Mobile development
Assignment 3, mobile programming
Bitanov Assanali (23MD0392)
10.11.2024

Table of Contents

Introduction

In Assignment 3, I applied my knowledge to build a mobile application with a focus on user interaction and data management. The project was developed using Android's Jetpack Compose framework, a modern toolkit for building native UIs, along with the Room persistence library for local data storage [1]. This assignment involved creating a simple app that allows users to input, view, and manage a list of movies, complete with navigation between multiple screens.

I started with designing a UI using Jetpack Compose to create a simple interface. The UI included various screens where users could navigate to enter data, view saved data, and access a custom logging screen.

To manage user data between screens, I implemented ViewModels, which enabled efficient data handling and ensured that data persisted during configuration changes, such as screen rotations. ViewModels also simplified the retrieval and updating of data within the app, keeping the UI in sync with the underlying data.

The goal of this project is to learn how to work with data in Compose and improve my skills in Android development by using. The result was a functional and interactive app that provides a smooth experience for users to manage their list of movies.

Composable Navigation and State Management

In the first three exercises, I set up the app's main screen and navigation using Jetpack Compose. The MainScreen function, located in MainScreen.kt, serves as the main menu where users can select different screens like Input, Output, Movie List, and Lifecycle Logging. the MainScreen function is shown in Figure 1

.

Figure 1. The MainScreen function in the MainScreen.kt file

In MainScreen, I used buttons to let users switch between screens. Each button is linked to a different screen, replacing the traditional fragment system with Jetpack Compose's NavHost and NavController components. This setup makes switching between screens smooth and avoids the need for fragment transactions.

Instead of using Fragments, I used NavHost to define where each screen goes and how they connect. Each screen is managed within the NavigationGraph.kt file, as shown in Figure 2.

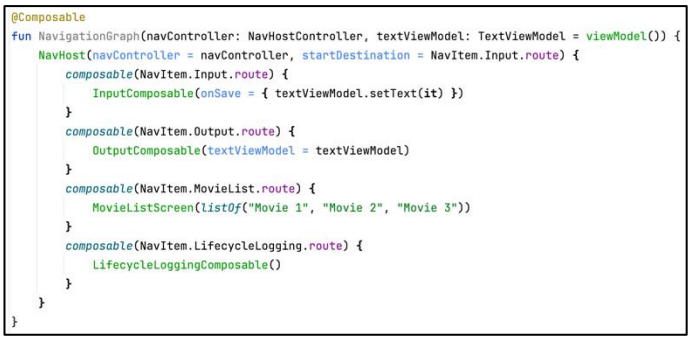

Figure 2. The NavigationGraph composable in the NavigationGraph.kt file

This method makes it easy to control the app's navigation and data sharing between screens using Jetpack Compose. A TextViewModel handles shared data, letting screens share information easily without fragments.

LazyColumn and List Handling in Jetpack Compose

To display a list of movies in the app, I used Jetpack Compose's LazyColumn with the MovieListScreen composable function in the MovieListScreen.kt file. This file allows us to se all the movie names. The MovieListScreen function is shown in Figure 3.

```
@Composable
fun MovieListScreen(movies: List<String>) {
    val context = LocalContext.current

    LazyColumn {
        items(movies) { movie ->
            Text(
                text = movie,
                modifier = Modifier
                    .padding(8.dp)
                    .clickable {
                        Toast.makeText(context, text: "Clicked on $movie", Toast.LENGTH_SHORT).show()
                    }
            )
        }
    }
}
```

Figure 3. The MovieListScreen function in the MovieListScreen.kt file

In Compose, handling clicks is simple and doesn't need a view setup. I added ClickableText for each movie name, so users can easily click on a title. When a movie is clicked, the app shows a Toast message with the movie's name.

Compose takes care of optimizing list items on its own, so we don't need the traditional ViewHolder pattern. LazyColumn is designed to handle item reuse and updates automatically, keeping things simple for developers. Using this approach, I achieve the same functionality as RecyclerView and Adapters but in a way that's more readable and requires less code in Compose.

ViewModel and State Management

In this part of the project, I used Jetpack Compose's ViewModel and State to manage and share data in the app. Instead of the ViewModel and LiveData, I used Compose's ViewModel and MutableState to update the UI based on data changes. I created a TextViewModel class to store and manage user input. It uses MutableState to keep track of the input and update the UI automatically when it changes. The TextViewModel class is shown in Figure 4.

```
class TextViewModel : ViewModel() {
    private val _text = MutableLiveData<String>()
    val text: LiveData<String> get() = _text

    fun setText(newText: String) {
        _text.value = newText
    }
}
```

Figure 4. The TextViewModel class in the TextViewModel.kt file

In the UI, I set up an input field that's directly connected to the TextViewModel. Whenever the user types, the ViewModel's MutableState updates immediately, and Compose re-draws the UI with the new input, all without needing extra code to observe changes.

Results

As a result, we have a fully functional app that allows users to navigate between different sections and interact with the content. The app's main screen serves as the central navigation hub, letting users access the input, output, movie list, and lifecycle logging screens. The main screen is shown in Figure 5.

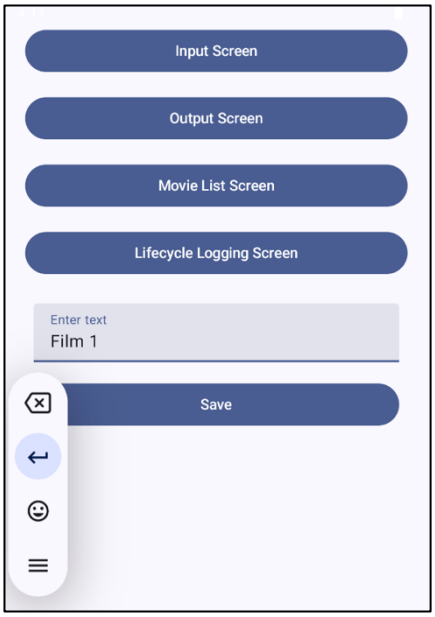

Figure 5. The main page

The main screen has buttons for each section: "Input Screen," "Output Screen," "Movie List Screen," and "Lifecycle Logging Screen." Each button takes the user to that screen. The Input Screen lets users enter text, which they can save by pressing a button. This saved text is shared across the app through a ViewModel, so it can be displayed on other screens.

By pressing the Output Screen button we move to the screen that is shown in Figure 6.

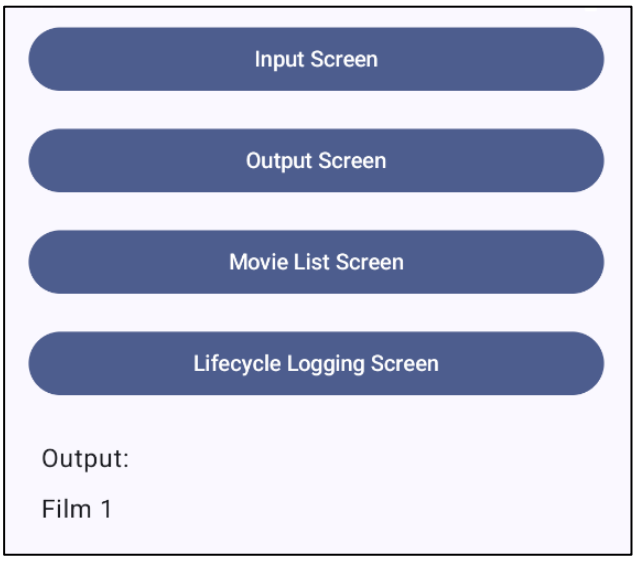

Figure 6. The Output Screen

The Movie List Screen shows a list of favorite movies in a simple, scrollable view. Each movie is a separate item, and clicking on one shows a quick message with the movie's title. This screen is easy to use, letting users scroll and tap to interact with the list. The Movie List Screen is shown in Figure 7.

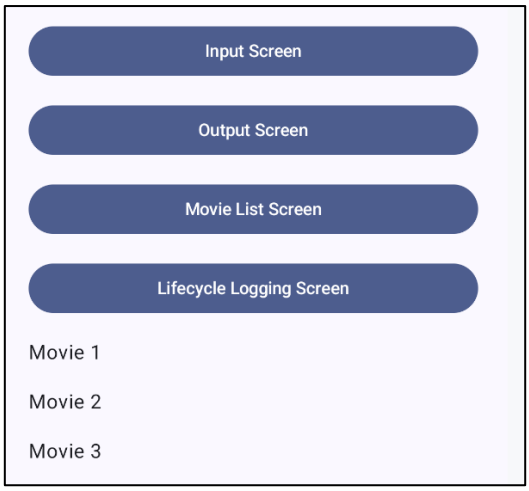

Figure 7. The Movie List Screen

The create new post page allows users to add a new blog post to the site. On this page, there is a form where users can enter the post's title, content, author name, and optionally upload an image. Once the user fills in the details and submits the form, the post is saved and will appear on the main list of posts.

Conclusion

In this project, I focused on building a mobile application using Android's Jetpack Compose framework, aiming to create a user-friendly app with multiple screens for managing a list of movies. The main goal was to explore Compose's declarative UI approach, utilize ViewModels for state management, and create a navigation without using traditional Fragments. Through this assignment, I gained practical knowledge in designing user interfaces and managing data effectively within an Android app.

I started by designing the app's main screens with Jetpack Compose. Using Compose components, I created an interface where users can interact with the app through various screens. The main screen provides buttons for each section: Input Screen, Output Screen, Movie List Screen, and Lifecycle Logging Screen, making navigation simple. Each screen serves a specific function, allowing users to input, view, and manage a list of movies, as well as observe lifecycle events.

To handle data across different screens, I implemented ViewModels. The Input Screen enables users to add movie titles, and the data is then displayed in the Output Screen through a shared ViewModel. Jetpack Compose's navigation system replaced the need for Fragments, offering a simplified way to manage transitions between screens. Using `NavHost` and `NavController`, I was able to define the app's navigation graph, linking each screen without the complexity of traditional Fragment transactions.

In this project, I gained hands-on experience with Jetpack Compose's UI toolkit, state management with ViewModels, and Compose's navigation system. The project reinforced my understanding of building interactive and organized mobile applications in Android. Working through these exercises helped me develop skills in managing data, creating flexible UI components, and enhancing user experience, all of which are crucial for future Android development projects.

References

1. State and Jetpack Compose [Android Developers] https://developer.android.com/develop/ui/compose/state