Assignment 3

Issayev Abzal

Date: 10.11.2024

Course: Mobile Programming

Github repo:
https://github.com/Karisbala/MobDev2024/tree/master/Midterm

Table of Contents

Introduction

Fragments are reusable parts of an application's user interface. Fragments can define and manage their own layout, have their own lifecycle, and handle their own input events. Fragments do not work alone. It must be placed in an activity or another fragment. A fragment's view hierarchy can be part of the host view hierarchy or attached to the host view hierarchy [1].

However, in this project, composable functions were used instead. Composable functions are a modern way of developing user interfaces in Android. It is a Kotlin function annotated with @Composable. This annotation indicates to the Compose compiler that the function is for creating user interfaces Unlike traditional XML layouts in Android, Composable functions allow you to create user interfaces using more intuitive and flexible Kotlin code [2].

RecyclerView allows you to efficiently display large datasets: the RecyclerView library dynamically creates items when they are needed. As the name implies, RecyclerView recycles individual items. When an item is scrolled off-screen, RecyclerView does not destroy its view. Instead, RecyclerView reuses the view for new items that scroll off the screen. RecyclerView improves the performance and responsiveness of the application and reduces power consumption [3].

However, the project instead utilizes LazyColumn, a composable feature in Jetpack Compose used to display a vertically scrolling list of items. This is a more efficient alternative to traditional columns, especially when working with large or dynamic datasets. Conventional columns display all child composites at once, which can lead to performance issues when displaying large lists, whereas LazyColumn optimizes rendering by composing and laying out only those items that are visible on the screen. LazyColumn optimizes rendering by composing and laying out only those elements that are visible on the screen [4].

The ViewModel class is a holder of business logic or state at the screen level; it exposes state to the user interface and encapsulates the associated business logic. Its main advantage is that the state is cached and retained when the configuration changes. This means that data does not need to be retrieved again after a configuration change, such as when the UI moves from one activity to another or rotates the screen [5].

LiveData is a holder class for observable data. Unlike regular observables, LiveData is lifecycle-aware. This means that it takes into account the lifecycles of other application components such as activities, fragments, and services. This ensures that LiveData updates only those application component observables that are in an active lifecycle state [6].

Components such as Fragments, RecyclerView, ViewModel, and LiveData are essential for Android development. Fragments help to create reusable UI components that work easily in different parts of the application. RecyclerView provides an efficient way to display large datasets with high performance; ViewModel is important for managing UI-related data as configuration changes occur, ensuring that data is preserved without interruption; LiveData allows the UI to automatically respond to data changes and react automatically, while honoring component lifecycles and preventing failures. Together, these tools make Android apps more responsive, modular and user-friendly. The project utilized Jetpack Compose components such as LazyColumn instead of RecyclerView and Composable functions instead of Fragments, resulting in a more flexible and modern UI design.

Exercise Descriptions

Exercise 1: Creating a Basic Fragment

Objective: Create a composable that displays a simple message ("Hello from Fragment!") and log lifecycle events.
Description of the implementation steps:

- Instead of creating a traditional Fragment, a BasicFragmentScreen composable was implemented to display the message.
- Lifecycle events (onCreate, onStart, onResume, etc.) were simulated using DisposableEffect [7] and LifecycleEventObserver [8] to mimic the Fragment lifecycle.
- The composable observed lifecycle changes of the LocalLifecycleOwner and logged these events.

Expected Outcome: A screen displaying "Hello from Fragment!" while lifecycle events are logged, replicating traditional Fragment lifecycle behavior in a composable.

Exercise 2: Fragment Communication

Objective: Implement communication between input and output using a shared ViewModel.
Description of the implementation steps:

- Instead of using traditional XML Fragments, two composable functions (InputComposable and OutputComposable) were created.
- A shared ViewModel (SharedViewModel) managed the communication between the input and output composables.
- The FragmentCommunicationScreen composable determined which composable to display based on the state (showOutput).
- When users entered data in InputComposable and submitted it, the shared ViewModel updated the state, and OutputComposable displayed the input text.

Expected Outcome: Users could enter text in InputComposable, and after submission, it was displayed in OutputComposable through the shared ViewModel.

Exercise 3: Fragment Transactions

Objective: Implement switching between two screens to simulate Fragment transactions.
Description of the implementation steps:

- Compose's navigation component (NavHost and NavController [9]) was used to replace traditional Fragment transactions.
- Two composables (FirstFragmentComposable and SecondFragmentComposable) were implemented, each representing different screens.
- Users navigate between the screens using buttons, simulating the behavior of adding, replacing, or removing Fragments.

Expected Outcome: Users could switch between two screens, emulating Fragment transactions through the navigation component in Compose.

Exercise 4: Building a RecyclerView

Objective: Create a list to display a collection of movies.
Description of the implementation steps:

- A MovieList composable was implemented using LazyColumn to display a list of movie titles.
- Each movie was represented by a MovieItem composable.

Expected Outcome: A scrollable list of movies, displayed efficiently using LazyColumn.

Exercise 5: Item Click Handling

Objective: Handle clicks on items in the movie list.
Description of the implementation steps:

- A clickable modifier was added to the MovieItem composable.
- A Toast message was displayed when a movie was clicked, showing the movie's name.

Expected Outcome: Users could click on a movie item and see a Toast message displaying the clicked movie's name.

Exercise 6: ViewHolder Pattern

Objective: Apply the ViewHolder pattern to optimize the movie list implementation.
Description of the implementation steps:
In Jetpack Compose, the concept of a ViewHolder pattern is not directly applicable, as in traditional RecyclerView. Instead, Composables are designed to be efficient by reusing state based on recomposition in the same way that the ViewHolder pattern [10] is used to optimize view rendering and binding.
Existing implementations already use Compose optimizations such as LazyColumn and items() that perform recycling and efficient rendering behind the scenes, just like the ViewHolder in the classic RecyclerView. And to implement ViewHolder-like behavior:

- The MovieItem composable served as a reusable layout for each item in the list, similar to the ViewHolder pattern.
- The approach ensured that layout definitions were isolated, improving reusability and efficiency.

Expected Outcome: The MovieItem composable provided a reusable and optimized representation for each movie, similar to the benefits provided by a traditional ViewHolder.

Exercise 7: Implementing ViewModel

Objective: Create a ViewModel to store and manage a list of users.
Description of the implementation steps:

- A UserViewModel was implemented to manage a list of users using LiveData.
- The ViewModel also managed two MutableLiveData properties for new user input (newUserName and newUserEmail).
- observeAsState() was used to automatically update the UI when the data changed [11].
- A function addUser() was implemented to add new users to the list.

Expected Outcome: The user list was managed by UserViewModel, and the UI reflected changes automatically.

Exercise 8: MutableLiveData for Input Handling

Objective: Handle user input via MutableLiveData in the ViewModel.
Description of the implementation steps:

- MutableLiveData properties were added in UserViewModel to manage new user input for name and email.
- UserInputFields composable was created to accept user input, which updated the MutableLiveData properties in real time.
- A button was added to submit user input, calling addUser() to add the new user to the list.

Expected Outcome: Users could enter their name and email, which was then automatically added to the list upon submission.

Exercise 9: Data Persistence

Objective: Integrate Room Database to persist user data.
Description of the implementation steps:

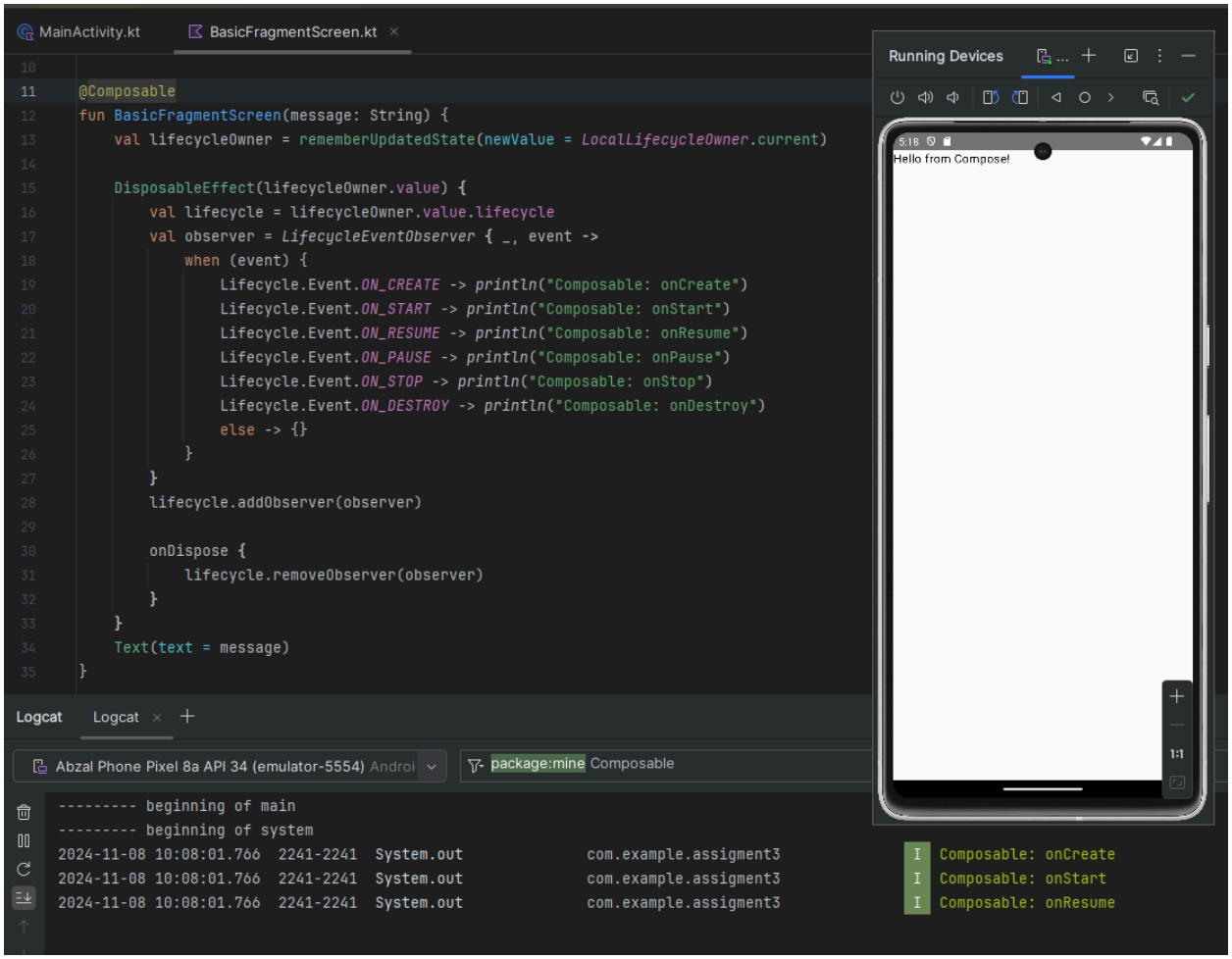
- Room components (User, UserDao, and UserDatabase) were added to store user information [12].
- The UserViewModel interacted with UserDao to insert and retrieve user data, using LiveData to observe changes.
- The addUser() function was modified to insert the user into the Room database in a background coroutine.
- Dependencies for Room and kapt were added to enable annotation processing.

Expected Outcome: User data was persisted locally, allowing users to remain in the list even after the app was restarted. The UI reflected the Room database data for long-term reliability.

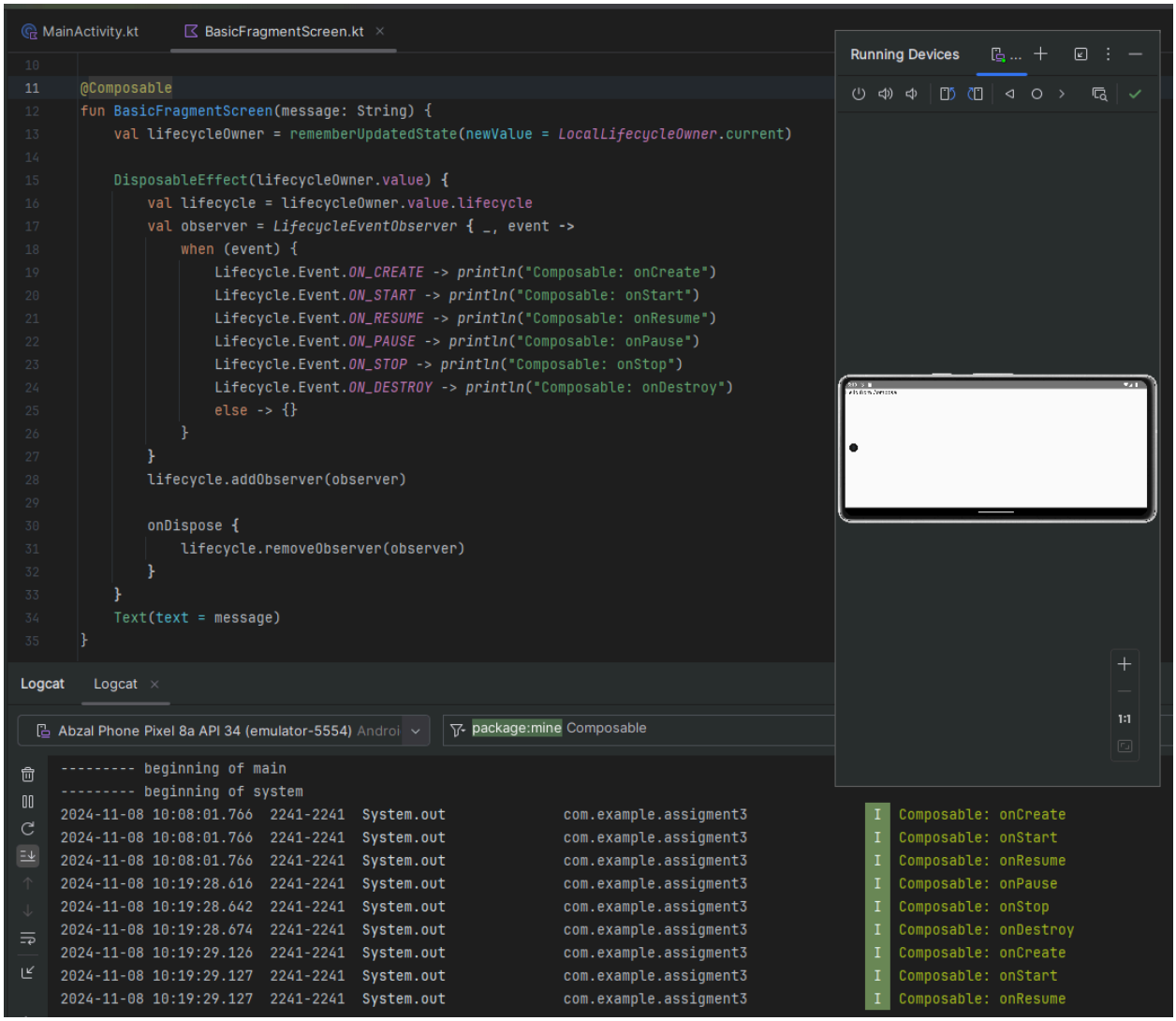
Results and Code Snippets

Exercise 1: Creating a Basic Fragment

The implementation successfully displayed the message and logged lifecycle events, providing a composable that closely mimics Fragment lifecycle behavior.



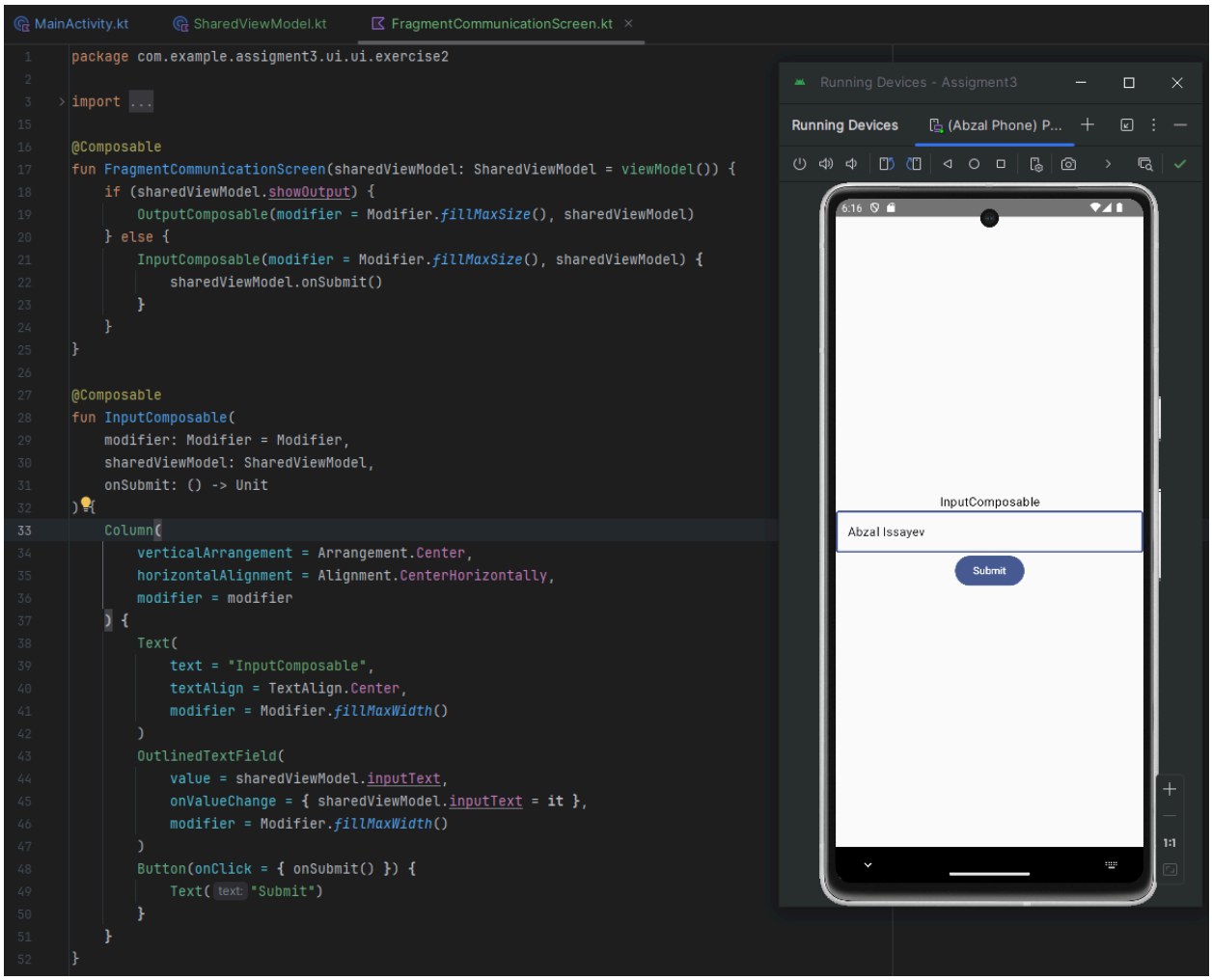
Screenshot 1. Code and app screenshot that represents the implementation logged of Lifecycle events when the app launches.



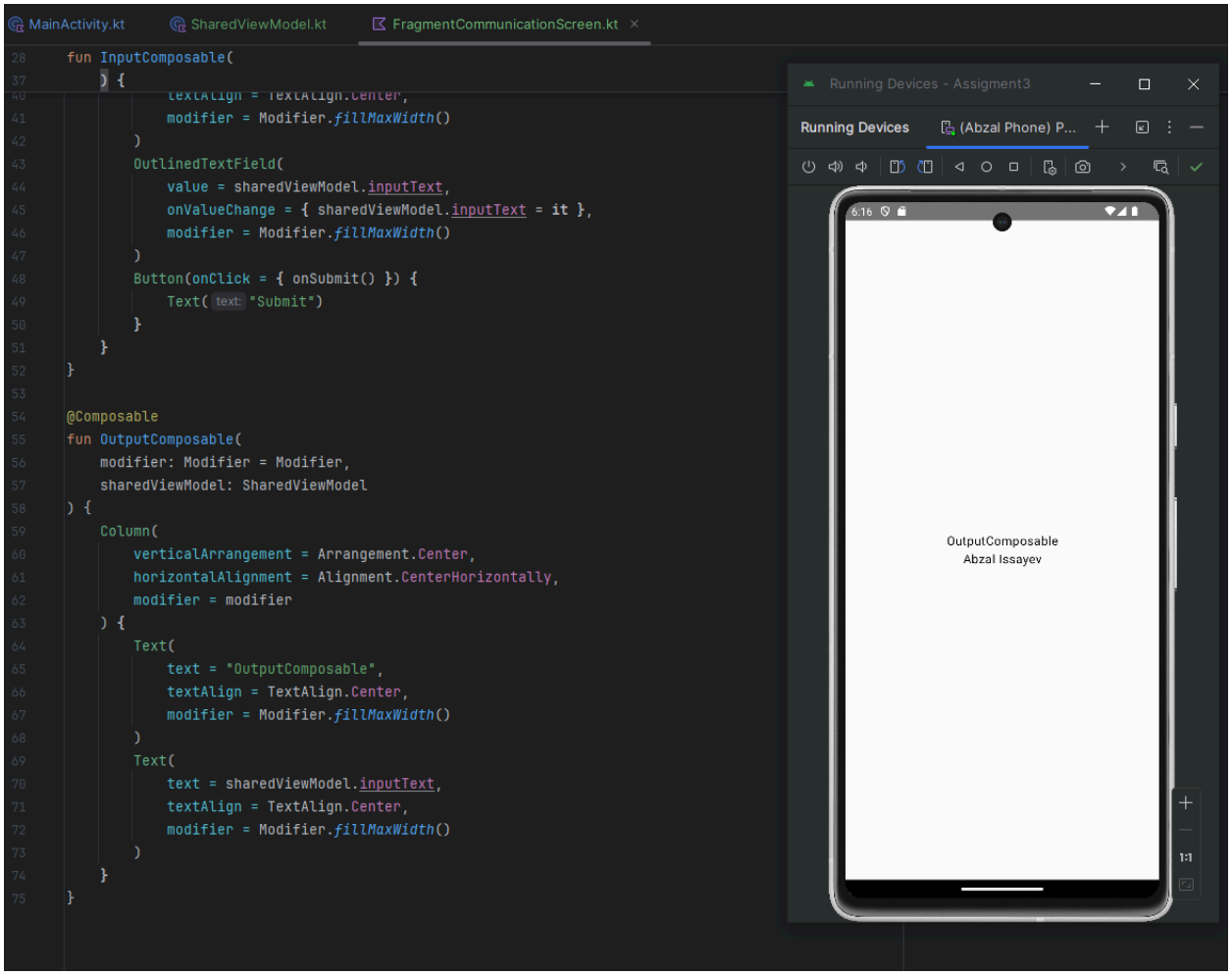
Screenshot 2. Code and app screenshot that represents the implementation logged of Lifecycle events when the app has configuration changes (screen rotation).

Exercise 2: Fragment Communication

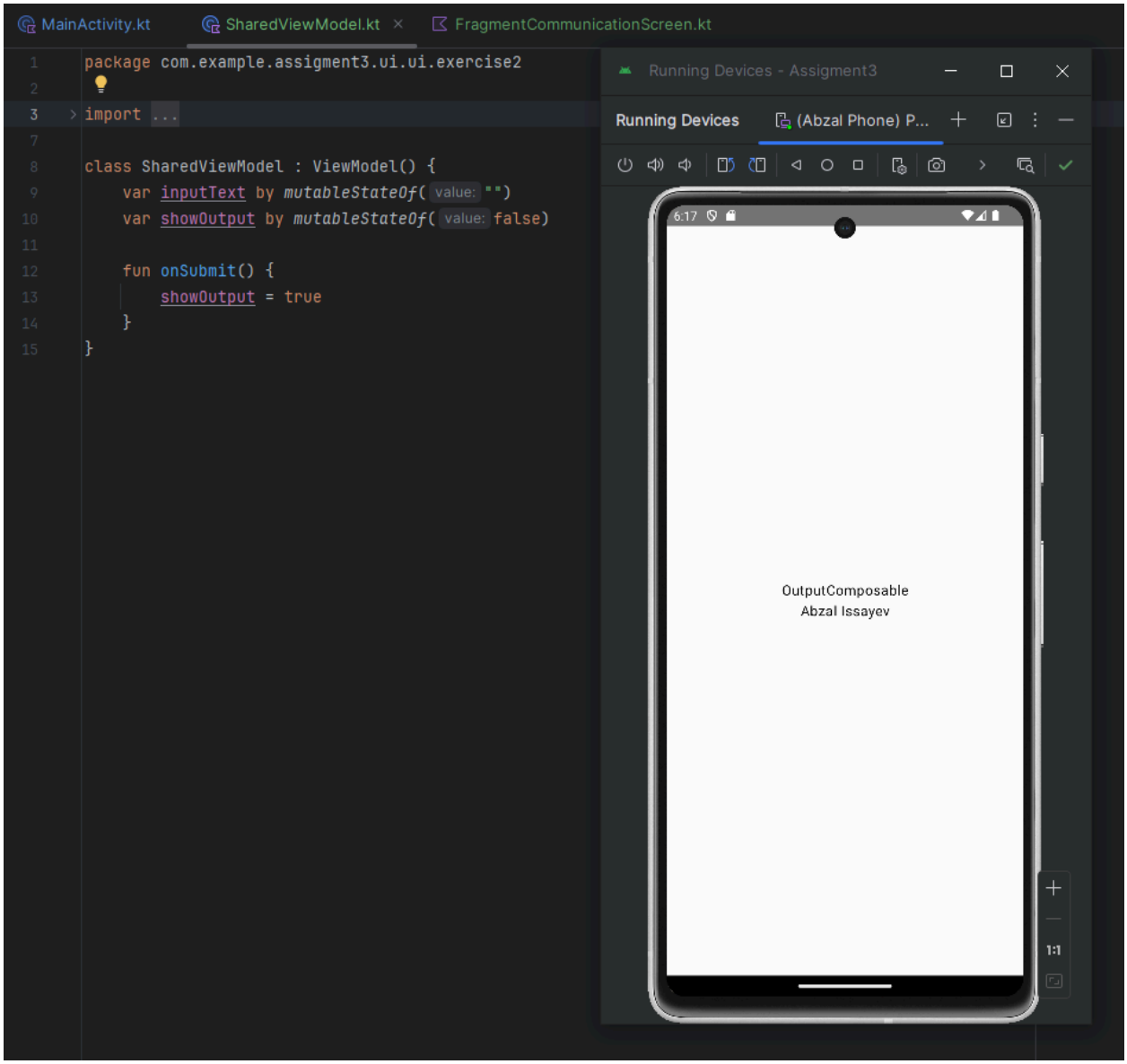
The communication between input and output composables worked as intended, with the input state being managed by the shared ViewModel.



Screenshot 3. Code and app screenshot that represents the InputComposable and Swapping Composables functionality.



Screenshot 4. Code and app screenshot that represents the OutputComposable.

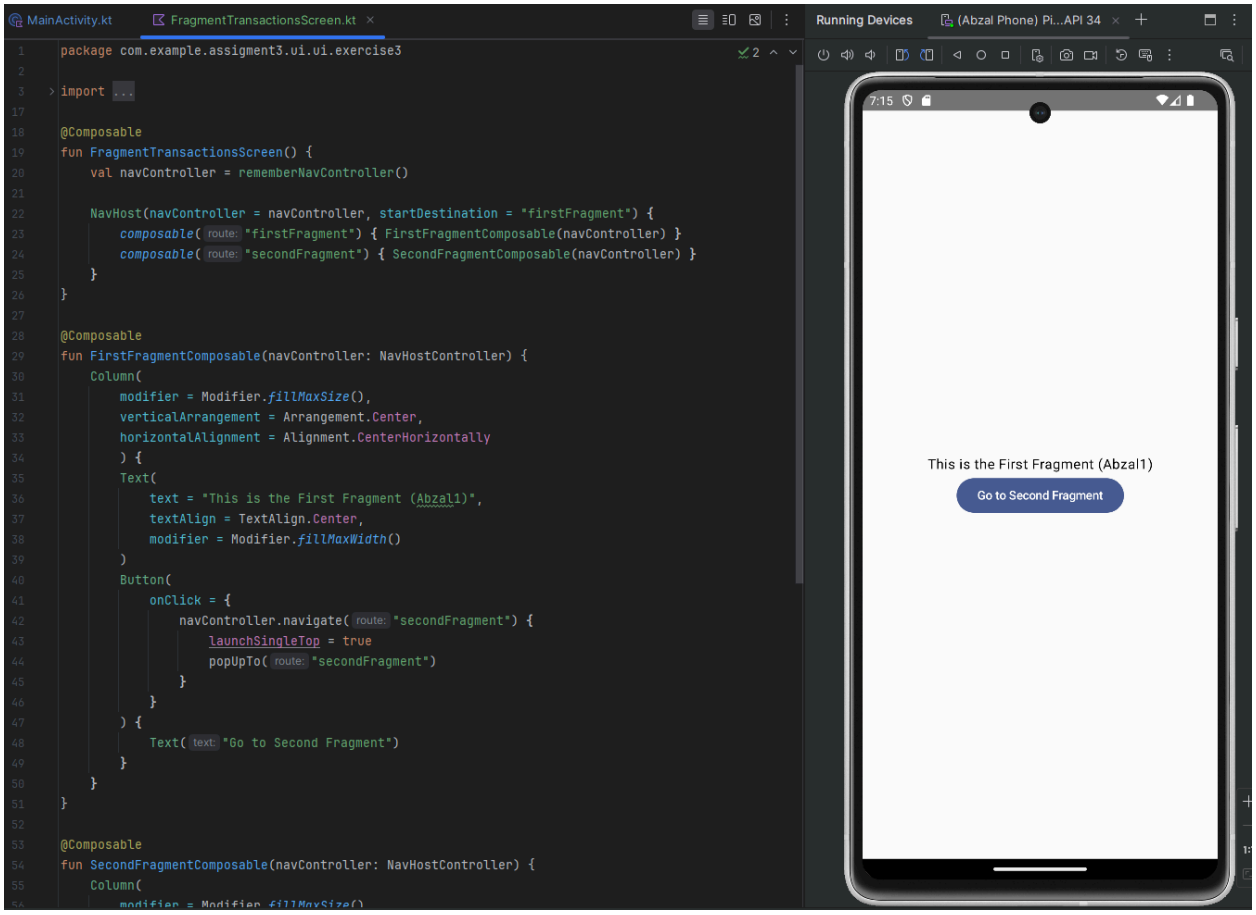


Screenshot 5. Code and app screenshot that represents the SharedViewModel implementation.

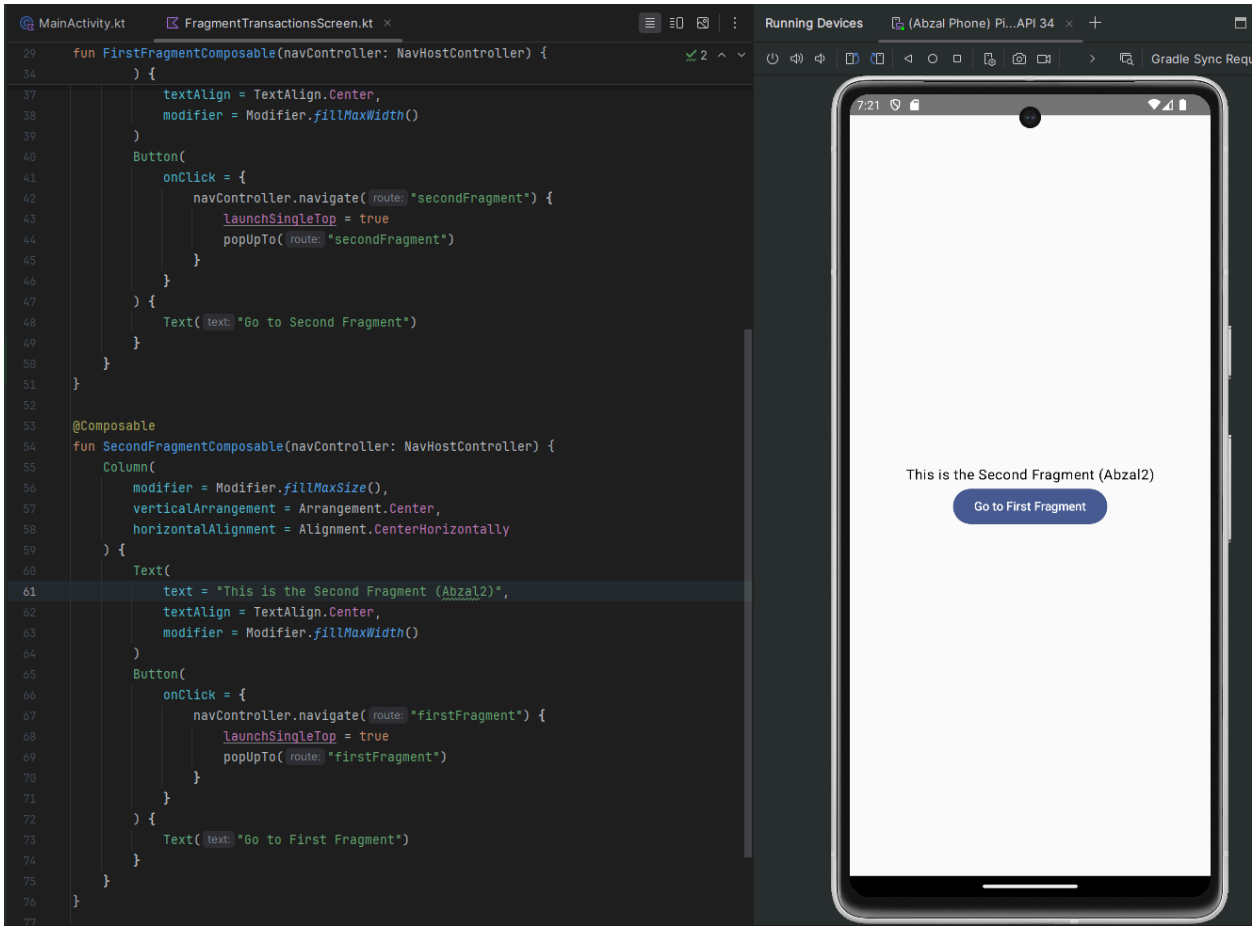
Exercise 3: Fragment Transactions

The navigation between screens worked smoothly, providing a user experience similar to traditional Fragment transactions.

Main problem was that it was forced to implement navigation without traditional Fragment transactions. As a solution NavHost and NavController from Jetpack Compose's navigation library was used to manage navigation efficiently.



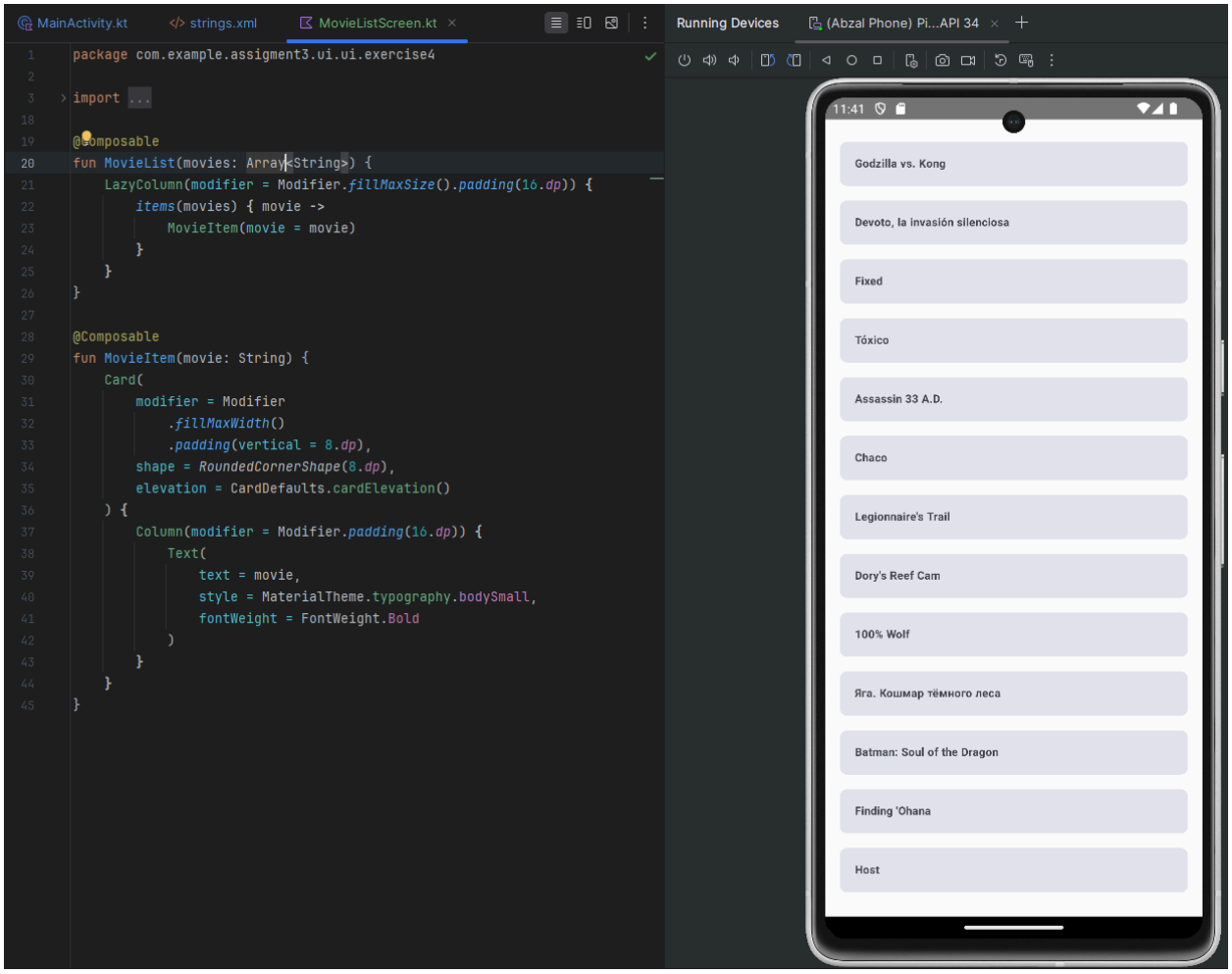
Screenshot 6. Code and app screenshot that represents the Navigation between composables (fragments) and first composable implementation.



Screenshot 7. Code and app screenshot that represents the second composable implementation.

Exercise 4: Building a RecyclerView

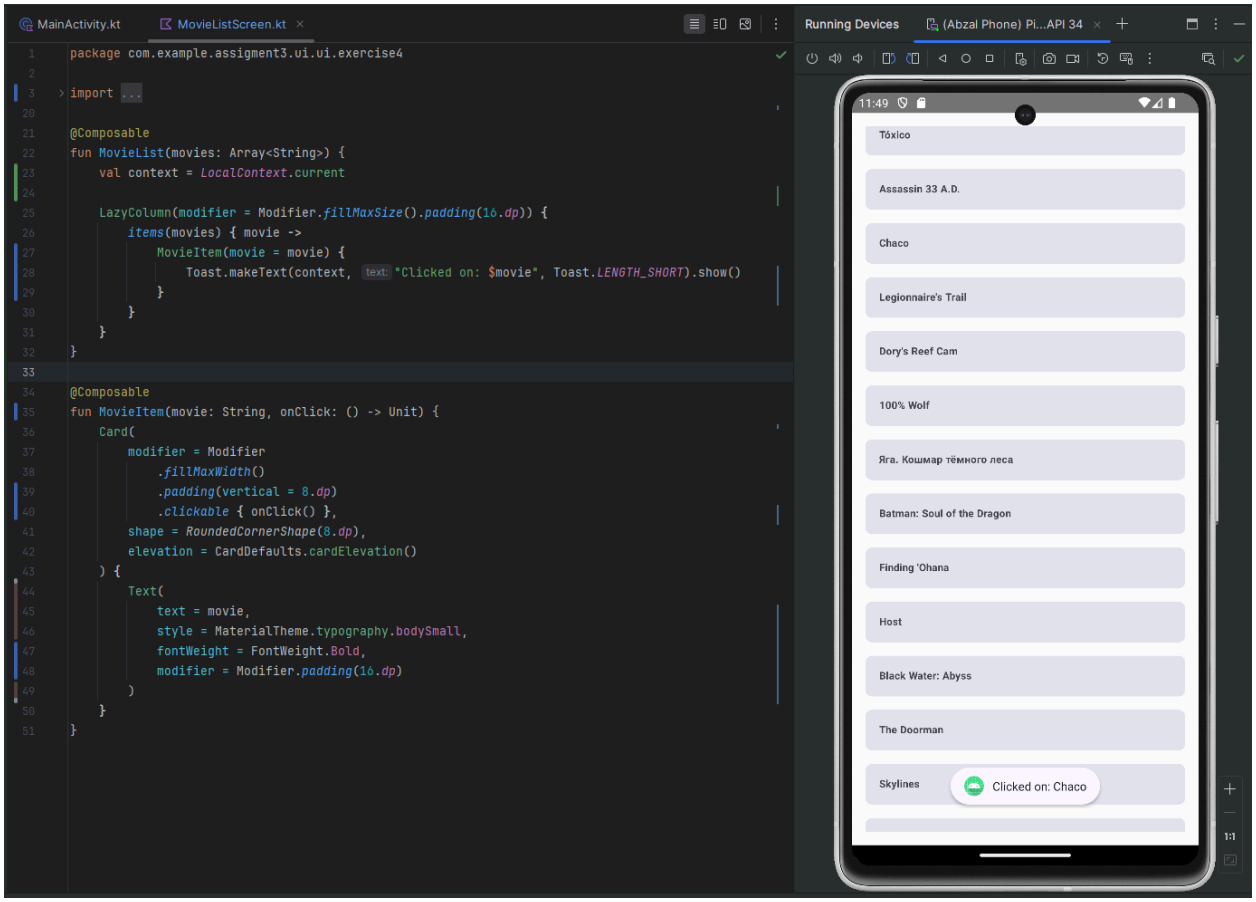
The movie list was displayed efficiently, and the use of LazyColumn ensured smooth scrolling performance.



Screenshot 8. Code and app screenshot that represents the LazyColumn and MovieList implementation.

Exercise 5: Item Click Handling

Clicking on movie items worked as intended, providing user feedback through Toast messages.



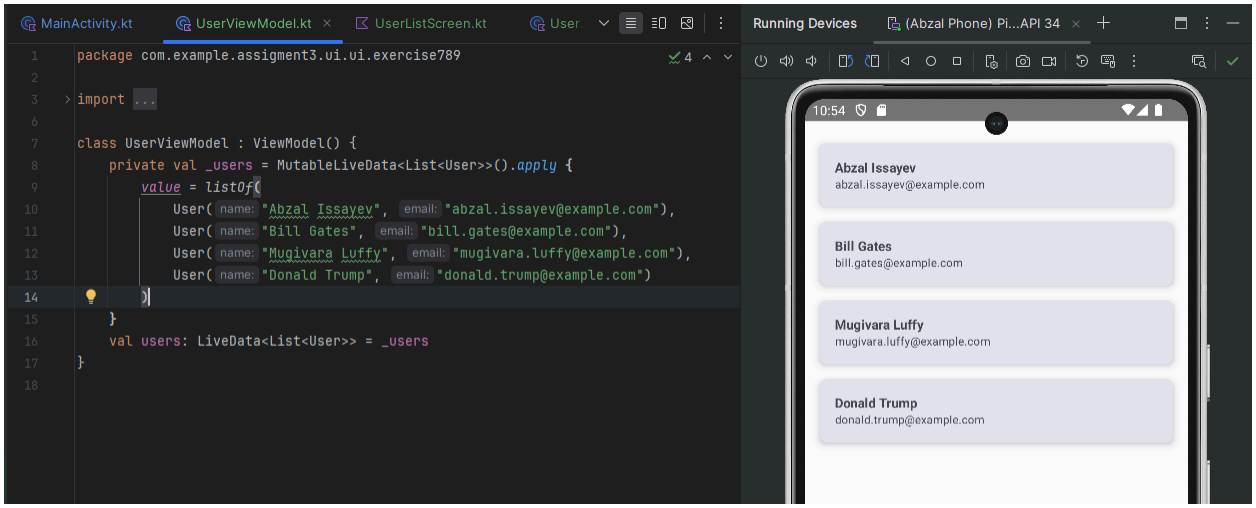
Screenshot 8. Code and app screenshot that represents the Toast message implementation.

Exercise 6: ViewHolder Pattern

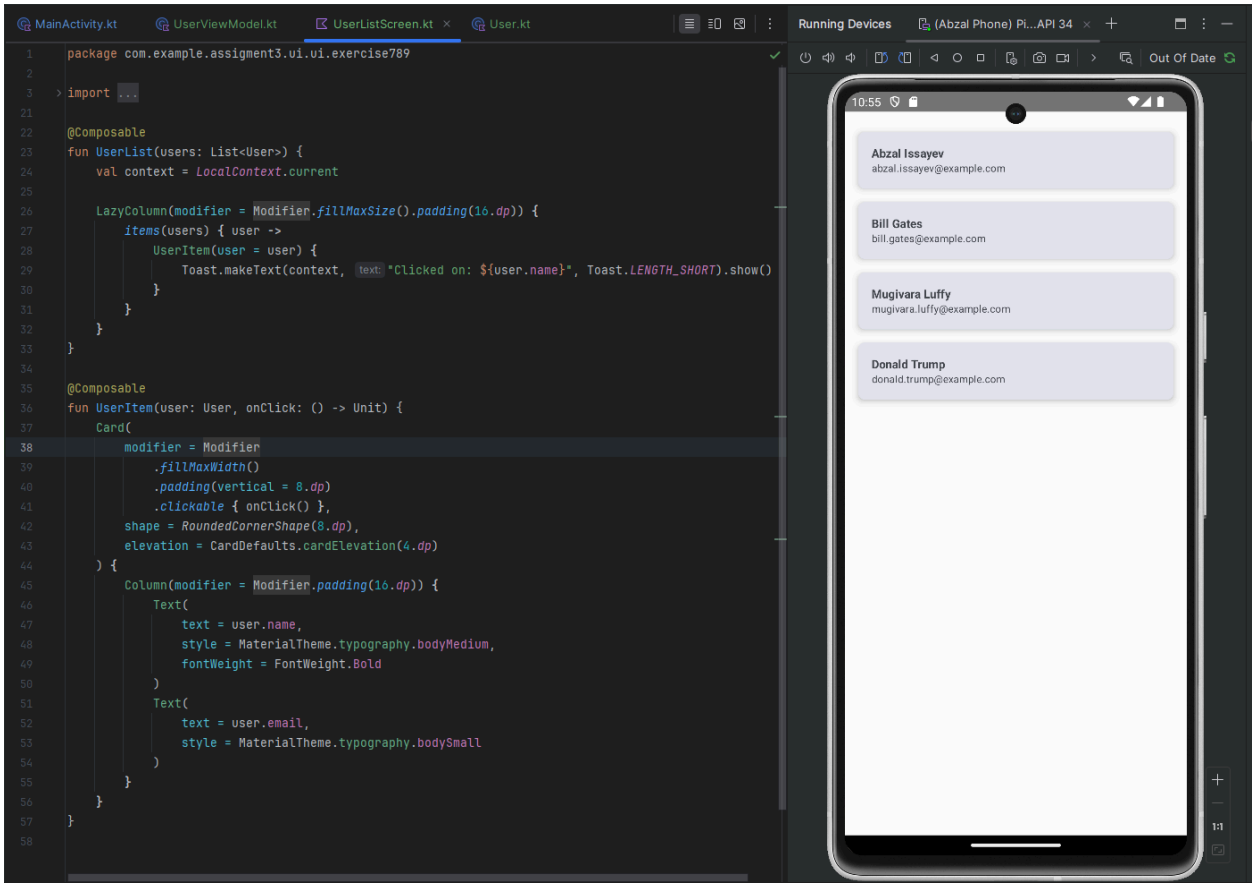
The MovieItem composable provided a clean, reusable representation of each movie, similar to a traditional ViewHolder, ensuring efficient rendering.

Exercise 7: Implementing ViewModel

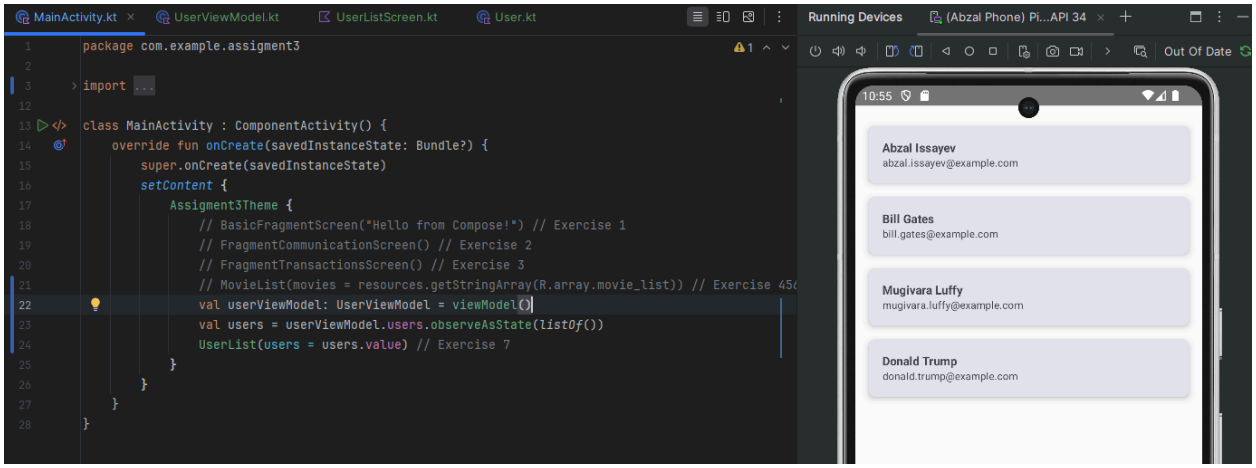
The UserViewModel effectively managed the list of users, and changes were observed by the UI components, which updated accordingly.



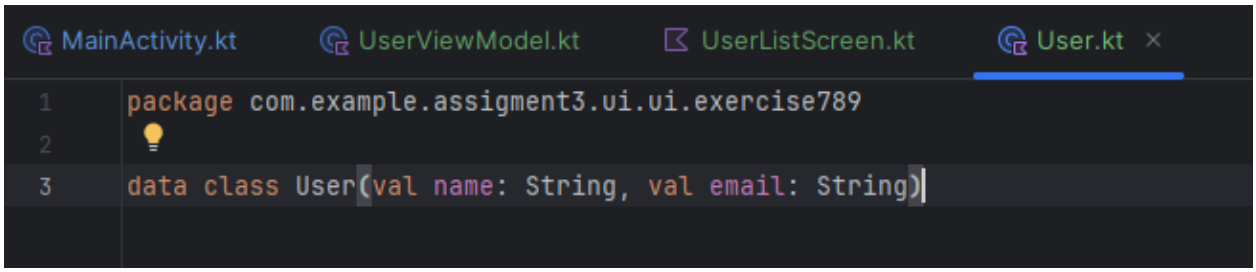
Screenshot 9. Code and app screenshot that represents the UserViewModel implementation.



Screenshot 10. Code and app screenshot that represents the User List implementation.



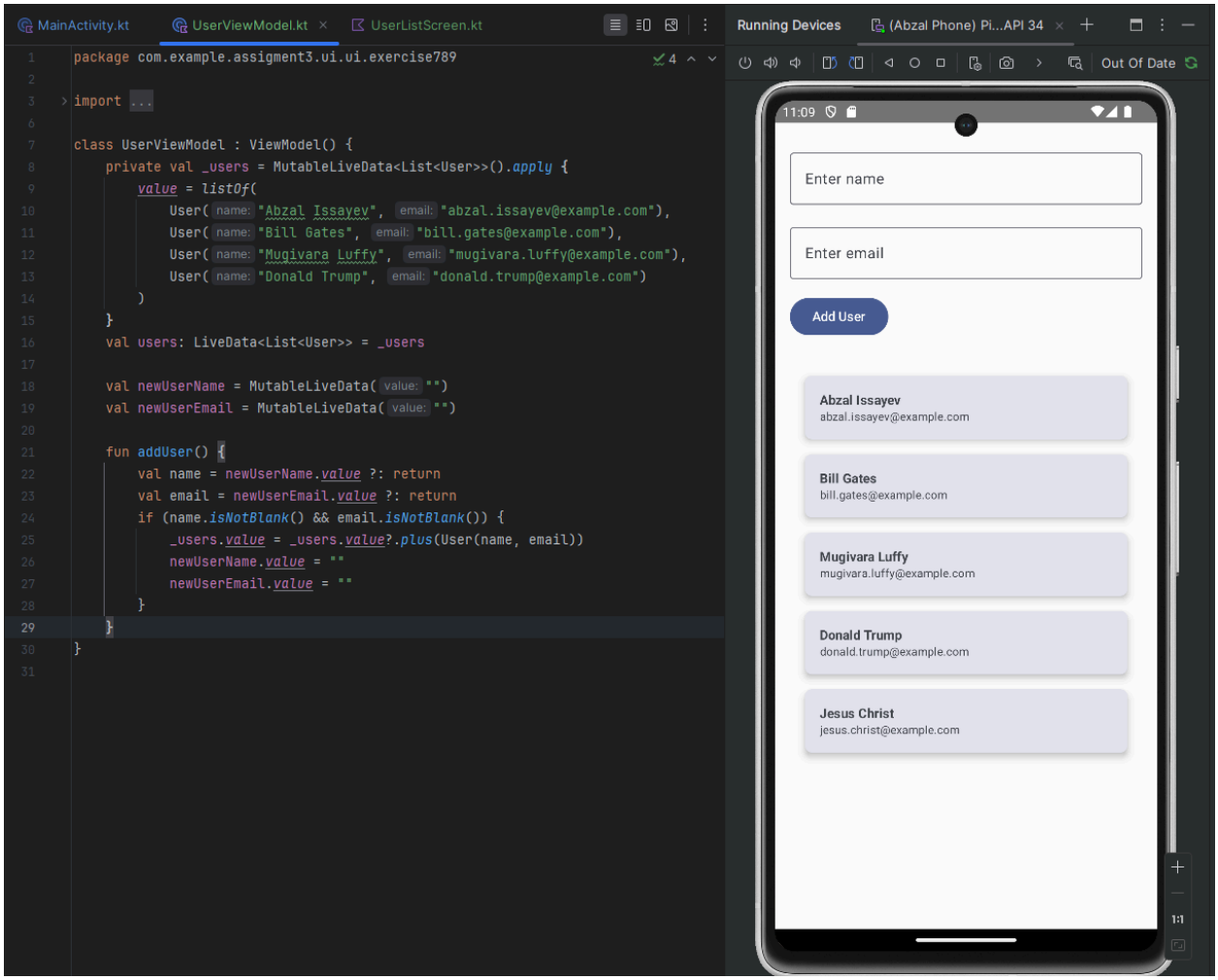
Screenshot 11. Code and app screenshot that represents the MainActivity implementation with viewModel.



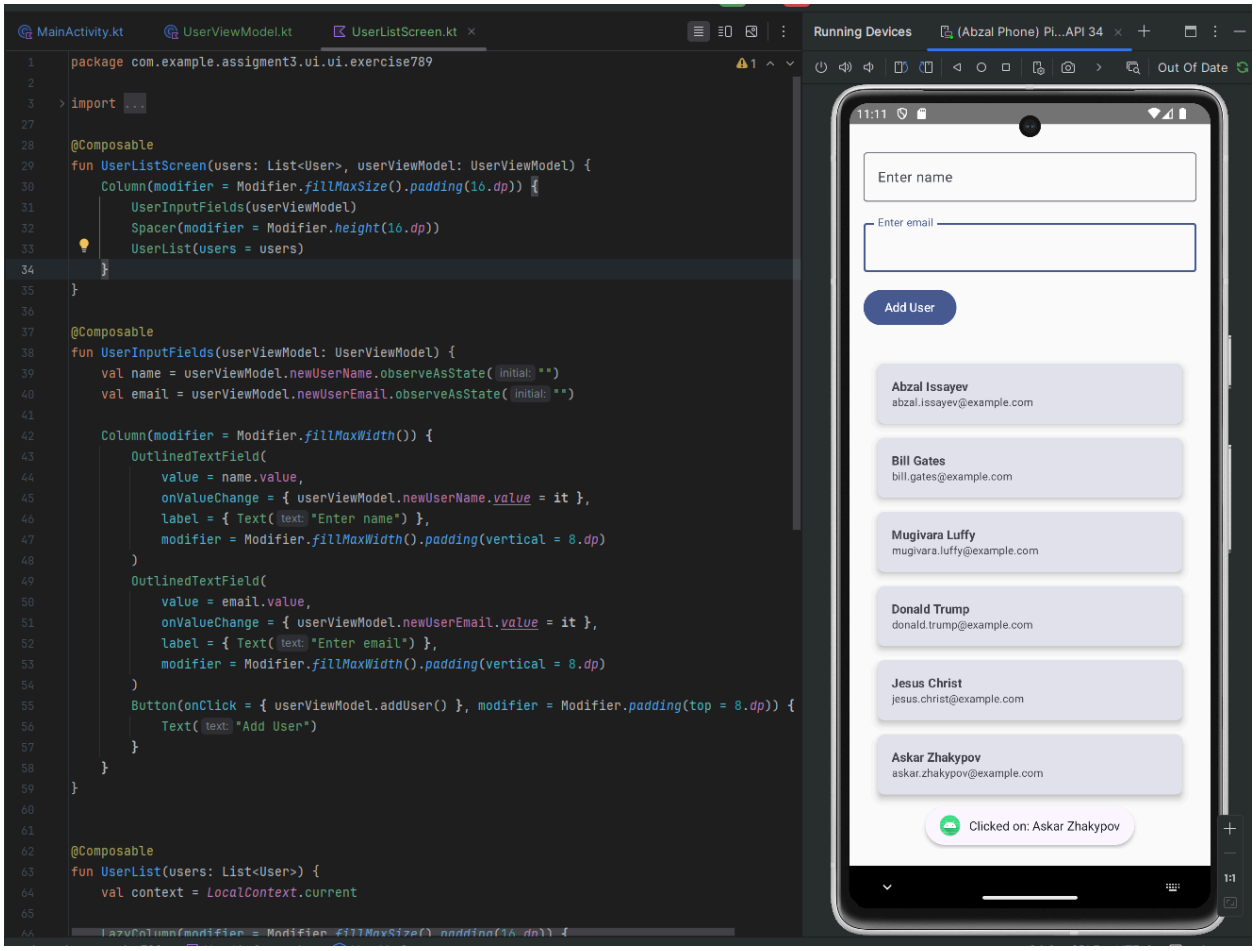
Screenshot 12. Code screenshot that represents the User data class implementation.

Exercise 8: MutableLiveData for Input Handling

User input was successfully managed through MutableLiveData, and the UI reflected changes as expected.



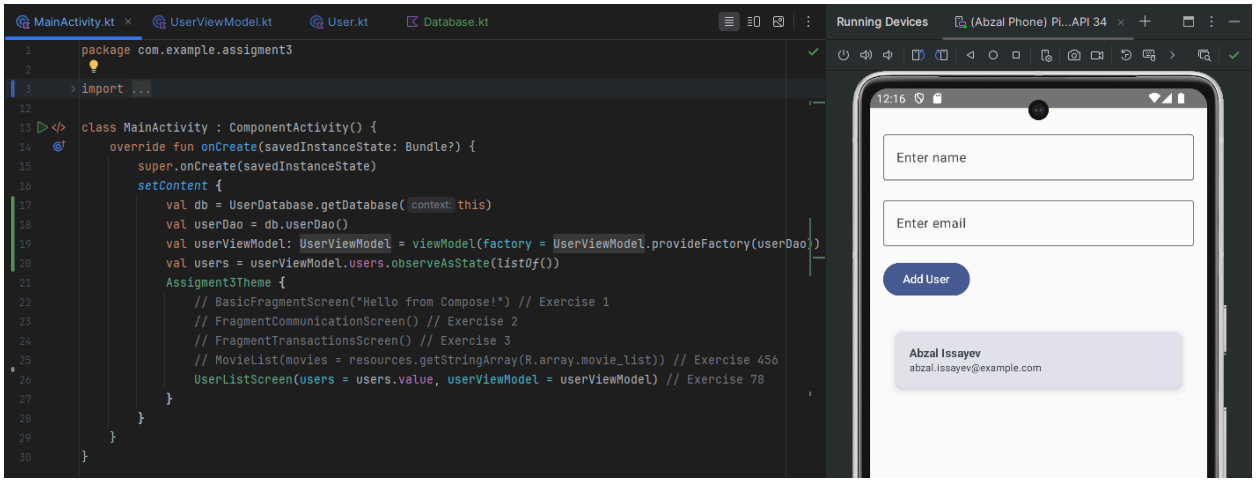
Screenshot 13. Code and app screenshot that represents the UserViewModel implementation with input handling.



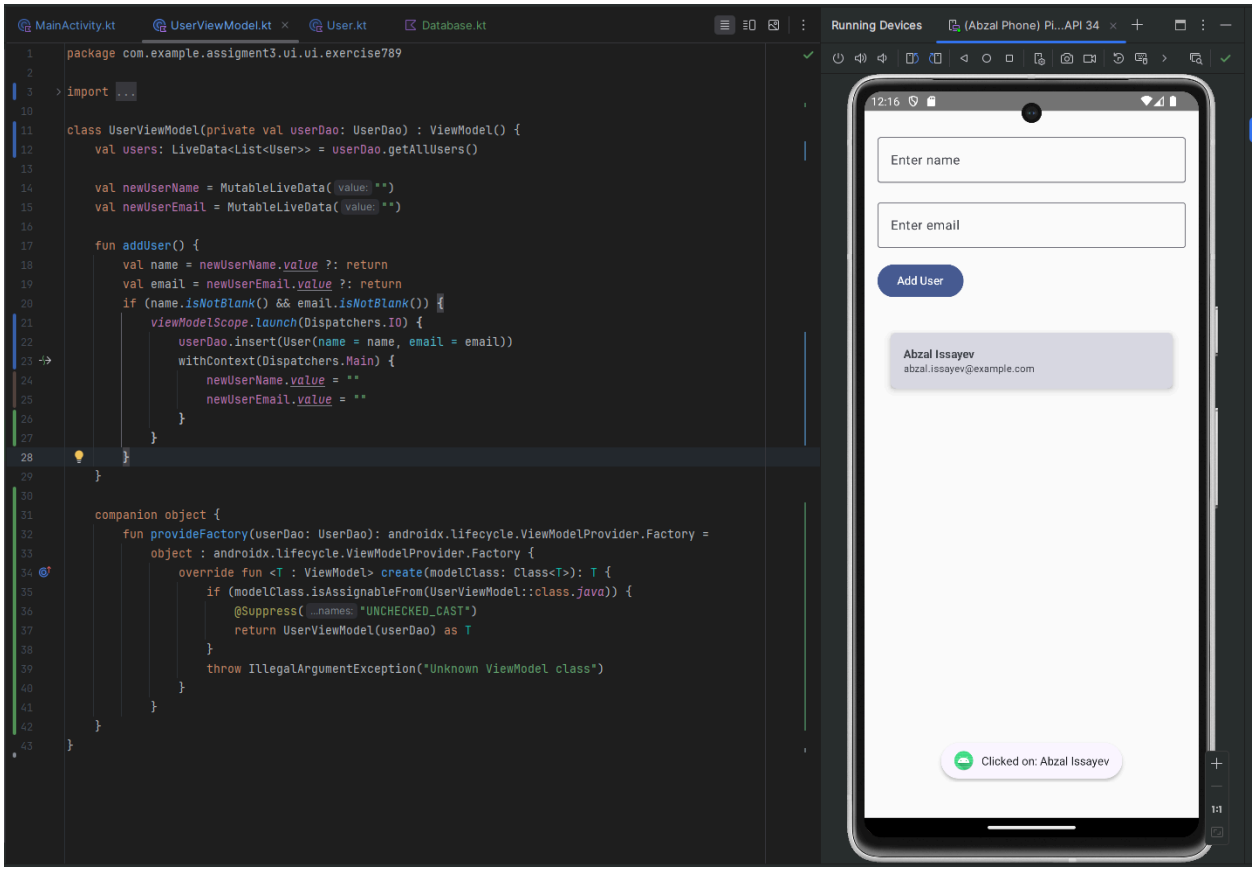
Screenshot 14. Code and app screenshot that represents the InputFields implementation with input handling working correctly.

Exercise 9: Data Persistence

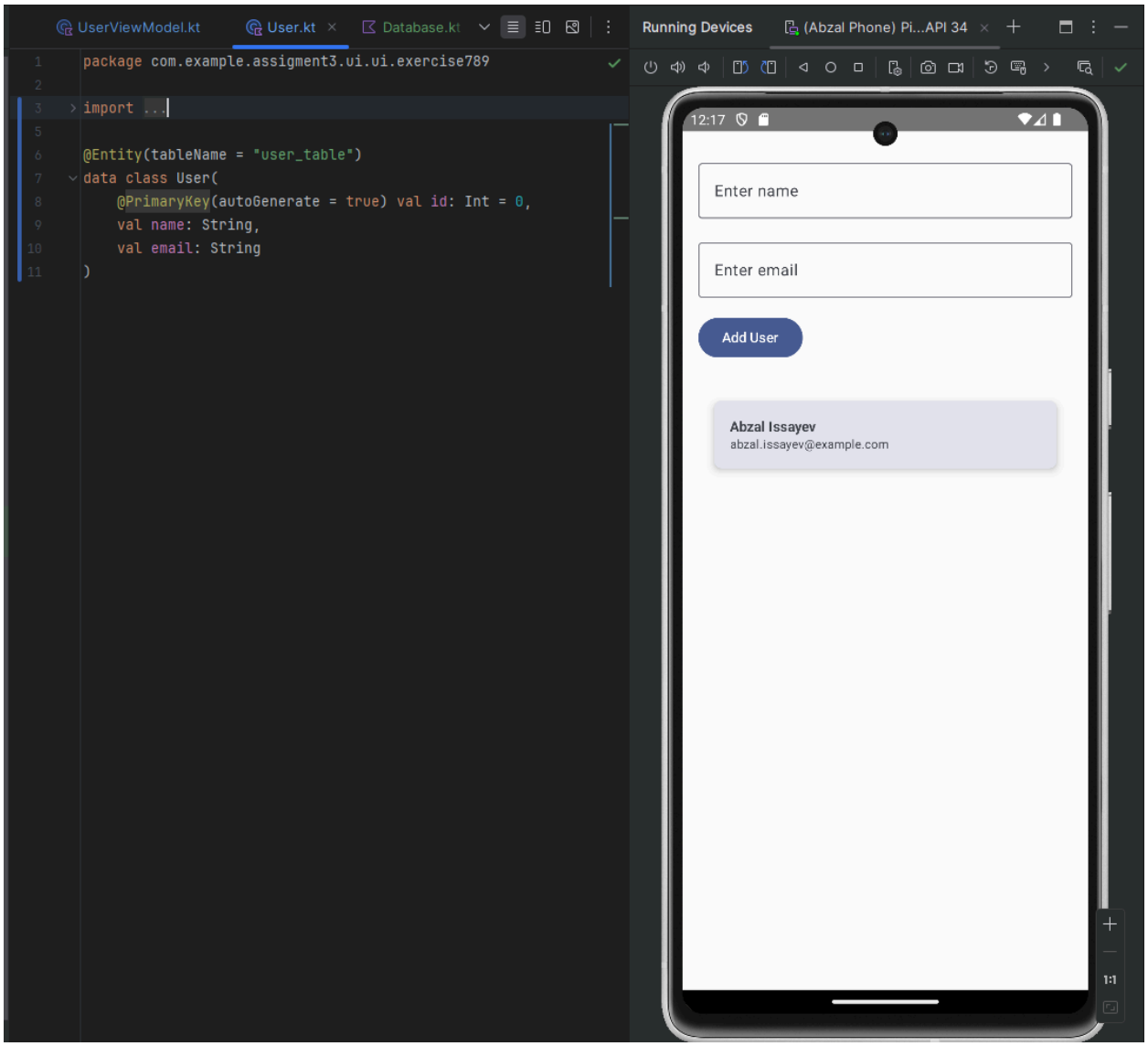
User data was successfully persisted in the Room database, providing consistent data even after app restarts.



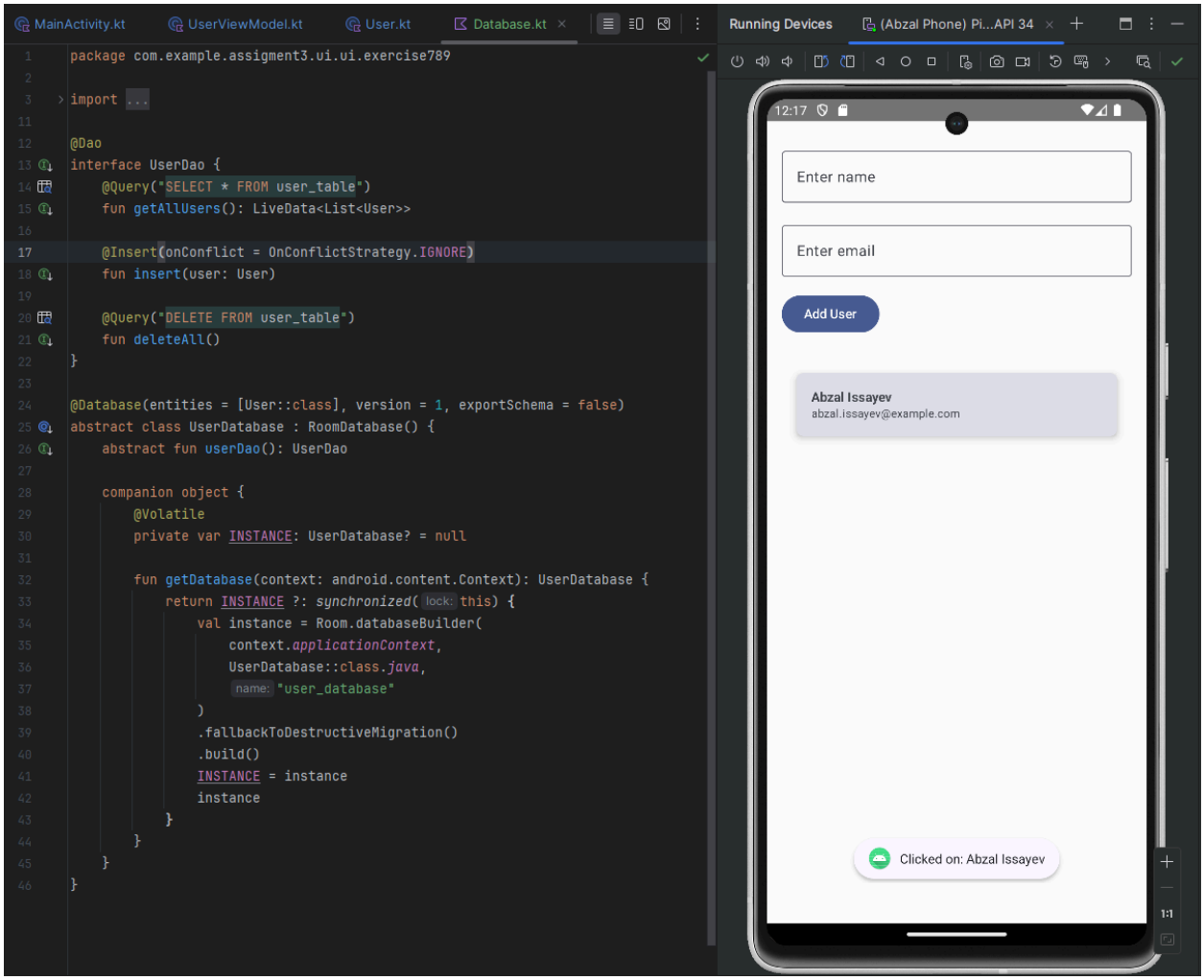
Screenshot 15. Code and app screenshot that represents the initialization of the Dao database and ViewModel from the MainActivity.



Screenshot 16. Code and app screenshot that represents modified UserViewModel to provide custom viewModelFactory.



Screenshot 17. Code and app screenshot that represents modified User data class to create a table of users inside the database.



Screenshot 18. Code and app screenshot that represents Dao database.

Conclusion

The exercises provided a broad learning experience, touching on key concepts and components of Android development: Fragments, RecyclerView, ViewModel and LiveData through Jetpack Compose, examining these traditional elements and gaining practical insights into how they can be reimagined in a modern context.

Fragments have traditionally helped to create reusable UI components with their own life cycle. In this exercise, Jetpack Compose's Composable Functions replaced Fragments, providing an easier and modern approach to UI creation. This change simplified the UI code and made it easier to handle lifecycle events without the burden of manually managing Fragment lifecycles.

RecyclerView, normally used to display lists of data, was replaced in Jetpack Compose with LazyColumn, which provides an efficient way to display large data sets while reducing the

complexity of traditional adapters. LazyColumn automatically handles recycled items in a similar way to RecyclerView, but is simpler to implement.

ViewModel played an important role in managing UI-related data as the configuration changes, ensuring that data continues uninterrupted. Maintaining state, managing user input and interacting with Room for data persistence made the application more robust and efficient.

LiveData was used to track data changes and automatically update the UI; LiveData's lifecycle awareness feature ensured that the UI only reacted to changes when it was active, preventing unwanted updates and crashes. This made the UI more reactive and ensured a consistent user experience.

Concluding everything, these components are essential for building sustainable, responsive and efficient Android apps, and Jetpack Compose has made development faster and more intuitive by offering a modern alternative to many traditional components.

References

1. https://developer.android.com/guide/fragments
2. https://medium.com/@williamrai13/exploring-composable-functions-in-android-an-introductory-guide-837504d51064
3. https://developer.android.com/develop/ui/views/layout/recyclerview
4. https://medium.com/@ramadan123sayed/lazycolumn-in-jetpack-compose-fa3287ef84da
5. https://developer.android.com/topic/libraries/architecture/viewmodel
6. https://developer.android.com/topic/libraries/architecture/livedata
7. https://developer.android.com/develop/ui/compose/side-effects
8. https://developer.android.com/reference/androidx/lifecycle/LifecycleEventObserver
9. https://developer.android.com/develop/ui/compose/navigation
10. https://medium.com/@sandeepkella23/what-is-viewholder-design-pattern-in-android-8e0fb075bb97
11. https://developer.android.com/develop/ui/compose/state
12. https://developer.android.com/training/data-storage/room

Appendix

Include any additional material (like screenshots of the app, if applicable).