive boilerplate code.

Retrofit was employed to manage network operations and communicate with external RESTful APIs. It streamlined the process of making API calls by converting HTTP requests into Kotlin interfaces, enabling clean and maintainable code. The integration with Gson facilitated automatic serialization and deserialization of JSON data, reducing the complexity of parsing responses.

The implementation of the MVVM architectural pattern enhanced the application's structure by promoting a clear separation of concerns. ViewModels managed data preparation and handled background tasks using coroutines, ensuring that the user interface remained responsive and that data operations did not block the main thread.

Caching mechanisms were implemented using Room to store API responses locally. This approach improved performance and provided a fallback option when network connectivity was unavailable, ensuring that users could access data consistently.

In summary, the combination of Room and Retrofit in a Kotlin Android application provided a robust solution for managing data effectively. The project's success demonstrated that leveraging these technologies leads to responsive, reliable applications that offer seamless user experiences with efficient data handling and up-to-date information.

Recommendations

To further enhance the application's database handling and API integration, implementing proper database migration strategies is recommended. Instead of relying on fallbackToDestructiveMigration(), which can lead to data loss when the schema changes, defining explicit migration paths would preserve user data and ensure a seamless transition between database versions.

Integrating a dependency injection framework such as Hilt or Dagger could improve the management of dependencies within the application. This would promote better code organization, facilitate testing, and adhere to best practices in modern Android development by providing a scalable and maintainable codebase.

References

List of all sources, libraries, and documentation referenced in the report.
Use a consistent citation style (e.g., APA, MLA).

1. Android Developers. (n.d.). *Room Persistence Library*. Retrieved from https://developer.android.com/training/data-storage/room
2. Square. (n.d.). *Retrofit*. Retrieved from https://square.github.io/retrofit/
3. Android Developers. (n.d.). *Jetpack Compose*. Retrieved from https://developer.android.com/jetpack/compose
4. Android Developers. (n.d.). *Save data using SQLite*. Retrieved from https://developer.android.com/training/data-storage/sqlite
5. Android Developers. (n.d.). *Guide to app architecture*. Retrieved from https://developer.android.com/jetpack/guide
6. Android Developers. (n.d.). *Asynchronous data flow*. Retrieved from https://developer.android.com/kotlin/flow
7. Android Developers. (n.d.). *Manage UI state with ViewModel*. Retrieved from https://developer.android.com/topic/libraries/architecture/viewmodel
8. Android Developers. (n.d.). *Lifecycle-aware components*. Retrieved from https://developer.android.com/topic/libraries/architecture/lifecycle
9. JSONPlaceholder. (n.d.). *Fake Online REST API for Testing and Prototyping*. Retrieved from https://jsonplaceholder.typicode.com/
10. Google Developers. (n.d.). *Gson User Guide*. Retrieved from https://github.com/google/gson/blob/master/UserGuide.md

Appendices

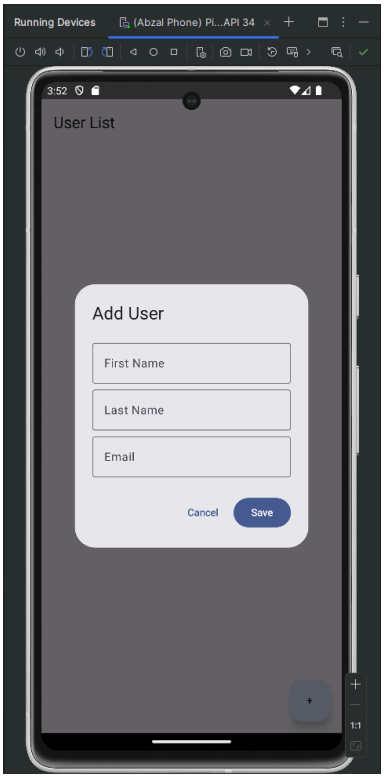
MainActivity file:

```
class MainActivity : ComponentActivity() {
    private val apiService =
ApiClient.createService(ApiService::class.java)

    //private val database by lazy { AppDatabase.getDatabase(this) }
    private val database by lazy { PostDatabase.getDatabase(this) }

    //private val repository by lazy {
UserRepository(database.userDao()) }
    private val repository by lazy { PostRepository(apiService,
database.postDao()) }


//    private val userViewModel: UserViewModel by viewModels {
//        UserViewModelFactory(repository)
//    }
    private val postViewModel: PostViewModel by viewModels {
```

```
        PostViewModelFactory(repository)
    }


    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Assignment4Theme {
                //UserListScreen(userViewModel = userViewModel)
                PostListScreen(postViewModel)
            }
        }
    }
}
```

Exercise 1:

Picture 1. Empty list of users screen.



Picture 2. Add User dialog after the “+” button is pressed to add a new user to the list.

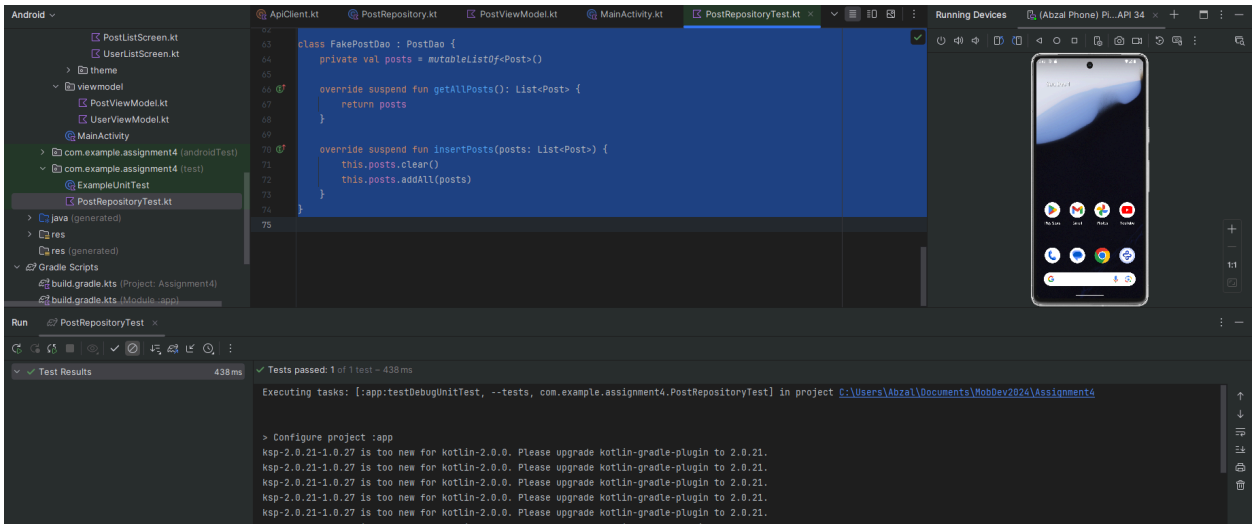
Picture 3. Populated list of users screen with 1 user.

Exercise 2:

Unit test file:

```
class PostRepositoryTest {

    private lateinit var mockWebServer: MockWebServer
    private lateinit var apiService: ApiService
    private lateinit var repository: PostRepository

    @Before
    fun setup() {
        mockWebServer = MockWebServer()
        mockWebServer.start()

        apiService = Retrofit.Builder()
            .baseUrl(mockWebServer.url("/"))
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(ApiService::class.java)

        repository = PostRepository(apiService, FakePostDao())
    }

    @After
    fun teardown() {
        mockWebServer.shutdown()
    }

    @Test
    fun `fetchPosts returns posts on success`() = runBlocking {
        val mockResponse = MockResponse()
        mockResponse.setBody(
            """
            [
                {"userId": 1, "id": 1, "title": "Test Title", "body":
"Test Body"}
            ]
            """
        )
        mockWebServer.enqueue(mockResponse)

        val result = repository.fetchPosts()
```

```
        assertTrue(result.isSuccess)
        val posts = result.getOrNull()
        assertNotNull(posts)
        assertEquals(1, posts?.size)
        assertEquals("Test Title", posts?.first()?.title)
    }
}

class FakePostDao : PostDao {
    private val posts = mutableListOf<Post>()

    override suspend fun getAllPosts(): List<Post> {
        return posts
    }

    override suspend fun insertPosts(posts: List<Post>) {
        this.posts.clear()
        this.posts.addAll(posts)
    }
}
```



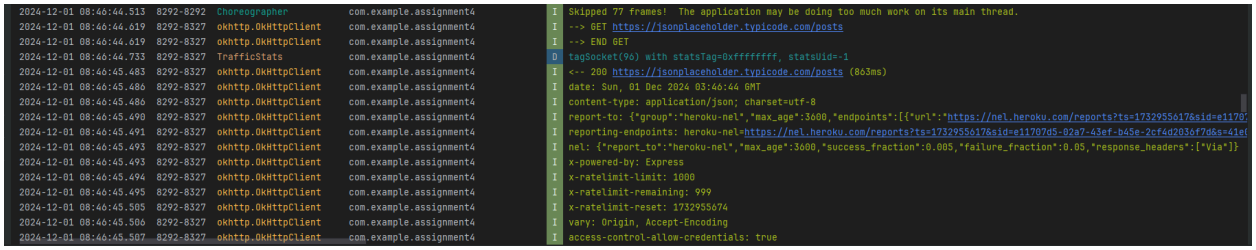
Picture 4. Successful PostRepository Unit test pass.

PostDatabase file:

```
@Database(entities = [Post::class], version = 2)
abstract class PostDatabase : RoomDatabase() {
    abstract fun postDao(): PostDao

    companion object {
        @Volatile private var instance: PostDatabase? = null
```

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

Picture 5. List of Posts fetched from the API.



Picture 6. 200 status code for GET request from the API.