Chapter 7: Introducing Presentation Patterns in Android

In this chapter, we're continuing our journey of exploring ways to architect Android applications. More precisely, we will be making sure that our applications split responsibilities correctly with the introduction of presentation patterns.

In the first section, *Introducing MVC, MVP, and MVVM as presentation patterns*, we will provide a short overview on why we need presentation patterns, and we will explore how most common patterns are implemented in Android projects.

Next up, in the *Refactoring our Restaurants App to fit a presentation pattern* section, we will refactor our Restaurants App to fit the MVVM presentation pattern, while also understanding why MVVM is best suited for our Compose-based app.

In the last section, *Improving state encapsulation in ViewModel*, we will see why it's important for the **user interface** (**UI**) state to be properly encapsulated inside the `ViewModel`, and we will explore how to achieve that.

To summarize, in this chapter, we're going to cover the following main topics:

- Introducing **Model-View-Controller** (**MVC**), **Model-View-Presenter** (**MVP**), and **Model-View-ViewModel** (**MVVM**) as presentation patterns
- Refactoring our Restaurants app to a presentation pattern
- Improving state encapsulation in ViewModel

Before jumping in, let's set up the technical requirements for this chapter.

Technical requirements

Building Compose-based Android projects for this chapter usually requires your day-to-day tools. However, to follow along smoothly, make sure you have the following:

- The Arctic Fox 2020.3.1 version of Android Studio. You can also use a newer Android Studio version or even Canary builds, but note that the IDE interface and other generated code files might differ from the ones used throughout this book.
- Kotlin 1.6.10 or newer plugin installed in Android Studio.
- The Restaurants app code from the previous chapter.

The starting point for this chapter is represented by the Restaurants app developed in the previous chapter. If you haven't followed the implementation from the previous chapter, access the starting point for this chapter by navigating to the **`Chapter_06`** directory of the repository and importing the Android project entitled **`chapter_6_restaurants_app`**.

To access the solution code for this chapter, navigate to the **`Chapter_07`** directory at https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_07/chapter_7_restaurants_app.

Introducing MVC, MVP, and MVVM as presentation patterns

In the beginning, most Android projects were designed as a bunch of **`Activity`** or **`Fragment`** classes that were setting content to their corresponding **Extensible Markup Language** (**XML**) layouts.

As projects grew and new features were requested, developers had to add more logic inside the **`Activity`** or **`Fragment`** class, development cycle after

development cycle. This means that anything from a new feature, improvement, or bug fix for a particular screen would have to be done inside those `Activity` or `Fragment` classes.

After some time, these classes became larger and larger, and at some point, adding an improvement or fixing a bug could become a nightmare. The reason for this would be that the `Activity` or `Fragment` classes were burdened with all the responsibilities from within a particular project. These classes would be doing the following:

- Defining the UI
- Preparing the data to be displayed and defining different UI states
- Obtaining data from different sources
- Applying different business rules to data

This approach introduces coupling between distinct responsibilities and concerns of a project. For such projects, if—for example—a portion of the UI must be changed, your changes could easily impact other concerns of the app: the way data is presented, the logic of obtaining that data, business rules, and so on.

The worst part of this happening is that when you need to change only a part (say, part of the UI) and you end up changing other parts (say, the presentation, or data logic) you risk breaking unrelated things that worked, therefore possibly introducing new bugs.

Having such an approach where all the code of a project is bundled inside the `Activity` or `Fragment` class, causes your project to develop the following issues:

- **Fragile and difficult to scale**: Adding new features or improvements can break other parts of your app.
- **Difficult to test**: Since all the logic of the app is bundled in one place, testing only one part of the logic is very difficult because all your logic is tangled and tied to platform-related dependencies.

- **Difficult to debug**: When responsibilities are intertwined, then parts of your code base are also intertwined and coupled. Debugging one specific issue becomes extremely difficult because it's hard to track the exact culprit.

To alleviate these issues, we can try to identify the core responsibilities of an app and then separate their corresponding logic and code into distinct components (or classes) that are part of specific layers. This way, we are trying to follow the principle of **separation of concerns** (**SoC**), whereby each layer will contain classes whose responsibilities are tightly related only to their corresponding layer's concern.

To make sure that our projects obey the SoC principle, we can split the app's responsibilities into two major ones and define a layer for each of them, as follows:

- The **Presentation layer** contains classes (or other components) responsible for defining the UI and preparing the data to be presented.
- The **Model layer** contains classes where the application's data is obtained, modeled, and updated.

Even though the two layers seem to do more than one thing, all these actions define a broader dedicated responsibility that encapsulates a specific concern.

In this chapter, while we will be mostly focusing on structuring the Presentation layer, we will also start working on the Model layer. We will continue refactoring the Model layer in *Chapter 8*, *Getting Started with Clean Architecture in Android*.

To separate concerns within the Presentation layer, you can make use of presentation design patterns. **Presentation design patterns** are architectural patterns that define how the Presentation layer is structured in our applications.

The Presentation layer is a part of our project that is tied to what the user sees: the UI and the presentation of that UI. In other words, the Presentation layer handles two granular, yet related responsibilities associated with two types of logic, as outlined here:

- **UI logic**: Defines the ability to display content on a device in a specific way for one screen or flow. For example, when building an XML layout or a composable hierarchy for a screen, we're defining the UI logic for that specific screen since we're defining its UI elements.
- **Presentation logic**: The logic that defines the state of the UI (for one screen or flow) and how it mutates when the user interacts with our UI, therefore defining how the data is being presented to the UI. We're writing presentation logic when, for example, we must do the following:
  - Ensure that the screen is in a loading state or error state at specific times
  - Present content for a screen in a specific manner by formatting it to some standards

For the Presentation layer to define UI logic and presentation logic, it needs some data to work with. That's why it must be connected to the Model layer, which provides it with raw data, be it from web services, local databases, or other sources. You can see an illustration of this in the following diagram:

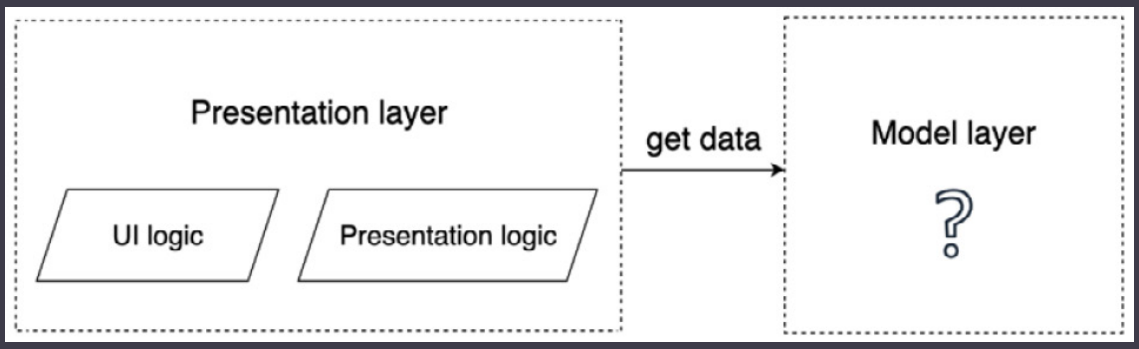

Figure 7.1 – Composition of the Presentation layer and its relation to the Model layer

For now, we will consider the Model layer a black box that just provides us with data.

We can say that such separations allow the UI to become a representation of the model's data through transformations that happen inside the Presentation layer while having components whose responsibilities don't overlap.

In Android, transformations from within the Presentation layer are modeled through three popular presentation patterns that are used in other technology stacks as well, as follows:

- MVC
- MVP
- MVVM
  NOTE
  As Android developers, we have adjusted the implementation of these presentation patterns to the specific needs of Android. That's why the way we will exemplify or implement them may vary from their original definitions given by their founders—all this is in pursuit of observing their common usages in Android projects.

These presentation patterns will allow us to separate UI logic from presentation logic for each screen or flow within our app. By doing so, we are ensuring that our Presentation layer has less coupled code that is easier to maintain, easier to scale with new features, and easier to test.

Historically, most Android projects have transitioned from MVC to MVP and, nowadays, to MVVM. Regardless of their structure, though, it's important to mention that the SoC promoted by these presentation patterns often translates into each UI flow being broken into classes or components that are instructed to do something specific, tightly related to their responsibility.

To see what I'm talking about, let's briefly cover them, starting out with MVC.

MVC

A common implementation in Android projects of the MVC pattern defines its layers like so:

- **View**: Views inflated from the XML layouts as a representation of the UI. This layer would only be rendering the content it receives from the Controller onto the screen.
- **Controller**: UI controllers such as `Activity` or `Fragment`. This component would define the state of the UI by preparing data received from the Model layer for presentation, or by intercepting UI events that in turn would mutate the state. Additionally, the Controller would be in charge of setting actual data to the View layer.
- **Model**: The entry point of data. The actual structure doesn't depend on MVC, but we can think of it as the layer that obtains content needed by the Presentation layer, by querying a local database or remote sources such as web **application programming interfaces** (**APIs**).

Let's visualize the actual separation brought by this pattern, as follows:

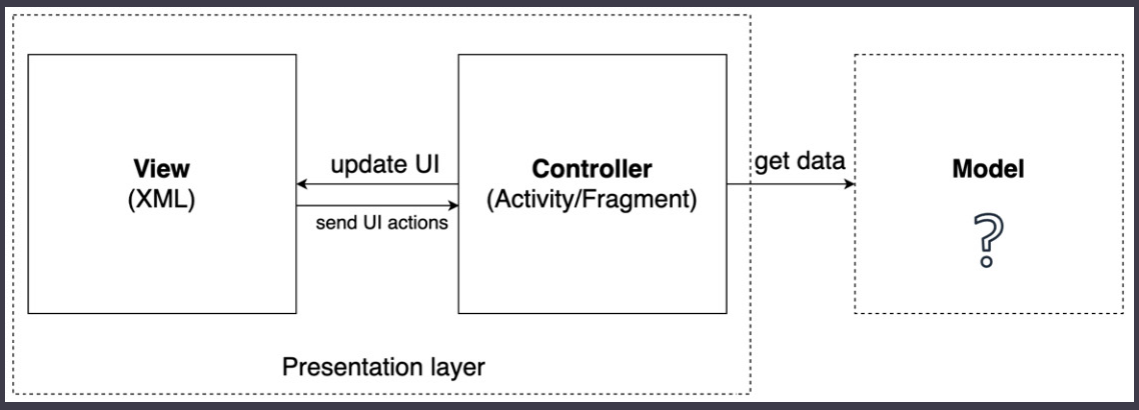

Figure 7.2 – Presentation layer in the MVC pattern

This implementation of the MVC pattern achieves a proper separation between the Presentation layer and the Model layer, therefore liberating `Activity` and `Fragment` controllers from being the ones that obtain data

from **REpresentational State Transfer** (**REST**) APIs or local databases. Yet at least in this form factor, MVC doesn't shine where it should because the actual separation within the Presentation layer could be improved.

Disadvantages of this pattern may include the following:

- High coupling between the Controller (`Activity` or `Fragment` controllers) and the View layer. Since the Controller is a component with a lifecycle and it also must provide the infrastructure of building and setting up Android views with content (such as building `Adapter` classes and passing data), testing it becomes difficult because it's tightly coupled with Android APIs.
- The Controller has two responsibilities: it handles the state of the UI (presentation logic) while also providing infrastructure for the View layer to function (UI logic). The two responsibilities become tangled up—when testing one, you would be testing the other too.

Let's move on to another popular presentation pattern in Android.

MVP

A common implementation in Android projects of the MVP pattern defines its layers like so:

- **View**: The UI layer defined by the `Activity` or `Fragment` class and their corresponding inflated views from XML. This layer now encapsulates the entire UI logic: it provides the infrastructure of building and setting up rendered Android views with content.
- **Presenter**: Presents data to the UI by manipulating the View layer indirectly through an interface. With this approach, a one-to-one relationship between a Presenter and a View layer (be it `Activity` or `Fragment`) is established. The interface allows the Presenter to pass data that is ready for presentation to the UI layer and to directly mutate the UI state at the UI level.

Unlike the Controller in MVC, the Presenter is no longer coupled to lifecycle components or Android View APIs, so it becomes much easier to test the presentation logic that it contains.

- **Model**: The same as in MVC.

Let's visualize the actual separation brought by this pattern, as follows:

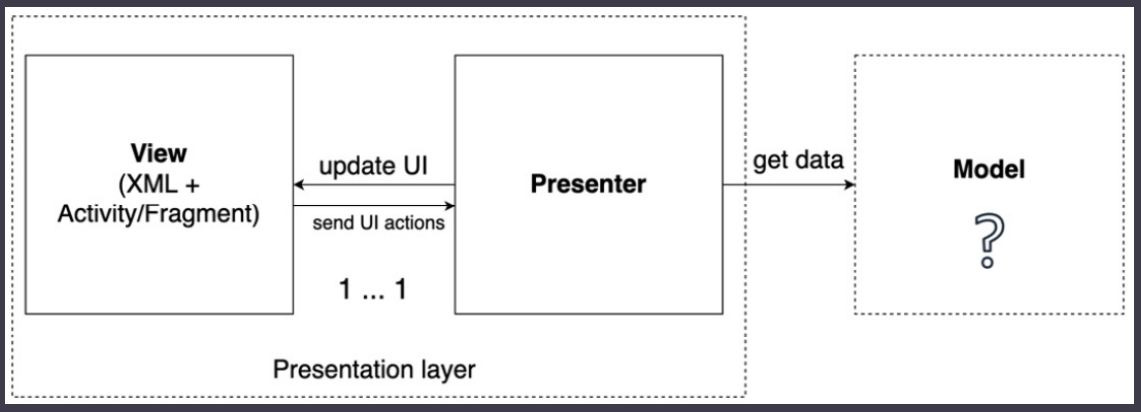

Figure 7.3 – Presentation layer in the MVP pattern

Unlike MVC, **`Activity`** and **`Fragment`** are now part of the View layer, which seems more natural because they are both tightly related to the Android UI. This approach allows the Presenter to be the one that prepares data that must be presented, while imperatively mutating the UI.

Since we now have a separate entity that is in charge of presenting data to the UI, we can say that, unlike MVC, MVP performs the SoC inside the Presentation layer somewhat better.

However, there are still some issues with this approach, as outlined here:

- The imperative approach of having the Presenter manually update the UI directly in the **`Activity`** or **`Fragment`** class can be prone to bugs and can cause illegal UI states (such as showing an error message and a loading status at the same time) as a project grows and new features are added. This is similar to how a UI controller (such as **`Activity`**) also imperatively mutates XML views—an approach that we deemed as

prone to issues when we introduced Compose with its declarative paradigm.
- If the interface contract between the Presenter and the View layer is not well designed or is missing entirely, the two would become coupled, and reusing the Presenter for other `Activity` or `Fragment` controllers might be difficult.

Let's move on to another important presentation pattern.

MVVM

MVVM is a very popular presentation pattern in Android, mostly because it addresses the concerns stated with the previously mentioned implementation of MVP.

A common implementation in Android projects of MVVM defines its layers like so:

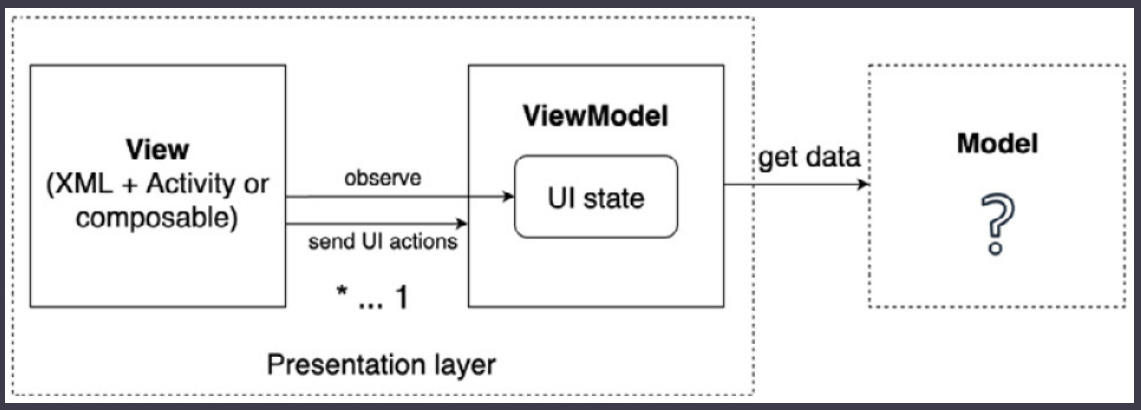

Figure 7.4 – Presentation layer in the MVVM pattern

Let's look at how the layers are defined:

- **View**: The UI layer is the `Activity` or `Fragment` class and its XML views, just as in MVP. Unlike in MVP, though, the View layer observes either an observable state or observable fields from the `ViewModel`, both containing UI data. Whenever new updates are received from those observable entities, the View layer updates the UI with the content received.

- **ViewModel**: This prepares the data received from the Model layer, just as the Presenter in MVP did. Unlike the Presenter, though, the `ViewModel` defines the UI state as an observable property (or multiple observable fields) and is totally decoupled from the View layer as it has no reference to it.
- **Model**: The same as in MVC or MVP.

One advantage of the `ViewModel`, as opposed to the Presenter in MVP, is that it's no longer coupled to the View layer, so it can be reused much more easily. In contrast with MVP, the View layer is responsible for referencing the `ViewModel` for obtaining and observing the observable state, and so the `ViewModel` no longer needs to reference the View layer, becoming totally independent.

In other words, the `ViewModel` in MVVM forces the View layer to subscribe to data, which is different from MVP, where the Presenter was manually setting up the View layer with data. This approach allows multiple Views to bind to the same `ViewModel`, therefore *sharing* the same UI state within the same `ViewModel`.

Another advantage is that since the View layer observes the UI state from the `ViewModel` and binds the received data as an effect, the `ViewModel` doesn't imperatively update the UI as the Presenter did through an interface in MVP. In other words, the View layer obtains the UI state from the `ViewModel` and binds it to the UI—this results in a unidirectional flow of data that is less likely to introduce bugs or illegal states.

NOTE

While considering the original definition of MVVM, the ViewModel shouldn't be confused with the Jetpack ViewModel component—the ViewModel can be a simple class that presents the data through an observable state. For us on Android, though, it's convenient to consider the Jetpack ViewModel as the actual ViewModel from MVVM because it brings some advantages out of the box.

However, the pattern's implementation that is commonly used in Android considers the Jetpack ViewModel as the ViewModel from MVVM, and this brings both a set of advantages and disadvantages.

Using the Jetpack ViewModel as the **`ViewModel`** from MVVM is beneficial for the following reasons:

- The Jetpack ViewModel is scoped to the lifetime of the View and provides convenient APIs for canceling work such as the **`onCleared()`** callback or the **`viewModelScope`** coroutine scope, therefore providing a convenient API for canceling asynchronous jobs and minimizing the risk of memory leaks.
- The Jetpack ViewModel survives configuration changes, therefore allowing you to preserve the UI state automatically if the user changes the orientation of the device, for example.
- You can easily restore the UI state after system-initiated process death because the Jetpack ViewModel is providing us with a **`SavedStateHandle`** object.

Unfortunately, this approach comes with the following downsides:

- The **`ViewModel`** is now a library dependency (the Jetpack ViewModel) that introduces coupling with the Android platform (as it exposes APIs such as **`SavedStateHandle`**). This prevents us from reusing presentation components for cross-platform projects with **Kotlin Multiplatform** (**KMP**).
- Because the Jetpack ViewModel is a library dependency that handles other responsibilities apart from data presentation, such as restoring the UI state after system-initiated process death, we could argue that the Presentation layer concerns are not very well separated.

Now that we have had a quick overview of presentation patterns, it's time for a practical example.

Refactoring our Restaurants app to fit a presentation pattern

We plan to refactor our Restaurants app to fit a presentation pattern. From our previous comparison, we can consider that MVVM is best suited for our Compose-based app. Don't worry—we will talk about this decision in more detail a bit later.

But before we do that, let's add more functionality inside the application to better highlight how mingling responsibilities can lead to unmaintainable code.

To summarize, in this section, we're going to be doing the following:

- Adding more functionality inside our Restaurants app
- Refactoring our Restaurants app to MVVM

Let's begin!

Adding more functionality inside our Restaurants app

When the Restaurants application is launched, the `RestaurantsScreen()` composable is rendered. Inside this screen, we are loading a bunch of restaurants from the server, and then we're displaying them to the user.

Yet while our app waits for the network request to finish and for the local caching to Room to happen (in order for it to receive restaurants for the UI), the screen remains blank, and the user has no idea what's going on. To provide a better **user experience** (**UX**), we should somehow suggest to the user the fact that we're waiting for content from the server.

We could do that through a loading progress bar! Inside the `RestaurantsScreen()` composable, we could add a loading UI element that

is displayed until the **`LazyColumn`** composable that renders a list of restaurants is populated. When the content arrives, we should hide it, thereby letting the user know that the application has loaded its content.

Let's do that right now, as follows:

1. First, inside the **`RestaurantsScreen()`** composable, save the restaurant list from the state (retrieved from **`RestaurantsViewModel`**) inside a **`restaurants`** variable, like this:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) ->
Unit) {
    val viewModel: RestaurantsViewModel =
viewModel()
    val restaurants = viewModel.state.value
    LazyColumn(…){
        items(restaurants) { restaurant ->
            RestaurantItem(…)
        }
    }
}
```

Make sure to also pass the **`restaurants`** variable to the **`LazyColumn`** composable's **`items`** **domain-specific language** (**DSL**) function.

2. We need to define a condition that lets us know when to show a loading indicator. As a first attempt, we could say that when the **`restaurants`** variable contains an empty **`List<Restaurant>`** as a value, which means that restaurants haven't arrived yet, the content is still loading. Add an **`isLoading`** variable that accounts for this, as follows:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) ->
Unit) {
    val viewModel: RestaurantsViewModel =
viewModel()
```

```
    val restaurants = viewModel.state.value
    val isLoading = restaurants.isEmpty()
    LazyColumn(…){ … }
}
```

If, however, restaurants arrive from the server, the **`state`** variable is updated, and the **`restaurants`** variable no longer contains an empty list of restaurants. At this point, the **`isLoading`** variable becomes **`false`**.

3. We want to display a loading indicator while the **`isLoading`** variable is **`true`**. To do that, wrap the **`LazyColumn`** composable in a **`Box`** composable, and below the **`LazyColumn`** code, check if the **`isLoading`** variable is **`true`** and pass a **`CircularProgressIndicator`** composable. The code is illustrated in the following snippet:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) ->
Unit) {
    …
    val isLoading = restaurants.isEmpty()
    Box() {
        LazyColumn(…){…}
        if(isLoading)
            CircularProgressIndicator()
    }
}
```

The **`Box`** composable allows us to overlay two composables: **`LazyColumn`** and **`CircularProgressIndicator`**. Because of the **`if`** condition that we've added, we now have the following two cases:

- **`isLoading`** is **`true`** (the app is waiting for restaurants), so both composables are composed. While the **`CircularProgressIndicator`** composable is displayed on top of the **`LazyColumn`** composable, the **`LazyColumn`** composable contains no elements, so it's not visible.

- **`isLoading`** is **`false`** (the app now has restaurants to display), so only the **`LazyColumn`** composable is composed and visible.

4. To center the **`CircularProgressIndicator`** composable, add the **`Alignment.Center`** alignment to the **`contentAlignment`** parameter of the **`Box`** composable, while also passing a **`Modifier.fillMaxSize()`** modifier. The code is illustrated in the following snippet:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) ->
Unit) {
    …
    Box(contentAlignment = Alignment.Center,
        modifier = Modifier.fillMaxSize()) {
        …
    }
}
```

5. Build and run the app. For a moment (until restaurants are loaded), you should see a loading progress indicator. When restaurants are displayed, this should go away.

Inside the UI layer, we have now added a loading indicator as well as the logic that decides when to display it. In this simple scenario, our logic works, but what happens if the server (or local database) returns an empty list of restaurants? Then the loading indicator will never go away.

Or, what happens if an error occurs? Our **`RestaurantsScreen`** composable has no idea that an error was generated. This means that not only does it not know when to display the error, but it also doesn't know when to hide the loading indicator if such an error were to occur.

These issues arise from the fact that we're trying to define presentation logic (when to show or hide a loading indicator; when to show an error message) inside the UI layer (where composables reside), thereby mixing UI logic with presentation logic.

We can now see just some limitations that derive from mixing UI logic with presentation logic, yet there's also the fact that in the previous chapters, we've mixed the presentation logic with the data logic. The long-term implications for our current approach are scary: debugging will be difficult, and testing even more so.

It's time to refactor our Restaurants app to MVVM so that we can better separate its responsibilities.

Refactoring our Restaurants app to MVVM

To better separate responsibilities, we will choose the most popular presentation pattern: MVVM. Despite its flaws, when you compare it to MVC and MVP following the definition we previously gave them, it's the best candidate so far for the following reasons:

- It provides a pretty good separation between the UI logic and the presentation logic.
- Our UI layer (the composables) is designed to expect an observable state (more precisely, the Compose **`State`** object), just like the one the **`ViewModel`** in MVVM is set to expose.

Now, our Restaurants app already uses the Jetpack ViewModel (that exposes a Compose **`State`** object that is observed and consumed inside the composables), so we can say that we unknowingly started implementing this modified version of the pattern, whereby the Jetpack ViewModel is the ViewModel from MVVM.

NOTE

We will consider for now that the advantages of using the Jetpack ViewModel as the **`ViewModel`** *in MVVM are outweighing the disadvantages that it brings, so we will keep it as it is.*

However, just because we used a **`ViewModel`**, that doesn't mean we also implemented the MVVM presentation pattern correctly. Let's first have a look at how we structured our components and classes for the first screen displaying a list of restaurants. You can see how this looks here:

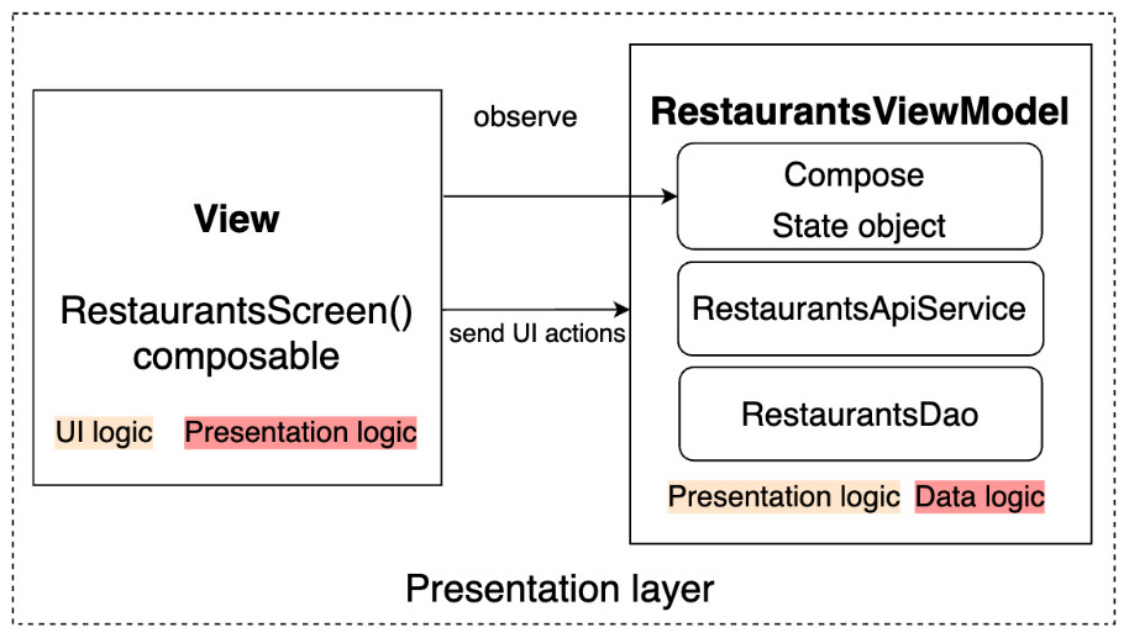

Figure 7.5 – Components with poorly separated responsibilities per layer in the MVVM pattern

For this screen, we notice two violations where layers contain more than one responsibility, as outlined here:

- The View layer (represented by the **`RestaurantsScreen()`** composable) performs both UI logic and presentation logic. While this composable should only contain UI logic (the stateless composables that consume the state content), some presentation logic lurked in when the **`isLoading`** variable was calculated, as illustrated in the following code snippet:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) ->
Unit) {
    …
    val isLoading = restaurants.isEmpty()
    …
```

```
}
```

The composables shouldn't be in charge of deciding their own state as in this case—the **`RestaurantsScreen()`** composable shouldn't hold presentation logic; instead, this should be moved inside the **`ViewModel`**.

- The **`RestaurantsViewModel`** class contains both presentation logic (such as holding and updating the state of the UI) and data logic (as it works with the Retrofit service Room **Data Access Object** (**DAO**) when it obtains and caches restaurants), as illustrated in the following code snippet:

```
class RestaurantsViewModel() : ViewModel() {
    private var restInterface:
RestaurantsApiService
    private var restaurantsDao = ...
    val state =
mutableStateOf(emptyList<Restaurant>())
    private suspend fun getAllRestaurants(): … {…}
        ...
    private suspend fun refreshCache() {...}
}
```

It's clear that presentation logic occurs when the **`state`** variable is updated, but there's also a lot of data logic when restaurants are obtained from the **`restInterface`** variable, then cached and updated in the **`restaurantsDao`** variable, and so on.

All this data logic shouldn't reside inside the **`ViewModel`** but instead inside the Model layer because the **`ViewModel`** should only present the data and not care to know about the data sources and how they are used—it only knows that it should receive some data.

Now, let's have a look at how we should correctly structure our classes (to follow MVVM) for the first flow of displaying a list of restaurants. The components should look like this:

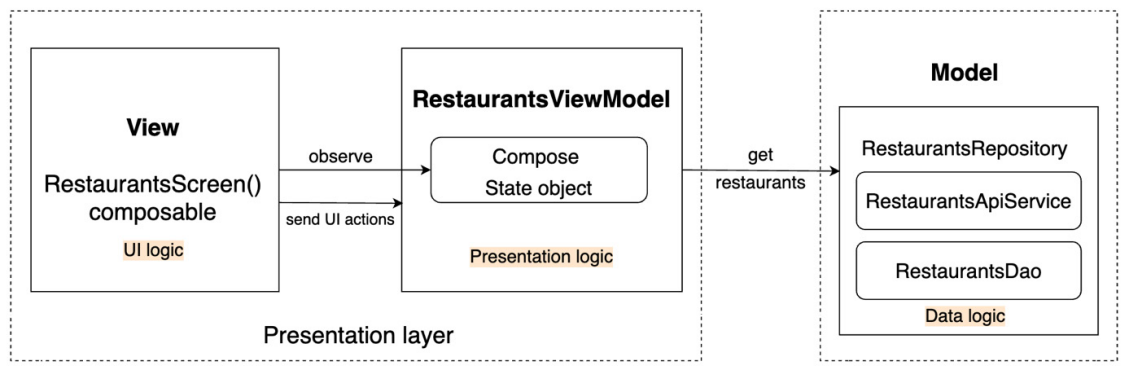

Figure 7.6 – Components with well-separated responsibilities per layer in the MVVM pattern

In the previous diagram, each component handles its own responsibility, as follows:

- The View component contains only composables (**`RestaurantsScreen`**) with UI logic (consuming the UI state).
- The ViewModel component (**`RestaurantsViewModel`**) contains only presentation logic (holds the UI state and mutates it).
- The Model component (where we will create a **`RestaurantsRepository`** class—more on that soon) contains only data logic (obtains restaurants from remote sources, caches them into a local source, and so on).

To achieve this separation, in this section, we will be doing the following:

- Separating UI logic from presentation logic
- Separating presentation logic from data logic

Let's start!

Separating UI logic from presentation logic

The UI logic (rendering composables) is already at the UI level (Compose UI), so we don't have to do anything from this point of view. However, we need to extract the presentation logic from the UI layer to the **`ViewModel`**, where it should reside.

More specifically, from within the **`RestaurantsScreen()`** composable, we want to move the calculation of the **`isLoading`** variable to the **`RestaurantsViewModel`** class, simply because the **`ViewModel`** should decide and also know better when the screen should be in a loading state.

To do that, we will create a **`state`** class that will hold all the information the UI needs in order to render the correct state. This approach is much more efficient because the **`ViewModel`** is responsible for requesting data and therefore knows better when content arrives, and so on. Because of this, later on, it will be very simple for us to also allow the **`ViewModel`** to also dictate when the screen must show an error state. Proceed as follows:

1. Create a class that will model the UI state for the **`RestaurantsScreen()`** composable. Do that by clicking on the application package, selecting **New**, and then **Kotlin Class/File**. Enter **`RestaurantsScreenState`** as the name and select **Data class** as the type. Inside the new file, add fields that define this screen's state, a **`restaurants`** list, and an **`isLoading`** flag. The code is illustrated in the following snippet:

```
data class RestaurantsScreenState(
    val restaurants: List<Restaurant>,
    val isLoading: Boolean)
```

Since we've used a **`data class`** instead of a regular **`class`**, we will be able to easily perform mutation on this object with the **`.copy()`** function, thereby ensuring that since the Compose **`state`** object will receive a new object, it will know to trigger recomposition.

2. Inside the **`RestaurantsViewModel`** class, update the initial state value of the **`state`** variable and pass a **`RestaurantsScreenState`** object, as follows:

```
class RestaurantsViewModel() : ViewModel() {
    …
    val state = mutableStateOf(
        RestaurantsScreenState(
```

```
                restaurants = listOf(),
                isLoading = true)
        )
        …
    }
```

We've marked the **`restaurants`** field as an empty list, and **`isLoading`** is **`true`** because from this point on, we're waiting for restaurants and the UI should render a loading state.

3. Still inside the **`RestaurantsViewModel`** class, find the **`getRestaurants()`** method and update the way we update the **`state`** variable, as follows:

```
    private fun getRestaurants() {
        viewModelScope.launch(errorHandler) {
            val restaurants = getAllRestaurants()
            state.value = state.value.copy(
                restaurants = restaurants,
                isLoading = false)
        }
    }
```

We first stored restaurants inside a **`restaurants`** variable. Then, we used the **`copy()`** function to pass a new restaurants list that we received to the **`restaurants`** field, and also marked the **`isLoading`** field to **`false`** because the data has arrived and the UI should no longer render a loading state.

4. Still in the **`RestaurantsViewModel`** class, make sure that the **`toggleFavorite()`** method is correctly updating the **`state`** variable object using the **`copy()`** function, as follows:

```
    fun toggleFavorite(id: Int, oldValue: Boolean) {
        viewModelScope.launch(errorHandler) {
            val updatedRestaurants = […]
            state.value = state.value.copy(restaurants
    =
                updatedRestaurants)
```

```
    }
}
```

All right—we've added all the presentation logic within the **`ViewModel`**, and it's now time to update the UI (our composables) to render new possible UI states.

5. Refactor the **`RestaurantsScreen()`** composable to consume the new UI state content, as follows:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) ->
Unit) {
    val viewModel: RestaurantsViewModel =
viewModel()
    val state = viewModel.state.value
    Box(…) {
        LazyColumn(…) {
             items(state.restaurants) {…}
        }
        if (state.isLoading)
            CircularProgressIndicator()
    }
}
```

Let's break down what we've done, as follows:

- We renamed the **`restaurants`** variable as **`state`** to better suggest that this variable holds the state of this screen.
- We passed **`state.restaurants`** to the **`LazyColumn`** composable's **`items`** DSL function.
- We deleted this line: **`val isLoading = restaurants.isEmpty()`**.
- We updated the condition for when to show **`CircularProgressIndicator()`** based on the **`state.isLoading`** value—no more decision-making logic inside this composable.

6. Build and run the app.

You should be able to see the loading indicator, just as before, yet the difference is that the presentation logic is better separated and held by the **`ViewModel`**. With our new approach, if for any reason we receive an empty list from our data sources (Retrofit and Room), the application won't misbehave and show a loading state because the UI is checking whether the list is empty or not.

To see how simple it is to add a new state to our Compose-based UI, let's continue by setting an error state when any error is thrown inside the **`ViewModel`**.

7. Inside the **`RestaurantsScreenState`** class, add an **`error: String`** parameter that will hold an error message if any error occurs, as follows:

```
data class RestaurantsScreenState(
    val restaurants: List<Restaurant>,
    val isLoading: Boolean,
    val error: String? = null
)
```

To simplify our work with state handling inside the **`ViewModel`**, we've set a default value of **`null`** to the **`error`** field, since the initial state of the screen shouldn't ever contain an error.

8. Inside the **`RestaurantsViewModel`** class, find the **`errorHandler`** variable that we use to catch any exception that might be thrown by our coroutines, and update the **`state`** object by passing an **`exception.message`** error message to the **`error`** field. The code is illustrated in the following snippet:

```
class RestaurantsViewModel() : ViewModel() {
    …
    private val errorHandler =
        CoroutineExceptionHandler { _, exception ->
            exception.printStackTrace()
```

```
                state.value = state.value.copy(
                    error = exception.message,
                    isLoading = false
                )
            }
        ...
    }
```

Additionally, we've set the **`isLoading`** field to **`false`** on the new state simply because if an error is thrown, we don't want the UI to be in a loading state.

If, however, you want to add a retry button that is pressed after an error has occurred and was shown, you would have to set the **`error`** field to **`null`** when that button is pressed so that the UI won't remain in an error state indefinitely.

9. Inside the **`RestaurantsScreen()`** composable, add another **`if`** statement in the **`Box`** composable. This statement checks whether the **`state`** object contains an error message to be shown, and if that is **`true`**, add a **`Text`** composable that will display the error message. The code is illustrated in the following snippet:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) ->
Unit) {
    …
    Box(…) {
        LazyColumn(...) {…}
        if (state.isLoading)
            CircularProgressIndicator()
        if (state.error != null)
            Text(state.error)
    }
}
```

10. Build the project, and now, let's test the error scenario. Yet to see the error message, we need to simulate an error.

If you remember, inside our **`RestaurantsViewModel`** class's **`getAllRestaurants()`** method, we check if we failed in retrieving restaurants from the server (Retrofit client), and if this happens while the Room DAO is also empty, we throw this error message: **`"Something went wrong. We have no data."`**.

To reproduce this scenario, make sure that the following applies:

- You have cleared the cache of the application. To do that, inside your device or emulator, go to **Settings**, then **Applications**, and search for our Restaurants app and press on it. Then, press **Storage and Cache** and then **Clear Storage**.
- Your device/emulator is disconnected from the internet.

11. Run the application. You should see this message in the center of the screen: **`"Something went wrong. We have no data."`**.
    NOTE
    For the sake of simplicity, we made sure that the UI logic is separated from the presentation logic only within the first screen of our app. When you're looking to move logic to corresponding classes, thereby ensuring SoC, you need to make sure to do so for all other screens within the app, together with their **`ViewModel`** *classes, and so on.*

Now that we've separated UI logic from presentation logic, it's time to separate some data logic.

Separating presentation logic from data logic

While the **`RestaurantsViewModel`** class contains data logic because it interacts with the Retrofit service and the Room DAO to obtain and cache restaurants, it should only hold presentation logic because its core responsibility is to govern the UI state.

Another sign that our **`RestaurantsViewModel`** has piled up a lot of logic is that it currently stands at around 90 lines of code—this might not seem much yet, remember that our application is pretty simple and we have little presentation logic, so 90 lines will definitely turn into thousands for production-ready applications.

We want to move the data logic out of the **`RestaurantsViewModel`** into a different class. Since data logic is part of the Model layer of our application, in this section, we will start exploring how to define the Model layer with the help of Repository classes.

The **Repository** pattern represents a strategy for abstracting data access inside your application. In other words, Repository classes hide away from the caller all the complexity associated with parsing data from the server, storing it in local databases, or performing any caching/refreshing mechanisms.

In our app, the **`RestaurantsViewModel`** class must decide whether to get data from the **`restInterface`** (remote) source or from the **`restaurantsDao`** (local) source, while also making sure to refresh the cache. The following snippet shows the code that is executed:

```
class RestaurantsViewModel() : ViewModel() {
    private var restInterface: RestaurantsApiService
    private var restaurantsDao = [...]
        ...
    private suspend fun refreshCache() {...}
}
```

This is obviously wrong. The **`ViewModel`** shouldn't care which particular data source to call as it shouldn't need to be the one that initiates caching to local sources. The **`ViewModel`** should only care about receiving some content that it will prepare for presentation.

Let's lift this burden from the **`RestaurantsViewModel`** class by creating a Repository class that will abstract all the data logic, as it will be interact-

ing with the two data sources (web API and Room DAO) to do the following:

- Provide a **`List<Restaurant>`** object to the Presentation layer
- Handle any caching logic such as retrieving restaurants from the web API and caching them to the Room local database
- Define a **single source of truth** (**SSOT**) for data—the Room database

To do that, we must only move the data logic out of the **`ViewModel`** and separate it in a Repository class. Let's begin, as follows:

1. Create a Repository class by clicking on the application package, selecting **New**, and then **Kotlin Class/File**. Enter **`RestaurantsRepository`** as the name and select **Class** as the type:

```
class RestaurantsRepository { }
```

Now, let's start moving some code!

2. From inside the **`RestaurantsViewModel`** class, cut the **`restInterface`** variable and its initialization logic from the **`init`** block and paste it inside **`RestaurantsRepository`**, as follows:

```
class RestaurantsRepository {
    private var restInterface:
RestaurantsApiService =
        Retrofit.Builder()
            .addConverterFactory(…)
            .baseUrl(…)
            .build()
            .create(RestaurantsApiService::class.ja
va)
}
```

3. Do the same for the **`restaurantsDao`** variable, as follows:

```
class RestaurantsRepository {
    private var restInterface:
RestaurantsApiService = …
```

```
    private var restaurantsDao = RestaurantsDb
        .getDaoInstance(
            RestaurantsApplication.getAppContext())
}
```

4. Inside the **RestaurantsViewModel** class, add a **repository** variable and instantiate it with the **RestaurantsRepository()** constructor, like this:

```
class RestaurantsViewModel() : ViewModel() {
    private val repository =
RestaurantsRepository()
    val state = mutableStateOf(…)
    private val errorHandler =
CoroutineExceptionHandler { … }
    init {
        getRestaurants()
    }
     […]
}
```

Make sure that the **RestaurantsViewModel** no longer contains the **restInterface** variable, the **restaurantsDao** variable, or their initialization code from within the **init** block.

5. Move the **toggleFavoriteRestaurant()**, **getAllRestaurants()**, and **refreshCache()** methods of the **RestaurantsViewModel** class to the **RestaurantsRepository** class, as follows:

```
class RestaurantsRepository {
    private var restInterface:
RestaurantsApiService = …
    private var restaurantsDao = […]
    private suspend fun toggleFavoriteRestaurant(…)
= […]
    private suspend fun getAllRestaurants(): […] {
… }
    private suspend fun refreshCache() { … }
}
```

6. Make sure that apart from the **`init { }`** block, the **`RestaurantsViewModel`** class only contains the **`toggleFavorite()`** and **`getRestaurants()`** methods, as follows:

```
class RestaurantsViewModel() : ViewModel() {
    […]
    init { getRestaurants() }
    fun toggleFavorite(id: Int, oldValue: Boolean)
{…}
    private fun getRestaurants() {…}
}
```

7. Inside the **`RestaurantsRepository`** class, remove the **`private`** modifier for the **`getAllRestaurants()`** and **`toggleFavoriteRestaurant()`** methods as **`RestaurantsViewModel`** will need to call them, so they must be public. The code is illustrated in the following snippet:

```
class RestaurantsRepository {
    […]
    suspend fun toggleFavoriteRestaurant(…) = […]
    suspend fun getAllRestaurants(): […] { … }
    private suspend fun refreshCache() { … }
}
```

8. Going back inside the **`RestaurantsViewModel`** class, update the **`getRestaurants()`** method to now call **`repository.getAllRestaurants()`**, as follows:

```
private fun getRestaurants() {
    viewModelScope.launch(errorHandler) {
        val restaurants =
repository.getAllRestaurants()
        state.value = state.value.copy(…)
    }
}
```

9. Still inside the **`RestaurantsViewModel`** class, update the **`toggleFavorite()`** method to now call **`repository.toggleFavoriteRestaurant()`**, as follows:

```
fun toggleFavorite(id: Int, oldValue: Boolean) {
    viewModelScope.launch(errorHandler) {
```

```
            val updatedRestaurants = repository
                .toggleFavoriteRestaurant(id, oldValue)
            state.value = state.value.copy(…)
        }
    }
```

And we're done! While the functionality of the first screen should stay the same, we have now divided the responsibilities within this first flow not only within the Presentation layer but also between the Presentation layer and the Model layer.

ASSIGNMENT

You can try to practice what we've learned in this section on the details screen of the Restaurants application.

Next up, let's return for a while to the Presentation layer and inspect how the UI state is exposed from within our **`ViewModel`**.

Improving state encapsulation in ViewModel

Let's have a look at how the UI state is defined in the **`RestaurantsViewModel`** class, as follows:

```
class RestaurantsViewModel() : ViewModel() {
    …
    val state =
mutableStateOf(RestaurantsScreenState(
        restaurants = listOf(),
        isLoading = true))
    …
}
```

Inside the **`RestaurantsViewModel`**, we are holding the state within the **`state`** variable with the **`MutableState<RestaurantsScreenState>`** inferred type. This variable is public, so inside the UI layer, from within the **`RestaurantsScreen()`** composable, we can consume it by accessing the **`viewModel`** variable and directly obtaining the **`state`** object, as follows:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) -> Unit)
{
    val viewModel: RestaurantsViewModel = viewModel()
    val state = viewModel.state.value
    Box(…) {…}
}
```

The problem with this approach might not be obvious, but since the **`state`** variable is of type **`MutableState`**, not only can we read its value but we can also write its value. In other words, from within the composable UI layer, we have write access to the **`state`** variable through the **`.value`** accessor.

The danger here is that then we (or other colleagues within our development team) could mistakenly update the UI state from within the UI layer, like so:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) -> Unit)
{
    val viewModel: RestaurantsViewModel = viewModel()
    val state = viewModel.state.value
    Box(…) {…}
    viewModel.state.value =
viewModel.state.value.copy(
        isLoading = false)
}
```

You can try to add the previously highlighted line of code but remove it afterwards!

This represents a violation of responsibilities within the Presentation layer: the UI layer shouldn't perform presentation logic. In other words, the UI layer shouldn't be able to mutate its own state that is stored inside the **`ViewModel`**; instead, only the **`ViewModel`** should have the right to do so.

This way, the **`ViewModel`** is the only entity responsible for presentation logic such as defining or mutating the UI state. At the same time, the responsibilities within our presentation patterns would be properly divided and respected.

To fix this, we must somehow force the **`RestaurantsViewModel`** class to expose a public **`state`** variable of type **`State`** instead of **`MutableState`**. This will prevent the UI layer from accidentally mutating its own state.

We can do this by having the Kotlin **backing property** feature implemented for our **`state`** variable. This feature states that if a class has two properties that are conceptually the same, yet one of them is part of the public API and the other one is an implementation detail, we can use an underscore to prefix the private property.

Let's see what this means by applying it directly in code, as follows:

1. First, within the **`RestaurantsViewModel`** class, let's prevent our **`state`** variable from being accessed because it's of type **`MutableState`**, as follows:

```
class RestaurantsViewModel() : ViewModel() {
    …
    private val state = mutableStateOf(…)
    …
}
```

2. Then, still in the **`RestaurantsViewModel`** class, rename the **`state`** variable **`_state`**. You can do that by selecting the **`state`** variable, and then pressing *Shift + F6*. Make sure that all previous usages of **`state`** are now called **`_state`**. The code is illustrated in the following snippet:

```
class RestaurantsViewModel() : ViewModel() {
```

```
    …
    private val _state = mutableStateOf(…)
    private val errorHandler =
        CoroutineExceptionHandler {
            …
            exception.printStackTrace()
            _state.value = _state.value.copy(…)
        }
    […]
    fun toggleFavorite(id: Int, oldValue: Boolean)
{
        viewModelScope.launch(errorHandler) {
            val updatedRestaurants = …
            _state.value = _state.value.copy(…)
        }
    }
    private fun getRestaurants() {
        viewModelScope.launch(errorHandler) {
            val restaurants =
                repository.getAllRestaurants()
            _state.value = _state.value.copy(…)
        }
    }
}
```

The **`_state`** variable is now the private state of type **`MutableState`**, so it's the variable that we referred to as the implementation detail. This means that the **`ViewModel`** can mutate it, but it shouldn't be exposed to the outer world. Yet what should we expose to the UI layer?

3. Still inside the **`RestaurantsViewModel`**, create another **`state`** variable called **`state`** of type **`State<RestaurantsScreenState>`** and define its custom getter through the **`get()`** syntax, as follows:

```
class RestaurantsViewModel() : ViewModel() {
    …
```

```
    private val _state = mutableStateOf(...)
    val state: State<RestaurantsScreenState>
        get() = _state
    …
}
```

The **state** variable is now the public state of type **State** (so, it's part of the public API), and this means that when the UI layer will try to get its value, the **get()** syntax will be called and the content within the **_state** variable will be returned.

Behind the scenes, the **_state** variable's type **MutableState** is downcasted to type **State** of the **state** variable. This means that composables won't be able to ever mutate the state within the **ViewModel**.

Conceptually, both the **state** variable and the **_state** variable are the same, yet **state** is used as part of a public contract with the outside world (so that it can be consumed by the UI layer), and **_state** is used as an internal implementation detail (a **MutableState** object that can be updated by the **ViewModel**).

4. Finally, inside the **RestaurantsScreen()** composable, make sure that the **state** variable is consumed, like this:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) ->
Unit) {
    val viewModel: RestaurantsViewModel =
viewModel()
    val state = viewModel.state.value
    Box(…) {…}
}
```

If you now try to mutate the **state** variable's value, as we did at the beginning of this section, then the **Integrated Development Environment**

(**IDE**) will show you a compilation error telling you that you need to reassign a **`val`** variable, as illustrated in the following code snippet:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) -> Unit)
{
    val viewModel: RestaurantsViewModel = viewModel()
    val state = viewModel.state.value
    Box(…) {…}
    viewModel.state.value =
viewModel.state.value.copy(
        isLoading = false)
}
```

This effectively means that our UI can't mutate its own state by accident anymore.

ASSIGNMENT

You can try to practice what we've learned in this section on the details screen of the Restaurants application.

Summary

In this chapter, we had a first look at the SoC principle. We understood why we must split an application's responsibilities across several layers and explored how we can do that with the help of presentation design patterns.

In the first part of this chapter, we had a quick look over the implementations for the most common presentation patterns in Android: MVC, MVP, and MVVM.

After that, we established that MVVM might be an appropriate choice for our Compose-based Restaurants application. We understood in which

layer each type of logic must reside, and then tried to achieve SoC as well as possible in our application.

In the last part of this chapter, we noticed how easy it is for our UI layer to extend its responsibilities and start performing presentation logic by mutating the UI state within the **`ViewModel`**. To counter that, we learned how to better encapsulate the UI state by using backed properties.

Let's continue our journey of improving our application's architecture in the next chapter where we will try to adopt some design decisions from the well-known Clean Architecture software design philosophy.