



**KAZAKH-BRITISH
TECHNICAL
UNIVERSITY**

Midterm

Mobile Programming
Building a Simple Task Management App
in Android Using Kotlin

Prepared by:
Seitbekov S.
Checked by:
Serek A.

Almaty, 26.10.2024

Table of Contents

| | |
|---|----|
| Executive Summary | 3 |
| Introduction | 3 |
| Project Objectives | 4 |
| Overview of Android Development and Kotlin..... | 4 |
| Functions and Lambdas in Kotlin | 7 |
| Object-Oriented Programming in Kotlin | 9 |
| Working with Collections in Kotlin..... | 10 |
| Android UI Design with Jetpack Compose | 12 |
| Activity and User Input Handling..... | 14 |
| Activity Lifecycle Management | 15 |
| Composables and State Management in Jetpack Compose | 16 |
| Implementing Login Logic and Data Persistence with Room | 18 |
| Conclusion | 21 |
| References..... | 22 |
| Appendices..... | 23 |

Executive Summary

This report presents the development of a task management application in Android using Kotlin and Jetpack Compose. It is expected that users will have seamless workflow while managing their day-to-day activities: operate sign-up, authentication, create tasks, update, mark them completed and also view and delete them.

Best practices of modern Android development will be the components making up the current project with the expressive language features of Kotlin and the declarative UI paradigm of Jetpack Compose. Room is used for persistence in-app to reliably store data and tasks from users persisting across sessions.

The given development considers the following principles in mind: object-oriented programming, effective state management, and efficient handling of collections. This project focuses on user input treatment, activity lifecycle management, ViewModel integration combined with LiveData for reactive updates of a user interface in general.

In summary, this report will be an extended guide to understand how a modern Android application development goes about in terms of using the latest technologies along with best practices in place.

Introduction

Today, living in the era of smartphones, mobile applications have become key tools in helping organize certain parts of our lives. Among these, task management like ToDo applications play a considerable role in arranging one's schedule and reminding one through this to increase productivity.

Kotlin turned out to be the standard for Android application development due to its concise syntax, extra safety features, and seamless interoperability with Java. Its modern language constructs make maintainability much easier and reduce runtime errors.

Jetpack Compose is a big shift in developing Android UIs; instead of the imperative pattern, it brought a declarative pattern where the developer would define their interface in composable functions. That design really reduced boilerplate code and allowed for building more responsive applications by automatically re-composing your UI when your app's state changes.

This project has been undertaken in capturing this advancement in Android development. The project is aimed at showcasing such technologies, Kotlin, and Jetpack Compose-just how technologies can be used to solve practical problems; hence, the development of a task management application. Besides providing the essential features for task management, this application provides user authentication and data persistence, making it personalized and reliable.

Development process provides insight into setting up the environment, designing the architecture of an application, implementing a UI, and handling user interactions. It also covers various problems in state management, navigation, and lifecycle handling using the latest tools and libraries provided by Android's Jetpack suite.

This report is a review of the stages involved in the project, from methodologies put into place through every step, to the logic that inspired any design decisions taken, and what the outcome of such a decision was. The report shall be helpful in analyzing just how complex modern Android development is and how Kotlin and Jetpack Compose can serve efficiently in crafting solid mobile applications.

Project Objectives

The objective of the project is not to just build a task management application but develop a good looking, production-ready solution with proper design and features. Key goals for application are listed here:

1. **Functional Task Management Application:** It refers to the designing of an Android application through which one can create, read, update, and delete tasks in the most user-friendly manner possible.
2. **User Authentication:** Capability of the application to let a new user sign up and allow the user to log in to profile. This would keep the data separate and distinguished among different users.
3. **Room data persistence:** the Room Android database library to persist user credentials and tasks data locally. This is so that data persists between sessions and restarts of the application.
4. **Advanced Kotlin Features:** Take advantage of functions, lambdas, and object-oriented features in Kotlin to bring into effect clean, efficient, and maintainable code.
5. **Modern UI with Jetpack Compose:** The UI of the application will be designed using Jetpack Compose and Material3. Replace traditional XML layouts with declarative composable functions.
6. **User Input and Events Handling:** The application shall respond to user input, and it should be intuitive to such an extent as to make the user feel free to use it without problems.
7. **Activity Lifecycle:** Understanding of an extensive activity life cycle of Android and, therefore, implementing respective lifecycle methods when necessary.
8. **State Management and Navigation:** The app is using the View Model, LiveData for the state management of the app, and Jetpack Compose Navigation for smooth screen navigation.
9. **Code Quality and Maintainability:** Following the best practices in coding standards, best practices, and modularization means the codebase is maintainable.

Application provides the essential CRUD operations, besides a modern user experience, relying on advanced Kotlin features, Jetpack Compose UI, and lifecycle-aware components. This application will securely authenticate, persist data locally with Room, and follow best practices to ensure a reliable and maintainable solution for personal task management.

Overview of Android Development and Kotlin

The task management application was designed to be an easy, responsive, and user-friendly application for its users. The main programming language used in this project is Kotlin, with Jetpack Compose, which is the modern UI toolkit for Android. The approach was toward delivering a modular, maintainable, and scalable application by following best practices in addition to using the modern development tools of Android.

For development I use Android Studio. This IDE, Android Studio Ladybug (2024.2.1), covers all toolsets used in coding, debugging, and performance analysis. The choice of Kotlin as the major language is due to its concise syntax and modern features. Moreover, the latest Android SDKs have been installed to keep the program backward compatible with newer versions of Android.

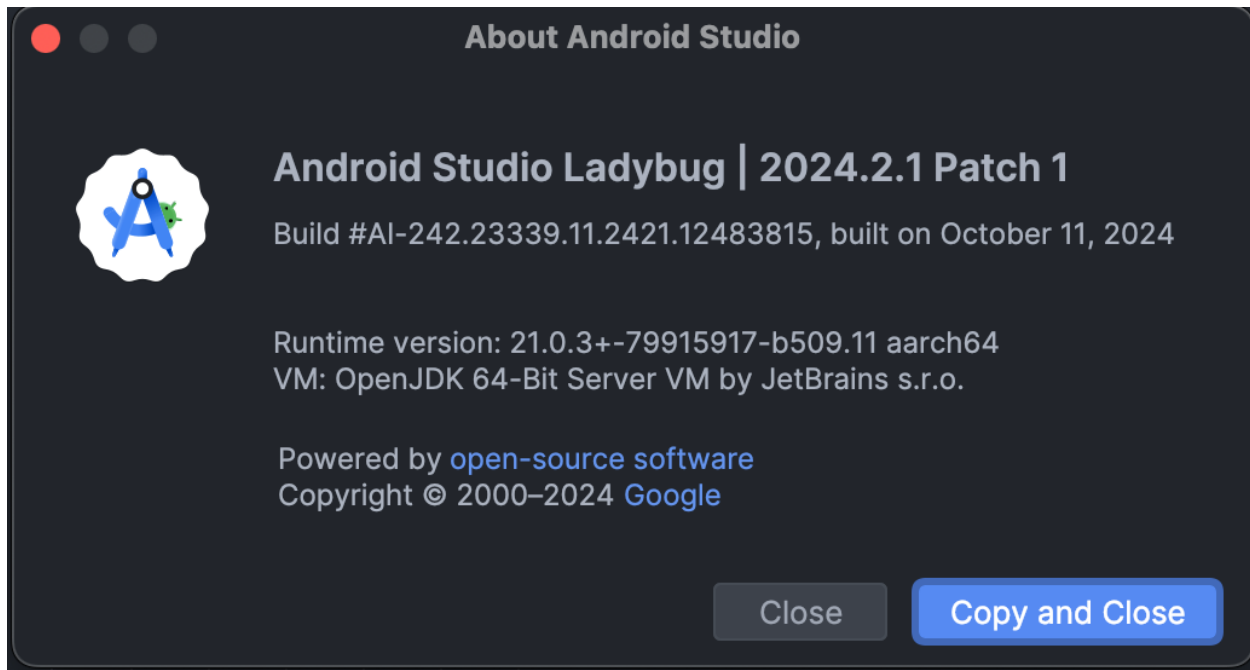


Figure 1. Android Studio Build Info.

Jetpack libraries were integrated into the project: Jetpack Compose for UI, Room for local database management, and lifecycle components to handle state. Gradle is the build automation system that managed dependencies and allowed seamless integration of libraries. The application was tested on emulators and physical devices to guarantee consistency and reliability on different platforms and hardware.

```
43 dependencies {
44     val room_version = "2.6.1"
45
46     implementation(libs.androidx.core.ktx)
47     implementation(libs.androidx.lifecycle.runtime.ktx)
48     implementation(libs.androidx.activity.compose)
49     implementation(platform(libs.androidx.compose.bom))
50     implementation(libs.androidx.ui)
51     implementation(libs.androidx.ui.graphics)
52     implementation(libs.androidx.ui.tooling.preview)
53     implementation(libs.androidx.material3)
54     implementation(libs.androidx.lifecycle.livedata.ktx)
55     implementation(libs.androidx.runtime.livedata)
56     testImplementation(libs.junit)
57     androidTestImplementation(libs.androidx.junit)
58     androidTestImplementation(libs.androidx.espresso.core)
59     androidTestImplementation(platform(libs.androidx.compose.bom))
60     androidTestImplementation(libs.androidx.ui.test.junit4)
61     debugImplementation(libs.androidx.ui.tooling)
62     debugImplementation(libs.androidx.ui.test.manifest)
63     implementation(libs.androidx.lifecycle.viewmodel.compose)
64     implementation(libs.androidx.navigation.compose)
65     implementation(libs.coil.compose)
66     implementation("androidx.room:room-runtime:$room_version")
67     annotationProcessor("androidx.room:room-compiler:$room_version")
68     ksp("androidx.room:room-compiler:$room_version")
69     implementation("androidx.room:room-ktx:$room_version")
70     implementation(libs.androidx.material.icons.extended)
71
72 }
```

Figure 2. Gradle Build Dependencies.

Kotlin was chosen because it features modern design paradigms, null safety, functional language support, and interoperability with Java, among other reasons. The syntax provides a lot of reduction in boilerplate code, making the code cleaner and more readable. Null safety features protect against common runtime crashes due to null pointer exceptions. Functional programming features like higher-order functions and lambda expressions make it flexible and enable concise and expressive code.

```
1 // Top-level build file where you can add configuration options common to all sub-projects/modules.
2 plugins {
3     id("com.android.application") version "8.7.1" apply false
4     id("org.jetbrains.kotlin.android") version "2.0.21" apply false
5     alias(libs.plugins.kotlin.compose) apply false
6     id("com.google.devtools.ksp") version "2.0.21-1.0.26" apply false
7 }
```

Figure 2. Gradle Build Dependencies.

Jetpack Compose was used for the development of the UI of this application. It introduces this declarative approach where the developer focuses on what should be in the UI given the current state, not how to construct it. The toolkit makes it easier to create user interfaces, doing away with XML layouts, and reducing code with Composable functions. These make the UI much modular and reusable; with tight integration of state management in Compose, any change in data automatically updates the UI. This improves the overall quality and reliability of the app.

```
12 @Composable
13 fun BottomNavBar(navController: NavController) {
14     val items = listOf(
15         BottomNavItem( label: "Tasks", route: "taskList", Icons.AutoMirrored.Filled.List),
16         BottomNavItem( label: "Profile", route: "profile", Icons.Default.Person)
17     )
18     NavigationBar {
19         val navBackStackEntry by navController.currentBackStackEntryAsState()
20         val currentRoute = navBackStackEntry?.destination?.route
21
22         items.forEach { item →
23             NavigationBarItem(
24                 label = { Text(item.label) },
25                 icon = { Icon(item.icon, contentDescription = item.label) },
26                 selected = currentRoute == item.route,
27                 onClick = {
28                     navController.navigate(item.route) {
29                         popUpTo(navController.graph.startDestinationId) {
30                             saveState = true
31                         }
32                         launchSingleTop = true
33                         restoreState = true
34                     }
35                 }
36             )
37         }
38     }
39 }
```

Figure 3. Example of Jetpack Compose function via @Composable decorator.

Functions and Lambdas in Kotlin

Function and lambda support are significant in Kotlin, helping to write concise, expressive, and maintainable code. Kotlin functions are utilized here for encapsulation of logic for reusability, while lambdas will be very helpful in event handling and higher-order functions elegantly, making the flow of an application smooth and dynamic.

Functions organize sequences of code in logical sets and allow for reuse, making code easier to read. Therefore, functions in the to-do list application encapsulate some of the actions, thus keeping the codebase clean and easy to maintain.

```
23
24      fun addTask(task: Task) = viewModelScope.launch {
25      ||      taskDao.insertTask(task)
26      }
27
```

Figure 4. Adding a Task in ViewModel.

Here I show example of functional usage of coroutines. `viewModelScope.launch` is used here to asynchronously perform the database operations so that operation will not block the main UI thread and thus impede the responsiveness of an app. Also, `addTask` function encapsulates the logic of adding a task, simplifying the ViewModel itself.

Lambdas are anonymous functions used throughout the app to handle user interactions and to define UI behaviors, particularly in Jetpack Compose components. A typical case would be using a lambda expression in the Button composable. The `onClick` parameter takes in a lambda that defines what should happen when the button is clicked.

```
Button(
    onClick = {
        val newTask = Task(
            username = currentUser?.username ?: "",
            title = title,
            description = description
        )
        taskViewModel.addTask(newTask)
        onBackPressed()
    },
    modifier = Modifier.fillMaxWidth(),
    enabled = title.isNotBlank() && description.isNotBlank()
) {
    Text(text: "Add Task")
}
```

Figure 5. Lambda function example.

This approach allows inline definition of the behavior of the button, such as adding a new task and calling a navigation function `onNavigateBack`, showing how lambdas might seamlessly control app flow.

```
16 @Composable
17 fun TaskListScreen(
18     taskViewModel: TaskViewModel,
19     authViewModel: AuthViewModel,
20     onAddTask: () → Unit,
21     onTaskClick: (Int) → Unit
22 ) {
23     val tasks by taskViewModel.tasks.observeAsState(emptyList())
24
25     Scaffold(
26         topBar = {
27             TopAppBar(
28                 title = { Text(text: "To-Do List") },
29                 colors = TopAppBarDefaults.topAppBarColors(
30                     containerColor = MaterialTheme.colorScheme.primaryContainer
31                 )
32             )
33         },
34         floatingActionButton = {
35             FloatingActionButton(
36                 onClick = onAddTask,
37                 containerColor = MaterialTheme.colorScheme.primary
38             ) {
```

Figure 5. Higher order function example.

Jetpack Compose also makes heavy use of higher-order functions: `TaskListScreen` composable takes the `onAddTask` and `onTaskClick` parameters, which let the caller define what should happen when such user interactions happen. This decouples the UI logic from any navigation or business logic, which is good for reusability.

```
LazyColumn(
    contentPadding = paddingValues,
    modifier = Modifier.fillMaxSize()
) {
    items(tasks.size) { index →
        val task = tasks[index]
        TaskItem(task = task, onTaskClick = onTaskClick)
        HorizontalDivider()
    }
}
```

Figure 6. Handling Collections with lambda function example.

The items function in LazyColumn takes a lambda that defines how every item is to be rendered, hence optimizing the UI to perform and render only the visible items on the screen. Therefore, it is ideal for handling large lists. The composable TaskItem is reusable to display different data of tasks in various parts of the app. That is good modularity and code reusability.

Functions and lambdas, make Kotlin more expressive and readable, especially in Jetpack Compose. Functions encapsulate complex logic that can be reused, while lambdas define behavior inline in a concise way, modularizing the code. More importantly, with the use of higher-order functions, developers have the capability to pass behavior as parameters, enabling the decoupling of UI logic from business logic.

Object-Oriented Programming in Kotlin

The project needed to be structured as scalable and maintainable, so important principles of OOP were followed. Classes, data classes, encapsulation, and inheritance were applied in Kotlin for the modeling of real entities so that application logic could be handled efficiently.

Data classes, for instance, like Task and User, had been created to represent entities in the application:

```
3      import androidx.room.Entity
4      import androidx.room.PrimaryKey
5
6      @Entity(tableName = "tasks")
7      data class Task(
8          @PrimaryKey(autoGenerate = true) val id: Int = 0,
9          val username: String,
10         val title: String,
11         val description: String,
12         val isCompleted: Boolean = false
13     )
```

Figure 7. Task Data Class.

The @Entity annotation indicates that this class is a table in the Room database. The Task data class models necessary data that will be used to manage tasks including to username, title, description, and isCompleted status.

```
10     class TaskViewModel(application: Application) : AndroidViewModel(application) {
11         private val taskDao = AppDatabase.getDatabase(application).taskDao()
12
13         private val _tasks = MutableLiveData<List<Task>>()
14         val tasks: LiveData<List<Task>>get() = _tasks
15     }
```

Figure 8. Encapsulation Example.

Encapsulation here is achieved using private properties exposing them through public getters. In the TaskViewModel, the `_tasks` `MutableLiveData` is private - hence not directly modifiable from outside of the class. It's exposed as an immutable `LiveData` through the public task's property.

```
9      class AuthViewModel(application: Application) : AndroidViewModel(application) {
10          private val userDao = AppDatabase.getDatabase(application).userDao()
11
12          private val _isAuthenticated = MutableLiveData<Boolean>(value: false)
13          val isAuthenticated: LiveData<Boolean> get() = _isAuthenticated
14
15          private val _currentUser = MutableLiveData<User?>()
```

Figure 9. Inheritance Example.

Inheritance was applied to extend class functionalities. `AuthViewModel` is extended from `AndroidViewModel`; hence, it becomes lifecycle aware and can have an application context. That makes `ViewModel` survive configuration changes and manage its data accordingly.

The principles of OOP could be applied by using Kotlin, which allowed for a neat, modular approach toward the application. It modeled the entities of an application through classes and data classes. Encapsulation ensured that the integrity of data and logic was maintained by not allowing any unwanted interference in data through external elements. Inheritance allowed code reusability; furthermore, class capabilities were enhanced through extension of their functionality.

Working with Collections in Kotlin

Kotlin collections are the idiomatic way to manage groups of data. The standard library extends collections by a rich set of functions for manipulating collections in an expressive and safe way.

App itself used collections extensively to store and show tasks. Practical example is filtering tasks that are incomplete to be shown in the app as the separated section. It was implemented as:

```
20      @OptIn(ExperimentalMaterial3Api::class)
21      @Composable
22      fun TaskSummaryScreen(
23          taskViewModel: TaskViewModel,
24          onNavigateBack: () → Unit
25      ) {
26          val tasks by taskViewModel.tasks.observeAsState(emptyList())
27
28          val sortedIncompleteTaskTitles = tasks
29              .filter { !it.isCompleted }
30              .sortedBy { it.title }
31              .map { it.title }
```

Figure 10. Filter and Sort Incomplete Tasks Example.

This filtering and sorting of tasks into incompleteTaskTitles that showed in different section, further enhancing the user experience by making it crystal clear which tasks were done, and which were pending.

```
64 Text(  
65     text = "Incomplete Tasks",  
66     style = MaterialTheme.typography.titleMedium,  
67     modifier = Modifier.padding(vertical = 8.dp)  
68 )  
69 LazyColumn {  
70     items(sortedIncompleteTaskTitles) { title →  
71         TaskTitleItem(title = title, isCompleted = false)  
72     }  
73 }  
74
```

Figure 11. Mapping of Title from sortedIncompleteTaskTitles Example.

Another application of collection functions was mapping tasks to extract specific information, like, for example, a list of task titles for a summary feature. It does this by filtering the tasks into completedTasks and incompleteTasks, which then can be listed separately for a user tasks summary.

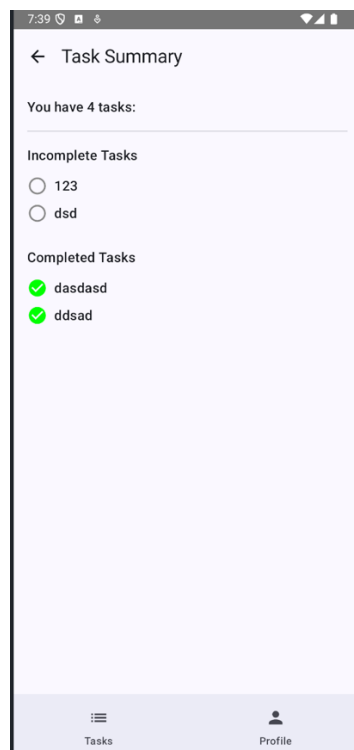


Figure 12. Task Summary Screen.

In the application we incorporated TaskSummaryScreen to display this list of task titles. It allowed users to visualize an overview of all their tasks.

Kotlin's collection functions provide concise and expressive code for data manipulation. I utilized functions like filter, map, and sortedBy for complex operations like filtering completed and incompleted tasks, sort them by title and map their title for displaying. These operations are enhanced the application's experience and functionality.

Android UI Design with Jetpack Compose

Jetpack Compose provides a declarative approach to Android UI development. In this framework, developers declare their UI by calling composable functions that are used to define what they want to draw/display on the screen. Code is concise and more maintainable.

While designing the task summary screen of the application, the following TaskSummaryScreen composable function was in place.

```
21 @Composable
22 fun TaskSummaryScreen(
23     taskViewModel: TaskViewModel,
24     onNavigateBack: () → Unit
25 ) {
26     val tasks by taskViewModel.tasks.observeAsState(emptyList())
27
28     val sortedIncompleteTaskTitles = tasks
29         .filter { !it.isCompleted }
30         .sortedBy { it.title }
31         .map { it.title }
32
33     val sortedCompletedTaskTitles = tasks
34         .filter { it.isCompleted }
35         .sortedBy { it.title }
36         .map { it.title }
37
38     Scaffold(
39         topBar = {
40             TopAppBar(
41                 title = { Text(text = "Task Summary") },
42                 navigationIcon = {
43                     IconButton(onClick = onNavigateBack) {
44                         Icon(Icons.Default.ArrowBack, contentDescription = "Back")
45                     }
46                 }
47             )
48         }
49     ) { paddingValues →
50         Column(
51             modifier = Modifier
52                 .padding(paddingValues)
53                 .padding(16.dp)
54         ) {
55             Text(
56                 text = "You have ${tasks.size} tasks:",
57                 style = MaterialTheme.typography.titleMedium,
58                 modifier = Modifier.padding(bottom = 16.dp)
59             )
60         }
61     }
```

Figure 13. TaskSummaryScreen Example.

I embedded the filtering logic directly into the UI layer, hence enhancing the user experience with a way of separating tasks according to their completeness. The above example shows how the different collection operations get seamlessly woven into the feature set of an application.

```
21 @Composable
22 fun ProfileScreen(
23     authViewModel: AuthViewModel
24 ) {
25     val currentUser by authViewModel.currentUser.observeAsState()
26     var profileImageUri by remember { mutableStateOf<Uri?>(value: null) }
27
28     val launcher = rememberLauncherForActivityResult(ActivityResultContracts.GetContent()) { uri →
29         if (uri != null) {
30             profileImageUri = uri
31             authViewModel.updateProfileImage(uri.toString())
32         }
33     }
34
35     LaunchedEffect(currentUser) {
36         profileImageUri = currentUser?.profileImageUri?.let { Uri.parse(it) }
37     }
38
39     Scaffold(
40         topBar = {
41             TopAppBar(
42                 title = { Text(text: "Profile") }
43             )
44         }
45     ) { paddingValues →
46         // Use Box to position content at the top and bottom
47         Box(
48             modifier = Modifier
49                 .fillMaxSize() // Make the Box fill the entire screen
50                 .padding(paddingValues)
51                 .padding(16.dp)
52         ) {
53             // Profile content (Top)
54             Column(
55                 modifier = Modifier.align(Alignment.TopCenter), // Align at the top
56                 horizontalAlignment = Alignment.CenterHorizontally
57             ) {
58                 if (profileImageUri != null) {
59                     Image(
```

Figure 13. ProfileScreen Example.

In the ProfileScreen composable, the user was allowed to select and display a profile image in a circular avatar using `rememberLauncherForActivityResult` for smooth integration. Additionally, `LaunchedEffect` is used to update image URI when user state changed. Also, screen layout is organized with `Scaffold`. It contains `TopAppBar`, `Box` and `Column` elements for filling screen.

Jetpack Compose highly simplified the development of UI code by providing composable functions and reactive state management. The declarative approach cleaned it up, while composability through functions greatly increased reusability and modularity.

Activity and User Input Handling

Therefore, efficient handling of the user's input or events played a big role in ensuring a seamless users' experience. In Jetpack Compose, user interactions are handled in terms of lambdas and state variables.

Firstly, in the MainActivity, the onCreate() method was overridden to set the initial activity state.

```
12 <> class MainActivity : ComponentActivity() {  
13     override fun onCreate(savedInstanceState: Bundle?) {  
14         super.onCreate(savedInstanceState)  
15         setContent {  
16             val taskViewModel: TaskViewModel = viewModel()  
17             val authViewModel: AuthViewModel = viewModel()  
18             MidtermTheme {  
19                 MainApp(taskViewModel)  
20             }  
21         }  
22     }  
23 }
```

Figure 14. MainActivity overridden Example.

The setContent replaces the older setContentView; it allows the definition of UI with composable functions. Instances of ViewModel were obtained with the viewModel() delegate; the latter allows retrieving instances which are bound to the activity so that they survive configuration changes.

Input provided by the user was managed using state variables and event handlers. As an example, in handling text input:

```
23 var title by remember { mutableStateOf( value: "" ) }  
24 var description by remember { mutableStateOf( value: "" ) }  
25  
26 Scaffold(  
27     topBar = {  
28         TopAppBar(title = { Text( text: "Add Task" ) },  
29             navigationIcon = {  
30                 IconButton(onClick = onNavigateBack) {  
31                     Icon(  
32                         imageVector = Icons.AutoMirrored.Filled.ArrowBack,  
33                         contentDescription = "Back"  
34                     )  
35                 }  
36             }  
37         )  
38     },  
39     content = { paddingValues →  
40         Column(  
41             modifier = Modifier  
42                 .padding(paddingValues)  
43                 .padding(16.dp)  
44         ) {  
45             OutlinedTextField(  
46                 value = title,  
47                 onValueChange = { title = it },  
48                 label = { Text( text: "Title" ) },  
49                 modifier = Modifier.fillMaxWidth()  
50             )  
51         }  
52     }  
53 )
```

Figure 15. User input handling Example.

In this case, remember and mutableStateOf create a state variable for the current value of the text field. And the onChange lambda updates that state whenever the user types something. Then, automatically, the UI updates in response.

```
        .fillMaxWidth()
        .height(100.dp),
        maxLines = 4
    )
    Spacer(modifier = Modifier.height(16.dp))
    Button(
        onClick = {
            val newTask = Task(
                username = currentUser?.username ?: "",
                title = title,
                description = description
            )
            taskViewModel.addTask(newTask)
            onNavigateBack()
        },
        modifier = Modifier.fillMaxWidth(),
        enabled = title.isNotBlank() && description.isNotBlank()
    ) {
        Text(text: "Add Task")
    }
}
```

Figure 15. User button onClick handling Example.

The onClick parameter takes a lambda defining what should happen once the button is clicked: a new task is created, and the app navigates back to the previous screen.

Handling user input and events in Jetpack Compose feels quite intuitive. Once the state changes, refreshes in UI happen automatically. In Compose, declarative logic makes it easier to handle even complex user interactions.

Activity Lifecycle Management

Jetpack Compose handles most of the UI concerns internally, placing less of a requirement on overriding many lifecycle methods. Still, onCreate is one of the most essential lifecycle methods, which should be thoughtfully designed. In onCreate, an Activity will initialize the resources and set up the UI content.

```

3      import android.os.Bundle
4      import androidx.activity.ComponentActivity
5      import androidx.activity.compose.setContent
6      import androidx.lifecycle.viewmodel.compose.viewModel
7      import com.example.midterm.ui.theme.MainApp
8      import com.example.midterm.ui.theme.MidtermTheme
9      import com.example.midterm.viewmodel.AuthViewModel
10     import com.example.midterm.viewmodel.TaskViewModel
11
12     ▶ </> class MainActivity : ComponentActivity() {
13     🚩     override fun onCreate(savedInstanceState: Bundle?) {
14         super.onCreate(savedInstanceState)
15         setContent {
16             val taskViewModel: TaskViewModel = viewModel()
17             val authViewModel: AuthViewModel = viewModel()
18             MidtermTheme {
19                 MainApp(taskViewModel)
20             }
21         }
22     }
23 }

```

Figure 16. onCreate override method.

Compose does most internal lifecycle handling, which handles many of these concerns internally. The UI composables are automatically ready to react to changes in state and recompositions, it requires less and less explicit lifecycle handling unmethods like onResume or onPause.

Keeping most of logic within onCreate lets the activity correctly set up before the UI is rendered. Compose has life-cycle-aware components and such side-effect APIs as LaunchedEffect and DisposableEffect, so you could handle tasks that need to respond to lifecycle events inside Composable functions. The good side effect of this is that these are hooked into the lifecycle of the composable, not necessarily managing an activity's lifecycle methods themselves.

Composables and State Management in Jetpack Compose

The declarative UI paradigm of Jetpack Compose with strong state management was key to this responsive and maintainable task management application. This section provides a summary of how composable functions and state management strategies have been employed to assure a seamless experience for the user in the project.

The UI is decomposed into modular, reusable composable functions, per best practices of Compose.


```

15 @Composable
16 fun TaskItem(task: Task, onTaskClick: (Int) → Unit) {
17     Surface(
18         modifier = Modifier
19             .fillMaxWidth()
20             .clickable { onTaskClick(task.id) }
21             .padding(8.dp),
22         tonalElevation = 2.dp,
23         shape = MaterialTheme.shapes.medium,
24         color = MaterialTheme.colorScheme.surface
25     ) {
26         Row(
27             modifier = Modifier.padding(16.dp),
28             verticalAlignment = Alignment.CenterVertically
29         ) {
30             Icon(
31                 imageVector = if (task.isCompleted) Icons.Filled.CheckCircle else Icons.Filled.Circle,
32                 contentDescription = null,
33                 tint = if (task.isCompleted) MaterialTheme.colorScheme.primary else MaterialTheme.colorScheme.onSurfaceVariant,
34                 modifier = Modifier.size(24.dp)
35             )
36             Spacer(modifier = Modifier.width(16.dp))
37             Column {
38                 Text(
39                     text = task.title,
40                     style = MaterialTheme.typography.titleMedium,
41                     color = if (task.isCompleted) MaterialTheme.colorScheme.onSurfaceVariant else MaterialTheme.colorScheme.onSurface
42                 )
43                 Spacer(modifier = Modifier.height(4.dp))
44                 Text(
45                     text = task.description,
46                     style = MaterialTheme.typography.bodyMedium,
47                     color = if (task.isCompleted) MaterialTheme.colorScheme.onSurfaceVariant else MaterialTheme.colorScheme.onSurface
48                 )
49             }
50         }
51     }
52 }
53

```

Figure 17. Task Item Example.

TaskItem shows the listing of individual tasks along with its title and description and whether it was completed. The composable would contain UI elements and user interaction - for example, the navigation to the detailed view of one of the selected tasks by user. Also, UI remains synchronized with the underlying data of tasks.

```

10 class TaskViewModel(application: Application) : AndroidViewModel(application) {
11     private val taskDao = AppDatabase.getDatabase(application).taskDao()
12
13     private val _tasks = MutableLiveData<List<Task>>()
14     val tasks: LiveData<List<Task>>get() = _tasks
15
16     fun loadTasksForUser(username: String) {
17         viewModelScope.launch {
18             taskDao.getTasksForUser(username).collect { taskList →
19                 _tasks.postValue(taskList)
20             }
21         }
22     }
23
24     fun addTask(task: Task) = viewModelScope.launch {
25         taskDao.insertTask(task)
26     }
27
28     fun updateTask(task: Task) = viewModelScope.launch {
29         taskDao.updateTask(task)
30     }
31
32     fun deleteTask(task: Task) = viewModelScope.launch {
33         taskDao.deleteTask(task)
34     }
35 }

```

Figure 18. TaskViewModel Example.

Here, TaskViewModel will manage the business logic and operations on data, such as getting tasks for the user and changing the status of tasks. View Model helps the app to maintain the state on configuration change.

```
50 @Composable
51 fun MainContent(
52     navController: NavHostController,
53     taskViewModel: TaskViewModel,
54     authViewModel: AuthViewModel
55 ) {
56     Scaffold(
57         bottomBar = {
58             BottomNavBar(navController)
59         }
60     ) { innerPadding →
61         NavHost(
62             navController = navController,
63             startDestination = "taskList",
64             modifier = Modifier.padding(innerPadding)
65         ) {
66             composable( route: "taskList") {
67                 TaskListScreen(
68                     taskViewModel = taskViewModel,
69                     authViewModel = authViewModel,
70                     onAddTask = { navController.navigate( route: "addTask") },
71                     onTaskClick = { taskId →
72                         navController.navigate( route: "taskDetail/$taskId")
73                     },
74                     onViewSummary = { navController.navigate( route: "taskSummary") },
75                 )
76             }
67
```

Figure 19. Navigation Control Example.

State management is closely connected with navigation between screens for proper data flow of an application. Passing Data with Routes: it sends the ID of the task as a route argument while navigating from TaskListScreen to TaskDetailScreen, which enables the TaskDetailScreen to fetch and show data relevant to that task.

The to-do list application uses Jetpack Compose for composable functions and efficient state management with ViewModel and LiveData. Application keeps the consistency of the UI and data in front of the user to maintain seamless and intuitive user experience.

Implementing Login Logic and Data Persistence with Room

Implementing user authentication was crucial for the security and personalization issues of the app. The AuthViewModel managed authentication state and current user information.

```

9      class AuthViewModel(application: Application) : AndroidViewModel(application) {
10         private val userDao = AppDatabase.getDatabase(application).userDao()
11
12         private val _isAuthenticated = MutableLiveData<Boolean>(value: false)
13         val isAuthenticated: LiveData<Boolean> get() = _isAuthenticated
14
15         private val _currentUser = MutableLiveData<User?>()
16         val currentUser: LiveData<User?> get() = _currentUser
17
18         fun login(username: String, password: String) = viewModelScope.launch {
19             val user = userDao.getUserByUsername(username)
20             if (user != null && user.password == password) {
21                 _isAuthenticated.postValue(value: true)
22                 _currentUser.postValue(user)
23             } else {
24                 _isAuthenticated.postValue(value: false)
25             }
26         }
27
28         fun logout() {
29             _isAuthenticated.value = false
30             _currentUser.value = null
31         }
32
33         fun register(user: User) = viewModelScope.launch {
34             userDao.insertUser(user)
35             _isAuthenticated.postValue(value: true)
36             _currentUser.postValue(user)
37         }
38
39         fun updateProfileImage(uri: String) = viewModelScope.launch {
40             val user = _currentUser.value
41             if (user != null) {
42                 val updatedUser = user.copy(profileImageUri = uri)
43                 userDao.updateUser(updatedUser)
44                 _currentUser.postValue(updatedUser)
45             }
46         }
47     }

```

Figure 20. AuthViewModel Logic Example.

The implemented login functionality asynchronously authenticates the user by using coroutines. It performs the check of the credentials against the database and updates all necessary LiveData properties.

Room was the persistence library used to store data of users and tasks locally. AppDatabase was a singleton instance of the database, so only one instance could exist, avoiding multiples connections.

```

10 @Database(entities = [Task::class, User::class], version = 4)
11 abstract class AppDatabase : RoomDatabase() {
12     abstract fun taskDao(): TaskDao
13     abstract fun userDao(): UserDao
14
15
16     companion object {
17         @Volatile private var INSTANCE: AppDatabase? = null
18
19         private val MIGRATION_2_3 = object : Migration(2, 3) {
20             override fun migrate(db: SupportSQLiteDatabase) {
21                 db.execSQL("ALTER TABLE tasks ADD COLUMN username TEXT NOT NULL DEFAULT '')")
22             }
23         }
24
25         private val MIGRATION_3_4 = object : Migration(3, 4) {
26             override fun migrate(db: SupportSQLiteDatabase) {
27                 db.execSQL("ALTER TABLE users ADD COLUMN profileImageUri TEXT")
28             }
29         }
30
31         fun getDatabase(context: Context): AppDatabase {
32             return INSTANCE ?: synchronized(lock: this) {
33                 val instance = Room.databaseBuilder(
34                     context.applicationContext,
35                     AppDatabase::class.java,
36                     name: "todo_database"
37                 )
38                     .addMigrations(MIGRATION_2_3, MIGRATION_3_4)
39                     .build()
40                 INSTANCE = instance
41                 instance
42             }
43         }
44     }
45 }

```

Figure 21. AppDatabase with migrations using Room Library.

DAOs included TaskDao and UserDao, which, in turn, provided several methods to the database to query, insert, update, and delete data.

The integration of Room with ViewModel and LiveData made the flow of data from the database to UI smooth and well-structured. Moreover, authentication enhanced the app's security and allowed personalization, hence making it robust and reliable for users.

Conclusion

In conclusion, this Task Manager application showed how effective use of Kotlin and Jetpack Compose in the development of a more modern and user-friendly Android application. The declarative UI paradigm with Jetpack Compose was employed in this project for a clean and modularly designed interface that enhances development efficiency and maintainability.

Composable functions were in place, integrating key functionalities like task creation, categorization of tasks into incomplete and completed tasks, detailed task view, and summary analytics. View model and LiveData ensured proper state management; the UI reflected changes in data real-time, or else was giving a smooth user experience.

Navigation among screens was efficiently done through the help of a navigation component provided by Jetpack Compose, which enabled users to switch between the task list, detail views, and summary screens with ease. Adding authentication mechanisms made sure the user's data is secured; therefore, a user can keep his private tasks visible only to himself.

According to best practices, this entire project has followed, for example modular composable, and one-way data flow to keep the codebase scalable and robust. Layout management and navigation conflict resolution issues were resolved professionally to create a polished and workable application.

Therefore, this project not only fulfilled but also put a very good grounding for further development that could be made in the future. Further development could be allowed selecting the level of priority of the task and setting deadlines for the tasks and notifying them. Second, optimizations within the app performance can be further enhanced when there is a huge amount of task data. In general, Task Manager Application serves proof about what Kotlin and Jetpack Compose are capable of in relation to efficient and aesthetic Android application development.

References

1. Jetpack Compose Documentation - <https://developer.android.com/jetpack/compose>
2. Kotlin Language Documentation - <https://kotlinlang.org/docs/reference/>
3. Material Design - <https://m3.material.io/>
4. ViewModel Guide - <https://developer.android.com/topic/libraries/architecture/viewmodel>
5. Room Persistence Library - <https://developer.android.com/training/data-storage/room>
6. Jetpack Compose Navigation - <https://developer.android.com/jetpack/compose/navigation>

Appendices

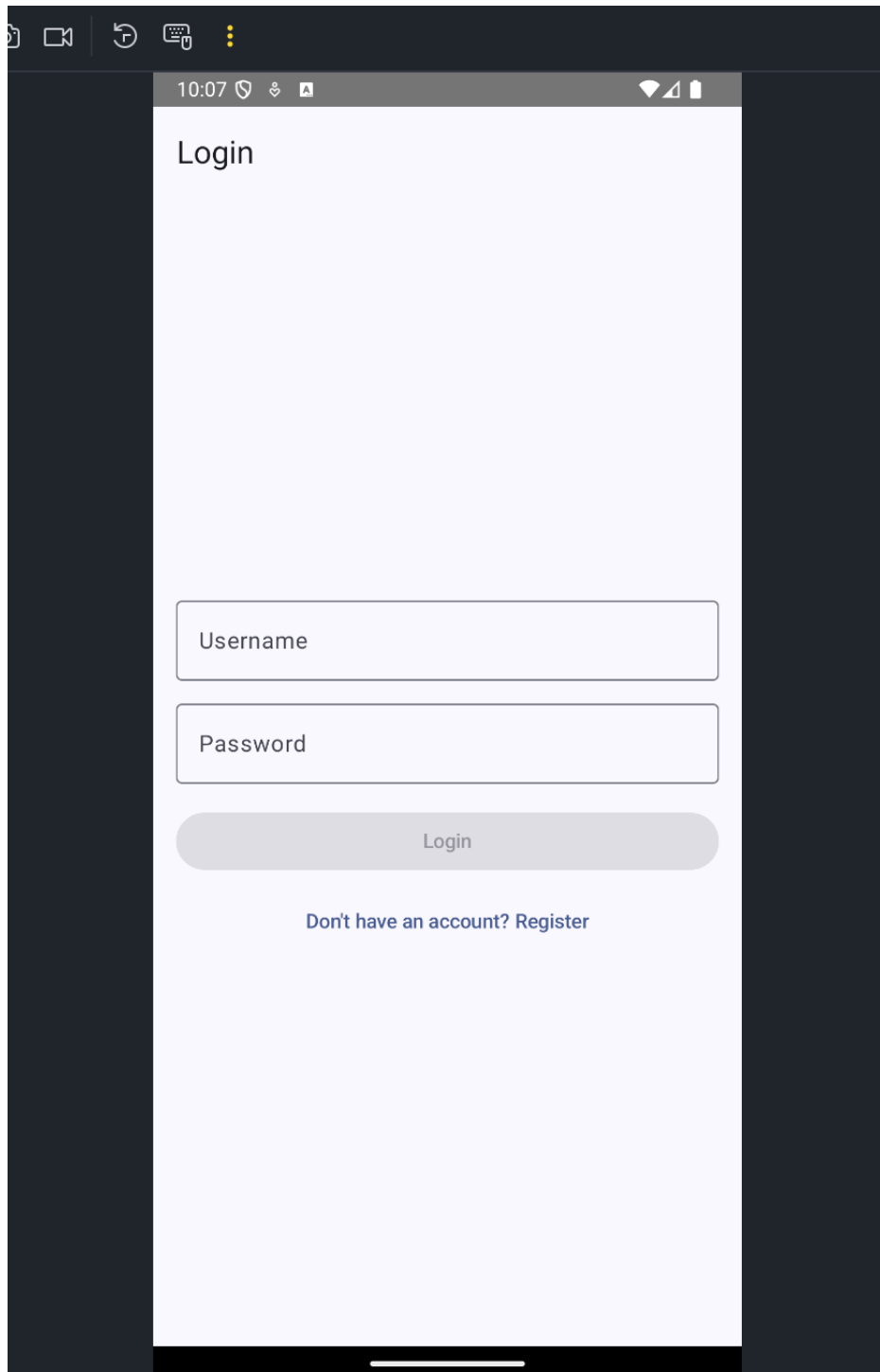
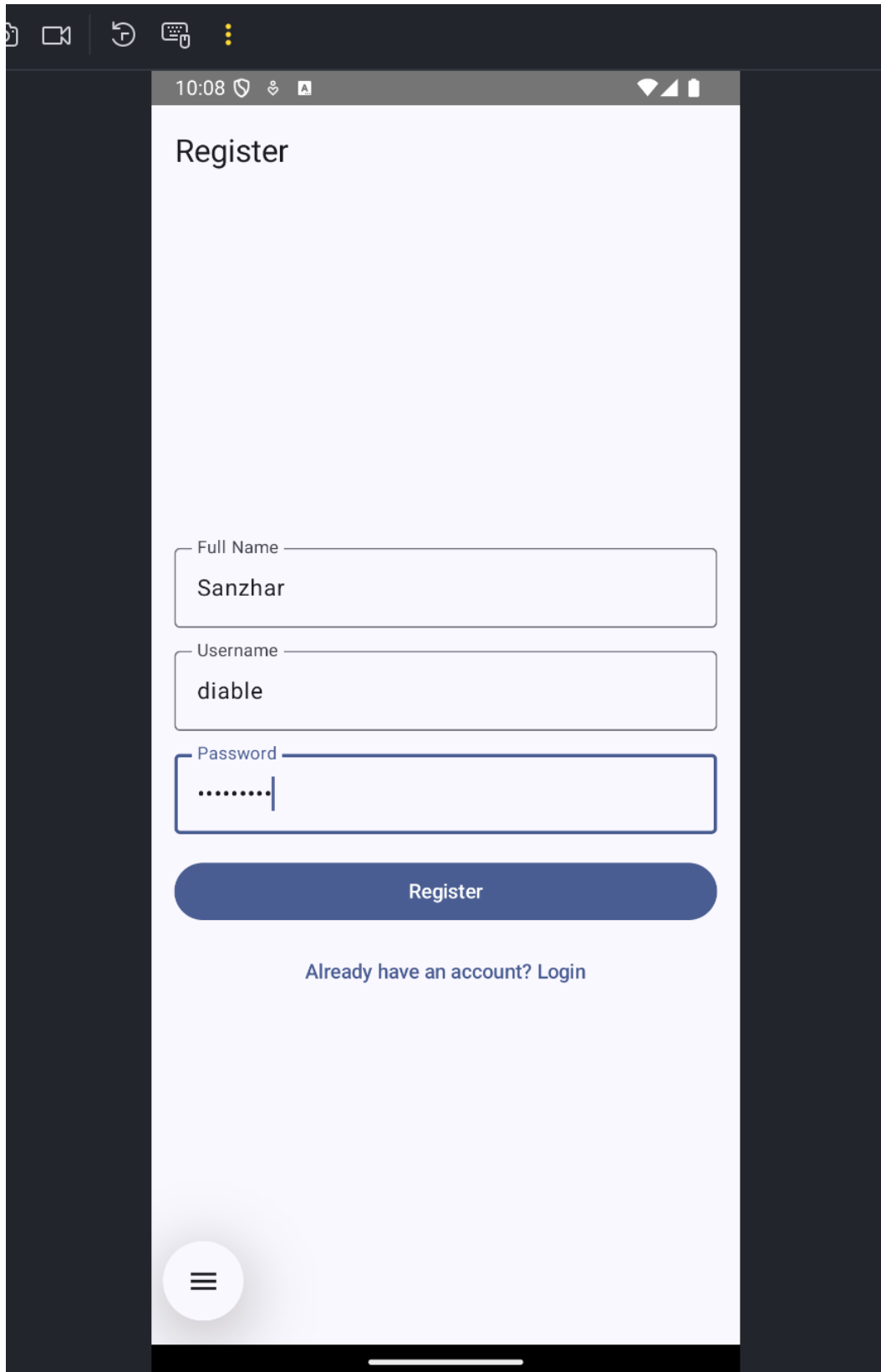


Figure 22. Login Page.



A screenshot of a mobile application's registration page. The page has a white background with a dark blue header bar at the top. The header bar contains a status bar with the time 10:08, a shield icon, a location pin icon, and a battery icon. Below the header bar, the word "Register" is displayed in a large, bold, dark blue font. The registration form consists of three input fields: "Full Name" with the text "Sanzhar", "Username" with the text "diable", and "Password" with a masked password ".....". Each input field has a corresponding label above it. Below the input fields is a large, rounded, dark blue button with the text "Register" in white. Below the button is a link that says "Already have an account? Login". At the bottom left of the page, there is a circular icon with three horizontal lines, representing a menu or navigation button.

Register

Full Name
Sanzhar

Username
diable

Password
.....

Register

Already have an account? [Login](#)

Figure 23. Register Page.

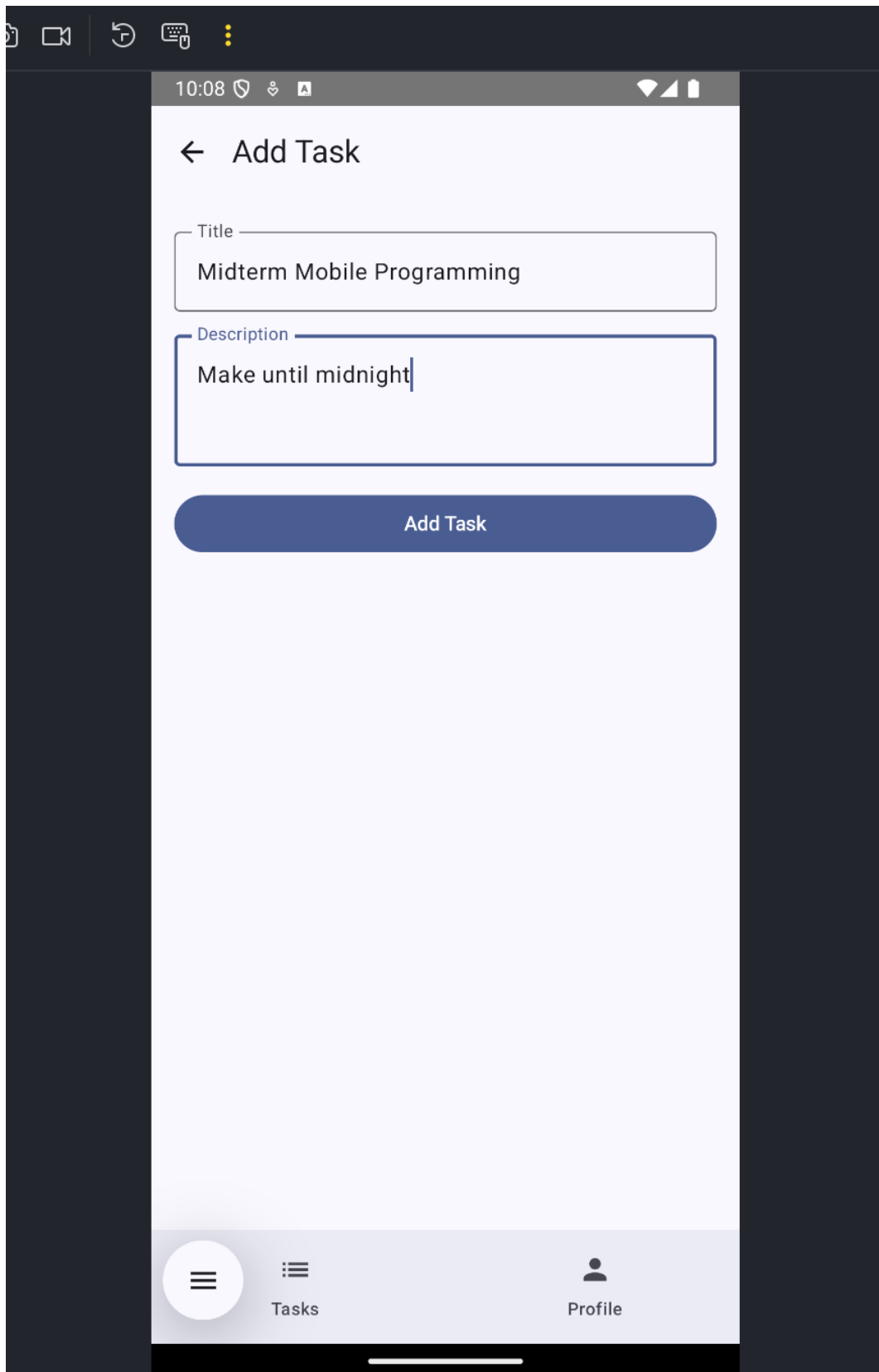


Figure 24. Add Task Page.

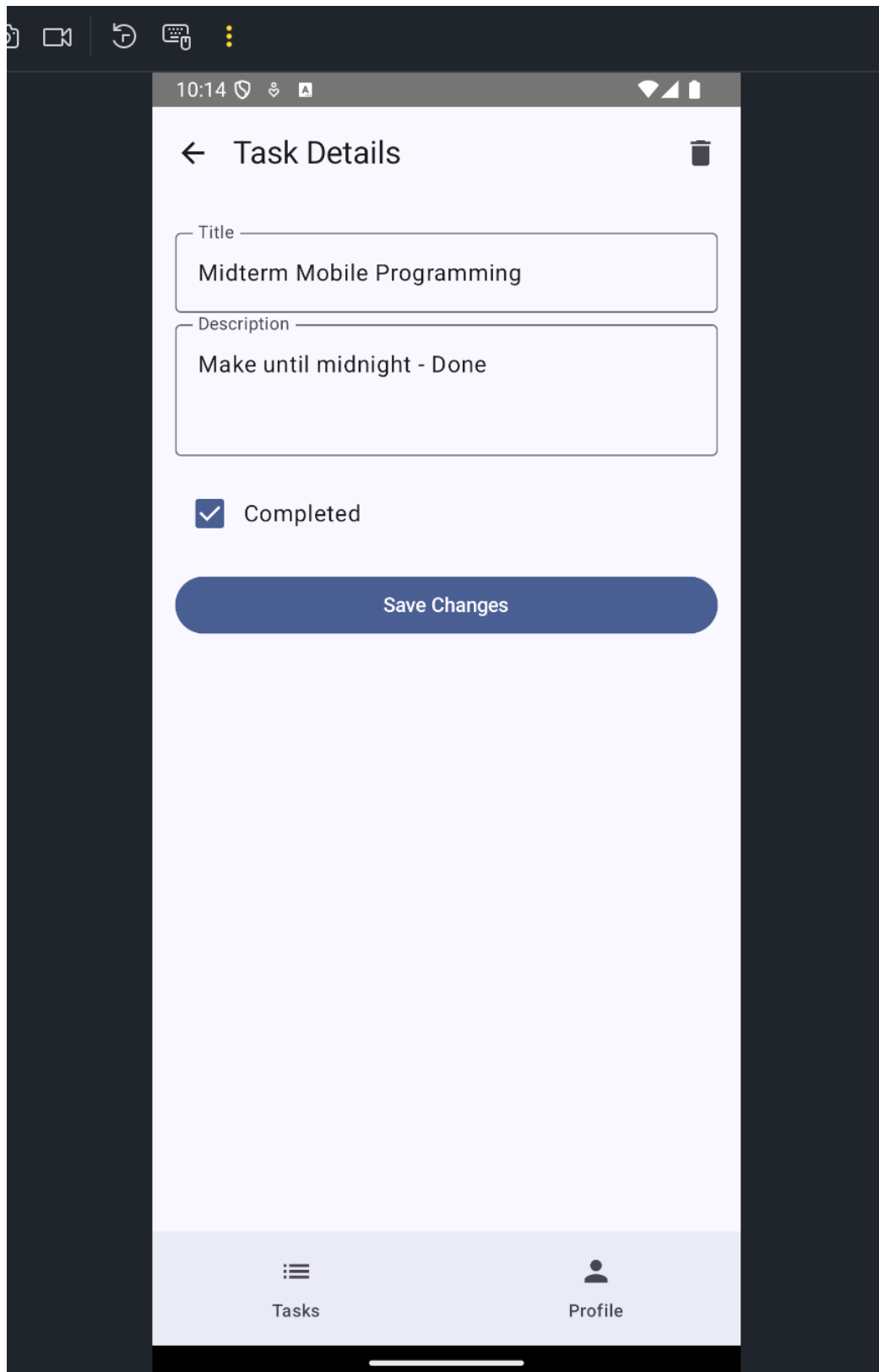


Figure 25. Task Detail Page.

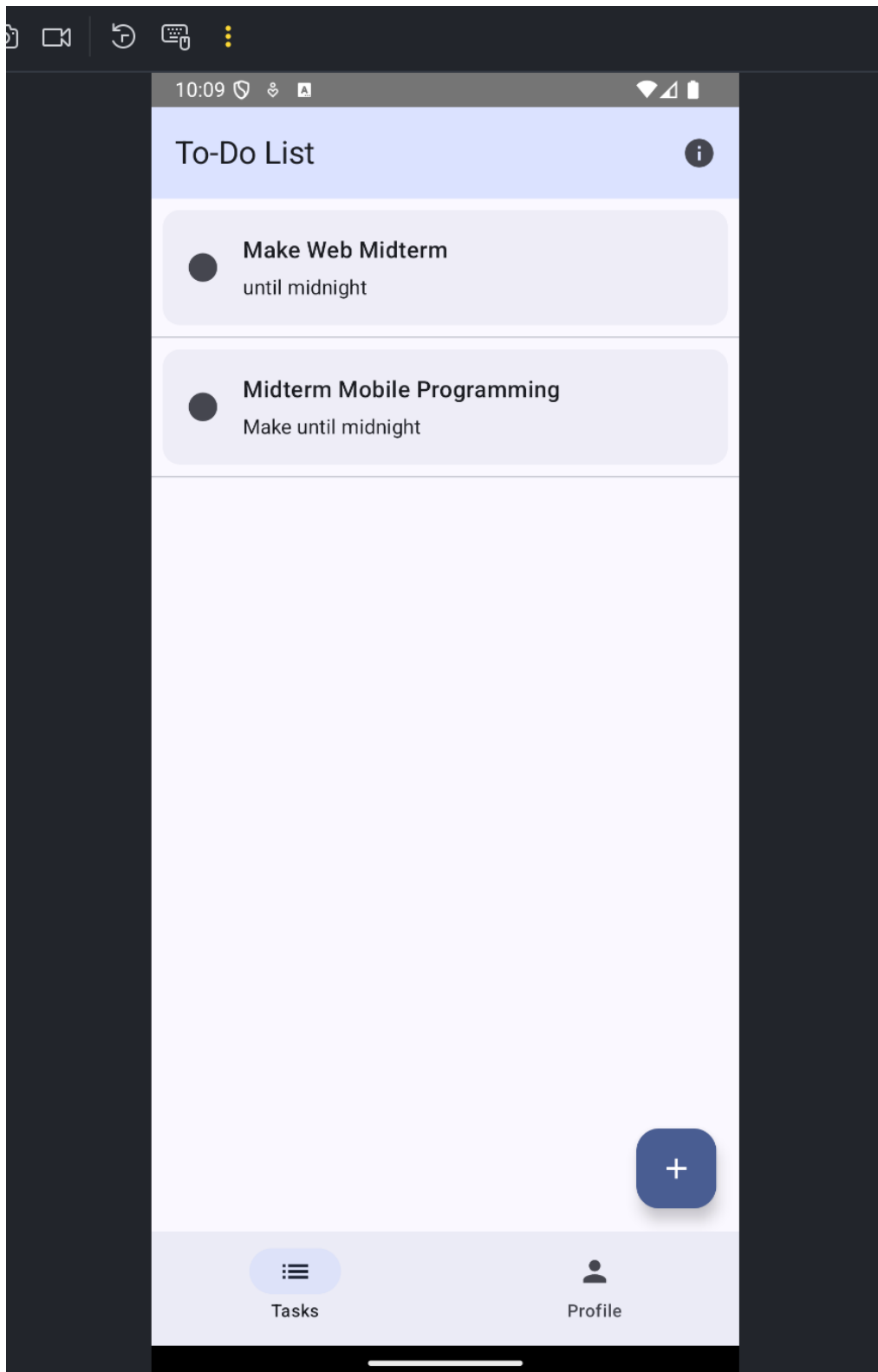


Figure 26. List of Taks Page.

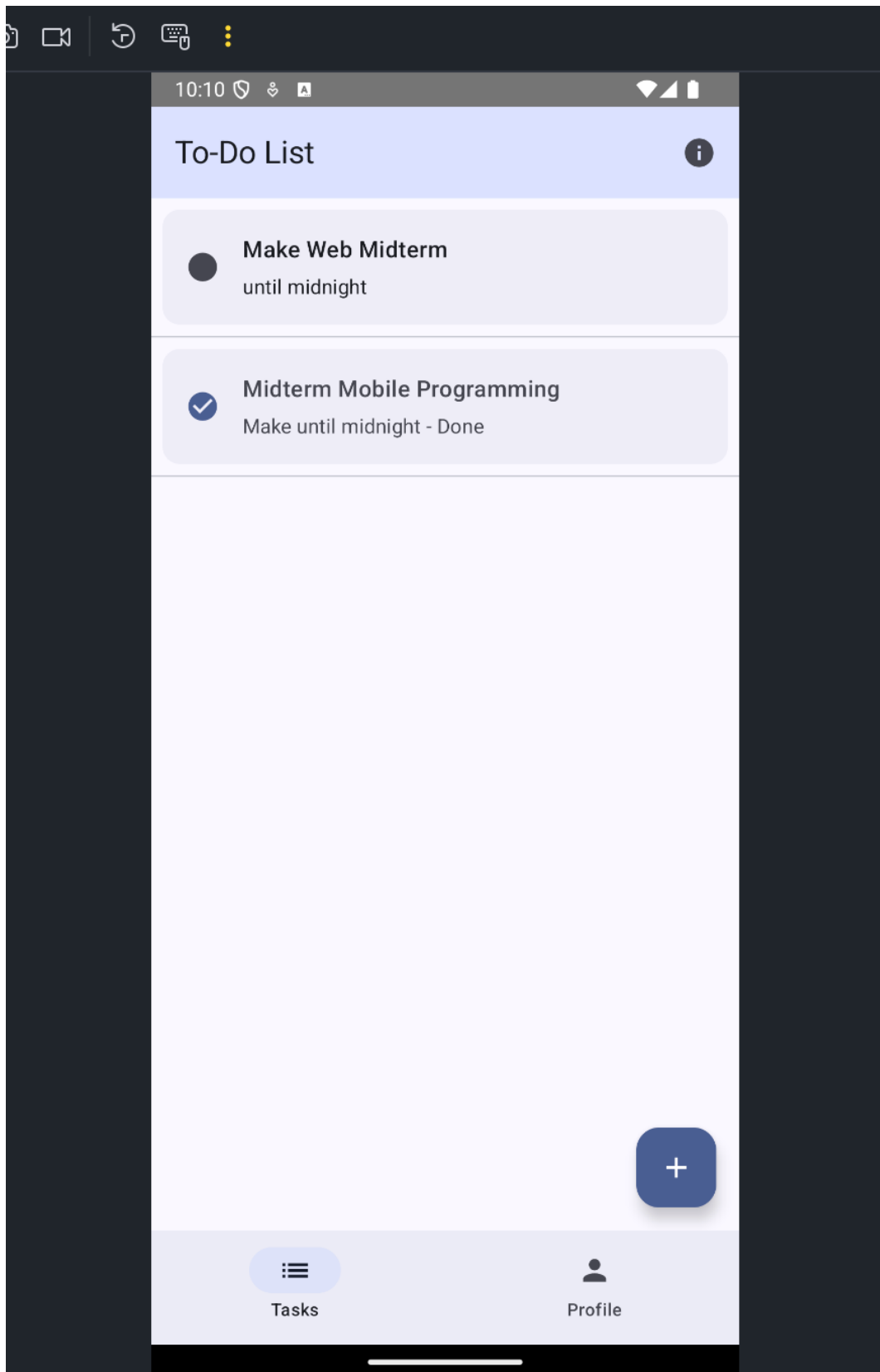


Figure 27. Updated description and status of second task in List of Tasks Page.

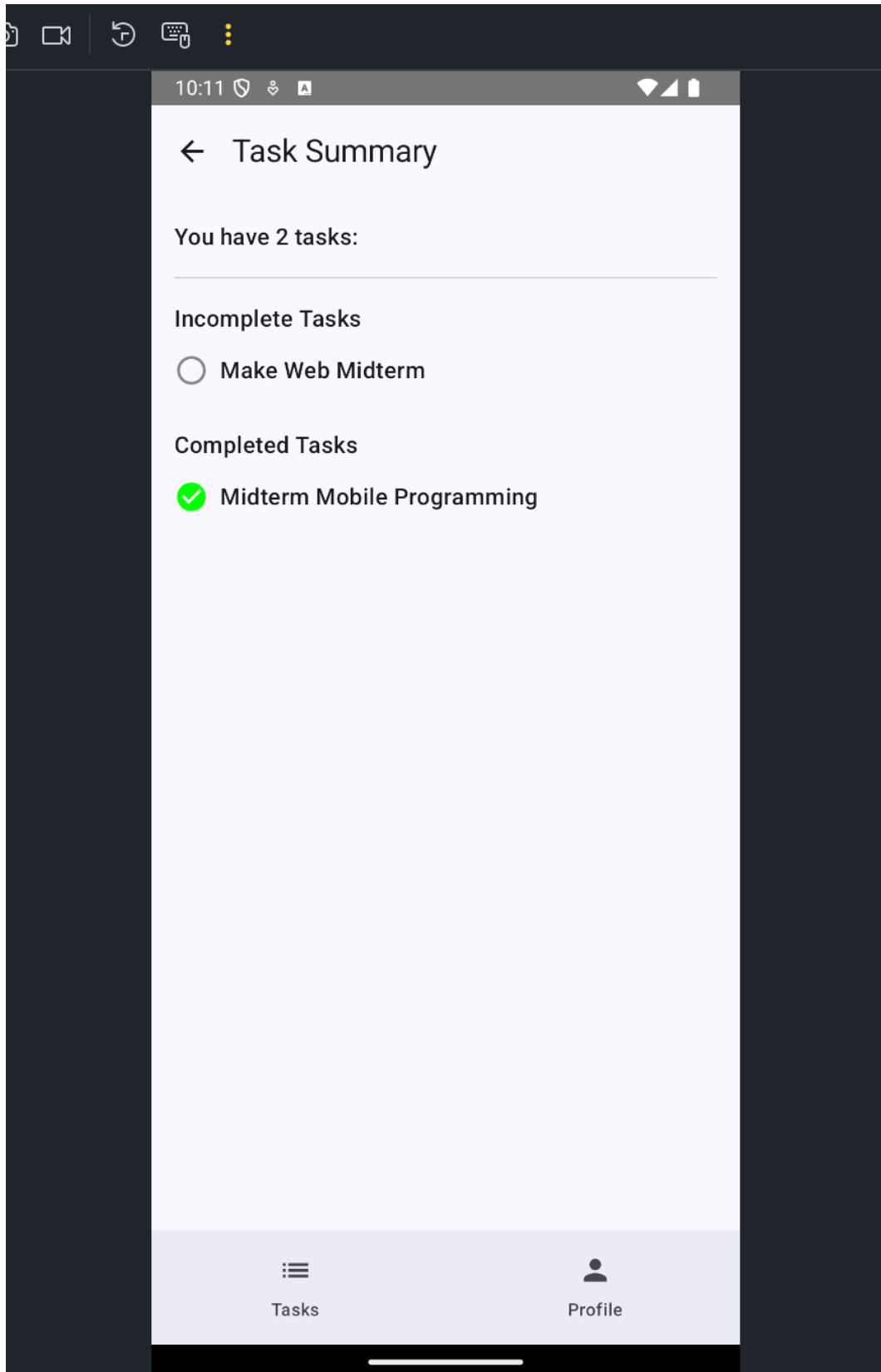


Figure 28. Task Summary Page.

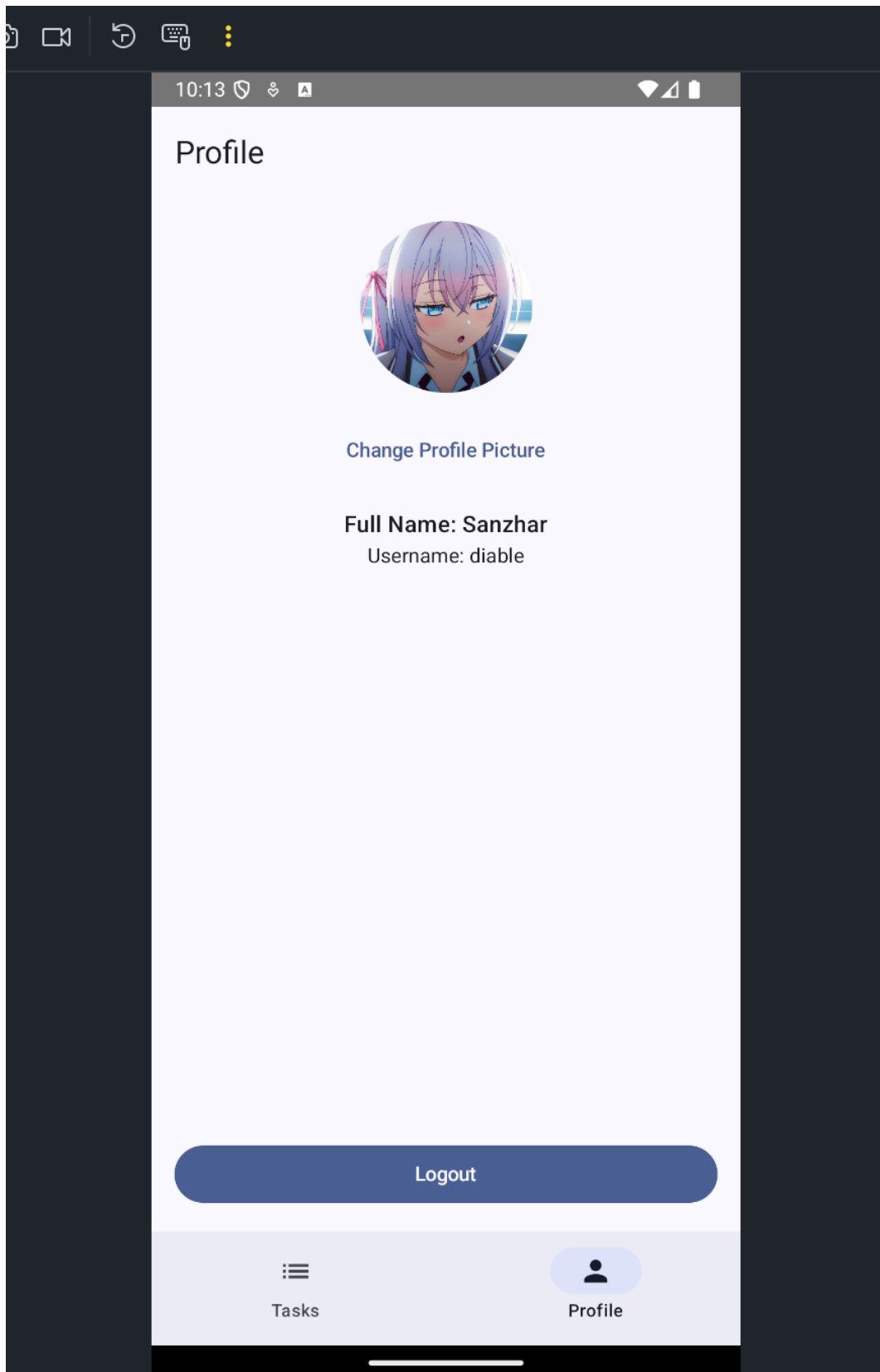


Figure 29. Profile Page.

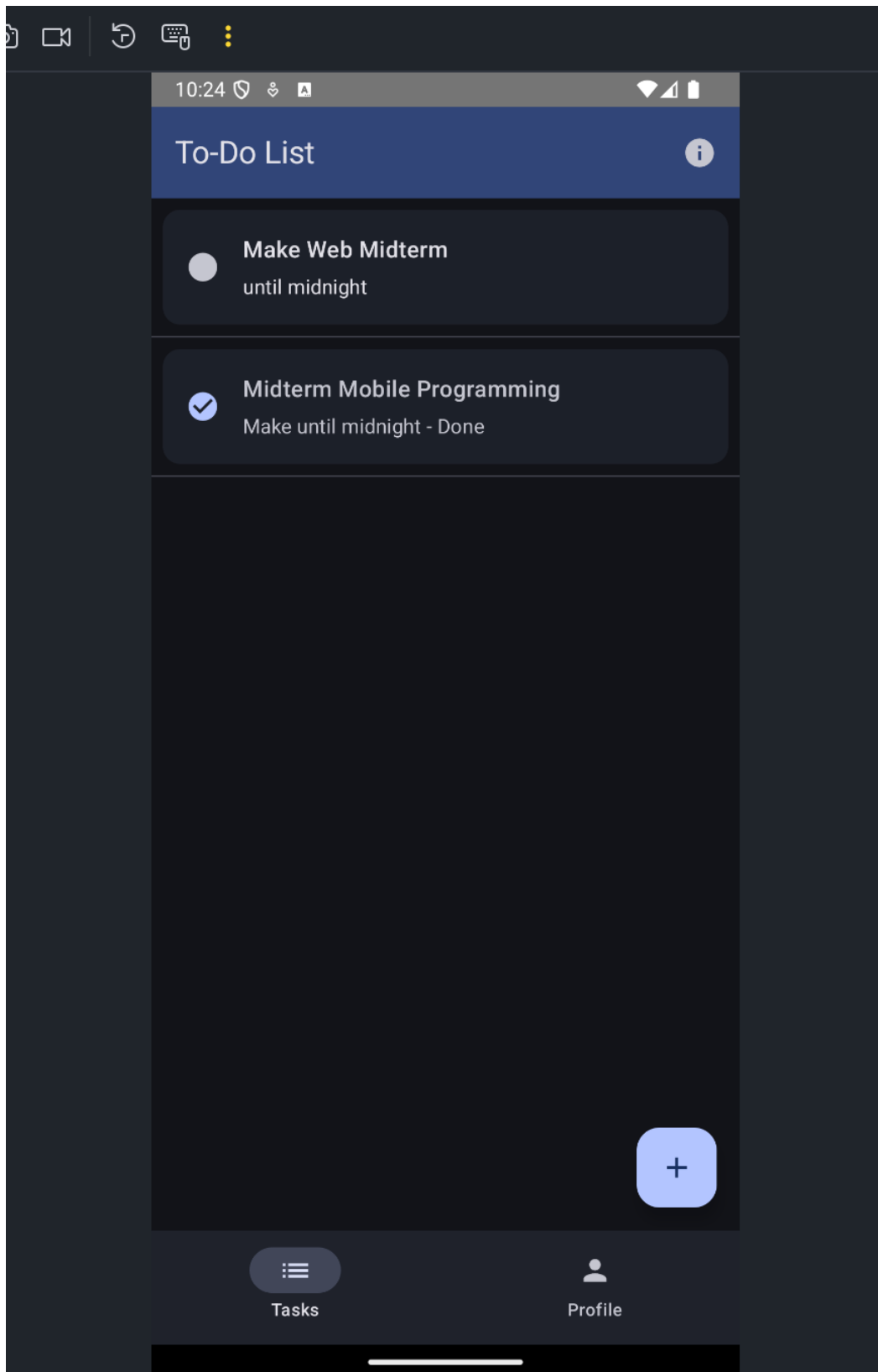


Figure 30. Dark Theme of Application.