KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Assignment 3
Mobile Programming
Implementing UI components and State Management with Jetpack Compose

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

Almaty, 2024

Table of Contents

Introduction 3
Jetpack Compose and Modern Android UI Development 4
Exercise 1: Creating a Basic Composable Function 4
Exercise 2: Composable Communication 6
Exercise 3: Navigation Between Composables 7
LazyColumn and List Handling in Compose 9
Exercise 4: Building a LazyColumn List 9
Exercise 5: Item Click Handling 10
Exercise 6: Efficiency in LazyColumn 11
ViewModel and State Management with LiveData and StateFlow 11
Exercise 7: Implementing ViewModel with StateFlow 11
Exercise 8: MutableState for Input Handling 13
Exercise 9: Data Persistence with Room Database 15
Results 17
Conclusion 18
References 19
Appendices 20

Introduction

Jetpack Compose is a game-changer in every way for Android development, and it provides a new set of tools to natively construct declarative UI. In this report, I dive deep into how one can implement UI components and handle state using Jetpack Compose as an alternative for the traditional use of XML layouts, fragments, and RecyclerViews. I am deepening the exercises: composable functions creation, state management with ViewModel and LiveData Lists management with LazyColumn, data persistence with Room.

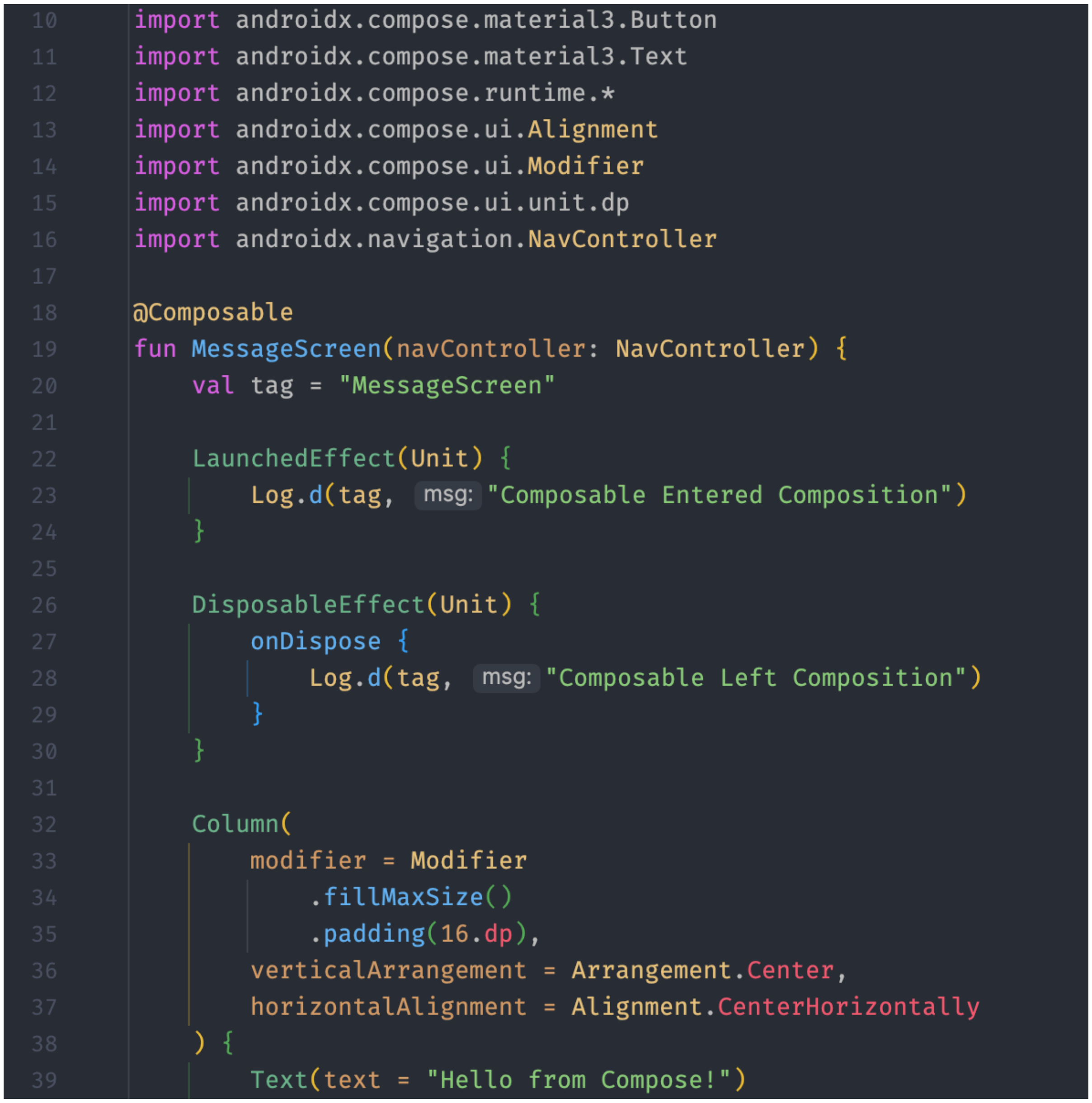
The developers will be able to use building blocks for any modern Android application to craft efficient, scalable, and maintainable applications. Moving over to Jetpack Compose makes UI development easier; it's more performant and intuitive in terms of code structure. This report attempts to describe these benefits by implementing concrete examples of various implementations and show why the adoption of Jetpack Compose is important in modern Android development.

Jetpack Compose and Modern Android UI Development

Exercise 1: Creating a Basic Composable Function

The goal of this exercise is to create a simple composable displaying a message ("Hello from Compose!") and to implement logging of its lifecycle. More precisely, using LaunchedEffect and DisposableEffect I'll try to log when the composable is appearing and disappearing from composition.

Setup of the project for Jetpack Compose is the first step, adding dependencies and necessary plugins in the build.gradle file. Then comes enabling Jetpack Compose in the Android application module, which supports the composable functions.



```kotlin
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.navigation.NavController

@Composable
fun MessageScreen(navController: NavController) {
    val tag = "MessageScreen"

    LaunchedEffect(Unit) {
        Log.d(tag,  msg: "Composable Entered Composition")
    }

    DisposableEffect(Unit) {
        onDispose {
            Log.d(tag,  msg: "Composable Left Composition")
        }
    }

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(text = "Hello from Compose!")
```

Figure 1. Composable Message Screen

Next, I implement the composable function MessageScreen. This displays a Text composable with "Hello from Compose!". I implement LaunchedEffect (Unit) that will log when the composable has entered the composition and a DisposableEffect(Unit) with an onDispose callback that will log when the composable has left the composition.

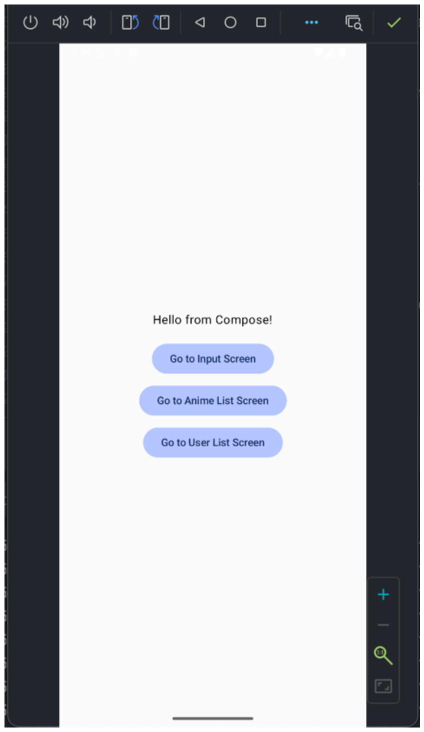
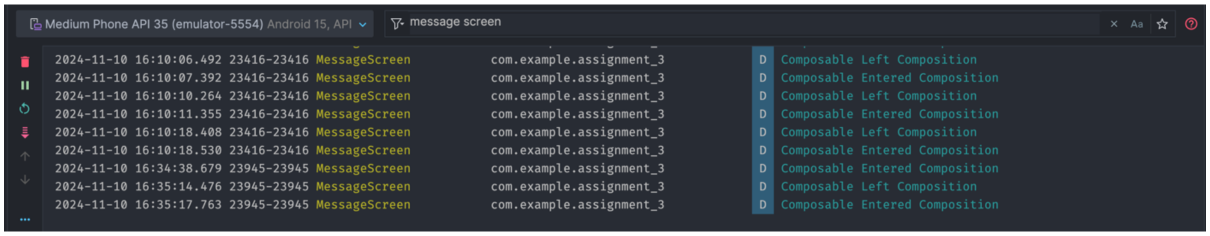


Figure 2. Hello from Compose message and Main Screen

When the app runs, it should display the text "Hello from Compose!" in the middle of the screen. It also logs some information about what happens when the composable is added to and removed from composition, thus lifecycle events.



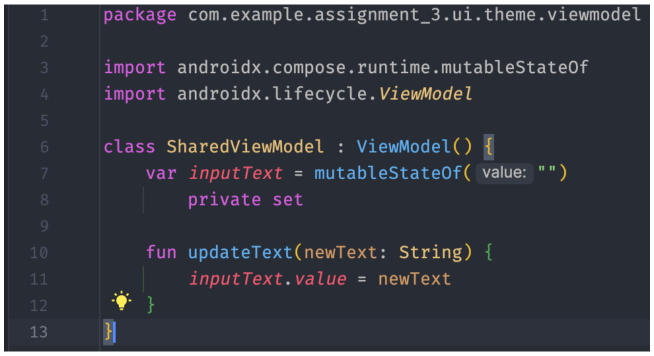
```
2024-11-10 16:10:06.492 23416-23416 MessageScreen          com.example.assignment_3          D  Composable Left Composition
2024-11-10 16:10:07.392 23416-23416 MessageScreen          com.example.assignment_3          D  Composable Entered Composition
2024-11-10 16:10:10.264 23416-23416 MessageScreen          com.example.assignment_3          D  Composable Left Composition
2024-11-10 16:10:11.355 23416-23416 MessageScreen          com.example.assignment_3          D  Composable Entered Composition
2024-11-10 16:10:18.408 23416-23416 MessageScreen          com.example.assignment_3          D  Composable Left Composition
2024-11-10 16:10:18.530 23416-23416 MessageScreen          com.example.assignment_3          D  Composable Entered Composition
2024-11-10 16:34:38.679 23945-23945 MessageScreen          com.example.assignment_3          D  Composable Entered Composition
2024-11-10 16:35:14.476 23945-23945 MessageScreen          com.example.assignment_3          D  Composable Left Composition
2024-11-10 16:35:17.763 23945-23945 MessageScreen          com.example.assignment_3          D  Composable Entered Composition
```

Figure 3. Logging Messages

In MessageScreen, I use LaunchedEffect(Unit) to execute a side effect when the composable enters the composition, logging "Composable Entered Composition". The DisposableEffect(Unit) provides an onDispose callback that logs "Composable Left Composition" when the composable is removed from the composition. The Text composable displays the message on the screen

Exercise 2: Composable Communication

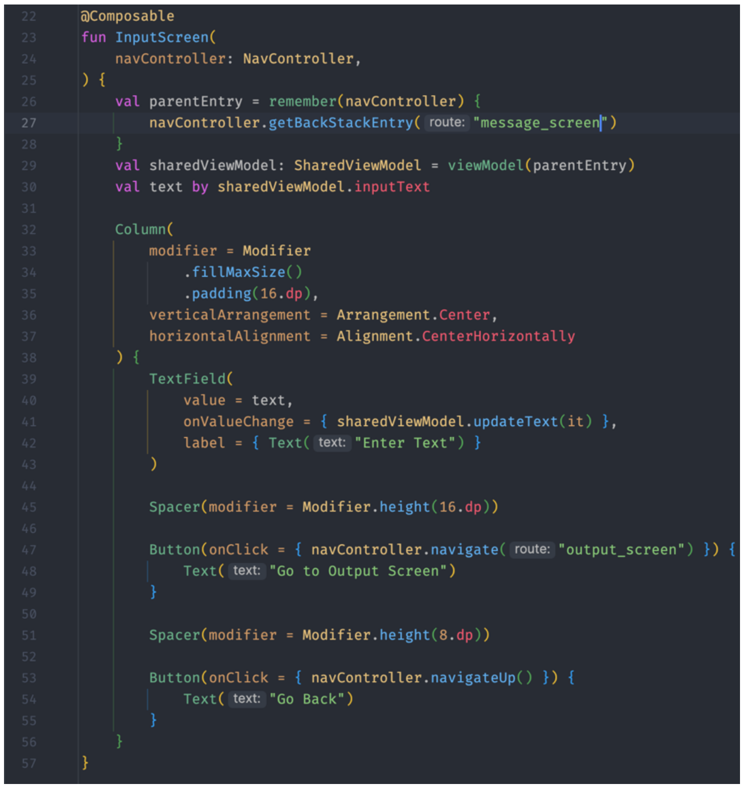
This exercise is going to be composed of two screens: one screen having a text field for input, and the other one displaying text as output. To share the data between these two screens, I implement ViewModel to show the text written in the input screen on the output screen.



```kotlin
package com.example.assignment_3.ui.theme.viewmodel

import androidx.compose.runtime.mutableStateOf
import androidx.lifecycle.ViewModel

class SharedViewModel : ViewModel() {
    var inputText = mutableStateOf( value: "")
        private set

    fun updateText(newText: String) {
        inputText.value = newText
    }
}
```

Figure 4. Shared View Model Snippet

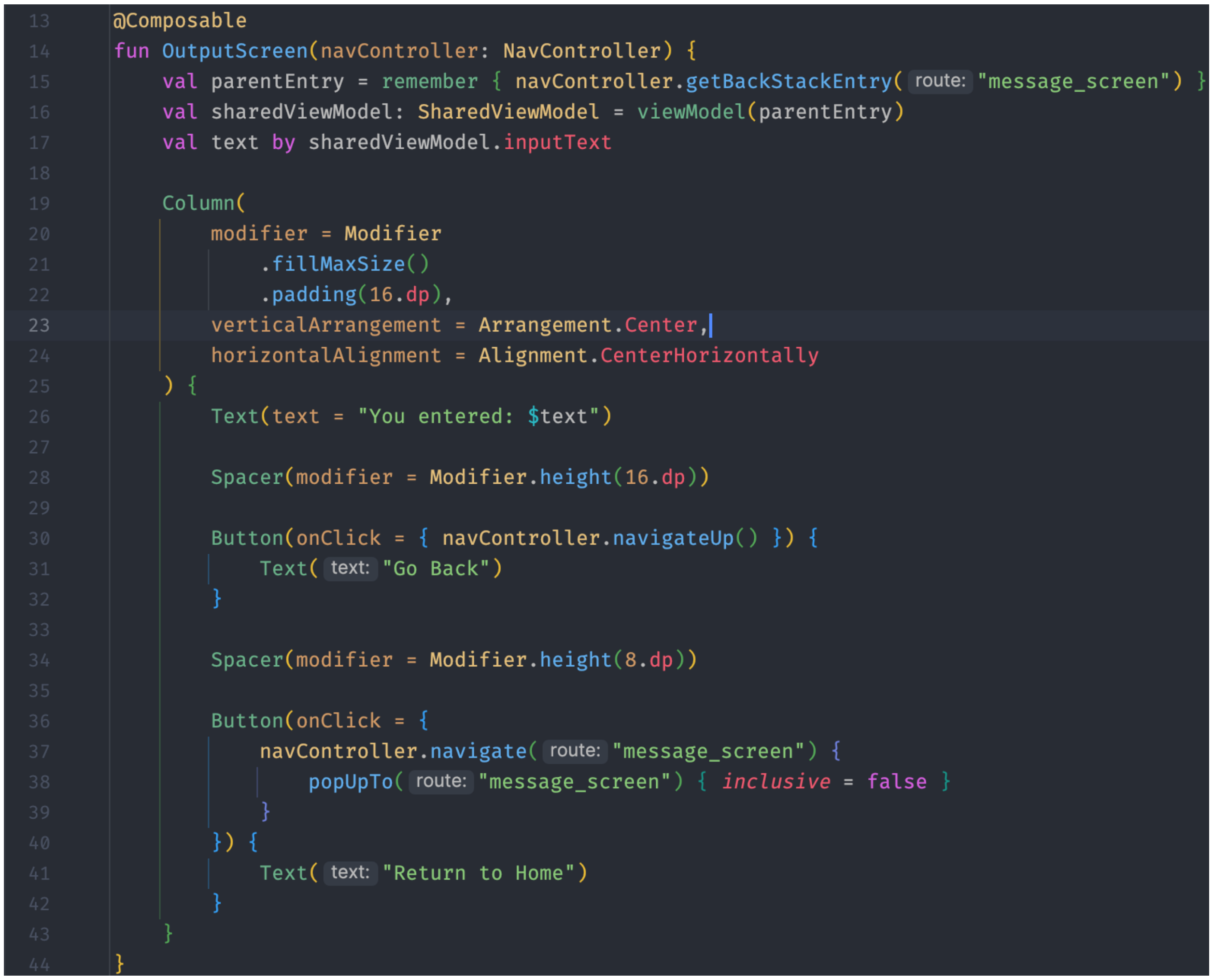
First, I define a SharedViewModel class that extends ViewModel. In it, I define the MutableState property - inputText, initialized with an empty string, and then add the method updateText(newText: String) that updates its value.



```kotlin
@Composable
fun InputScreen(
    navController: NavController,
) {
    val parentEntry = remember(navController) {
        navController.getBackStackEntry( route: "message_screen")
    }
    val sharedViewModel: SharedViewModel = viewModel(parentEntry)
    val text by sharedViewModel.inputText

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        TextField(
            value = text,
            onValueChange = { sharedViewModel.updateText(it) },
            label = { Text( text: "Enter Text") }
        )

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

        Button(onClick = { navController.navigate( route: "output_screen") }) {
            Text( text: "Go to Output Screen")
        }

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

        Button(onClick = { navController.navigateUp() }) {
            Text( text: "Go Back")
        }
    }
}
```

Figure 5. Input Screen Composable

Next, I create the InputScreen composable. It contains a TextField whose value is bound to sharedViewModel.inputText. When the user types in text, sharedViewModel.updateText(it) updates the state in the ViewModel. A button labeled "Go to Output Screen" navigates to the OutputScreen using the NavController.



```kotlin
@Composable
fun OutputScreen(navController: NavController) {
    val parentEntry = remember { navController.getBackStackEntry( route: "message_screen") }
    val sharedViewModel: SharedViewModel = viewModel(parentEntry)
    val text by sharedViewModel.inputText

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(text = "You entered: $text")

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

        Button(onClick = { navController.navigateUp() }) {
            Text( text: "Go Back")
        }

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

        Button(onClick = {
            navController.navigate( route: "message_screen") {
                popUpTo( route: "message_screen") { inclusive = false }
            }
        }) {
            Text( text: "Return to Home")
        }
    }
}
```

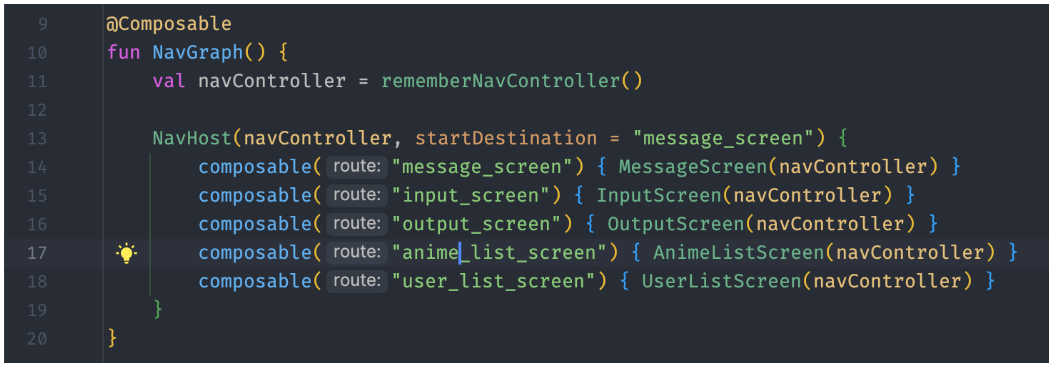
Figure 6. Output Screen Composable

I will create the OutputScreen composable, observing sharedViewModel.inputText and displaying text through a Text composable. Now, for the app to switch between screens, I must configure Jetpack Compose Navigation: add dependencies and provide NavHost in NavGraph.kt. I define routes for InputScreen and OutputScreen.

The application should provide the user with an opportunity to enter text on the InputScreen and then go to OutputScreen showing the text just entered by a user. The ViewModel should save state while navigation.

Exercise 3: Navigation Between Composables

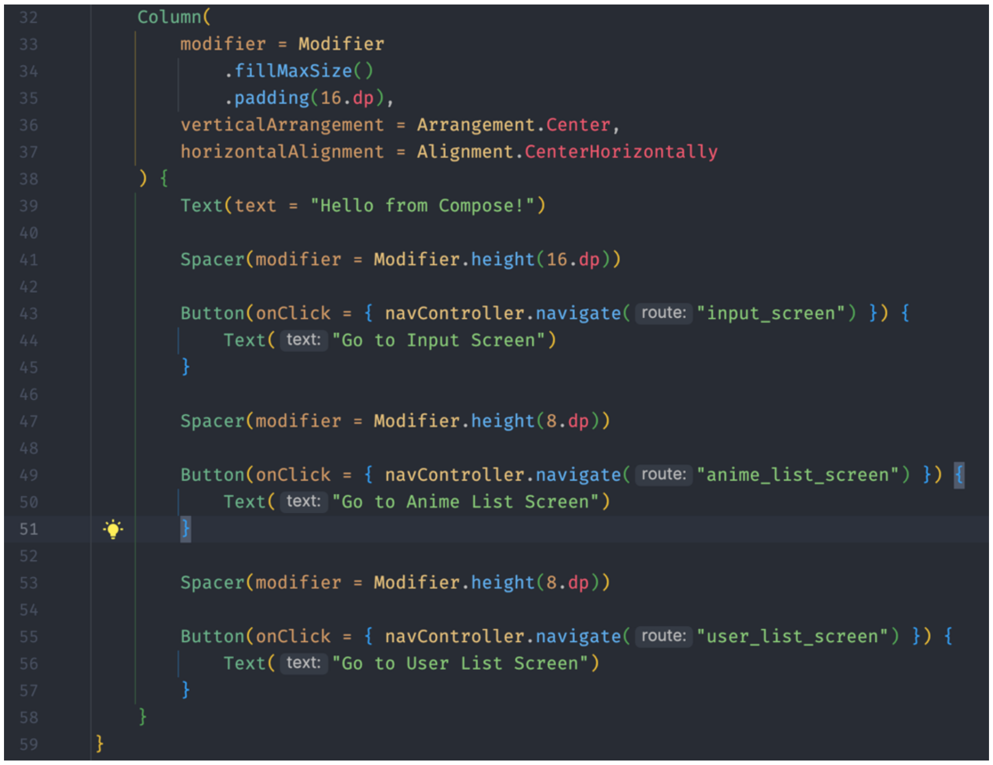
This application should provide an opportunity to host several composable screens, and I must implement some buttons to switch between them using the Compose Navigation Component. Let us follow a few actions to implement this. I added the dependency of navigation-compose in my project-level build.gradle file. Then I create a NavController using rememberNavController() and set up a NavHost in NavGraph.kt, specifying routes for each composable screen.



```kotlin
@Composable
fun NavGraph() {
    val navController = rememberNavController()

    NavHost(navController, startDestination = "message_screen") {
        composable( route: "message_screen") { MessageScreen(navController) }
        composable( route: "input_screen") { InputScreen(navController) }
        composable( route: "output_screen") { OutputScreen(navController) }
        composable( route: "anime_list_screen") { AnimeListScreen(navController) }
        composable( route: "user_list_screen") { UserListScreen(navController) }
    }
}
```

Figure 7. Navigation Graph Snippet

I create a MainScreen composable which is the entry of our app. Inside this screen, I place buttons that go to InputScreen and now to a new screen, for example - AnimeListScreen. Each button calls navController.navigate("route_name") with the wanted route. Then, I make sure that each screen receives a NavController parameter for managing navigation and can go back by using navController.navigateUp().



```kotlin
        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(16.dp),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Text(text = "Hello from Compose!")

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

            Button(onClick = { navController.navigate( route: "input_screen") }) {
                Text( text: "Go to Input Screen")
            }

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

            Button(onClick = { navController.navigate( route: "anime_list_screen") }) {
                Text( text: "Go to Anime List Screen")
            }

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

            Button(onClick = { navController.navigate( route: "user_list_screen") }) {
                Text( text: "Go to User List Screen")
            }
        }
    }
```

Figure 8. Navigation Control in Main Screen

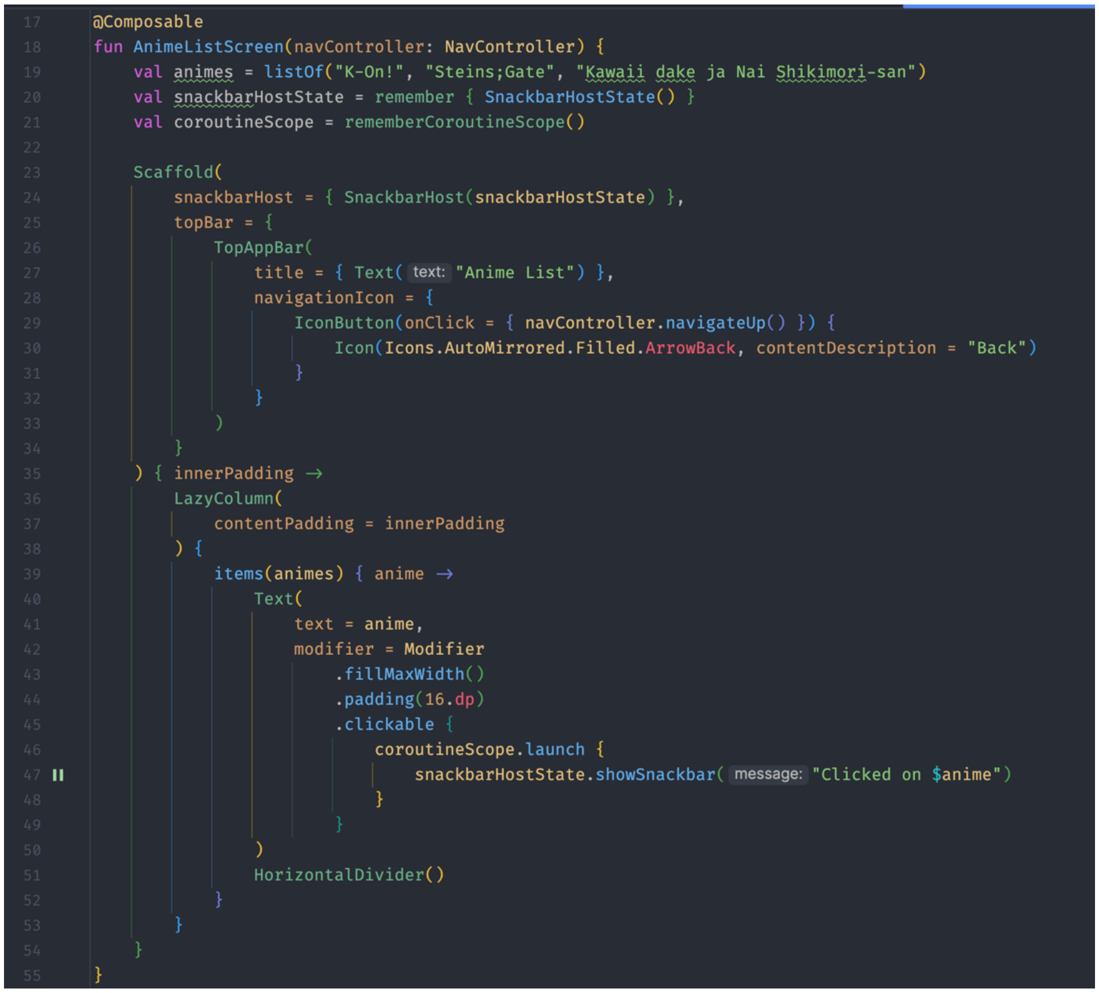
Application has a main screen with buttons so that the user can switch between screens without any snag in transitions. Transitions will be smooth, with an ability to go back to the main screen. In NavGraph, I define the navigation routes and associate them with composable functions. The MainScreen provides buttons to navigate to InputScreen and AnimeListScreen. Each button uses navController.navigate("route_name") to navigate to the respective screen. The NavController is passed to each screen to manage navigation.

LazyColumn and List Handling in Compose

Exercise 4: Building a LazyColumn List

The target is to have the following listing appearing in a LazyColumn so items can show up, for instance, some favorite anime. This exercise shows how to efficiently display data with using RecyclerView in Jetpack Compose.

I define a data source, which is a list of anime titles to be displayed. In the AnimeListScreen composable, I use LazyColumn to render this list. Inside LazyColumn, I use the items function for iterating through the list of anime.



```kotlin
@Composable
fun AnimeListScreen(navController: NavController) {
    val animes = listOf("K-On!", "Steins;Gate", "Kawaii dake ja Nai Shikimori-san")
    val snackbarHostState = remember { SnackbarHostState() }
    val coroutineScope = rememberCoroutineScope()

    Scaffold(
        snackbarHost = { SnackbarHost(snackbarHostState) },
        topBar = {
            TopAppBar(
                title = { Text( text: "Anime List") },
                navigationIcon = {
                    IconButton(onClick = { navController.navigateUp() }) {
                        Icon(Icons.AutoMirrored.Filled.ArrowBack, contentDescription = "Back")
                    }
                }
            )
        }
    ) { innerPadding ->
        LazyColumn(
            contentPadding = innerPadding
        ) {
            items(animes) { anime ->
                Text(
                    text = anime,
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(16.dp)
                        .clickable {
                            coroutineScope.launch {
                                snackbarHostState.showSnackbar( message: "Clicked on $anime")
                            }
                        }
                )
                HorizontalDivider()
            }
        }
    }
}
```

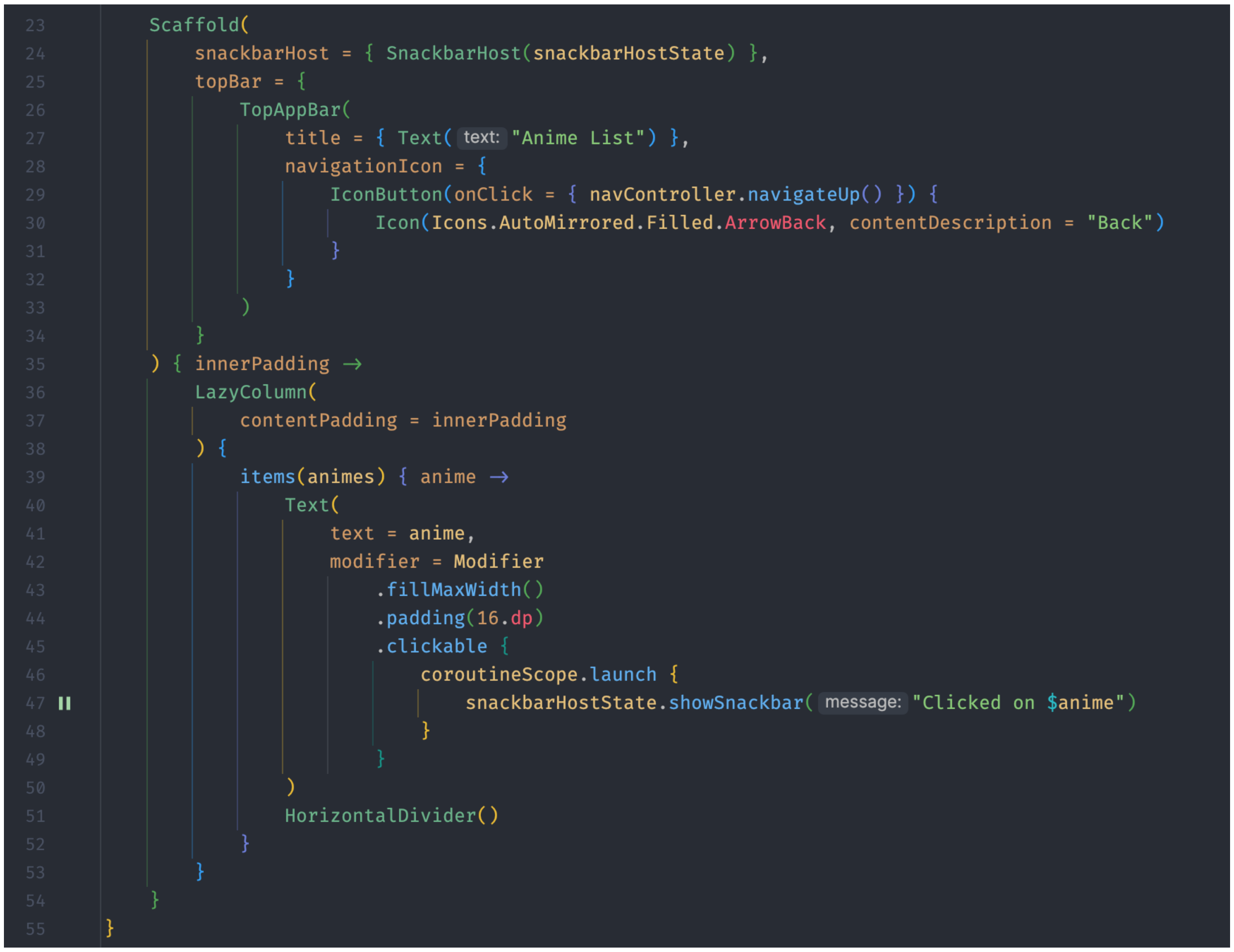
Figure 9. Anime List Screen

For each anime I present, I am showing the title using the Text composable. To visually add some line spacing between items, I add a Divider.

It shows the scrollable list of anime titles, with appropriate spacing and separated by dividers. LazyColumn efficiently renders only the visible items on the screen, improving performance for large lists. The item function simplifies the process of displaying each anime title. The Modifier is used to set the width and padding for each Text composable.

Exercise 5: Item Click Handling

I expand the above exercise by adding the click handler for each of the items in the list. On clicking the item, it should show the name of the clicked item in a Snackbar. In AnimeListScreen I wrap the Text composable with Modifier.clickable making the items click responsive. I create a SnackbarHostState to control the snackbar appearance.



```
    Scaffold(
        snackbarHost = { SnackbarHost(snackbarHostState) },
        topBar = {
            TopAppBar(
                title = { Text( text: "Anime List") },
                navigationIcon = {
                    IconButton(onClick = { navController.navigateUp() }) {
                        Icon(Icons.AutoMirrored.Filled.ArrowBack, contentDescription = "Back")
                    }
                }
            )
        }
    ) { innerPadding ->
        LazyColumn(
            contentPadding = innerPadding
        ) {
            items(animes) { anime ->
                Text(
                    text = anime,
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(16.dp)
                        .clickable {
                            coroutineScope.launch {
                                snackbarHostState.showSnackbar( message: "Clicked on $anime")
                            }
                        }
                )
                HorizontalDivider()
            }
        }
    }
}
```

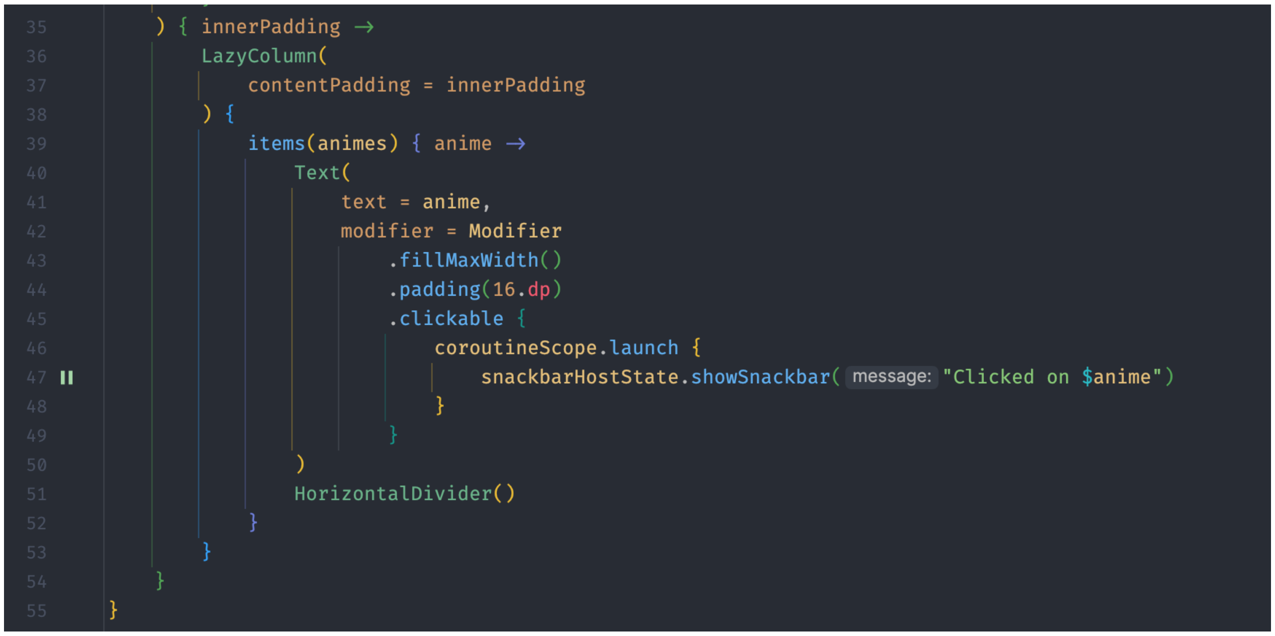
Figure 10. Item OnClick Handling

I use rememberCoroutineScope to get a coroutine scope in composable. On item click, it launches a coroutine to call the suspend function snackbarHostState.showSnackbar("Clicked on $anime"). Next, I will update the Scaffold composable by adding a snackbarHost parameter and passing SnackbarHostState. OnClick of any anime title in the list, at the bottom of the screen, a snackbar with the text "Clicked on [Anime Name]" should appear.

Exercise 6: Efficiency in LazyColumn

It explains why the pattern ViewHolder is redundant in Jetpack Compose and how LazyColumn works efficiently to handle list items. Compose manages the state and UI recomposition in such a way as to avoid the overhead that happens in the traditional ViewHolder pattern used in RecyclerView.

Compose takes advantage of composable functions and the declarative UI paradigm to dynamically compose and dispose of UI nodes according to whether they are needed. This ensures top-notch performance. List items will be automatically optimized in LazyColumn and makes the ViewHolder pattern not necessary with Jetpack Compose.



```
    ) { innerPadding ->
        LazyColumn(
            contentPadding = innerPadding
        ) {
            items(animes) { anime ->
                Text(
                    text = anime,
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(16.dp)
                        .clickable {
                            coroutineScope.launch {
                                snackbarHostState.showSnackbar( message: "Clicked on $anime")
                            }
                        }
                )
                HorizontalDivider()
            }
        }
    }
}
```

Figure 11. Lazy Column Snippet

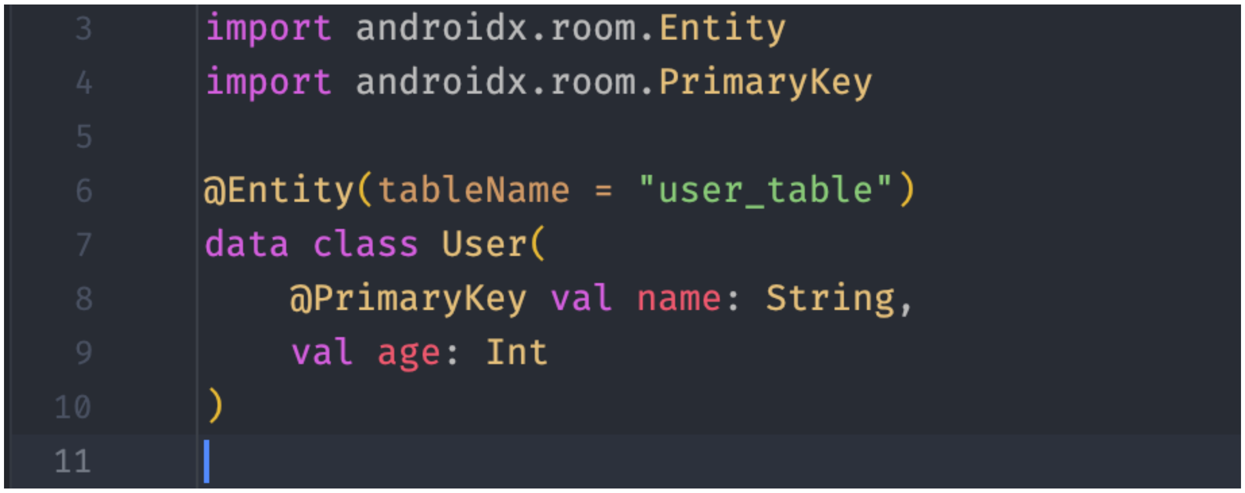
By default, Android's RecyclerView relies on the recycling of views by the ViewHolder pattern on scroll for long lists to improve performance. Jetpack Compose's LazyColumn simplifies this process. It manages the composition of items internally; it only composes the items that become visible on screen and discards items when they are not needed anymore.

This solution minimizes boilerplate code and possible errors associated with the use of the ViewHolder pattern. The declarative nature of Compose takes away this complexity, as state management is internal and very efficient, allowing a developer to think only about the UI logic and not about performance optimizations.

ViewModel and State Management with LiveData and StateFlow

Exercise 7: Implementing ViewModel with StateFlow

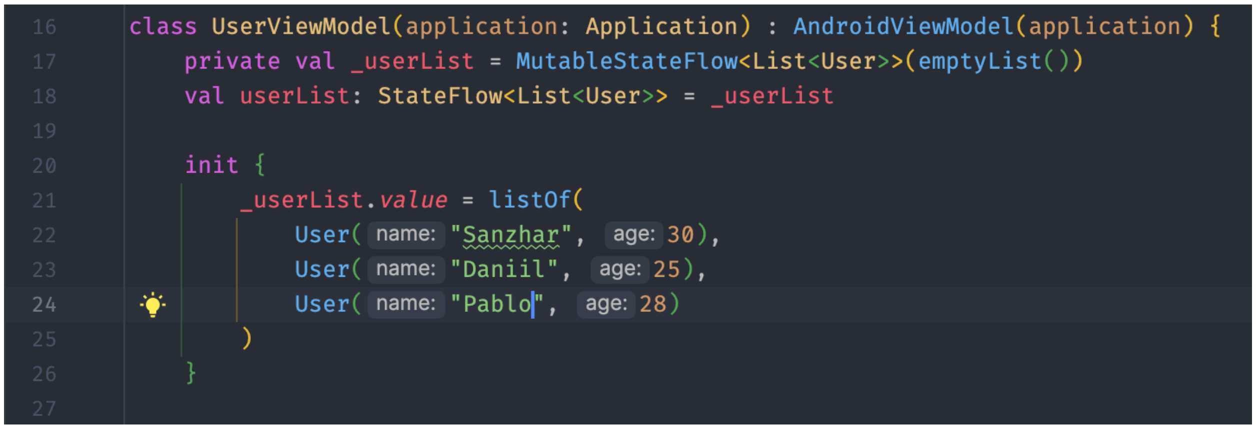
This exercise involves creating a ViewModel that keeps a list of items. Then, with the help of StateFlow, observe changes of data in the ViewModel in a composable. Thereafter, update UI reactively when the data changes.



```kotlin
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "user_table")
data class User(
    @PrimaryKey val name: String,
    val age: Int
)

```

Figure 12. User Data Class

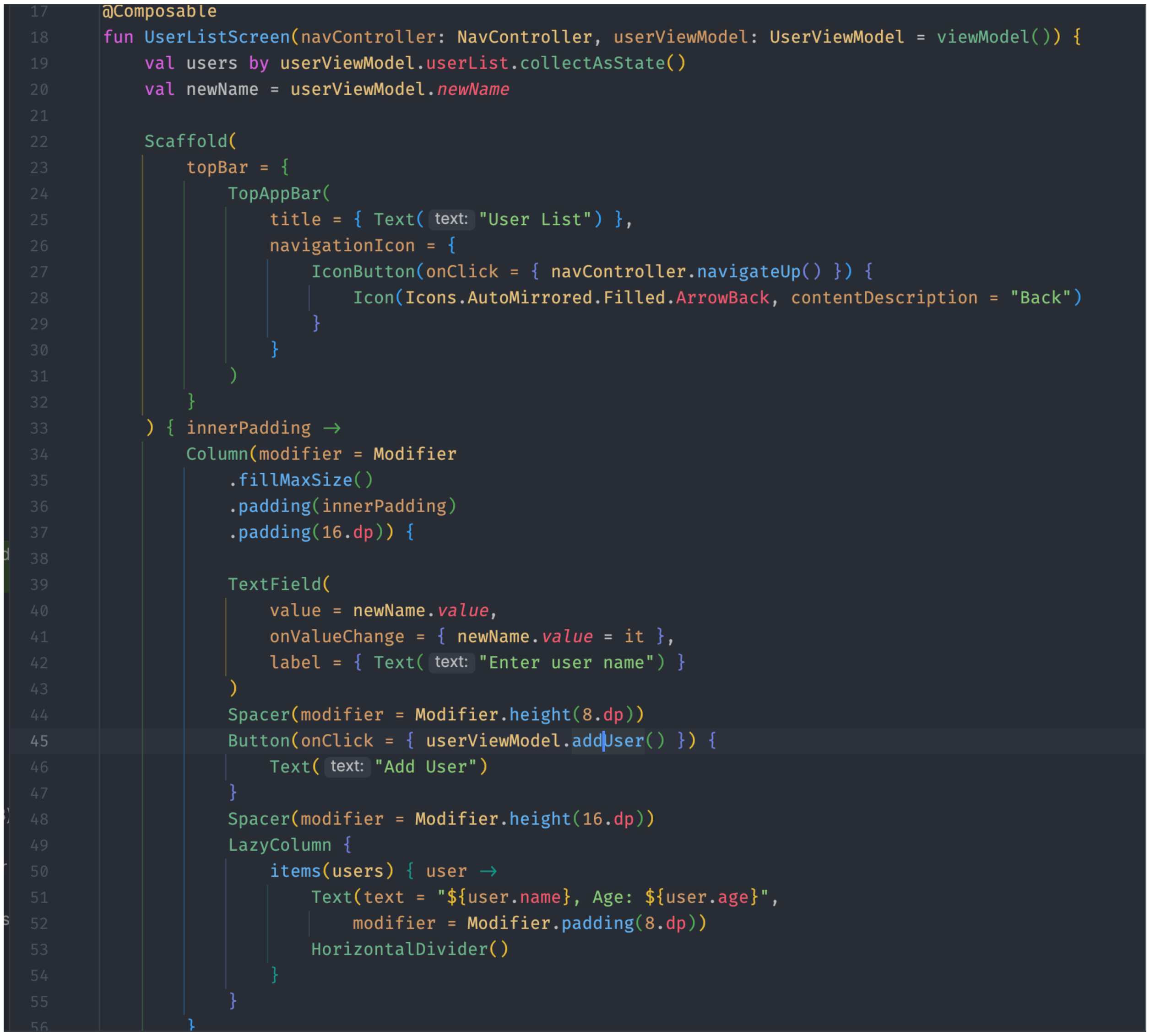
First, define a data class User with properties name and age. I then create UserViewModel, extending ViewModel. I use MutableStateFlow to hold the list of users and expose it as an immutable StateFlow. I initialize the user list with sample data.



```kotlin
class UserViewModel(application: Application) : AndroidViewModel(application) {
    private val _userList = MutableStateFlow<List<User>>(emptyList())
    val userList: StateFlow<List<User>> = _userList

    init {
        _userList.value = listOf(
            User( name: "Sanzhar",  age: 30),
            User( name: "Daniil",  age: 25),
            User( name: "Pablo",  age: 28)
        )
    }

```

Figure 13. User View Model Sample Data

In the UserListScreen composable, I get an instance of UserViewModel by calling viewModel(). I collect the userList StateFlow as state using collectAsState(). Then, show the users in a LazyColumn showing each user's name and age. It should show a list of users. Changes to the user list in the ViewModel should be updated on the UI.



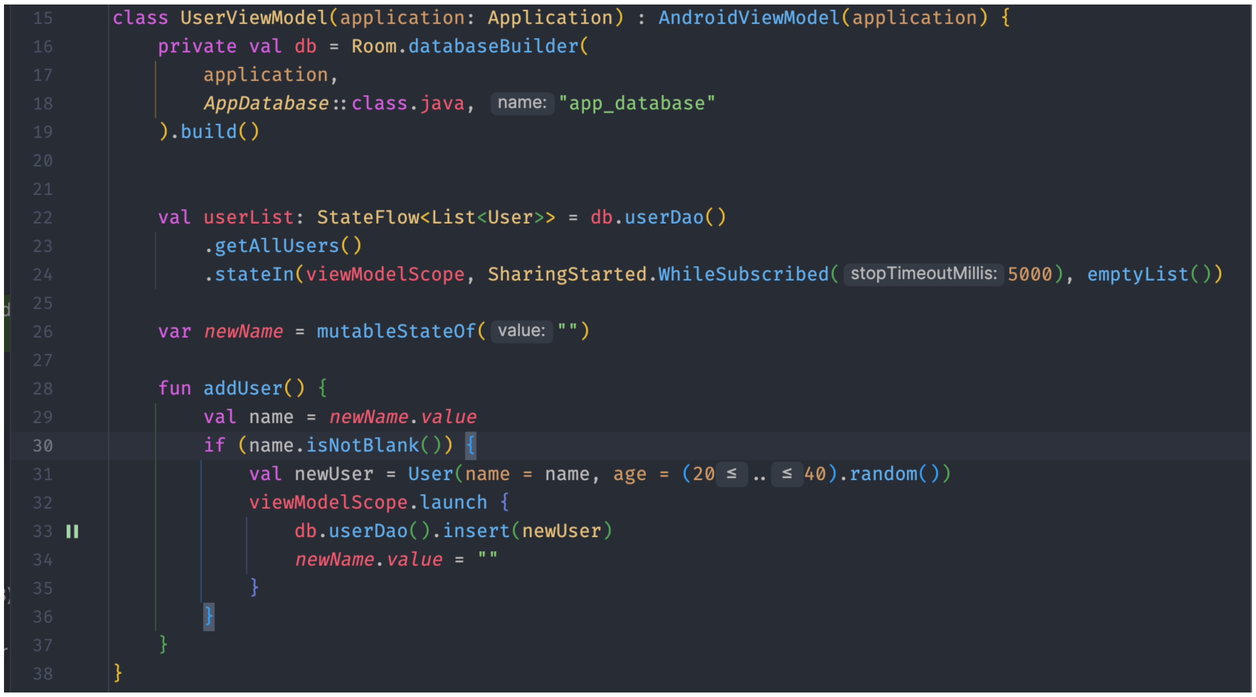
```
@Composable
fun UserListScreen(navController: NavController, userViewModel: UserViewModel = viewModel()) {
    val users by userViewModel.userList.collectAsState()
    val newName = userViewModel.newName

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text( text: "User List") },
                navigationIcon = {
                    IconButton(onClick = { navController.navigateUp() }) {
                        Icon(Icons.AutoMirrored.Filled.ArrowBack, contentDescription = "Back")
                    }
                }
            )
        }
    ) { innerPadding ->
        Column(modifier = Modifier
            .fillMaxSize()
            .padding(innerPadding)
            .padding(16.dp)) {

            TextField(
                value = newName.value,
                onValueChange = { newName.value = it },
                label = { Text( text: "Enter user name") }
            )
            Spacer(modifier = Modifier.height(8.dp))
            Button(onClick = { userViewModel.addUser() }) {
                Text( text: "Add User")
            }
            Spacer(modifier = Modifier.height(16.dp))
            LazyColumn {
                items(users) { user ->
                    Text(text = "${user.name}, Age: ${user.age}",
                        modifier = Modifier.padding(8.dp))
                    HorizontalDivider()
                }
            }
        }
```

Figure 14. User List Screen

The UserViewModel maintains the user list using MutableStateFlow. In the composable, I observe userViewModel.userList using collectAsState(), which allows the UI to reactively update when the data changes. The LazyColumn displays each user's details.

Exercise 8: MutableState for Input Handling

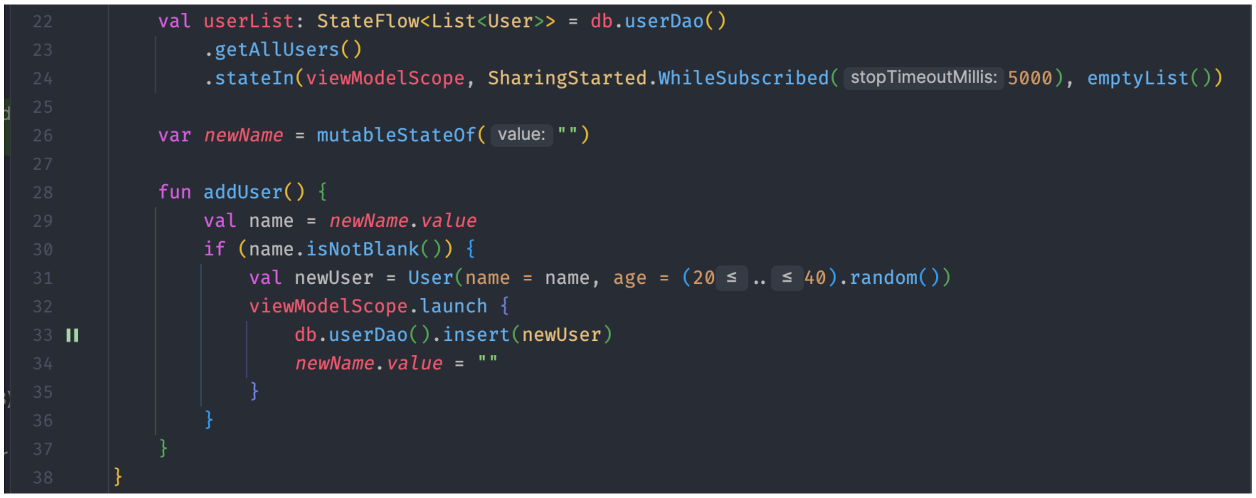
I extend the UserViewModel from the previous exercise to handle the user input via MutableState. Our target in this exercise is to provide an input field in UI that updates the view model and observe the changes to reflect the change in UI.



```kotlin
class UserViewModel(application: Application) : AndroidViewModel(application) {
    private val db = Room.databaseBuilder(
        application,
        AppDatabase::class.java,  name: "app_database"
    ).build()


    val userList: StateFlow<List<User>> = db.userDao()
        .getAllUsers()
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed( stopTimeoutMillis: 5000), emptyList())

    var newName = mutableStateOf( value: "")

    fun addUser() {
        val name = newName.value
        if (name.isNotBlank()) {
            val newUser = User(name = name, age = (20 ≤ .. ≤ 40).random())
            viewModelScope.launch {
                db.userDao().insert(newUser)
                newName.value = ""
            }
        }
    }
}
```

Figure 15. Mutable State for Input Handling

To achieve the above, I first add a MutableState property in the UserViewModel, newName, to hold user input. Then I create an addUser() function, which adds a new User to the _userList if newName is not blank.



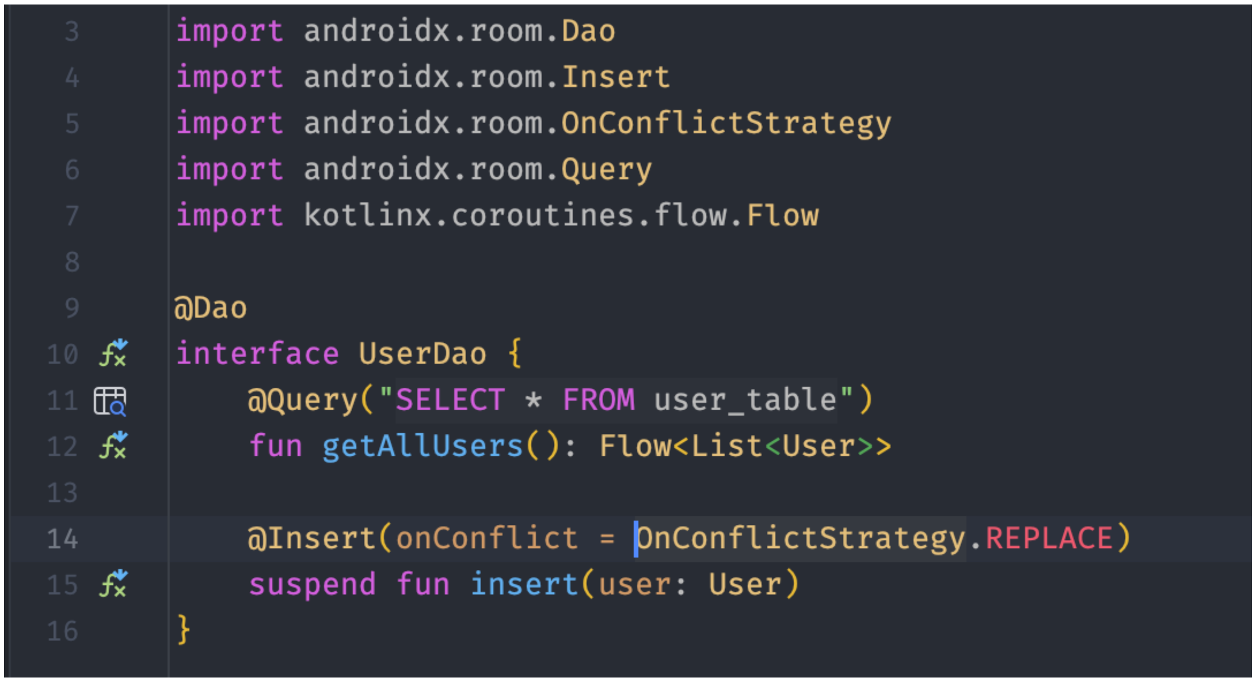
```kotlin
    val userList: StateFlow<List<User>> = db.userDao()
        .getAllUsers()
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed( stopTimeoutMillis: 5000), emptyList())

    var newName = mutableStateOf( value: "")

    fun addUser() {
        val name = newName.value
        if (name.isNotBlank()) {
            val newUser = User(name = name, age = (20 ≤ .. ≤ 40).random())
            viewModelScope.launch {
                db.userDao().insert(newUser)
                newName.value = ""
            }
        }
    }
}
```

Figure 16. Add User Function

I use MutableState for newName to handle two-way data binding between the TextField and the ViewModel. When the user types in the TextField, newName.value updates. When the "Add User" button is clicked, addUser() adds the new user, and the UI updates automatically due to the reactive nature of StateFlow.

Exercise 9: Data Persistence with Room Database

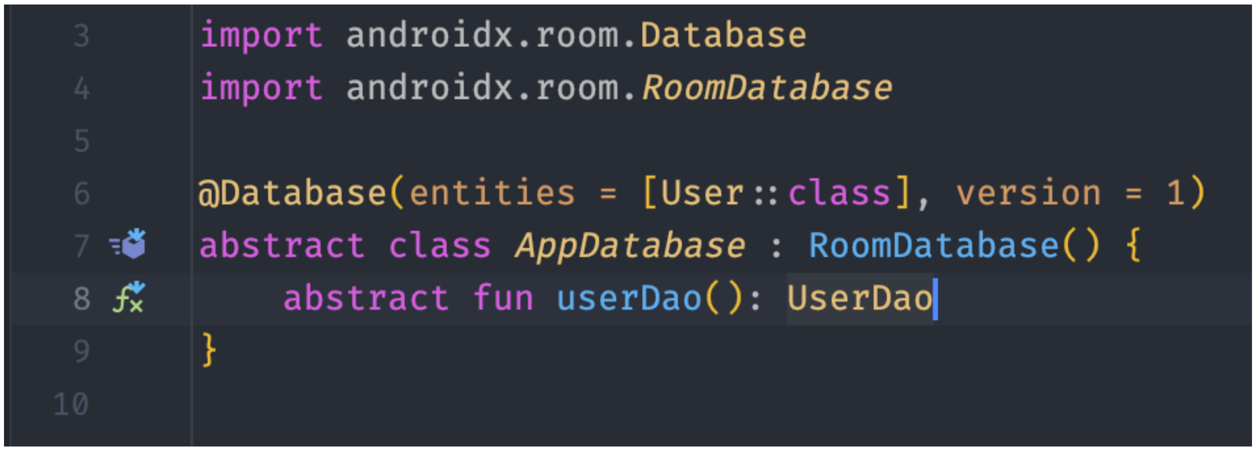
Our aim is to provide a ViewModel that would fetch the data from the local database, using Room. I use Flow to observe database changes and update the UI based on that. I add the Room dependencies in the build.gradle file, which includes kapt for annotation processing. Now I define the User entity by using the @Entity annotation and create a UserDao interface that defines methods for user insertion and fetching.



```kotlin
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Dao
interface UserDao {
    @Query("SELECT * FROM user_table")
    fun getAllUsers(): Flow<List<User>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(user: User)
}
```

Figure 17. User Dao Interface

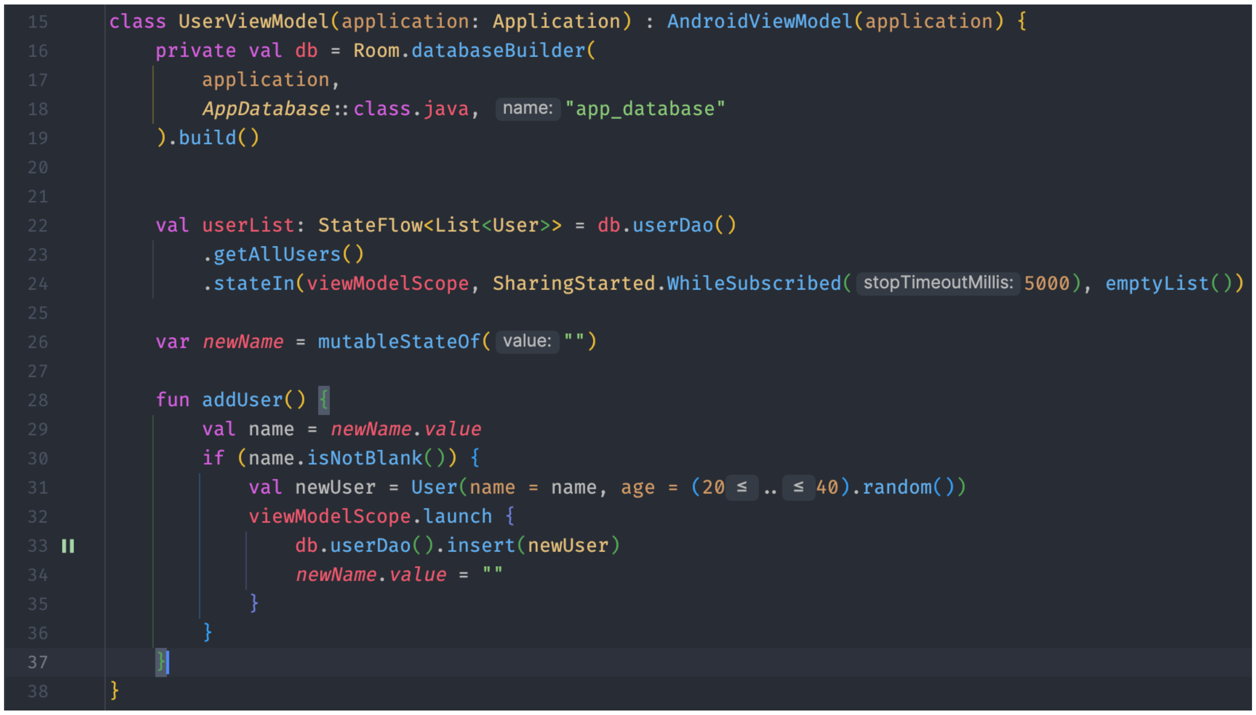
I will be implementing AppDatabase as an abstract class extending RoomDatabase and include the UserDao.



```kotlin
import androidx.room.Database
import androidx.room.RoomDatabase

@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}

```

Figure 18. App Database Snippet

UserViewModel Now, I initialize the database and get an instance of UserDao. Here I use the userDao.getAllUsers() which returns users as a Flow and convert to StateFlow with stateIn(). I updated the addUser() function to inset the new user into the database.



```kotlin
class UserViewModel(application: Application) : AndroidViewModel(application) {
    private val db = Room.databaseBuilder(
        application,
        AppDatabase::class.java, name: "app_database"
    ).build()


    val userList: StateFlow<List<User>> = db.userDao()
        .getAllUsers()
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed( stopTimeoutMillis: 5000), emptyList())

    var newName = mutableStateOf( value: "")

    fun addUser() {
        val name = newName.value
        if (name.isNotBlank()) {
            val newUser = User(name = name, age = (20 ≤ .. ≤ 40).random())
            viewModelScope.launch {
                db.userDao().insert(newUser)
                newName.value = ""
            }
        }
    }
}
```

Figure 19. Updated User View Model for Room Database

The application should save users' data in a Room database stored locally. Adding new users should update the database automatically and reflect changes in the UI.
I integrate Room to persist data locally. The UserDao provides methods to interact with the database. In UserViewModel, I retrieve users from the database as a Flow and convert it to a StateFlow for Compose compatibility. The addUser() function inserts new users into the database asynchronously using viewModelScope.launch.

Results

Each exercise was able to correctly apply modern Android development using Jetpack Compose. I eliminated the use of XML layouts, Fragments, and RecyclerViews and replaced them with composable functions that improved readability and maintainability.

Exercise 1 consisted of a simple composable that showed a message and logged lifecycle events to demonstrate the basics of the Composable lifecycle in Jetpack Compose.

Exercise 2 consisted of creating two composable screens and communicating between them using a common ViewModel, showcasing the state management of composables.

Exercise 3 showed the Navigation between those composable screens through Navigation Component for Compose, which showcased how easily screen transitions can be performed without using Fragments.

Exercises 4 and 5: I created a list using LazyColumn, then handled clicks on items with snackbar. This is where RecyclerView could have been used, but it also showed how Compose does lists seamlessly.

Exercise 6 explained the LazyColumn efficiency and how anyone wouldn't need the pattern ViewHolder. I really insisted on the internal Compose optimization in that respect.

Exercises 7 to 9 introduced State management with ViewModel and StateFlow, handling user input, persistence with Room. I showed the UI updates reactively with synchronous data across views and database.

One of the challenges was managing the scope of ViewModel instances across different Composable destinations. I resolved this by appropriately scoping the ViewModel instances using viewModel() in the navigation context.

Another challenge is how to integrate Room using coroutines and flows. Ensuring that the suspend functions were invoked from within a coroutine required judicious use of viewModelScope.launch.

Conclusion

This set of exercises shows the great benefits of using Jetpack Compose in modern Android development. I followed a declarative approach to UI, which has simplified our codebase, reduced boilerplate, and increased readability/maintainability of an application.

Composable functions of Jetpack Compose, if combined with effective state management with ViewModel and reactive streams like StateFlow, allow developers to build dynamic and responsive UIs without hustle. Also, navigation is integrated smoothly, and Fragments can be removed.

Mastery of these modern practices will, therefore, be crucial in the development of robust and scalable Android applications. Further use of Room for data persistence shows how Compose interacts with existing architecture components to give a coherent development experience.

This experience then highlights the fact that embracing Jetpack Compose is the future of developing Android UIs, as it aligns with modern paradigms of development and ultimately enhances overall productivity.

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

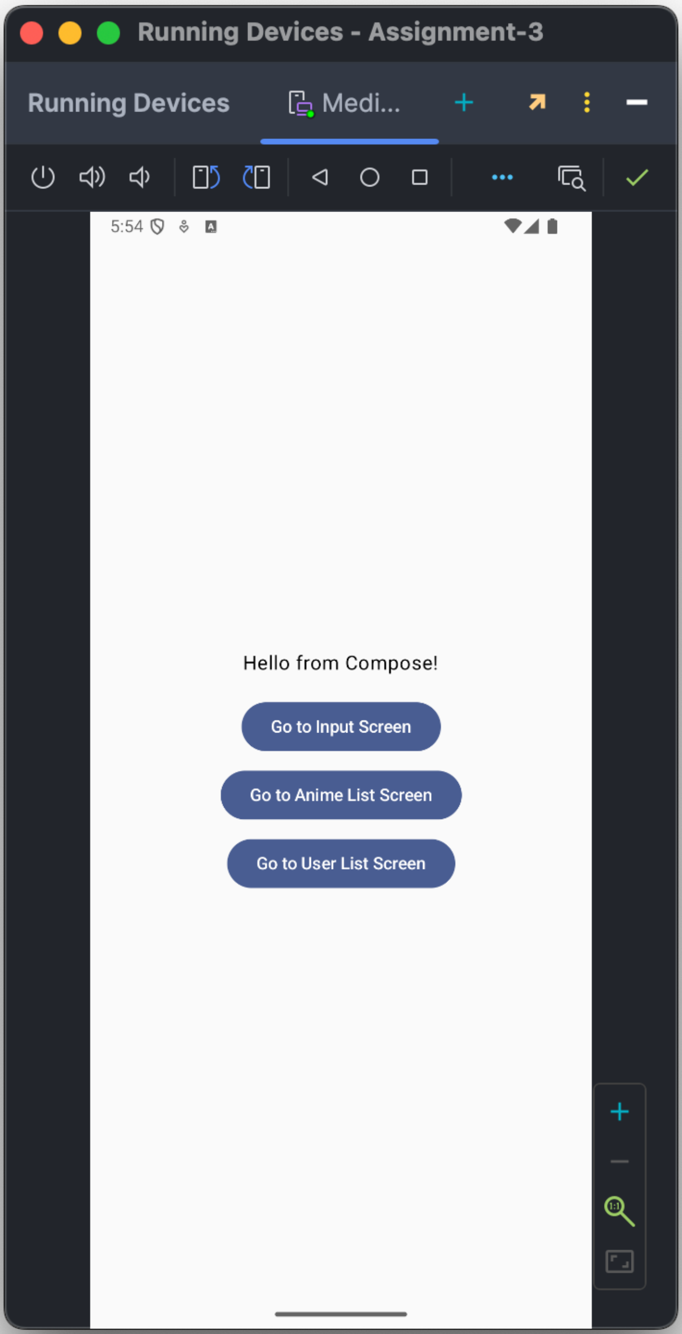


Figure 20. Main Screen

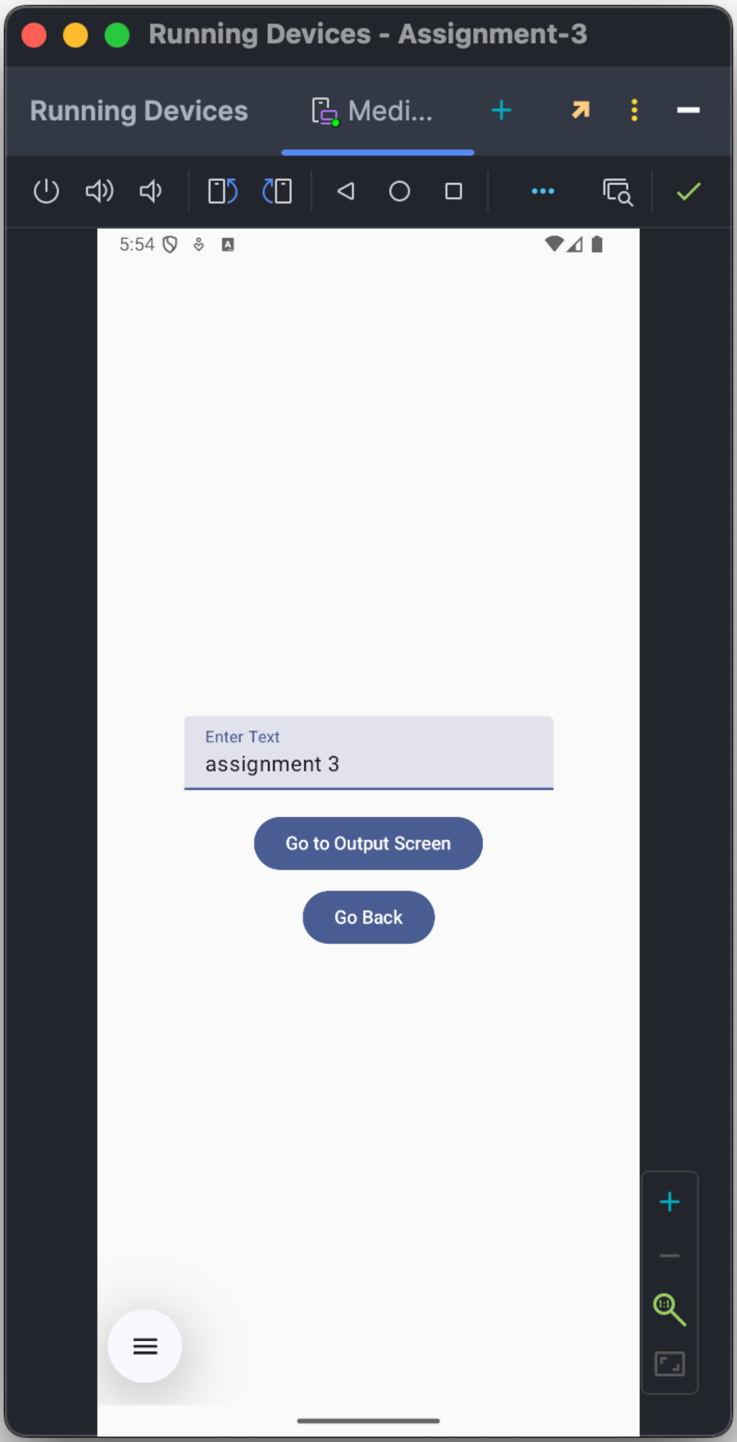
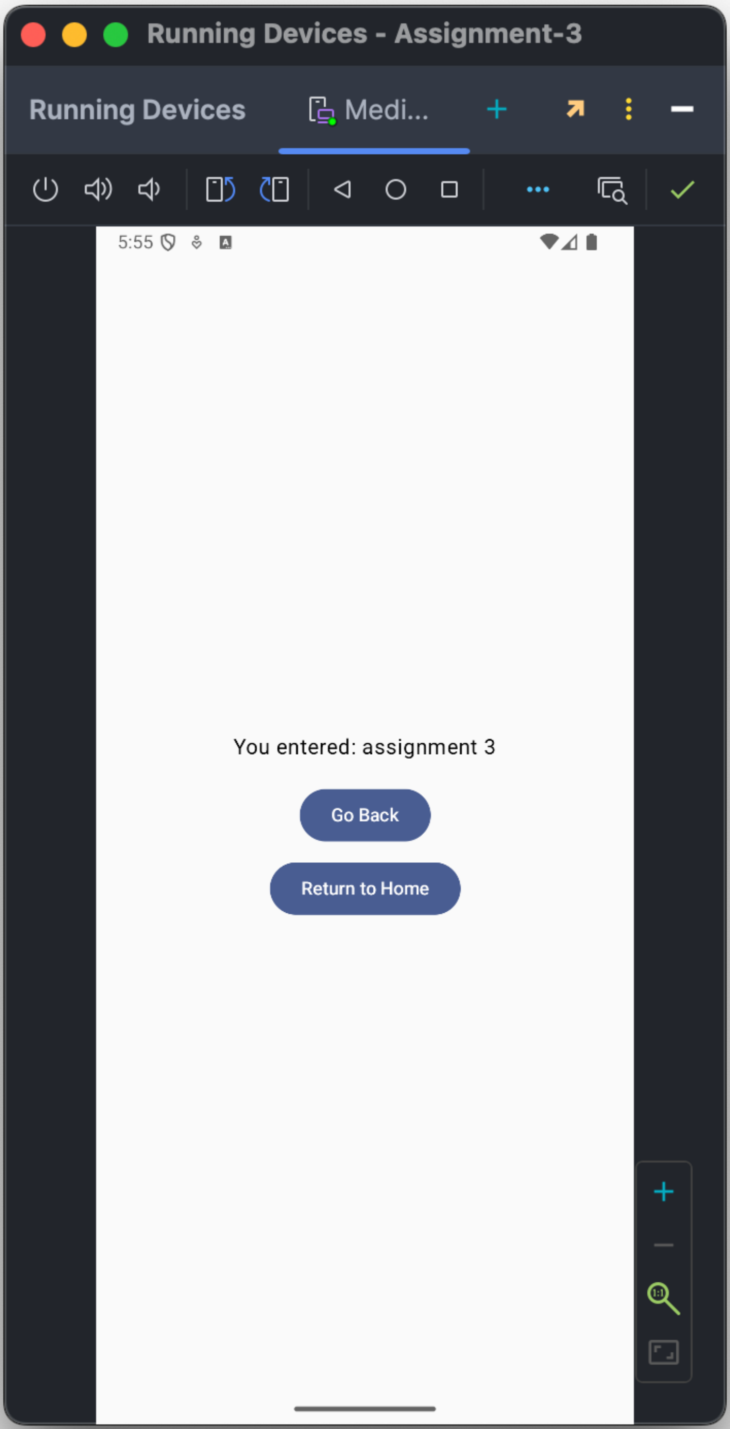
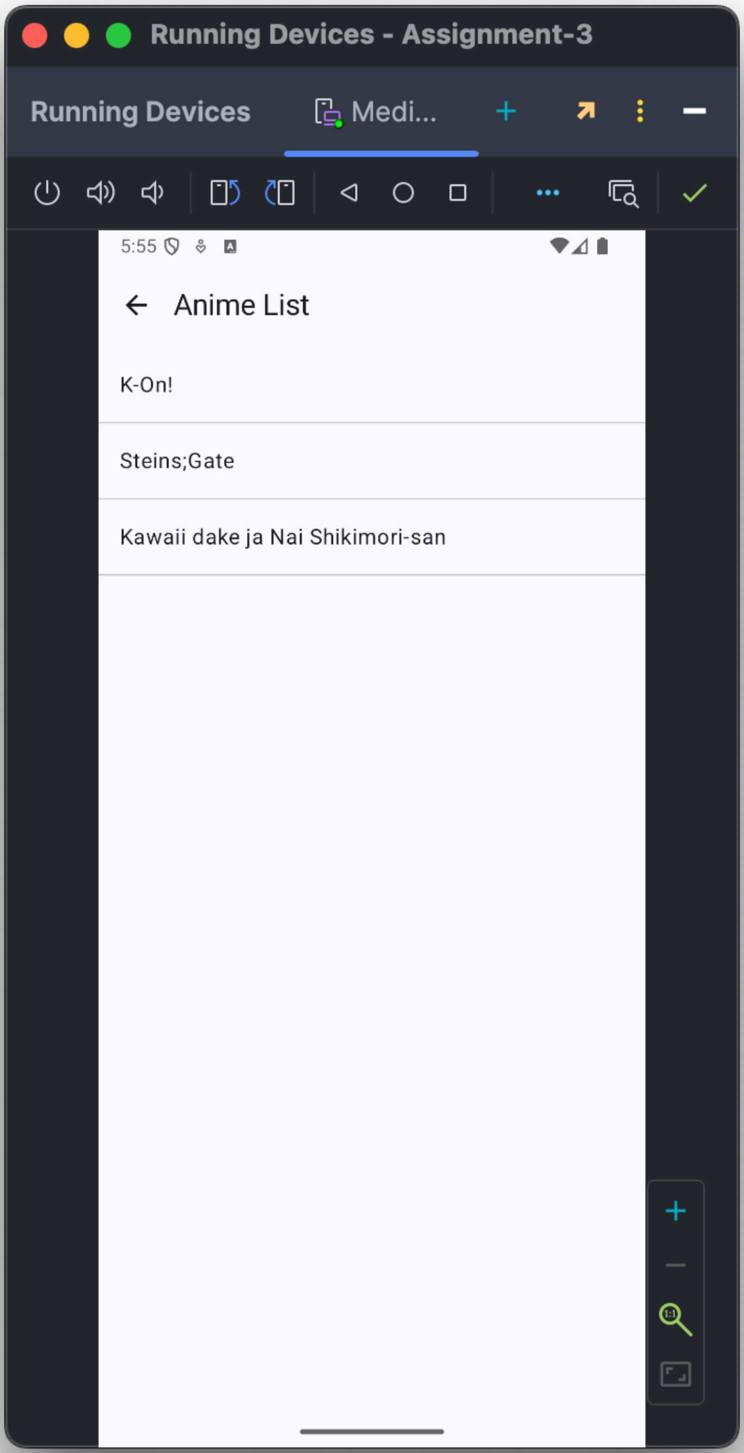


Figure 21. Input Screen



You entered: assignment 3

Figure 22. Output Screen



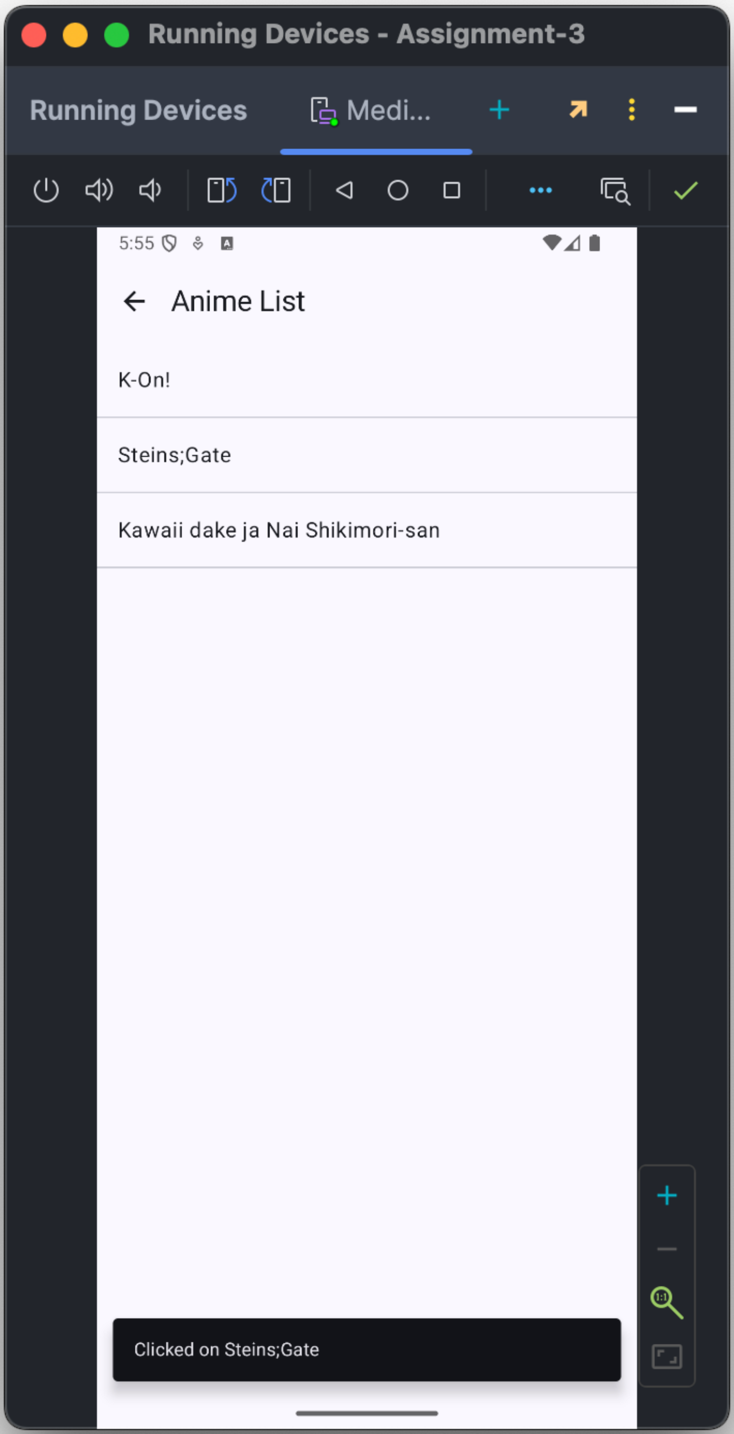
Anime List

K-On!

Steins;Gate

Kawaii dake ja Nai Shikimori-san

Figure 23. Anime List Screen



Anime List

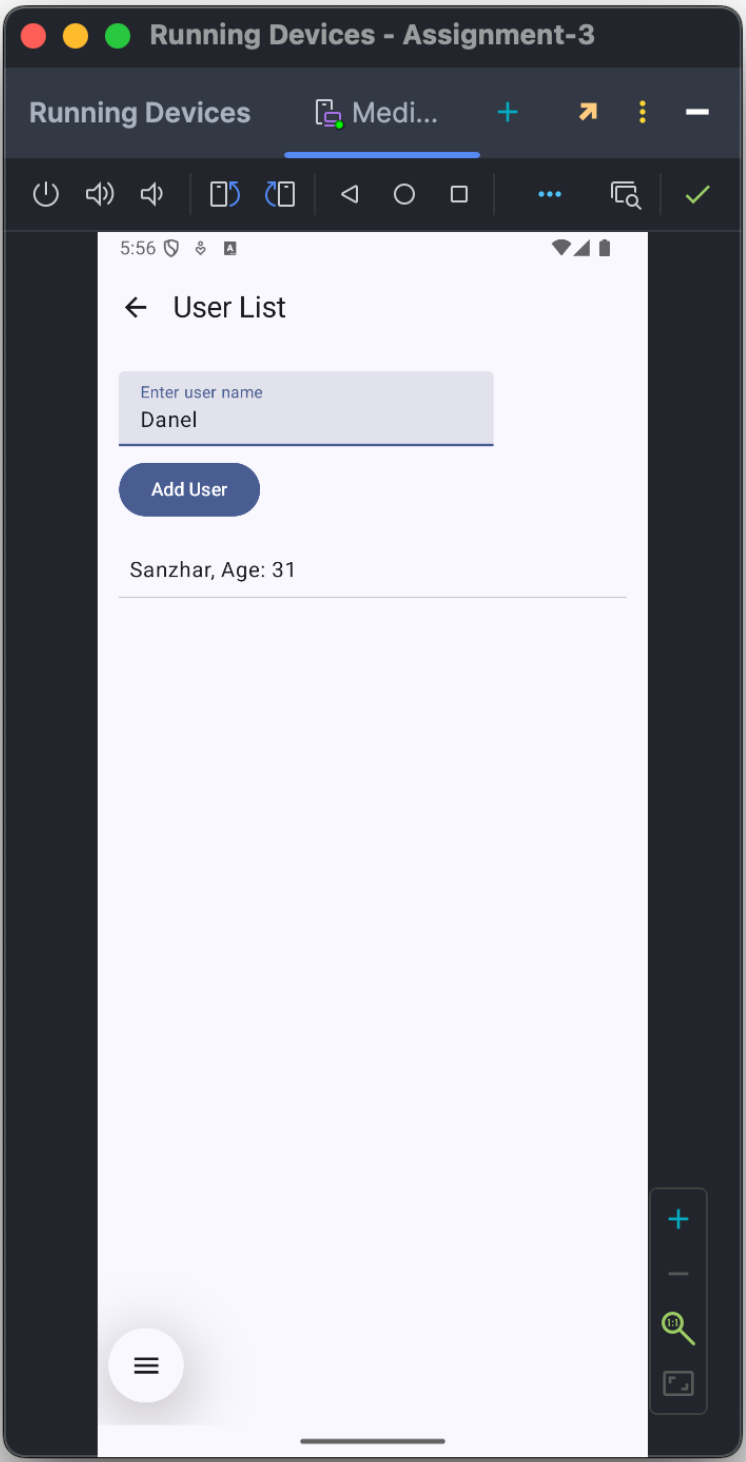
K-On!

Steins;Gate

Kawaii dake ja Nai Shikimori-san

Clicked on Steins;Gate

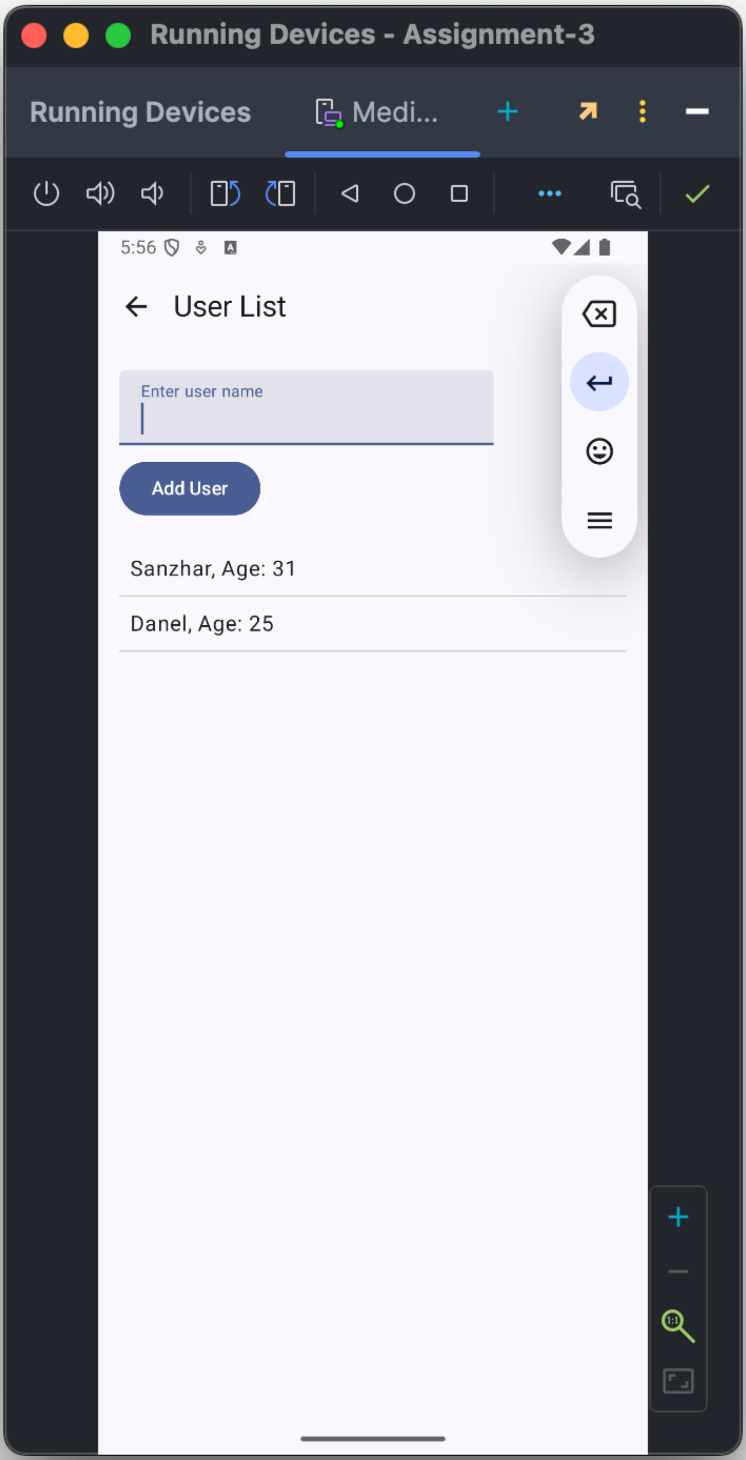
Figure 24. On Click Event in Anime List



Enter user name
Danel

Sanzhar, Age: 31

Figure 25. User List with Input Handling



Sanzhar, Age: 31

Danel, Age: 25

Figure 26. Updated User List with new User