KAZAKH-BRITISH TECHNICAL UNIVERSITY

Prepared by: Serzhan Yerasil

Assignment 3

Mobile Programing

09.11.2024

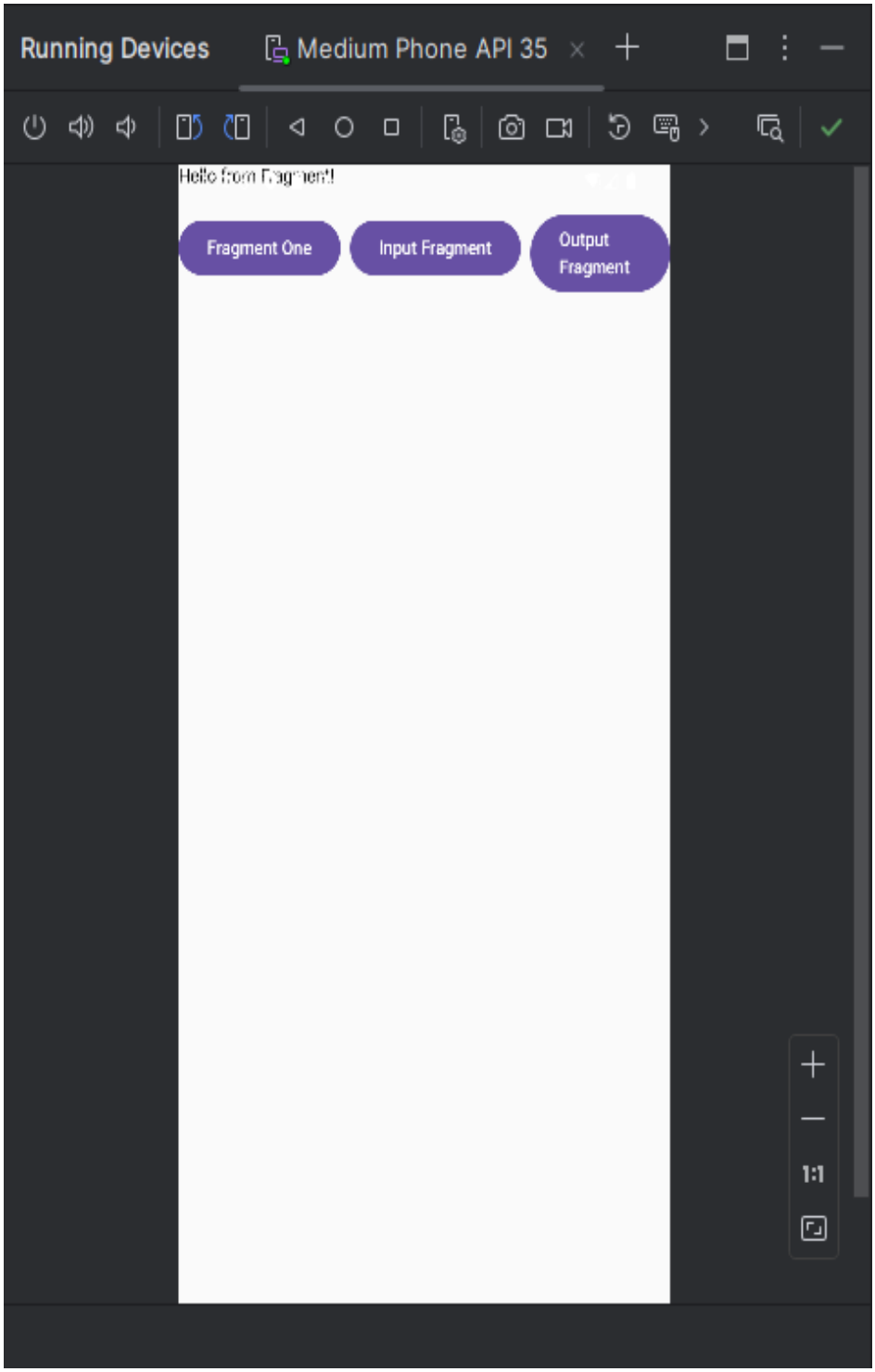
Table of Contents

Introduction

In this project, we will implement the basic concepts of state management in Android using Fragments, RecyclerView, ViewModel. These exercises help us understand how we can store and manage data in our app so that it automatically updates the UI when it changes. We will create an app that allows adding users to a list using user input, and see how ViewModel and LiveData make it easy to work with and display data.

Exercise 1, 2, 3

The first fragment simply prints the text "Hello from Fragment!" and records its lifecycle events. This is necessary to understand when the fragment is created. The second and third fragments work together. In the second fragment, the user enters text into an input field. This text is immediately passed to the third fragment, where it is displayed. This allows us to see how data can be passed between fragments. The main screen will have buttons that allow you to switch between fragments. For example, you can press a button to show either the first fragment with the text "Hello from Fragment!", or the second fragment for entering text, or the third fragment that displays the text you entered.

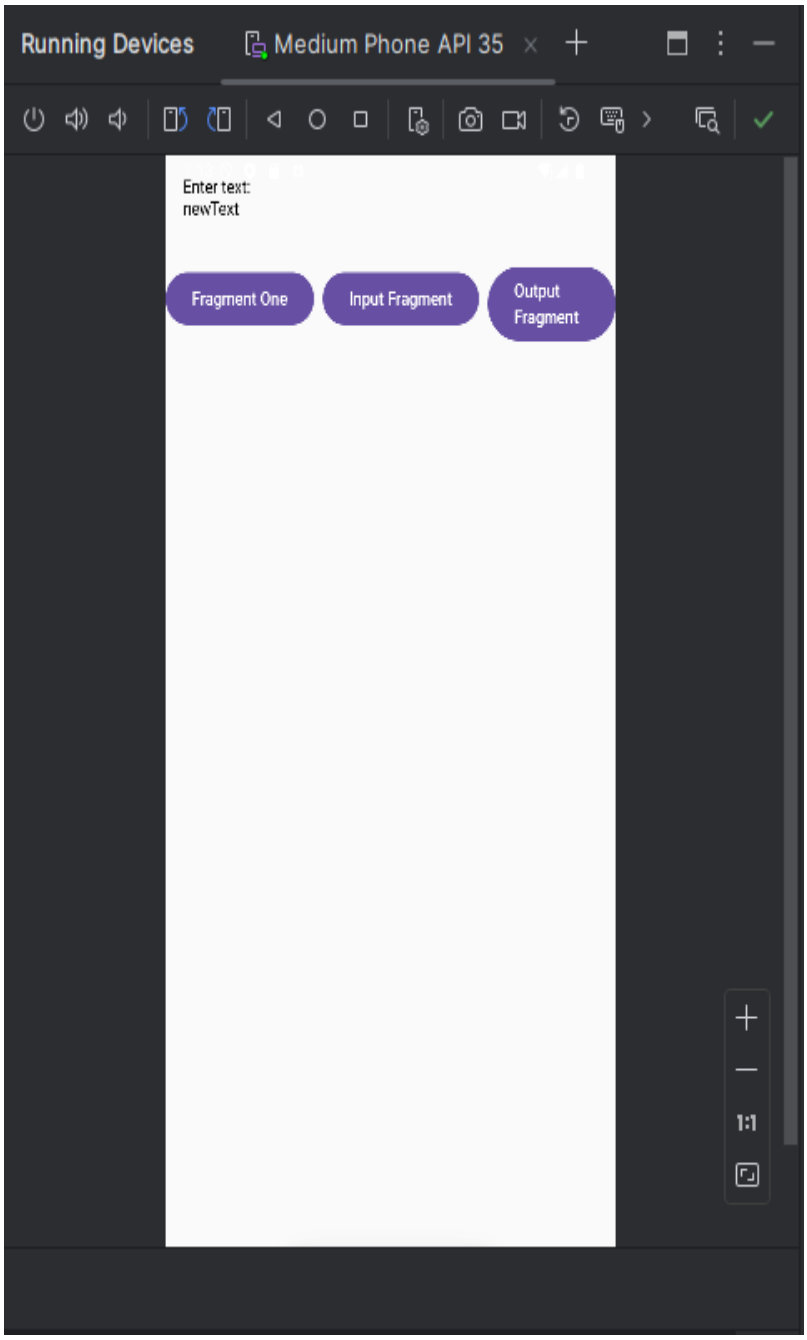


Pic1. First Fragment

```
@Composable
fun FragmentOne() {
    LogLifecycle( tag: "com.example.assignment3_1.FragmentOne") // Logs the lifecycle events
    Text(text = "Hello from Fragment!")
}


@Composable
fun LogLifecycle(tag: String) {
    DisposableEffect(Unit) {
        Log.d(tag, msg: "onCreateView")
        onDispose {
            Log.d(tag, msg: "onDestroyView")
        }
    }
}
```

FragmentOne: Shows the text "Hello from Fragment!" and logs its lifecycle events (creation and destruction).

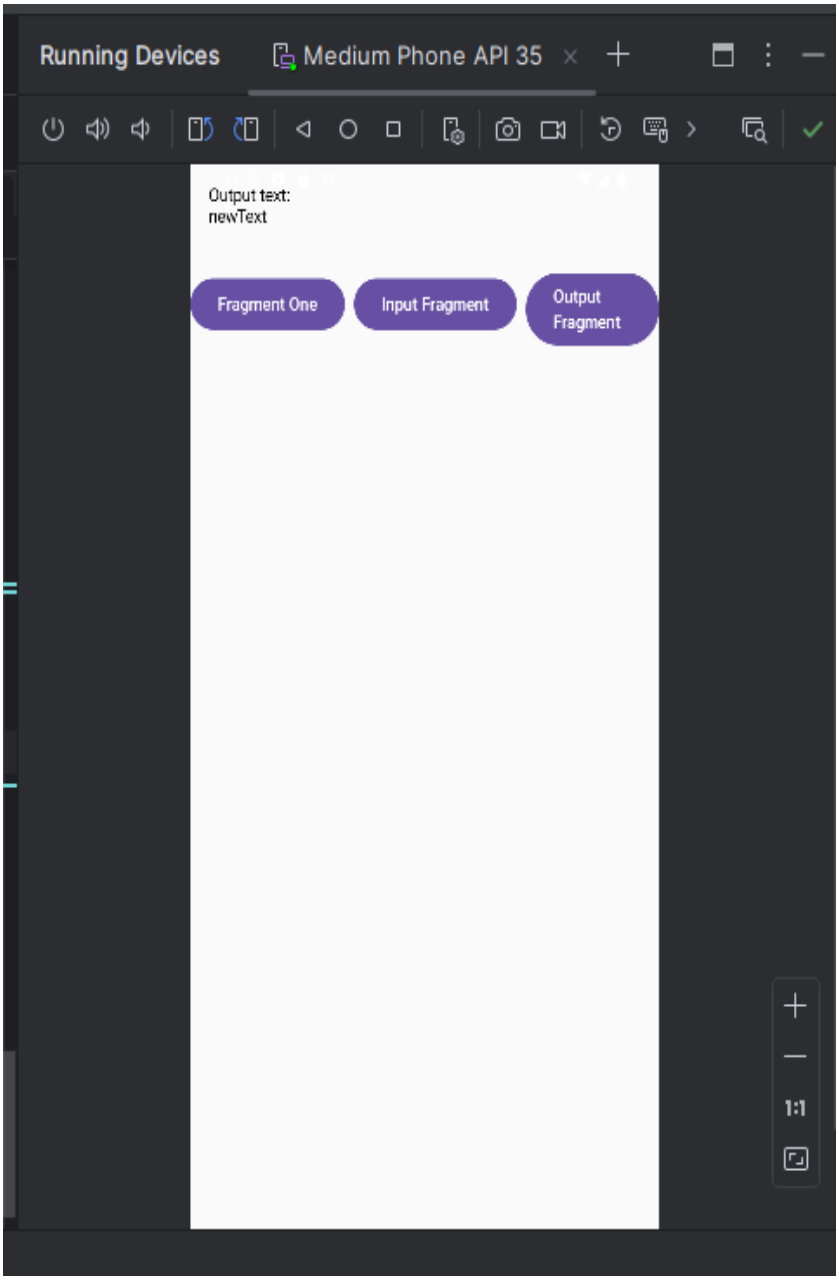
LogLifecycle: Logs the "onCreateView" message when FragmentOne is created, and the "onDestroyView" message when it is closed.



Pic2. Second Fragment

```
class SharedViewModel : ViewModel() {
    var inputText by mutableStateOf( value: "")
}

@Composable
fun FragmentInput(viewModel: SharedViewModel) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = "Enter text:")
        BasicTextField(
            value = viewModel.inputText,
            onValueChange = { viewModel.inputText = it },
            modifier = Modifier.fillMaxWidth()
        )
    }
}

@Composable
fun FragmentOutput(viewModel: SharedViewModel) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = "Output text:")
        Text(text = viewModel.inputText)
    }
}
```

Stores text inputText that can be updated and used in different "fragments". Shows a text input field BasicTextField.
The entered text is stored in viewModel.inputText.
Shows the text entered in FragmentInput, displaying the value of viewModel.inputText.



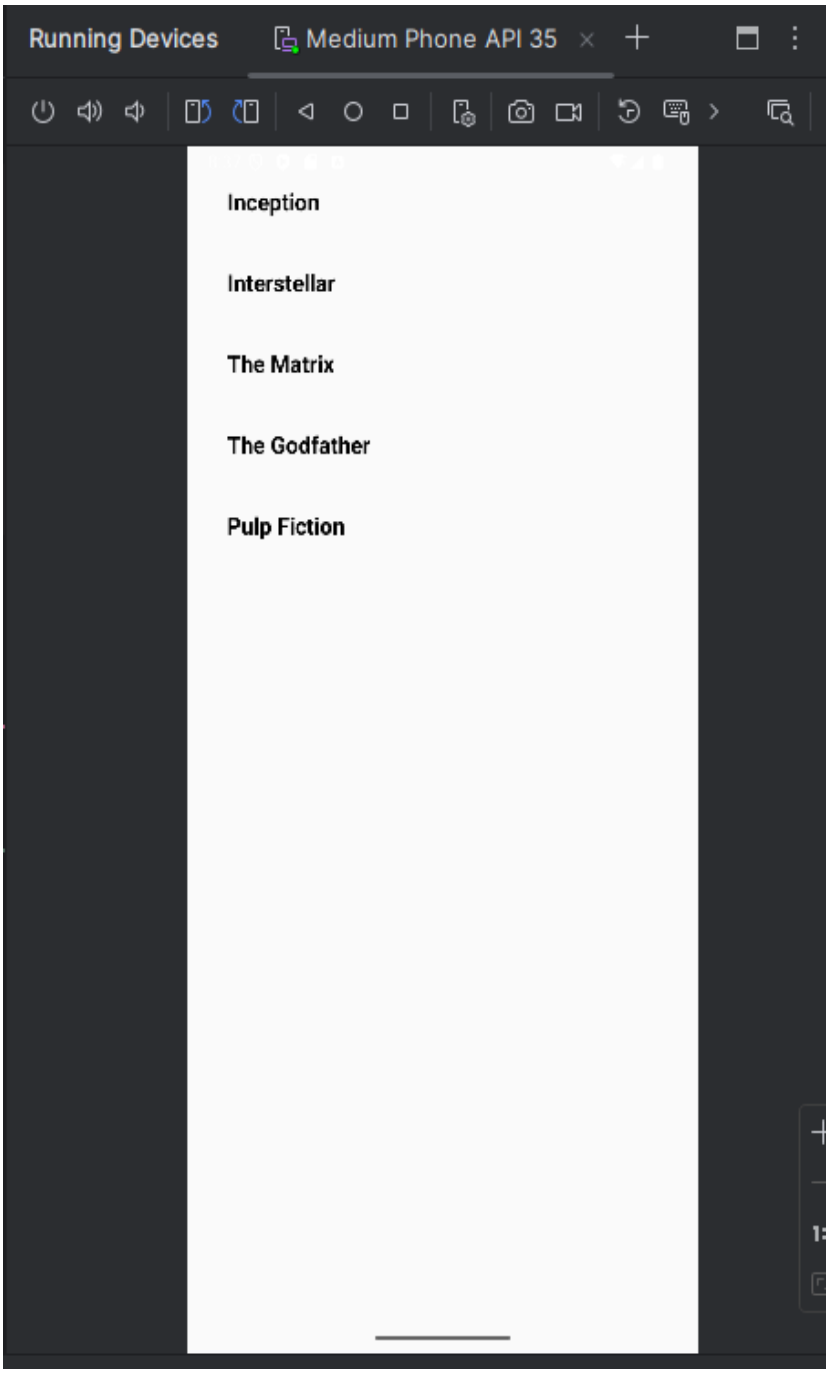
Pic3. Third Fragment

```
@Composable
fun FragmentLifecycleApp() {
    val viewModel: SharedViewModel = viewModel()
    var currentFragment by remember { mutableStateOf( value: 1) }

    Column {
        when (currentFragment) {
            1 -> FragmentOne()
            2 -> FragmentInput(viewModel)
            3 -> FragmentOutput(viewModel)
        }

        Spacer(modifier = Modifier.height(20.dp))

        Row(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
            Button(onClick = { currentFragment = 1 }) { Text( text: "Fragment One") }
            Button(onClick = { currentFragment = 2 }) { Text( text: "Input Fragment") }
            Button(onClick = { currentFragment = 3 }) { Text( text: "Output Fragment") }
        }
    }
}
```

Here, a viewModel is created, which is used to store data (such as text) that is common to different "fragments". This allows one "fragment" to write data and another to read it, since they have access to the same viewModel. currentFragment is a variable that stores the currently selected "fragment". The default value of 1 means that the first "fragment" is initially shown. Inside the Column are placed "fragments" and buttons. The Column arranges elements one under another.

Exercise 4, 5, 6

We create a list of movies using LazyColumn in Compose (similar to RecyclerView).
This list shows the title of each movie on the screen. We add the ability to click on each movie in the list.
When a movie is clicked, a toast is shown with its title. We use a separate MovieItem element to optimally render each list item.
This "pattern" helps make the list display more efficient and easier to maintain. The application displays a list of movies, each item can be clicked to display the movie title in a toast, and the code structure is organized so that each list item is processed efficiently.



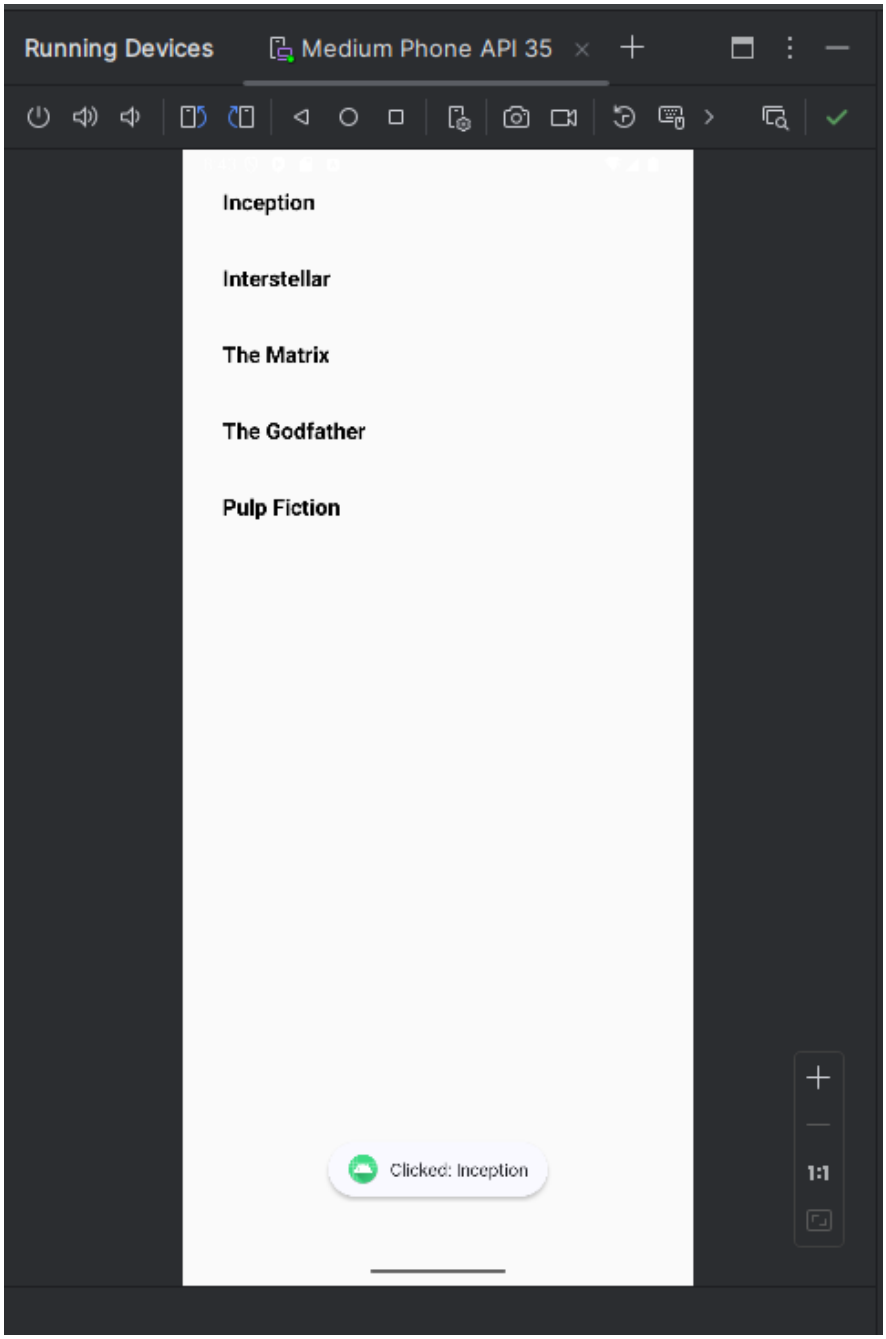
Pic4. Display all items

```
@Composable
fun MovieListScreen(viewModel: MovieViewModel = viewModel()) {
    val context = LocalContext.current

    // Displaying the list of movies in a LazyColumn (Compose's RecyclerView alternative)
    LazyColumn(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(viewModel.movies) { movie ->
            MovieItem(
                movie = movie,
                onClick = {
                    Toast.makeText(context, text: "Clicked: ${movie.name}", Toast.LENGTH_SHORT).show()
                }
            )
        }
    }
}
```

MovieListScreen The main function that displays the list of movies on the screen:

LazyColumn This is the equivalent of RecyclerView in Compose. It allows you to scroll and display a list of items.

items(viewModel.movies) Iterates through each movie in the movies list of the ViewModel and calls MovieItem for each movie.



Pic5. Handling a click on a list item

```
@Composable
fun MovieItem(movie: Movie, onClick: () -> Unit) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .clickable(onClick = onClick)
            .padding(16.dp),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text(
            text = movie.name,
            fontSize = 18.sp,
            fontWeight = FontWeight.Bold
        )
    }
}
```

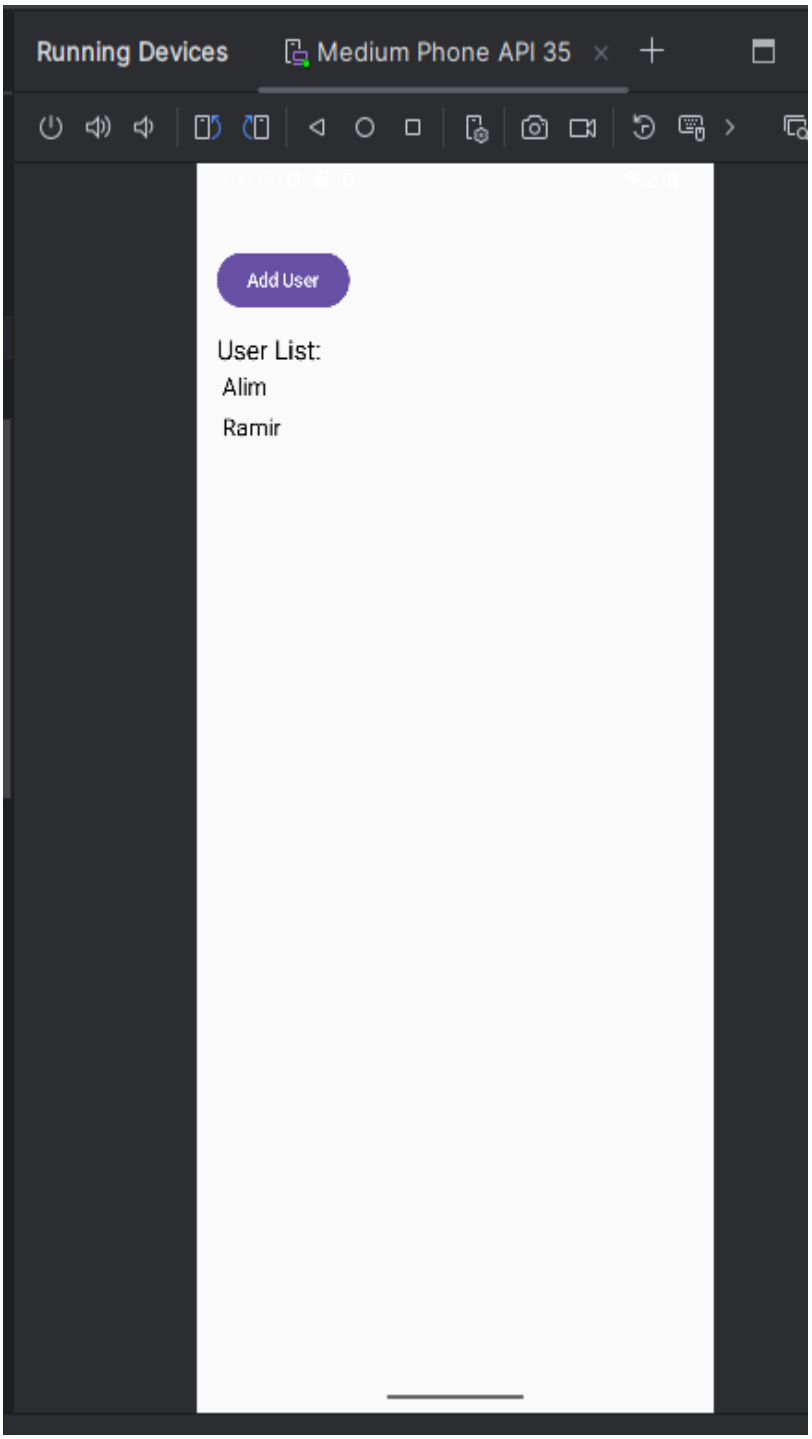
Added an onClick parameter that represents a function to execute when the item is clicked.

Modifier.clickable(onClick = onClick): Makes the item clickable and runs the onClick function when it is clicked.

When a MovieItem list item is clicked, it calls onClick, which is passed from MovieListScreen, and displays a Toast with the movie title.

Exercise 7, 8, 9

I create a ViewModel that stores a list of users.
LiveData allows to automatically update the UI when the list of users changes, so I don't have to update it manually. I add a text input field to the ViewModel. When the user types something, that value is stored in LiveData.
I also add a button to add a user. When the user clicks the button, the entered text is added as a new user to the list, and the input field is cleared. I don't use a real database (e.g. Room), but the ViewModel stores a list of users that is "saved" in memory.



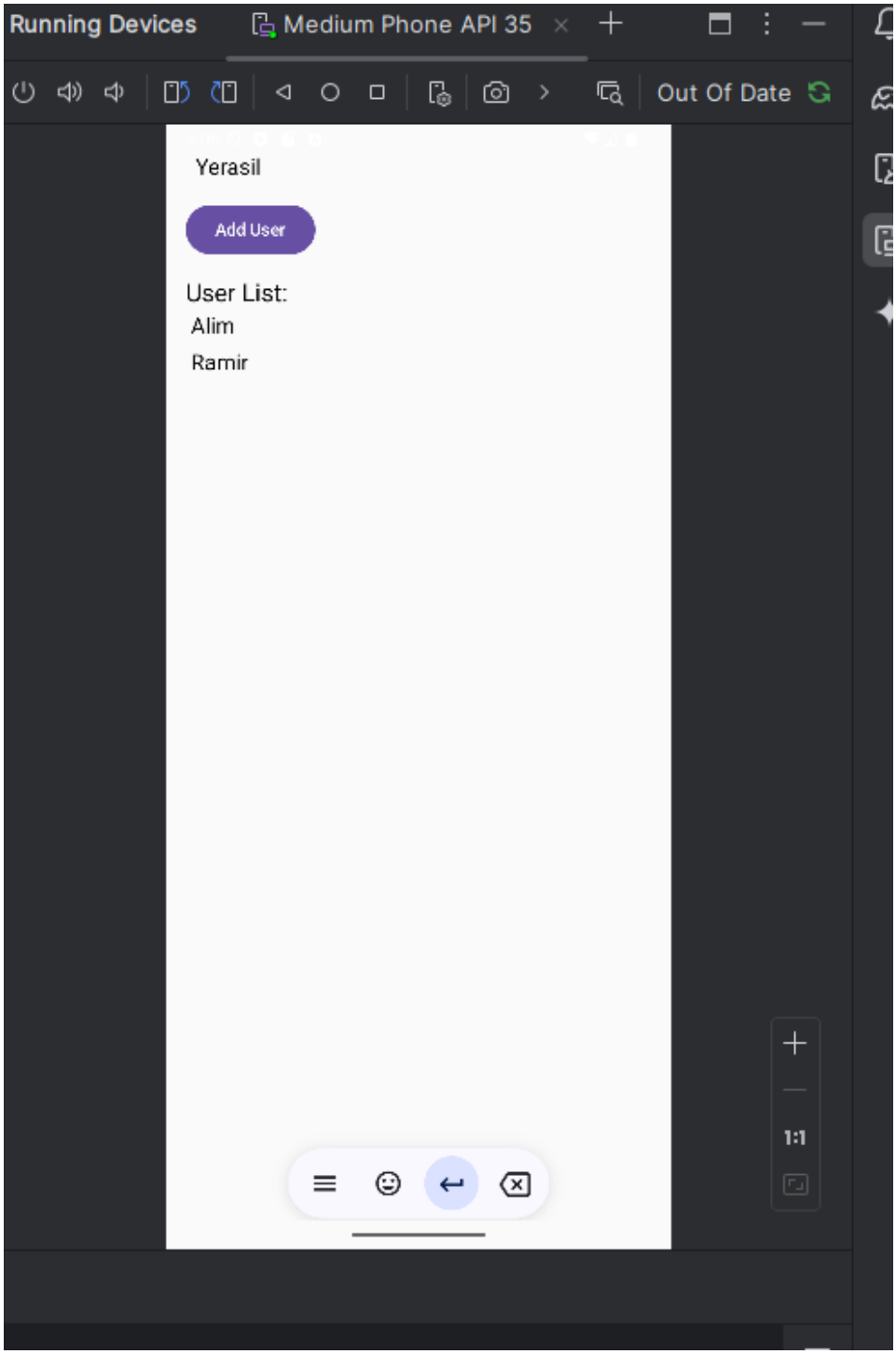
Pic6. Display all items

```
@Composable
fun UserApp(viewModel: UserViewModel = viewModel()) {
    Column(modifier = Modifier.padding(16.dp)) {
        // Exercise 8: Input field to add a new user
        val userInput by viewModel.userInput.observeAsState( initial: "")
        BasicTextField(
            value = userInput,
            onValueChange = viewModel::onUserInputChange,
            modifier = Modifier
                .fillMaxWidth()
                .padding(8.dp),
            textStyle = androidx.compose.ui.text.TextStyle(fontSize = 18.sp)
        )
        Spacer(modifier = Modifier.height(8.dp))
        Button(onClick = { viewModel.addUser() }) {
            Text(text = "Add User")
        }

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

        val users by viewModel.users.observeAsState(emptyList())
        Text(text = "User List:", fontSize = 20.sp)
        users.forEach { user ->
            Text(text = user.name, fontSize = 18.sp, modifier = Modifier.padding(4.dp))
        }
    }
}
```

val users by viewModel.users.observeAsState(emptyList()): Here we observe users in the ViewModel. observeAsState watches LiveData and automatically updates the list on the screen when the data changes. The ViewModel stores the list of users using LiveData.

LiveData automatically updates the UI when the data changes.

The UI displays the list of users and updates when users are added or changed in the ViewModel.



Pic7. Adding user

```
class UserViewModel : ViewModel() {

    private val _users = MutableLiveData<List<User>>(listOf(User( name: "Alim"), User( name: "Karim")))
    val users: LiveData<List<User>> = _users

    private val _userInput = MutableLiveData( value: "")
    val userInput: LiveData<String> = _userInput

    fun onUserInputChange(newInput: String) {
        _userInput.value = newInput
    }

    fun addUser() {
        val currentList = _users.value ?: listOf()
        val newUser = User( name: _userInput.value ?: "")
        if (newUser.name.isNotEmpty()) {
            _users.value = currentList + newUser
            _userInput.value = "" // Reset input after adding
        }
    }
}
```

onUserInputChange(newInput: String): This function updates the value of userInput when the user types text. For example, if the user types a name, this function will update _userInput with that value. addUser(): This function adds a new user to the list:

It takes the current value of userInput and creates a new user.

If the name is not empty, the new user is added to the _users list.

After adding, the input is cleared so that the next name can be entered.

Conclusion

During the exercises, we successfully learned how to manage data and interface state using Fragments, RecyclerView, ViewModel. We created an application that updates in real time in response to user actions, such as adding a new user. These exercises demonstrated how ViewModel and LiveData help make an application more organized, improving its performance and maintainability. These skills are essential for creating dynamic and responsive Android applications.