Kazakh-British Technical University

Mobile Programming

Midterm

Building a Simple Task Management App in Android Using Kotlin

Full Name: Tamiris Abildayeva

ID: 23MD0503

Link to GitHub: https://github.com/TamirisK/university-mobile-programming

27.10.2024

Table of Contents

Executive Summary

The task management Android application aims to provide users with a seamless and efficient way to organize, manage, and track their tasks on mobile devices. The primary objective is to deliver a user-friendly interface that empowers individuals to create, modify, and delete tasks while offering essential features for categorization and prioritization.

This application is developed using Kotlin, leveraging Firebase as the backend for real-time database functionalities and user authentication. The use of Firebase enables secure data storage and seamless synchronization across devices, ensuring that users can access their task lists anytime, anywhere.

Outcomes Achieved:

1. Implemented secure user authentication processes, allowing users to create accounts and log in with robust security measures to protect their personal information.
2. Developed core task management features that enable users to add, edit, and delete tasks, including functionalities to specify task titles, descriptions, creation dates, and priority levels.
3. Designed an intuitive user interface that offers a clear overview of tasks, including visual indicators for task status (e.g., completed, pending) to enhance user experience.
4. Integrated categorization features, allow users to organize tasks into different groups such as “Work,” “Personal,” and “Other” for better organization and management.
5. Enabled real-time updates and synchronization across devices through Firebase, ensuring users have the most current information regardless of the device they are using.

Future upgrades are planned to include reminder notifications for deadlines, advanced filtering options for task views, and collaborative features that will allow users to share tasks with others, fostering teamwork and project management.

Introduction

In today's digital world, mobile applications are indispensable for enhancing productivity and convenience in our daily lives. As smartphones become increasingly integral to our routines, the demand for intuitive and effective applications continues to rise. Mobile apps empower users to tackle tasks on the go, providing immediate access to information and services that streamline workflows and elevate overall efficiency.

Kotlin has established itself as the leading language for modern Android development, thanks to its concise and expressive syntax that surpasses Java. Its seamless interoperability with existing Java code, coupled with strong support for functional programming features, positions it as the preferred choice for developers aiming to build robust and maintainable applications. Additionally, Kotlin's focus on safety—including null safety—effectively prevents common programming errors, significantly enhancing the stability of Android apps.

The drive to create a task management app arises from a clear need for improved organization in our fast-paced world. With countless responsibilities vying for attention, a dedicated tool for managing tasks is essential for maximizing productivity. This application is designed to provide users with an efficient platform to easily create, modify, and categorize tasks.

I am motivated to develop this task management application in Kotlin as a means to deepen my understanding of back-end development, an area I am keen to master alongside my front-end expertise. This project is an excellent opportunity for me to enhance my skills while aiming for a top grade on my midterm exam. With a current GPA of 3.9, I am dedicated to maintaining strong academic performance, where achieving at least a 90 is crucial. This project not only serves as a valuable learning experience but also aligns perfectly with my academic objectives, allowing me to create a practical tool for task management while advancing my technical capabilities.

Project Objectives

The task management application project has clearly defined objectives that will drive its success:

1. **Develop a Functional Mobile Application**: I will create a powerful and fully functional task management app for mobile devices. This app will enable users to efficiently create, edit, delete, and categorize tasks, all supported by an intuitive user interface designed for optimal task management.
2. **Master the Android Lifecycle**: I intend to gain a comprehensive understanding of the Android application lifecycle, focusing on activity and fragment management. This expertise is essential for ensuring seamless transitions, maintaining application state, and effectively persisting data to enhance user interactions.
3. **Leverage Kotlin Features:** I will take full advantage of the advanced features of the Kotlin programming language. This includes implementing coroutines to improve asynchronous programming, utilizing extension functions for better code structure, and applying null safety principles to eliminate common programming errors. My code will be clean, efficient, and maintainable, adhering to the highest development standards.
4. **Integrate Firebase for Data Storage**: I plan to integrate Firebase as the backend solution for secure storage of user tasks and preferences. This integration will allow for real-time data synchronization, giving users access to their information anytime and anywhere. Additionally, I will implement user authentication through Firebase to ensure robust security for personal data.
5. **Enhance User Experience (UX)**: I am committed to designing a user-friendly interface that significantly improves the overall user experience. By conducting thorough user research, I will identify needs and implement simple navigation, ensuring the app is accessible to users of all technical backgrounds.
6. **Implement Task Categorization and Prioritization**: I will develop features that allow users to categorize tasks based on criteria such as urgency, importance, and type (e.g., work, personal). Additionally, I will enable task prioritization, providing users with the tools they need to manage their responsibilities effectively and improve their time management skills.
7. **Test and Debug the Application**: I plan to conduct rigorous testing and debugging throughout the development process, including unit tests, integration tests, and user acceptance testing. My goal is to identify and resolve any technical issues, ensuring a stable, reliable, and high-quality end product that exceeds user expectations.
8. **Document the Development Process**: I will maintain comprehensive documentation covering all stages of development. This documentation will include design decisions, code snippets, challenges faced, and lessons learned. It will serve as a valuable resource for future improvements and contribute to my growth and expertise in mobile app development.

By confidently pursuing these objectives, I aim to deliver an impactful and efficient tool for task management while enhancing my skills in mobile development and backend integration. This project will not only achieve its goals but also lay the groundwork for more advanced projects in the future, fostering continuous personal and professional growth.

Overview of Android Development and Kotlin

Overview of Android Development

Android development involves creating applications for devices running the Android operating system using the Android SDK (Software Development Kit). While various programming languages can be used, Java and Kotlin are the most popular.

Development Environment

Android Studio, the official IDE for Android development, can be set up for Kotlin with these steps:

1. **Download and Install**: Visit the [Android Studio website](URL_1) and install the latest version for macOS.
2. **Install SDK**: The setup wizard will prompt need to install the necessary components, including the Android SDK and AVD.
3. **Create a New Project**: Select "New Project" and choose a template, specifying project details.
4. **Enable Kotlin**: During setup, choose "Include Kotlin support." For existing Java projects, add Kotlin via *File > New > Kotlin File/Class*.
5. **Run Application**: Use the built-in emulator or a physical device to test the application by clicking the "Run" button.

Overview of Kotlin

Kotlin, developed by JetBrains, is a modern programming language fully interoperable with Java, enhancing Android development through its features:

1. **Conciseness**: Reduces boilerplate code.
2. **Null Safety**: Helps prevent null pointer exceptions.
3. **Extension Functions**: Allows adding functionality to existing classes.
4. **Higher-Order Functions**: Supports functional programming by passing functions as parameters.
5. **Data Classes**: Automatically generates standard functions for classes intended to hold data.
6. **Coroutines**: Simplifies asynchronous programming.

Kotlin offers improved readability and brevity, null safety, full interoperability with Java, and modern programming features.

Kotlin enhances the efficiency and safety of Android development, allowing for the creation of powerful, user-friendly applications using Android Studio.

Functions and Lambdas in Kotlin

Kotlin stands out as a modern programming language with its concise syntax and powerful features, including first-class functions and lambda expressions. This section clearly outlines the essential function implementations utilized in this task management app and effectively demonstrates the use of lambdas for managing UI events and callbacks.

Function Implementation

In this task management app, I implement a range of functions that are crucial for task management activities. Below are key functions defined in the *TaskDashboardActivity* class, each serving a specific and important purpose:

1. **Adding a Task**: The *onEditTask* function serves as an interface for users to either create a new task or modify an existing one. It constructs an Intent directed at *AddTaskActivity*, populating it with the task's relevant details through putExtra. This method ensures that all necessary task attributes—such as title, description, deadline, and category—are passed to the new activity for seamless editing.

```
private fun onEditTask(task: Task) {
    val intent = Intent(this, AddTaskActivity::class.java).apply {
        putExtra("taskId", task.taskId)
        putExtra("taskTitle", task.title)
        putExtra("taskDescription", task.description)
        putExtra("taskDeadline", task.deadline) // Pass additional fields
        putExtra("taskCategory", task.category)
    }
    startActivityForResult(intent, EDIT_TASK_REQUEST_CODE)
}
```

Fig. 1: The *onEditTask* function.

This function creates an Intent to launch *AddTaskActivity*, passing necessary task details as extras. Using *startActivityForResult*, allows the app to receive a result back, such as confirmation of a task being added or updated.

2. **Retrieving Tasks**: The *loadTasks* function is responsible for querying the Firestore database to retrieve tasks associated with the currently authenticated user. It utilizes Firestore's querying capabilities to filter tasks based on the user's unique identifier, ensuring that users only see their tasks.

```
private fun loadTasks() {
            Log.d("TaskDashboardActivity",    "Loading    tasks    for    user:
${auth.currentUser?.uid}")

    val userId = auth.currentUser?.uid
    if (userId == null) {
                        Toast.makeText(this,    "User    not    authenticated",
Toast.LENGTH_SHORT).show()
        return
    }
```

```
    firestore.collection("tasks")
        .whereEqualTo("userId", userId)
        .get()
        .addOnSuccessListener { documents ->
            tasks.clear()
            for (document in documents) {
                val task = document.toObject(Task::class.java)
                tasks.add(task)
            }
            if (tasks.isEmpty()) {
                addDefaultTasks() // To check
            } else {
                taskAdapter.notifyDataSetChanged()
            }
        }
        .addOnFailureListener { e ->
                Toast.makeText(this, "Failed to load tasks: ${e.message}",
Toast.LENGTH_SHORT).show()
        }
}
```

Fig. 2: The *loadTasks* function.

This function employs asynchronous operations to fetch data from Firestore, ensuring that the UI remains responsive. It effectively handles both successful and failed data retrieval scenarios, updating the UI dynamically based on the fetched results, thus providing a smooth user experience.

3. **Deleting a Task**: The *onDeleteTask* function facilitates the deletion of a specific task from the Firestore database. It interacts with the Firestore API to remove the task document corresponding to the provided task ID.

```
private fun onDeleteTask(task: Task) {
    firestore.collection("tasks").document(task.taskId)
        .delete()
        .addOnSuccessListener {
            Toast.makeText(this, "Task deleted", Toast.LENGTH_SHORT).show()
            tasks.remove(task)
            taskAdapter.notifyDataSetChanged()
        }
        .addOnFailureListener { e ->
                Toast.makeText(this, "Failed to delete task: ${e.message}",
Toast.LENGTH_SHORT).show()
        }
}
```

Fig. 3: The *onDeleteTask* function.

Upon successful deletion, this function not only removes the task from the local list but also refreshes the task adapter to reflect the changes in the UI immediately. This two-way synchronization between the local state and Firestore ensures that users always see the most accurate data.

Using Lambdas

Kotlin's support for lambda expressions allows for clear and efficient management of callbacks and events. In my app, I harness lambdas specifically for UI event handling. Below is a prime example from the *TaskDashboardActivity*, where click listeners are defined with lambdas:

```
val profileButton: ImageButton = findViewById(R.id.profileButton)
profileButton.setOnClickListener {
    val intent = Intent(this, ProfileActivity::class.java)
    startActivity(intent)
    finish()
}
```

Fig. 4: The function to open new Activity.

In this example, both *setOnClickListener* methods utilize lambdas to clearly define the actions triggered by button clicks. This strategy enhances code readability and eliminates unnecessary boilerplate.

The implementation of functions and lambdas in Kotlin substantially elevates the code efficiency and readability of this task management app. Functions establish a modular framework, while lambdas streamline event handling, leading to a clean and maintainable codebase.

Object-Oriented Programming in Kotlin

Object-Oriented Programming (OOP) helps us organize this application code to make it reusable, easy to maintain, and structured. In this task management app, I use Kotlin's OOP features, especially classes and data classes.

Classes and Data Classes

In Kotlin, classes define how to create objects. The group data (properties) and actions (methods). In my task management app, I mainly use a data class to represent tasks.

```
package com.taskmanagement

data class Task(
   var title: String = "",
   var description: String = "",
   var createdDate: String = "",
   var deadline: String = "",
   var userId: String = "",
   var taskId: String = "",
   var category: String = ""
)
```

Fig. 4: Example of a Data Class: *Task*.

Explanation:

Properties: The *Task* data class holds all the important details about a task:

1. *taskId*: A unique ID for the task.
2. *title*: The name of the task.
3. *description*: More details about the task.
4. *userId*: The ID of the user who owns the task.
5. *createdDate*: When the task was created.
6. *deadline*: When the task is due.
7. *category*: The type of task (e.g., Work, Personal).

Methods: Data classes in Kotlin come with built-in methods like *toString(), hashCode()*, and *equals()*. This makes it easy to compare tasks or use them in lists without additional coding.

Using a data class helps us clearly represent tasks and work with them easily throughout the app.

Encapsulation

Encapsulation is an OOP principle that limits access to some parts of an object to protect its data. It is essential for keeping task data safe in this application.

Implementation in this App:

1. **Private Properties:**

   I can mark some properties as *private* in this data class and other classes to control access. This means they can only be accessed or changed through specific methods.

```
class TaskManager {
    private val tasks = mutableListOf<Task>()

    fun addTask(task: Task) {
        tasks.add(task)
    }

    fun getTasks(): List<Task> {
        return tasks.toList()
    }
}
```

Fig. 5: Realization of *TaskManager* class.

2. **Public Methods:**

   I make public methods like *addTask* and *getTasks* available to interact with the *tasks* list. This allows us to control how tasks are added and retrieved, ensuring that each action can include necessary checks.

3. **Modularity and Organization:**

   By grouping related properties and methods in classes, I keep project code organized. This makes it easier to read, maintain, and debug.

Example of Encapsulation in Use:

When a user adds a task through the interface, the task is created and added to the list using the *addTask* method. The list itself is not directly accessible. This way, I make sure tasks are only added through proper methods, keeping the data secure.

By using classes and data classes along with encapsulation, this task management app follows good OOP practices. This improves code organization and reusability and makes the app easier to maintain.

Working with Collections in Kotlin

Collections in Kotlin provide robust tools for managing groups of data, and facilitating operations such as storage, retrieval, and efficient manipulation of information. In the context of my task management application, I employ collections, specifically lists, to manage tasks effectively.

Collection Usage

In this application, I predominantly utilize *lists* to maintain a dynamic collection of tasks. A mutable list is particularly suitable for this purpose, as it allows for seamless additions and deletions of tasks in response to user interactions with the application.

```
private val tasks = mutableListOf<Task>()
```

Fig. 6: Example of a *Task List*.

In this instance, *tasks* is a mutable list designed to hold *Task* objects. This collection is modifiable throughout the application's lifecycle, enabling functionalities such as the addition of new tasks, the deletion of existing ones, and the display of tasks to the user.

Operations on Collections

Kotlin's comprehensive collection library offers a wide array of functions for executing various operations on lists. The following are common operations implemented in this task management application:

```
fun addTask(task: Task) {
    tasks.add(task)
}
```

Fig. 6: Example of an *Adding task*.

When a new task is generated, it is incorporated into the *tasks* list:

```
fun getTasksByCategory(category: String): List<Task> {
    return tasks.filter { it.category == category }
}
```

Fig. 7: Example of a *Filtering task*.

Filtering enables the retrieval of tasks based on specified criteria, such as category:

The *filter* function produces a new list containing only the tasks that correspond to the designated category. This capability is beneficial for presenting categorized views of tasks.

Sorting tasks is essential for organizing the user interface. For example, tasks can be sorted by their deadlines:

```
fun getSortedTasksByDeadline(): List<Task> {
   return tasks.sortedBy { it.deadline }
}
```

Fig. 8: Example of a *Sorting task*.

The *sortedBy* function yields a new list of tasks arranged in ascending order based on the deadline property. This operation enhances the user's ability to identify upcoming tasks.

When a user opts to delete a task, it can be removed from the list:

```
fun removeTask(task: Task) {
   tasks.remove(task)
}
```

Fig. 9: Example of a *Removing task*.

The *remove* function modifies the *tasks* list directly, eliminating the specified task. This operation is vital for sustaining an accurate representation of the remaining tasks.

By effectively leveraging Kotlin's collections, this task management application can proficiently manage and manipulate tasks. The ability to add, filter, sort, and transform collections significantly enhances user experience and ensures that the application remains responsive to user needs. These operations are fundamental to establishing a dynamic and intuitive task management environment.

Android Layout Design

The user interface of this task management application is developed using XML layout files, which provide a systematic framework for defining layout elements and their associated properties. This section elucidates the design of the application's user interface and evaluates the appropriateness of various layout types utilized within the application.

XML Layouts

In this application, XML layout files serve as the foundation for designing the user interface across different activities, including the main activity, sign-up, login, task dashboard, and profile screens. Each layout file delineates the UI elements, their arrangement, and styling properties in a declarative format.

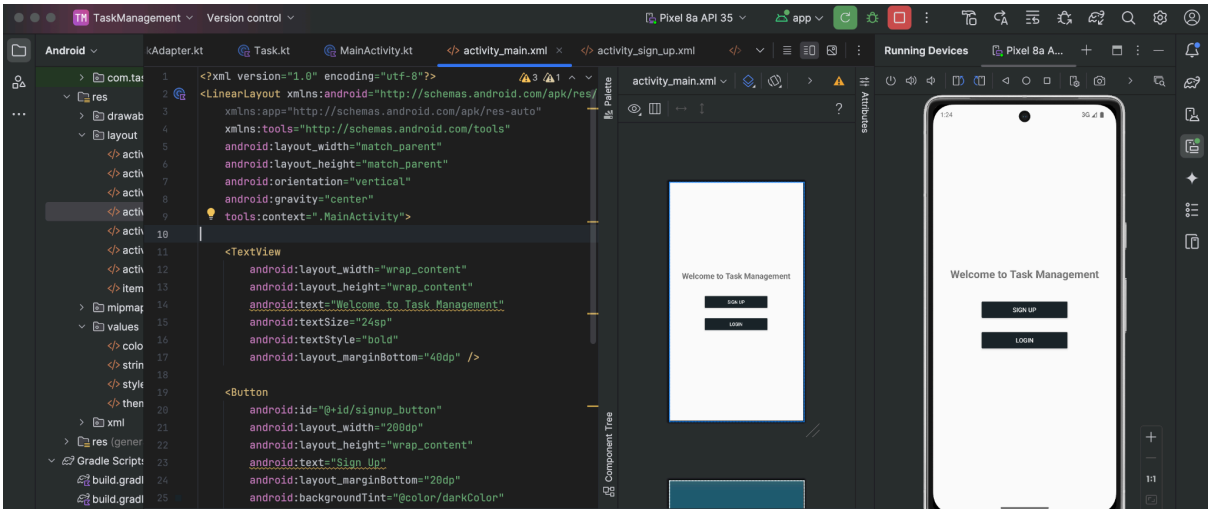

Fig. 9: Example of the *Main Activity Layout*.

This layout employs a *LinearLayout* with a vertical orientation, positioning the *TextView* and *Button* elements in a vertically aligned and centered manner. The use of *wrap_content* for both width and height ensures that elements occupy only the amount of space necessary.

Layout Types

The selection of layout types in this application is purposefully strategic, tailored to the specific requirements of each activity. Below are some of the layouts employed, along with their suitability for the application:

1. **LinearLayout:** This layout is applied in the main activity, sign-up, and login screens. The vertical orientation facilitates a straightforward arrangement of UI components in a stacked manner. This layout type is particularly effective for simple user interfaces where elements must be presented in a linear format.

2. **ConstraintLayout:** Implemented in the *TaskDashboardActivity* and the profile layout, ConstraintLayout offers enhanced flexibility in positioning UI elements. It supports more complex designs by defining constraints between elements, making it easier to create responsive layouts that adapt to various screen sizes.

```
      <androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
         android:layout_width="match_parent"
         android:layout_height="match_parent">

         <androidx.appcompat.widget.Toolbar
             android:id="@+id/toolbar"
             android:layout_width="match_parent"
             android:layout_height="wrap_content"
             app:title="Task Dashboard"
             app:titleTextColor="@android:color/white"
             android:background="?attr/colorPrimary" />

         <RecyclerView
             android:id="@+id/taskRecyclerView"
             android:layout_width="match_parent"
             android:layout_height="0dp"
             app:layout_constraintTop_toBottomOf="@id/toolbar"
             app:layout_constraintBottom_toTopOf="@id/addTaskButton" />


<com.google.android.material.floatingactionbutton.FloatingActionButton
             android:id="@+id/addTaskButton"
             android:layout_width="wrap_content"
             android:layout_height="wrap_content"
             android:src="@drawable/baseline_add_24"
             android:contentDescription="Add Task"
             app:layout_constraintBottom_toBottomOf="parent"
             app:layout_constraintEnd_toEndOf="parent" />
      </androidx.constraintlayout.widget.ConstraintLayout>
```

Fig. 10: Example of the *ConstraintLayout*.

This layout organizes the toolbar, *RecyclerView*, and *FloatingActionButton* using constraints, which facilitates a responsive design. The *RecyclerView* dynamically adjusts its height based on available space, thereby ensuring a seamless user experience.

3. **CardView**: Utilized in the sign-up and login layouts, CardView delivers a material design aesthetic, enabling UI elements to stand out. It enhances visual appeal while ensuring a clear separation between distinct sections of the user interface.

```
<androidx.cardview.widget.CardView
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:layout_margin="30dp"
   app:cardCornerRadius="30dp"
   app:cardElevation="20dp">
</androidx.cardview.widget.CardView>
```

Fig. 10: Example of the *CardView*.

CardView encapsulates UI elements and provides a rounded corner appearance with elevation, thereby creating a distinct and refined look that improves user interaction.

The XML layout design in this task management application effectively employs a diverse array of layout types tailored to the specific needs of each screen. By utilizing LinearLayouts for straightforward configurations and ConstraintLayouts for more intricate user interfaces, the application maintains a clean and user-centered design. The incorporation of CardViews further enhances the visual appeal, contributing to an overall polished user experience.

Activity and User Input Handling

In the context of this task management application, activities are fundamental for managing user interactions and conveying information. This section delineates the primary activity responsible for displaying tasks and elucidates the process by which user inputs are collected and processed.

Activity Creation

The main activity functions as the central interface of the application, enabling users to view, add, and manage their tasks effectively. It is designed to present a list of tasks in an intuitive manner, employing a RecyclerView for dynamic task display.

Key Components of the Main Activity:

1. *Toolbar*: A toolbar positioned at the top provides navigation options and branding elements.
2. *RecyclerView*: This component showcases a list of tasks, allowing users to scroll through them efficiently. Each task is represented as an individual item within the RecyclerView, thereby providing an overview of task titles and statuses.
3. *Floating Action Button (FAB)*: A prominently featured button for adding new tasks, which enhances user engagement and accessibility.

```
class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       enableEdgeToEdge()

       setContentView(R.layout.activity_main)

       val signupButton = findViewById<Button>(R.id.signup_button)
       val loginButton = findViewById<Button>(R.id.login_button)

       signupButton.setOnClickListener {
                           Toast.makeText(this,  "Sign  Up  button  clicked",
Toast.LENGTH_SHORT).show()
           Log.d("MainActivity", "Sign Up button clicked")
           val intent = Intent(this, SignupActivity::class.java)
           startActivity(intent)
       }
```

Fig. 11: Example of the *MainActivity*.

The *onCreate* method initializes the RecyclerView and its adapter, establishes the layout, and manages user interactions via the FAB. When the FAB is selected, it initiates an intent to navigate to the activity designated for adding new tasks.

User Input Handling

User input management is a vital component of the task management application, particularly with regard to the collection of task details such as title, description, and due date. The application facilitates user input through specifically designated fields within a separate activity.

Collecting User Input:

1. *Input Fields*: In the AddTaskActivity, users may input the task title, description, and any additional details utilizing EditText fields.

```
      <?xml version="1.0" encoding="utf-8"?>
      <LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         android:orientation="vertical"
         android:padding="16dp">

         <EditText
             android:id="@+id/titleEditText"
             android:layout_width="match_parent"
             android:layout_height="wrap_content"
             android:hint="Task Title" />

         <EditText
             android:id="@+id/descriptionEditText"
             android:layout_width="match_parent"
             android:layout_height="wrap_content"
             android:hint="Task Description" />

      // AND ANOTHER FIELDS

         <Button
             android:id="@+id/saveButton"
             android:layout_width="wrap_content"
             android:layout_height="wrap_content"
             android:text="Save Task" />
      </LinearLayout>
```

Fig. 13: Example of the *Add Task Layou*t.

Upon completion of the designated fields, users may click the "Save Task" button, prompting the collection and processing of input data in the *AddTaskActivity*.

```
      class AddTaskActivity : AppCompatActivity() {

         private val firestore = FirebaseFirestore.getInstance()
         private var isEditMode = false
         private var taskId: String? = null

         override fun onCreate(savedInstanceState: Bundle?) {
             super.onCreate(savedInstanceState)
             setContentView(R.layout.activity_add_task)

             val titleEditText: EditText = findViewById(R.id.titleEditText)
             val descriptionEditText: EditText =
findViewById(R.id.descriptionEditText)
             val deadlineEditText: EditText =
findViewById(R.id.deadlineEditText)
             val categorySpinner: Spinner =
findViewById(R.id.categorySpinner)
             val saveButton: Button = findViewById(R.id.saveButton)

             val categories = arrayOf("Work", "Personal", "Shopping",
"Other")
             val adapter = ArrayAdapter(this,
android.R.layout.simple_spinner_item, categories)

      // THERE ARE SOME CODE

             saveButton.setOnClickListener {
                 val title = titleEditText.text.toString().trim()
                 val description =
descriptionEditText.text.toString().trim()
                 val deadline = deadlineEditText.text.toString().trim()
                 val selectedCategory =
categorySpinner.selectedItem.toString()

                 if (title.isNotEmpty()) {
                     if (isEditMode && taskId != null) {
                         updateTaskInFirestore(taskId!!, title, description,
deadline, selectedCategory)
                     } else {
                         val task = Task(
                             title = title,
                             description = description,
                             createdDate =
System.currentTimeMillis().toString(),
                             deadline = deadline,
                             category = selectedCategory
                         )
                         saveTaskToFirestore(task)
                     }
                 } else {
                     Toast.makeText(this, "Title is required",
Toast.LENGTH_SHORT).show()
                 }
             }
         }
```

Fig. 13: Example of the *Input Handling Logic*.

The *setOnClickListener* for the save button captures the values entered into the EditText fields. Subsequently, when the user selects "Save Task," the application processes the input (for example, through database storage) and navigates back to the main activity.

The main activity serves as the primary interface for task management, presenting tasks within a RecyclerView and enabling user input through a distinct activity. By effectively handling user inputs, the application ensures a seamless and engaging experience, empowering users to create and manage their tasks with efficiency.

Activity Lifecycle Management

Understanding the activity lifecycle is fundamental for the effective management of an Android application's state. Lifecycle methods enable developers to appropriately respond to events such as the creation, pausing, resuming, and destruction of activities, ensuring that the application reacts suitably to user interactions and system events.

Lifecycle Methods

The Android activity lifecycle consists of a series of states that an activity undergoes, defined by a specific set of lifecycle methods. Each method corresponds to a particular event in the activity's lifecycle:

1. **onCreate()**: This method is invoked when the activity is first created. It is essential for initializing the activity, setting up the user interface, and preparing any necessary data.
2. **onStart()**: This method is called when the activity becomes visible to the user. It is advisable to initiate animations and other visual elements at this stage.
3. **onResume()**: This method is triggered when the activity begins interacting with the user. It is the appropriate time to resume any paused tasks or processes.
4. **onPause()**: This method is invoked when the system is about to resume another activity. It is important to pause ongoing tasks that do not need to continue while the activity is not in the foreground.
5. **onStop()**: This method is called when the activity is no longer visible to the user. It is a suitable moment to release resources or save data.
6. **onDestroy()**: This method is executed before the activity's destruction. It serves as the final call received before the activity is terminated.

These methods are imperative for managing the application's state as they facilitate the efficient handling of resources and contribute to a seamless user experience.

Example Implementation

The following illustrates how to override several lifecycle methods within an activity to manage state effectively:

```
lass MainActivity : AppCompatActivity() {
   private var isUserLoggedIn: Boolean = false

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_main)
```

…

```
override fun onStart() {
   super.onStart()
}
```

…

```
override fun onSaveInstanceState(outState: Bundle) {
   super.onSaveInstanceState(outState)
```

```
    // Preserve the state of the activity
    outState.putBoolean("user_logged_in", isUserLoggedIn)
}
```

Fig. 13: Example of the *implementation*.

1. **onCreate()**: Initializes the activity and restores any available saved state.
2. **onStart()**: Prepares any necessary processes that should be active when the activity is visible.
3. **onResume()**: Resumes operations and updates the user interface depending on the user's logged-in status.
4. **onPause()**: Suspends tasks that should not continue while the activity is not in the foreground.
5. **onStop()**: Saves state or releases resources once the activity is no longer visible.
6. **onDestroy()**: Manages final cleanup operations.
7. **onSaveInstanceState()**: Saves the activity's state, which can be restored in the *onCreate()* method.

Managing the activity lifecycle is essential for ensuring that the application functions as intended, preserves user data, and makes optimal use of system resources. By overriding lifecycle methods, developers can cultivate a responsive and efficient user experience in their Android applications.

Fragments and Fragment Lifecycle

Fragment Creation

In the realm of Android development, fragments serve as modular sections of activity, facilitating more dynamic and adaptable user interface (UI) designs. Within the context of a task management application, fragments can be employed to represent various views or functionalities, such as displaying a list of tasks, adding new tasks, or viewing task details.

To create a fragment, developers typically subclass the *Fragment* class and override its lifecycle methods (such as *onCreateView*, *onCreate*, etc.) to define the fragment's behavior and UI elements. For example, one may implement a *TaskListFragment* to showcase a list of tasks in a *RecyclerView*, while an *AddTaskFragment* could be utilized to add new tasks.

Fragments are integrated into the activity using a *FragmentManager*, which effectively manages the addition, removal, and replacement of fragments within the activity's layout. This methodology enhances code reusability and permits diverse UI configurations depending on the screen size of the device.

Fragment Lifecycle

The lifecycle of a fragment is intricately linked to that of its parent activity. A comprehensive understanding of the fragment lifecycle is essential for the effective management of UI components and resources. The principal lifecycle methods of a fragment include:

1. **onAttach**: This method is invoked when the fragment is initially attached to its context. It enables the fragment to establish references to the parent activity.
2. **onCreate**: This method is called for one-time initializations, such as setting up data or constructing views that do not require any UI elements.
3. **onCreateView**: In this method, the fragment’s layout is inflated and UI components are initialized. The returned `View` is what is rendered on the screen.
4. **onActivityCreated**: This method is executed following `onCreateView`, and it is often utilized for final initializations that necessitate the activity being fully created, such as the establishment of the `ViewModel` or the retrieval of activity data.
5. **onStart()**: This method is activated when the fragment becomes visible to the user, serving as an appropriate juncture for initiating animations or updates requiring user interaction.
6. **onResume()**: This method indicates that the fragment is now in the foreground and can engage with the user, making it a suitable moment to start processes that require user focus.

7. **onPause()**: This method is invoked when the fragment no longer interacts with the user. It serves as an opportunity to pause ongoing activities, such as animations or network requests.
8. **onStop()**: This method signifies that the fragment is no longer visible. It is an opportune time to release resources or preserve relevant data.
9. **onDestroyView()**: This method is called when the fragment's view is being destroyed, allowing developers to clean up any references to UI components and mitigate the risk of memory leaks.
10. **onDetach()**: This final callback indicates that the fragment is no longer attached to the activity, at which point any necessary cleanup activities should be performed.

Throughout its lifecycle, a fragment may interact with the activity via callbacks or interfaces, facilitating seamless exchange of data and coordination of UI updates. By leveraging fragments, a task management application can deliver a more modular and responsive user experience across a variety of device configurations.

Conclusion

The task management application created with Kotlin and Firebase meets its main goals, giving users an easy and effective way to handle tasks. By using modern technology and best practices in Android development, the app boosts user productivity and engagement.

The project has achieved several key successes:

1. **User Authentication**: I have implemented a secure login, protecting users' personal information. Firebase Authentication makes it easy to manage accounts.
2. **Core Task Management Features**: The app allows users to add, edit, and delete tasks, making it simple to organize and prioritize their responsibilities.
3. **Intuitive User Interface**: The app's design is user-friendly, with clear visual indicators for task status. This improves the overall user experience.
4. **Real-Time Synchronization**: Firebase's real-time database updates tasks immediately across devices. Users always have access to the latest task information, which is important for managing tasks on the go.

The chosen technologies have helped us achieve these results. Kotlin's clear syntax has made development easier and created a more stable app. Firebase provides a reliable backend for updates and user authentication.

While this version is strong, I can make several enhancements to improve functionality and user experience:

1. **Reminder Notifications**: Adding a notification system for upcoming deadlines can help users stay organized and productive.
2. **Advanced Filtering and Sorting**: Implementing filters for tasks by category, priority, or due date will help users navigate and manage tasks easily.
3. **Collaboration Features**: Enabling sharing capabilities will allow users to collaborate effectively, making the app useful for team projects
4. **Offline Functionality**: Allowing the app to work offline and sync data later will help in areas with low internet connectivity.
5. **User Feedback and Analytics:** Adding feedback options and tracking usage will help us improve future updates based on user needs.

In summary, this task management application uses Kotlin and Firebase to create a valuable tool for productivity. With future enhancements, the app can keep growing and remain a trusted companion for users managing their tasks.

Appendices

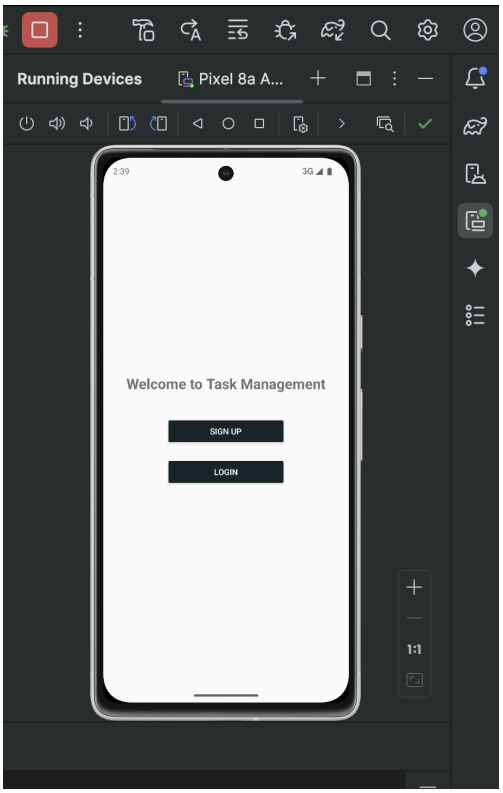

Fig. 14: Welcome page interface.

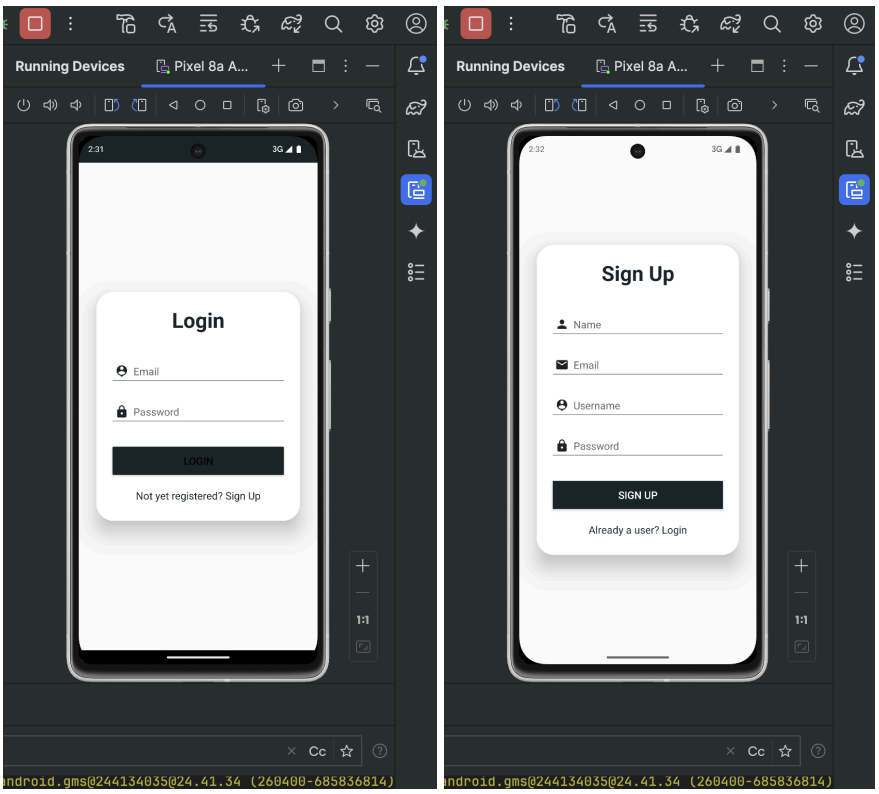

Fig. 15: Example of the *login & signup pages*.

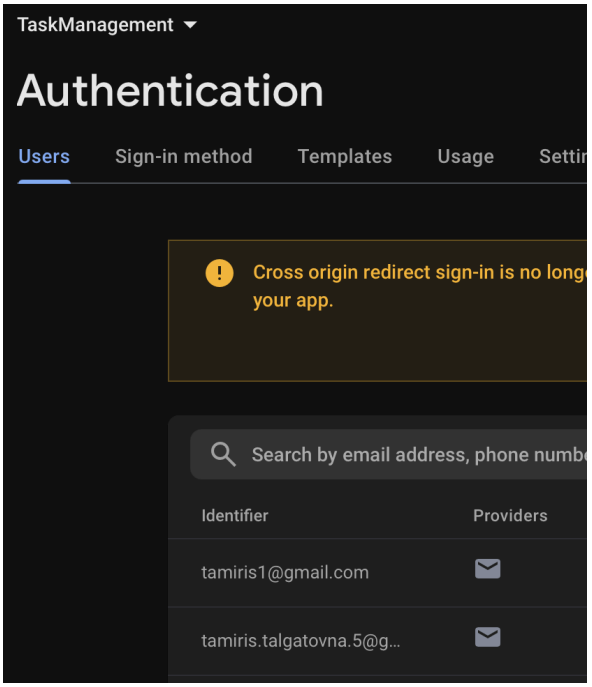

Fig. 16: Example of the *Firebase* database users.

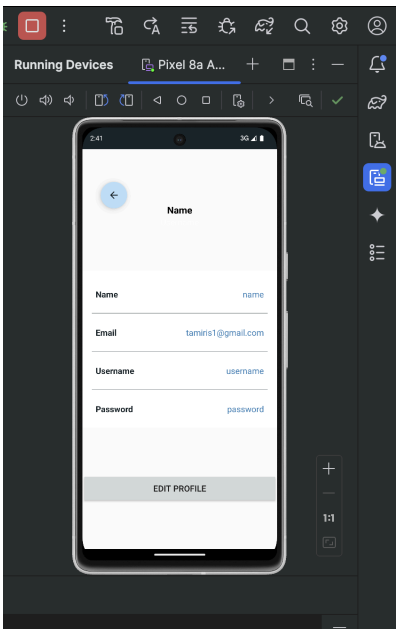

Fig. 17: Profile page interface.