Chapter 9: Implementing an MVI Architecture

In this chapter, we will introduce the concept of **Model-View-Intent** (**MVI**) and the benefits it provides for managing the state of an application. We will begin by analyzing what MVI is and then move on to implementing it using Kotlin flows. In this chapter's exercise, we will build upon the previous chapter's exercises, and we will re-implement them using the MVI pattern to highlight how this pattern can be integrated into the presentation layer of an application with multiple modules.

In this chapter, we will cover the following topics:

- Introducing MVI
- Implementing MVI with Kotlin flows

By the end of the chapter, you will be able to implement the MVI architecture pattern inside a multimodule Android application, using Kotlin flows.

Technical requirements

The 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/Chapter9 ↗️.

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

Introducing MVI

In this section, we will look at what the MVI architecture pattern is, the problems it is trying to solve, and the solutions it presents for solving those problems.

Let's imagine you need to develop a configuration screen for an application. It will load the existing configuration and it will need to toggle various switches and prepopulate input fields with the existing data. After that data is loaded, then the user can modify each of those fields. To achieve this, you would probably need to keep mutable references for the data represented in those fields so that when the user changes a value, the reference changes.

This may pose a problem because of the mutability of those fields, especially when dealing with concurrent operations or their order. A solution to this problem is to make the data immutable and combine it into a state that the user interface can observe. Any changes the app or user will need to make on the user interfaces will be through a reactive data flow. The flow will then create a new state representing the change and update the user interface.

This is essentially how MVI operates. In MVI, the **View** plays the same role as in MVP or MVVM and the **Model** holds the state of the user interface, and it represents the single source of truth. The **Intent** is represented by any changes that should be made to the state, which will then be updated. In *Figure 9.1*, we can see how the **View** will send an **Intent** to the **Model**, which will then trigger a change in state, which will update the **View**:

NOTE

The term Intent in the context of MVI is different from the Android `Intent` *class used to interact with different Android components.*

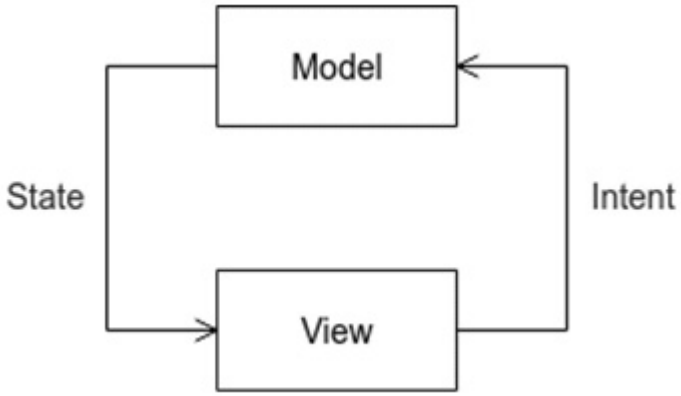

Figure 9.1 – MVI diagram

What is missing from this diagram is the equivalent of a **ViewModel** or a **Presenter**. This is because the MVI pattern isn't a replacement for those patterns but instead builds on top of them.

To visualize how this might look, let's look at an example of a **`ViewModel`**:

```
class MyViewModel @Inject constructor(
    private val getMyDataUseCase: GetMyDataUseCase
) : ViewModel() {
    private val _myDataFlow =
        MutableStateFlow<MyData>(MyData())
    val myDataFlow: StateFlow<MyData> = _myDataFlow
    var text: String = ""
    fun loadMyData() {
        viewModelScope.launch {
            getMyDataUseCase.execute
                (GetPostsWithUsersWithInteractionUseC
ase.
                    Request)
                .collect {
                    _myDataFlow.value = it
                }
        }
    }
}
```

In the preceding example, we define a class named **`MyViewModel`** in which we have a use case to load data and a **`text`** variable that will be changed by the View when the user changes it. We can see that the **`text`** variable is a mutable variable accessible from the **View**. We also have a **`StateFlow`** variable holding the data to be loaded and we have a method to load the data. To transition the preceding code to MVI, we will need to first define a state that will hold the data to be loaded and the text. This will represent our source of truth. For the preceding example, this state will look as in the following example:

```
data class MyState(
        val myData: MyData = MyData(),
        val text: String = ""
)
```

In the **`MyState`** class, we move the data to be loaded and the text to be changed. Now, we will need to identify the actions; in this case, we have two actions: loading the data and updating the value of the text with a new value introduced by the user:

```
    sealed class MyAction {
        object LoadAction : MyAction()
        data class UpdateAction(val text: String) :
            MyAction()
    }
```

In the preceding example, we have represented the action as a sealed class and defined two actions for loading and updating the text. Next, we will need to create the appropriate data flows for handling the actions and managing the state:

```
    private val _myStateFlow =
MutableStateFlow<MyState>
        (MyState())
    val myStateFlow: StateFlow<MyState> = _myDataFlow
    private val _actionFlow:
MutableSharedFlow<MyAction> =
```

```
            MutableSharedFlow()
```

In the preceding example, we have changed the **`StateFlow`** variables to hold the state object defined previously and added a similar **`SharedFlow`** variable, which will be responsible for managing the actions inserted by the user. We will now need to subscribe and handle the actions:

```
class MyViewModel @Inject constructor(
    private val getMyDataUseCase: GetMyDataUseCase
) : ViewModel() {
    …
    init {
        viewModelScope.launch {
            action.collect { action ->
                when (action) {
                    is
MyViewModel.MyAction.LoadAction -> {
                        loadMyData()
                    }
                    is
MyViewModel.MyAction.UpdateAction -> {
                        _myDataFlow.emit(_myDataFlow.
value.copy(text =
    action.text))
                    }
                }
            }
        }
    }
    fun submitAction(action: MyAction) {
        viewModelScope.launch {
            _action.emit(action)
        }
    }
    private fun loadMyData() {
        getMyDataUseCase.execute
```

```
                (GetPostsWithUsersWithInteractionUseCase.
                    Request)
                .collect {
                    _myDataFlow.value = it
                }
        }
        …
    }
```

In the **`init`** block, we are collecting the actions and then, for each action, we perform the required operation. The View will invoke the **`submitAction`** method and pass the action it wants the ViewModel to perform. For this example, **`MyAction`** plays the role of the Intent within the MVI context and the ViewModel will sit between the View and Model and will be responsible for managing the flow of data between the Model and the View, as well as managing the state.

When it comes to the implementation of the MVI pattern, there are many different variations for different technologies and different architecture patterns. From RxJava to **`LiveData`**, to flows and coroutines, to MVVM and MVP, there are different approaches to the pattern with different variations.

Some are built using concepts such as state machines, others use basic streams, and others use third-party open source libraries. From the preceding example, we can see that the pattern introduces a little bit of boilerplate code, so it is important to perform research and monitor the initial introduction of the pattern into any application. In the section that follows, we will look at how we can implement MVI using Kotlin flows.

Implementing MVI with Kotlin flows

In this section, we will look at how we can implement the MVI architecture pattern using Kotlin flows and the benefits and pitfalls of this

approach.

In the previous section, we defined an MVI approach using **`StateFlow`** and **`SharedFlow`**, as in the following example:

```
    private val _myStateFlow =
MutableStateFlow<MyState>(MyState())
    val myStateFlow: StateFlow<MyState> = _myDataFlow
    private val actionFlow:
MutableSharedFlow<MyAction> = MutableSharedFlow()
```

The different types of flows used here serve different purposes. **`MutableStateFlow`** will emit the last value held, which is good for the user interface because we want it to display the last data loaded, like how **`LiveData`** works. **`SharedFlow`** doesn't have this feature, which is useful for actions because we do not want the last action to be emitted twice. Another aspect we will need to consider is one-shot events, which should be emitted using a channel flow. This will be useful when the View will need to respond to events in a channel to display a toast alert or handle navigation to a new screen. We can apply this using the following:

```
class MyViewModel @Inject constructor(
    private val getMyDataUseCase: GetMyDataUseCase
) : ViewModel() {
    …
    private val _myStateFlow =
MutableStateFlow<MyState>
        (MyState())
    val myStateFlow: StateFlow<MyState> = _myDataFlow
    private val actionFlow:
MutableSharedFlow<MyAction> =
        MutableSharedFlow()
    private val _myOneOffFlow =
Channel<MyOneOffEvent>()
    val myOneOffFlow = _myOneOffFlow.receiveAsFlow()
    …
```

```
}
```

In the preceding example, we have integrated the **`Channel`** information with the rest of the **`ViewModel`**. Because an application will end up having multiple ViewModels, we can create a template that will be used across the application. We can start by defining abstractions for each of the state, action, and one-off events:

```
interface UiState
interface UiAction
interface UiSingleEvent
```

Here, we have opted for a simple interface to represent each of the flows of data the **`ViewModel`** will use. We can next define a template for the **`ViewModel`**, which can be inherited by the ViewModels used in the application:

```
abstract class MviViewModel<S : UiState, A :
UiAction, E : UiSingleEvent> : ViewModel() {
    private val _uiStateFlow: MutableStateFlow<S> by
lazy {
        MutableStateFlow(initState())
    }
    val uiStateFlow: StateFlow<S> = _uiStateFlow
    private val actionFlow: MutableSharedFlow<A> =
        MutableSharedFlow()
    private val _singleEventFlow = Channel<E>()
    val singleEventFlow =
_singleEventFlow.receiveAsFlow()
    …
}
```

In the preceding example, we have used generics for each of the flows that the **`ViewModel`** will use. This creates a problem for **`MutableStateFlow`**, which requires an initial value. Because we don't have any concrete value to initialize, we will need to create an abstract method that will provide the initial value:

```
abstract class MviViewModel<S : UiState, A :
UiAction, E : UiSingleEvent> : ViewModel() {
    …
    init {
        viewModelScope.launch {
            actionFlow.collect {
                handleAction(it)
            }
        }
    }
    abstract fun initState(): S
    abstract fun handleAction(action: A)
}
```

In addition to the **`initState`** abstraction, we have also added the **`handleAction`** abstraction. This will be called when new actions are submitted because of user actions or a screen load. Because the mutable variables are set to private, we will need to expose methods that emit events into these flows:

```
abstract class MviViewModel<S : UiState, A :
UiAction, E :
    UiSingleEvent> : ViewModel() {
    …
    fun submitAction(action: A) {
        viewModelScope.launch {
            actionFlow.emit(action)
        }
    }
    fun submitState(state: S) {
        viewModelScope.launch {
            _uiStateFlow.value = state
        }
    }
    fun submitSingleEvent(event: E) {
        viewModelScope.launch {
```

```
                _singleEventFlow.send(event)
            }
        }
    }
```

In the preceding example, we have added the methods that emit, send, or change the value on each of the specific data flows. To implement the template for a specific scenario, we will need to create concretions for **`UiState`**:

```
sealed class MyUiState : UiState {
    data class Success(val myData: MyData) :
MyUiState()
    object Error : MyUiState()
    object Loading : MyUiState()
}
```

In the preceding example, we have defined different states that the screen might have. We can now create a concretion for **`UiAction`**:

```
sealed class MyUiAction : UiAction {
    object Load : MyUiAction()
    object Click : MyUiAction()
}
```

Here, we defined an action for when the data will need to be loaded and another for when something is clicked on the user interface:

```
sealed class MyUiSingleEvent : UiSingleEvent {
    data class ShowToast(val text: String) :
        MyUiSingleEvent()
}
```

For the single event fired, we have defined a show toast alert event. Finally, we can implement the concretion for the **`ViewModel`**:

```
class MyViewModel : MviViewModel<MyUiState,
MyUiAction,
    MyUiSingleEvent>() {
```

```
    override fun initState(): MyUiState =
MyUiState.Loading
    override fun handleAction(action: MyUiAction) {
        when (action) {
            is MyUiAction.Load -> {
                viewModelScope.launch {
                    val state: UiState = // Fetch UI
state
                    submitState(state)
                }
            }
            is MyUiAction.Click -> {
                // Handle logic for clicks
                submitSingleEvent(MyUiSingleEvent.
                    ShowToast("Toast"))
            }
        }
    }
}
```

In the preceding example, we have extended the **`MviViewModel`** class and passed **`MyUiState`**, **`MyUiAction`**, and **`MyUiSingleEvent`** for the generics. In the **`initState`** method, we return the **`Loading`** state, and in the **`handleAction`** method, we check the actions and then load the data or handle the click event, which will then submit the event to show a toast alert.

If we want to integrate the **`ViewModel`** with Jetpack Compose, we will have to use something like the following example:

```
@Composable
fun MyScreen(
    viewModel: MyViewModel
) {
    viewModel.submitAction(MyUiAction.Load)
    viewModel.uiStateFlow.collectAsState().value.let
{
```

```
            state ->
            when (state) {
                is MyUiState.Loading -> {
                }
                is MyUiState.Success -> {
                    MySuccessScreen(state.myData) {
                        viewModel.submitAction(MyUiAction
.
                            Click)
                    }
                }
                is MyUiState.Error -> {
                }
            }
        }
    }
```

We can see that observing **`UiState`** will remain the same as for MVVM; however, if we wish to notify the **`ViewModel`** of any changes, we will need to use the **`submitAction`** method. For the **`UiSingleEvents`** object, we will need to use the **`LaunchedEffect`** function because we don't want Jetpack Compose to keep recomposing and re-executing the same block; we only want it to be executed once, so we will need to use something such as the following:

```
    @Composable
    fun MyScreen(
        viewModel: MyViewModel
    ) {
        …
        LaunchedEffect(Unit, {
            viewModel.singleEventFlow.collectLatest {
                when (it) {
                    is MyUiSingleEvent.ShowToast -> {
                        // Show Toast
                    }
```

```
                }
            }
        })
    }
```

In this example, we collect the data from **`Channel`** inside the **`LaunchedEffect`** method and then show a toast alert when the **`ShowToast`** event is received. **`LaunchedEffect`** can also be used to ensure that we do not trigger multiple data loads because of the Jetpack Compose recomposition mechanism:

```
@Composable
fun MyScreen(
    viewModel: MyViewModel
) {
    LaunchedEffect(Unit, {
        viewModel.submitAction(MyUiAction.Load)
    }
}
```

In the preceding snippet, we have moved the call to **`submitAction`** inside **`LaunchedEffect`**, to avoid triggering the loading multiple times. More information about Jetpack Compose side effects can be found here: https://developer.android.com/jetpack/compose/side-effects ↗️.

In this section, we have shown how we can integrate the MVI architecture pattern with flows and Jetpack Compose. We have seen how we have translated the interactions between the View and the **`ViewModel`** into intents using the **`UiAction`** interface and the implementations of this interface. We have also seen some of the downsides of the pattern because of the addition of boilerplate code and, in the case of Jetpack Compose, having to use methods such as **`LaunchedEffect`** and **`Channel`** for emitting one-off events. In the following section, we will create an application in which we will migrate a previous exercise to use MVI.

Exercise 09.01 – Transitioning to MVI

Modify *Exercise 08.02 – Multi-module data presentation* from *Chapter 8*, *Implementing an MVVM Architecture*, so that the presentation layer uses the MVI architecture pattern. The **`UiState`** class will remain and represent the state of each screen. New interfaces will be added in the **`presentation-common`** module representing actions and one-off events. In the same module, an **`MviViewModel`** abstract class will be implemented, which will be the template for the other ViewModels used in the application. For **`PostListViewModel`**, we will create new user actions for loading the data, clicking on the post, and clicking on the author, and two new one-off events will be needed for opening each of those screens. For **`PostViewModel`** and **`UserViewModel`**, we will create only a single user action, which will be responsible for loading the data on each screen.

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

1. In **`presentation-common`**, create an interface called **`UiAction`** and an interface called **`UiSingleEvent`**, and then create the **`MviViewModel`** template.
2. In the **`list`** package of the **`presentation-post`** module, create a sealed class called **`PostListUiAction`**, which will contain three subclasses called **`Load`**, **`UserClick`**, and **`PostClick`**. Then, create a sealed class called **`PostListUiSingleEvent`**, which will have two subclasses named **`OpenUserScreen`** and **`OpenPostScreen`**. Then, modify **`PostListViewModel`** and **`PostListScreen`** to use the specified actions and events.
3. In the **`single`** package of the **`presentation-post`** module, create a sealed class called **`PostUiAction`**, which will have one subclass named **`Load`**, which will contain the ID of the post. Then, modify **`PostViewModel`** and **`PostScreen`** to instead use the specified action.
4. In the **`single`** package of the **`presentation-user`** module, create a sealed class called **`UserUiAction`**, which will have one subclass named **`Load`**, which will contain the ID of the user. Then, modify **`UserViewModel`** and **`UserScreen`** to instead use the specified action.

Follow these steps to complete the exercise:

1. In the state package of the **`presentation-common`** module, create an interface called **`UiAction`**:
   ```
   interface UiAction
   ```
2. In the same package, create an interface called **`UiSingleEvent`**:
   ```
   interface UiSingleEvent
   ```
3. In the same package, create an abstract class called **`MviViewModel`**:
   ```
   abstract class MviViewModel<T : Any, S :
   UiState<T>, A : UiAction, E : UiSingleEvent> :
   ViewModel() {
   }
   ```

Because we are using the **`UiState`** class with generics, we will need to also supply that generic field in the generic specification of **`MviViewModel`**.

4. In the **`MviViewModel`** class, add the necessary flows and channels that will hold the states, actions, and events:
   ```
   abstract class MviViewModel<T : Any, S :
   UiState<T>, A : UiAction, E : UiSingleEvent> :
   ViewModel() {
       private val _uiStateFlow: MutableStateFlow<S>
   by
           lazy {
           MutableStateFlow(initState())
       }
       val uiStateFlow: StateFlow<S> = _uiStateFlow
       private val actionFlow: MutableSharedFlow<A> =
           MutableSharedFlow()
       private val _singleEventFlow = Channel<E>()
       val singleEventFlow = _singleEventFlow.
           receiveAsFlow()
   }
   ```

In this snippet, we have defined **`StateFlow`** variables to hold the last value that was emitted, which will be used to manage the state of the user interface, **`SharedFlow`**, which is used for handling user actions, and **`Channel`** for

handling emitting one-off events. In the **`MviViewModel`** class, we are also defining generics so that we bind states, actions, and one-off events to their respective types.

5. In **`MviViewModel`**, add the abstract methods for initializing the state and handling the actions:

```
abstract class MviViewModel<T : Any, S :
UiState<T>, A : UiAction, E : UiSingleEvent> :
ViewModel() {
    …
    init {
        viewModelScope.launch {
            actionFlow.collect {
                handleAction(it)
            }
        }
    }
    abstract fun initState(): S
    abstract fun handleAction(action: A)
}
```

In this snippet, we are adding the abstraction required to provide an initial value for **`StateFlow`**, and then we handle the collection of the user actions, which will be handled in the **`handleAction`** method.

6. In **`MviViewModel`**, add the required methods to submit the state, events, and actions:

```
abstract class MviViewModel<T : Any, S :
UiState<T>, A : UiAction, E : UiSingleEvent> :
ViewModel() {
    …
    fun submitAction(action: A) {
        viewModelScope.launch {
            actionFlow.emit(action)
        }
```

```
        }
        fun submitState(state: S) {
            viewModelScope.launch {
                _uiStateFlow.value = state
            }
        }
        fun submitSingleEvent(event: E) {
            viewModelScope.launch {
                _singleEventFlow.send(event)
            }
        }
    }
```

In this snippet, we are defining a set of methods to emit data into the two **`Flow`** objects and the **`Channel`** object.

7. In the **`list`** package of the **`presentation-post`** module, create the **`PostListUiAction`** class and its subclasses:

```
sealed class PostListUiAction : UiAction {
    object Load : PostListUiAction()
    data class UserClick(val userId: Long, val
        interaction: Interaction) :
PostListUiAction()
    data class PostClick(val postId: Long, val
        interaction: Interaction) :
PostListUiAction()
}
```

Here, we define a sealed class for loading the data and clicking on the user and the post. Each of them will implement the **`UiAction`** interface.

8. In the same package, create the **`PostListUiAction`** class and its subclasses:

```
sealed class PostListUiSingleEvent : UiSingleEvent
{
```

```
        data class OpenUserScreen(val navRoute: String)
    :
            PostListUiSingleEvent()
        data class OpenPostScreen(val navRoute: String)
    :
            PostListUiSingleEvent()
    }
```

Here, we define a sealed class for the one-off events that will be emitted when we want the user and post screens to be opened, which is why we are implementing **`UiSingleEvent`**.

9. In the same package, modify **`PostListViewModel`** to extend **`MviViewModel`**:

```
@HiltViewModel
class PostListViewModel @Inject constructor(
    private val useCase:
        GetPostsWithUsersWithInteractionUseCase,
    private val converter: PostListConverter,
    private val updateInteractionUseCase:
        UpdateInteractionUseCase
) : MviViewModel<PostListModel,
UiState<PostListModel>
    , PostListUiAction, PostListUiSingleEvent>() {
     …
}
```

In this snippet, we are extending **`MviViewModel`** and providing the types we defined previously as well as the existing **`PostListModel`** type to the generic fields. This is because we want this **`ViewModel`** to be bound to the data, actions, and one-off events that occur in **`PostListScreen`**.

10. Implement the **`initState`** method in the **`PostListViewModel`** class:

```
@HiltViewModel
class PostListViewModel @Inject constructor(
```

```
    …
) : MviViewModel<PostListModel,
UiState<PostListModel>
    , PostListUiAction, PostListUiSingleEvent>() {
    override fun initState():
UiState<PostListModel> =
        UiState.Loading
}
```

In this snippet, we are implementing the **`initState`** method and providing the **`UiState.Loading`** value, which will in turn make the **`uiStateFlow`** field from the parent class be initialized with the **`Loading`** value.

11. Implement the **`handleAction`** method in the **`PostListViewModel`** class:

```
@HiltViewModel
class PostListViewModel @Inject constructor(
    …
) : MviViewModel<PostListModel,
UiState<PostListModel>
    , PostListUiAction, PostListUiSingleEvent>() {
    …
    override fun handleAction(action:
        PostListUiAction) {
        when (action) {
            is PostListUiAction.Load -> {
                loadPosts()
            }
            is PostListUiAction.PostClick -> {
                updateInteraction(action.interactio
n)
                submitSingleEvent(
                    PostListUiSingleEvent.
                        OpenPostScreen(
                        NavRoutes.Post.routeForPost
(
```

```
                                PostInput(action.postId
    )
                            )
                        )
                    )
                }
                is PostListUiAction.UserClick -> {
                    updateInteraction(action.interactio
    n)
                    submitSingleEvent(
                        PostListUiSingleEvent.
                            OpenUserScreen(
                            NavRoutes.User.routeForUser
    (
                                UserInput(action.userId
    )
                            )
                        )
                    )
                }
            }
        }
    }
```

In this snippet, we are implementing the **`handleAction`** method, which will check what action we will need to handle and perform the necessary operation for each. For loading, we will invoke the **`loadPosts`** method, and for clicking on a user and a post, we will invoke the **`updateInteraction`** method and then submit a one-off event to open the user and post screens.

12. Implement the **`loadPosts`** method in the **`PostListViewModel`** class:

```
@HiltViewModel
class PostListViewModel @Inject constructor(
    …
```

```
) : MviViewModel<PostListModel,
UiState<PostListModel>
    , PostListUiAction, PostListUiSingleEvent>() {
    …
    private fun loadPosts() {
        viewModelScope.launch {
            useCase.execute
            (GetPostsWithUsersWithInteractionUseCas
e.
            Request)
                .map {
                    converter.convert(it)
                }
                .collect {
                    submitState(it)
                }
        }
    }
}
```

In this snippet, we load the data from **`GetPostsWithUsersWithInteractionUseCase`** and collect it and update **`uiStateFlow`** through the **`submitState`** method inherited from the parent class.

13. Implement the **`updateInteraction`** method in the **`PostListViewModel`** class:

```
@HiltViewModel
class PostListViewModel @Inject constructor(
    …
) : MviViewModel<PostListModel,
UiState<PostListModel>
    , PostListUiAction, PostListUiSingleEvent>() {
    …
    private fun updateInteraction(interaction:
        Interaction) {
```

```
            viewModelScope.launch {
                updateInteractionUseCase.execute(
                    UpdateInteractionUseCase.Request(
                        interaction.copy(
                            totalClicks = interaction.
                                totalClicks + 1
                        )
                    )
                ).collect()
            }
        }
    }
```

In this method, we implement the **`updateInteraction`** method, which will submit a new value with an incremented click count using **`UpdateInteractionUseCase`**.

14. Modify the **`PostListScreen`** method in the **`PostListScreen`** file in the **`list`** package in the **`presentation-post`** module so that it will instead use the **`submitAction`** method:

```
@Composable
fun PostListScreen(
    viewModel: PostListViewModel,
    navController: NavController
) {
    LaunchedEffect(Unit) {
        viewModel.submitAction(PostListUiAction.Loa
d)
    }
    viewModel.uiStateFlow.collectAsState().value.le
t {
        state ->
        CommonScreen(state = state) {
            PostList(it, { postListItem ->
                viewModel.submitAction
```

```
                        (PostListUiAction.PostClick
                        (postListItem.id,
    it.interaction))
                }) { postListItem ->
                    viewModel.submitAction
                        (PostListUiAction.UserClick
                        (postListItem.id,
    it.interaction))
                }
            }
        }
    }
```

Here, we are changing how we interact with **`PostListViewModel`**. Instead of invoking each separate method for loading and updating the interaction, we instead use the **`submitAction`** method from **`MviViewModel`**. In order to load the data, we are using **`LaunchedEffect`** so that when Jetpack Compose triggers recomposition, the data load won't be retriggered. We are also subscribing to **`uiStateFlow`** instead of **`postListFlow`**, which no longer exists.

15. In the same method, subscribe to **`singleEventFlow`** so that it opens **`PostScreen`** and **`UserScreen`** when the appropriate events are received:

```
@Composable
fun PostListScreen(
    viewModel: PostListViewModel,
    navController: NavController
) {
    …
    LaunchedEffect(Unit) {
        viewModel.singleEventFlow.collectLatest {
            when (it) {
                is PostListUiSingleEvent.
                    OpenPostScreen ->
{    navController.navigate
```

```
                        (it.navRoute)
                    }
                    is PostListUiSingleEvent.
                        OpenUserScreen -> {
                        navController.navigate
                        (it.navRoute)
                    }
                }
            }
        }
    }
```

In this snippet, we will need to monitor the events from **`singleEventFlow`** and then check the events emitted and open the appropriate screen.

16. In the **`single`** package of the **`presentation-post`** module, create the **`PostUiAction`** class and its subclass:

```
sealed class PostUiAction : UiAction {
    data class Load(val postId: Long) :
PostUiAction()
}
```

17. In the same package, modify **`PostViewModel`** so that it extends **`MviViewModel`**:

```
@HiltViewModel
class PostViewModel @Inject constructor(
    private val postUseCase: GetPostUseCase,
    private val postConverter: PostConverter
) : MviViewModel<PostModel, UiState<PostModel>,
    PostUiAction, UiSingleEvent>() {
}
```

Here, we are using the newly created **`PostUiAction`**, but because we have no one-off events to subscribe to, we will use the **`UiSingleEvent`** interface.

18. In the same class, implement the **`initState`** and **`handleAction`** methods:

```
@HiltViewModel
class PostViewModel @Inject constructor(
    …
) : MviViewModel<PostModel, UiState<PostModel>,
    PostUiAction, UiSingleEvent>() {
    override fun initState(): UiState<PostModel> =
        UiState.Loading
    override fun handleAction(action: PostUiAction)
{
        when (action) {
            is PostUiAction.Load -> {
                loadPost(action.postId)
            }
        }
    }
    private fun loadPost(postId: Long) {
        viewModelScope.launch {
            postUseCase.execute
                (GetPostUseCase.Request(postId))
                .map {
                    postConverter.convert(it)
                }
                .collect {
                    submitState(it)
                }
        }
    }
}
```

Here, we are implementing the **`initState`** method and returning the **`UiState.Loading`** value and the **`handleAction`** method. For **`handleAction`**, we only have the action to load the data, which will use **`GetPostUseCase`** to retrieve the post data and then update **`uiStateFlow`** through the **`submitState`** method.

19. Modify the **`PostScreen`** method from the **`PostScreen`** file in the **`single`** package in the **`presentation-post`** module so that it instead uses the **`Load`** action:

```
@Composable
fun PostScreen(
    viewModel: PostViewModel,
    postInput: PostInput
) {
    viewModel.uiStateFlow.collectAsState().value.le
t {
        result ->
        CommonScreen(result) { postModel ->
            Post(postModel)
        }
    }
    LaunchedEffect(postInput.postId) {
        viewModel.submitAction(PostUiAction.
            Load(postInput.postId))
    }
}
```

In this snippet, we are following the same principle as in **`PostListScreen`** where we replace the interaction with **`PostViewModel`** to use the **`submitAction`** method and use **`LaunchedEffect`** to isolate the data loading.

20. In the **`single`** package of the **`presentation-user`** module, create the **`UserUiAction`** class and its subclass:

```
sealed class UserUiAction : UiAction {
    data class Load(val userId: Long) :
UserUiAction()
}
```

21. In the same package, modify **`UserViewModel`** so that it extends the **`MviViewModel`** class:

```
@HiltViewModel
class UserViewModel @Inject constructor(
```

```
    private val userUseCase: GetUserUseCase,
    private val converter: UserConverter
) : MviViewModel<UserModel, UiState<UserModel>,
    UserUiAction, UiSingleEvent>() {
}
```

Here, we are using the newly created **`UserUiAction`**, but because we have no one-off events to subscribe to, we will use the **`UiSingleEvent`** interface.

22. In the same class, implement the **`initState`** and **`handleAction`** methods:

```
@HiltViewModel
class UserViewModel @Inject constructor(
    …
) : MviViewModel<UserModel, UiState<UserModel>,
    UserUiAction, UiSingleEvent>() {
    override fun initState(): UiState<UserModel> =
        UiState.Loading
    override fun handleAction(action: UserUiAction)
{
        when (action) {
            is UserUiAction.Load -> {
                loadUser(action.userId)
            }
        }
    }
    private fun loadUser(userId: Long) {
        viewModelScope.launch {
            userUseCase.execute
                (GetUserUseCase.Request(userId))
                .map {
                    converter.convert(it)
                }
                .collect {
                    submitState(it)
                }
```

```
            }
        }
    }
```

Here, we are following the same principle as for **`PostViewModel`**, which is to implement the **`initState`** method to return **`UiState.Loading`**, then in **`handleAction`**, we check the type, and for the **`Load`** action, we load the user information.

23. Modify the **`UserScreen`** method from the **`UserScreen`** file in the **`single`** package in the **`presentation-user`** module so that it instead uses the **`Load`** action:

```
@Composable
fun UserScreen(
    viewModel: UserViewModel,
    userInput: UserInput
) {
    viewModel.uiStateFlow.collectAsState().value.le
t {
        result ->
        CommonScreen(result) { userModel ->
            User(userModel)
        }
    }
    LaunchedEffect(userInput.userId) {
        viewModel.submitAction(UserUiAction.
            Load(userInput.userId))
    }
}
```

In this snippet, we are following the same principle as in **`PostScreen`** where we replace the interaction with **`UserViewModel`** to use the **`submitAction`** method and use **`LaunchedEffect`** to isolate the data loading.

If we run the application, we will see the same output as in *Exercise 08.02 – Multi-module data presentation*:

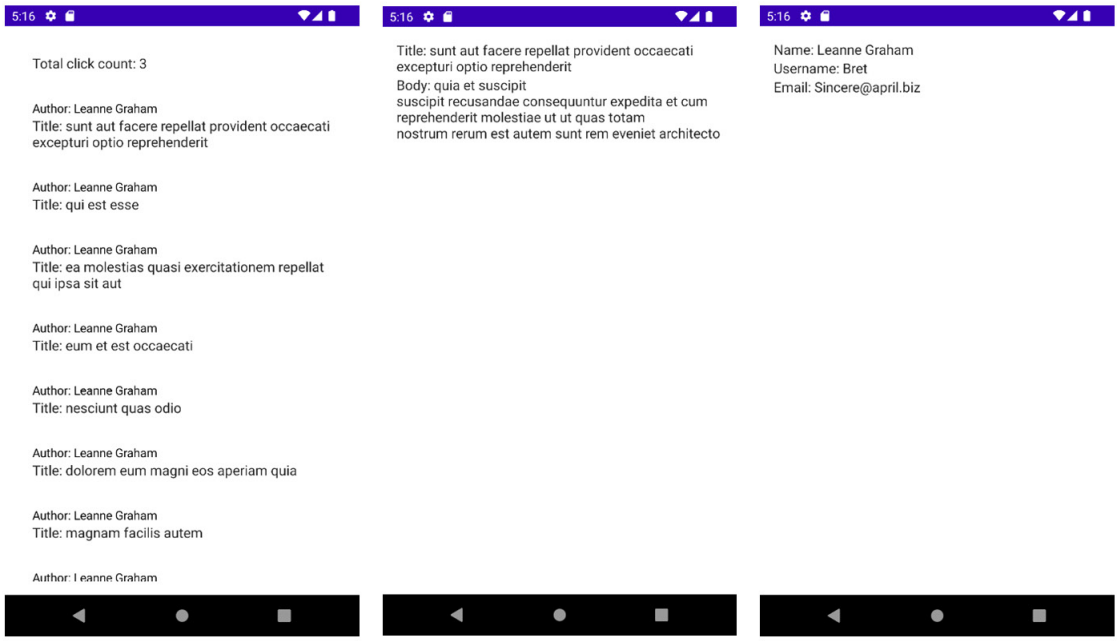

Figure 9.2 – Output of Exercise 09.01

After introducing MVI into the exercise, we can see that we already had the groundwork because of how Jetpack Compose requires states to manage the user interface. This represents one of the reasons we ended up creating the **`UiState`** class in previous chapters. We have also observed the downsides of the pattern through the addition of boilerplate code and the handling of one-off events, the latter not being limited to MVI. The use of **`MviViewModel`** shows how we can have the same template across different modules of the presentation layer.

From a Clean Architecture perspective, we can see that the changes we have done in our presentation layer haven't affected the rest of the layers of the application, which is a sign that we are going down the right path.

Summary

In this chapter, we studied the MVI architecture pattern and the benefits it provides to applications using reactive streams of data, by centralizing

user and application actions into a unidirectional flow of data.

We then looked at how we can implement this pattern using Kotlin flows and the role it plays when combined with other patterns, such as MVP and MVVM, with a focus on MVVM. We can observe the downsides of the pattern on simple presentations, but its benefits become more visible in applications with complicated user interfaces that take in multiple user inputs, which can change the states of other inputs. In the chapter's exercise, we looked at how we can transition an application with MVVM to MVI and how it fits into Clean Architecture.

In the next chapter, we will take a step back and look at what we have implemented and studied so far. We will see what we can improve and how we can take advantage of the different layers of the application, as well as how we can swap dependencies for the various configurations an application might have.