MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

KAZAKH-BRITISH TECHNICAL UNIVERSITY

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING

REPORT

Working with Databases and Retrofit in Kotlin Android Applications

Mobile Programming

Assignment 4

Presented to Serek A.G.

Done by Kabyl Dauren

Student ID: 23MD0452

Almaty 2024

Executive Summary

This project focuses on building a Task Management application using tools like Jetpack Compose, Room, Retrofit, ViewModel, and LiveData. The first part of the project explores local database management by implementing Room to handle task data, leveraging the repository pattern for clean data access and lifecycle-aware components for seamless UI updates. The second part focuses on integrating network operations using Retrofit to consume RESTful APIs, enabling real-time data interaction and response handling. Advanced features like caching, error handling, and coroutines ensure optimal performance and user experience. By combining these technologies, the project provides a comprehensive understanding of building efficient, maintainable, and responsive Android applications. The final deliverable is a functional and extensible application demonstrating clean architecture and robust development practices.

Table of contents

Introduction

The purpose of this project is to explore the foundations of building Android applications using Kotlin and Jetpack components like Room, Retrofit, ViewModel, and LiveData. The project is structured around two key exercises: implementing a local database using Room and integrating API calls using Retrofit. The goal is to create a functional and efficient Task Management application that demonstrates modern Android development practices.

In the first exercise, the focus is on understanding how to set up and manage a local database using Room. Room provides an abstraction layer over SQLite, making it easier to work with databases while adhering to best practices. We'll study how to define entities, create a Data Access Object (DAO) for database operations, and set up a repository to interact with the database. Through the integration of ViewModel and LiveData, the application ensures lifecycle-aware UI updates and seamless background thread operations using coroutines.

In the second exercise, the objective shifts to API integration with Retrofit. Retrofit simplifies the process of making network requests by providing a robust framework for handling HTTP methods, request parameters, and JSON serialization/deserialization. We'll explore how to set up Retrofit, define API services, and process responses. Additional features like error handling and response caching will be implemented to enhance the user experience and optimize network usage.

Objectives:

1. Learn how to define and manage local databases using Room;
2. Understand the repository pattern for clean data access;
3. Using ViewModel and LiveData to manage data and UI state;
4. Implementing CRUD operations for tasks within a Task Management application;
5. Exploring lifecycle awareness and background threading using coroutines;
6. Learn to consume RESTful APIs using Retrofit;
7. Handle JSON serialization/deserialization with Gson;
8. Implement robust error handling and caching mechanisms;
9. Integrate API data with the user interface and provide real-time updates.

Link to Github repository: https://github.com/diikiin/assignment4-mob-pro

Working with Databases in Kotlin Android

Overview of Room Database

Room is a persistence library provided by Google as part of the Jetpack suite, designed to simplify database management in Android applications [1]. It serves as an abstraction layer over SQLite, providing a more robust, type-safe, and developer-friendly API for managing local databases. Room reduces boilerplate code while ensuring compile-time verification of SQL queries, which minimizes runtime errors. Unlike raw SQLite, Room integrates seamlessly with other Jetpack components, such as LiveData, Flow, and ViewModel, enabling lifecycle-aware data handling.

To add Room to our project we need to implement several dependencies:

1. androidx.room:room-ktx - Room for Kotlin;
2. androidx.room:room-runtime - Using room during runtime;
3. androidx.room:room-compiler - Compiler for generating Room implementations.

The key advantages of Room over SQLite include:

1. Ease of Use: Room eliminates the need to write complex SQL boilerplate code, making database operations simpler and cleaner;
2. Compile-Time Query Validation: Room validates SQL queries during compilation, reducing runtime query errors;
3. Seamless Integration: Room works effortlessly with LiveData and Kotlin coroutines, ensuring reactive and lifecycle-aware data management;
4. Abstraction of Common Tasks: Room automatically handles common operations like migrations and database creation, reducing development effort.

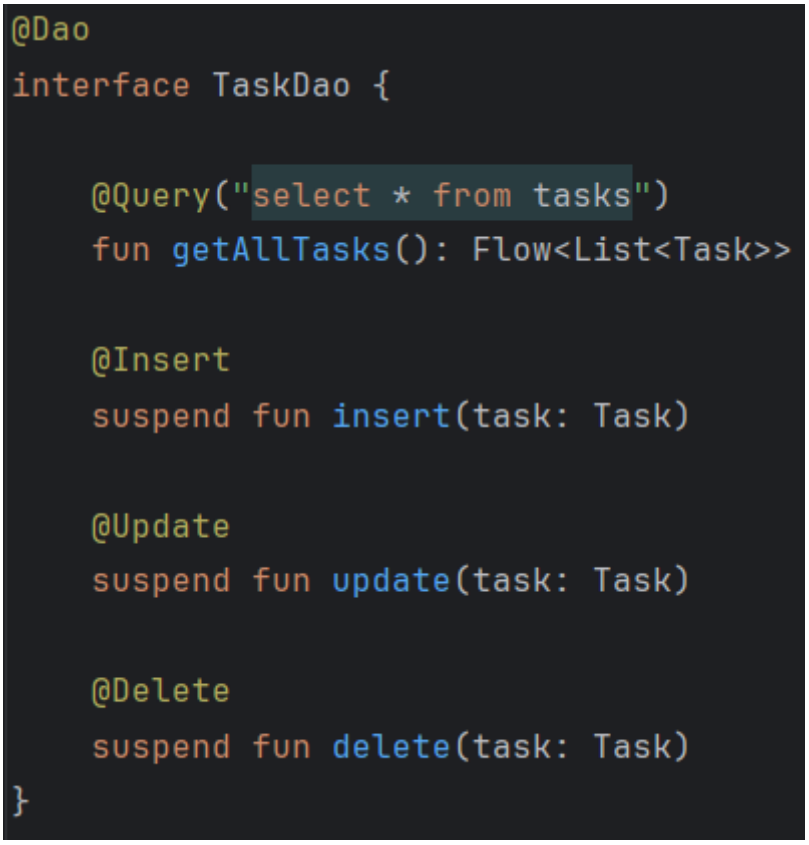
Data Models and DAO

In the Task Management application, the primary data model is the **Task** entity. This data model represents the schema for the tasks table in the Room database. It is annotated with **@Entity**, @PrimaryKey, and other Room annotations to define its structure and constraints. The Task entity includes fields for an id (primary key), title, description, and isDone (status).

```kotlin
@Entity(tableName = "tasks")
data class Task(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val title: String,
    val description: String,
    val isDone: Boolean = false
)
```

Image 1. Task entity

The Data Access Object (DAO) is an interface that defines methods for interacting with the Room database [2]. These methods are annotated with Room annotations such as @Insert, @Update, @Delete, and @Query. The DAO provides a clean and reusable API for performing database operations. These DAOs abstract database interactions, ensuring a clean separation of concerns and enabling the repository to focus solely on business logic. Together, the data model and DAO form the foundation of the Room database, enabling efficient and reliable data storage and retrieval.



```
@Dao
interface TaskDao {

    @Query("select * from tasks")
    fun getAllTasks(): Flow<List<Task>>

    @Insert
    suspend fun insert(task: Task)

    @Update
    suspend fun update(task: Task)

    @Delete
    suspend fun delete(task: Task)
}
```

Image 2. Task DAO

Here:

1. **getAllTasks**: Retrieves all tasks as a Flow, allowing real-time data observation.
2. **insert**: Inserts a new task to the database.
3. **update**: Updates an existing task in the database.
4. **delete**: Deletes a specific task from the database.

Database Setup and Repository Pattern

The Room database setup involves creating a database class that extends **RoomDatabase**. This class serves as the main access point to the database and defines the entities and DAO to be used. In this application, the AppDatabase is the singleton implementation of the Room database. It includes the Task entity and TaskDao, ensuring a centralized and thread-safe way to access the database. The AppDatabase ensures only one instance of the database is created during the application's lifecycle, adhering to the Singleton pattern for efficient resource management.

```kotlin
@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun taskDao(): TaskDao

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

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

Image 3. Database initialization

The repository pattern is implemented to provide a clean API for data access and to decouple database operations from the rest of the application. The TaskRepository acts as a mediator between the TaskDao and the ViewModel, handling both database operations and business logic. The repository simplifies the ViewModel's interaction with the database, ensuring that all database operations are performed in background threads using Kotlin coroutines.

```kotlin
class TaskRepository(private val taskDao: TaskDao) {

    val allTasks: Flow<List<Task>> = taskDao.getAllTasks()

    suspend fun create(task: Task) {
        taskDao.insert(task)
    }

    suspend fun update(task: Task) {
        taskDao.update(task)
    }

    suspend fun delete(task: Task) {
        taskDao.delete(task)
    }
}
```

Image 4. Task repository

User Interface Integration

The user interface is built using Jetpack Compose instead of traditional RecyclerView and XML layouts. Compose's LazyColumn is used to display a list of tasks, offering a simpler and more declarative approach to building scrollable lists [3]. Each task is displayed using the TaskItem composable, which includes a dropdown menu for Update and Delete actions. Full structure of the Task List screen is given in *Appendix I*.

Description of Task List Screen:

1. TaskListScreen:
    a. Displays a header and a button for adding tasks.
    b. Uses LazyColumn to show a scrollable list of tasks managed by TaskItem.
    c. Integrates with TaskViewModel to observe and manipulate task data using LiveData.
2. TaskItem:
    a. Represents an individual task card with task details.
    b. Includes a dropdown menu for actions like marking a task as done, updating, or deleting it.
    c. Clicking on the card opens a detailed TaskDetailDialog.
3. TaskDetailDialog:
    a. Displays task details, including the title, description, and completion status, with a close button.
4. AddOrUpdateTaskDialog:
    a. Handles task creation or updating.
    b. Pre-fills fields for editing existing tasks and validates input before saving.

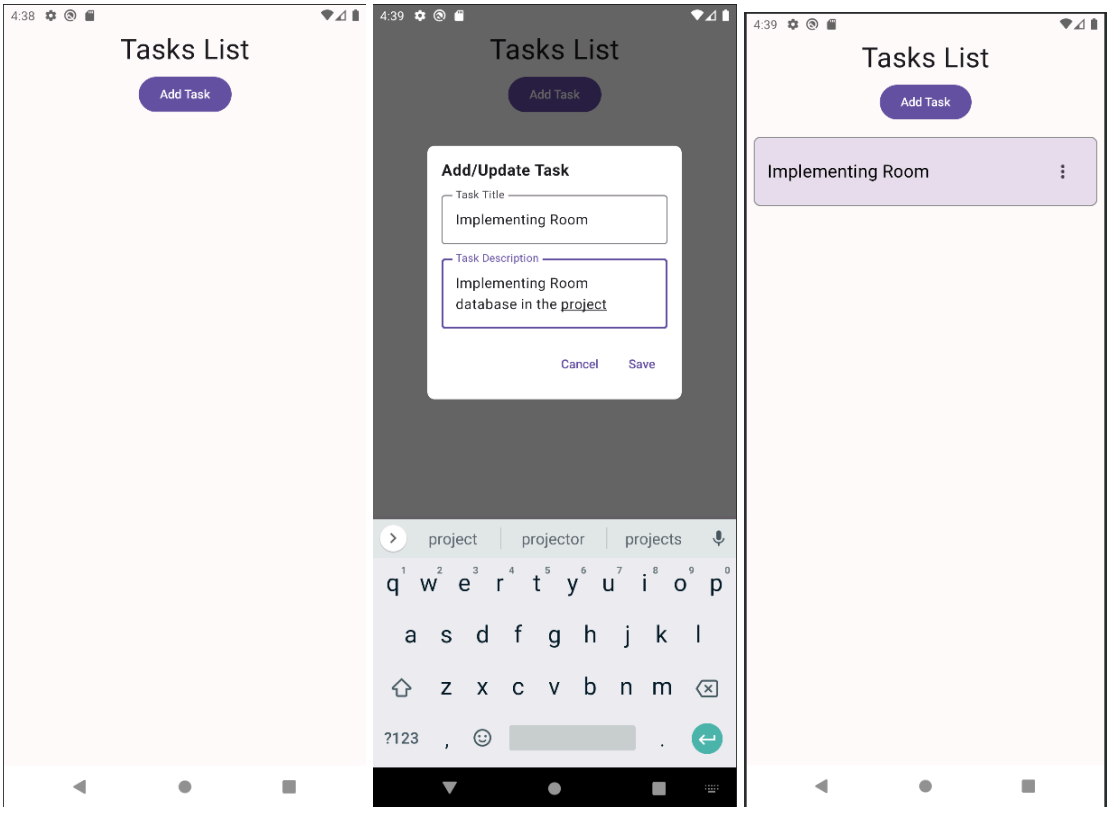


Image 5. Creating new task

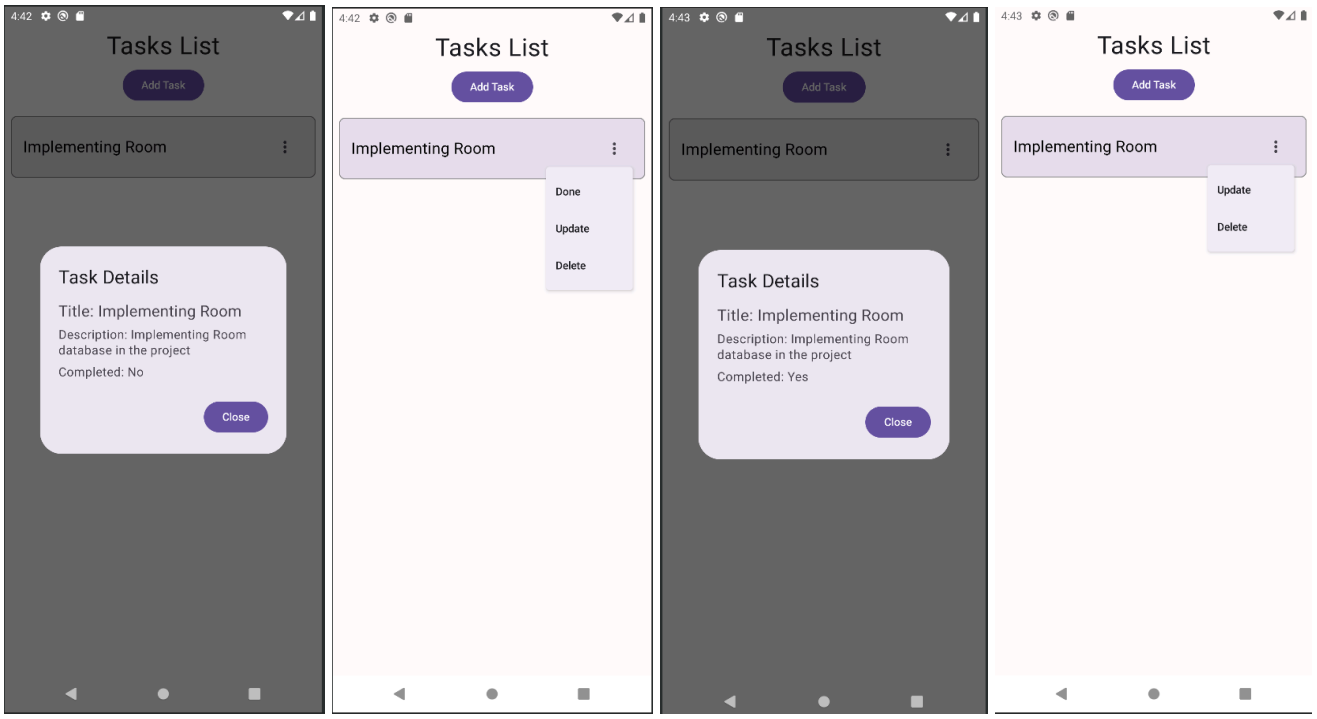


Image 6. Marking task as done

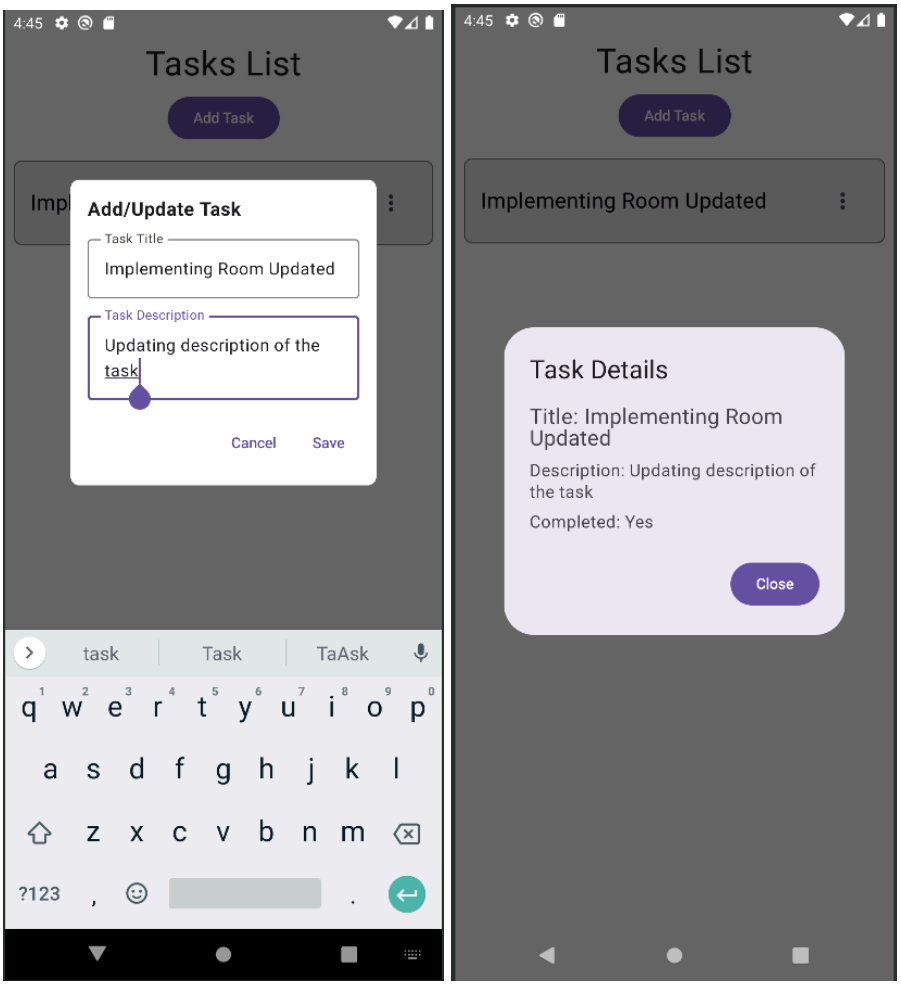


Image 7. Updating the task data

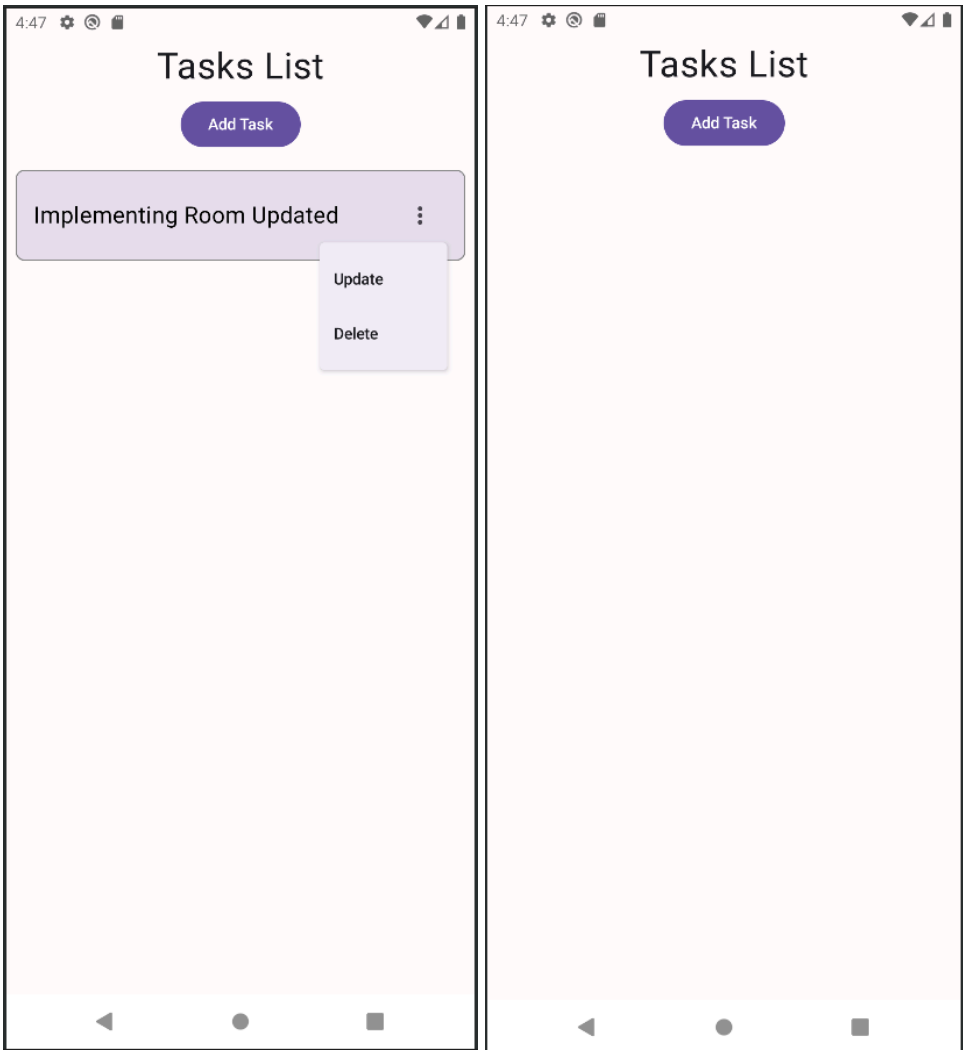
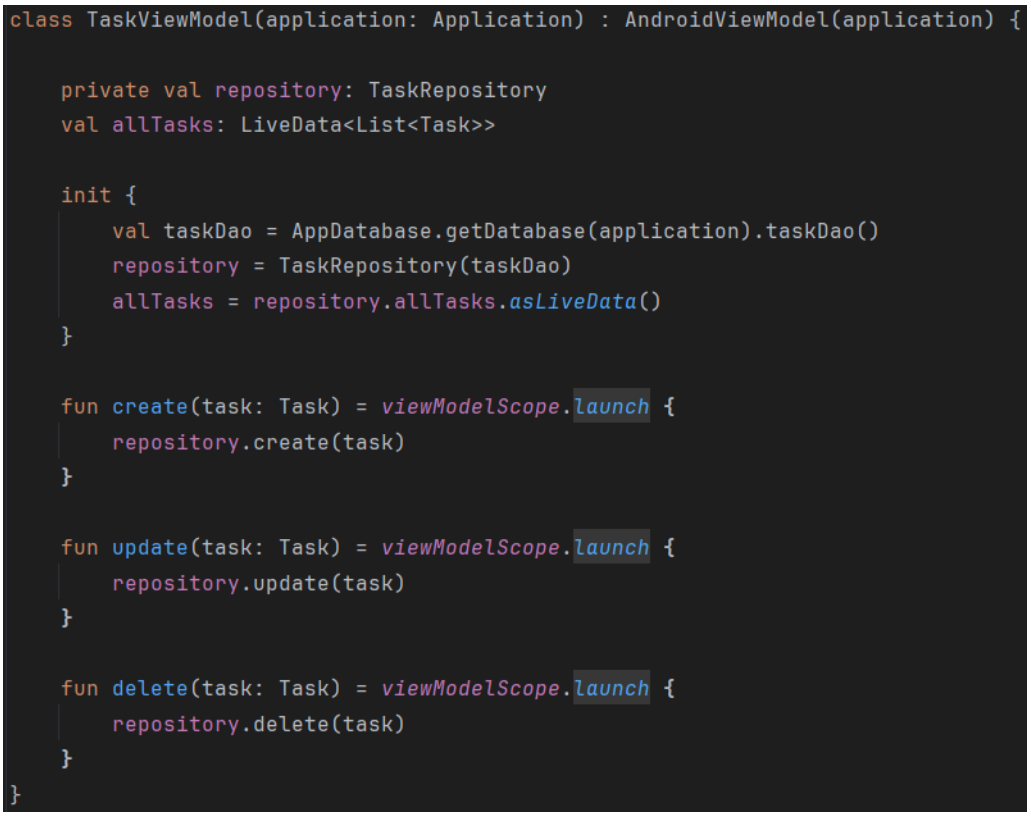


Image 8. Deleting the task

Lifecycle Awareness

ViewModel is a lifecycle-aware class that acts as a central repository for data and logic in your app. LiveData is a special class that we can observe to know when data inside it changed to apply changes in application UI [4]. To add LiveData to our project we need to implement dependency **androidx.compose.runtime:runtime-livedata**. To implement ViewModel and LiveData to our project we created TaskViewModel class that extends ViewModel class and with field tasks with type LiveData inside it. The TaskViewModel uses LiveData to observe changes in the database and automatically update the UI. By converting the repository's Flow into LiveData using asLiveData(), the application maintains lifecycle-aware data handling.



```
class TaskViewModel(application: Application) : AndroidViewModel(application) {

    private val repository: TaskRepository
    val allTasks: LiveData<List<Task>>

    init {
        val taskDao = AppDatabase.getDatabase(application).taskDao()
        repository = TaskRepository(taskDao)
        allTasks = repository.allTasks.asLiveData()
    }

    fun create(task: Task) = viewModelScope.launch {
        repository.create(task)
    }

    fun update(task: Task) = viewModelScope.launch {
        repository.update(task)
    }

    fun delete(task: Task) = viewModelScope.launch {
        repository.delete(task)
    }
}
```

Image 9. Task ViewModel

All database operations are performed in a background thread using Kotlin coroutines (viewModelScope.launch). This ensures that the main thread remains unblocked, preventing UI freezes or performance issues. By combining LiveData and coroutines, the application achieves a robust and lifecycle-aware architecture, ensuring smooth data updates and efficient resource management.

Using Retrofit in Kotlin Android

Overview of Retrofit

The integration of Retrofit into the Task Management application is aimed at enabling seamless communication with a RESTful API. This process involves setting up the Retrofit library, defining API endpoints, creating data models for JSON parsing, and implementing API calls in a ViewModel or Repository class. The use of coroutines ensures asynchronous handling of network requests, contributing to a responsive user interface.

By integrating Retrofit into the Task Management application, the project achieves efficient and seamless communication with a RESTful API. The structured setup of Retrofit, combined with Room and ViewModel, creates a robust system for managing tasks. With asynchronous API calls and error handling, the application ensures a responsive and user-friendly experience. This approach demonstrates modern Android development practices and sets the foundation for scalable app enhancements.

To add Room to our project we need to implement several dependencies:

1. com.squareup.retrofit2:retrofit - Retrofit;
2. com.squareup.retrofit2:converter-gson - Gson converter for Retrofit;

Advantages of Using Retrofit

1. Ease of Use: Simplifies network requests with minimal boilerplate code.
2. Integration with Gson: Automatically parses JSON responses into data models.
3. Asynchronous Handling: Ensures non-blocking UI with Kotlin coroutines.
4. Reusable API Interface: Provides a clear and consistent structure for API endpoints.

```kotlin
object RetrofitInstance {

    private const val BASE_URL = "https://dummyjson.com/"

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

    fun <T> createService(service: Class<T>): T {
        return retrofit.create(service)
    }
}
```

Image 10. Setting up Retrofit

API Service Definition

The ProductService interface defines the API endpoints for CRUD operations on tasks. Retrofit annotations like @GET, @POST, @PUT, and @DELETE specify the HTTP methods and request parameters.

```
interface ProductService {

    @GET("products")
    suspend fun getAll(): List<Product>

    @POST("products")
    suspend fun create(@Body product: Product): Product

    @PUT("products/{id}")
    suspend fun update(@Path("id") id: Int, @Body product: Product): Product

    @DELETE("products/{id}")
    suspend fun delete(@Path("id") id: Int)
}
```

Image 11. Product service

Data Models

The Product data model represents the structure of the JSON response from the API.

```
@Entity(tableName = "products")
data class Product(
    @PrimaryKey
    val id: Int = 0,
    val title: String,
    val description: String,
    val category: String = "",
    val price: Float = 0f,
    val discountPercentage: Float = 0f,
    val rating: Float = 0f
)
```

Image 12. Product entity

API Calls and Response Handling

A sealed class is used to define possible API response states, including success, loading, and error scenarios.

```
sealed class ApiResponse<out T> {
    data class Success<out T>(val data: T) : ApiResponse<T>()
    data class Error(val message: String, val throwable: Throwable? = null) : ApiResponse<Nothing>()
    data object Loading : ApiResponse<Nothing>()
}
```

Image 13. Api Response class

The ProductRepository handles interactions with both the Room database and the Retrofit API. The repository manages API calls and handles errors gracefully:

```
class ProductRepository(
    private val productDao: ProductDao,
    private val productService: ProductService
) {

    private val cachedProducts: Flow<List<Product>> = productDao.getAllTasks()

    suspend fun getAll(): ApiResponse<List<Product>> {
        return try {
            val products = productService.getAll()
            productDao.insertAll(products)
            ApiResponse.Success(products)
        } catch (e: Exception) {
            val cachedProducts = cachedProducts.firstOrNull()
            if (!cachedProducts.isNullOrEmpty()) {
                ApiResponse.Success(cachedProducts)
            } else {
                ApiResponse.Error( message: "Failed to get tasks", e)
            }
        }
    }

    suspend fun syncTasks() {
        val tasks = productService.getAll()
        productDao.insertAll(tasks)
    }

    suspend fun create(product: Product) {
        val newTask = productService.create(product)
        productDao.insert(newTask)
    }
```

Image 14. Product repository

The TaskViewModel interacts with the repository and provides lifecycle-aware data handling for the UI.

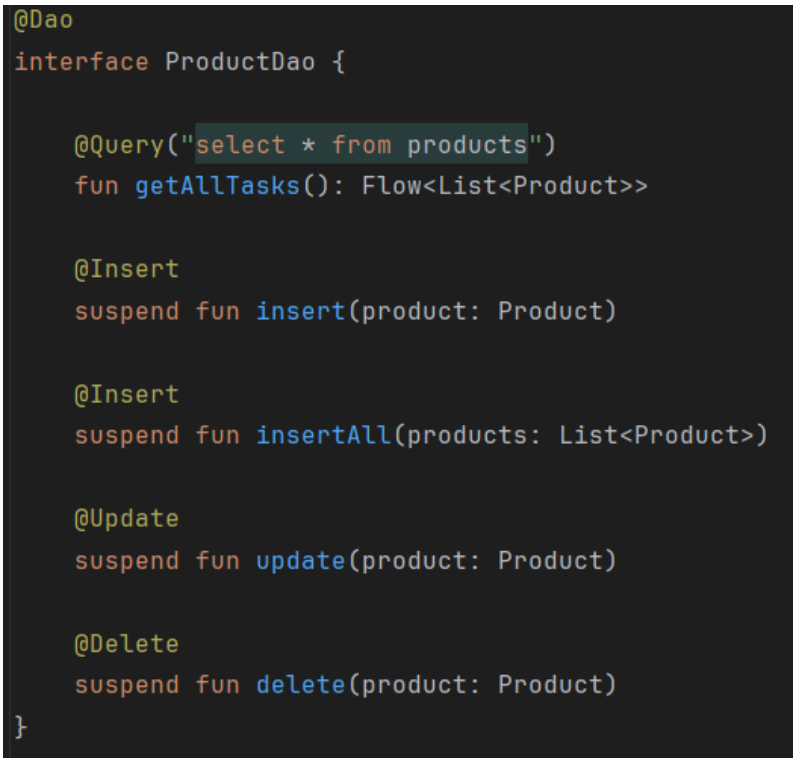
```
class ProductViewModel(application: Application) : AndroidViewModel(application) {

    private val repository: ProductRepository
    private val _productsState = MutableLiveData<ApiResponse<List<Product>>>()
    val productsState: LiveData<ApiResponse<List<Product>>> get() = _productsState

    init {
        val productDao = AppDatabase.getDatabase(application).productDao()
        val productService = RetrofitInstance.createService(ProductService::class.java)
        repository = ProductRepository(productDao, productService)
        loadTasks()
    }

    fun loadTasks() = viewModelScope.launch {
        _productsState.value = ApiResponse.Loading
        _productsState.value = repository.getAll()
    }

    fun create(product: Product) = viewModelScope.launch {
        repository.create(product)
    }

    fun update(product: Product) = viewModelScope.launch {
        repository.update(product)
    }

    fun delete(product: Product) = viewModelScope.launch {
        repository.delete(product)
    }
}
```

Image 15. Product ViewModel

Caching Responses

Caching API responses are implemented using Room and SharedPreferences to improve user experience and minimize unnecessary network calls. Room is utilized to store fetched data locally, ensuring that tasks are accessible offline and providing a fallback in case of network failures. SharedPreferences is employed for lightweight caching, such as storing metadata like the last fetch timestamp, to optimize data synchronization. By combining these approaches, the application ensures seamless functionality and reduces reliance on the network, enhancing both performance and reliability. This caching strategy ensures a responsive and consistent user experience, even in challenging network conditions.

The repository ensures that API responses are cached locally using Room:



```
@Dao
interface ProductDao {

    @Query("select * from products")
    fun getAllTasks(): Flow<List<Product>>

    @Insert
    suspend fun insert(product: Product)

    @Insert
    suspend fun insertAll(products: List<Product>)

    @Update
    suspend fun update(product: Product)

    @Delete
    suspend fun delete(product: Product)
}
```

Image 16. Product DAO for caching in Room

Conclusion

This project successfully demonstrates the implementation of modern Android development practices by building a functional Task Management application. By using Room for local database management, the project highlights the importance of efficient data storage and retrieval in mobile applications. The repository pattern and lifecycle-aware components, such as ViewModel and LiveData, ensure a clean architecture and seamless user experience. The integration of Retrofit showcases how to consume RESTful APIs effectively, enabling real-time data synchronization and interaction. Features like error handling, caching, and coroutines further enhance the application's reliability and performance.

Through this project, we gained hands-on experience with Jetpack Compose for building intuitive and responsive user interfaces. The modular and scalable approach adopted in this application makes it easier to maintain and extend functionality in the future. Overall, the project provides a strong foundation for understanding modern Android app development, preparing developers to build robust and user-friendly applications in real-world scenarios.

Recommendations

To further enhance the functionality and usability of the Task Management application, it is recommended to implement additional features such as task prioritization, reminders, and search functionality to improve user experience. Integration with cloud-based services, such as Firebase or a custom backend, can enable data synchronization across devices and support multi-user collaboration. For better user engagement, incorporating notifications for task deadlines and updates can be highly effective. Additionally, implementing testing strategies, such as unit tests for repository methods and UI tests for Jetpack Compose components, is crucial to ensure application stability and reliability. Finally, considering deployment on the Play Store with appropriate optimizations and support for multiple screen sizes will help make the application accessible to a broader audience. These enhancements will not only increase the app's usability but also showcase best practices in Android development.

References

1. Getting Started with Room Database in Android. URL: https://amitraikwar.medium.com/getting-started-with-room-database-in-android-fa1ca23ce21e
2. Accessing data using Room DAO's: https://developer.android.com/training/data-storage/room/accessing-data?hl=ru
3. LazyColumn in Jetpack Compose: https://medium.com/@ramadan123sayed/lazycolumn-in-jetpack-compose-fa3287ef84da
4. ViewModels and LiveData in Jetpack Compose: https://medium.com/@sujathamudadla1213/viewmodel-and-livedata-in-jetpack-compose-0a12f4d20e3d
5. Retrofit Documentation: https://square.github.io/retrofit/

Appendices

Appendix I. Task List Screen

```
@Composable
fun TaskListScreen(modifier: Modifier = Modifier, taskViewModel: TaskViewModel = viewModel())
{
    val tasks by taskViewModel.allTasks.observeAsState(emptyList())
    var selectedTask by remember { mutableStateOf<Task?>(null) }
    var showDialog by remember { mutableStateOf(false) }

    Column(
        modifier = modifier
    ) {
        Text(
            text = "Tasks List",
            fontSize = 32.sp,
            textAlign = TextAlign.Center,
            modifier = Modifier
                .padding(8.dp)
                .fillMaxWidth()
        )

        Button(
            onClick = {
                showDialog = true
                selectedTask = null
            },
            modifier = Modifier.align(Alignment.CenterHorizontally)
        ) {
            Text("Add Task")
        }
        Spacer(modifier = Modifier.height(8.dp))

        LazyColumn {
            items(tasks) { task ->
                TaskItem(
                    task = task,
                    onDone = {
                        taskViewModel.update(Task(task.id, task.title, task.description, true))
                    },
                    onUpdate = {
                        selectedTask = it
                        showDialog = true
```

```
                },
                onDelete = {
                    taskViewModel.delete(it)
                }
            )
        }
    }
}

if (showDialog) {
    AddOrUpdateTaskDialog(
        task = selectedTask,
        onCreate = { task ->
            taskViewModel.create(task)
            showDialog = false
        },
        onUpdate = { task ->
            taskViewModel.update(task)
            showDialog = false
        },
        onDismiss = { showDialog = false }
    )
}
}

@Composable
fun TaskItem(
    task: Task,
    onDone: (Task) -> Unit,
    onUpdate: (Task) -> Unit,
    onDelete: (Task) -> Unit
) {
    var showDialog by remember { mutableStateOf(false) }
    var expanded by remember { mutableStateOf(false) }

    Card(
        shape = RoundedCornerShape(8.dp),
        border = BorderStroke(1.dp, Color.Gray),
        modifier = Modifier
            .padding(8.dp)
            .fillMaxWidth()
            .clickable { showDialog = true }
    ) {
        Row(
            modifier = Modifier
```

```
            .fillMaxWidth()
            .padding(16.dp),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Column(modifier = Modifier.weight(1f)) {
            Text(
                text = task.title,
                fontSize = 20.sp,
                color = Color.Black,
            )
        }
        Box {
            IconButton(onClick = { expanded = true }) {
                Icon(Icons.Default.MoreVert, contentDescription = "Settings")
            }

            DropdownMenu(
                expanded = expanded,
                onDismissRequest = { expanded = false }
            ) {
                if (!task.isDone) {
                    DropdownMenuItem(
                        onClick = {
                            expanded = false
                            onDone(task)
                        },
                        text = { Text("Done") }
                    )
                }
                DropdownMenuItem(
                    onClick = {
                        expanded = false
                        onUpdate(task)
                    },
                    text = { Text("Update") }
                )
                DropdownMenuItem(
                    onClick = {
                        expanded = false
                        onDelete(task)
                    },
                    text = { Text("Delete") }
                )
            }
        }
```

```
        }
    }

    if (showDialog) {
        TaskDetailDialog(task = task, onDismiss = { showDialog = false })
    }
}

@Composable
fun TaskDetailDialog(task: Task, onDismiss: () -> Unit) {
    AlertDialog(
        onDismissRequest = { onDismiss() },
        title = { Text(text = "Task Details") },
        text = {
            Column {
                Text(text = "Title: ${task.title}", fontSize = 20.sp)
                Spacer(modifier = Modifier.height(8.dp))
                Text(text = "Description: ${task.description}", fontSize = 16.sp)
                Spacer(modifier = Modifier.height(8.dp))
                Text(text = "Completed: ${if (task.isDone) "Yes" else "No"}", fontSize = 16.sp)
            }
        },
        confirmButton = {
            Button(onClick = onDismiss) {
                Text("Close")
            }
        }
    )
}

@Composable
fun AddOrUpdateTaskDialog(
    task: Task?,
    onCreate: (Task) -> Unit,
    onUpdate: (Task) -> Unit,
    onDismiss: () -> Unit
) {
    var taskTitle by remember { mutableStateOf(task?.title ?: "") }
    var taskDescription by remember { mutableStateOf(task?.description ?: "") }

    Dialog(onDismissRequest = onDismiss) {
        Surface(
            shape = RoundedCornerShape(8.dp),
            color = Color.White,
            modifier = Modifier.padding(16.dp)
```

```
) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = "Add/Update Task", fontSize = 18.sp, fontWeight = FontWeight.Bold)

        Spacer(modifier = Modifier.height(8.dp))

        OutlinedTextField(
            value = taskTitle,
            onValueChange = { taskTitle = it },
            label = { Text("Task Title") },
            modifier = Modifier.fillMaxWidth()
        )

        Spacer(modifier = Modifier.height(8.dp))

        OutlinedTextField(
            value = taskDescription,
            onValueChange = { taskDescription = it },
            label = { Text("Task Description") },
            modifier = Modifier.fillMaxWidth()
        )

        Spacer(modifier = Modifier.height(16.dp))

        Row(
            horizontalArrangement = Arrangement.End,
            modifier = Modifier.fillMaxWidth()
        ) {
            TextButton(onClick = onDismiss) {
                Text("Cancel")
            }

            Spacer(modifier = Modifier.width(8.dp))

            TextButton(
                onClick = {
                    if (taskTitle.isNotBlank() && taskDescription.isNotBlank()) {
                        if (task != null) {
                            onUpdate(Task(task.id, taskTitle, taskDescription, task.isDone))
                        } else {
                            onCreate(Task(title = taskTitle, description = taskDescription))
                        }
                    }
                }
            ) {
```

```
                    Text("Save")
                }
            }
        }
    }
}
```