Chapter 8: Implementing an MVVM Architecture

In this chapter, we will look at how data can be presented by Android applications to end users. We will look over the available architecture patterns for data presentation and analyze the differences between them. Later, we will look at the **Model-View-ViewModel (MVVM)** pattern, the role it plays in separating business logic and user interface updates, and how we can implement it using **Android Architecture Components**. Finally, we will look at how we can split the presentation layer across multiple library modules. In the exercises of this chapter, we will integrate the layers built in the previous chapters with a presentation layer built using MVVM, we will create a presentation layer that will plug into the domain layer to fetch and update the data, and we will also look at how we handle common logic between different modules in the presentation layer.

In this chapter, we will cover the following topics:

- Presenting data in Android applications
- Presenting data with MVVM
- Presenting data in multiple modules

By the end of the chapter, you will be able to implement the MVVM architecture pattern in an Android application using the ViewModel architecture component and be able to split the presentation layer into separate library modules.

Technical requirements

This chapter has the following hardware and software requirements:

- 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/Chapter8 ↗️.

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

Presenting data in Android applications

In this section, we will look at various architecture patterns suitable for presenting data in an Android application and analyze their benefits and drawbacks.

Early Android applications relied on a pattern similar to the **Model-View-Controller (MVC)** architecture pattern, where an activity is the Controller, the View is represented by the `android.widget.View` hierarchy, and the Model is responsible for managing the application's data. The relationship between the components would look something like the following:

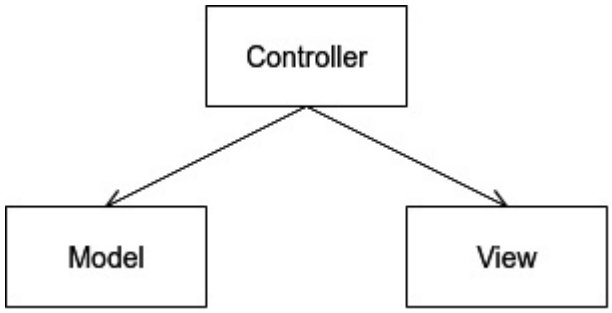

Figure 8.1 – Android MVC relationship

From *Figure 8.1*, we can see that the Controller represented by the activity would interact with the Model to fetch and manipulate the data, and then it would update the View with the relevant information.

The idea is to have each **`Activity`** sandboxed as much as possible so that they can be offered and shared between multiple applications (like how the Camera application is opened by other applications to take photos and offer those photos to those applications). Because of this, activities need to be started using intents and not by instantiating them. By removing the ability to instantiate an **`Activity`** directly, we lose the ability to inject dependencies through the constructor. Another factor we need to consider is that activities have life cycle states, and we inherit these states in each **`Activity`** in our application. All these factors combined make an **`Activity`** very hard or next to impossible to unit test unless we use a library such as **Robolectric** or rely on instrumented tests on an Android device or emulator. Both options are slow and, in the case of instrumented tests, can be expensive when we need to run the tests in testing clouds such as **Firebase Test Lab**.

To solve the problem of unit testing logic that was present in activities and later fragments, various adaptations of the **Humble Object** pattern emerged. More information about the pattern can be found here: http://xunitpatterns.com/Humble%20Object.html ↗️. The idea was to separate as much as possible the logic present in activities into separate objects and unit test those objects. One of the most popular solutions was the **Model-View-Presenter** (**MVP**) architecture pattern. In this pattern, the **`Activity`** along with the **`android.widget.View`** hierarchy becomes the View, the Presenter is responsible for fetching the data from the model and performing the logic required, updating the View, and the Model has the same responsibility as in MVC to handle the application's data. The relationship between these components looks like the following figure:

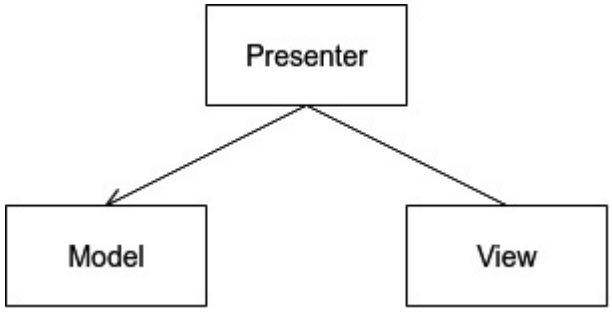

Figure 8.2 – MVP relationship

The interesting aspect of the relationship between the components is the relationship between the Presenter and the View, which goes both ways. The Presenter will update the View with the relevant data, but the View will also invoke the Presenter, if necessary for user interactions. Because of the relationship between the two components, the definition of a contract is required, which looks like the following:

```
interface Presenter {
    fun loadUsers()
    fun validateInput(text: String)
}
interface View {
    fun showUsers(users: List<User>)
    fun showInputError(error: String)
}
```

Here, we have a **`View`** interface and a **`Presenter`** interface. The implementation of the **`Presenter`** might look something like this:

```
class PresenterImpl(
    private val view: View,
    private val getUsersUseCase: GetUsersUseCase
) : Presenter {
    private val scope =
CoroutineScope(Dispatchers.Main)
    override fun loadUsers() {
        scope.launch {
            getUsersUseCase.execute()
                .collect { users ->
                    view.showUsers(users)
                }
        }
    }
    override fun validateInput(text: String) {
        if (text.isEmpty()) {
            view.showInputError("Invalid input")
```

```
        }
    }
}
```

Here, the **`PresenterImpl`** class has a dependency on the **`View`** and on a **`GetUsersUseCase`** object, which will return a **`Flow`** object containing a list of users. When the **`Presenter`** receives the list of users, it will call the **`showUsers`** method from the **`View`**. When the **`validateInput`** method is called, the **`Presenter`** will check whether the text is empty and invoke the **`showInputError`** method from the **`View`** with an error message. The implementation of the **`View`** might look like the following:

```
class MainActivity : ComponentActivity(), View {
    @Inject
    private lateinit var presenter: Presenter
    private lateinit var usersAdapter: UsersAdapter
    private lateinit var editText: EditText
    private lateinit var errorView: TextView
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        …
        editText.addTextChangedListener(object :
            TextWatcher {
            …
            override fun afterTextChanged(s:
Editable?) {
                presenter.validateInput(s?.toString()
.orEmpty())
            }
        })
        presenter.loadUsers()
    }
    override fun showUsers(users: List<User>) {
        usersAdapter.add(users)
    }
```

```
    override fun showInputError(error: String) {
        errorView.text = error
    }
}
```

Here, we implement the **`View`** interface in **`MainActivity`**; in the implementation of the methods, we call the appropriate **`View-`** related classes to show the relevant data, such as showing the error message for an invalid input in a **`TextView`** object and setting the data in a **`RecyclerView.Adapter`** object. For validating the input, when the text changes in an **`EditText`** object, it will invoke the **`Presenter`** to validate the new text. The **`Presenter`** dependency will be injected using some form of dependency injection.

Because presenters will end up performing background operations, we run the risk of causing **`Context`** leaks. This means that we need to factor the life cycle of the **`Activity`** into the MVP contract. To achieve this, we will need to define a **`close`** method in the **`Presenter`**:

```
interface Presenter {
    …
    fun close()
}
```

In the preceding snippet, we added the **`close`** method, which will be called in the **`onDestroy`** method of the **`Activity`** as follows:

```
override fun onDestroy() {
        presenter.close()
        super.onDestroy()
}
```

The implementation of the **`close`** method will have to clean up all the resources that might cause any leaks:

```
class PresenterImpl(
    private val view: View,
    private val getUsersUseCase: GetUsersUseCase
) : Presenter {
```

```
        private val scope =
    CoroutineScope(Dispatchers.Main)
        …
        override fun close() {
            scope.cancel()
        }
    }
```

Here, we are canceling the subscription to the **`Flow`** object so that we will not receive any updates after the **`Activity`** is destroyed.

In this section, we have looked at previous architecture patterns used in Android applications, from the MVC-like approach that was used in early Android applications to MVP, which aimed to solve some of the problems of the initial approach. Although MVP was popular in the past and is still present in some Android applications, it has slowly been phased out, mainly because of the release of Android Architecture Components, which rely on the MVVM pattern, and additionally, Jetpack Compose, which works better with data flows, which are more suited to MVVM. In the section that follows, we will look at the MVVM architecture pattern and how it is different from MVP as a concept.

Presenting data with MVVM

In this section, we will analyze the **Model-View-ViewModel** architecture pattern and how it is implemented for Android applications.

MVVM represents a different approach to the Humble Object pattern, which attempts to extract the logic out of activities and fragments. In MVVM, the View is represented by activities and fragments as it was in MVP, the Model plays the same role, managing the data, and the ViewModel sits between the two by requesting the data from the Model when the View requires it. The relationship between the three is as follows:

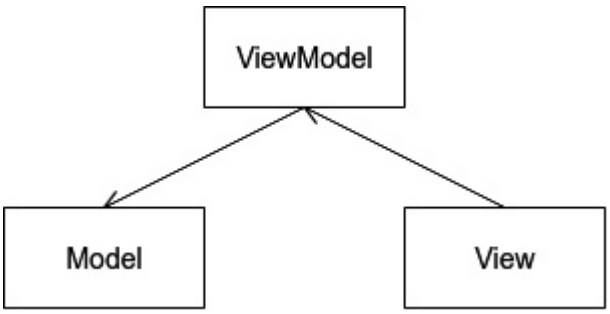

Figure 8.3 – MVVM relationship

In *Figure 8.3*, we see a unidirectional relationship between the three components. The View has a dependency on the ViewModel, and the ViewModel has a dependency on the Model. This allows for more flexibility because multiple Views can use the same ViewModel. For the data to be updated in the View, MVVM requires an implementation of the **Observer** pattern. This means that the ViewModel uses an **Observable**, which the View will subscribe to and react to changes in the data.

To develop Android applications, we have the possibility of using the Android Architecture Components libraries, which provide a **`ViewModel`** class that solves the issue of activity and fragment life cycles, combined with coroutine extensions useful for subscribing to flows or coroutines to stop the emission of data when the activities and fragments are in invalid states for data to be displayed and to avoid context leaks.

From the perspective of **Clean Architecture**, MVVM sits on the **Interface Adapter** layer. It has the role of fetching the data from the **Use Case** layer and converting the entities into objects that the **Framework** layer requires. It also handles changes to the data triggered by the user and converts this data back into entities, passing it back to the Use Case layer. In *Chapter 3*, *Understanding Data Presentation on Android*, we discussed the Android Architecture Components libraries and saw how we can implement ViewModels combined with **`LiveData`** (which acts as the observable that the View can subscribe to). An example of a **`ViewModel`** class might look like the following:

```
class MyViewModel(
```

```
        private val getUsersUseCase: GetUsersUserUseCase
    ) : ViewModel() {
        private val _usersFlow =
            MutableStateFlow<List<UiUser>>(listOf<UiUser>
    ())
        val usersFlow: StateFlow<List<UiUser>> =
    _usersFlow
        fun load() {
            viewModelScope.launch {
                getUsersUseCase.execute()
                    .map {
                        // Convert List<User> to
    List<UiUser>
                    }
                    .collect {
                        _usersFlow.value = it
                    }
            }
        }
    }
```

Here, we load a list of **`User`** objects and then keep that list inside a **`StateFlow`** object. This **`StateFlow`** object replaces **`LiveData`** and represents the observable that the View will subscribe to. When the View requires the list of users, it will invoke the **`load`** method.

In this section, we have analyzed the MVVM architecture pattern and the difference between it and the MVP pattern. In the following section, we will look at how we can present data using MVVM inside an Android application.

Exercise 08.01 – Implementing MVVM

Modify *Exercise 07.02*, *Building a local data source*, of *Chapter 7*, *Building Data Sources*, so that a new module called **`presentation-posts`** is created. The module will be responsible for displaying the data from

GetPostsWithUsersWithInteractionUseCase using MVVM. The data will be displayed in the following format:

- A header with the following text: "Total click count: x" where x is the number of clicks taken from the **totalClicks** field in the **Interaction** class
- A list of posts where each row contains the following: "Author: x" and "Title: y" where x is the **name** field in the **User** class, and y is the **title** field in the **Post** class
- A loading view for when the data is being loaded
- A **Snackbar** view for when there is an error

To complete this exercise, you will need to do the following:

1. Create the **presentation-post** module.
2. Create a new sealed class called **UiState**, which will have as subclasses **Loading**, **Error** (which will hold an error message), and **Success** (which will hold the post data).
3. Create a new class called **PostListItemModel**, which will have **id**, **author**, and **name** as fields.
4. Create a new class called **PostListModel**, which will have a **headerText** field and a list of **PostListItemModel** objects.
5. Create a new class called **PostListConverter**, which will convert a **Result.Success** object into a **UiState.Success**, which holds the **PostListModel** object and will convert a **Result.Error** object into a **UiState.Error** object.
6. Create a new class called **PostListViewModel**, which will load the data from **GetPostsWithUsersWithInteractionUseCase**, convert the data using **PostListConverter**, and store **UiState** in **StateFlow**.
7. Create a new Kotlin file, which will contain **@Composable** methods responsible for drawing the UI.
8. Modify **MainActivity** in the **app** module so that it will display the list of posts.

Follow these steps to complete the exercise:

1. Create a new module called **`presentation-post`**, which will be an Android library module.
2. Make sure that in the top-level **`build.gradle`** file, the following dependencies are set:

```
buildscript {
    …
    dependencies {
        classpath gradlePlugins.android
        classpath gradlePlugins.kotlin
        classpath gradlePlugins.hilt
    }
}
```

3. In the same file, add the persistence libraries to the library mappings:

```
buildscript {
    ext {
        …
        versions = [
                …
                viewModel            : "2.4.0",
                navigationCompose    : "2.4.0-
rc01",
                hiltNavigationCompose: "1.0.0-
rc01",
                …
        ]
        …
        androidx = [
                …
                viewModelKtx           : "androidx.
                    lifecycle:lifecycle-viewmodel-
                        ktx:${versions.viewModel}",
                viewModelCompose       : "androidx.
                    lifecycle:lifecycle-viewmodel-
                       compose:${versions.viewModel
}",
```

```
                navigationCompose      : "androidx.
                navigation:navigation-compose:$
                    {versions.navigationCompose}",
                hiltNavigationCompose  : "androidx.
                    hilt:hilt-navigation-compose:$
                     {versions.hiltNavigationCompos
e}"
        ]
        …
    }
    …
}
```

Here, we have added dependencies for the ViewModel library as well as the Navigation library (which will be used in later exercises).

4. In the **`build.gradle`** file of the **`presentation-post`** module, make sure that the following plugins are present:

```
plugins {
    id 'com.android.library'
    id 'kotlin-android'
    id 'kotlin-kapt'
    id 'dagger.hilt.android.plugin'
}
```

5. In the same file, change the configurations to the ones defined in the top-level **`build.gradle`** file:

```
android {
    compileSdk defaultCompileSdkVersion
    defaultConfig {
        minSdk defaultMinSdkVersion
        targetSdk defaultTargetSdkVersion
        …
    }
    …
    compileOptions {
```

```
            sourceCompatibility javaCompileVersion
            targetCompatibility javaCompileVersion
        }
        kotlinOptions {
            jvmTarget = jvmTarget
            useIR = true
        }
        buildFeatures {
            compose true
        }
        composeOptions {
            kotlinCompilerExtensionVersion versions.
                compose
        }
    }
```

Here, we keep the same configuration consistent with the other modules in the application, and we integrate the Jetpack Compose configuration.

6. In the same file, add the dependencies to the networking libraries and domain modules:

```
    dependencies {
        implementation(project(path: ":domain"))
        implementation coroutines.coroutinesAndroid
        implementation androidx.composeUi
        implementation androidx.composeMaterial
        implementation androidx.viewModelKtx
        implementation androidx.viewModelCompose
        implementation androidx.lifecycleRuntimeKtx
        implementation androidx.navigationCompose
        implementation di.hiltAndroid
        kapt di.hiltCompiler
        testImplementation test.junit
        testImplementation test.coroutines
        testImplementation test.mockito
```

```
}
```

7. In the **presentation-post** module, create a package called **list**.
8. In the **list** package, create the **UiState** class:

```
sealed class UiState<T : Any> {
    object Loading : UiState<Nothing>()
    data class Error<T : Any>(val errorMessage:
        String) : UiState<T>()
    data class Success<T : Any>(val data: T) :
        UiState<T>()
}
```

9. In the same package, create a file called **PostListModels**.
10. In the **PostListModels** file, create the **PostListItemModel** class:

```
data class PostListItemModel(
    val id: Long,
    val userId: Long,
    val authorName: String,
    val title: String
)
```

11. In the same file, create the **PostListModel** class:

```
data class PostListModel(
    val headerText: String = "",
    val items: List<PostListItemModel> = listOf()
)
```

12. In the **presentation-post** module, in the **src/main** folder, create a folder called **res**.
13. In the **res** folder, create a new folder called **values**.
14. In the **values** folder, create a file called **strings.xml**.
15. In the **strings.xml** file, add the following strings:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="total_click_count">Total click
        count: %d</string>
    <string name="author">Author: %s</string>
    <string name="title">Title: %s</string>
</resources>
```

16. In the **`list`** package, create the **`PostListConverter`** class:

```
class PostListConverter @Inject
constructor(@ApplicationContext private val
context: Context) {
    fun convert(postListResult: Result
        <GetPostsWithUsersWithInteractionUseCase.
            Response>): UiState<PostListModel> {
        return when (postListResult) {
            is Result.Error -> {
                UiState.Error(postListResult.
                 exception.localizedMessage.orEmpty
()) 
            }
            is Result.Success -> {
                UiState.Success(PostListModel(
                    headerText = context.getString(
                        R.string.total_click_count,
                        postListResult.data.
                            interaction.totalClicks
                    ),
                    items = postListResult.data.
                        posts.map {
                        PostListItemModel(
                            it.post.id,
                            it.user.id,
                            context.getString(R.str
ing.author, it.user.name),
                            context.getString(R.str
ing.title, it.post.title)
                        )
                    }
                ))
            }
        }
    }
```

```
}
```

Here, we convert the **`Result.Success`** and **`Result.Error`** objects into equivalent **`UiState`** objects, which will be used to display the information to the user.

17. In the **`list`** package, create the **`PostListViewModel`** class:

```
@HiltViewModel
class PostListViewModel @Inject constructor(
    private val useCase:
        GetPostsWithUsersWithInteractionUseCase,
    private val converter: PostListConverter
) : ViewModel() {
    private val _postListFlow =
        MutableStateFlow<UiState
            <PostListModel>>(UiState.Loading)
    val postListFlow:
        StateFlow<UiState<PostListModel>> =
            _postListFlow
    fun loadPosts() {
        viewModelScope.launch {
            useCase.execute
              (GetPostsWithUsersWithInteractionUseC
ase
                    .Request)
                .map {
                    converter.convert(it)
                }
                .collect {
                    _postListFlow.value = it
                }
        }
    }
}
```

Here, we get the list of posts and users from the **`GetPostsWithUsersInteractionUseCase`** object, then we convert it to the **`UiState`** object, and finally, we update **`StateFlow`** with the **`UiState`** object.

18. In the **`list`** package, create a file called **`PostListScreen`**.
19. In the **`PostListScreen`** file, add a method to display a loading widget and a **`Snackbar`** method:

```
@Composable
fun Error(errorMessage: String) {
    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Bottom
    ) {
        Snackbar {
            Text(text = errorMessage)
        }
    }
}
@Composable
fun Loading() {
    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment =
            Alignment.CenterHorizontally,
    ) {
        CircularProgressIndicator()
    }
}
```

20. In the same file, add a method to display the list of posts and the header:

```
@Composable
fun PostList(
    postListModel: PostListModel
) {
```

```
        LazyColumn(modifier = Modifier.padding(16.dp))
    {
            item(postListModel.headerText) {
                Column(modifier =
    Modifier.padding(16.dp)) {
                    Text(text =
    postListModel.headerText)
                }
            }
            items(postListModel.items) { item ->
                Column(
                    modifier = Modifier
                        .padding(16.dp)
                ) {
                    Text(text = item.authorName)
                    Text(text = item.title)
                }
            }
        }
    }
```

21. In the same file, add a method that will monitor the value of **`postList-Flow`** and invoke one of the preceding three methods, depending on the value of the state:

```
@Composable
fun PostListScreen(
    viewModel: PostListViewModel
) {
    viewModel.loadPosts()
    viewModel.postListFlow.collectAsState().value.l
et { state ->
        when (state) {
            is UiState.Loading -> {
                Loading()
            }
            is UiState.Error -> {
```

```
                    Error(state.errorMessage)
                }
                is UiState.Success -> {
                    PostList(state.data)
                }
            }
        }
    }
```

22. In the **`build.gradle`** file of the **`app`** module, make sure that the following plugins are added:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'kotlin-kapt'
    id 'dagger.hilt.android.plugin'
}
```

23. In the same file, make sure that the following dependencies are added:

```
dependencies {
    implementation(project(path: ":presentation-
        post"))
    implementation(project(path: ":domain"))
    implementation(project(path: ":data-remote"))
    implementation(project(path: ":data-local"))
    implementation(project(path: ":data-
repository"))
    implementation androidx.core
    implementation androidx.appCompat
    implementation material.material
    implementation androidx.composeUi
    implementation androidx.composeMaterial
    implementation androidx.composeUiToolingPreview
    implementation androidx.lifecycleRuntimeKtx
    implementation androidx.composeActivity
    implementation androidx.navigationCompose
    implementation androidx.hiltNavigationCompose
```

```
        implementation di.hiltAndroid
        kapt di.hiltCompiler
        testImplementation test.junit
    }
```

24. In the **app** module, create a package called **injection**.
25. In the **injection** package, create a class called **AppModule**:

```
    @Module
    @InstallIn(SingletonComponent::class)
    class AppModule {
        @Provides
        fun provideUseCaseConfiguration() =
            UseCase.Configuration(Dispatchers.IO)
    }
```

Here, we provide a **UseCase.Configuration** dependency, which will be injected into all the **UseCase** subclasses.

26. In the **app** module, create a class called **PostApplication**:

```
    @HiltAndroidApp
    class PostApplication : Application()
```

27. Add the **PostApplication** class to the **AndroidManifest.xml** file of the **app** module:

```
        <application
            …
            android:name=".PostApplication"
            …
        >
```

28. Modify the **MainActivity** class so that it will use the navigation library to go to the **PostListScreen** function from the **presentation-post** module:

```
    @AndroidEntryPoint
    class MainActivity : ComponentActivity() {
        override fun onCreate(savedInstanceState:
    Bundle?) {
            super.onCreate(savedInstanceState)
```

```
            setContent {
                CleanAppTheme {
                    Surface(color = MaterialTheme.
                        colors.background) {
                        val navController =
                            rememberNavController()
                        App(navController =
navController)
                    }
                }
            }
        }
    }
    @Composable
    fun App(navController: NavHostController) {
        NavHost(navController, startDestination =
            "/posts") {
            composable(route = "/posts") {
                PostListScreen(hiltViewModel())
            }
        }
    }
```

If we run the application, we should see the following screen:

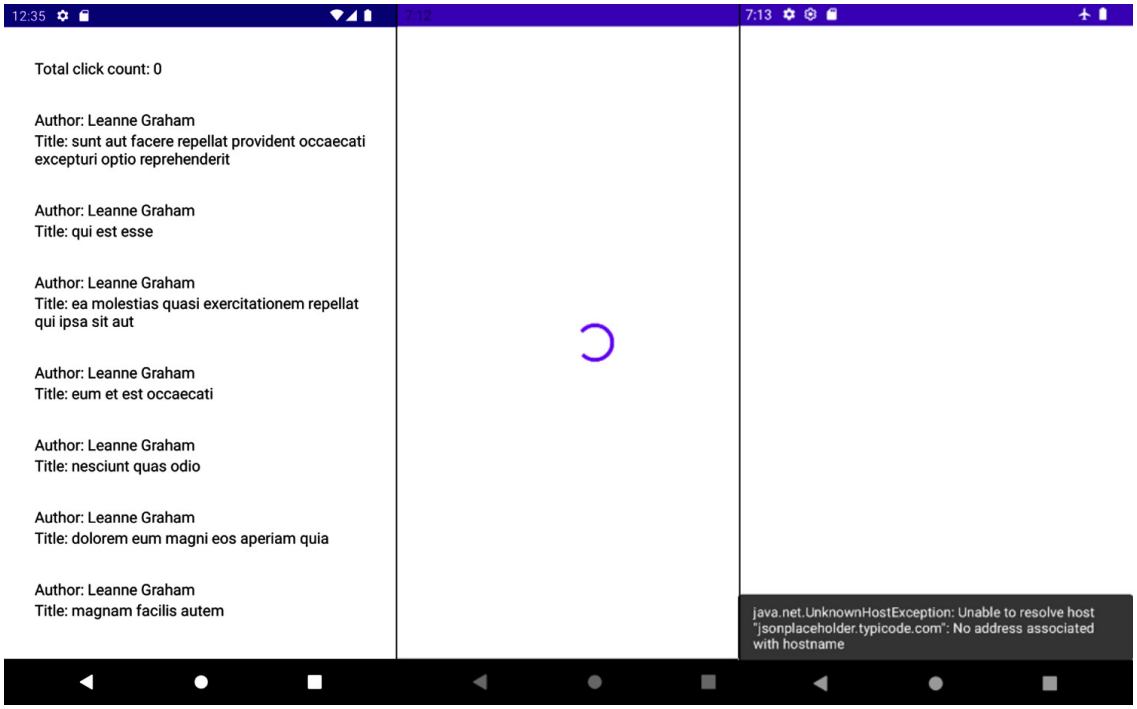

Figure 8.4 – Output of Exercise 08.01

We can see the list of post titles and the author's name for each post. The total click count is, for now, 0 because we haven't connected any logic and are yet to modify that value. We will add that logic in the exercises that follow. If an error occurs while loading this list, then we will see a snack-bar with the description of the **`Exception`** object, and while the data is loaded, an indeterminate progress bar will be displayed.

In this section, we have implemented the presentation layer of an Android application using the MVVM architecture pattern and connected the layer to the domain layer of the application to display data to the user. In the section that follows, we will expand this layer across multiple modules and see how we can navigate between screens in different modules.

Presenting data in multiple modules

In this section, we will look at how we can separate the presentation layer into multiple modules, how we can handle the interaction between these modules, and how they can share the same data.

When developing Android applications, we can group screens into different modules. For example, we can group a login or registration flow inside a library module called *authentication*, or if we have a settings section, we can group those screens inside a separate module. Sometimes these screens will have commonalities with the rest of the application, such as using the same loading progress bar or the same error mechanism. Other times, these screens must navigate to screens from other modules. The question we now need to ask is how this can happen without creating a dependency between the two modules or other modules that are on the same level. Having a direct dependency on these modules will risk creating a cyclic dependency as shown here:

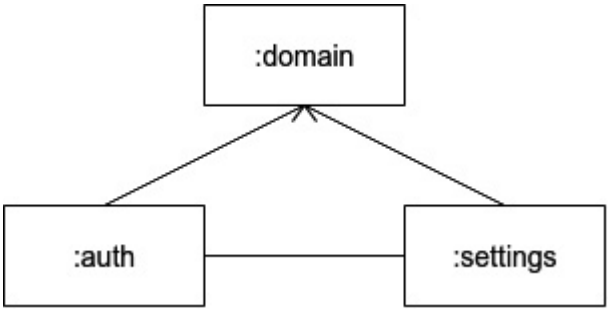

Figure 8.5 – Module cyclic dependency

In *Figure 8.5*, we show what might happen if we want to navigate from the **`:auth`** module to the **`:settings`** module and vice versa. This currently is impossible because of the cyclic dependency between the two modules. To solve this issue, we will need to create a new module. This module will hold the common logic shared between the two modules and common data. This will look like the following figure:

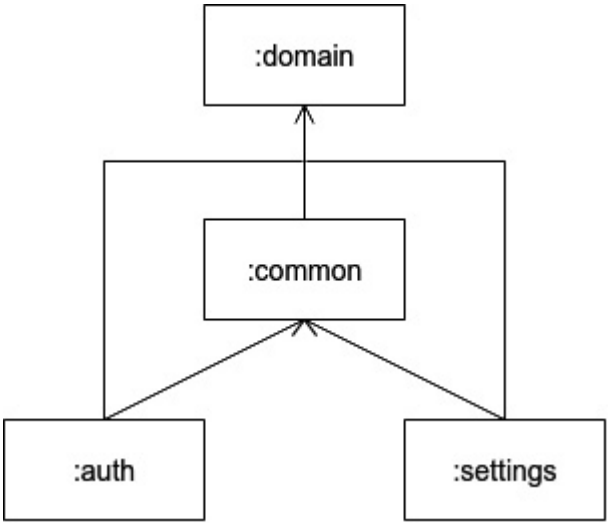

Figure 8.6 – Common presentation module

In *Figure 8.6*, we added the **`:common`** module, which will hold reusable views or **`@Composable`** functions and the navigation data from the app. Over time, this module will grow, so it can be split into different modules holding different common features of the app (navigation, UI, common logic, and so on).

If we are using Jetpack Compose for our application, then we can rely on the work done in *Exercise 03.02 – Navigating using Jetpack Compose* of *Chapter 3*, *Understanding Data Presentation on Android*, where we defined the following structure for the app navigation:

```
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)
    }
```

```
}
```

The **`routeForName`** method was called from the **`Users`** screen when a user in the list was clicked, and then the **`NavHost`** method would use that route to open the **`User`** screen. When dealing with multiple modules, the routes that will be shared by the modules can be stored in the **`:common`** module so that each module will have access to the route. The **`:app`** module, which will have **`NavHost`**, will then be able to navigate between each screen.

When it comes to handling common logic between different modules, such as displaying the same error or loading views, we can declare the composable functions inside the **`:common`** module:

```
@Composable
fun Error(errorMessage: String) {
    …
}
@Composable
fun Loading() {
    …
}
```

If the same state is shared between the different screens in the different modules, we can have something like the following:

```
@Composable
fun <T> CommonScreen(state: State<T>, onSuccess:
    @Composable (T) -> Unit) {
    when (result) {
        is State.Success -> {
            onSuccess(result.data)
        }
        is State.Error -> {
            Error(result.errorMessage)
        }
        is State.Loading -> {
            Loading()
```

```
            }
        }
    }
```

Here, we will check the current state and display the common error and loading views, leaving the screens themselves to only concern themselves with the successful state.

In this section, we have looked at how we can split the presentation layer into multiple modules and how to handle the common elements between these modules. In the following section, we will look at an exercise on how to achieve this. Splitting the presentation layer into multiple modules will decrease application build times because Gradle caching will only rebuild modules that contain changes. Another benefit comes in the form of drawing boundaries around the application's scope, which will be beneficial when it comes to exporting only certain features of an application.

Exercise 08.02 – Multi-module data presentation

Modify *Exercise 08.01 – Implementing MVVM* so that two new modules are created: **`presentation-post`** and **`presentation-common`**.

The **`presentation-common`** module will have the following:

- The **`UiState`** class, which will be moved from the **`presentation-post`** module.
- **`CommonResultConverter`**, which will be an abstract class with two methods: **`convert`**, which is a concrete method that will convert the **`Result`** object into a **`UiState`** object, and **`convertSuccess`**, which is an abstract method used to convert the data from **`Result.Success`**.
- **`CommonScreen`**, which will have the **`@Composable`** method for displaying the different types of **`UiState`** and two additional methods for displaying the error snackbar and the progress bar. The two methods will be moved from **`PostListScreen`**.

- **`AppNavigation`**, which will hold the routes to navigate to the list of posts, a single post, and a single user.
- The **`presentation-post`** module will have an additional package to display the information of a single post in the following format: Title: x and Body: y, where x is the title of a post and y is the body of the post. To display this information, a new **`ViewModel`** and **`Converter`** class will need to be created, which will convert the data from **`GetPostUseCase`**. When the author text is clicked, the app will navigate to the user screen, and when the **`Post`** list item is clicked, the app will navigate to the post screen. When either of these is clicked, **`UpdateInteractionUseCase`** is invoked to increase the number of clicks, which will then be reflected in the list header.
- **`presentation-user`** will display the information about a single user in the following format: Name: x, Username: y, and Email: z, where x, y, and z are represented by the information inside the **`User`** entity. The user data will be loaded from **`GetUserUseCase`**.
- The **`app`** module will be updated to handle the navigation between all these screens.

To complete this exercise, you will need to do the following:

1. Create the **`presentation-common`** module.
2. Move the **`UiState`** class and the **`Error`** and **`Loading`** **`@Composable`** functions and create a new **`@Composable`** function, which will handle each type of **`UiState`** object inside the **`CommonScreen`** file.
3. Create the **`CommonResultConverter`** class.
4. Create the **`AppNavigation`** class.
5. Modify the classes in **`presentation-post`** to reuse the preceding classes and methods.
6. Create the **`PostScreen`**, **`PostViewModel`**, **`PostConverter`**, and **`PostModel`** classes responsible for displaying the information about a single post.
7. Create the **`presentation-user`** module.
8. Create the **`UserScreen`**, **`UserViewModel`**, **`UserConverter`**, and **`UserModel`** classes responsible for displaying the information about a single post.
9. Implement the navigation between the screens.

10. Add the logic to update the number of clicks inside **`PostListViewModel`**.

Follow these steps to complete the exercise:

1. Create the **`presentation-common`** and **`presentation-user`** Android library modules.
2. Apply steps 3–5 from *Exercise 08.01 – Implementing MVVM* for each of these new modules.
3. In the **`build.gradle`** file of the **`presentation-post`** and **`presentation-user`** modules, make sure that the dependency to **`presentation-common`** is added:

```
dependencies {
    …
    implementation(project(path: ":presentation-
common"))
    …
}
```

4. In the **`presentation-common`** module, create a new package called **`state`**.
5. Move the **`UiState`** class into the preceding package.
6. In the same package, create the **`CommonResultConverter`** class:

```
abstract class CommonResultConverter<T : Any, R :
Any> {
    fun convert(result: Result<T>): UiState<R> {
        return when (result) {
            is Result.Error -> {
                UiState.Error(result.exception.
                    localizedMessage.orEmpty())
            }
            is Result.Success -> {
                UiState.Success(convertSuccess
                    (result.data))
            }
        }
    }
    abstract fun convertSuccess(data: T): R
```

```
}
```

Here, we return **`UiState.Error`** for any **`Result.Error`** object with the exception message, and for **`Result.Success`**, we return **`UiState.Success`** and use an abstraction for the data inside the **`Result.Success`** object. This represents a solution for how we can extract the common logic for displaying the error.

7. Modify the **`PostListConverter`** class from the **`presentation-post`** module so that it will extend **`CommonResultConverter`** and provide an implementation for the **`convertSuccess`** method:

```
class PostListConverter @Inject constructor
(@ApplicationContext private val context: Context)
:
    CommonResultConverter<GetPostsWithUsersWithInte
raction
    UseCase.Response, PostListModel>() {
    override fun convertSuccess(data:
        GetPostsWithUsersWithInteractionUseCase.
            Response): PostListModel {
        return PostListModel(
            headerText = context.getString(
                R.string.total_click_count,
                data.interaction.totalClicks
            ),
            items = data.posts.map {
                PostListItemModel(
                    it.post.id,
                    it.user.id,
                    context.getString(R.string.auth
or,
                        it.user.name),
                    context.getString(R.string.titl
e,
                        it.post.title)
```

```
                    )
                }
            )
        }
    }
```

Here, we only deal with converting **`GetPostsWithUsersWithInteractionUseCase.Response`** into **`PostListModel`**, allowing the parent class to handle the error only.

8. In the **`state`** package from the **`presentation-common`** module, create a new file called **`CommonScreen`**.
9. In the **`CommonScreen`** file, add a **`CommonScreen @Composable`** method, which will check **`UiState`** and invoke **`Error`** for **`UiState.Error`** and **`Loading`** for **`UiState.Loading`**:

```
@Composable
fun <T : Any> CommonScreen(state: UiState<T>,
onSuccess: @Composable (T) -> Unit) {
    when (state) {
        is UiState.Loading -> {
            Loading()
        }
        is UiState.Error -> {
            Error(errorMessage =
state.errorMessage)
        }
        is UiState.Success -> {
            onSuccess(state.data)
        }
    }
}
```

10. Move the **`Error`** and **`Loading`** **`@Composable`** functions from **`PostListScreen`** into the **`CommonScreen`** file.
11. Modify the **`PostListScreen`** **`@Composable`** method from the **`presentation-post`** module so that it will use the **`CommonScreen`** method:

```
@Composable
fun PostListScreen(
    viewModel: PostListViewModel
) {
    viewModel.loadPosts()
    viewModel.postListFlow.collectAsState().value.l
et
        { state ->
        CommonScreen(state = state) {
            PostList(postListModel = it)
        }
    }
}
```

Now the entire logic for converting and showing the list of posts will only deal with the associated objects, leaving the error and loading scenarios in the **`presentation-common`** module.

12. In **`presentation-common`**, create a new package called **`navigation`**.
13. In the **`navigation`** package, create a class called **`PostInput`**:

```
data class PostInput(val postId: Long)
```

This class is meant to represent the input that the post screen will require to load its data.

14. In the same package, create a class called **`UserInput`**:

```
data class UserInput(val userId: Long)
```

This class is meant to represent the input that the user screen will require to load its data.

15. In the same package, create a new class called **`NavRoutes`**:

```
private const val ROUTE_POSTS = "posts"
private const val ROUTE_POST = "posts/%s"
private const val ROUTE_USER = "users/%s"
```

```
private const val ARG_POST_ID = "postId"
private const val ARG_USER_ID = "userId"
sealed class NavRoutes(
    val route: String,
    val arguments: List<NamedNavArgument> =
        emptyList()
) {
    …
}
```

Here, we define the paths for each screen. The posts screen will have no arguments, but the user and post screens will require the **`postId`** and **`userId`** values.

16. Create the **`Posts`** class in the **`NavRoutes`** class:

```
sealed class NavRoutes(
    val route: String,
    val arguments: List<NamedNavArgument> =
        emptyList()
) {
    object Posts : NavRoutes(ROUTE_POSTS)
}
```

17. Create the **`Post`** class in the **`NavRoutes`** class:

```
sealed class NavRoutes(
    val route: String,
    val arguments: List<NamedNavArgument> =
        emptyList()
) {
    object Post : NavRoutes(
        route = String.format(ROUTE_POST,
            "{$ARG_POST_ID}"),
        arguments = listOf(navArgument(ARG_POST_ID)
{
            type = NavType.LongType
        })
```

```
    ) {
        fun routeForPost(postInput: PostInput) =
           String.format(ROUTE_POST,
postInput.postId)
        fun fromEntry(entry: NavBackStackEntry):
            PostInput {
            return PostInput(entry.arguments?.
                getLong(ARG_POST_ID) ?: 0L)
        }
    }
}
```

Here, we will need to break down the **`Post`** input into the arguments for the URL. The **`routeForPost`** method will create a **`/posts/1`** URL for a **`Post`** object that has the ID **`1`**. The **`fromEntry`** method will re-assemble the **`PostInput`** object from the navigation entry object. The reason we are taking this approach is that the navigation library discourages the use of **`Parcelable`**, which means that passing data between different screens will have to be done through the URL. To avoid any issues with keeping track of the arguments across multiple modules, we can instead use objects and keep the logic to read from arguments and construct the arguments isolated to this class.

18. Create the **`User`** class inside the **`NavRoutes`** class:

```
sealed class NavRoutes(
    val route: String,
    val arguments: List<NamedNavArgument> =
emptyList()
) {
    object User : NavRoutes(
        route = String.format(ROUTE_USER,
            "{$ARG_USER_ID}"),
        arguments = listOf(navArgument(ARG_USER_ID)
{
            type = NavType.LongType
```

```
            })
        ) {
            fun routeForUser(userInput: UserInput) =
               String.format(ROUTE_USER,
    userInput.userId)
            fun fromEntry(entry: NavBackStackEntry):
                UserInput {
                return
    UserInput(entry.arguments?.getLong
                    (ARG_USER_ID) ?: 0L)
            }
        }
    }
```

Here, we apply the same principle as we did for the **`Post`** class.

19. Create a new package called **`single`** in the **`presentation-post`** module.
20. In the **`single`** package, create the **`PostModel`** class:

```
data class PostModel(
    val title: String,
    val body: String
)
```

21. In the **`single`** package, create the **`PostConverter`** class:

```
class PostConverter @Inject
constructor(@ApplicationContext private val
context: Context) :
    CommonResultConverter<GetPostUseCase.Response,
        PostModel>() {
    override fun convertSuccess(data:
        GetPostUseCase.Response): PostModel {
        return PostModel(
            context.getString(R.string.title,
                data.post.title),
            context.getString(R.string.body,
                data.post.body)
```

```
            )
        }
    }
```

22. Add the **`body`** string to **`strings.xml`** of the **`presentation-post`** module:

```
<resources>
    …
    <string name="body">Body: %s</string>
</resources>
```

23. In the **`single`** package, create the **`PostViewModel`** class:

```
@HiltViewModel
class PostViewModel @Inject constructor(
    private val postUseCase: GetPostUseCase,
    private val postConverter: PostConverter
) : ViewModel() {
    private val _postFlow =
        MutableStateFlow<UiState<PostModel>>
(UiState.Loading)
    val postFlow: StateFlow<UiState<PostModel>> =
        _postFlow
    fun loadPost(postId: Long) {
        viewModelScope.launch {
            postUseCase.execute(GetPostUseCase.
                Request(postId))
                .map {
                    postConverter.convert(it)
                }
                .collect {
                    _postFlow.value = it
                }
        }
    }
}
```

Here, we are using **`GetPostUseCase`** to load the information about a particular post and are using the converter defined earlier to convert the data

into **PostModel**, which will be set in the **Flow** object.

24. In the **single** package, create the **PostScreen** file, which will display the post information:

```
@Composable
fun PostScreen(
    viewModel: PostViewModel,
    postInput: PostInput
) {
    viewModel.loadPost(postInput.postId)
    viewModel.postFlow.collectAsState().value.let {
        result ->
        CommonScreen(result) { postModel ->
            Post(postModel)
        }
    }
}
@Composable
fun Post(postModel: PostModel) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = postModel.title)
        Text(text = postModel.body)
    }
}
```

Here, we follow the same principle as for the **PostListScreen** file, where we split into two methods, **PostScreen** for observing the **UiState** object and **PostListScreen** to deal with drawing the user interface.

25. In the **presentation-user** module, create a new package called **single**.
26. In the **single** package, create a new class called **UserModel**:

```
data class UserModel(
    val name: String,
    val username: String,
    val email: String
```

```
)
```

27. In the **single** package, create a new class called **UserConverter**:

```
class UserConverter @Inject
constructor(@ApplicationContext private val
context: Context) :
    CommonResultConverter<GetUserUseCase.Response,
        UserModel>() {

    override fun convertSuccess(data:
GetUserUseCase.
        Response): UserModel {
        return UserModel(
            context.getString(R.string.name,
                data.user.name),
            context.getString(R.string.username,
                data.user.username),
            context.getString(R.string.email,
                data.user.email)
        )
    }
}
```

28. Create the **res/values/strings.xml** file inside the **main** folder in the **presentation-user** module:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="name">Name: %s</string>
    <string name="username">Username: %s</string>
    <string name="email">Email: %s</string>
</resources>
```

29. Inside the **single** package, create **UserViewModel**:

```
@HiltViewModel
class UserViewModel @Inject constructor(
    private val userUseCase: GetUserUseCase,
    private val converter: UserConverter
) : ViewModel() {
```

```
        private val _userFlow =
            MutableStateFlow<UiState<UserModel>>
                (UiState.Loading)
        val userFlow: StateFlow<UiState<UserModel>> =
            _userFlow
        fun loadUser(userId: Long) {
            viewModelScope.launch {
                userUseCase.execute
                    (GetUserUseCase.Request(userId))
                    .map {
                        converter.convert(it)
                    }
                    .collect {
                        _userFlow.value = it
                    }
            }
        }
    }
```

Here, we take the user data from **`GetUserUseCase`**, convert it using **`UserConverter`**, and post the result in the **`Flow`** object.

30. In the **`single`** package, create the **`UserScreen`** file:

```
    @Composable
    fun UserScreen(
        viewModel: UserViewModel,
        userInput: UserInput
    ) {
        viewModel.loadUser(userInput.userId)
        viewModel.userFlow.collectAsState().value.let {
            result ->
            CommonScreen(result) { userModel ->
                User(userModel)
            }
        }
```

```
}
@Composable
fun User(userModel: UserModel) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = userModel.name)
        Text(text = userModel.username)
        Text(text = userModel.email)
    }
}
```

Here, we take the same approach as the other screens, where in one method, we subscribe to changes in **`UiState`**, and in the other, we display the user information.

31. Add the click listeners in **`PostListScreen`**:

```
@Composable
fun PostList(
    postListModel: PostListModel,
    onRowClick: (PostListItemModel) -> Unit,
    onAuthorClick: (PostListItemModel) -> Unit
) {
    LazyColumn(modifier = Modifier.padding(16.dp))
{
        …
        items(postListModel.items) { item ->
            Column(modifier = Modifier
                .padding(16.dp)
                .clickable {
                    onRowClick(item)
                }) {
                ClickableText(text =
AnnotatedString(
                    text = item.authorName),
onClick =
                {
```

```
                        onAuthorClick(item)
                    })
                    Text(text = item.title)
                }
            }
        }
    }
```

In the preceding snippet, we specify click listeners for when the row is clicked and for when the author is clicked. Because we are applying state hoisting, we want to propagate the click listeners to the caller of the **`PostList`** method. To achieve this, we define a parameter for each click listener as a lambda function that has as input the row data and requires no result. More information about lambdas can be found here: https://kotlinlang.org/docs/lambdas.html#function-types ↗️.

32. Modify the **`PostListScreen`** **`@Composable`** method so that when the user is clicked, we navigate to the user screen, and when the row is clicked, we navigate to the post:

```
@Composable
fun PostListScreen(
    viewModel: PostListViewModel,
    navController: NavController
) {
    viewModel.loadPosts()
    viewModel.postListFlow.collectAsState().value.l
et
        { state ->
        CommonScreen(state = state) {
            PostList(it, { postListItem ->
                navController.navigate(NavRoutes.Po
st.
                    routeForPost(PostInput
                        (postListItem.id)))
            }) { postListItem ->
```

```
                    navController.navigate(NavRoutes.Us
er.
                        routeForUser(UserInput
                            (postListItem.userId)))
                }
            }
        }
    }
```

33. In **`build.gradle`** of the **`app`** module, make sure that the dependencies to **`presentation-common`** and **`presentation-user`** are added:

```
dependencies {
    …
    implementation(project(path: ":presentation-
        user"))
    implementation(project(path: ":presentation-
        common"))
    …
}
```

34. In the **`MainActivity`** file, modify the **`App`** method so that the navigation between the different screens is implemented:

```
@Composable
fun App(navController: NavHostController) {
    NavHost(navController, startDestination =
        NavRoutes.Posts.route) {
        composable(route = NavRoutes.Posts.route) {
            PostListScreen(hiltViewModel(),
                navController)
        }
        composable(
            route = NavRoutes.Post.route,
            arguments = NavRoutes.Post.arguments
        ) {
            PostScreen(
                hiltViewModel(),
                NavRoutes.Post.fromEntry(it)
```

```
                )
            }
            composable(
                route = NavRoutes.User.route,
                arguments = NavRoutes.User.arguments
            ) {
                UserScreen(
                    hiltViewModel(),
                    NavRoutes.User.fromEntry(it)
                )
            }
        }
    }
```

Here, we add all the screens in the application to the navigation graph, and in the case of **`UserScreen`** and **`PostScreen`**, we extract the **`UserInput`** and **`PostInput`** objects from the navigation graph entries. We will now need to add the interaction.

35. Add an **`Interaction`** field inside **`PostListModel`**:

```
data class PostListModel(
    …
    val interaction: Interaction
)
```

36. Modify **`PostListConverter`** to include the **`interaction`** field:

```
class PostListConverter @Inject constructor
(@ApplicationContext private val context: Context)
:
    CommonResultConverter<GetPostsWithUsersWithInte
raction
    UseCase.Response, PostListModel>() {
    override fun convertSuccess(data:
        GetPostsWithUsersWithInteractionUseCase.
            Response): PostListModel {
        return PostListModel(
```

```
                …
                interaction = data.interaction
            )
        }
    }
```

37. Add a reference to **`UpdateInteractionUseCase`** in **`PostListViewModel`** and a method to update the interaction:

```
@HiltViewModel
class PostListViewModel @Inject constructor(
    …
    private val updateInteractionUseCase:
        UpdateInteractionUseCase
) : ViewModel() {
    …
    fun updateInteraction(interaction: Interaction)
{
        viewModelScope.launch {
            updateInteractionUseCase.execute(
                UpdateInteractionUseCase.Request(
                    interaction.copy(
                        totalClicks = interaction.
                            totalClicks + 1
                    )
                )
            ).collect()
        }
    }
}
```

38. Modify the **`PostListScreen`** **`@Composable`** method so that it will call to update the interaction for each click:

```
@Composable
fun PostListScreen(
    viewModel: PostListViewModel,
    navController: NavController
) {
```

```
            …
            viewModel.postListFlow.collectAsState().value.l
et
                { state ->
                CommonScreen(state = state) {
                    PostList(it, { postListItem ->
                        viewModel.updateInteraction(it.inte
raction)
                        …
                    }) { postListItem ->
                        viewModel.updateInteraction(it.inte
raction)
                        …
                    }
                }
            }
}
```

If we run the application, we will see an output like the one in the following figure:

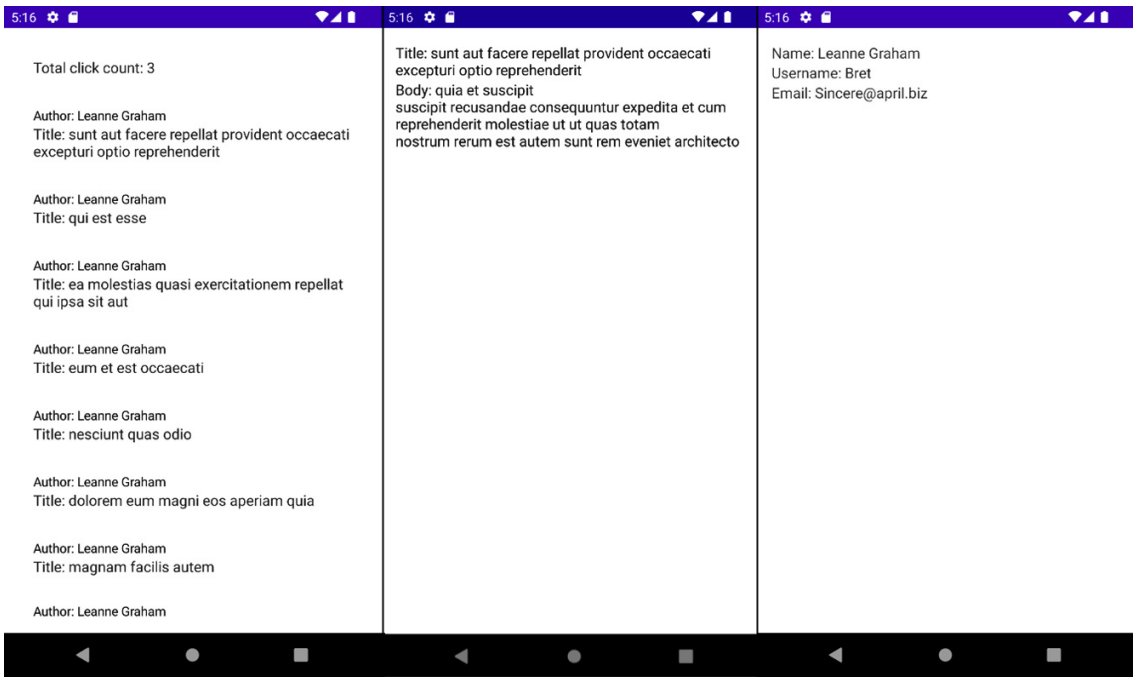

Figure 8.7 – Output of Exercise 08.02

We can see from *Figure 8.7* that when the row is clicked, we are taken to the screen displaying the post information, and when the author is clicked, we are taken to the user information. By placing the **`NavRoutes`** class in the **`presentation-common`** module, we can navigate from the post list on a screen located in the same module (post) and a screen located in a different module (user). The solution of creating additional modules is a good way to avoid cyclic dependencies not only for modules in the presentation layer but also for modules in the other layers as well.

In this exercise, we have learned how to split the presentation layer into separate modules and how we can use a common module to hold shared logic and data required by all the modules in the layer. This is a technique that can be used for other layers in the application if we want them split up as well.

Summary

In this chapter, we explored the presentation layer of an Android application and a few different approaches for implementing this layer, such as MVC, MVP, and MVVM. We decided to focus on the MVVM approach because of the many benefits involving the life cycle and the compatibility with Jetpack Compose. We then looked at what happens when we want to split the presentation layer across multiple modules and how we can solve the common logic between these modules. In the chapter that follows, we will further build upon the MVVM pattern and study the **Model-View-Intent** (**MVI**) pattern, which further takes advantage of the Observable pattern to incorporate the user actions into states that can be observed.