

Assignment 3, mobile programming
Android Development Exercises Report
Temirbolat Amanzhol
06.11.2024

Table of Contents

1. Introduction	p. 1
2. Exercise Descriptions	p. 2
○ Fragment Lifecycle	
○ Fragment Communication	
○ Fragment Transactions	
○ Building a RecyclerView	
○ Item Click Handling	
○ ViewHolder Pattern	
○ Implementing ViewModel	
○ MutableLiveData for Input Handling	
○ Data Persistence	
3. Code Snippets	p. 15
4. Results	p. 28
5. Conclusion	p. 29

Introduction

This report covers the core components of Android development such as Fragments, RecyclerView, ViewModel, and LiveData. These tools are key to creating interactive and responsive apps that can efficiently handle large amounts of data and preserve their state even when the device configuration changes.

1. Fragments. Fragments are used to separate the UI into separate, manageable parts. This allows you to create flexible layouts that can adapt to different screen sizes and device configurations. Fragments can be easily reused across activities or switched between, ensuring a smooth and dynamic experience for your app.

2. RecyclerView. This is a component that was created to display large lists of data more efficiently compared to 'ListView'. 'RecyclerView' allows you to dynamically add and remove items, easily manage their positioning, and handle interaction events such as taps on items. Using the 'ViewHolder' pattern in 'RecyclerView' improves performance by minimizing unnecessary calls to UI components.

3. `ViewModel` and `LiveData`. These components help separate the application logic from its interface. `ViewModel` allows you to save the state of the data so that it is not lost when the screen orientation changes or the activity is recreated. `LiveData` allows you to observe data changes and automatically update the interface, which makes it easier to manage asynchronous data and makes the code cleaner and more maintainable.

Each of these components solves specific problems, making Android application development more convenient, efficient and scalable. For example, `ViewModel` and `LiveData` help prevent memory leaks and improve the responsiveness of the interface, and `RecyclerView` makes it possible to smoothly display long lists of data. In practice, these components are often used together, creating a well-thought-out and responsive interface with minimal effort from the developer.

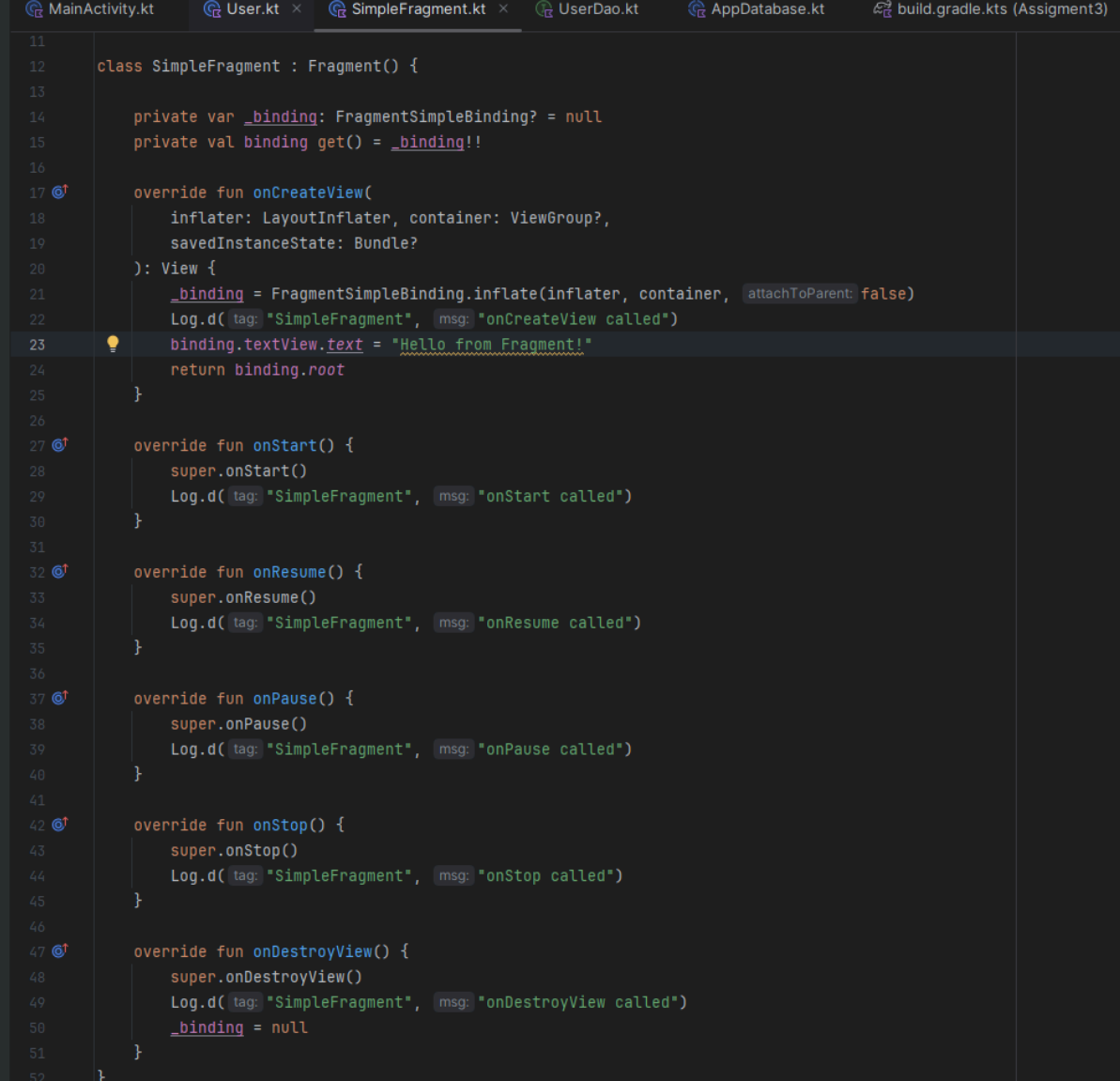
The goal of these exercises is to learn and apply key Android components to create a modern and interactive application. During the exercises, a functional architecture was built that allows managing the state of data, its presentation, and the interaction between the various interface components.

Fragment Lifecycle

Objective: To understand the lifecycle of fragments in Android, learn when different lifecycle methods are triggered, and practice tracking these events.

Description of Implementation: Fragments in Android have their own lifecycle, which is similar to that of activities but with some unique differences. To explore this, we created a simple fragment that displays a message on the screen, like “Hello from Fragment!”. Then, we implemented the main lifecycle methods: `onCreateView`, `onStart`, `onResume`, `onPause`, `onStop`, and `onDestroyView`. In each of these methods, we added code to log a message so we could track when each method is called.

With this setup, we can see the exact sequence of method calls as the fragment is created, displayed, paused, or destroyed. For instance, when the fragment is first created, `onCreateView` is called, followed by `onStart` and then `onResume`. When the fragment is no longer visible, `onPause` is triggered, followed by `onStop`, and finally `onDestroyView` when it is removed from memory.



```

11
12 class SimpleFragment : Fragment() {
13
14     private var _binding: FragmentSimpleBinding? = null
15     private val binding get() = _binding!!
16
17     override fun onCreateView(
18         inflater: LayoutInflater, container: ViewGroup?,
19         savedInstanceState: Bundle?
20     ): View {
21         _binding = FragmentSimpleBinding.inflate(inflater, container, attachToParent: false)
22         Log.d(tag: "SimpleFragment", msg: "onCreateView called")
23         binding.textView.text = "Hello from Fragment!"
24         return binding.root
25     }
26
27     override fun onStart() {
28         super.onStart()
29         Log.d(tag: "SimpleFragment", msg: "onStart called")
30     }
31
32     override fun onResume() {
33         super.onResume()
34         Log.d(tag: "SimpleFragment", msg: "onResume called")
35     }
36
37     override fun onPause() {
38         super.onPause()
39         Log.d(tag: "SimpleFragment", msg: "onPause called")
40     }
41
42     override fun onStop() {
43         super.onStop()
44         Log.d(tag: "SimpleFragment", msg: "onStop called")
45     }
46
47     override fun onDestroyView() {
48         super.onDestroyView()
49         Log.d(tag: "SimpleFragment", msg: "onDestroyView called")
50         _binding = null
51     }
52 }

```

Expected Outcome: The fragment should display properly on the screen, and log messages should appear for each stage of its lifecycle. This setup allows the developer to observe which lifecycle methods are triggered and in what order as the fragment is added, displayed, paused, or removed. Understanding the fragment lifecycle is essential for managing resources and data in an Android application. For example, if you need to pause a specific feature when the fragment is no longer visible, you can use the `onPause` method. And to release resources when the fragment is fully destroyed, you can use `onDestroyView`. This approach helps build stable, responsive applications that efficiently use device resources like memory and processing power.

Fragment Communication

Objective: To learn how to establish communication between two fragments in an Android application using a shared `ViewModel`. This setup allows data entered in one fragment to be

displayed in another fragment, enabling smooth data sharing without tightly coupling the fragments.

Description of Implementation: In this exercise, we created two fragments: an Input Fragment and an Output Fragment. The Input Fragment includes an `EditText` field where users can type text, while the Output Fragment contains a `TextView` to display this text.

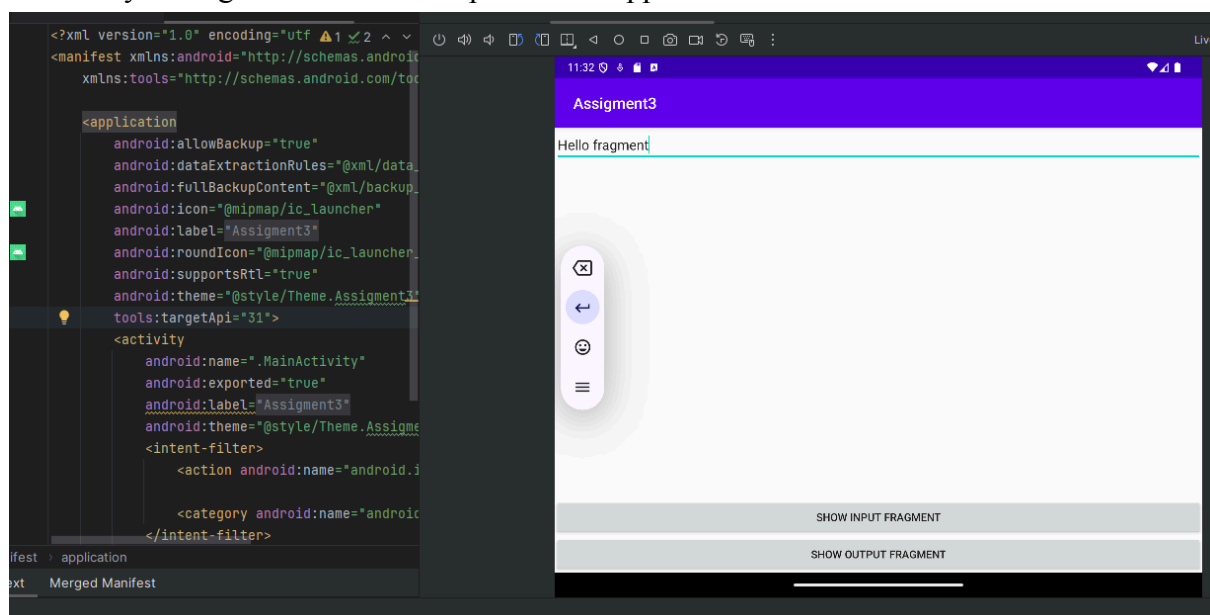
To enable communication between these fragments, we used a shared `ViewModel`. First, we defined a `SharedViewModel` class that contains a `MutableLiveData` object for storing the text data. The `ViewModel` provides methods to update the text and observe changes. Here's how the communication works:

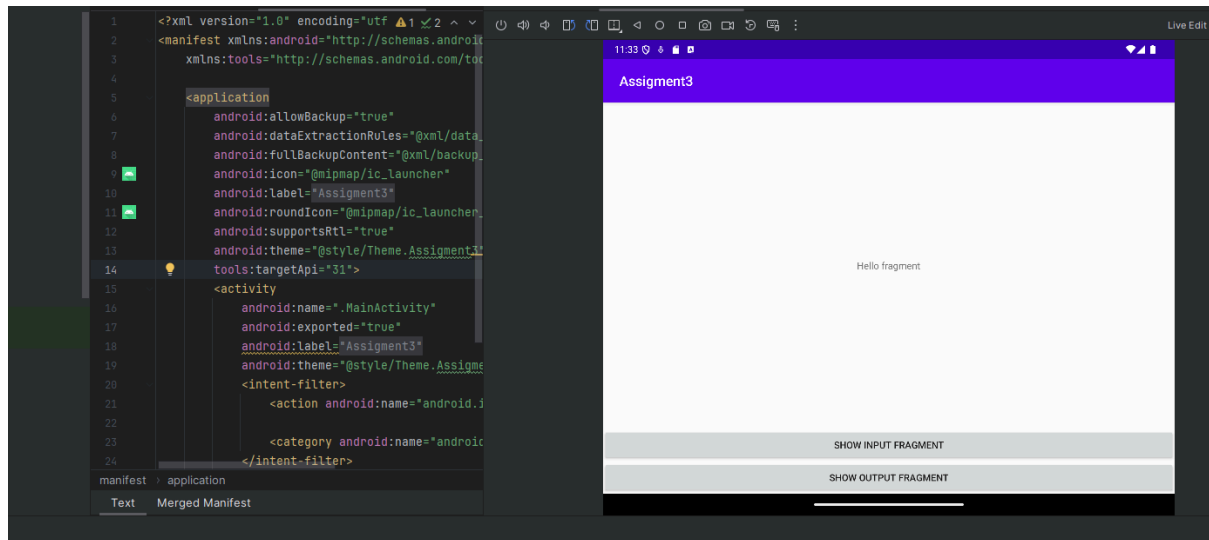
- In the Input Fragment, whenever the user types in the `EditText`, the text is updated in the `ViewModel` using a method that sets the `MutableLiveData` value.
- The Output Fragment observes this `LiveData` in the `ViewModel`. When the `LiveData` changes, the observer in the Output Fragment is triggered, updating the `TextView` with the new text from the Input Fragment.

By using a shared `ViewModel`, we allow these fragments to communicate indirectly through a shared data source, making the design more modular and manageable.

Expected Outcome:

When the user types text in the Input Fragment, the same text should automatically appear in the Output Fragment. This demonstrates effective inter-fragment communication using `ViewModel` and `LiveData`. Fragment communication is crucial in Android applications where different parts of the UI need to share data. Using `ViewModel` for this purpose allows fragments to communicate without directly referencing each other, making the app architecture more flexible and reducing dependencies between fragments. This approach is particularly helpful for modular development, as it enables easier maintenance, testing, and reusability of fragments in different parts of an application.





Fragment Transactions

Objective: To understand and implement fragment transactions, which involve adding, replacing, and removing fragments within an activity. This helps in building dynamic UIs that can adapt to user interactions by showing or hiding different fragments.

Description of Implementation: In this exercise, we set up an activity to manage two fragments, allowing users to switch between them using buttons. The activity includes a container (`FrameLayout`) to host the fragments and two buttons to control the fragment display. We used the fragment transaction methods `add`, `replace`, and `remove` to manage the fragments within this container.

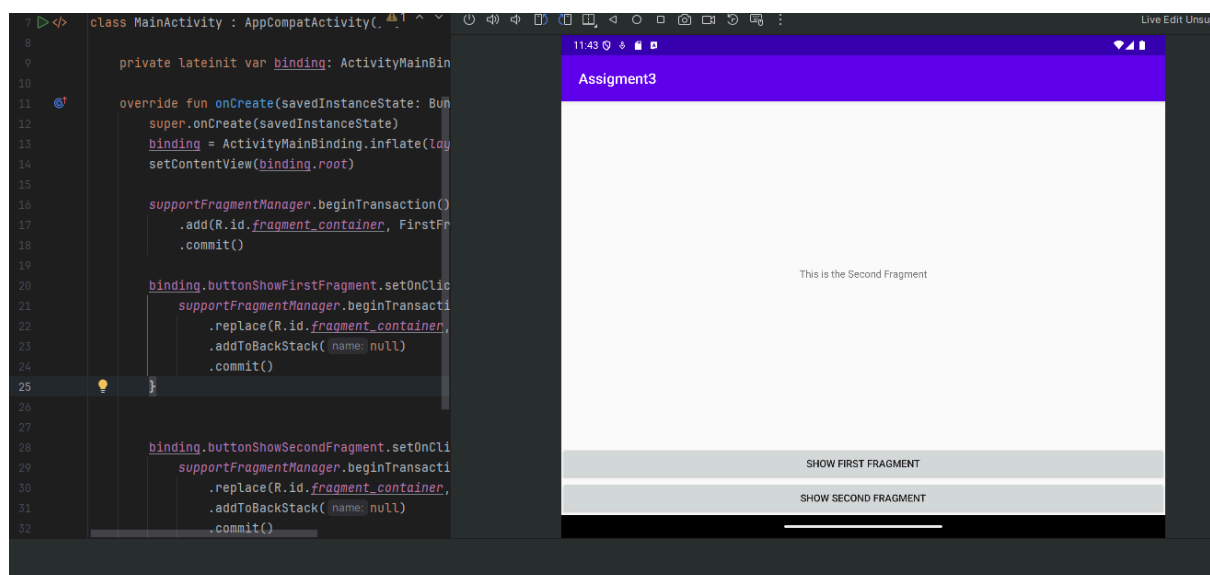
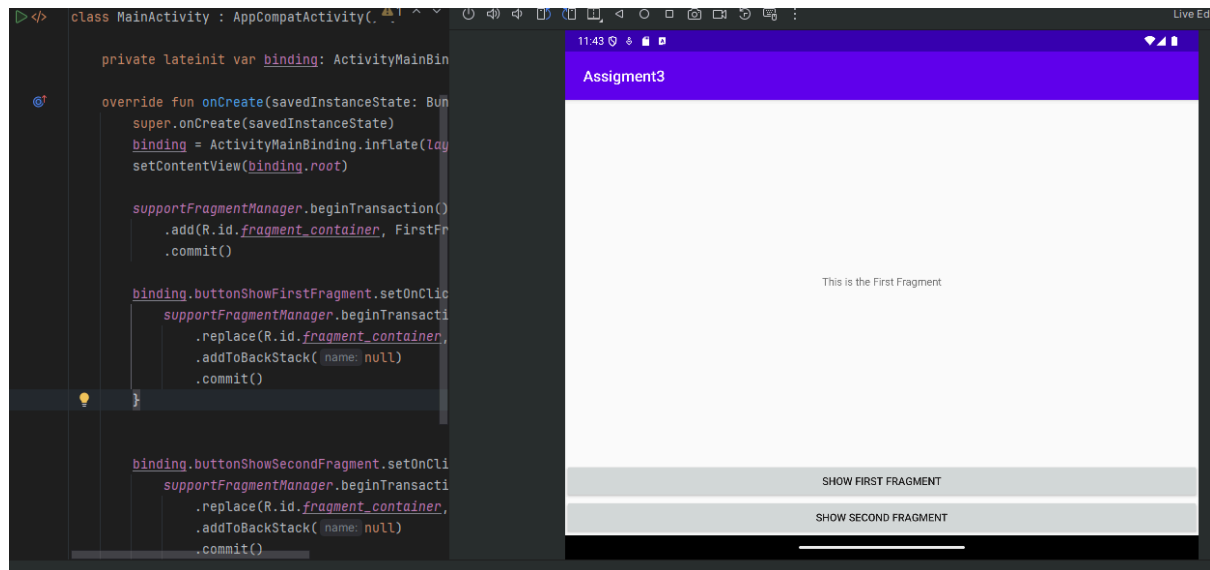
Adding a Fragment: When the user clicks the "Add Fragment" button, we use the `add` method to place a fragment into the container without removing any existing fragment. This approach is useful when layering fragments or stacking multiple screens.

Replacing a Fragment: To switch between fragments, we used the `replace` method, which removes the current fragment from the container and adds a new one in its place. This method is ideal for switching content within a single container, as it ensures only one fragment is visible at a time.

Removing a Fragment: We implemented a "Remove Fragment" button that uses the `remove` method to delete the current fragment from the container. This leaves the container empty until another fragment is added.

Additionally, we used `addToBackStack` with these transactions to allow users to navigate back through the fragment stack using the back button. This provides a seamless navigation experience, as users can return to the previous fragment.

Expected Outcome: The user should be able to add, switch, and remove fragments in the container by clicking the respective buttons. The back button should also function to navigate back through the stack of fragments.



Building a RecyclerView

Objective: To create a 'RecyclerView' for displaying a list of items efficiently, using an 'Adapter' to populate the 'RecyclerView' with data. This exercise focuses on learning how to set up and use 'RecyclerView' to display lists in Android applications.

Description of Implementation: In this exercise, we created a 'RecyclerView' to display a list of favorite movies. Setting up 'RecyclerView' involved several steps:

1. Defining the Item Layout: First, we created an XML layout file (`item_movie.xml`) for individual items in the list. This layout includes a `TextView` to display the title of each movie.

2. Setting Up the RecyclerView in the Activity Layout: In the activity's main layout file (`activity_main.xml`), we added the `RecyclerView` widget, which will display the list of movie items.

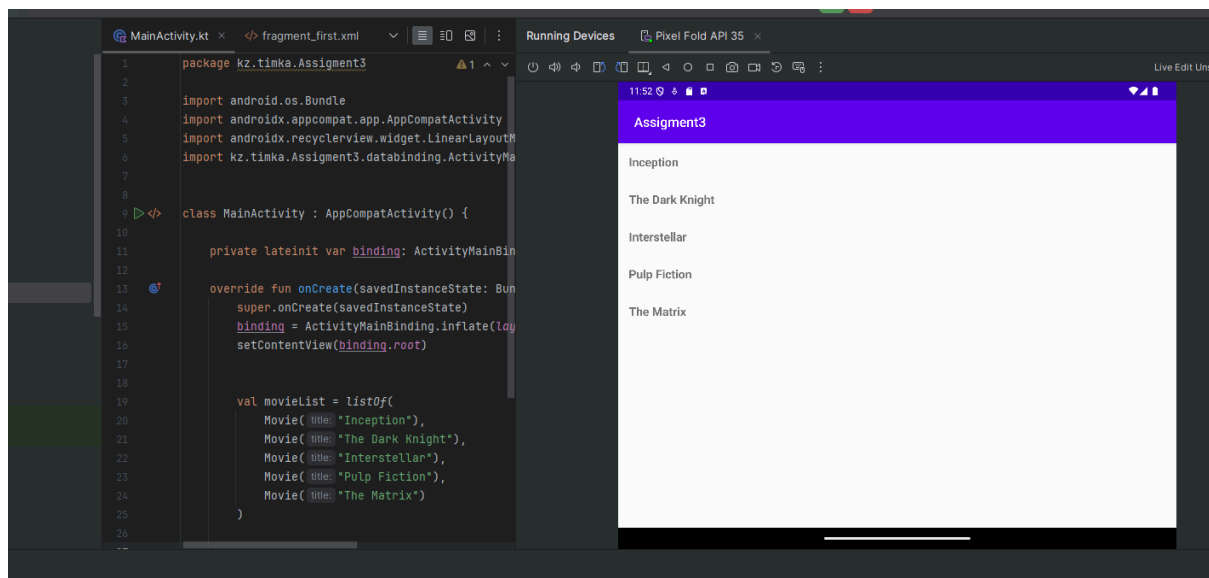
3. Creating the Data Class: We defined a `Movie` data class with a single property, `title`, to represent each movie in the list.

4. Implementing the Adapter: We created a `MovieAdapter` class, which extends `RecyclerView.Adapter`. The adapter is responsible for binding the data from our list of movies to each item view. Inside `MovieAdapter`, we also defined a `MovieViewHolder` class to hold references to the `TextView` in each item view, which displays the movie title.

5. Connecting Data to RecyclerView: In `MainActivity`, we created a list of movie titles and initialized the `MovieAdapter` with this data. Then, we set up the `RecyclerView` by assigning it a `LayoutManager` (to control the layout of items) and connecting it to the `MovieAdapter`.

With this setup, the `RecyclerView` efficiently displays the list of movies, using `MovieAdapter` to populate each item in the list.

Expected Outcome: The `RecyclerView` should display a scrollable list of favorite movies, each represented by a title. The list should be responsive and efficiently manage memory, reusing item views as they scroll off-screen.



Item Click Handling

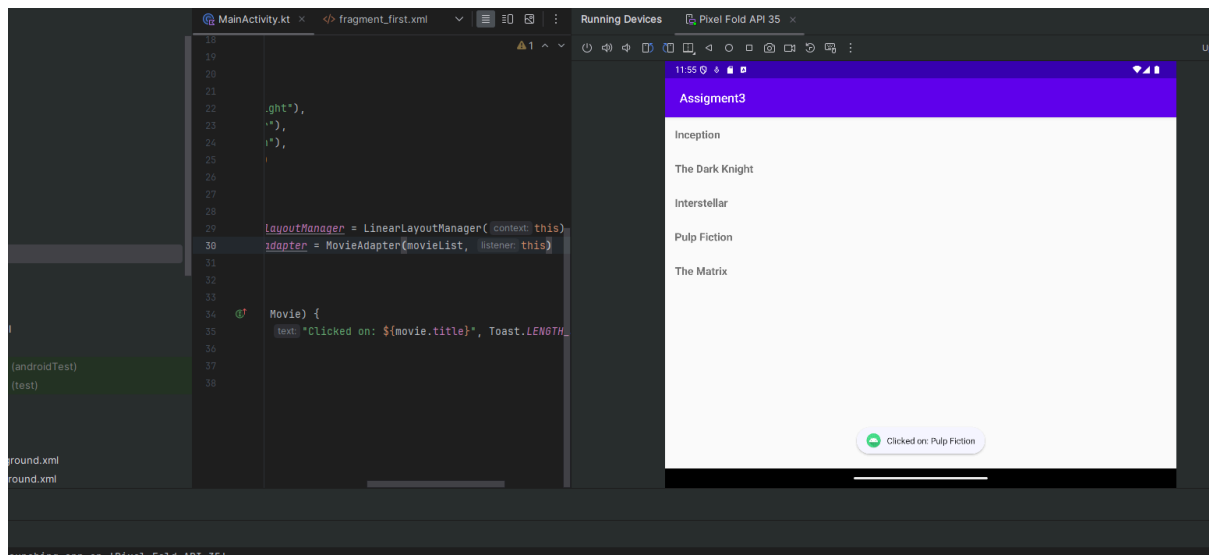
Objective: To implement item click handling in a `RecyclerView`, allowing each item in the list to respond to user interaction. This exercise demonstrates how to detect clicks on individual items and trigger specific actions, such as displaying a message with the item's name.

Description of Implementation: In this exercise, we extended the `RecyclerView` from Exercise 4 by adding click functionality to each item in the list. When an item is clicked, a `Toast` message displays the name of the clicked movie. Here's how we implemented this:

- 1. Defining a Click Listener Interface:** To handle item clicks, we created an `OnItemClickListener` interface in `MovieAdapter`. This interface defines a method `onItemClick(movie: Movie)`, which takes a `Movie` object as a parameter. This allows us to send data about the clicked item back to the activity.
- 2. Implementing the Click Listener in MainActivity:** In `MainActivity`, we implemented the `OnItemClickListener` interface. This allowed us to override the `onItemClick` method to define the specific action to take when an item is clicked. In our case, we used a `Toast` message to display the title of the clicked movie.
- 3. Setting Up Click Handling in the Adapter:** Inside `MovieAdapter`, we set up the click listener in `onBindViewHolder` by attaching it to each item view. Whenever an item is clicked, the `onItemClick` method in `MainActivity` is triggered with the clicked `Movie` object.
- 4. Connecting the Listener to the Adapter:** Finally, we passed `MainActivity` as the listener when initializing `MovieAdapter`, allowing the adapter to call back to the activity when an item is clicked.

This setup enables each item in the `RecyclerView` to respond to click events, passing relevant data (the movie title) back to `MainActivity` to be displayed in a `Toast`.

Expected Outcome: When a user clicks on a movie title in the `RecyclerView`, a `Toast` message should appear, displaying the title of the selected movie. This confirms that item click handling works correctly and that each item can respond to user interactions.



ViewHolder Pattern

Objective: To optimize the performance of a RecyclerView by implementing the ViewHolder pattern within the adapter. This pattern reduces the number of calls to findViewById and improves scrolling performance by reusing item views in the RecyclerView.

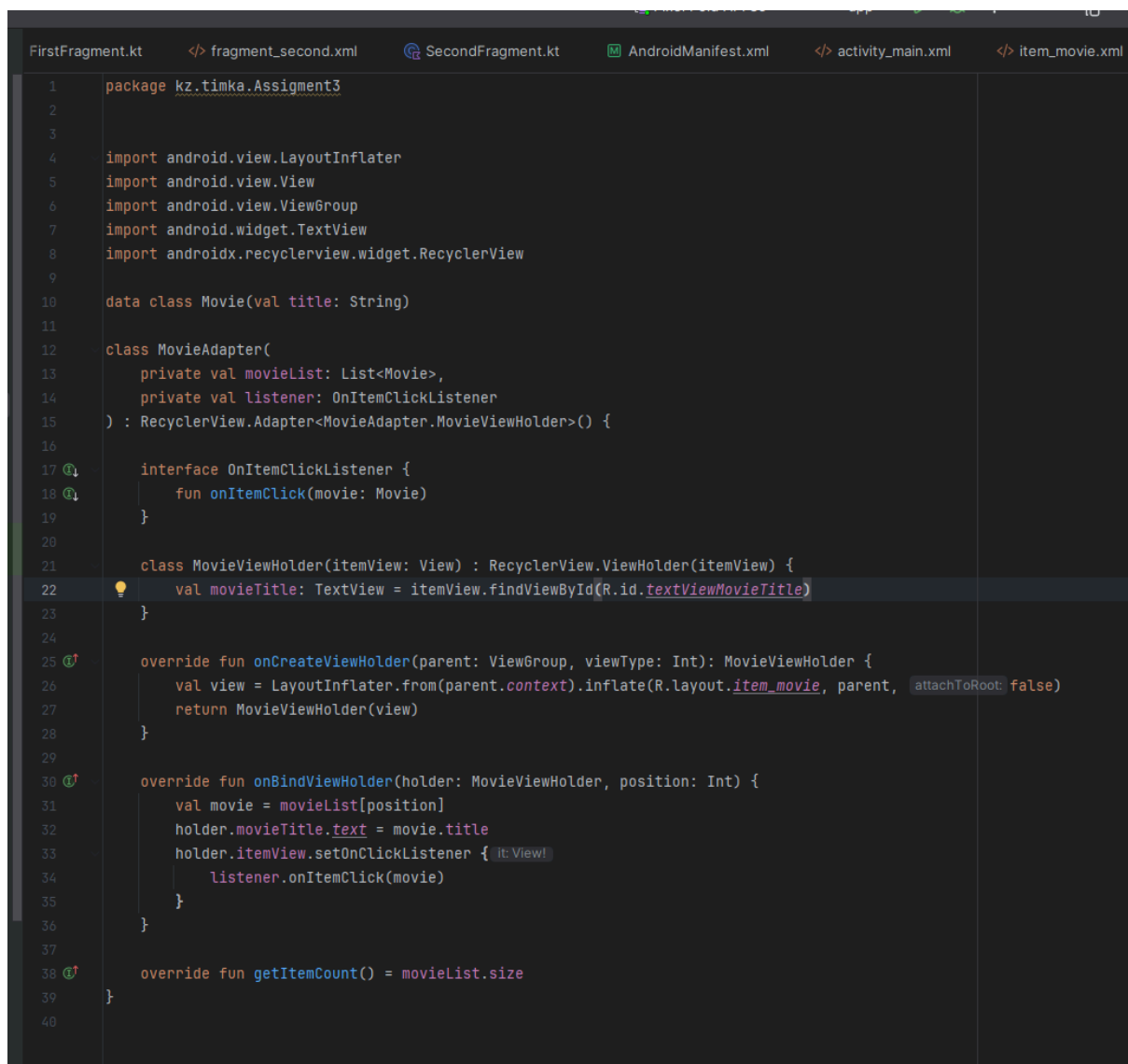
Description of Implementation: In this exercise, we optimized our RecyclerView from previous exercises by fully implementing the ViewHolder pattern within MovieAdapter. The ViewHolder pattern is key to making RecyclerView efficient, as it minimizes unnecessary lookups and enhances memory management.

Here's how we implemented the ViewHolder pattern:

- 1. Creating the ViewHolder Class:** Inside MovieAdapter, we defined an inner class MovieViewHolder that extends RecyclerView.ViewHolder. This class holds references to the individual views (like TextView) for each item. By storing these views once, the ViewHolder pattern prevents repeated calls to findViewById, which can be costly in terms of performance.
- 2. Binding Data to the ViewHolder:** We used onBindViewHolder in MovieAdapter to bind data to each MovieViewHolder. In this method, we populated the views (e.g., setting the movie title in the TextView) using data from the Movie object at the specified position. Since ViewHolder instances are reused as we scroll, this binding process is very efficient.
- 3. Using ViewHolder in onCreateViewHolder:** When a new item view needs to be created, onCreateViewHolder inflates the item_movie.xml layout and returns a MovieViewHolder instance. By returning a ViewHolder, the RecyclerView can manage a pool of reusable item views, meaning each view is only created once and then reused.

4. Benefits of ViewHolder Pattern: With this implementation, the RecyclerView reuses item views that scroll off-screen, meaning we avoid creating new views unnecessarily. This makes the list scroll smoothly, even when displaying a large amount of data. Additionally, by minimizing calls to `findViewById`, we reduce the workload on the UI thread, improving the app's performance.

Expected Outcome: The RecyclerView should display a list of movie titles and scroll smoothly without noticeable delays or stuttering. This is especially important as the list grows, as RecyclerView should handle large data sets efficiently.



```
1 package kz.timka.Assignment3
2
3
4 import android.view.LayoutInflater
5 import android.view.View
6 import android.view.ViewGroup
7 import android.widget.TextView
8 import androidx.recyclerview.widget.RecyclerView
9
10 data class Movie(val title: String)
11
12 class MovieAdapter(
13     private val movieList: List<Movie>,
14     private val listener: OnItemClickListener
15 ) : RecyclerView.Adapter<MovieAdapter.MovieViewHolder>() {
16
17     interface OnItemClickListener {
18         fun onItemClick(movie: Movie)
19     }
20
21     class MovieViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
22         val movieTitle: TextView = itemView.findViewById(R.id.textViewMovieTitle)
23     }
24
25     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MovieViewHolder {
26         val view = LayoutInflater.from(parent.context).inflate(R.layout.item_movie, parent, attachToRoot: false)
27         return MovieViewHolder(view)
28     }
29
30     override fun onBindViewHolder(holder: MovieViewHolder, position: Int) {
31         val movie = movieList[position]
32         holder.movieTitle.text = movie.title
33         holder.itemView.setOnClickListener { it View!
34             listener.onItemClick(movie)
35         }
36     }
37
38     override fun getItemCount() = movieList.size
39 }
40
```

Implementing ViewModel

Objective: To use a ViewModel for storing a list of items and observe LiveData in an Activity or Fragment to update the UI when data changes. This helps manage UI-related data in a way that survives configuration changes, such as screen rotations.

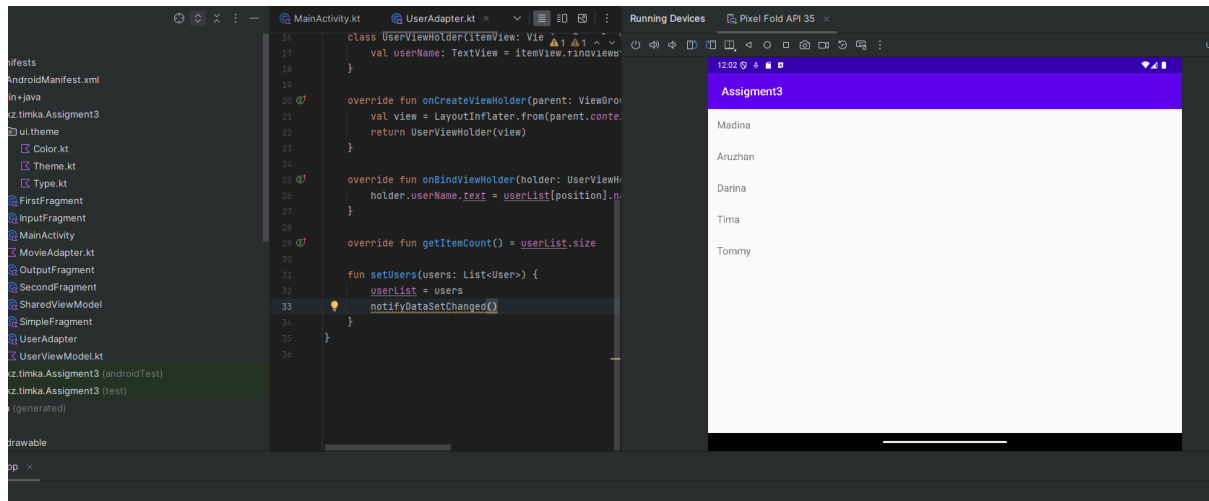
Description of Implementation: In this exercise, we created a ViewModel to store a list of users and update the UI whenever this data changes. By using ViewModel along with LiveData, we separated the data management from the Activity, which improves code organization and prevents data loss during configuration changes.

Steps taken:

1. **Creating the ViewModel Class:** We created a UserViewModel class that extends ViewModel. Inside this class, we defined a LiveData variable to hold a list of User objects. LiveData was used because it allows the UI to observe changes and update automatically.
2. **Initializing Data in the ViewModel:** In UserViewModel, we initialized the list of users and populated it with sample data. The LiveData object encapsulates this list, allowing it to be observed by any activity or fragment that needs to display the data.
3. **Observing LiveData in MainActivity:** In MainActivity, we obtained an instance of UserViewModel using the viewModels delegate. Then, we set up an observer on the users LiveData, which triggers whenever the list of users changes. When triggered, this observer updates the RecyclerView to display the current list of users.
4. **Displaying the List in RecyclerView:** We connected the UserViewModel to the RecyclerView in MainActivity, so any changes to the users list in the ViewModel automatically refresh the RecyclerView. This ensures that the UI stays in sync with the data in the ViewModel without any direct dependency on the data source.

Expected Outcome:

The RecyclerView should display the list of users and automatically refresh when the users list in the ViewModel changes. The data in ViewModel should also persist through screen rotations and other configuration changes, ensuring a stable user experience.



MutableLiveData for Input Handling

Objective: To handle user input through MutableLiveData in ViewModel, allowing real-time updates of the UI when the user enters or changes data. This exercise demonstrates how to use MutableLiveData for responsive input handling in an Android application.

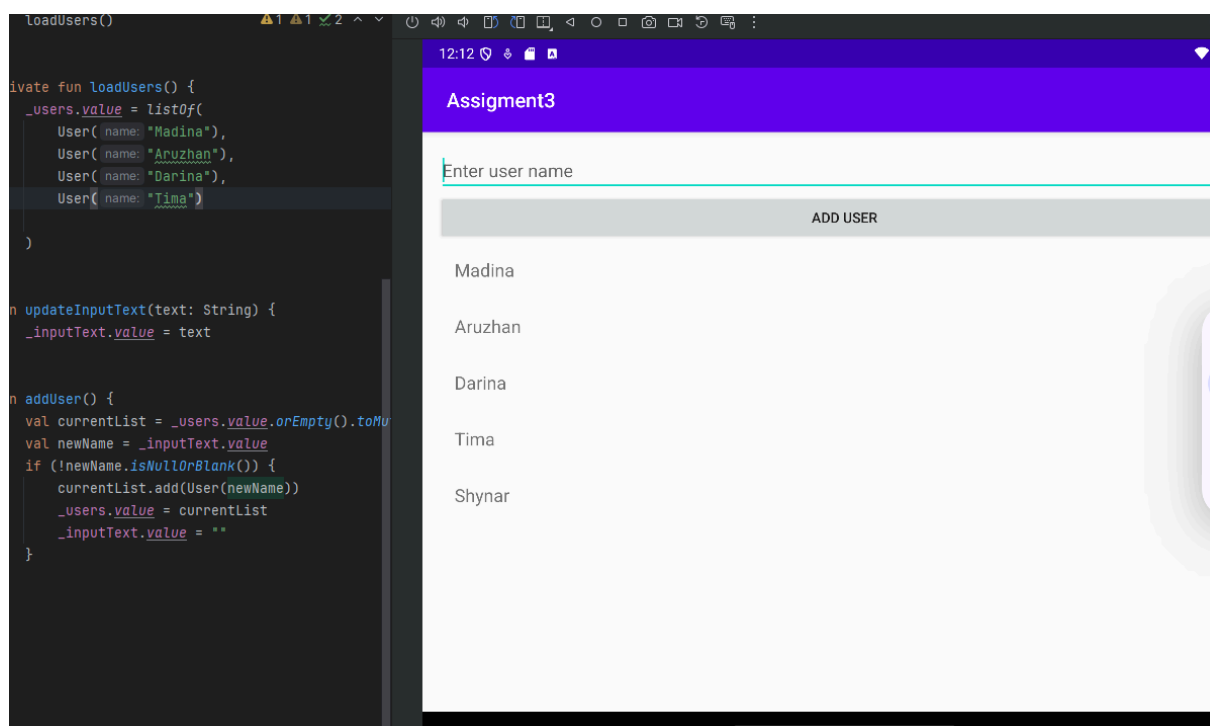
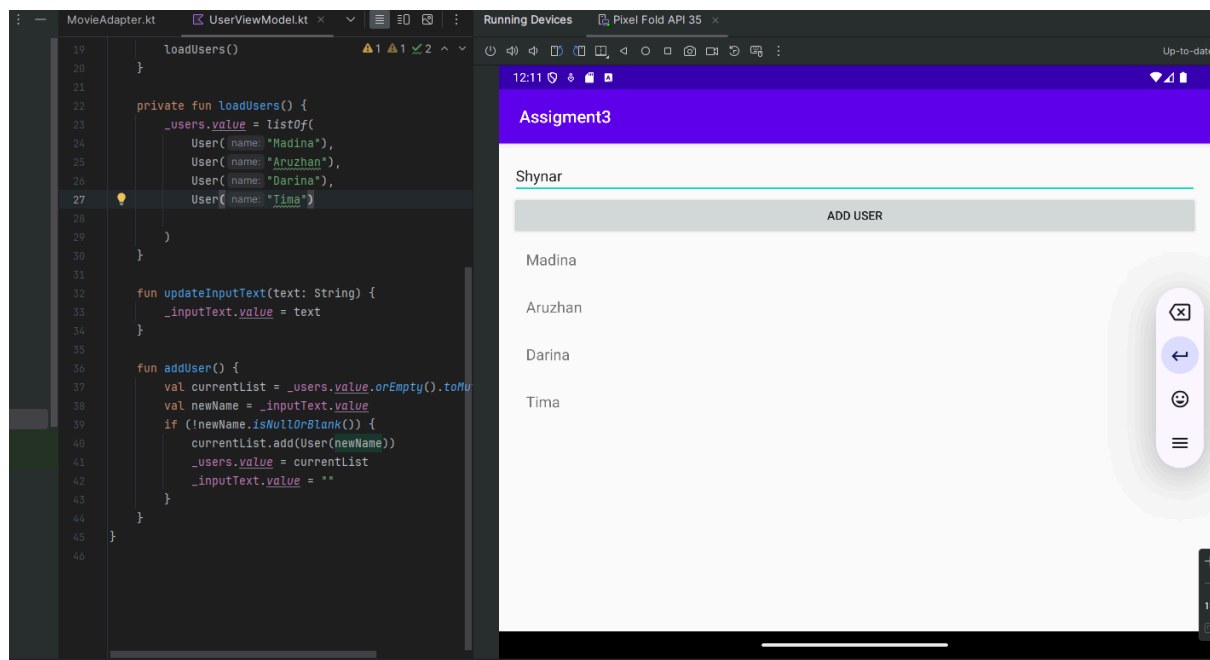
Description of Implementation: In this exercise, we extended our existing UserViewModel to support user input using MutableLiveData. We set up an input field (EditText) in MainActivity to take user input and update the ViewModel with each change. By observing MutableLiveData, the UI can respond immediately to input changes.

Steps taken:

1. **Adding MutableLiveData for Input in ViewModel:** In UserViewModel, we created a MutableLiveData property called inputText to store the user's current input. MutableLiveData allows both setting and observing values, making it ideal for handling live user input.
2. **Setting Up EditText for Real-Time Updates:** In MainActivity, we added an EditText field to take user input. We used a TextWatcher to listen for changes in the EditText field. Whenever the user types something, the TextWatcher calls a method in UserViewModel to update inputText in MutableLiveData.
3. **Observing MutableLiveData in Activity:** We set up an observer on inputText in MainActivity. Whenever inputText changes, this observer is triggered, and the updated text is displayed in another part of the UI (e.g., in a TextView or by adding it to a list of users in RecyclerView). This creates a responsive input experience where changes in the EditText field immediately reflect in the UI.

4. Updating the UI with New Data: To demonstrate how the data is dynamically updated, we also added a button that, when pressed, takes the current value in `inputText` and adds it to the list of users in the `ViewModel`. This value is then observed by the `RecyclerView`, which automatically updates to show the new user in the list.

Expected Outcome: When the user types in the `EditText` field, the input should be immediately reflected in the observed part of the UI. Pressing the button should add the input to the list of users in the `RecyclerView`, demonstrating that the input is being handled and stored effectively.



Data Persistence

Objective: To implement data persistence in an Android application by integrating Room, a local database solution. This exercise demonstrates how to store data in a local database and observe it with LiveData to ensure the UI reflects any changes made to the data in real-time.

Description of Implementation: In this exercise, we set up Room to store a list of users in a local database, enabling persistent data storage that remains available even if the app is closed or the device restarts. By using LiveData with Room, we created an architecture where changes in the database are automatically observed by the UI.

Steps taken:

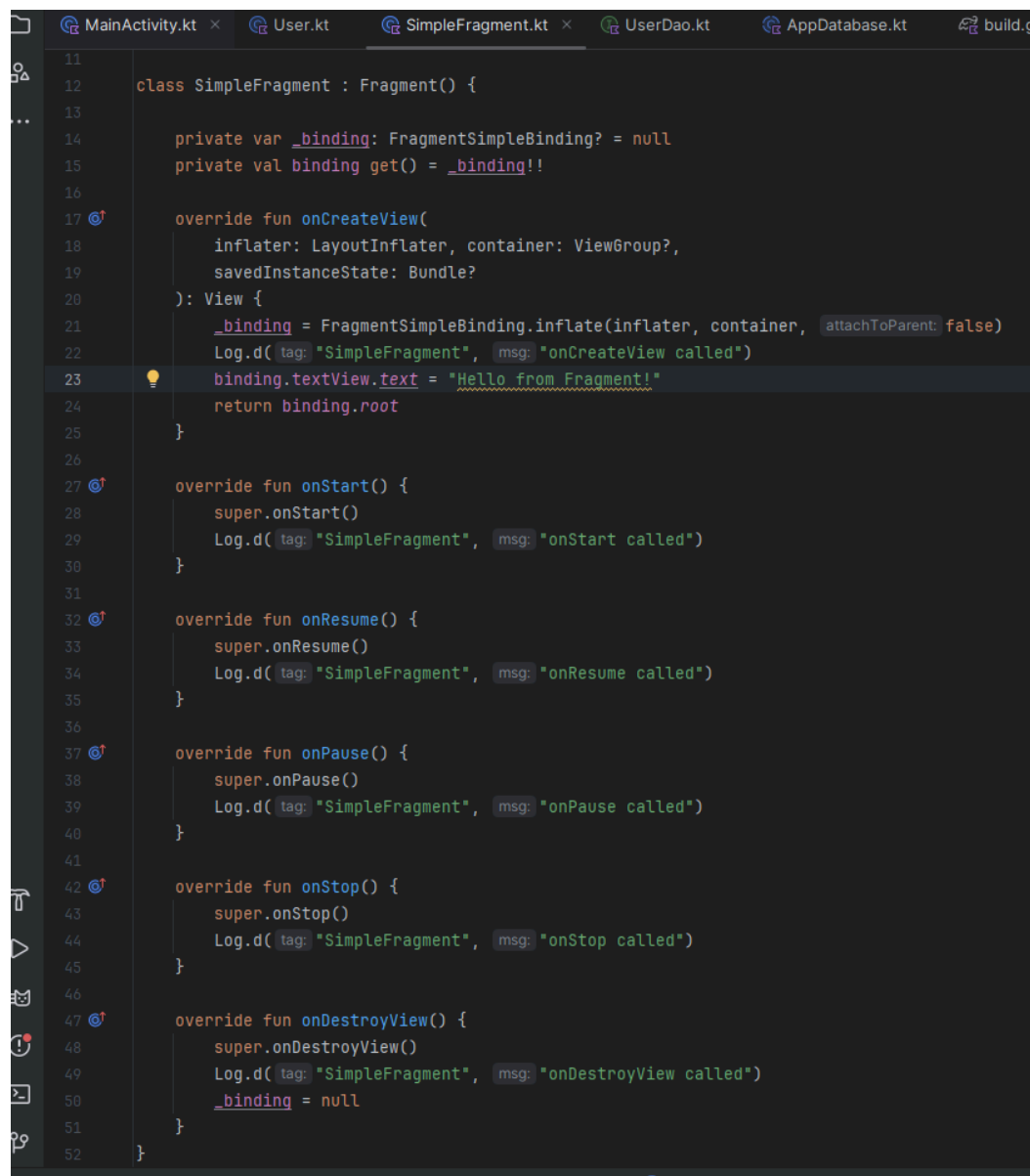
1. **Setting Up the Entity:** We started by defining a User entity with annotations from Room. This entity represents a table in the Room database, where each instance of User is a row in the table. The User entity includes an auto-generated primary key (`'id'`) and a field for the user's name.
2. **Creating the DAO (Data Access Object):** We created a UserDao interface to define methods for accessing the User table. This includes methods for inserting new users and querying the list of all users. Room generates the SQL code for these methods automatically. We used suspend functions for insert operations, which allows them to be run asynchronously in a coroutine.
3. **Setting Up the Database:** We defined an AppDatabase class that extends RoomDatabase. This class provides access to UserDao and is used to initialize the Room database. The database is created as a singleton using Room's `'databaseBuilder'` method, ensuring that only one instance of the database is used throughout the app.
4. **Using ViewModel to Interact with the Database:** In UserViewModel, we used the UserDao to retrieve all users from the database. The list of users is wrapped in LiveData, so any changes in the database are automatically reflected in the UI. We also added a method for adding a new user, which calls the insert method in UserDao within a coroutine.
5. **Observing Database Changes in the Activity:** In MainActivity, we observed the LiveData list of users from UserViewModel. Whenever a new user was added, the LiveData object notified the observer, triggering the RecyclerView to refresh and display the updated list.

Expected Outcome: The app should display a list of users in RecyclerView, with data stored in Room. Adding a new user through the UI should insert that user into the database, and the UI should automatically update to display the new user, demonstrating real-time data persistence.

Code Snippets

1) We implemented and tracked the lifecycle methods of a fragment in **MainFragment**. When the fragment's view is created, the **onCreateView** method is called, and we log this event to track it. As the fragment becomes visible, **onStart** is called, followed by **onResume**, which indicates that the fragment is fully interactive for the user.

When the fragment starts to go out of view, **onPause** is triggered, which can be useful for pausing certain actions (like stopping animations or video playback). Once the fragment is completely invisible, **onStop** is called. Finally, when the fragment's view is destroyed, **onDestroyView** logs the event, indicating that resources associated with the fragment can be released. This setup gives a complete overview of the fragment lifecycle, showing exactly when each method is triggered.

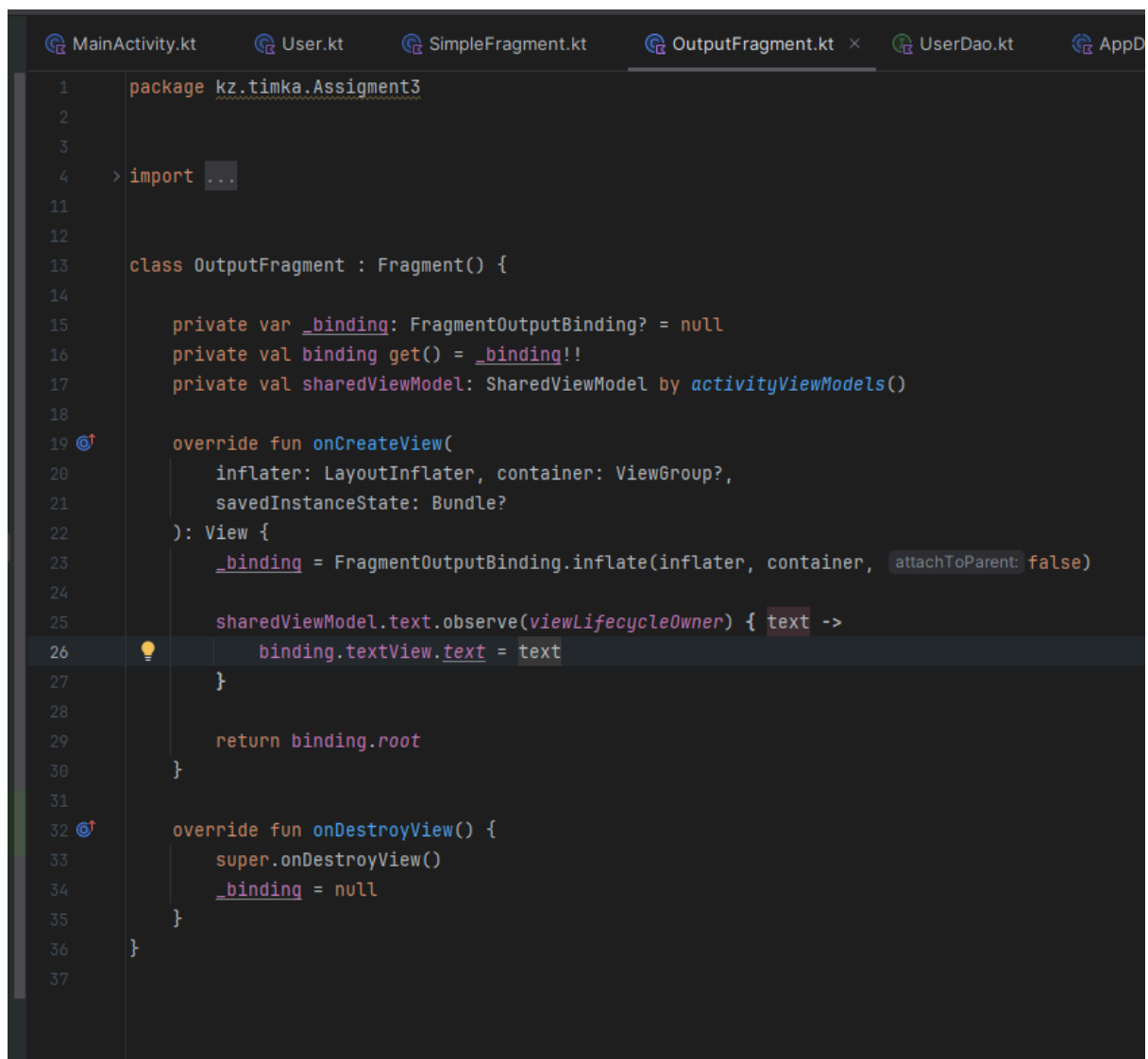
A screenshot of an IDE window showing the code for SimpleFragment.kt. The code implements the lifecycle methods of an Android Fragment. The methods include onCreateView, onStart, onResume, onPause, onStop, and onDestroyView, each with logging statements to track when they are called. The onCreateView method inflates a binding and sets the text of a TextView to "Hello from Fragment!". The onDestroyView method sets the binding to null.

```
11
12 class SimpleFragment : Fragment() {
13
14     private var _binding: FragmentSimpleBinding? = null
15     private val binding get() = _binding!!
16
17     override fun onCreateView(
18         inflater: LayoutInflater, container: ViewGroup?,
19         savedInstanceState: Bundle?
20     ): View {
21         _binding = FragmentSimpleBinding.inflate(inflater, container, attachToParent: false)
22         Log.d(tag: "SimpleFragment", msg: "onCreateView called")
23         binding.textView.text = "Hello from Fragment!"
24         return binding.root
25     }
26
27     override fun onStart() {
28         super.onStart()
29         Log.d(tag: "SimpleFragment", msg: "onStart called")
30     }
31
32     override fun onResume() {
33         super.onResume()
34         Log.d(tag: "SimpleFragment", msg: "onResume called")
35     }
36
37     override fun onPause() {
38         super.onPause()
39         Log.d(tag: "SimpleFragment", msg: "onPause called")
40     }
41
42     override fun onStop() {
43         super.onStop()
44         Log.d(tag: "SimpleFragment", msg: "onStop called")
45     }
46
47     override fun onDestroyView() {
48         super.onDestroyView()
49         Log.d(tag: "SimpleFragment", msg: "onDestroyView called")
50         _binding = null
51     }
52 }
```

2) For fragment communication, we created a shared `ViewModel` called `SharedViewModel` to facilitate data sharing between two fragments: `InputFragment` and `OutputFragment`.

In `SharedViewModel`, we defined a `MutableLiveData` object called `inputText` to store text data. The **Input Fragment** contains an `EditText` field where the user can type text. Each time the user enters or changes text, the `setText` method in `SharedViewModel` is called to update `inputText`.

The **Output Fragment** observes `inputText` in `SharedViewModel`. Whenever the `inputText` value changes (i.e., when the user types something new in the Input Fragment), the Output Fragment receives this update and displays the new text. This approach enables seamless communication between the fragments, with changes in one fragment immediately reflected in the other.



```
1 package kz.timka.Assignment3
2
3
4 > import ...
5
6
7
8
9
10
11
12
13 class OutputFragment : Fragment() {
14
15     private var _binding: FragmentOutputBinding? = null
16     private val binding get() = _binding!!
17     private val sharedViewModel: SharedViewModel by activityViewModels()
18
19     override fun onCreateView(
20         inflater: LayoutInflater, container: ViewGroup?,
21         savedInstanceState: Bundle?
22     ): View {
23         _binding = FragmentOutputBinding.inflate(inflater, container, attachToParent: false)
24
25         sharedViewModel.text.observe(viewLifecycleOwner) { text ->
26             binding.textView.text = text
27         }
28
29         return binding.root
30     }
31
32     override fun onDestroyView() {
33         super.onDestroyView()
34         _binding = null
35     }
36 }
37
```



```
Pixel Fold API 35 app
MainActivity.kt User.kt SimpleFragment.kt OutputFragment.kt InputFragment.kt UserDao.kt

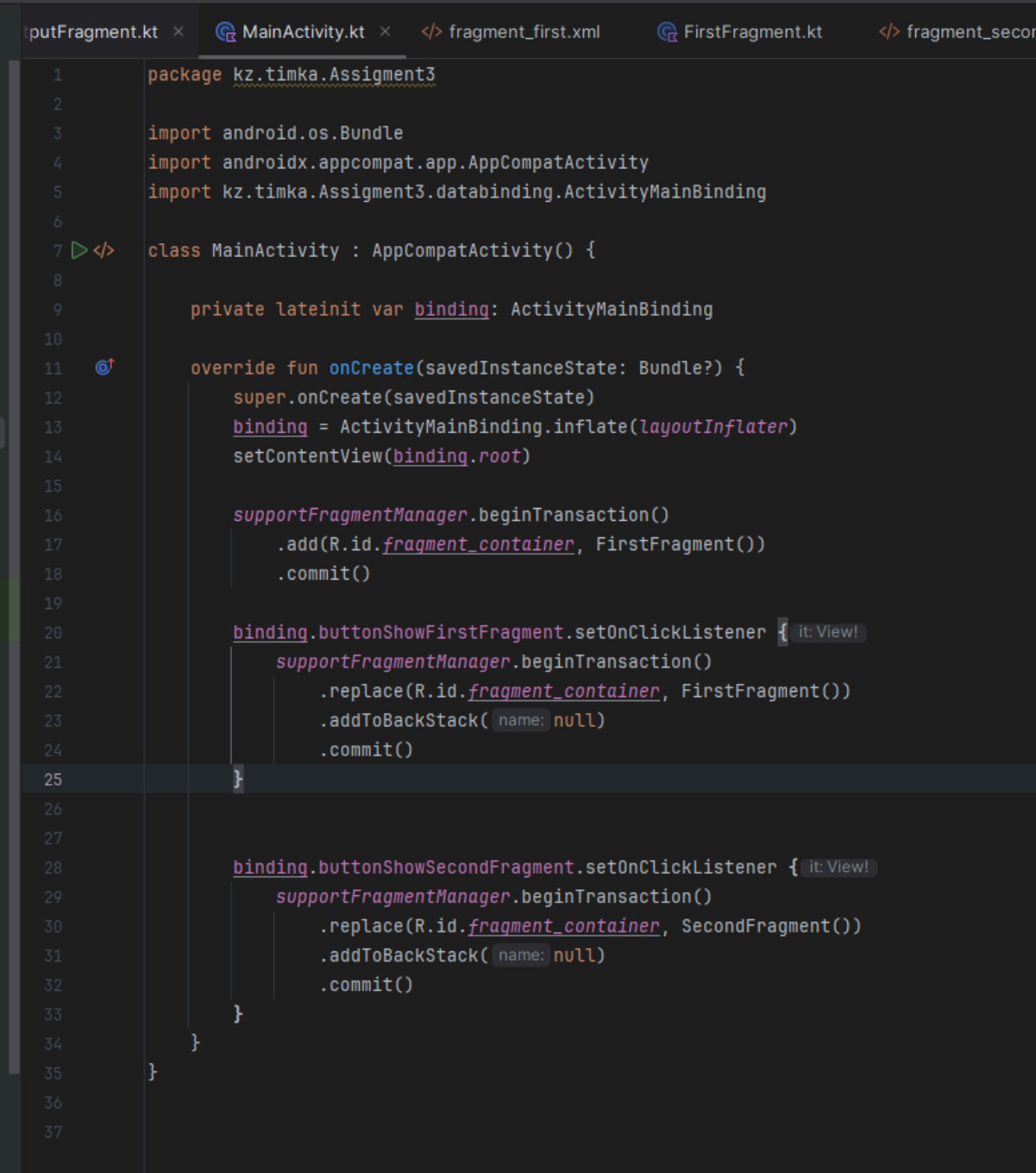
1 package kz.timka.Assignment3
2
3 > import ...
12
13
14 class InputFragment : Fragment() {
15
16     private var _binding: FragmentInputBinding? = null
17     private val binding get() = _binding!!
18     private val sharedViewModel: SharedViewModel by activityViewModels()
19
20     override fun onCreateView(
21         inflater: LayoutInflater, container: ViewGroup?,
22         savedInstanceState: Bundle?
23     ): View {
24         _binding = FragmentInputBinding.inflate(inflater, container, attachToParent: false)
25
26         binding.editText.addTextChangedListener(object : TextWatcher {
27             override fun afterTextChanged(s: Editable?) {
28                 sharedViewModel.setText(s.toString())
29             }
30
31             override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {}
32             override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {}
33         })
34
35         return binding.root
36     }
37
38     override fun onDestroyView() {
39         super.onDestroyView()
40         _binding = null
41     }
42 }
43
44
```

3) To manage multiple fragments dynamically, we implemented fragment transactions in an activity. We created two fragments and used buttons in the activity to control which fragment is displayed.

When the user presses the "Add Fragment" button, we use the **add** method to insert a fragment into a container view in the activity. The "Replace Fragment" button allows us to switch the displayed fragment by calling the **replace** method, which removes the current fragment from the container and adds a new one in its place. Finally, the "Remove Fragment" button uses the **remove** method to delete the fragment from the container, leaving it empty.

To enable back navigation between fragments, we included **addToBackStack** in each transaction, so users can navigate back to previous fragments using the back button. This

setup allows for dynamic management of fragments, enabling the app to respond to user interactions by showing or hiding different fragments.



```
1 package kz.timka.Assignment3
2
3 import android.os.Bundle
4 import androidx.appcompat.app.AppCompatActivity
5 import kz.timka.Assignment3.databinding.ActivityMainBinding
6
7 <?xml>
8 class MainActivity : AppCompatActivity() {
9
10     private lateinit var binding: ActivityMainBinding
11
12     override fun onCreate(savedInstanceState: Bundle?) {
13         super.onCreate(savedInstanceState)
14         binding = ActivityMainBinding.inflate(layoutInflater)
15         setContentView(binding.root)
16
17         supportFragmentManager.beginTransaction()
18             .add(R.id.fragment_container, FirstFragment())
19             .commit()
20
21         binding.buttonShowFirstFragment.setOnClickListener { it: View!
22             supportFragmentManager.beginTransaction()
23                 .replace(R.id.fragment_container, FirstFragment())
24                 .addToBackStack(name: null)
25                 .commit()
26
27         binding.buttonShowSecondFragment.setOnClickListener { it: View!
28             supportFragmentManager.beginTransaction()
29                 .replace(R.id.fragment_container, SecondFragment())
30                 .addToBackStack(name: null)
31                 .commit()
32
33     }
34 }
35
36
37
```

```

1 package R2.Limka.Assignments
2
3
4 > import ...
5
6
7
8
9
10
11 class FirstFragment : Fragment() {
12
13     private var _binding: FragmentFirstBinding? = null
14     private val binding get() = _binding!!
15
16     override fun onCreateView(
17         inflater: LayoutInflater, container: ViewGroup?,
18         savedInstanceState: Bundle?
19     ): View {
20         _binding = FragmentFirstBinding.inflate(inflater, container, attachToParent: false)
21         binding.textView.text = "This is the First Fragment"
22         return binding.root
23     }
24
25     override fun onDestroyView() {
26         super.onDestroyView()
27         _binding = null
28     }
29 }
30

```

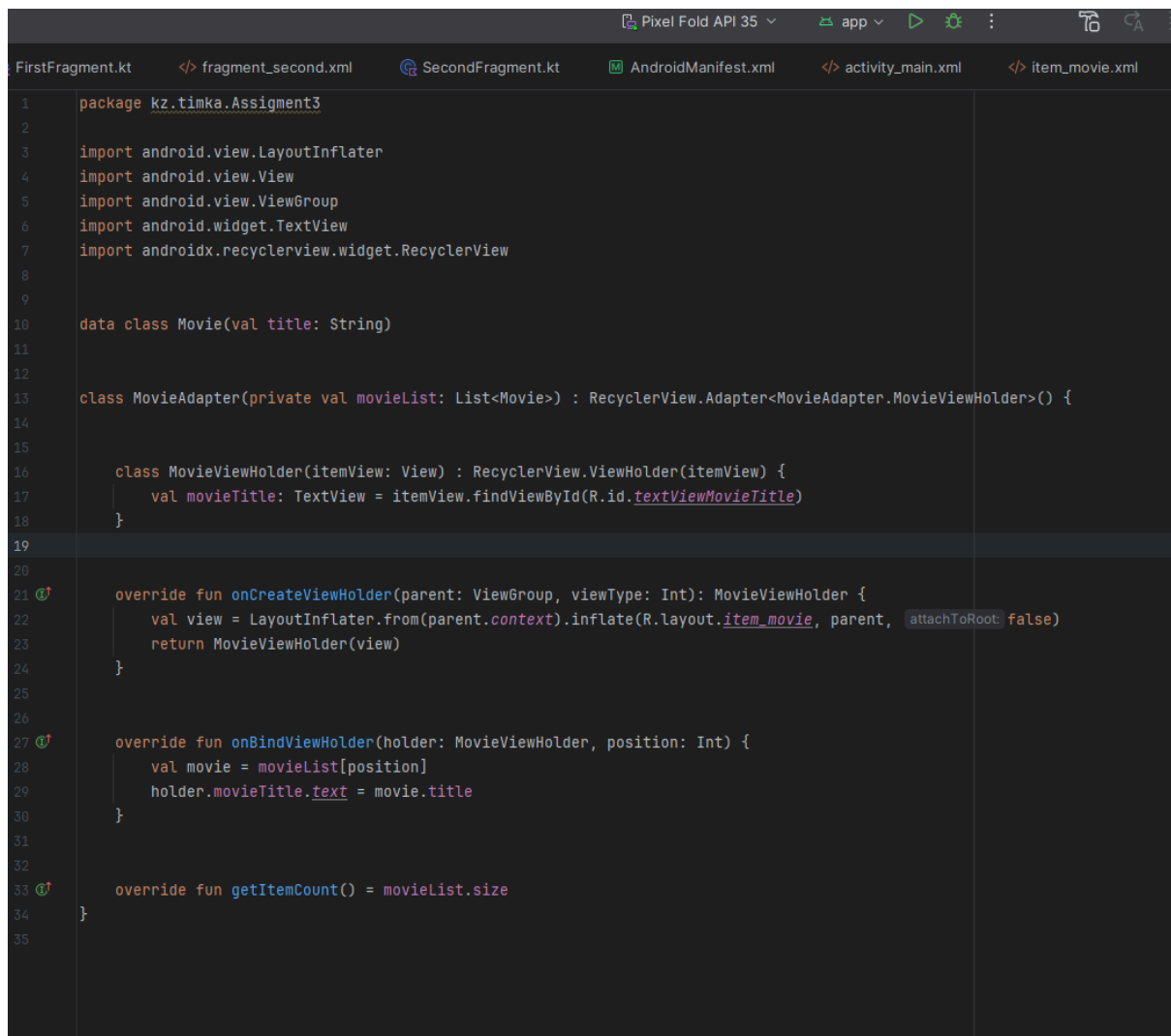
```

3
4 > import ...
5
6
7
8
9
10
11
12 class SecondFragment : Fragment() {
13
14     private var _binding: FragmentSecondBinding? = null
15     private val binding get() = _binding!!
16
17     override fun onCreateView(
18         inflater: LayoutInflater, container: ViewGroup?,
19         savedInstanceState: Bundle?
20     ): View {
21         _binding = FragmentSecondBinding.inflate(inflater, container, attachToParent: false)
22         binding.textView.text = "This is the Second Fragment"
23         return binding.root
24     }
25
26     override fun onDestroyView() {
27         super.onDestroyView()
28         _binding = null
29     }
30 }
31
32
33
34
35
36
37
38
39
40
41
42

```

4) we built a `RecyclerView` to display a list of favorite movies. First, we created an XML layout file, `item_movie.xml`, which includes a `TextView` to show each movie title. In `MainActivity`, we added a `RecyclerView` widget to hold this list.

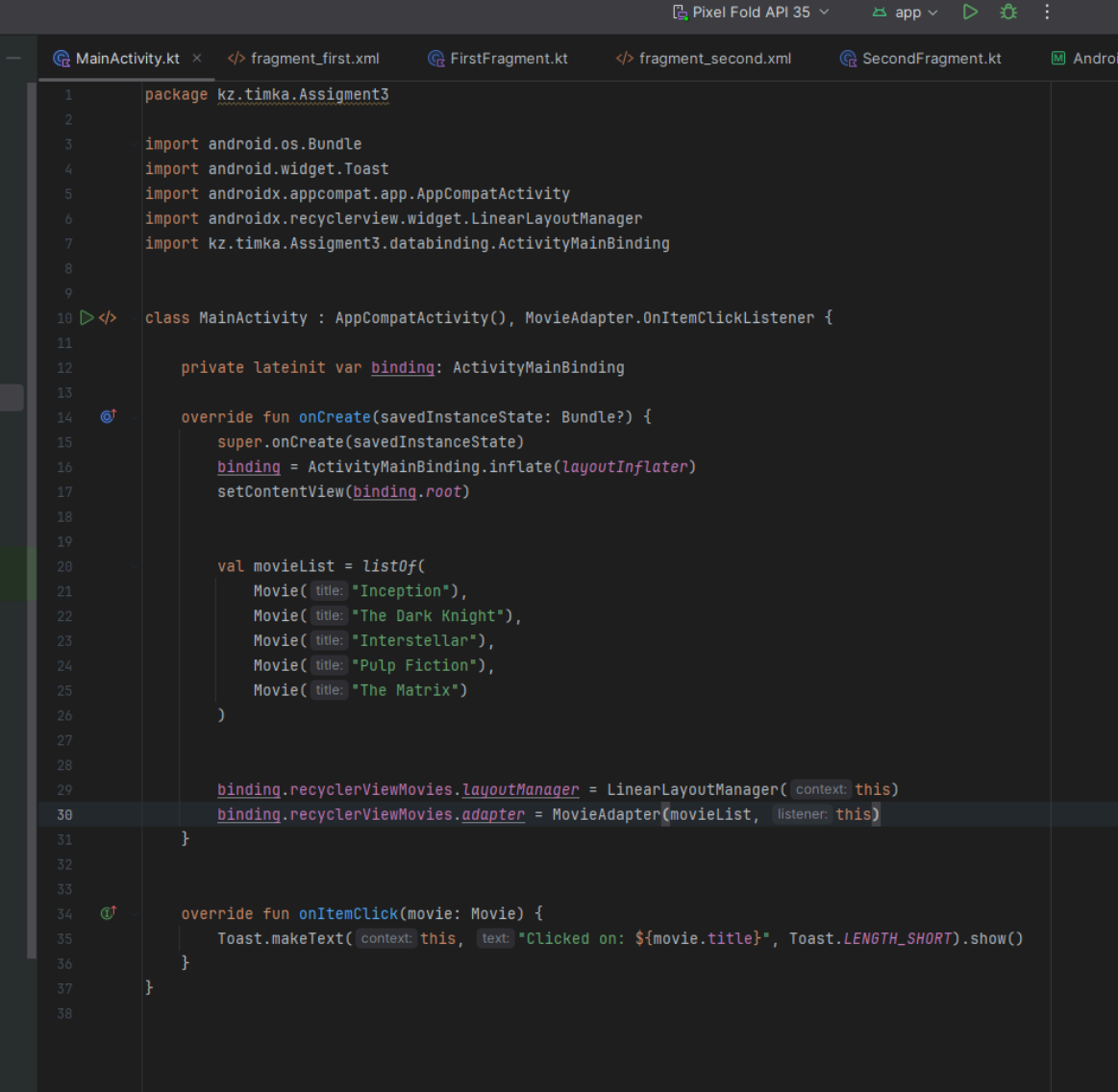
We created a `MovieAdapter` class, which extends `RecyclerView.Adapter`, to manage the data and bind each item to the `RecyclerView`. In the adapter, we defined a `MovieViewHolder` class to hold references to the `TextView` in each item. In `MainActivity`, we initialized the adapter with a list of movie titles and set it on the `RecyclerView`. This setup displays the list of movies efficiently, with each item showing a title.

A screenshot of the Android Studio IDE showing the implementation of a RecyclerView adapter. The top toolbar shows the 'Pixel Fold API 35' target, a dropdown for 'app', and icons for running, debugging, and testing. The bottom toolbar shows icons for switching between code, layout, and preview views. The file explorer at the top lists several files: 'FirstFragment.kt', 'fragment_second.xml', 'SecondFragment.kt', 'AndroidManifest.xml', 'activity_main.xml', and 'item_movie.xml'. The main editor area displays the code for 'MovieAdapter.kt'. The code starts with package and import statements for Android and RecyclerView. It defines a data class 'Movie' with a 'title' property. Then, it defines the 'MovieAdapter' class, which extends 'RecyclerView.Adapter' and implements 'onCreateViewHolder', 'onBindViewHolder', and 'getItemCount' methods. The 'onCreateViewHolder' method inflates the 'item_movie' layout and returns a 'MovieViewHolder'. The 'onBindViewHolder' method binds the movie data to the 'movieTitle' TextView in the holder. The 'getItemCount' method returns the size of the movie list. The code is as follows:

```
1 package kz.timka.Assignment3
2
3 import android.view.LayoutInflater
4 import android.view.View
5 import android.view.ViewGroup
6 import android.widget.TextView
7 import androidx.recyclerview.widget.RecyclerView
8
9
10 data class Movie(val title: String)
11
12
13 class MovieAdapter(private val movieList: List<Movie>) : RecyclerView.Adapter<MovieAdapter.MovieViewHolder>() {
14
15
16     class MovieViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
17         val movieTitle: TextView = itemView.findViewById(R.id.text_view_movie_title)
18     }
19
20
21     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MovieViewHolder {
22         val view = LayoutInflater.from(parent.context).inflate(R.layout.item_movie, parent, attachToRoot: false)
23         return MovieViewHolder(view)
24     }
25
26
27     override fun onBindViewHolder(holder: MovieViewHolder, position: Int) {
28         val movie = movieList[position]
29         holder.movieTitle.text = movie.title
30     }
31
32
33     override fun getItemCount() = movieList.size
34 }
35
```

5) To enable click handling for each item in the `RecyclerView`, we created an `OnItemClickListener` interface within `MovieAdapter`. This interface defines a method, `onItemClick`, which passes the clicked `Movie` object back to `MainActivity`.

In **MainActivity**, we implemented **OnItemClickListener** to handle item clicks. For each click, a **Toast** message displays the title of the clicked movie. Inside **MovieAdapter**, we set up the click listener in **onBindViewHolder**, so whenever an item is clicked, **MainActivity** receives the clicked item data. This setup makes each movie item responsive to clicks, displaying a message with its title whenever clicked.



```
1 package kz.timka.Assignment3
2
3 import android.os.Bundle
4 import android.widget.Toast
5 import androidx.appcompat.app.AppCompatActivity
6 import androidx.recyclerview.widget.LinearLayoutManager
7 import kz.timka.Assignment3.databinding.ActivityMainBinding
8
9
10 class MainActivity : AppCompatActivity(), MovieAdapter.OnItemClickListener {
11
12     private lateinit var binding: ActivityMainBinding
13
14     override fun onCreate(savedInstanceState: Bundle?) {
15         super.onCreate(savedInstanceState)
16         binding = ActivityMainBinding.inflate(layoutInflater)
17         setContentView(binding.root)
18
19
20         val movieList = listOf(
21             Movie(title: "Inception"),
22             Movie(title: "The Dark Knight"),
23             Movie(title: "Interstellar"),
24             Movie(title: "Pulp Fiction"),
25             Movie(title: "The Matrix")
26         )
27
28
29         binding.recyclerViewMovies.layoutManager = LinearLayoutManager(context = this)
30         binding.recyclerViewMovies.adapter = MovieAdapter(movieList, listener = this)
31     }
32
33
34     override fun onItemClick(movie: Movie) {
35         Toast.makeText(context = this, text = "Clicked on: ${movie.title}", Toast.LENGTH_SHORT).show()
36     }
37 }
38
```

```

1 package kz.timka.Assignments
2
3
4 > import ...
5
6
7
8
9
10 data class Movie(val title: String)
11
12 class MovieAdapter(
13     private val movieList: List<Movie>,
14     private val listener: OnItemClickListener
15 ) : RecyclerView.Adapter<MovieAdapter.MovieViewHolder>() {
16
17     interface OnItemClickListener {
18         fun onItemClick(movie: Movie)
19     }
20
21     class MovieViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
22         val movieTitle: TextView = itemView.findViewById(R.id.textViewMovieTitle)
23     }
24
25     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MovieViewHolder {
26         val view = LayoutInflater.from(parent.context).inflate(R.layout.item_movie, parent, attachToRoot: false)
27         return MovieViewHolder(view)
28     }
29
30     override fun onBindViewHolder(holder: MovieViewHolder, position: Int) {
31         val movie = movieList[position]
32         holder.movieTitle.text = movie.title
33         holder.itemView.setOnClickListener { it: View?
34             listener.onItemClick(movie)
35         }
36     }
37
38     override fun getItemCount() = movieList.size
39 }
40

```

6) In **MovieAdapter**, we used the **ViewHolder** pattern to optimize the **RecyclerView**. Inside **MovieAdapter**, we defined a **MovieViewHolder** class, which holds references to the views in each item layout, such as **TextView** for the movie title. This approach eliminates the need to call **findViewById** repeatedly, which improves performance by reducing view lookups.

The **MovieViewHolder** class is used in **onCreateViewHolder** to create and reuse item views, which allows **RecyclerView** to recycle views as they scroll off-screen. This setup minimizes memory usage and improves scrolling smoothness, making it ideal for lists with many items.

```
FirstFragment.kt    </> fragment_second.xml    SecondFragment.kt    AndroidManifest.xml    </> activity_main.xml    </> item_movie.xml

1  package kz.timka.Assignment3
2
3
4  import android.view.LayoutInflater
5  import android.view.View
6  import android.view.ViewGroup
7  import android.widget.TextView
8  import androidx.recyclerview.widget.RecyclerView
9
10 data class Movie(val title: String)
11
12 class MovieAdapter(
13     private val movieList: List<Movie>,
14     private val listener: OnItemClickListener
15 ) : RecyclerView.Adapter<MovieAdapter.MovieViewHolder>() {
16
17     interface OnItemClickListener {
18         fun onItemClick(movie: Movie)
19     }
20
21     class MovieViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
22         val movieTitle: TextView = itemView.findViewById(R.id.textViewMovieTitle)
23     }
24
25     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MovieViewHolder {
26         val view = LayoutInflater.from(parent.context).inflate(R.layout.item_movie, parent, attachToRoot: false)
27         return MovieViewHolder(view)
28     }
29
30     override fun onBindViewHolder(holder: MovieViewHolder, position: Int) {
31         val movie = movieList[position]
32         holder.movieTitle.text = movie.title
33         holder.itemView.setOnClickListener { it: View!
34             listener.onItemClick(movie)
35         }
36     }
37
38     override fun getItemCount() = movieList.size
39 }
40
```

7) we implemented **UserViewModel** to store and manage a list of users.

UserViewModel includes a **LiveData** list of **User** objects, which holds the data to be displayed in the **RecyclerView**.

In **MainActivity**, we obtained an instance of **UserViewModel** and set up an observer for the **users** LiveData list. Whenever the data in **UserViewModel** changes, the observer updates the **RecyclerView** with the new list of users. This setup keeps the UI in sync with the data, and **ViewModel** ensures that data is preserved across configuration changes.

```
MainActivity.kt x UserAdapter.kt x item_user.xml fragment_first.xml fragment_second.xml activity_main.xml
1 package kz.timka.Assignment3
2
3
4
5 import android.view.LayoutInflater
6 import android.view.View
7 import android.view.ViewGroup
8 import android.widget.TextView
9 import androidx.recyclerview.widget.RecyclerView
10
11 class UserAdapter : RecyclerView.Adapter<UserAdapter.UserViewHolder>() {
12
13     private var userList = emptyList<User>()
14
15
16     class UserViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
17         val userName: TextView = itemView.findViewById(R.id.textViewUserName)
18     }
19
20     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): UserViewHolder {
21         val view = LayoutInflater.from(parent.context).inflate(R.layout.item_user, parent, attachToRoot: false)
22         return UserViewHolder(view)
23     }
24
25     override fun onBindViewHolder(holder: UserViewHolder, position: Int) {
26         holder.userName.text = userList[position].name
27     }
28
29     override fun getItemCount() = userList.size
30
31     fun setUsers(users: List<User>) {
32         userList = users
33         notifyDataSetChanged()
34     }
35 }
36
```

```
item_user.xml fragment_first.xml fragment_second.xml
1 package kz.timka.Assignment3
2
3
4 import androidx.lifecycle.LiveData
5 import androidx.lifecycle.MutableLiveData
6 import androidx.lifecycle.ViewModel
7
8 data class User(val name: String)
9
10 class UserViewModel : ViewModel() {
11
12
13     private val _users = MutableLiveData<List<User>>()
14
15     val users: LiveData<List<User>> get() = _users
16
17
18     init {
19         loadUsers()
20     }
21
22
23     private fun loadUsers() {
24         _users.value = listOf(
25             User(name: "Madina"),
26             User(name: "Aruzhan"),
27             User(name: "Darina"),
28             User(name: "Tima"),
29             User(name: "Tommy")
30         )
31     }
32 }
33
```



```
MainActivity.kt x UserAdapter.kt </> item_user.xml </> fragment_first.xml </> fragment_second
1 package kz.timka.Assignment3
2
3
4 import android.os.Bundle
5 import androidx.activity.viewModels
6 import androidx.appcompat.app.AppCompatActivity
7 import androidx.recyclerview.widget.LinearLayoutManager
8 import kz.timka.Assignment3.databinding.ActivityMainBinding
9
10
11 <img alt="Run icon" data-bbox="175 258 205 270"/> class MainActivity : AppCompatActivity() {
12
13     private lateinit var binding: ActivityMainBinding
14     private val userModel: UserViewModel by viewModels()
15     private lateinit var userAdapter: UserAdapter
16
17     @<img alt="Run icon" data-bbox="190 348 205 360"/> override fun onCreate(savedInstanceState: Bundle?) {
18         super.onCreate(savedInstanceState)
19         binding = ActivityMainBinding.inflate(layoutInflater)
20         setContentView(binding.root)
21
22
23         userAdapter = UserAdapter()
24         binding.recyclerViewMovies.layoutManager = LinearLayoutManager(context = this)
25         binding.recyclerViewMovies.adapter = userAdapter
26
27
28         userModel.users.observe(owner = this) { userList ->
29             userAdapter.setUsers(userList)
30         }
31     }
32 }
33
```

8) To handle user input dynamically, we added **MutableLiveData** for input text in **UserViewModel**. This property, **inputText**, stores the current text entered by the user. In **MainActivity**, we used an **EditText** field to capture user input and a **TextWatcher** to listen for text changes.

Each time the user types, **inputText** in **UserViewModel** is updated with the new value. This value is observed in **MainActivity**, so any change to **inputText** immediately updates the UI. Additionally, we included a button to add the input as a new user in the list, demonstrating how the UI and data stay in sync with real-time input.

we implemented Room to persist user data in a local database. We defined a **User** entity to represent each entry in the database, along with a **UserDao** interface for database operations, such as inserting and retrieving users.

In **AppDatabase**, we set up Room to manage the database and provide access to **UserDao**. In **UserViewModel**, we used **UserDao** to retrieve the list of users and store it in a **LiveData** object. This setup ensures that any changes to the database are immediately reflected in the UI, as LiveData automatically triggers updates. When a new user is added, the database and RecyclerView update simultaneously, demonstrating data persistence and real-time synchronization with Room.

```
1 package kz.timka.Assignment3
2
3 > import ...
4
5
6 @Entity(tableName = "users")
7 data class User(
8     @PrimaryKey(autoGenerate = true) val id: Int = 0,
9     val name: String
10 )
```

```
1 package kz.timka.Assignment3
2
3
4 import androidx.lifecycle.LiveData
5 import androidx.room.Dao
6 import androidx.room.Insert
7 import androidx.room.OnConflictStrategy
8 import androidx.room.Query
9
10 @Dao
11 interface UserDao {
12
13     @Query("SELECT * FROM users")
14     fun getAllUsers(): LiveData<List<User>>
15
16     @Insert(onConflict = OnConflictStrategy.REPLACE)
17     suspend fun insertUser(user: User): Long
18 }
19
```

```

> import ...

@Database(entities = [User::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {

    abstract fun userDao(): UserDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(lock: this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    name: "user_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}

```

```

1 package kz.timka.Assignment3
2
3
4
5 > import ...
6
7 class UserAdapter : RecyclerView.Adapter<UserAdapter.UserViewHolder>() {
8
9     private var userList = emptyList<User>()
10
11     class UserViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
12         val userName: TextView = itemView.findViewById(R.id.textviewUserName)
13     }
14
15     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): UserViewHolder {
16         val view = LayoutInflater.from(parent.context).inflate(R.layout.item_user, parent, attachToRoot: false)
17         return UserViewHolder(view)
18     }
19
20     override fun onBindViewHolder(holder: UserViewHolder, position: Int) {
21         holder.userName.text = userList[position].name
22     }
23
24     override fun getItemCount() = userList.size
25
26     fun setUsers(users: List<User>) {
27         userList = users
28         notifyDataSetChanged()
29     }
30 }

```

Results

1. **Fragment Lifecycle:** We successfully tracked the lifecycle events of a fragment. Each lifecycle method logged an entry, letting us see when the fragment was created, became visible, paused, and was destroyed. This gave us a clear understanding of how Android manages fragments and when different actions can be performed.
2. **Fragment Communication:** We enabled communication between two fragments using a shared ViewModel. When the user typed in one fragment, the text appeared in the other fragment automatically. This made it easy to share data between fragments without directly linking them, which helps keep the code clean and modular.
3. **Fragment Transactions:** We implemented buttons to switch between two fragments using add, replace, and remove operations. The user could switch views by clicking buttons, and the back button let them return to previous fragments. This added flexibility to the UI and made navigation feel smooth.
4. **Building a RecyclerView:** We created a RecyclerView to display a list of movie titles. The RecyclerView showed each movie as a separate item and scrolled smoothly, even with many items in the list. This setup gave us a practical way to display lists in Android applications.
5. **Item Click Handling:** We added click handling to the RecyclerView, so when the user clicked on a movie title, a Toast message appeared with the movie name. This added interactivity to the list and demonstrated how to respond to user actions.
6. **ViewHolder Pattern:** By implementing the ViewHolder pattern in the RecyclerView adapter, we optimized the app's performance. The app ran smoothly and used memory efficiently, even with a long list of items, because each item view was reused as it scrolled off-screen.
7. **Implementing ViewModel:** We used a ViewModel to manage the list of users, which allowed us to keep data consistent even if the screen was rotated. The ViewModel kept the data separate from the UI, making the code more organized and preventing data loss during configuration changes.
8. **MutableLiveData for Input Handling:** We handled user input dynamically by using MutableLiveData in the ViewModel. When the user typed in an EditText field, the data immediately updated in the ViewModel and reflected in the UI. This setup created a responsive user experience.
9. **Data Persistence:** We integrated Room for data persistence, storing user data in a local database. This allowed us to save user data permanently, so it remained available even after

closing and reopening the app. With Room, we achieved a setup where the UI automatically updated with any changes in the database, providing a seamless experience.

Conclusion

Through these exercises, we gained hands-on experience with important Android components like Fragments, RecyclerView, ViewModel, and Room for data persistence. Each component served a specific purpose and worked together to create a smooth, responsive app.

We learned how to use Fragments to manage different parts of the UI and switch between them, which is essential for building flexible layouts. By using RecyclerView with the ViewHolder pattern, we were able to display large lists of data efficiently, ensuring good performance. The ViewModel and LiveData made it easy to manage data separate from the UI, keeping the app organized and preventing data loss when the screen rotated or configuration changed.

Adding Room for data persistence allowed us to store user data locally, which kept information saved even when the app was closed. This integration helped us understand how to work with databases in Android while keeping the UI updated automatically with LiveData.

Overall, these exercises showed us how to build a modern Android app that is interactive, efficient, and user-friendly. The skills we learned are essential for developing reliable apps that perform well and provide a smooth experience for users.