Chapter 3: Understanding Data Presentation on Android

In this chapter, we will study the libraries available for presenting data on the **user interface** (**UI**). We will start this chapter by analyzing the life-cycles of activities and fragments (what responsibilities they had in the past and what responsibilities they have now) with the introduction of the **`ViewModel`** and **`Lifecycle`** libraries. We will then move on to analyze aspects of how the UI works and look at how the Jetpack Compose library revolutionized building UIs through its declarative approach. Finally, we will look at how we can navigate between different screens that are built in Compose by using the **`Navigation`** library with the **`Compose`** extension.

In this chapter, we will cover the following main topics:

- Analyzing lifecycle-aware components
- Using Jetpack Compose to build UIs

By the end of the chapter, you will become familiar with how to present data on the UI using ViewModel and Compose.

Technical requirements

Hardware and software requirements are as follows:

- Android Studio Arctic Fox 2020.3.1 Patch 3

The code files for this chapter can be found here: https://github.com/PacktPublishing/Clean-Android-Architecture/tree/main/Chapter3 ↗️.

Check out the following video to see the Code in Action:
https://bit.ly/3lmMIOg ↗️

Analyzing lifecycle-aware components

In this section, we will analyze the lifecycles of activities and fragments and the potential issues that are caused when working with them. We will also observe how the introduction of ViewModel and LiveData solves these problems.

When the Android operating system and its development framework were released, activities were the most commonly used components when developing an application, as they represent the entry point of the interaction between an application and a user. As technology in displays and resolutions improved, apps could then present more information and controls that the user could interact with. For developers, this meant that the code required to manage the logic for a single activity increased, especially when dealing with different layouts for landscape and portrait. The introduction of fragments was meant to solve some of these problems. Responsibilities for handling the logic in different parts of the screen could now be divided into different fragments.

The introduction of fragments, however, didn't solve all of the issues developers were dealing with, mainly because both activities and fragments have their own lifecycles. Dealing with lifecycles created the possibility of apps having context leaks, and the combination of lifecycles and inheritance made both activities and fragments hard to unit test.

The lifecycle of an activity is as follows:

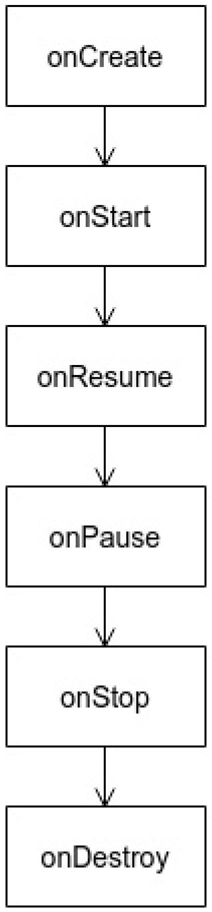


Figure 3.1 – Activity lifecycle

In *Figure 3.1*, we can see the six most well-known states of an activity:

- *CREATED*: The activity enters this state when the **`onCreate`** method is called. This will be called when the system creates the activity.
- *STARTED*: The activity enters this state when the **`onStart`** method is called. This will be called when the activity is visible to the user.
- *RESUMED*: The activity enters this state when the **`onResume`** method is called. This will be called when the activity is in focus (the user can interact with it).

The next three states are called when the activity is no longer in focus. This can be caused either by the user closing the activity, putting it in the background, or another component gaining focus:

- *PAUSED*: The activity enters this state when the **`onPause`** method is called. This will be called when the activity is visible but no longer in focus.

- *STOPPED*: The activity enters this state when the **`onStop`** method is called. This will be called when the activity is no longer visible.
- *DESTROYED*: The activity enters this state when the **`onDestroy`** method is called. This will be called when the activity is destroyed by the operating system.

When we use activities in our code, dealing with the lifecycle will look something like this:

```
class MyActivity : Activity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
    }
    override fun onStart() {
        super.onStart()
    }
    override fun onResume() {
        super.onResume()
    }
    override fun onPause() {
        super.onPause()
    }
    override fun onStop() {
        super.onStop()
    }
    override fun onDestroy() {
        super.onDestroy()
    }
}
```

We can see here that we need to extend the **`Activity`** class and, if we want to execute a particular operation in a particular state, we can override the method associated with the state and invoke the **`super`** call. This represents the main reason why activities are hard to unit test. The **`super`** calls would cause our test not only to invoke our code but also the parent class's code. Another reason activities are hard to test is because the system is the one instantiating the class, which means that we cannot use the constructor of the class for injection and must rely on setters to inject mock objects.

An important distinction should be made between the *DESTROYED* state and garbage collection. A *DESTROYED* activity doesn't mean it will be garbage collected. A simple definition of what garbage collection means is that garbage collection is the process of deallocating memory that is no longer used. Each created object takes a certain amount of memory. When the garbage collector wants to free memory, it will look at objects that are no longer referenced by other objects. If we want to make sure that objects will be garbage collected, we will need to make sure that other objects that live longer than them will have no reference to the objects we want to be collected. In Android, we want **context** (such as activity and service) objects, or other objects with lifecycles, to be collected when their `onDestroy` methods are called. This is because they tend to occupy a lot of memory and we will end up with crashes or bugs if we end up invoking methods after `onDestroy` is called. Leaks that prevent context objects from being collected are called **context leaks**. Let's look at a simple example of this:

```
interface MyListener {
    fun onChange(newText: String)
}
object MyManager {
    private val listeners = mutableListOf<MyListener>
()
    fun addListener(listener: MyListener) {
        listeners.add(listener)
    }
    fun performLogic() {
        listeners.forEach {
            it.onChange("newText")
        }
    }
}
```

Here, we have a **MyManager** class in which we collect a list of **MyListener** that will be invoked when **performLogic** is called. Note that the **MyManager** class is defined using the **object** keyword. This will make the **MyManager** class static, which means the instance of the class will live as long as the application process lives. If we want an activity to listen to when the **performLogic** method is called, we will have something like the following:

```
class MyActivity : Activity(), MyListener {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        MyManager.addListener(this)
    }
    override fun onChange(newText: String) {
        myTextView.setText(newText)
    }
}
```

Here, **MyListener** is implemented in **MyActivity**, and when **onChange** is called, **myTextView** will be updated. The context leak occurs here when the activity is destroyed. As **MyActivity** is a **MyListener** and a reference to it is kept in **MyManager**, which lives longer, the garbage collector will not remove the **MyActivity** instance from memory. If **performLogic** is called after **MyActivity** is destroyed, we will get **NullPointerException**, because **myTextView** will be set to null; or, if multiple instances of **MyActivity** leak, it could potentially lead to consuming the entire application's memory. A simple fix for this is to remove the reference to **MyActivity** when it is destroyed:

```
object MyManager {
    …
    fun removeListener(listener: MyListener){
        listeners.remove(listener)
    }
    …
}
class MyActivity : Activity(), MyListener {
    …
    override fun onDestroy() {
        MyManager.removeListener(this)
        super.onDestroy()
    }
    …
}
```

Here, we add a simple method to remove **`MyListener`** from the list and invoke it from the **`onDestroy`** method.

Working with fragments will lead to the same type of problems as activities. Fragments have their own lifecycle and inherit from a parent **`Fragment`** class, which makes them vulnerable to context leaks and hard to unit test.

The lifecycle of a fragment is as follows:

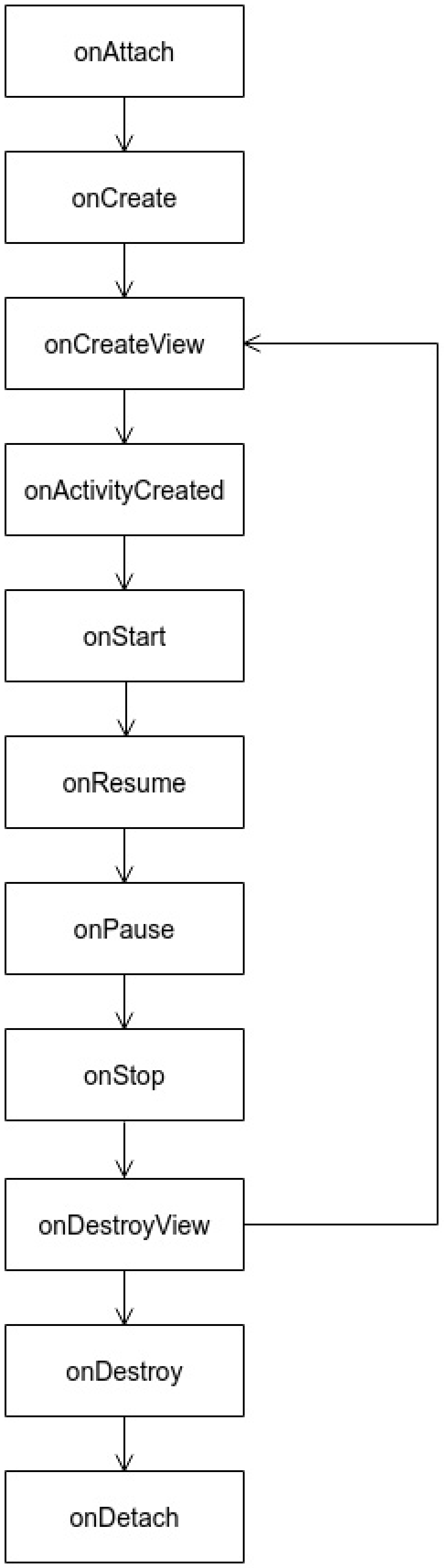


Figure 3.2 – Fragment lifecycle

In *Figure 3.2*, we can see that the fragment has similar lifecycle states to the activity. The **`onAttach`** and **`onDetach`** callbacks deal with when the fragment is attached to and detached from the activity. **`onActivityCreated`** is called when the activity completes its own **`onCreate`** call. The **`onCreateView`** and **`onDestroyView`** callbacks deal with inflating and destroying a fragment's views. One of the reasons these callbacks exist is because of the fragment back stack. This is a stack structure in which fragments are kept so that when the users press the *Back* button, the current fragment is popped out of the stack and the previous fragment is displayed. When fragments are replaced in the back stack, they aren't fully destroyed; just their views are destroyed to save memory. When they are popped back to be viewed by the user, they will not be re-created, and **`onCreateView`** will be called.

In order to solve the problems caused by dealing with activity and fragment lifecycles, a set of libraries was created that are part of the **`androidx.lifecycle`** group. The **`Lifecycle`** class was introduced, which is responsible for keeping the current lifecycle state and handling transitions between lifecycle events. The events and states of the **`Lifecycle`** class would be as follows:

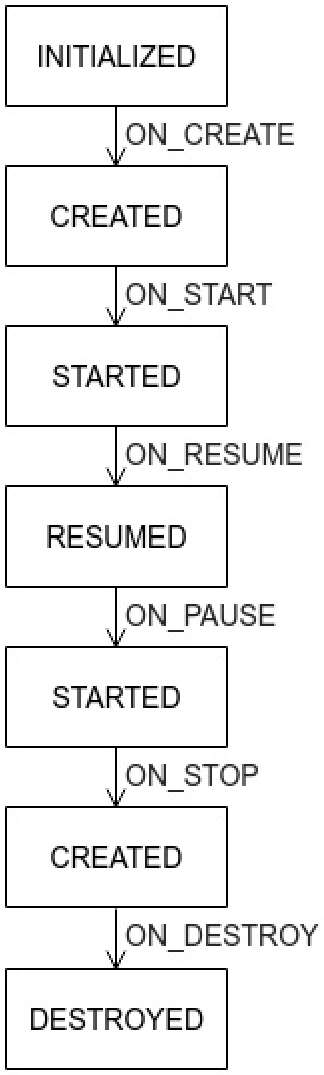


Figure 3.3 – Lifecycle states

In *Figure 3.3*, we can see that the **`Lifecycle`** class only has four states (*INITIALIZED*, *CREATED*, *STARTED*, and *DESTROYED*), and it will deal with six events (**`ON_CREATE`**, **`ON_START`**, **`ON_RESUME`**, **`ON_PAUSE`**, **`ON_STOP`**, and **`ON_DESTROY`**). If we wish for a certain class to be lifecycle-aware, it will need to implement the **`LifecycleOwner`** interface. Activities and fragments already implement this interface. We can see that for activities, the events match the existing callbacks, but for fragments, some changes are required to match these new events. The **`onAttach`**, **`onDetach`**, and **`onActivityCreated`** methods are deprecated, so they shouldn't be used with regard to the new **`Lifecycle`** library. The other change made for fragments is the introduction of a **`viewLifecycleObserver`** instance variable, which is used to handle the lifecycle between **`onCreateView`** and **`onDestroyView`**. This observer should be used when registering for lifecycle-aware components and you wish to update the UI.

In Android, when a configuration change (device rotation and language change, for example) occurs, then activities and fragments are re-created (the current instance is destroyed and a new instance will be created). This typically causes problems when these configuration changes occur while data is loaded or when we want to restore the previously loaded data. The **`ViewModel`** class is meant to solve this problem, along with the issue of testability of activities and fragments. A ViewModel will live until the activity or fragment it is connected to is destroyed and not re-created. The ViewModel comes with an **`onCleared`** method, which can be overwritten to clear any subscriptions to any pending operations.

ViewModels are often paired with a class called **`LiveData`**. This is a lifecycle-aware component that observes and emits data. The combination of the two classes eliminates the risks of context leaks, as **`LiveData`** will only emit data when the observer is in a *STARTED* or *RESUMED* state. An additional benefit is that it will keep the last data held; so, in the case of a configuration change, the last data kept in **`LiveData`** will be re-emitted. This benefit allows activities and fragments to observe the changes and restore the UI to the way it was before they were re-created. In Jetpack Compose, **`LiveData`** isn't necessary due to Compose's own set of state handling classes.

To use **`ViewModel`** and **`LiveData`**, you will need the following libraries to be added to **`build.gradle`**:

```
implementation "androidx.lifecycle:lifecycle-
viewmodel-ktx:2.4.0"
implementation "androidx.lifecycle:lifecycle-
livedata-ktx:2.4.0"
```

For integration with Jetpack Compose we will need the following:

```
implementation "androidx.lifecycle:lifecycle-
viewmodel-compose:2.4.0"
implementation "androidx.compose.runtime:runtime-
livedata:2.4.0 "
```

An example of a **`ViewModel`** and **`LiveData`** implementation will look something like this:

```
class MyViewModel : ViewModel() {
    private val _myLiveData = MutableLiveData("")
    val myLiveData: LiveData<String> = _myLiveData
    init {
        _myLiveData.value = "My new value"
    }
}
```

In the preceding example, we extend the **`ViewModel`** class and define two **`LiveData`** instance variables. The **`_myLiveData`** variable is defined as **`MutableLiveData`** and is set to private. This is to prevent other objects from changing the values of **`LiveData`**. The **`myLiveData`** variable is public and can be used by **`Lifecycle`** owners to observe changes on **`LiveData`**.

To obtain the instance of a ViewModel in an activity or fragment, we can use the following:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        …
        val myViewModel : MyViewModel by viewModels()
        …
    }
}
```

Here, the **`viewModels`** method will retrieve the instance of **`MyViewModel`**. The method provides the ability to pass along a **`ViewModelProvider.Factory`** object. This is useful in situations where we want to inject various objects in our ViewModel. This will look something like this:

```
        val myViewModel : MyViewModel by viewModels {
            object : ViewModelProvider.Factory {
                override fun <T : ViewModel>
                    create(modelClass: Class<T>): T {
                    return MyViewModel() as T
                }
            }
        }
```

If we want to observe the changes on **LiveData**, we would need to do something like this:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        …
        super.onCreate(savedInstanceState)
        val myViewModel: MyViewModel by viewModels()
        myViewModel.myLiveData.observe(this) { text -
>
            myTextView.text = text
        }
        …
    }
}
```

In the preceding example, we invoke the **observe** method, where we pass the activity as **LifecycleOwner** and pass a Lambda as **Observer**, which will be invoked when **LiveData** changes its value.

If we want to use **ViewModel** with **LiveData** in Jetpack Compose, we must do the following:

```
@Composable
```

```
fun MyScreen(viewModel: MyViewModel = viewModel()) {
    viewModel.myLiveData.observeAsState().value?.let
{
        MyComposable(it)
    }
}
@Composable
fun MyComposable(text: String){
    …
}
```

Here, we are using the **`viewModel`** method to obtain the **`MyViewModel`** instance. This method also offers the possibility of passing a **`ViewModelProvider.Factory`** instance, such as the previous **`viewModel`** method. The **`observeAsState`** extension method will observe changes on **`LiveData`** and convert them into a Compose **`State`** object.

In this section, we have discussed how lifecycles work in activities and fragments and the problems developers have when dealing with them. We have analyzed how the lifecycle-aware components (such as ViewModel and LiveData) solved these problems. The **`ViewModel`** class itself represents an implementation of the **Model-View ViewModel** (**MVVM**) pattern, which will be discussed in a future chapter. In the next section, we will look at an exercise in which we will use both ViewModel and LiveData and combine them with Kotlin flows.

Exercise 3.1 – Using ViewModel and LiveData

Modify *Exercise 2.5* from *Chapter 2*, *Deep Diving into Data Sources*, so that the state of the UI is kept in a **`LiveData`** object inside **`MainViewModel`**, instead of using the Compose **`State`** object, and display **`"Total request count: x"`**, where **`x`** is the number of requests at the top of the list.

To complete the exercise, you will need to build the following:

- Add the specified text in **`strings.xml`**.
- Create a **`MainTextFormatter`** class that will have one method that will return the **`"Total request count: x"`** text.
- Add a dependency to **`MainTextFormatter`** in **`MainViewModel`**, and pass the formatted text as a value for the **`UiState.count`** object.
- Remove **`resultState`** and replace it with a **`LiveData`** object.
- Update the **`@Composable`** functions to use **`LiveData`**.

Follow these steps to complete the exercise:

1. Add the **`LiveData`** extension library for Jetpack Compose to **`app/build.gradle`**:
```
        implementation
    "androidx.compose.runtime:runtime-
    livedata:$compose_version"
```
2. Add the **`"Total request count"`** text in **`strings.xml`**:
```
        <string name="total_request_count">Total
    request count: %d</string>
```
3. Create the **`MainTextFormatter`** class as follows:
```
    class MainTextFormatter(private val
        applicationContext: Context) {
        fun getCounterText(count: Int) =
            applicationContext.getString(R.string.total
    _request_co
        unt, count)
    }
```

The reason we created this class is to prevent possible context leaks by having a **`Context`** object inside the **`MainViewModel`** class. Here, we have a method that will take a count as a parameter and return the required text.

4. Inject **`MainTextFormatter`** in **`MainViewModel`** and use the formatted text as a value for the **`UiState.count`** object:
```
    class MainViewModel(
    …
    private val mainTextFormatter:
MainTextFormatter
) : ViewModel() {
    …
    init {
        viewModelScope.launch {
                …
                .flatMapConcat { users ->
                    appDataStore.savedCount.map {
                        count ->
                        UiState(
                            users,
                           mainTextFormatter.getCou
nterText(count)
                        )
                    }
                }
                …
        }
    }
}
```

5. Next, create the instance of the **`MainTextFormatter`** class in the **`MyApplication`** class:

```
class MyApplication : Application() {
    companion object {
        …
        lateinit var mainTextFormatter:
            MainTextFormatter
    }
    override fun onCreate() {
        super.onCreate()
        …
        mainTextFormatter = MainTextFormatter(this)
    }
}
```

6. Now, update **`MainViewModelFactory`** to use **`MainTextFormatter`**, which was just created, and pass it into **`MainViewModel`**:

```
class MainViewModelFactory :
ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass:
        Class<T>): T =
        MainViewModel(
            MyApplication.userService,
            MyApplication.userDao,
            MyApplication.appDataStore,
            MyApplication.mainTextFormatter
        ) as T
}
```

7. Next, add **`LiveData`** to **`MainViewModel`**:

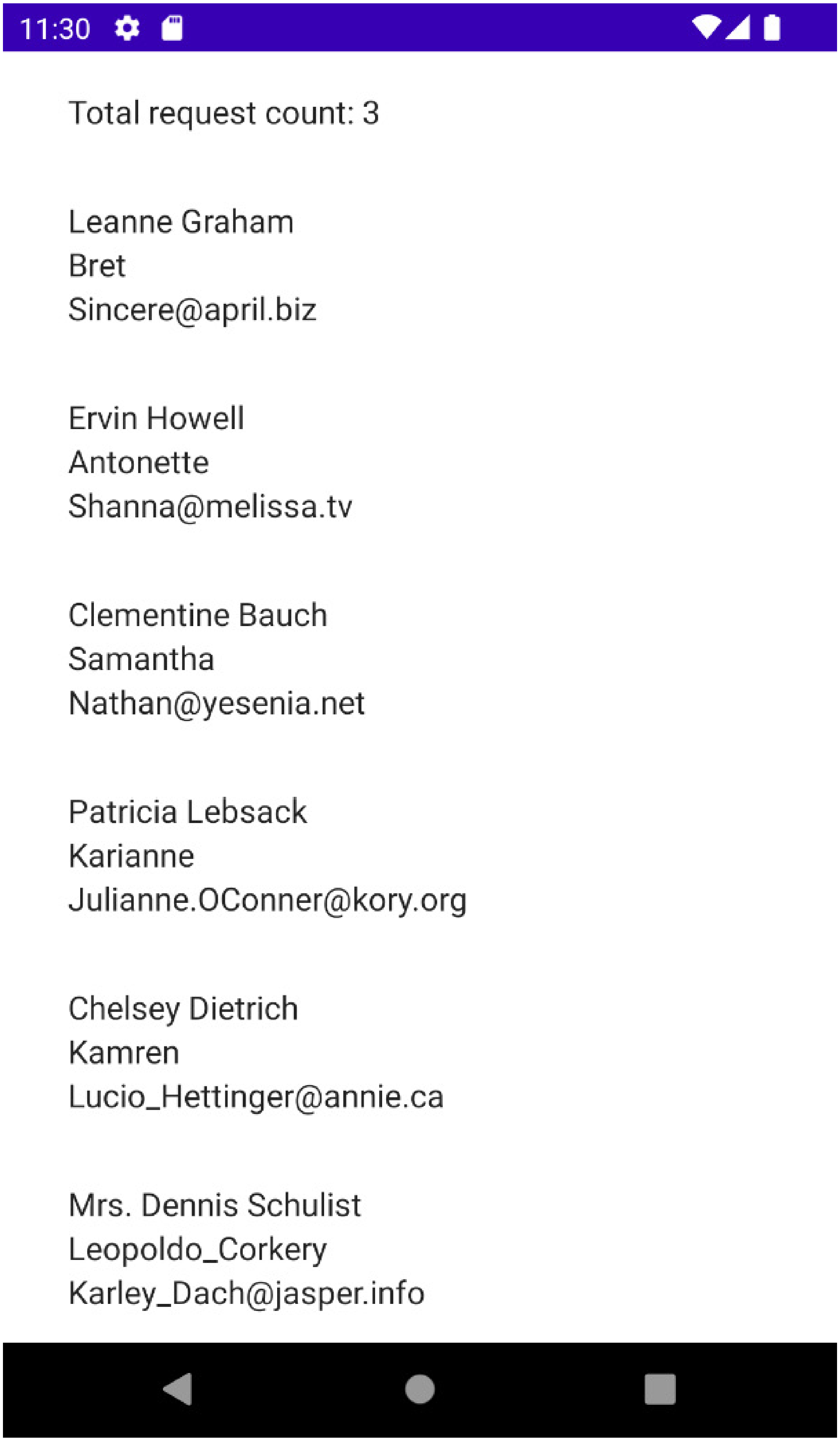
```
class MainViewModel(
    …
) : ViewModel() {
    private val _uiStateLiveData =
        MutableLiveData(UiState())
    val uiStateLiveData: LiveData<UiState> =
        _uiStateLiveData
    init {
        viewModelScope.launch {
                …
                .collect {
                    _uiStateLiveData.value = it
                }
        }
    }
}
```

Here, we have defined the two **`LiveData`** variables, one to update the value and the other to be observed, and in the **`collect`** method, we update the value of **`LiveData`**.

8. In **MainActivity**, update the **@Composable** functions to use **LiveData**:

```
…
@Composable
fun Screen(viewModel: MainViewModel =
viewModel(factory = MainViewModelFactory())) {
    viewModel.uiStateLiveData.observeAsState().valu
e?.let {
        UserList(uiState = it)
    }
}
…
```

Here, we call the **observeAsState** extension method on **LiveData** from **MainViewModel**, and then call the **UserList** method, which will redraw the UI for each new value.



11:30

Total request count: 3

Leanne Graham
Bret
Sincere@april.biz

Ervin Howell
Antonette
Shanna@melissa.tv

Clementine Bauch
Samantha
Nathan@yesenia.net

Patricia Lebsack
Karianne
Julianne.OConner@kory.org

Chelsey Dietrich
Kamren
Lucio_Hettinger@annie.ca

Mrs. Dennis Schulist
Leopoldo_Corkery
Karley_Dach@jasper.info

Figure 3.4 – Output of Exercise 3.1

If we run the application, we will see the same list of users, and at the top, we will see **`"Total request count: x"`** instead of just the **`x`** character that was there before, as shown in *Figure 3.4*. In this exercise, we used Jetpack Compose for rendering the UI. In the section that follows, we will analyze how Android handles UIs and go more in-depth into the Jetpack Compose framework.

Using Jetpack Compose to build UIs

In this section, we will analyze how to build UIs for Android applications using the **`View`** hierarchy and look at the implications this has for applications. We will then look at how Jetpack Compose simplifies and changes how UIs are built and how we can use Compose to create UIs. We will be looking at Jetpack Compose with the view of how we can integrate it with other libraries and how to build a simple UI. For more information on how to build more complex UIs, you can refer to the official documentation found here: https://developer.android.com/jetpack/compose ↗️.

The way Android deals with UIs is through the **`View`** hierarchy. The subclasses of **`View`** deal with specific UI components that the user can interact with. The hierarchy looks similar to the following diagram:

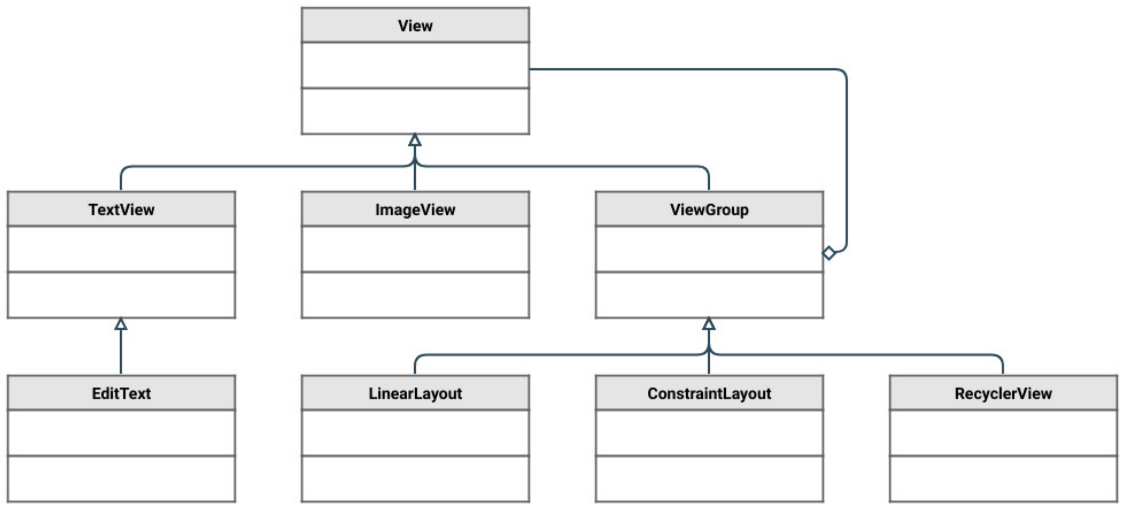


Figure 3.5 – View hierarchy

The `TextView` class deals with displaying text on the screen, `EditText` deals with handling text inputted by the user, and `Button` deals with rendering buttons on the screen. A specialized subclass of `View` is the `ViewGroup` class. This represents the base class for various layout classes that are responsible for how the views are grouped and arranged on the screen. Here, we find classes such as `LinearLayout` (which groups views one after the other either vertically or horizontally), `RelativeLayout` (which groups the views relative to the parent or to each other), or more recently, `ConstraintLayout`, which offers various ways to position views however we desire without creating many nested layouts (because it was bad for performance), which is why it became commonly used. When it comes to dealing with displaying lists of items of unknown lengths, objects such as `ListView` and `RecyclerView` are used. Both require creating adapters that will be responsible for pairing an object from a list with an associated `View` to render a row in the list in the UI.

Using `ListViews` is prone to inefficiencies caused when scrolling where views are recreated for each new row, so in a long list of items, a lot of views would be created and then garbage collected. To solve this, developers had to implement a pattern called a **ViewHolder**, which is responsible for keeping references to the views created for each row and re-using them for new rows when the user scrolls away. `RecyclerView` addresses this issue so the adapter `RecyclerView` uses requires `ViewHolder`. This means that if a user views a list of 100 items and 10 are visible on the screen, for the 10 that are visible on the screen there would be 10 views to represent each row. When the user scrolls down, the 10 views that were created at the beginning would then display the items for the currently visible items. Developers can also create custom views by extending any of the existing `View` classes. This is useful when certain UI components have to be re-used in different activities, fragments, or other custom views.

To display these views to the user, we would need to use activities and fragments. For activities, this would require invoking the `setContentView` method in the `onCreate` method, and in fragments, we would need to return a **View** object in the **onCreateView** method. We have the possibility of creating the entire layout for an activity or fragment in Java or Kotlin, but this would lead to a lot of code being written. This, and the fact that we can have different layouts for different screen sizes or device rotation, led to using the **res/layout** folder, in which we can specify how a layout might look. An example of how this might look is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res
/android"
    xmlns:app="http://schemas.android.com/apk/res-
auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

In the preceding example, we define **ConstraintLayout**, which contains only **TextView** that displays a **"Hello World"** text. To obtain a reference to **TextView** to allow us to change the text because of an action or data being loaded, we would need to use the **findViewById** method from either the **Activity** class or the **View** class. This would look something like the following:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
```

```
            val textView =
                findViewById<TextView>(R.id.text_view)
            textView.text = "Hello new world"
        }
    }
```

This approach would lead to possible crashes within an application. Developers would need to make sure that when a layout was set for **`Activity`** or **`Fragment`** and **`findViewById`** was used, then the view was added to the **`layout`** file. With the introduction of Kotlin, this was initially addressed through the Kotlin Synthetics framework, which generated extensions for the declared views in a layout. Kotlin Synthetics would generate an extension for a View's **`android:id`** XML tag, which would be accessible in the code. Later, this was replaced with **`ViewBinding`**. When **`ViewBinding`** is used in a project, a class is generated for each layout that will hold references to all the views in the layout, eliminating potential crashes related to **`findViewById`**. All these approaches with regard to creating your UI are defined as **imperative** because we need to specify the views that our interface uses and control how we update the views when data is changed.

An alternative approach to this is the **declarative** way of creating the UI. This concept allows us to describe what we want to show on the UI and the framework by using the appropriate views based on the description we provide. The notions of **state** are introduced here, where the UIs are redrawn when states change, rather than updating the existing views. In Android, we can use Jetpack Compose to create UIs in a declarative way. We no longer have to deal with the **`View`** hierarchy and instead use **`@Composable`** functions, in which we specify what we want to display on the screen without thinking of how we need to display it, and we can also create the UI using Kotlin using less code than we would normally. In Compose, the **`Hello World`** example would look something like the following code:

```
    class MainActivity : ComponentActivity() {
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Surface {
                HelloWorld()
            }
        }
    }
}
@Composable
fun HelloWorld() {
    Text(text = "Hello World")
}
```

If we want to update the text because of a change in data, we will need to use **`State`** objects from the **`Compose`** library. Compose will observe these states and, when the values are changed, Compose will redraw the UI associated with that state. An example of this is as follows:

```
@Composable
fun HelloWorld() {
    val text = remember { mutableStateOf("Hello
World") }
    ShowText(text = text.value) {
        text.value = text.value + "0"
    }
}
@Composable
fun ShowText(text: String, onClick: () -> Unit) {
    ClickableText(
        text = AnnotatedString(text = text),
        onClick = {
            onClick()
        })
}
```

In this example, when the text is clicked, the **`0`** character is appended to the text and the UI is updated. This is because of the use of **`mutableStateOf`**. The **`remember`** method is needed because this state is kept inside a **`@Composable`** function, and it is used to keep the state intact while recomposition happens (the UI is redrawn). To make the text clickable, we needed to change from **`Text`** to **`ClickableText`**. The reason we are using two **`@Composable`** functions is that we want to keep the **`@Composable`** functions as re-usable as possible. This is called **state hoisting**, where we separate the stateful (**`HelloWorld`**) components from the stateless components (**`ShowText`**).

When it comes to rendering lists of items, Compose offers a simple way of rendering them in the form of **`Column`** (for when the length of the list is known and short), and **`LazyColumn`** (when the list of items is unknown and could potentially be long). An example of this is from *Exercise 3.1*:

```
LazyColumn(modifier = Modifier.padding(16.dp)) {
        item(uiState.count) {
            Column(modifier =
Modifier.padding(16.dp)) {
                Text(text = uiState.count)
            }
        }
        items(uiState.userList) {
            Column(modifier =
Modifier.padding(16.dp)) {
                Text(text = it.name)
                Text(text = it.username)
                Text(text = it.email)
            }
        }
    }
```

Here, we display a header at the top of the item list, and we use another column to set the padding for the row; then, we display the entire list of items with the use of the **`items`** function, and for each row, we set the padding and display a group with three texts.

If we want to display input fields and buttons, then we can look at how we implemented the UI in *Exercise 2.1*, from Chapter 2, *Deep Diving into Data Sources*:

```
@Composable
fun Calculator(
    a: String, onAChanged: (String) -> Unit,
    b: String, onBChanged: (String) -> Unit,
    result: String,
    onButtonClick: () -> Unit
) {
    Column(modifier = Modifier.padding(16.dp)) {
        OutlinedTextField(
            value = a,
            onValueChange = onAChanged,
            keyboardOptions =
KeyboardOptions(keyboardType
                = KeyboardType.Number),
            label = { Text("a") }
        )
        OutlinedTextField(
            value = b,
            onValueChange = onBChanged,
            keyboardOptions =
KeyboardOptions(keyboardType
                = KeyboardType.Number),
            label = { Text("b") }
        )
        Text(text = result)
        Button(onClick = onButtonClick) {
            Text(text = "Calculate")
        }
    }
}
```

Here, we used **`OutlinedTextField`** to render the equivalent of **`TextInputLayout`**. We could have used **`TextField`** if we wanted the equivalent of a simple **`EditText`**. For displaying a button, we can use the **`Button`** method, which uses **`Text`** for rendering the text on the button.

Compose also has integrations with other libraries, such as **`ViewModel`** and **`LiveData`**:

```
@Composable
fun Screen(viewModel: MainViewModel =
viewModel(factory = MainViewModelFactory())) {
    viewModel.uiStateLiveData.observeAsState().value?
.let {
        UserList(uiState = it)
    }
}
```

Here, we can pass **`ViewModel`** as a parameter in our **`Composable`** function and use the **`observeAsState`** function to convert **`LiveData`** into a **`State`** object, which will then be observed by Compose to redraw the UI. Compose also supports integration with the **`Hilt`** library. When Hilt is added to a project, then there is no need to specify **`Factory`** for the ViewModel.

Another important feature of Compose is how it deals with navigation between different screens. The Compose navigation is built upon the **`androidx.navigation`** library. This allows Compose to use the **`NavHost`** and **`NavController`** components to navigate between different screens. The screens are built using Compose, which means that an application using only Compose would ideally have only one activity. This eliminates any potential problems regarding activity and fragment lifecycles. To introduce navigation into a project, the following library is required:

```
dependencies {
    …
    implementation "androidx.navigation:navigation-
compose:2.4.0-rc01"
    …
}
```

If we want to navigate from one screen to another, we will need to obtain **NavHostController** and pass it into a **@Composable** method that will represent the structure of the application:

```
Surface {
    val navController = rememberNavController()
    AppNavigation(navController = navController)
}
```

The **AppNavigation** **@Composable** method will look something like this:

```
@Composable
fun AppNavigation(navController: NavHostController) {
    NavHost(navController, startDestination =
"screen1") {
        composable(route = "screen1") {
            Screen1(navController)
        }
        composable(
            route = "screen2/{param}",
            arguments = listOf(navArgument("param") {
type
                = NavType.StringType })
        ) {
            Screen2(navController,
                it.arguments?.getString("param").orEm
pty())
        }
    }
}
```

In **AppNavigation**, we invoke the **NavHost** **@Composable** function in which we will place the screens of the application along with a route to each of them. In this case, **`Screen1`** will have a simple route to navigate to and **`Screen2`** will require an argument when it is navigated to indicated through the **`{param}`** notation. For arguments, we will need to specify the type of the argument. In this case, it will be **`String`**, and **`NavType.StringType`** indicates this. If we wish to pass more complex arguments, then we will need to supply our own custom types and indicate how they should be serialized and deserialized. When we want to navigate from **`Screen1`** to **`Screen2`**, then we will need to do the following:

```
@Composable
fun Screen1(navController: NavController) {
    Column(modifier = Modifier.clickable {
        navController.navigate("screen2/test")
    }) {
        Text(text = "My text")
    }
}
```

When **`Column`** is clicked in **`Screen1`**, it will invoke **`NavController`** to navigate to **`Screen2`** and pass the **`test`** argument. **`Screen2`** will look like the following:

```
@Composable
fun Screen2(navController: NavController, text:
String) {
    Column {
        Text(text = text)
    }
}
```

`Screen2` will use the text extracted from **`it.arguments?.getString("param").orEmpty()`** and it will display it on the UI.

In this section, we have discussed how Android deals with UIs. We have looked over the imperative approach and then introduced the declarative approach for Uis. We have analyzed the Jetpack Compose library and the problems it attempts to solve, such as less code and no XML declarations for layouts. It follows the principles of libraries from other technologies (such as React and SwiftUI) and shows how UIs can be built from a functional programming point of view. In the next section, we will look at an exercise for how we can use Compose to navigate between two screens in an application.

Exercise 3.2 – Navigating using Jetpack Compose

Modify *Exercise 3.1* so that the current **`@Composable`** functions are moved into a new file named **`UserListScreen`**, then create a new file with new **`@Composable`** functions that will render a simple text called **`UserScreen`**. When a user from the list is clicked, the new screen is opened and it will display the name of the user.

To complete the exercise, you will need to build the following:

1. Create an **`AppNavigation`** sealed class that will have two variables. The first variable, named **`route`**, will be **`String`** and the second variable, named **`argumentName`**, will be **`String`** and default to **`empty`**. Two subclasses of **`AppNavigation`** will be **`Users`** (which will set the route variable to **`"users"`**) and **`User`** (which will set the route to **`"users/{name}"`**, then **`argumentName`** to **`name`**, and a method to create the route for a specific name).
2. In **`MainActivity`**, rename the screen **`@Composable`** function to **`Users`**, and using the **`NavController`** object, set up a click listener on the list row and navigate to the route from the **`User`** class in **`AppNavigation`**.
3. Create a new **`@Composable`** function named **`User`**, which will be responsible for showing a simple **`Text`** and will have the text displayed as a parameter.
4. In **`MainActivity`**, create a **`@Composable`** function named **`MainApplication`**, which will use the **`NavHost`** **`@Composable`** function to link the navigation between the two screens.

Follow these steps to complete the exercise:

1. Add the **navigation** library for Compose in **app/build.gradle**:

```
dependencies {
    …
    implementation "androidx.navigation:navigation-
compose:2.4.0-rc01"
    …
}
```

2. Create the **AppNavigation** class, which will hold the information for the routes and arguments for each of our screens:

```
private const val ROUTE_USERS = "users"
private const val ROUTE_USER = "users/%s"
private const val ARG_USER_NAME = "name"
sealed class AppNavigation(val route: String, val
    argumentName: String = "") {
    object Users : AppNavigation(ROUTE_USERS)
    object User : AppNavigation
        (String.format(ROUTE_USER, "
{$ARG_USER_NAME}")
            , ARG_USER_NAME) {
        fun routeForName(name: String) =
            String.format(ROUTE_USER, name)
    }
}
```

As the navigation relies on URLs to identify the different screens, we can take advantage of sealed classes and objects in Kotlin to keep track of the required inputs for each screen.

3. Rename the screen **@Composable** function to **Users** in **MainActivity** and add **NavController** as a parameter:

```
@Composable
fun Users(
    navController: NavController,
    viewModel: MainViewModel = viewModel(factory
=
        MainViewModelFactory())
) {
    viewModel.uiStateLiveData.observeAsState().valu
e?.let {
        UserList(uiState = it, navController)
    }
}
```

4. Next, pass the **NavController** parameter to **UserList** and implement the click listener for the user row:

```
@Composable
fun UserList(uiState: UiState, navController:
NavController) {
    LazyColumn(modifier = Modifier.padding(16.dp))
{
        item(uiState.count) {
            Column(modifier =
Modifier.padding(16.dp)) {
                Text(text = uiState.count)
            }
        }
        items(uiState.userList) {
            Column(modifier = Modifier
                .padding(16.dp)
                .clickable {
                    navController.navigate
                      (AppNavigation.User.routeForN
ame
                           (it.name))
                }) {
                Text(text = it.name)
                Text(text = it.username)
                Text(text = it.email)
            }
        }
    }
}
```

5. Create the **`User @Composable`** function in **`MainActivity`**:

```
@Composable
fun User(text: String) {
    Column {
        Text(text = text)
    }
}
```

6. Now, create an **`App @Composable`** function that will use **`NavHost`** to set up the navigation between the two screens in **`MainActivity`**:

```
@Composable
fun App(navController: NavHostController) {
    NavHost(navController, startDestination =
        AppNavigation.Users.route) {
        composable(route =
AppNavigation.Users.route) {
            Users(navController)
        }
        composable(
            route = AppNavigation.User.route,
            arguments = listOf(navArgument
                (AppNavigation.User.argumentName) {
                type = NavType.StringType
            })
        ) {
            User(it.arguments?.getString(AppNavigat
ion.User.argumentName).orEmpty())
        }
    }
}
```

7. Finally, invoke the **`App`** function when the **`Activity`** content is set in **`MainActivity`**:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Exercise0302Theme {
        // Replace this with your application's
theme
                Surface {
                    val navController =
                        rememberNavController()
                    App(navController =
navController)
                }
            }
        }
    }
}
```

Figure 3.6 – Output of Exercise 3.2

If we run the application, we should see the same list of users as before, and if we click on a user, it will transition to a new screen that will display the selected user's name, as shown in *Figure 3.6*. If we press the *Back* button, we should see the initial list of users; that's because, by default, the **`navigation`** library handles back navigation.

In this exercise, we have analyzed how we can use Jetpack Compose to navigate between two screens in an application. In future chapters, we will revisit navigation when we must navigate between different screens in different modules.

Summary

In this chapter, we have analyzed how data can be presented in Android and discussed the libraries we have available now. We have looked at

Android lifecycles and the potential issues that applications could have regarding lifecycles and then looked at how libraries such as **`ViewModel`** and **`LiveData`** solve most of these problems. We then looked at how the UI works in Android and how we would need to deal with using XML to define layouts in which we would insert the views that the layouts needed to display, and how we would need to update the state of the views when the data changes. We then looked at how Jetpack Compose solves these issues in a declarative functional way. We built upon the exercises in the previous chapter to show how we can integrate multiple libraries in a single application and display data from the internet.

In the next chapter, we will deal with managing the dependencies inside an application and the libraries available for doing so.