KAZAKH-BRITISH TECHNICAL UNIVERSITY

Prepared by: Serzhan Yerasil

Midterm

Mobile Programing

Building a Simple Task Management App in Android Using Kotlin

26.10.2024

Table of Contents

Executive summary

The goal of the project is to develop an Android task management application using Kotlin. The application includes user-friendly task display and entry features, using Android Jetpack components such as Lifecycle and ViewModel to manage the application state and RecyclerView to display task lists. Technologies used include Android Studio, Kotlin, and Jetpack compose layouts. The result is a functional, modular task management application that has a simple and clear interface.

Introduction

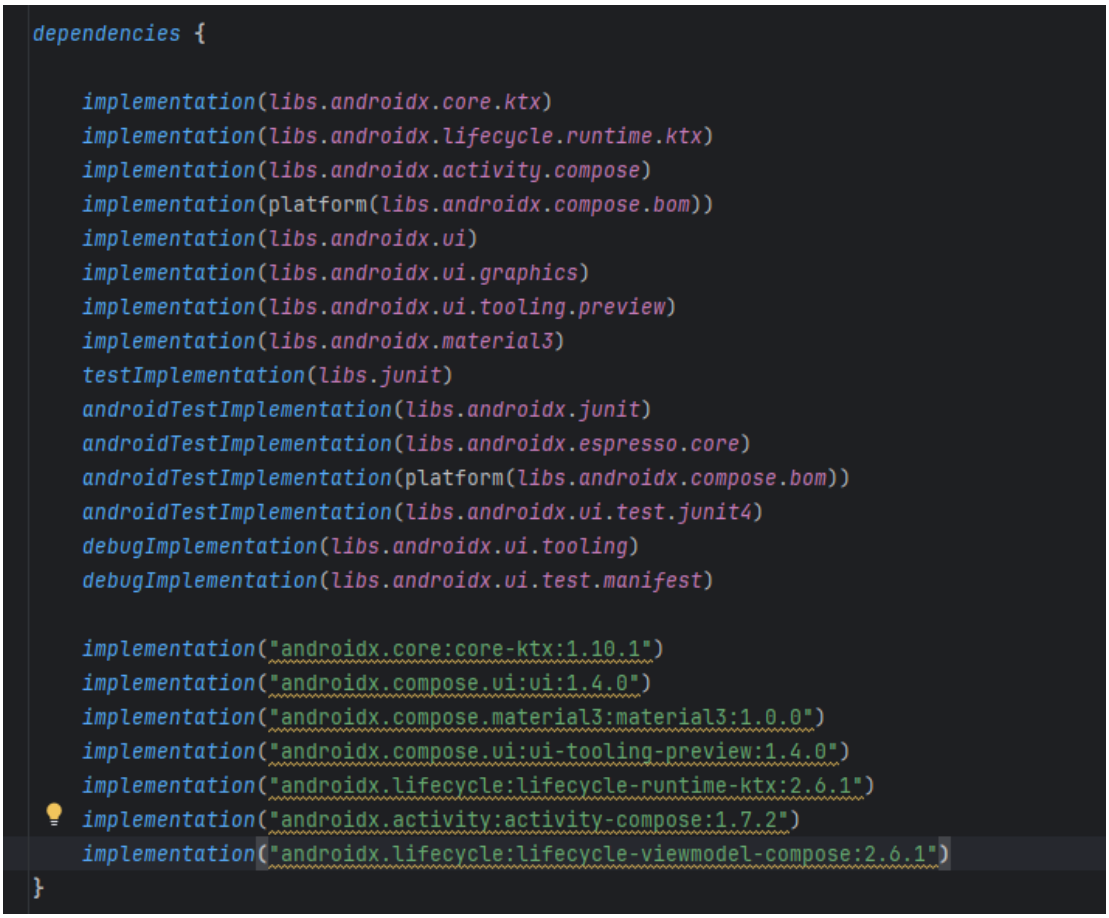
We use mobile apps every day for our various tasks. In Android development, Kotlin stands out as a powerful, simple in syntax language. The motivation for creating a task management app is that it is a fairly simple idea, but at the same time very useful from a practical point of view. This project demonstrates the use of Kotlin to create a fully functional application with a user interface and functionality.

Project Objectives

I used the following libraries:

•Jetpack Compose: Selected for declarative construction of the UI, making UI development much easier and more straightforward.

•Material3: Concretely implements Material Design 3 at Compose. Material3 is designed to offer modern UI components and theming. Use Jetpack Compose to build a responsive and dynamic user interface, integrating Material 3 design principles for a modern and visually appealing app layout. Apply OOP principles to design the app's architecture, utilizing classes, data classes, and encapsulation for clean and maintainable code.



```
dependencies {

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
    implementation(libs.androidx.activity.compose)
    implementation(platform(libs.androidx.compose.bom))
    implementation(libs.androidx.ui)
    implementation(libs.androidx.ui.graphics)
    implementation(libs.androidx.ui.tooling.preview)
    implementation(libs.androidx.material3)
    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)
    androidTestImplementation(platform(libs.androidx.compose.bom))
    androidTestImplementation(libs.androidx.ui.test.junit4)
    debugImplementation(libs.androidx.ui.tooling)
    debugImplementation(libs.androidx.ui.test.manifest)

    implementation("androidx.core:core-ktx:1.10.1")
    implementation("androidx.compose.ui:ui:1.4.0")
    implementation("androidx.compose.material3:material3:1.0.0")
    implementation("androidx.compose.ui:ui-tooling-preview:1.4.0")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.6.1")
    implementation("androidx.activity:activity-compose:1.7.2")
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.6.1")
}
```

Overview of Android Development and Kotlin

Setting up Android Studio, the official IDE for Android development, involves configuring it to support Kotlin, which is now a preferred language for Android. After installing Android Studio, the Kotlin plugin is enabled, allowing seamless Kotlin project creation and code compilation. Kotlin brings a range of powerful features that enhance Android development over Java, such as concise syntax, null safety, and functional programming support. Unlike Java, Kotlin reduces boilerplate code, making codebases easier to maintain and reducing the chance of runtime errors. Its compatibility with Java also enables smooth integration with existing Android libraries, accelerating development and improving code quality.

Functions and Lambdas in Kotlin

Created a function addTask(title: String, description: String) to add new tasks to the task list. Used getTasks() to return a list of all tasks, supporting UI updates and task display. Functions like updateTask(task: Task) and deleteTask(taskId: Int) allow seamless task management. Lambdas streamline click handling, like marking tasks as complete onComplete = { updateTask(it.copy(isCompleted = true)) }

```
@Composable
fun TaskList(tasks: List<Task>, onTaskUpdated: (Task) -> Unit, onTaskDeleted: (Int) -> Unit) {
    LazyColumn(modifier = Modifier.fillMaxSize()) {
        items(tasks) { task ->
            NeonTaskCard(
                task = task,
                onComplete = {
                    onTaskUpdated(task.copy(isCompleted = true))
                }
            )
        }
    }
}
```

This TaskList function is a Compose component for displaying a list of tasks. It takes a list of tasks and onTaskUpdated and onTaskDeleted functions for updating and deleting tasks. Inside the LazyColumn, TaskItem elements are dynamically created for each task, filling the entire available screen.

Functions and Lambdas in Kotlin

The core data structure of the app is a Task data class, designed to represent each task with properties such as id, title, description

The task data class encapsulates task properties, while helper methods like copy allow for easy creation of modified instances (e.g., marking a task as completed). The TaskViewModel class centralizes task operations, like addTask, updateTask, and deleteTask, ensuring the task list is managed in a controlled manner.

```
class TaskViewModel : ViewModel() {

    private val _taskList = MutableStateFlow(
        listOf(
            Task( id: 1,  title: "Finish report",  description: "Finish midterm projects report"),
            Task( id: 2,  title: "Complete 2 weekly sprint",  description: "Mark all tasks as completed")
        )
    )
    val taskList: StateFlow<List<Task>> = _taskList

    fun addTask(title: String, description: String) {
        val newTask = Task(id = _taskList.value.size + 1, title = title, description = description)
        _taskList.value = _taskList.value + newTask
    }

    fun updateTask(task: Task) {
        _taskList.value = _taskList.value.map {
            if (it.id == task.id) task else it
        }
    }

    fun deleteTask(taskId: Int) {
        _taskList.value = _taskList.value.filter { it.id != taskId }
    }

    fun getTaskById(taskId: Int): Task? {
        return _taskList.value.find { it.id == taskId }
    }
}
```

Working with Collections in Kotlin

A List is used to store tasks, allowing for easy retrieval, addition, and modification of tasks in a sequential order. The task list is wrapped in a MutableStateFlow, enabling real-time UI updates whenever tasks are added, modified, or deleted. The filter function is used to retrieve tasks based on their status (e.g., filtering completed tasks). Tasks can be sorted by properties such as id or title to provide a better user experience.

```
private val _taskList = MutableStateFlow(
    listOf(
        Task( id: 1,  title: "Finish report",  description: "Finish midterm projects report"),
        Task( id: 2,  title: "Complete 2 weekly sprint",  description: "Mark all tasks as completed")
    )
)
```

```
fun addTask(title: String, description: String) {
    val newTask = Task(
        id = (_taskList.value.maxOfOrNull { it.id } ?: 0) + 1,
        title = title,
        description = description
    )
    _taskList.value = _taskList.value + newTask
}
```

_taskList is a private field of type MutableStateFlow initialized with a list of tasks. It contains tasks such as "Finish report" and "Complete 2 weekly sprint". MutableStateFlow allows tracking changes to the task list, ensuring that the UI is updated in real time when data changes.

Android Layout Design with Jetpack Compose

The app's UI was created entirely using Jetpack Compose, Google's modern toolkit for building native Android UIs declaratively. This approach replaced traditional XML layouts, allowing for more flexible, dynamic, and responsive UI creation directly in Kotlin code.The app extensively uses Column and Row composables to arrange UI elements vertically and horizontally, respectively. This structure allows intuitive layout management for forms, lists, and buttons, ensuring clear organization and alignment. Used for displaying the list of tasks. LazyColumn is optimal for large or dynamic lists, as it only renders visible items, conserving memory and improving performance. Each task is encapsulated within a Card composable, providing a visually distinct, customizable container that includes padding, shadow, and optional borders. Cards enhance visual hierarchy, making individual tasks stand out in the list.

```kotlin
@Composable
fun TaskItem(task: Task, onTaskUpdated: (Task) -> Unit, onTaskDeleted: (Int) -> Unit) {
    var isEditing by remember { mutableStateOf( value: false) }
    var updatedTitle by remember { mutableStateOf(task.title) }
    var updatedDescription by remember { mutableStateOf(task.description) }

    if (isEditing) {
        Card(
            modifier = Modifier
                .fillMaxWidth()
                .padding(8.dp),
            elevation = CardDefaults.cardElevation(defaultElevation = 6.dp),
            colors = CardDefaults.cardColors(containerColor = MaterialTheme.colorScheme.primaryContainer)
        ) {
            Column(
                modifier = Modifier.padding(16.dp)
            ) {
                OutlinedTextField(
                    value = updatedTitle,
                    onValueChange = { updatedTitle = it },
                    label = { Text( text: "Edit Title") },
                    modifier = Modifier.fillMaxWidth()
                )

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

                OutlinedTextField(
                    value = updatedDescription,
                    onValueChange = { updatedDescription = it },
                    label = { Text( text: "Edit Description") },
                    modifier = Modifier.fillMaxWidth()
                )

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

Activity and User Input Handling

The app’s primary activity, MainActivity, sets up the main layout using Jetpack Compose. This activity initializes the UI, which includes the task list, task creation fields, and action buttons. Within MainActivity, TaskManagerApp composable displays a top bar, the task input form, and the task list. It leverages Scaffold to manage the overall layout structure.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun TaskManagerApp(taskViewModel: TaskViewModel = viewModel()) {
    val tasks = taskViewModel.taskList.collectAsState()

    var title by remember { mutableStateOf( value: "") }
    var description by remember { mutableStateOf( value: "") }

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text( text: "Task Manager") },
                colors = TopAppBarDefaults.topAppBarColors(
                    containerColor = MaterialTheme.colorScheme.primary,
                    titleContentColor = MaterialTheme.colorScheme.onPrimary
                )
            )
        },
        content = { padding ->
            Column(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(padding)
                    .padding(16.dp)
            ) {

                // Поля для ввода новой задачи
                OutlinedTextField(
                    value = title,
                    onValueChange = { title = it },
                    label = { Text( text: "Task Title") },
                    modifier = Modifier.fillMaxWidth(),
                    colors = TextFieldDefaults.outlinedTextFieldColors(
                        focusedBorderColor = MaterialTheme.colorScheme.primary,
                        cursorColor = MaterialTheme.colorScheme.primary
                    )
                )
```

Compose OutlinedTextField components capture user input for task titles and descriptions. Each field is bound to a mutable state to handle input changes in real time.

```
    OutlinedTextField(
        value = title,
        onValueChange = { title = it },
        label = { Text( text: "Task Title") },
        modifier = Modifier.fillMaxWidth(),
        colors = TextFieldDefaults.outlinedTextFieldColors(
            focusedBorderColor = MaterialTheme.colorScheme.primary,
            cursorColor = MaterialTheme.colorScheme.primary
        )
    )
```

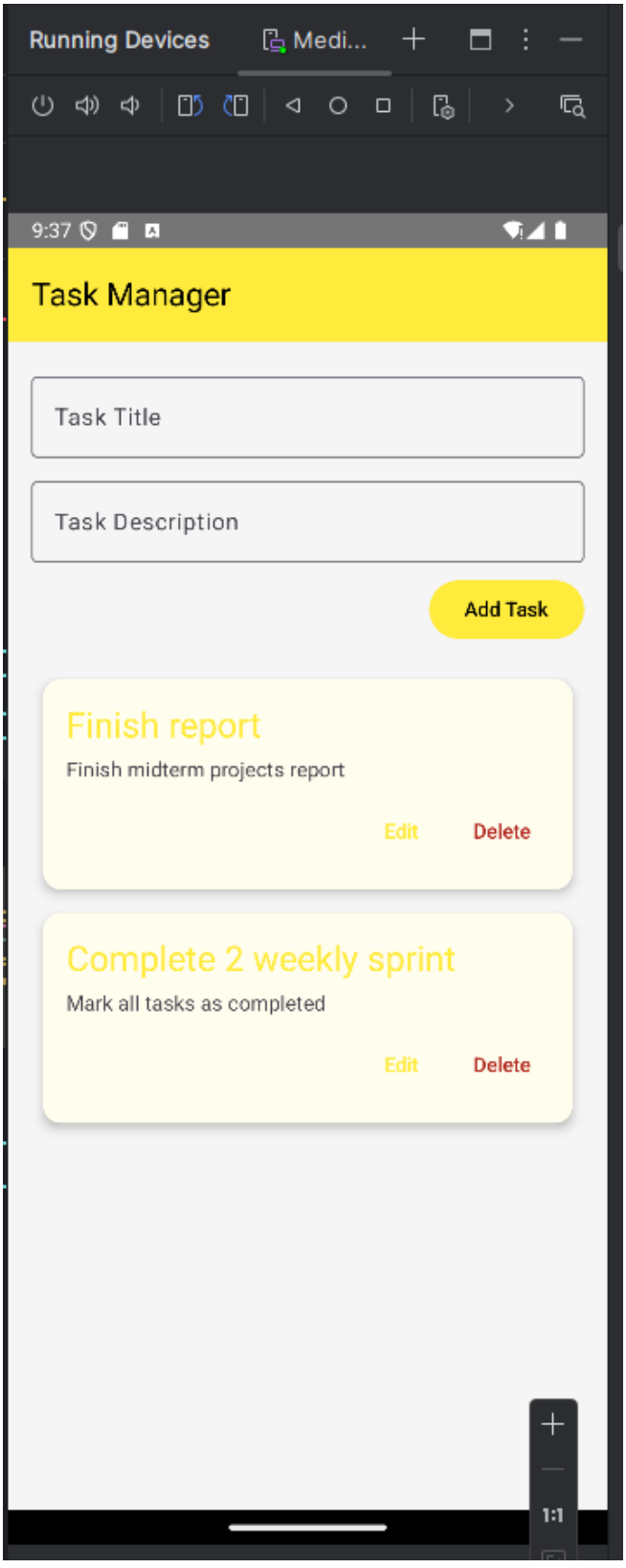


Fig 1. List all tasks

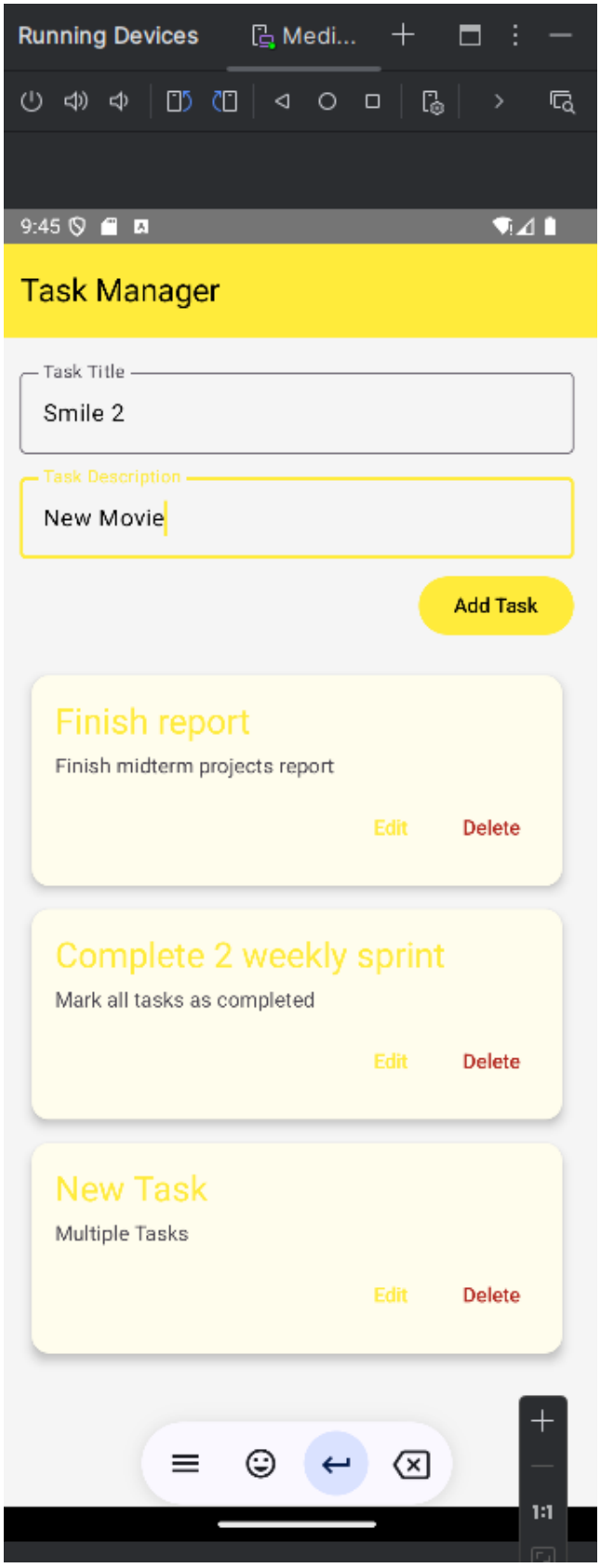


Fig 2. Adding new todo

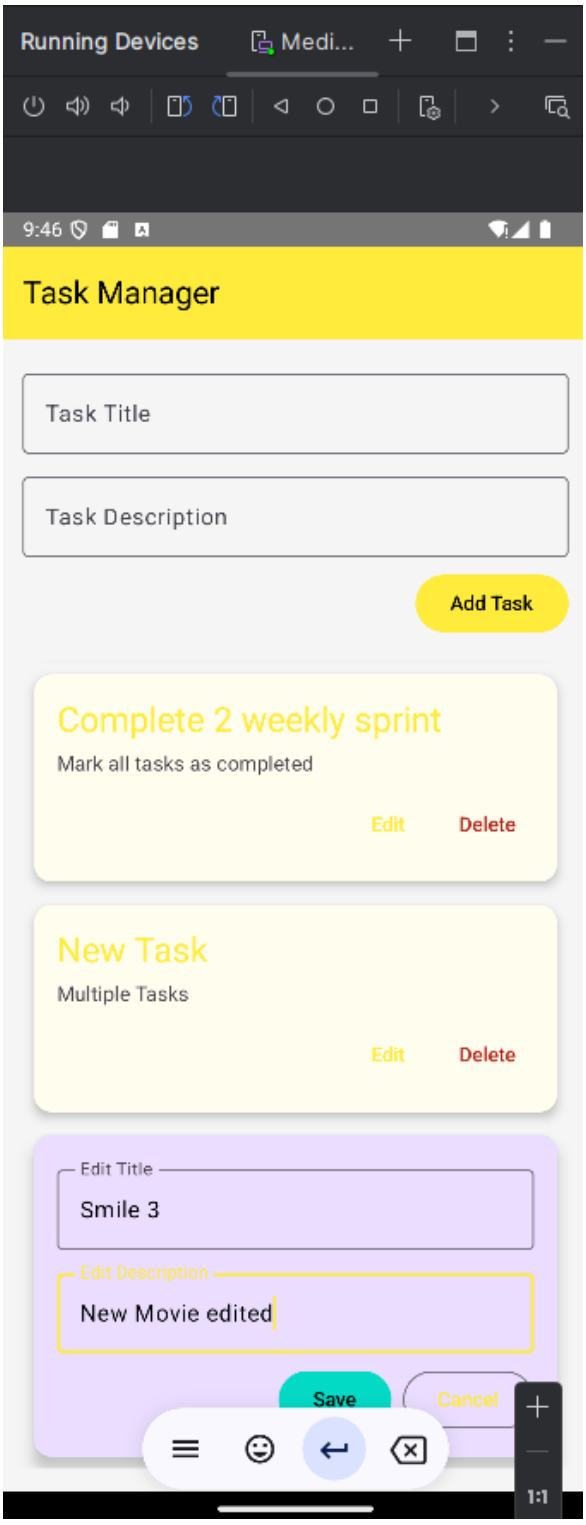


Fig 3. Editing todo

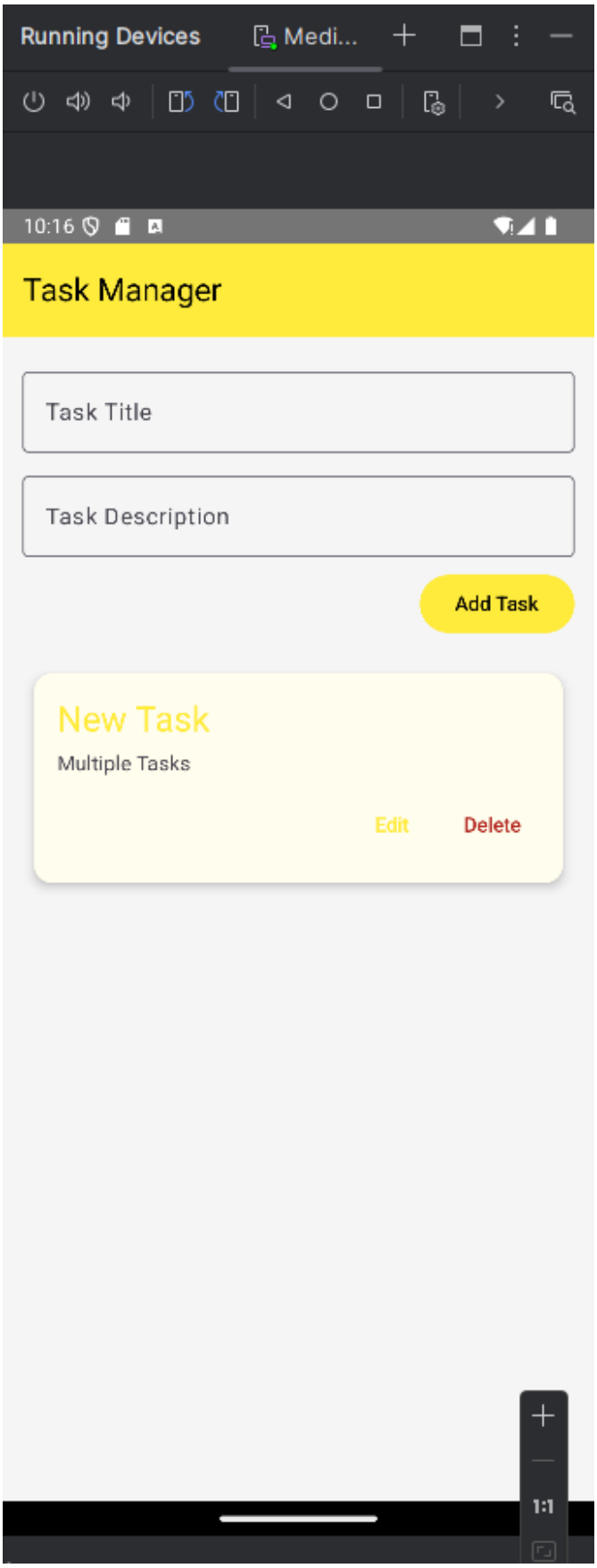


Fig 4. Deleting todo

Activity Lifecycle Management

In Android, lifecycle methods (e.g., onCreate, onStart, onDestroy) play a key role in managing the state of an application, especially when configuration changes or when the application is moved to the background. These methods allow you to manage resources such as the database, network, and user data, and to save the state of the application when it is stopped or restarted. In MainActivity, the onCreate method is used to initialize the interface and set the content using setContent. onStart and onResume can be overridden to resume operations such as data refresh or network requests. onPause and onStop are used to save data or free memory resources while the application is in the background.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
        TaskManagerTheme {
            TaskManagerApp()
        }
    }
}
```

onCreate: Initializes the UI and launches the main content of the application.
This method help manage the state of the application and ensure its stability when conditions change, such as changing configuration or moving the application to the background.

Fragments and Fragment Lifecycle

In an application, fragments are used to separate the user interface into independent components. For example, you can create a separate TaskListFragment to display a list of tasks and a TaskDetailFragment to display task details. Fragments are initialized in the main activity (MainActivity) and added dynamically using FragmentTransaction, which allows for flexible interface management and support for different screen configurations.

```
class TaskListFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        val view = inflater.inflate( resource: 0, container)
        return view
    }

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

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

The fragment lifecycle includes the main methods: onStart, onDestroyView.
Fragments follow the activity lifecycle, but have additional methods (onCreateView and onDestroyView) that allow you to manage UI components.

Conclusion

This project successfully delivered a functional and user-friendly task management application for Android, leveraging Kotlin and Jetpack Compose to create a beauty, dynamic interface. The technologies used—particularly Kotlin's concise syntax, Jetpack Compose's declarative UI approach, and Android Jetpack components—proved effective for building a responsive and maintainable app structure. These tools simplified user input handling, UI state management, and lifecycle handling, resulting in a cleaner codebase and smoother user experience. Potential future features include:

1. Integrating a database to save tasks locally, allowing tasks to persist across app sessions.
2. Adding reminders or notifications to alert users about upcoming tasks.
3. Allowing users to personalize the app's appearance through different themes or color schemes.