Building Modern Apps for Android

Compose, Kotlin, Coroutines, Jetpack and the best tools for native development.

Yair Carreno

Building Modern Apps for Android

Compose, Kotlin, Coroutines, Jetpack, and the best tools for native development.

Yair Carreno

This book is for sale at http://leanpub.com/building-modern-apps-for-android

This version was published on 2022-07-04

Contents

Preface

About the book

I must confess that I had to rethink the edition of the book several times before I managed to structure it to the current version.

The reason was straightforward. While I was writing the book's content, the announcement appeared for improvements in the architecture components with *Jetpack*; later, from *Kotlin*, more powerful tools were introduced, and later enabled *Flow Coroutines* as an option for reactive programming. As if that were not enough, the introduction of *Compose* is announced.

Mind Blown!

I did not hesitate twice. We had to reinvent ourselves (yes, also that typical phrase around here).

And it is that *Compose* and the advent of declarative views in both *Android* and *iOS* were inescapable.

Therefore, I decided that for this book to serve as a guide in developing applications for *Android*, I had to involve the latest and best tools available in the ecosystem to design and implement mobile applications.

I think that the first reader who benefited from this book was me.

It has allowed me to explore and build components differently from how they had been until a few years ago and take advantage of the maximum of these recent changes that both Google and JetBrains have been contributing to the solutions and software development industry.

I have tried to be practical in the presentation of the topics, without much theory, rather, leaving the references for readers to investigate and delve into a particular topic and presenting the code of a project of *e-commerce* application without going into much detail.

However, leaving the functional and complete source code in a repository for the reader on their own to analyze, digest, and understand in their own time.

I am sincere in admitting that I have been quite excited about the capabilities *Compose*, *Kotlin*, *Jetpack Components*, and these other modern tools provide when implementing a native mobile app.

Once you learn to dominate this set of technologies, there is no going back. This modern style is my first choice of mobile app design, even though I'm older of experience working with the *old style*.

I admit that it was not easy at the beginning; reviewing and studying a concept several times as necessary until I understood it clearly.

Fortunately, *Google's engineering* team has documented and shared vital design guides as an implementation reference, many of which I have referenced in the sections of this book.

Whether you are an experienced developer or new to the arena, this book will provide an initial understanding of adopting the modern style of building native mobile applications for Android.

I hope you like this work and find it helpful and, above all, practical.

Who this book is for

This work is a guide and tool for every Architect or Developer of mobile applications. Many design concepts applied here will be helpful regardless of the technology or operating system used.

Questions and contact

If the reader finds any aspect in the book that deserves to be reviewed, comments and feedback are welcome. For this and any questions or concerns, the following channels are available:

- Email: yaircarreno@gmail.com
- Twitter: @yaircarreno[1]
- Blog: yaircarreno.com[2]
- Medium: Yair Carreno[3]

About the source code

This book includes the repositories with the source code of the examples presented in each practice and implementation chapter:

- Chapter 2: Codelab - Practicing with states
- Chapter 5: Skeleton: Main structure
- Chapter 6: Designing navigation in App
- Chapter 7: Implementing “Features”

The objective is that the reader has the implementation reference of each concept studied.

It is the reader’s free choice to follow the recommended implementation or make variations. Also, you should not take the examples presented here as pieces of code ready for production; they should first be subjected to quality tests as in any software design process.

Change log

Changelog is provided to keep the reader informed about updates and changes in this book.

[1] https://twitter.com/yaircarreno
[2] https://www.yaircarreno.com/
[3] https://medium.com/@yaircarreno

Chapter 1: Design principles

“State” is the heart of declarative views

The first paradigm we must have clear when designing declarative views through frameworks like *Compose* or *SwiftUI* is the **State**.

A UI component combines its graphic representation (*View*) and its state.

Any property or data that changes in the UI component can be represented as a state.

For example, in a UI component of type `TextFiel`, the text entered by the user is a variable that can change; therefore, `value` is a variable that could be represented as a state (*name*), as shown in the following code snippet 1.1.

Code snippet 1.1

```
TextField(
    label = { Text("User name") },
    value = name,
    onValueChange = onNameChange
)
```

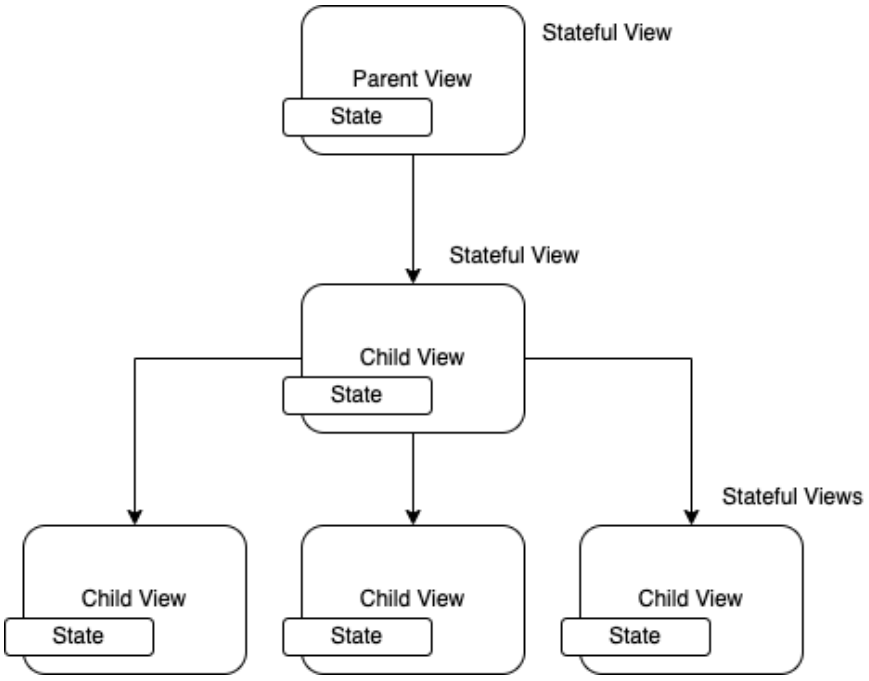

Figure 1.1 Declarative Views’s Hierarchy

A mobile application screen can comprise a hierarchy of views, as shown in figure 1.1.

Each view, in turn, can contain multiple state variables. For example, all views in Figure 1.1 have a state.

Views that contain or depend on a state are called *Stateful Views*, and views that do not have a state dependency are known as *Stateless Views*.

Both *Google* and *Apple* recommend as a good practice to design, as far as possible, *stateless views* due to the following advantages of using this type:

- You can reuse them.
- They allow you to delegate state management to other components.
- They are functional and avoid side effects.

According to these recommendations, the design must be oriented towards stateless views and convert those *stateful views* to *stateless views*.

And, how is that achieved? We'll find out in the next section.

Applying "State hoisting" to delegate states

State hoisting is a technique for converting *stateful views* to *stateless views*. That is achieved through inversion of control, as shown in the following code snippet 1.2:

Code snippet 1.2

```
// This is a Stateful View
@Composable
fun OrderScreen() {

    var name by remember { mutableStateOf("") }
    var phone by remember { mutableStateOf("") }

    ContactInformation(
        name = name,
        onNameChange = { name = it },
        phone = phone,
        onPhoneChange = { phone = it })
}

// This is a Stateless View
@Composable
fun ContactInformation(
    name: String,
    onNameChange: (String) -> Unit,
    phone: String,
    onPhoneChange: (String) -> Unit
) {
```

```
23
24      Column(
25          modifier = Modifier
26              .fillMaxSize()
27              .padding(8.dp),
28          horizontalAlignment = Alignment.CenterHorizontally
29      ) {
30          TextField(
31              label = {
32                  Text("User name")
33              },
34              value = name,
35              onValueChange = onNameChange
36          )
37          Spacer(Modifier.padding(5.dp))
38          TextField(
39              label = {
40                  Text("Phone number")
41              },
42              value = phone,
43              onValueChange = onPhoneChange
44          )
45          Spacer(Modifier.padding(5.dp))
46          Button(
47              onClick = {
48                  println("Order generated for $name and phone $phone")
49              },
50          ) {
51              Text("Pay order")
52          }
53      }
54  }
```

In code snippet 1.2, the `name` and `phone` state control is delegated to the *OrderScreen*, so the *ContactInformation* doesn't care about its data state and could be reused by other views.

OrderScreen becomes **stateful** and *ContactInformation* becomes **stateless**.

Code snippet 1.3

```
@Composable
fun OrderScreen() {

    // States name and phone
    var name by remember { mutableStateOf("") }
    var phone by remember { mutableStateOf("") }

    ContactInformation(
        name = name,
        onNameChange = { name = it },
        phone = phone,
        onPhoneChange = { phone = it })
}

@Composable
fun ContactInformation(
    name: String,
    onNameChange: (String) -> Unit,
    phone: String,
    onPhoneChange: (String) -> Unit,
    payOrder: () -> Unit
) {
    // Code omitted for simplicity
}
```

In the Code snippet 1.3 example, the inversion of control is achieved through *Higher-order functions*, allowing the definitions of the states and operations to be passed as arguments to the *ContactInformation* view.

Defining the "Source of Truth". Who is responsible for providing the states?

First, let's clarify what the term **source of truth** is.

Source of truth refers to the reliable source that provides the data that a view requires to be presented on the screen and with which the user will be interacting.

In our analysis, **data** is closely related to **states**. Views use states to receive the information (data) needed to do their job.

In figure 1.1, we see how the states are found in their respective views. That means that each view in the said diagram is a *source of truth*.

Even the variable `name` of the UI `TextField` component that we talked about before (*code snippet 1.1*) could be a state and, therefore, it is also a **source of truth**.

Is it reasonable to have so many sources of truth in a view hierarchy?

The answer is **no**.

It is recommended that the *source of truth* be limited to a single component (or to the minimum possible), so you can have greater control over the flow and avoid state inconsistencies.

Having a single, clearly defined *source of truth* also helps in the correct implementation of the *Unidirectional Data Flow design pattern*[4], which is the pattern promoted by declarative views like *Compose* or *SwiftUI*.

In the section Understanding data flow, I will say a little more about this pattern.

And how to reduce the number of sources of truth in my design?

That is possible by reducing the number of *Stateful views* through the *State hoisting* technique explained above and by centralizing the state in one view. Generally, the delegate is the view with the highest hierarchical level, a parent view.

For instance, figure 1.2 shows that there is only one source of truth, and it is the parent view.

On the one hand, the child's views are only responsible for propagating the **events** received by the interaction with the user. On the other hand, they receive the **states** that will render the view (recomposition[5]) to reflect the changes in the UI.

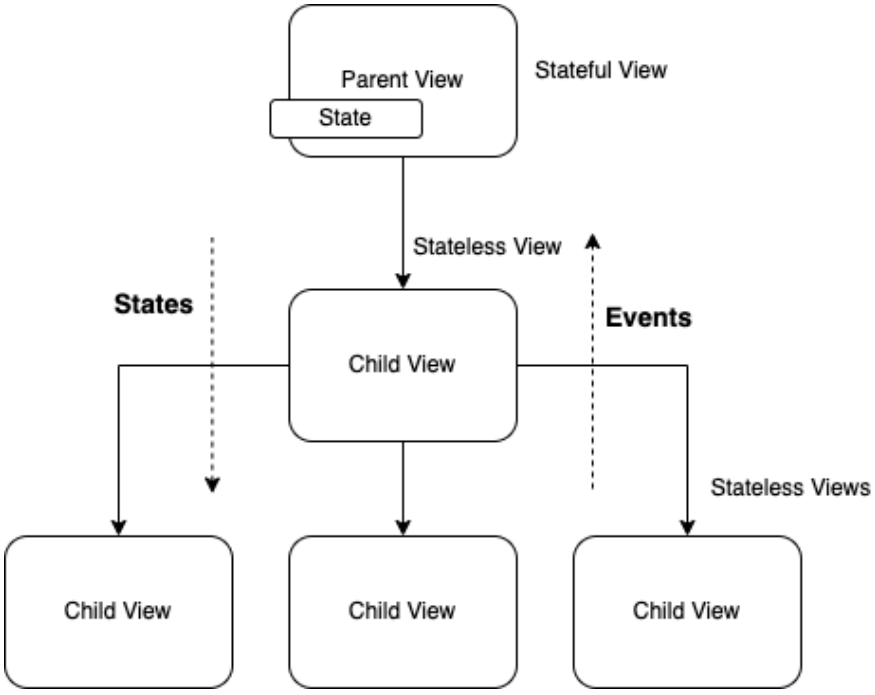

Figure 1.2 Delegating state handling to a View

[4] Manage state with Unidirectional Data Flow
[5] Recomposition

Is there an option other than delegating all state-handling responsibility to just one view?

The answer is **yes**.

A better option is to delegate this responsibility to a *State Holder* or a *ViewModel* that fulfills that role[6]. Let's see more detail in the next section.

ViewModel as Source of truth

Another component is called upon to handle state management to prevent the view from being overwhelmed with responsibilities. The proper element for this purpose is the well-known *ViewModel*.

As shown in figure 1.3, moving the states from *View* to the *ViewModel* creates a separation of responsibilities, allowing presentation logic and its effects on the state to be centralized.

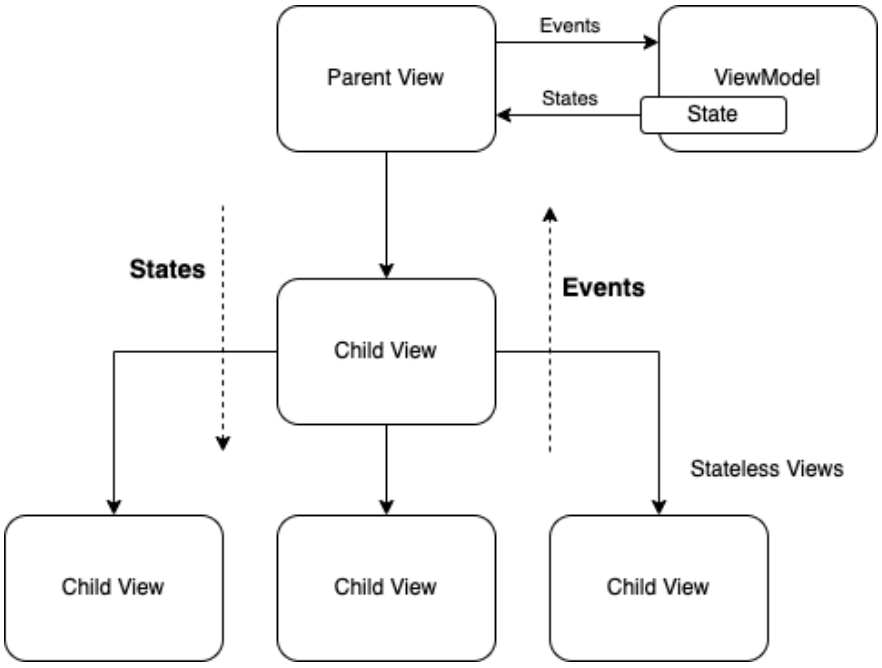

Figure 1.3 Delegating state handling to a ViewModel

Even though this component (*ViewModel*) is optional in implementations, I strongly recommend that it be used since it provides many advantages, such as effective management of the life cycle between data and views.

For more information on this architecture component, I recommend reviewing the official *Google* documentation on *ViewModels*[7].

Communication between *View* and *ViewModel* consists of only two types of messages, *Events* and *States*:

[6] Managing state in Compose
[7] ViewModel Overview

- **Events** are the actions notified to the *ViewModel* by any *View* or *Sub-View* as a consequence of a user action or interaction with the UI components.
- **States** represent the information (data) that the *ViewModel* delivers to the *Views* for their respective graphical interpretation.

The primary function of the *ViewModel* is to receive the events sent from the views, interpret them, apply business logic and transform them into states to be delivered back to the views.

The task of the *View* is to receive the states sent by the *ViewModel* and translate them into a graphical UI representation through recomposition.

Now, having a little more clarity about the responsibility of each component and the messages between them, let us now analyze what happens with the flow of information.

Understanding data flow, "Unidirectional Data Flow Pattern"

If we simplify the diagram in figure 1.3, the result will make the following diagram in figure 1.4:

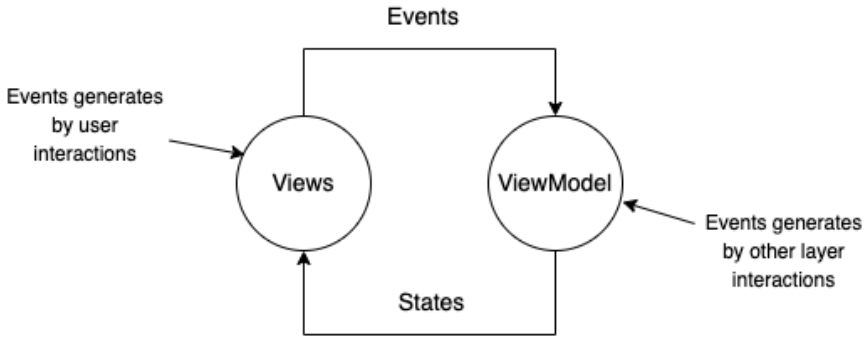

Figure 1.4 Unidirectional Data Flow

It's a **cycling message** between the *View* and the *ViewModel*. The flow of information only follows a single direction, hence the name of the *Unidirectional Data Flow pattern*.

The external factors that can inject events into the cycle are user interactions, such as a scroll in a list, a click on a button, and interactions with other application layers, such as a response from the Repository or a response from a user, background timer, or perhaps the arrival of a push notification.

The cycle cannot be interrupted, as any induced interruption or delay will result in a poor user experience. The user will perceive the application as slow, blocked, and poor quality.

Therefore, the design should keep in mind the following rules as far as possible:

- *Composable* that defines the view must be idempotent and functional.
- On the view side, there can be no tasks slowing down the cycle. Any task requiring extensive processing must be delegated to the *ViewModel*, which, through reactive programming and *Flow Coroutines*, will execute those tasks asynchronously.

Now that you have a better idea of the flow of data and the messages exchanged between *View* and *ViewModel*, it is logical to ask:

How is the communication channel between View and ViewModel implemented?

We will see it next.

Let's connect View and ViewModel components

As figure 1.4 show us, the two types of communication channels that need to be implemented are clearly identified.

The first channel is the events channel that goes in the direction *View –> ViewModel*.

For this implementation, it is only required that the *ViewModel* expose the public operations that can be called by the *View*, as shown in the following code snippet 1.4.

Code snippet 1.4

```
//UI's Events
fun onNameChange(): (String) -> Unit = {
    name = it
}

fun onPhoneChange(): (String) -> Unit = {
    phone = it
}
```

The second channel is the states channel that goes in the direction *ViewModel –> View*.

How does the UI know that the state has changed?

Observing the states. To follow the states, first, *ViewModel* must expose them to the UI through the *mutableStateOf* component like so:

Code snippet 1.5

```
// UI's states
var name by mutableStateOf("")
    private set
var phone by mutableStateOf("")
    private set
```

mutableStateOf will not only allow the state to be exposed to the view, but it will also allow the view to subscribe to be notified of any change in that state.

Let's see the complete implementation of the *ViewModel* and the *View* (*Composable*):

Code snippet 1.6: ViewModel

```
class OrderViewModel : ViewModel() {

    // UI's states
    var name by mutableStateOf("")
        private set
    var phone by mutableStateOf("")
        private set

    //UI's Events
    fun onNameChange(): (String) -> Unit = {
        name = it
    }

    fun onPhoneChange(): (String) -> Unit = {
        phone = it
    }

    fun payOrder(): () -> Unit = {
        println("Order generated for $name and phone $phone")
    }
}
```

Code snippet 1.7: View (Composables)

```
@Composable
fun OrderScreen(viewModel: OrderViewModel = viewModel()) {

    ContactInformation(
        name = viewModel.name,
        onNameChange = viewModel.onNameChange(),
        phone = viewModel.phone,
        onPhoneChange = viewModel.onPhoneChange(),
        payOrder = viewModel.payOrder()
    )
}

@Composable
fun ContactInformation(
    name: String,
    onNameChange: (String) -> Unit,
    phone: String,
    onPhoneChange: (String) -> Unit,
    payOrder: () -> Unit
) {

```

```
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(8.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        TextField(
            label = {
                Text("User name")
            },
            value = name,
            onValueChange = onNameChange
        )
        Spacer(Modifier.padding(5.dp))
        TextField(
            label = {
                Text("Phone number")
            },
            value = phone,
            onValueChange = onPhoneChange
        )
        Spacer(Modifier.padding(5.dp))
        Button(
            onClick = payOrder,
        ) {
            Text("Pay order")
        }
    }
}
```

So far, we have seen that states, such as `name` and `phone`, are representations of a `String` variable; that is, the state represents a *primitive variable*. However, we can extend the state representation to **components** and **screens**.

In the next section, we will look at other options for representing states.

Structures represented as states

In *Compose* and declarative views in general, states could represent different types of UI structures, as shown in Figure 1.5 below.

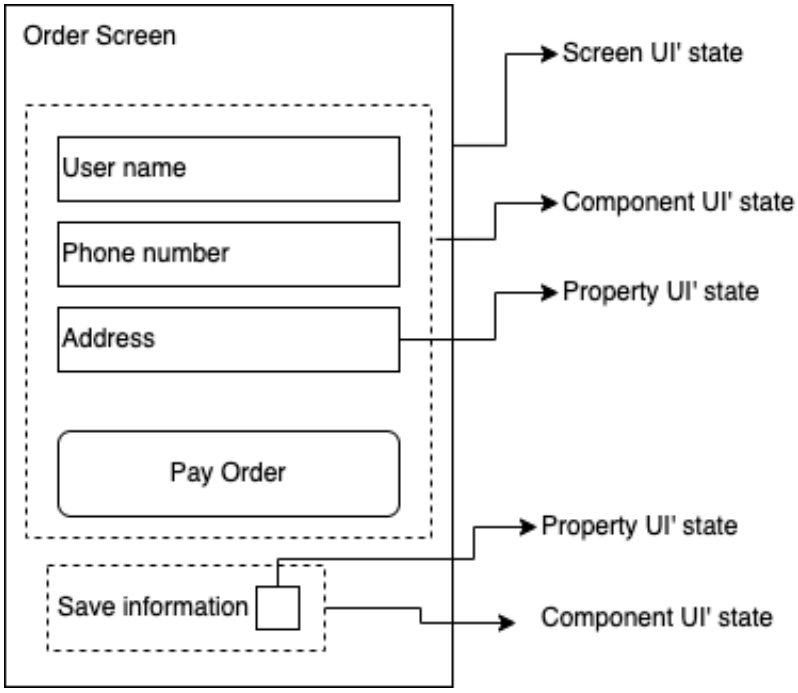

Figure 1.5 Structures represented by states

- **Property UI's state**: They are primitive variables represented as states. In Figure 1.5, text input fields such as `name`, `phone`, or `address` are of this type.
- **Component UI's state**: Represents the states associated with a component that groups related UI elements. For example, on the *OrderScreen*, a component called *ContactInformationForm* could group the required data, such as contact information. This component could have states of `NameValueChanged`, `PhoneValueChanged`, and `SuccessValidated`.
- **Screen UI's state**: It represents the states associated with a screen that can be treated as absolute and independent states; for instance, a screen called *OrderScreen* could have the following states: `Loading`, `Loaded successfully`, or `Load failed`.

Now let's see what implementation options exist in *Android* and *Kotlin* to define these states.

Property UI's state

They are states declared from a primitive type variable, such as *String*, *Boolean*, *List*, or *Int*, among others.

If it is declared in *ViewModel* (ViewModel as a Source of truth), its definition could be like this:

Code snippet 1.8

```
var name by mutableStateOf("")
    private set

var phone by mutableStateOf("")
    private set

var address by mutableStateOf("")
    private set

var payEnable by mutableStateOf(false)
    private set
```

If it is declared in *View* (View as a Source of truth), its definition in Composable could be like this:

Code snippet 1.9

```
var name by remember { mutableStateOf("") }
var phone by remember { mutableStateOf("") }
var address by remember { mutableStateOf("") }
var payEnable by remember { mutableStateOf(false) }
```

remember is a *Composable* that allows you to hold the state of the variable during recomposition temporarily.

As it is a *Composable*, this property can only be defined in declarative views, that is, in *Composable functions*.

Always remember that to use delegation through the *"by"* keyboard, you need to import:

Code snippet 1.10

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
```

In previous examples, we have only talked about representing properties or variables through states using *mutableStateOf* component.

However, it is also possible that data streams can be represented as states and observed by *Composables*. These additional options are related to *Flow*, *LiveData* o *RxJava*. In Capítulo 7: Implementando "Features" we will see several examples using *StateFlow*.

Component UI's state

When you have a set of interrelated UI elements, their states could be grouped into a single structure or UI component with a single state.

In figure 1.5, for instance, the elements *User name*, *Phone number*, *Address*, and even *Pay Order* button could be grouped into a single UI component and its states represented in a single state called, for example, *FormUiState*.

Code snippet 1.11

```
data class FormUiState(
    val nameValueChanged: String = "",
    val phoneValueChanged: String = ""
    val addressValueChanged: String = ""
)

val FormUiState.successValidated: Boolean get() = nameValueChanged.length > 1
                                                  && phoneValueChanged.length > 3
```

In this case, modeling multiple states in a consolidated class of states works very well since the variables are related and even define the value of other variables. For example, this happens with the `successValidated` variable, which depends on the `nameValueChanged` and `phoneValueChanged` variables.

Consolidating states adds benefit to implementation, centralizes control, and organizes code. It will be the technique that will be used most frequently in our implementation.

Screen UI's state

If what is required is to model states that can be independent and to be part of the same family, you could use the following definition:

Code snippet 1.12

```
sealed class OrderScreenUiState {
    data class Success(val order: Order): OrderScreenUiState()
    data class Failed(val message: String): OrderScreenUiState()
    object Loading: OrderScreenUiState()
}
```

That type of implementation is proper when working with absolute and exclusive states; you have one state or another, but not both at the same time.

Generally, simple screens of this type, such as the *OnboardignScreen* or *ResultScreen*, can be modeled with these states.

When the screen is more complex and contains many UI elements that operate independently and have multiple relationships, I recommend that the reader prefer the definition of states with the *Property UI' state* and *Component UI' state* techniques.

Modeling and grouping events

Returning to the *OrderScreen* example, we will now look at modeling *Events* and how to group them similarly to *States*.

Consider a screen like the one shown in the following figure 1.6:

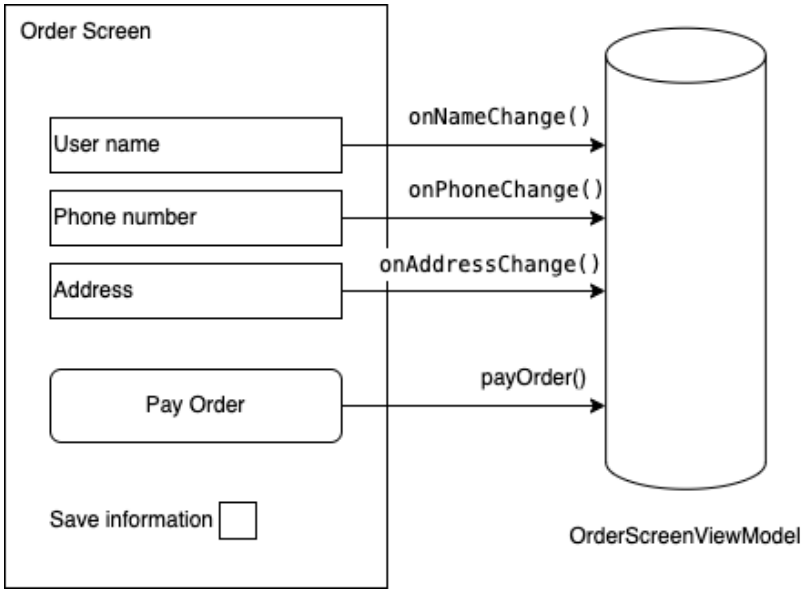

Figure 1.6 Multiple events

ViewModel exposes four operations (events) to the view, each used by a *View UI element.*

Analyzing the four events is related to a form for entering the user's contact information, so it makes sense to think of grouping them into a single type of event, as shown in the following figure 1.7:

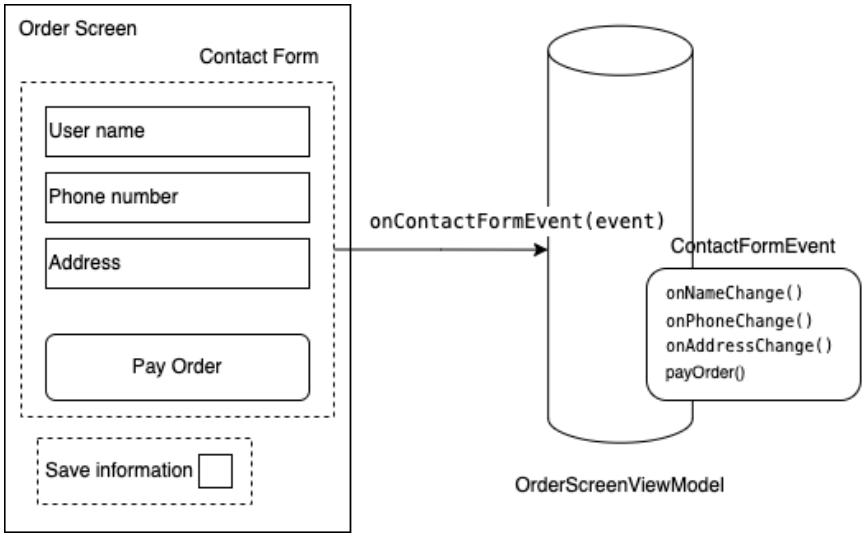

Figure 1.7 Grouping events

The implementation to represent the different types of events could be like this:

Code snippet 1.13

```
sealed class ContactFormEvent {
    data class OnNameChange(val name: String): FormUiEvent()
    data class OnPhoneChange(val phone: String): FormUiEvent()
    data class OnAddressChange(val address: String): FormUiEvent()
    object PayOrder: FormUiEvent()
}
```

Finally, you don't have to be so strict when simplifying states or events. It is necessary to analyze the advantages and disadvantages of each use and make the corresponding decisions.

For those related UI components, having them grouped makes a lot of sense; some other cross-cutting elements will be healthier to leave them independent.

Summary

In this first chapter, we have reviewed the main concepts used in the modern development of *Android* applications.

Concepts such as *States and Events*, *State hoisting*, *Source of truth*, and *Unidirectional Data Flow* are essential to understand before implementing *Jetpack Compose*, *ViewModels*, and other architecture components available for Android. That has been the reason why we have started with these concepts in this first chapter.

In the following chapters, we enter the definitions of architecture and design in a mobile application, for which we will use the concepts presented in this chapter as a reference.

Later, a mobile application called "Order Now" will be implemented using *e-commerce* as a concept. This application will have the main parts of *e-commerce*, such as a shopping cart, product list, and checkout process.

That work introduces the reader to a design and development experience close to a real and productive application.

But first, we will apply the concepts learned in this chapter to implement a simple form.

That will be the topic of the next chapter described below.

Chapter 2: Codelab - Practicing with states

Introduction

This chapter is a lab for applying the concepts learned in Chapter 1: Design principles.

The goal of the lab covers the following implementations:

- Create an application that uses a *View* as *source of truth*.
- Modify the application to use a *ViewModel* as *source of truth*.
- Group states and events to simplify messaging between *View* and *ViewModel*.

For example, in this laboratory, we will implement a part of one of the screens of *e-commerce*.

That *e-commerce* will be an example of an application designed and developed throughout the book Chapter 3: OrderNow, A Real Application.

The screen will be *OrderScreen* which contains information about an order requested by the user and other contact details of the user or buyer.

We will implement only part of the screen in the laboratory for simplification. The goal is to practice the different ways of managing states.

In the section, Checkout - Chapter 7 , the reader can find the complete implementation of the screen.

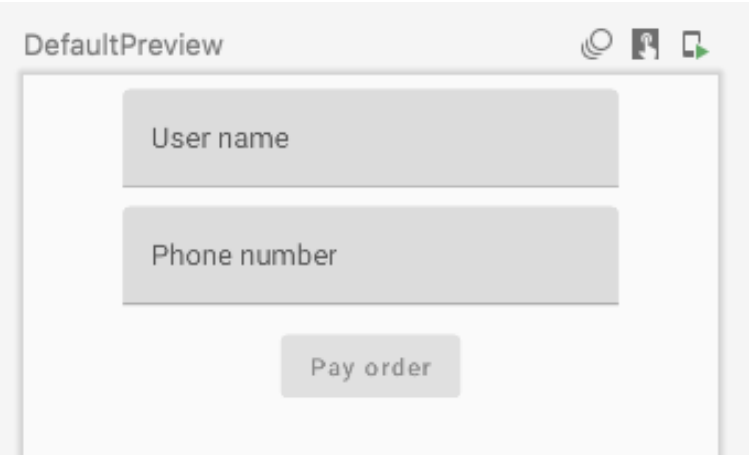

Figure 2.1 Screen Example Codelab

Source code

You can find the source code at Codelab - Practicing with states[8].

[8] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_02

The goal is to implement a form with two fields:

- User name
- Phone number

And additionally, a “Pay Order” button will be enabled or disabled, depending on the correct validation of the “User name” and “Phone number” fields.

Different implementation options are shown below.

“Views” as source of truth

The first step is identifying which UI elements can change and represent states on the screen. In this case, they would be:

- *Text value entered for the User name.*
- *Text value entered for the Phone number.*
- *Enable/disable the property of the Pay order button.*

Therefore, in View (*Composables*), we can represent properties like this:

Code snippet 2.1

```
var name by remember { mutableStateOf("") }
var phone by remember { mutableStateOf("") }
```

And, what about the state of the (enable/disable) property of the “Pay order” button?

Well, in this case, this state is derived from the other two states: *name* and *phone*. Therefore, that state requires no further definition.

View code could be like this:

Code snippet 2.2: Composables

```
@Composable
fun OrderScreen() {

    var name by remember { mutableStateOf("") }
    var phone by remember { mutableStateOf("") }

    ContactInformation(
        name = name,
        onNameChange = { name = it },
        phone = phone,
        onPhoneChange = { phone = it }
    )
```

```
}

@Composable
fun ContactInformation(
    name: String,
    onNameChange: (String) -> Unit,
    phone: String,
    onPhoneChange: (String) -> Unit
) {

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(8.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
    ) {
        TextField(
            label = {
                Text("User name")
            },
            value = name,
            onValueChange = onNameChange
        )
        Spacer(Modifier.padding(5.dp))
        TextField(
            label = {
                Text("Phone number")
            },
            value = phone,
            onValueChange = onPhoneChange
        )
        Spacer(Modifier.padding(5.dp))
        Button(
            onClick = {
                println("Order generated for $name and phone $phone")
            },
            enabled = name.length > 3 && phone.length > 4
        ) {
            Text("Pay order")
        }
    }
}
```

And what about the events?

In this screen example, the identified events are:

- Event when "User name" is modified.
- Event when "Phone number" is modified.
- Event when the "Pay Order" button is selected (clicked).

These events are managed like this:

Code snippet 2.3

```
//User name changed
onNameChange = { name = it }

...
//Phone number changed
onPhoneChange = { phone = it }

//Pay order clicked
Button(
    onClick = {
        println("Order generated for $name and phone $phone")
    },
    ...
)
```

Both **states** and **events** are managed by *View*, meaning that *View* is the only *Source of truth* on the screen.

Source code

You can find the source code at ViewAsSourceOfTruth[9].

"ViewModel" as source of truth

We will introduce another actor, a *ViewModel*[10], in the second implementation option.

ViewModel, as I mentioned in the first chapter, is an architecture component that we should include in our applications. This component is key to the separation of responsibilities in views and is part of the presentation layer of a mobile application.

ViewModel can be integrated into two ways: through manual configuration or a dependency manager.

[9] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_02/ViewAsSourceOfTruth
[10] ViewModel Overview

In this lab, we will include it manually; however, in the Chapter 7: Implementing "Features", it will be shown how to incorporate it through the Hilt[11] dependency manager.

Similar to the previous option, the first thing is to identify the states and events on the screen.

As they are already identified, the next step is to transfer them from the *View* to the *ViewModel* like this:

Code snippet 2.4: ViewModel definition

```
class OrderViewModel : ViewModel() {

    // UI's states
    var name by mutableStateOf("")
        private set
    var phone by mutableStateOf("")
        private set

    //UI's Events
    fun onNameChange(): (String) -> Unit = {
        name = it
    }

    fun onPhoneChange(): (String) -> Unit = {
        phone = it
    }

    fun payOrder(): () -> Unit = {
        println("Order generated for $name and phone $phone")
    }
}
```

In the previous code of *ViewModel*, we see the definition of the **states** and the **events**.

How do we bind the ViewModel to the View?

In view, we include the definition of the *ViewModel*, and at the same time, it is instantiated like this:

Code snippet 2.5

```
@Composable
fun OrderScreen(viewModel: OrderViewModel = viewModel()) {

    ContactInformation(
        name = viewModel.name,
        onNameChange = viewModel.onNameChange(),
        phone = viewModel.phone,
        onPhoneChange = viewModel.onPhoneChange(),
        payOrder = viewModel.payOrder()
```

[11] Dependency injection with Hilt

```
    )
}
```

Please don't forget!

To include the *ViewModel* manually, it is vital in the project to have the dependencies required and documented by Google in ViewModel, declaring dependencies[12]:

Code snippet 2.6

```
dependencies {
    def lifecycle_version = "2.5.0-rc01"

    // ViewModel
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"

    // ViewModel utilities for Compose
    implementation "androidx.lifecycle:lifecycle-viewmodel-compose:$lifecycle_version"
}
```

Complete implementation of the view would be as follows:

Code snippet 2.7: Compose definition

```
@Composable
fun OrderScreen(viewModel: OrderViewModel = viewModel()) {

    ContactInformation(
        name = viewModel.name,
        onNameChange = viewModel.onNameChange(),
        phone = viewModel.phone,
        onPhoneChange = viewModel.onPhoneChange(),
        payOrder = viewModel.payOrder()
    )
}

@Composable
fun ContactInformation(
    name: String,
    onNameChange: (String) -> Unit,
    phone: String,
    onPhoneChange: (String) -> Unit,
    payOrder: () -> Unit
) {

    Column(
```

[12] https://developer.android.com/jetpack/androidx/releases/lifecycle#declaring_dependencies

```
        modifier = Modifier
            .fillMaxSize()
            .padding(8.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        TextField(
            label = {
                Text("User name")
            },
            value = name,
            onValueChange = onNameChange
        )
        Spacer(Modifier.padding(5.dp))
        TextField(
            label = {
                Text("Phone number")
            },
            value = phone,
            onValueChange = onPhoneChange
        )
        Spacer(Modifier.padding(5.dp))
        Button(
            onClick = payOrder,
            enabled = name.length > 3 && phone.length > 4
        ) {
            Text("Pay order")
        }
    }
}
```

In this second implementation option, we see that both **states** and **events** are delegated to the *ViewModel*; therefore, *ViewModel* becomes the Source of truth.

With the Source of truth change, the design gains the flexibility to apply centralized business or presentation logic to the *ViewModel*.

So far, we have a proper and working implementation. But, it can be improved through the definition of *Component UI's state*, as we will see next.

Source code

You can find the source code at ViewModelAsSourceOfTruth[13].

[13] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_02/ViewModelAsSourceOfTruth

Grouping the “States”

In the example above, you can see that the fields are part of a form. Even the state of the button depends on the fields of the form. It then makes sense to group these UI elements into a single UI element containing them.

Since there are only three UI elements in the example, the benefits of grouping them may not be as obvious; however, let’s think about a screen that has many other sections on the screen with many other UI elements.

The first thing is to group the states in a structure called *FormUiState* in the following way:

Code snippet 2.8

```
data class FormUiState(
    val name: String = "",
    val phone: String = ""
)

val FormUiState.successValidated: Boolean get() = name.length > 3
                                                  && phone.length > 4
```

In *ViewModel*, we replace the states with a single state as follows:

Code snippet 2.9

```
class OrderViewModel : ViewModel() {

    // UI's states
    var formUiState by mutableStateOf(FormUiState())
        private set

    //UI's Events
    fun onNameChange(): (String) -> Unit = {
        formUiState = formUiState.copy(name = it)
    }

    fun onPhoneChange(): (String) -> Unit = {
        formUiState = formUiState.copy(phone = it)
    }

    fun payOrder(): () -> Unit = {
        println("Order generated for ${formUiState.name} and phone ${formUiState.phone}")
    }
}
```

In *View*, we update the use of states as follows:

Code snippet 2.10

```
@Composable
fun OrderScreen(viewModel: OrderViewModel = viewModel()) {

    ContactInformation(
        name = viewModel.formUiState.name,
        onNameChange = viewModel.onNameChange(),
        phone = viewModel.formUiState.phone,
        onPhoneChange = viewModel.onPhoneChange(),
        payOrder = viewModel.payOrder(),
        isValidPayOrder = viewModel.formUiState.successValidated
    )
}

@Composable
fun ContactInformation(
    name: String,
    onNameChange: (String) -> Unit,
    phone: String,
    onPhoneChange: (String) -> Unit,
    payOrder: () -> Unit,
    isValidPayOrder: Boolean
) {

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(8.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        TextField(
            label = {
                Text("User name")
            },
            value = name,
            onValueChange = onNameChange
        )
        Spacer(Modifier.padding(5.dp))
        TextField(
            label = {
                Text("Phone number")
            },
            value = phone,
            onValueChange = onPhoneChange
        )
        Spacer(Modifier.padding(5.dp))
```

```
        Button(
            onClick = payOrder,
            enabled = isValidPayOrder
        ) {
            Text("Pay order")
        }
    }
}
```

Grouping-related UI elements become more relevant when you have many UI elements on a screen. Grouping UI elements in *Components UI's State* simplifies, organizes, and generates cleaner code in the implementation.

The same technique can be applied to **events**. The difference, as will be shown below, is mainly in the type of representation.

Source code

You can find the source code at GroupingStates[14].

Grouping the "Events"

To further organize the code, we will now group the related events of the form.

The first thing is to create a structure to group the events in the following way:

Code snippet 2.11

```
sealed class FormUiEvent {
    data class OnNameChange(val name: String): FormUiEvent()
    data class OnPhoneChange(val phone: String): FormUiEvent()
    object PayOrderClicked: FormUiEvent()
}
```

The reader will notice that the grouping of the events is similar to the technique explained in the Screen UI's state section of the first chapter. Remember that this is applicable since the different types of events we are defining are related, but they can be mutually exclusive and independent.

In *ViewModel*, the messages are simplified to one as follows:

[14] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_02/GroupingStates

Code snippet 2.12

```
class OrderViewModel : ViewModel() {

    // UI's states
    var formUiState by mutableStateOf(FormUiState())
        private set

    //UI's Events
    fun onFormEvent(formEvent: FormUiEvent) {
        when (formEvent) {
            is FormUiEvent.OnNameChange -> {
                formUiState = formUiState.copy(name = formEvent.name)
            }
            is FormUiEvent.OnPhoneChange -> {
                formUiState = formUiState.copy(phone = formEvent.phone)
            }
            is FormUiEvent.PayOrderClicked -> {
                println("Sending form with parameters:
                ${formUiState.name} and ${formUiState.phone}")
            }
        }
    }
}
```

From *View*, the implementation would be:

Code snippet 2.13

```
@Composable
fun OrderScreen(viewModel: OrderViewModel = viewModel()) {

    ContactInformation(
        name = viewModel.formUiState.name,
        onNameChange = { viewModel.onFormEvent(FormUiEvent.OnNameChange(it)) },
        phone = viewModel.formUiState.phone,
        onPhoneChange = { viewModel.onFormEvent(FormUiEvent.OnPhoneChange(it)) },
        payOrder = { viewModel.onFormEvent(FormUiEvent.PayOrderClicked) },
        isValidPayOrder = viewModel.formUiState.successValidated
    )
}

@Composable
fun ContactInformation(
    name: String,
    onNameChange: (String) -> Unit,
    phone: String,
```

```
    onPhoneChange: (String) -> Unit,
    payOrder: () -> Unit,
    isValidPayOrder: Boolean
) {

    ...
}
```

Source code

You can find the source code at GroupingEvents[15].

Bonus extra

Some readers may have noticed that I contained the field validation logic in the *FormUiState* state structure.

Since the logic is often more complex than validating character lengths, it's best to delegate the verification and validation task to the *ViewModel*.

So, the following changes are added to the *ViewModel* and *FormUiState*:

Code snippet 2.14: In ViewModel

```
// Business's logic or maybe some UI's logic for update the state
companion object {
    fun applyLogicToValidateInputs(name: String, phone: String): Boolean {
        return name.length > 1 && phone.length > 3
    }
}
```

Code snippet 2.15: In FormUiState

```
data class FormUiState(
    val name: String = "",
    val phone: String = ""
)

val FormUiState.successValidated: Boolean get() =
                OrderViewModel.applyLogicToValidateInputs(name, phone)
```

Now all the logic is on the *ViewModel* side.

[15] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_02/GroupingEvents

Summary

In this lab and practice chapter, we have reviewed the ways to manage state and events, using *Views* or *ViewModel* as a *Source of truth*.

Also, we have used some techniques to better organize the states and events in structures; to have a better organized and easy-to-follow implementation.

In the next chapter, we will see a summary of the "Order Now" application, e-commerce, that we will implement throughout the book to explain the concepts and techniques of modern **Android App** development.

Chapter 3: OrderNow, A Real Application

About the application

OrderNow is an example of a Minimum Viable Product (MVP) of a mobile e-commerce application that we will design and implement throughout this book. We will use this application as an example to apply the concepts learned in each chapter of the book.

Implementing an e-commerce solution will bring us closer to the challenges that an accurate and productive application demands.

The following are the leading e-commerce functionalities that we will develop in *OrderNow*:

- Present a list of categories.
- Present a list of products by category.
- Present the detail of a specific product.
- Manage products (add or remove) in a shopping cart.
- See the list of products selected for purchase.
- Fill out the information and data to make the purchase (checkout).
- Simulate the payment process.

Screens

The screens related to different functionalities would be:

- Home
- Product List
- Product Detail
- Cart
- Checkout
- Place order

Home, Product List and Product Detail

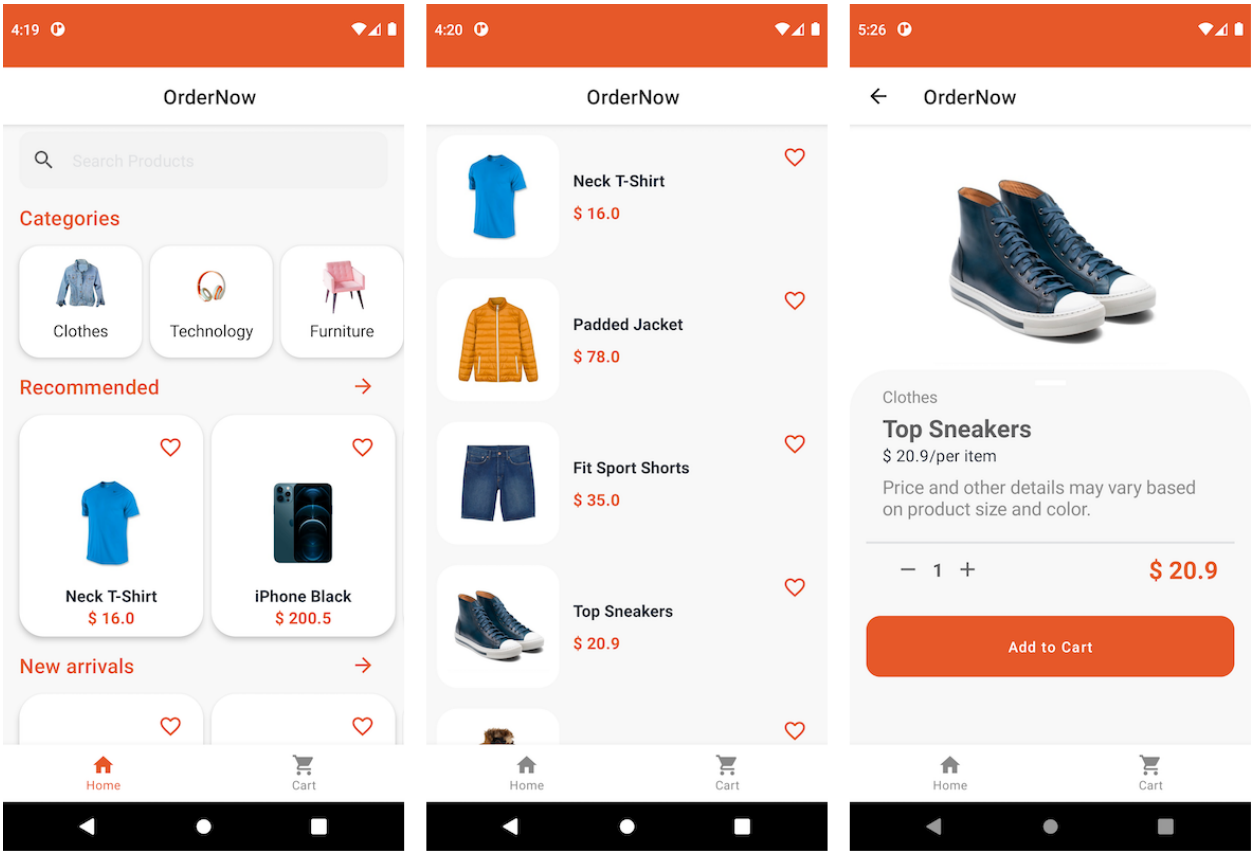

Figure 3.1 Screenshots: Home, Product List and Product detail

Cart and Checkout

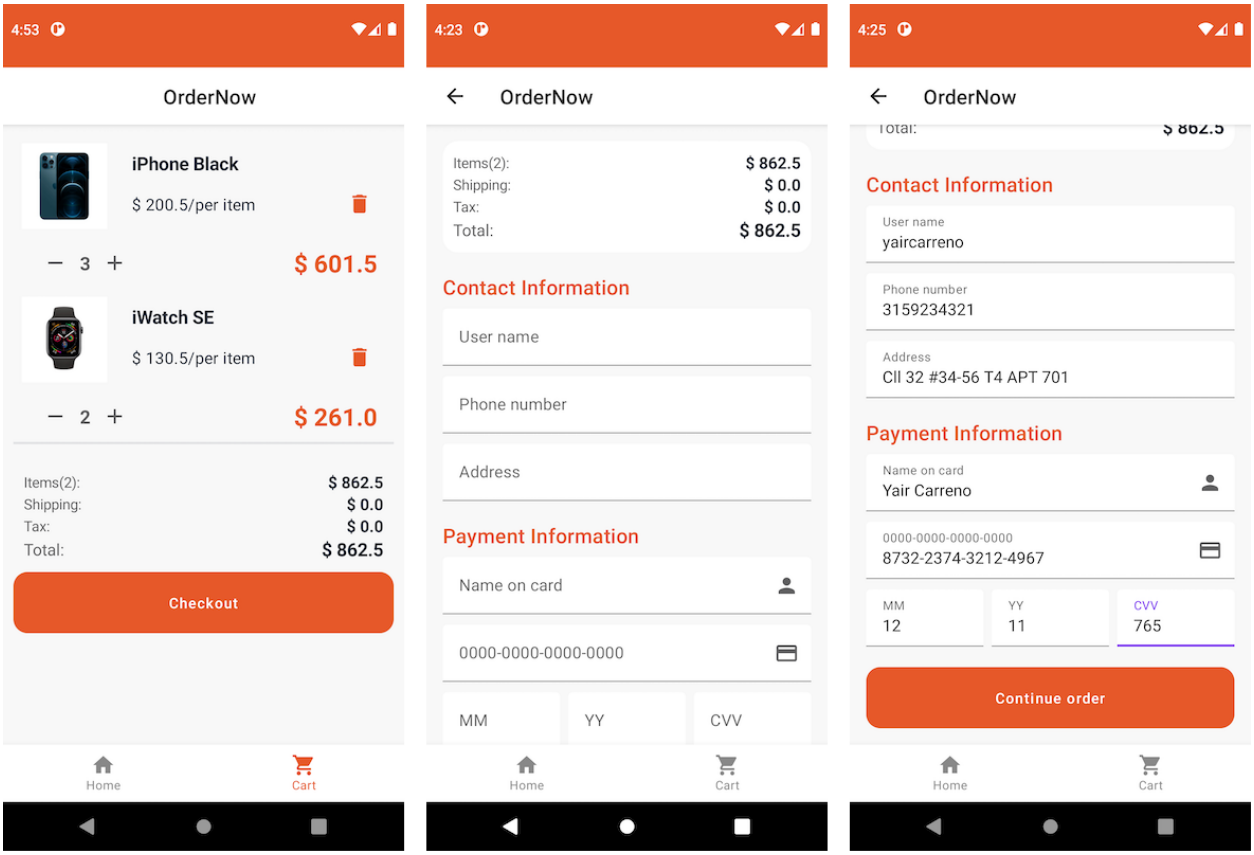

Figure 3.2 Screenshots: Cart and Checkout

Place Order

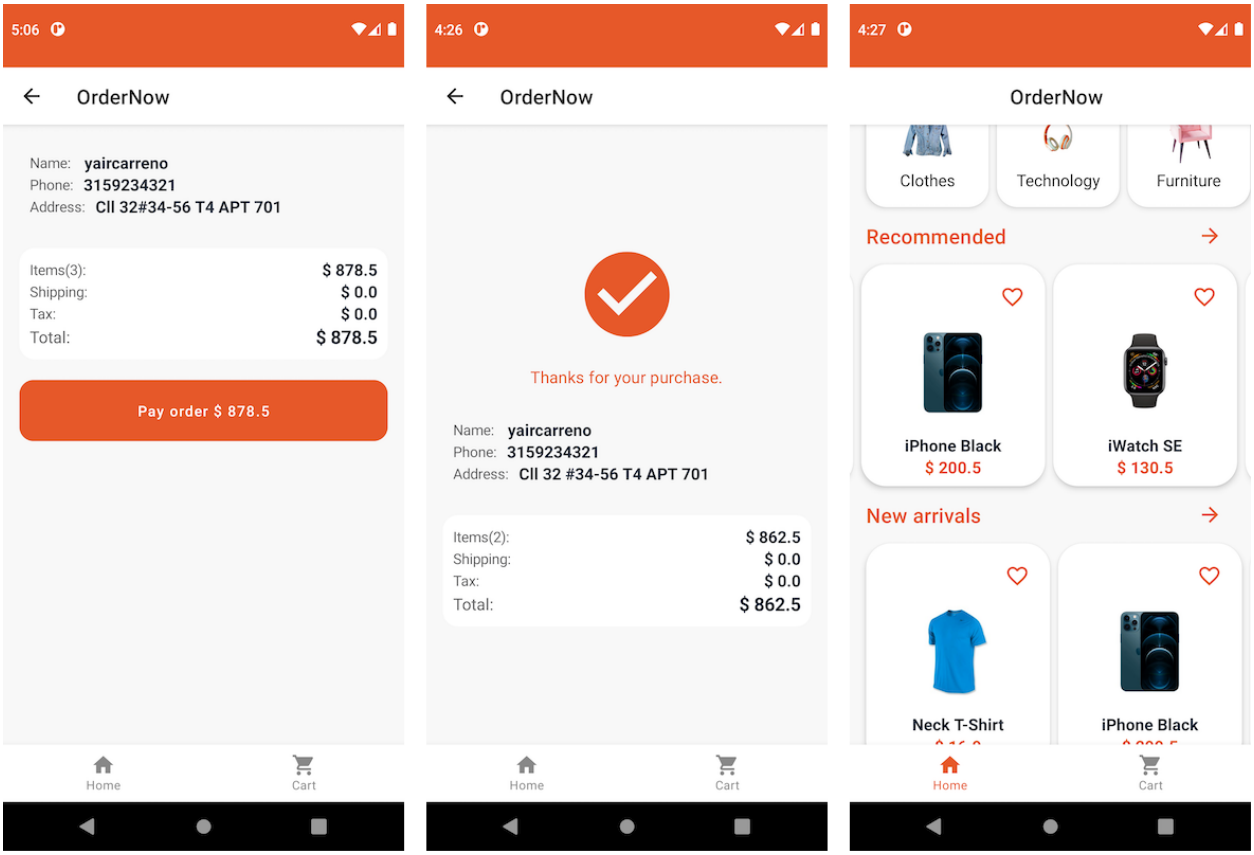

Figure 3.3 Screenshots: Place Order

Technologies

This section is a summary of the technical characteristics of *OrderNow* so that the reader is aware of the tools and design guide that will be used in the implementation.

Remember that this is an implementation proposal. The reader will be free to decide to replace or include some other tool with which they have experience or feel comfortable working.

Design and Architecture Guideline

In the chapters called Chapter 1: Design principles and Chapter 4: Application Architecture, the design and architecture guidelines are documented, that is, the *Minimum Viable Architecture (MVA)*[16], which will be used to develop *OrderNow*.

[16] A Minimum Viable Product Needs a Minimum Viable Architecture

Architecture components

- **Compose**[17]: It will be the framework for implementing declarative views in our presentation layer.
- **ViewModel**[18]: Architecture component in the presentation layer that we use to encapsulate business logic.
- **Flow**[19]: We will use *Flow Coroutines* for reactive programming in our application. *Flow* will allow messages between App components, whether synchronous or asynchronous, to be carried out in the most optimal way possible.
- **Navigation**[20]: Architecture component that we will use to implement navigation through the different screens of our application.

Dependencies

- Coil[21]: We use a library to load remote or local images in our APP, through *Kotlin* and with support for *Jetpack Compose*.

Out of the book's scope

Some topics are excluded from the book's content, not because they are less critical but rather to narrow the scope and meet specific goals.

Trying to cover all the related topics in an Android application could overextend the content and divert us from the main concepts that should be clear at the beginning.

The following capabilities are not included and are outside the scope of the *MVP* example:

- UI/UX Design Guide.
- Components of Authentication and Authorization.
- Testing.
- Accessibility.

Summary

This short chapter summarizes the technologies and components used in implementing *OrderNow*.

The reader not only will have the source code and tries to guess how it was built but also knows each decision made at the design and technical levels of implementation.

In the next chapter, I will describe the architecture and design decisions in the example App.

[17] Jetpack Compose
[18] ViewModel Overview
[19] Kotlin flows on Android
[20] Navigating
[21] Coil and Jetpack Compose

Chapter 4: Application Architecture

Choosing a style

The architecture that we will follow for the construction of *OrderNow* is based on the good practices and architecture guide recommended by Google in the **Guide to app architecture**[22].

These definitions include some *Clean Architecture*[23] principles for component definitions through the different layers.

Definition of the layers

In the application, we will define the following main layers:

- UI Layer
- Domain Layer
- Data Layer

UI Layer

This layer groups the UI elements, the Views (*composable functions*), *ViewModels*, and utilities of the presentation layer, such as format appliers and animations.

Considerations for designing this layer are:

- For state handling, follow the principles described in the Chapter 1: Design principles.
- For each screen, a corresponding *ViewModel* will be implemented.
- *ViewModels* will also function as State holders, that is, state managers.
- Navigation logic will be delegated to the View and will depend on the state of the APP.
- Side effects should be reported to the *ViewModel*.
- *ViewModels* should maintain their states when configuration changes occur.
- The use of *stateless Views* is encouraged.

[22] Guide to app architecture
[23] Clean Architecture

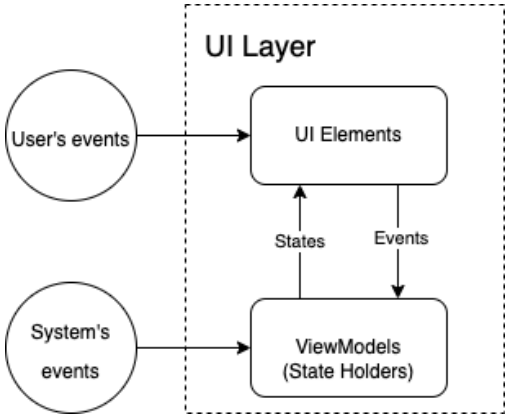

Figure 4.1 UI Layer

Domain Layer

Although this layer could be optional, I recommend that it be included to maintain a design consistent with the separation of responsibilities dictated by *Clean Architecture*.

This layer groups components called *UseCases*, which will manage business logic and all that logic that the *ViewModels* can reuse.

This layer also serves as a bridge between the *UI Layer* and the *Data Layer*.

The *Models type components* also belong to this layer. These components model the entities or data structures used by the presentation layer or the domain layer. For example, in *OrderNow*, both *Product* and *Category* represent models or entities.

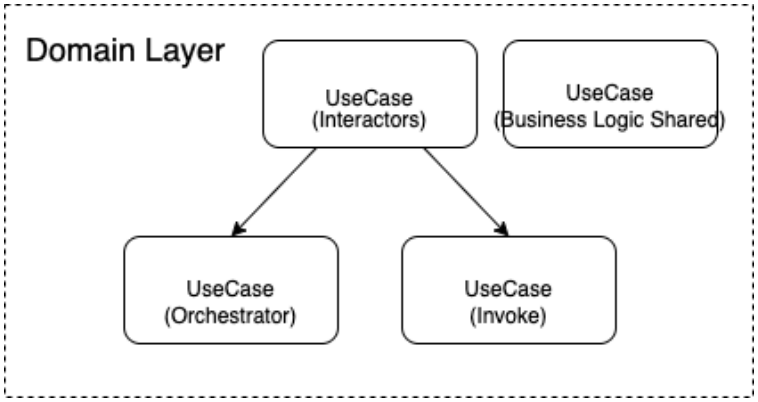

Figure 4.2 Domain Layer

Design considerations about this layer are:

- All presentation logic duplicated in views can be placed in a *useCase*.
- Components that belong to this layer can be stateless; they are components that do not require temporary persistence.
- Operations performed by *useCases* must be main-safe.

- *UseCases* can communicate with each other to orchestrate use case operations.
- Each *useCases* is responsible for one and only one operation.
- Each *usecCases* may make use of one or more *Repositories*.

Data Layer

This layer groups components called *Repositories* that orchestrate and encapsulate the integration logic to local and remote data sources. As their name suggests, they follow the *Repository*[24] pattern.

Other components of this layer are the *Datasources*, *Mappers*, and *DTOs*.

- **Datasource**: Contains the logic integration to external or local persistence sources.
- **DTO**: (Data Transfer Object)[25] It is the structure that models the persistence entity. Contains definitions used by the persistence mechanism. So that the other layers (*UI* and *Domain*) do not inherit these definitions, these types of entities are translated into models of the application's domain through *Mappers*.
- **Mapper**: They transform the DTOs into the model entities of the domain layer.

For the design of this layer, the following considerations are recommended:

- This layer can be used as a *source of truth*.
- The operations carried out by *Repositories* must be main-safe.
- One *Repository* is defined for each primary entity type, e.g., *ProductRepository*, *CategoryRepository*.

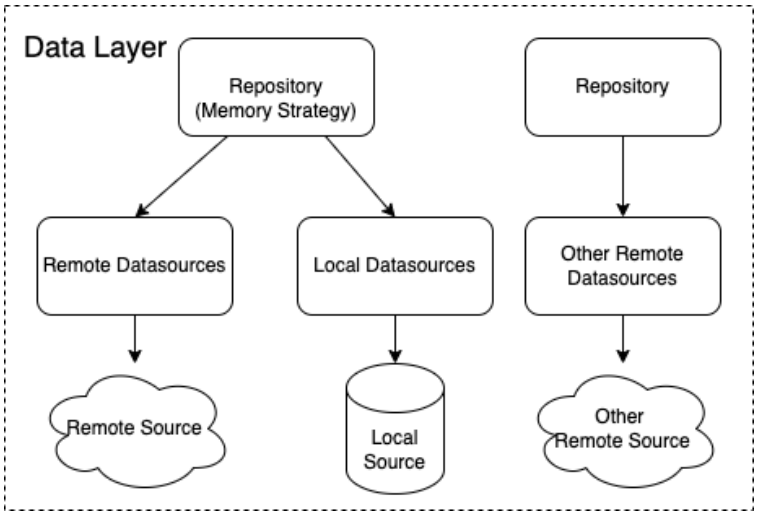

Figure 4.3 Data Layer

[24] Repository - Patterns of enterprise application architecture, page 322 by Robert Martin Fowler
[25] [Data Transfer Object: Patterns of enterprise application architecture, page 401 by Martin Fowler.

General architecture

A general diagram with the different integrated layers would be similar to the following:

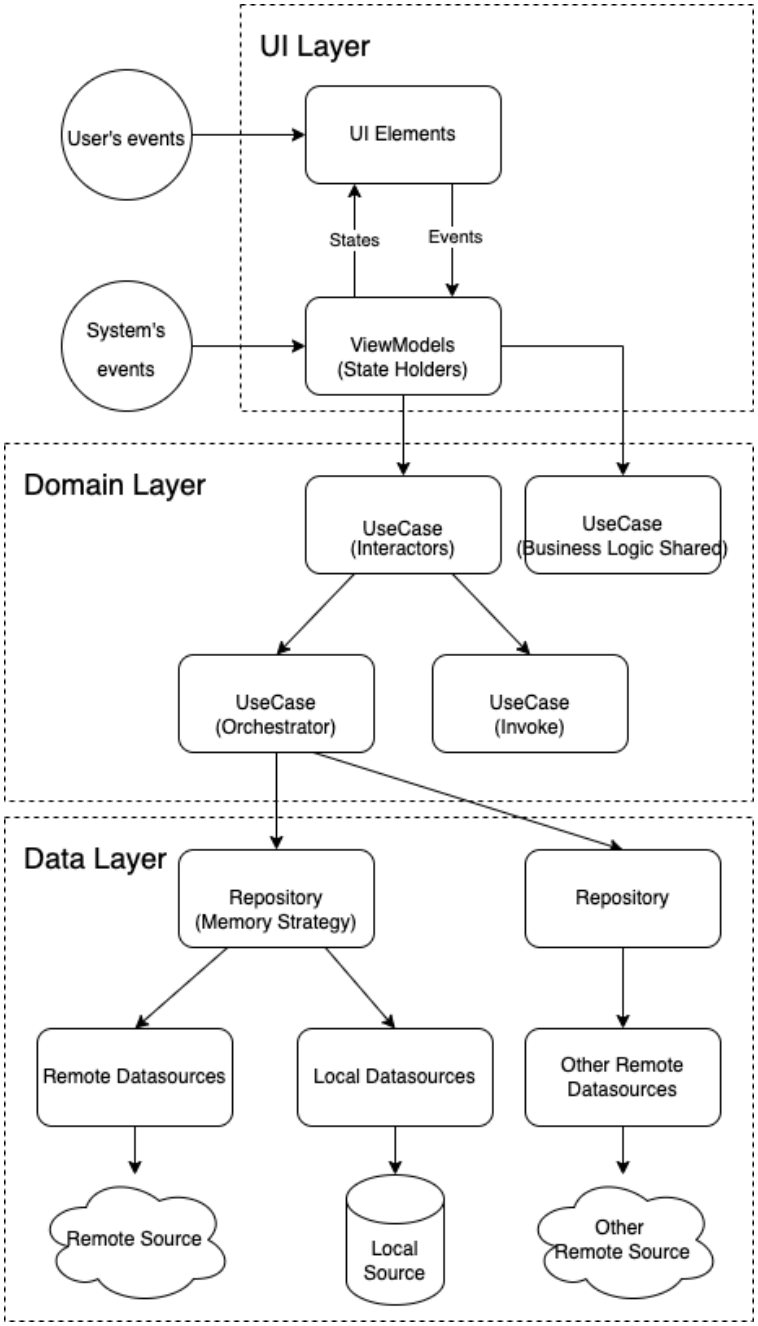

Figure 4.4 Architecture Layers

About other layers

Other auxiliary layers that complement the main layers of the architecture will be:

- **Main**: Contains the base artifacts of the application, such as MainActivity, Application, and ApplicationState, among others.
- **Common**: Contains cross-application artifacts such as navigation definitions, utilities used for other layers, and dependency manager, among others.

About the use of Ports

Clean Architecture, recommends including *ports* between the borders of the different layers[26].

This technique allows inverting control, decoupling the components that communicate between the boundaries of each layer.

That approach will add a better level of maintainability and adaptability to the design.

Our sample application (*OrderNow*) will add *ports* between the *Domain* and *Data* layer.

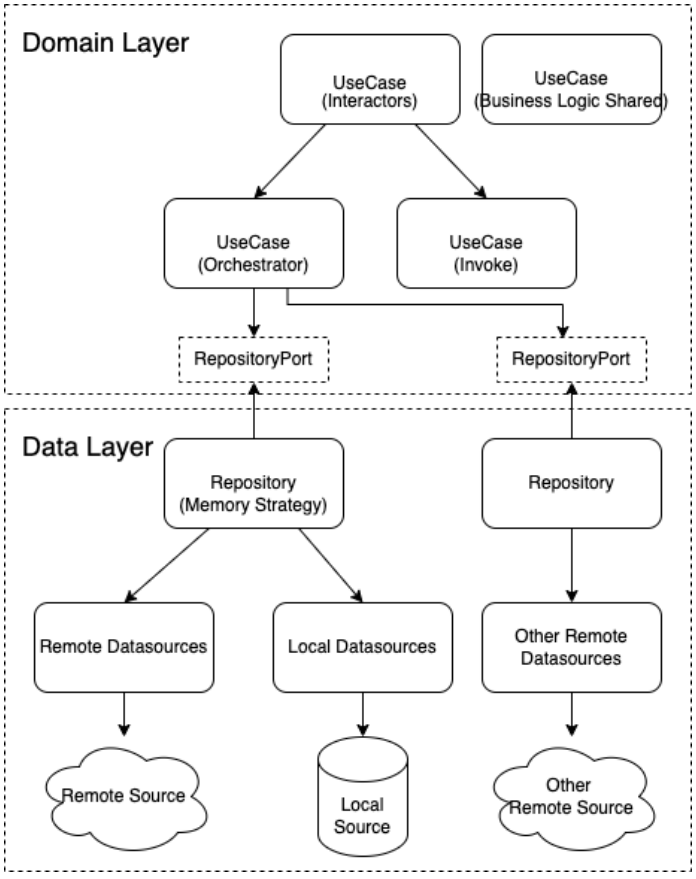

Figure 4.5 Ports between layers

[26] Ports and Adapters: Clean Architecture in iOS

Organizing directories

In our *OrderNow* example, the layers will be organized monolithically through directories in a single module for simplicity.

I leave it to the reader's discretion if, later in his projects, he decides to separate the layers by dedicating a module to each one of them.

The organization by directories is done through two definitions:

- In UI Layer, it will use organization by features.
- In Domain Layer and Data Layer, it will use organization by components.

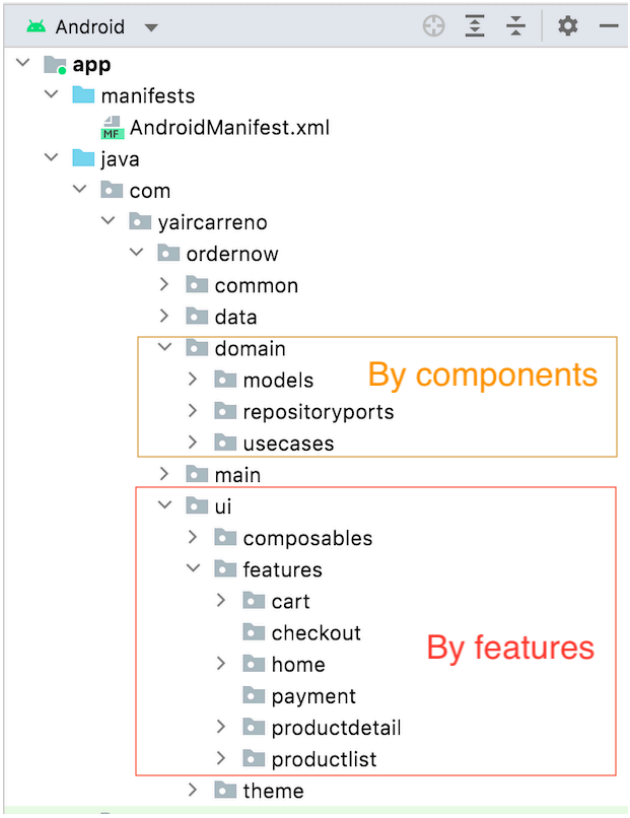

Figure 4.6 Directory organization

Nomenclature and naming elements

For component naming, we will use the following rules

Using suffixes

A suffix will be used in the component name only when:

- The name of the containing package does not infer its type.
- It is necessary to enforce the type of structure that the component represents, for instance, *ProductRepository*.
- To avoid confusion between component types, for instance, a model might be named *Category* and its repository *CategoryRepository*.

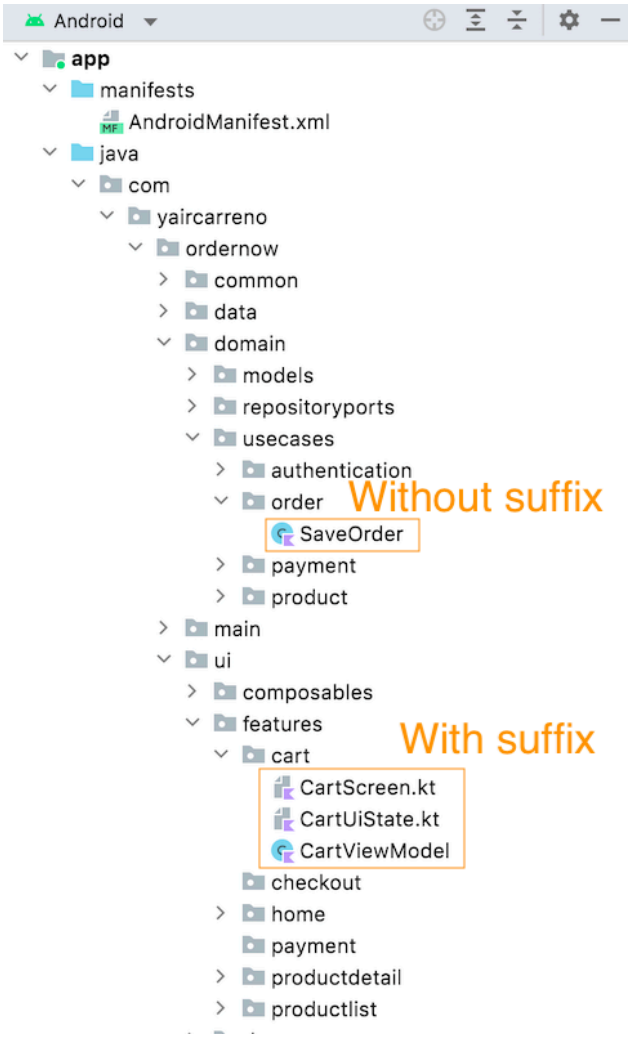

Figure 4.7 Naming with suffix

Naming packages

The name of the application packages must be lowercase, without separators or *camelcase*.

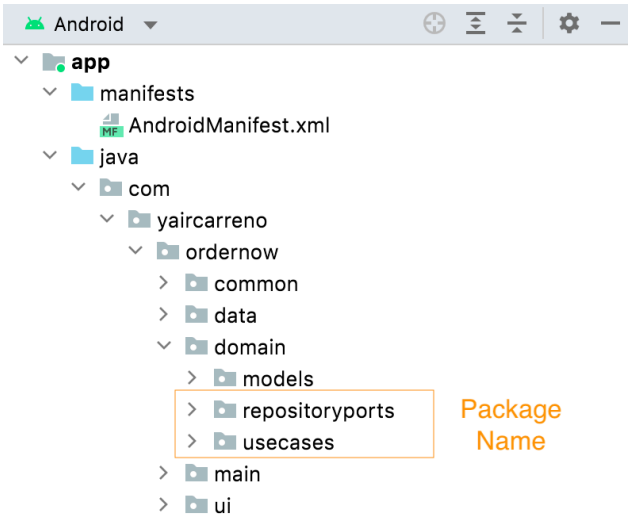

Figure 4.8 Naming packages

Naming components

For the name of the components type *UseCases*, the action representing the operation in the use case (*do*, *get*, *update*, *save*, *send*, *delete*, *add*) is used as a prefix.

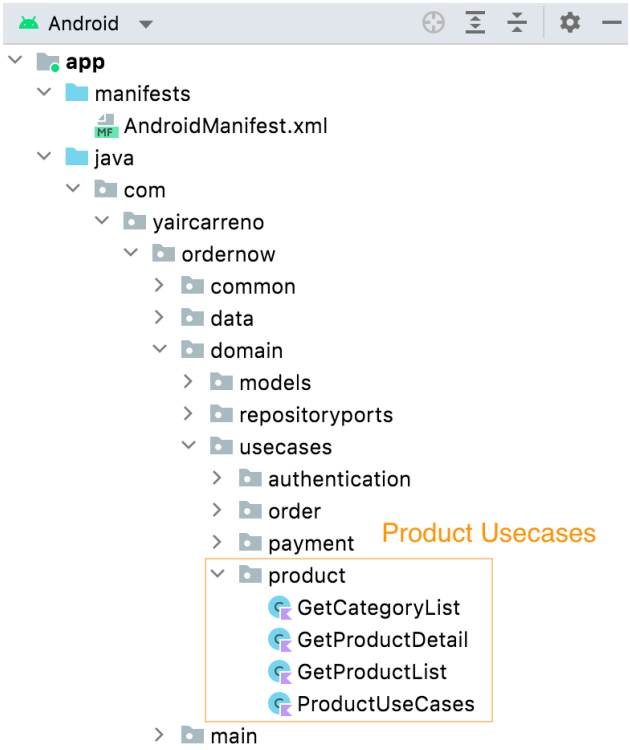

Figure 4.9 Naming UseCases

Naming composable functions

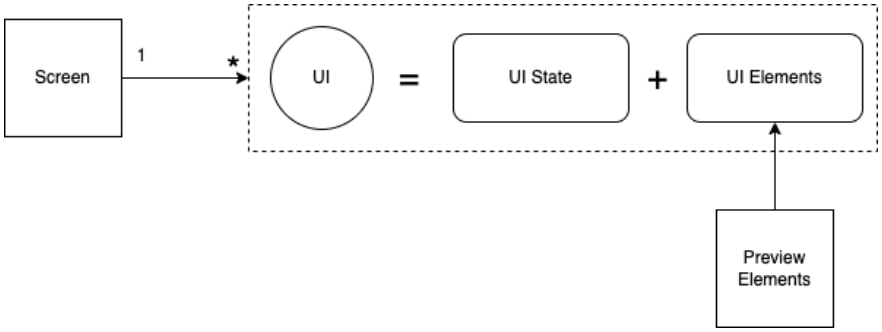

Figure 4.10 Naming Composables

For the definition of the UI components, that is, the *Composables functions*, the following denominations will be used that take as a reference the *Google documentation*[27] in the architecture guide:

Screen: Suffix used for *composables* that represent the entire screen.

UI: Suffix used for *composables* that combine the states of the view (*UI State*) with the graphical representation of the components (*UI Elements*).

Elements: Suffix used for *composables* that define the components of the UI library (Buttons, Layouts, Checkbox, TextFields, among others) that make up the view.

Preview: Prefix used for *preview composables* of the views (Elements). Screen and UI composables could also be previewed, but it becomes more complex due to the dependency on states and other variables.

Keep in mind

It is essential to clarify that there could be exceptions where it is not necessary to define all *composables* types and omit the definition of some. Depending on the degree of complexity of the screens, it will be decided which ones apply and which do not.

Summary

In this chapter, I wanted to describe the architecture definitions to follow before starting the implementation.

The rules used for the organization of the application project are also clarified.

I must clarify that the definitions given in this chapter are recommendations. The reader will be able to make his customizations or assume the conventions and rules that suit him best or with which he feels comfortable in his implementations.

In the next chapter, we will start implementing *OrderNow*, and the first thing to be built will be its *skeleton*, that is, its main structure.

[27] Guide to app architecture: Define UI state

Chapter 5: Skeleton: Main structure

Creating Screens and ViewModels

In the previous chapters, we have already analyzed the theoretical concepts and the organization we will give to the *OrderNow* project.

In this chapter, we will start with implementing the initial structure and template, which will unite each application piece.

The first step will be to add the following main screens with their respective *ViewModels*:

- Home
- Product List
- Product Detail
- Cart

An example of the added elements is shown in the following figure 5.1:

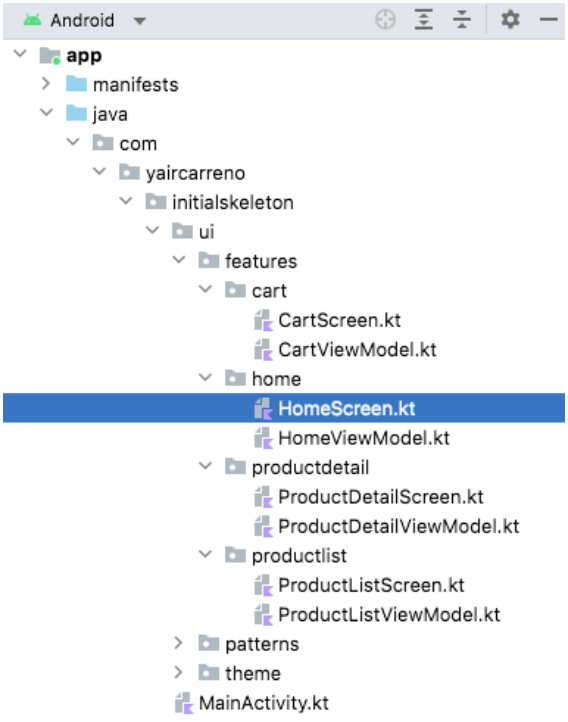

Figure 5.1 Sreens and ViewModels

Source code

You can find the source code at InitialSkeleton[28].

[28] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_05/InitialSkeleton

We'll use the *Hilt*[29] dependency manager in the app to bind each *ViewModel* to its respective screen.

To do this, we must first install *Hilt* in the application, as indicated in the Google Installing Hilt[30].

During the *Hilt* installation process, the definition of the *Application* type class will be requested, which in our example project will be *OrderNowApplication* like this:

Code snippet 5.1

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

Please don't forget to register the *OrderNowApplication* class also in *AndroidManifiest.xml*:

Code snippet 5.2

```
<application
    android:name=".main.OrderNowApplication"
    android:allowBackup="true"
    ... >
```

Additionally, we install the *Navigation Compose* library to the project, which will allow the view (*Composable*) to obtain the instance of its respective *ViewModel* during navigation:

Code snippet 5.3

```
dependencies {
    implementation 'androidx.hilt:hilt-navigation-compose:1.0.0'
}
```

Once the previous configuration has been correctly carried out in the project, we can inject the *ViewModel* into the view like this[31]:

Code snippet 5.4: Home ViewModel

```
@HiltViewModel
class HomeViewModel @Inject constructor() : ViewModel() {
    ...
}
```

[29] Dependency injection with Hilt
[30] https://developer.android.com/training/dependency-injection/hilt-android#setup
[31] Inject ViewModel objects with Hilt

Code snippet 5.5: Home Screen

```
@Composable
fun HomeScreen(viewModel: HomeViewModel = hiltViewModel()
) {
    ...
}
```

Each screen will be associated with its corresponding *ViewModel* through the *Hilt* dependency manager.

Up to this point, we would have integrated the following architecture components: *Compose*, *Navigation* and *ViewModel*.

That is a perfect combination of tools through **Jetpack**, and in later chapters, we will see its potential in mobile development. Thanks, Google team, you are the best!

Source code

You can find the source code at InitialSkeleton[32].

UI patterns: TopAppBar y BottomAppBar

Through the **Scaffold** component, we can implement in our applications two of the most common UI patterns in Material Design: *TopAppBar* and *BottomAppBar*.

A *Scaffold*[33] is a detailed view (*composable*) that will allow us to implement these patterns in the following way:

Code snippet 5.6: Scaffold

```
Scaffold(
    topBar = {
        TopAppBar { /* Top app bar content */ }
    },
    bottomBar = {
        BottomAppBar { /* Bottom app bar content */ }
    }
) { contentPadding ->
    // Screen content
}
```

Easy, don't you think?

In the 5.6 code snippet, we define the *topBar*, *bottomBar*, and the (not yet added) content of the screen.

In *Scaffold*, the *topBar* and *bottomBar* sections are optional; that is, the definition of some of these parts can be omitted. Additionally, in *Scaffold*, there are two more components that we can declare:

[32] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_05/InitialSkeleton
[33] Scaffold

- *scaffoldState*
- *snackbarHost*

In the next chapter, we will see how each is used. Let's move on with just the definition of *topBar* and *bottomBar*.

Now that we know how it will include these UI patterns in our application, the next step is to create the views (*composables*) representing both *TopAppBar* and *BottomAppBar*.

A directory called *patterns* is organized, and within it, the two views, *OrderNowTopBar* and *OrderNowBottomBar*, are added as follows:

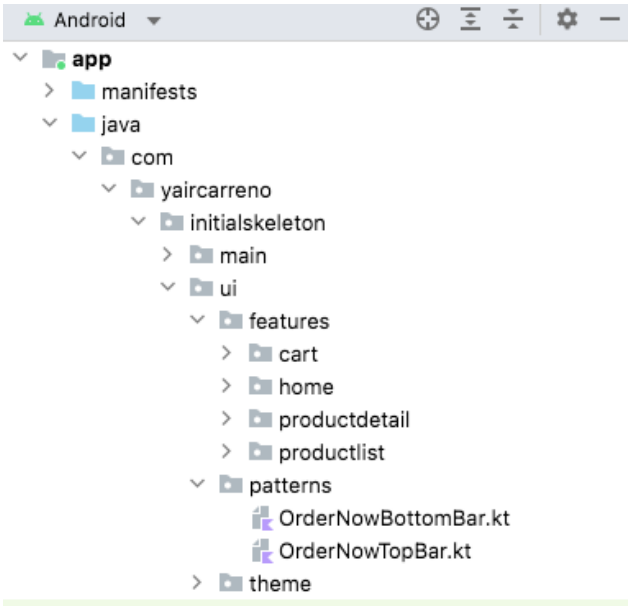

Figure 5.2 TopAppBar and BottomAppBar Composables

OrderNowTopBar

The implementation of *OrderNowTopBar* is simple. We only have to implement it initially in the following way:

Code snippet 5.7: TopAppBar

```
@Composable
fun OrderNowTopBar() {
    TopAppBar(
        title = {
            Text(
                text = stringResource(id = AppString.app_name),
                textAlign = TextAlign.Center,
                modifier = Modifier.fillMaxWidth()
            )
        },
```

```
        backgroundColor = MaterialTheme.colors.background,
        contentColor = contentColorFor(MaterialTheme.colors.background)
    )
}
```

The previous implementation does not yet contain elements such as the option to go back; however, in the section Adding "Back" to the TopAppBar in chapter 7, it will include such a navigation option with a strategy that uses the state.

OrderNowBottomBar

The implementation of *OrderNowBottomBar* could be a bit more elaborate since we need to include the navigation between the screens. However, we will leave that implementation detail for the following chapter, Navegación en la aplicación.

For now, we will include a static definition without navigation.

Code snippet 5.8: OrderNowBottomBar

```
@Composable
fun OrderNowBottomBar() {
    val selectedIndex = remember { mutableStateOf(0) }
    BottomNavigation(
        backgroundColor = MaterialTheme.colors.background,
        contentColor = contentColorFor(MaterialTheme.colors.background),
        elevation = 10.dp
    ) {
        BottomNavigationItem(icon = {
            Icon(imageVector = Icons.Default.Home, "")
        },
            label = { Text(text = "Home") },
            selected = (selectedIndex.value == 0),
            unselectedContentColor = Color.Gray,
            selectedContentColor = orange,
            onClick = {
                selectedIndex.value = 0
            })

        BottomNavigationItem(icon = {
            Icon(imageVector = Icons.Default.ShoppingCart, "")
        },
            label = { Text(text = "Cart") },
            selected = (selectedIndex.value == 1),
            unselectedContentColor = Color.Gray,
            selectedContentColor = orange,
            onClick = {
                selectedIndex.value = 1
```

```
29                 })
30         }
31 }
```

At this point, we already have the definitions of the *Screens*, *ViewModels*, and *Scaffold* (which includes *OrderNowBottomBar*, and *OrderNowTopBar*).

The next step is to put all the pieces together, and we will do this in the next section.

Putting all together

The first task is to create a directory called *main*. This directory will be transversal and will have the base classes or structures of the App.

Inside that directory, we place the *Application* class, move the *MainActivity* there, and the Main Screen of the application, which we will name *OrderNowScreen*, as shown in the following image.

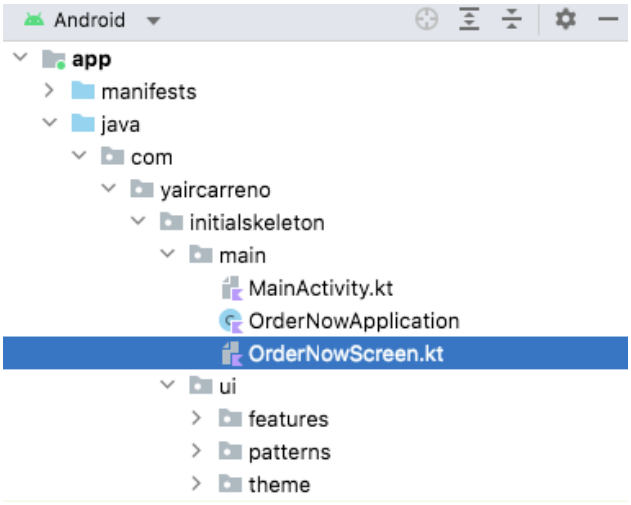

Figure 5.3 Main Components

Now we modify *MainActivity.kt* class so that it loads the login screen to the *OrderNowScreen* application like so:

Code snippet 5.9

```
1  @AndroidEntryPoint
2  class MainActivity : ComponentActivity() {
3      override fun onCreate(savedInstanceState: Bundle?) {
4          super.onCreate(savedInstanceState)
5          setContent {
6              OrderNowScreen()
7          }
8      }
9  }
```

Then, in *OrderNowScreen* view, we define the *Scaffold* of our application like this:

Code snippet 5.10

```
@Composable
fun OrderNowScreen() {
    InitialSkeletonTheme {
        Surface(
            modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colors.background
        ) {
            Scaffold(
                topBar = { OrderNowTopBar() },
                bottomBar = { OrderNowBottomBar() }
            ) { contentPadding ->
                println(contentPadding)
            }
        }
    }
}
```

When you run the application, the result should be similar to the following image:

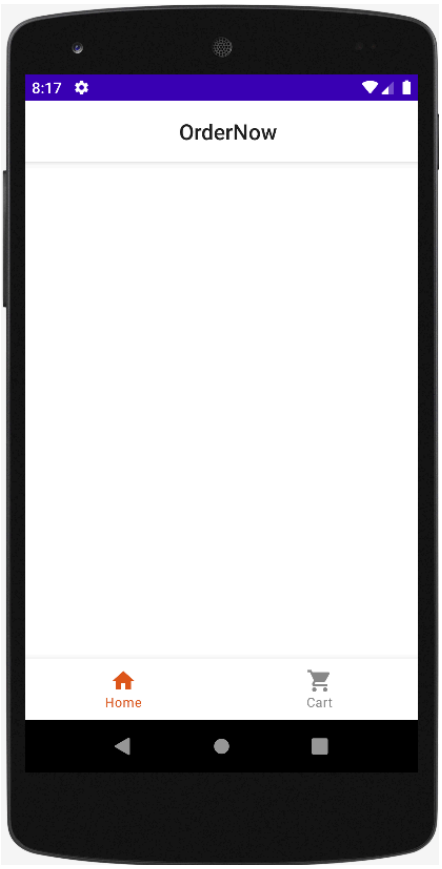

Figure 5.4 First version (Skeleton)

Source code

You can find the source code at InitialSkeleton[34].

Summary

In this chapter, we have built the initial structure of the *OrderNow* project.

The components defined and implemented here will be the basis to continue in the next chapter with navigation.

As we go through the chapters, we will improve the implementation of each piece of *OrderNow* to obtain; as a result, our e-commerce will be designed and implemented with the best possible practices.

[34] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_05/InitialSkeleton

Chapter 6: Designing navigation in App

In the implementation made in the previous chapter, Skeleton: Main structure, several Jetpack architecture components were left installed that will be used in this chapter.

Componentes like *Composables*, *ViewModels*, *Navigation* and *Hilt*. Also, we left the base structure integrating the *TopAppBar* and *BottomAppBar* UI patterns through *Scaffold*.

To continue improving the implementation, we need to add a new key tab, *"A general App's state"*.

App's State: A general state

The first chapter, Design Principles, discussed the **state**'s essential role in modern Android applications.

Three types of states that could exist in the design: *Property UI's state*, *Component UI's state*, and *Screen UI's state*.

In addition to these states, we can define a new type of state, the **App's state**.

This new state will define the general state of the application. It will be used for navigation purposes between screens, presentation of spontaneous messages (snackbars), and other available processes in the App.

In *main* directory, we are going to define the class called *OrderNowState*, which will be our *State Holder* that will represent this type of state.

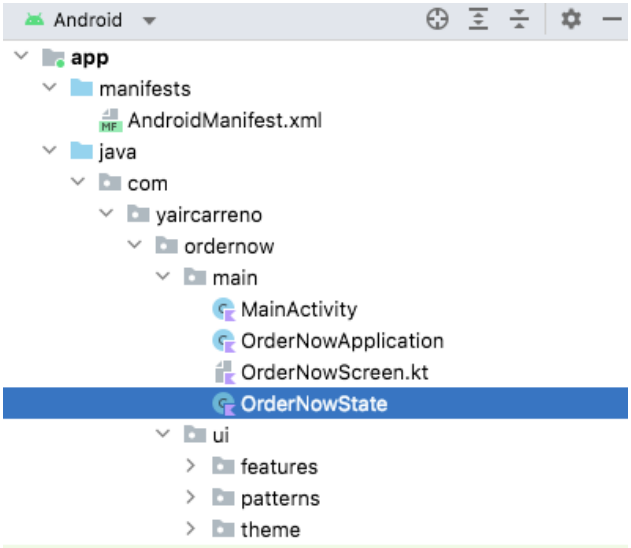

Figure 6.1 App's state class

Next, we perform the initial implementation of *OrderNowState* like this:

Code snippet 6.1

```
@Composable
fun rememberAppState(
    scaffoldState: ScaffoldState = rememberScaffoldState(),
    navController: NavHostController = rememberNavController(),
    resources: Resources = resources(),
    coroutineScope: CoroutineScope = rememberCoroutineScope()
) = remember(
    scaffoldState,
    navController,
    resources,
    coroutineScope
) {
    OrderNowState(scaffoldState, navController, resources, coroutineScope)
}

class OrderNowState(
    val scaffoldState: ScaffoldState,
    val navController: NavHostController,
    private val resources: Resources,
    coroutineScope: CoroutineScope
)
```

And later, we modify our *OrderNowScreen* view to include the previously defined state like so:

Code snippet 6.2

```
@Composable
fun OrderNowScreen() {
    OrderNowTheme {
        Surface(
            modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colors.background
        ) {
            val appState = rememberAppState()
            Scaffold(
                scaffoldState = appState.scaffoldState,
                topBar = { OrderNowTopBar() },
                bottomBar = { OrderNowBottomBar() }
            ) { contentPadding ->
                println(contentPadding)
            }
        }
    }
}

```

```
@Composable
@ReadOnlyComposable
fun resources(): Resources {
    LocalConfiguration.current
    return LocalContext.current.resources
}
```

Note: It was also necessary to add to *OrderNowScreen*, the resources function with which it will access the App resources.

The lines of code to highlight the change are the following:

Code snippet 6.3

```
val appState = rememberAppState()

Scaffold(
    scaffoldState = appState.scaffoldState,
    ...
) { contentPadding ->
    ...
}
```

With the code above, we can now tell *Scaffold* what state it should take as reference: the state of the App. That later will allow consistent navigation operations between views, presentation of spontaneous messages, and other exclusive tasks of *View* taking as *Source of truth* only **AppState**.

Source code

You can find the source code at NavigationBottomBar[35].

Now that we have a state for the APP defined, we can proceed with implementing the app's navigation.

Defining the navigation map

The navigation strategy that we use in the application consists of the following elements:

- **NavHost**: It is the component responsible for presenting the result of the navigation in the view. The result of the navigation is determined by the *NavigationController* and the definitions given in the *navigation graph*.
- **AppSoGraph**: It is the implementation of the *navigation graph*. The view or *Composable* to which it should direct the navigation according to the specified *route* is specified.

[35] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_06/navigationBottomBar/OrderNow

- **Screen routes**: They are the different screens of the application that can be reached using navigation. It doesn't matter if the navigation is activated from an options menu, a link, a button, or any other active agent. Each of these screens will have a unique *route* associated with them.

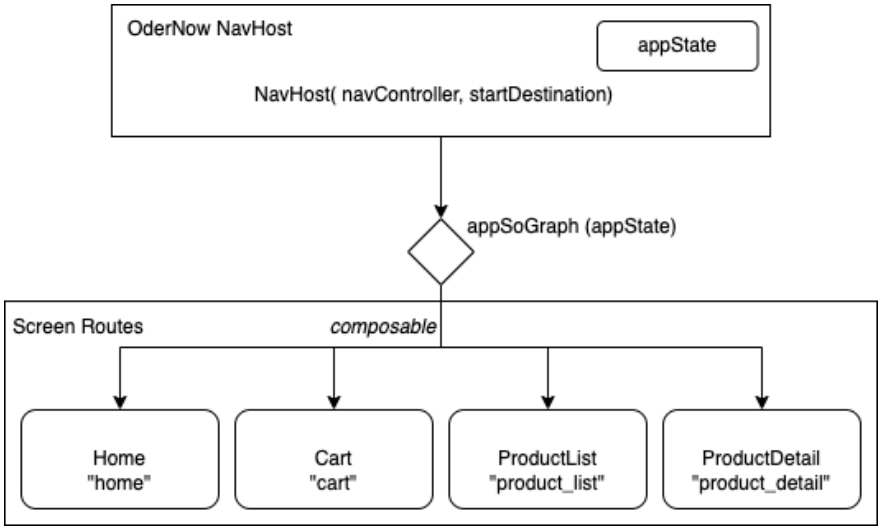

Figure 6.2 General Navigation Graph

We will proceed to include each of these elements in *OrderNow*.

OrderNowScreenRoute

First, a new traversal directory called *common -> navigation* is created. In that package, we add a class called *OrderNowScreenRoute* like this:

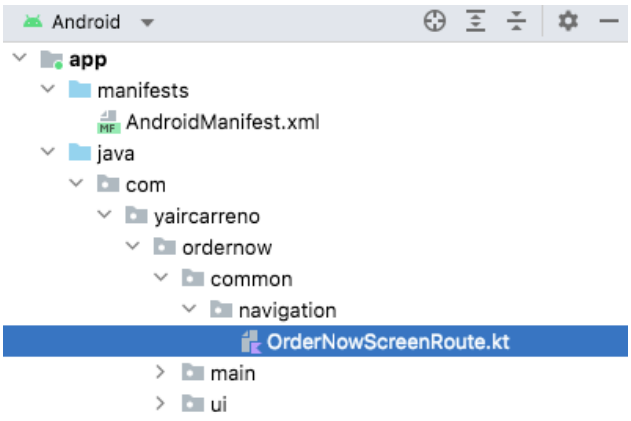

Figure 6.3 Screen routes

Within this class, the screens to which it will be possible to navigate are defined as follows.:

Code snippet 6.4

```
sealed class OrderNowScreenRoute (val route: String) {

    object Home : OrderNowScreenRoute("home")
    object Cart : OrderNowScreenRoute("cart")
    object ProductList : OrderNowScreenRoute("product_list")
    object ProductDetail : OrderNowScreenRoute("product_detail")
}
```

OrderNowNavHost y AppSoGraph

Now, we create *OrderNowNavHost* class that will represent the *NavHost* of the App like this:

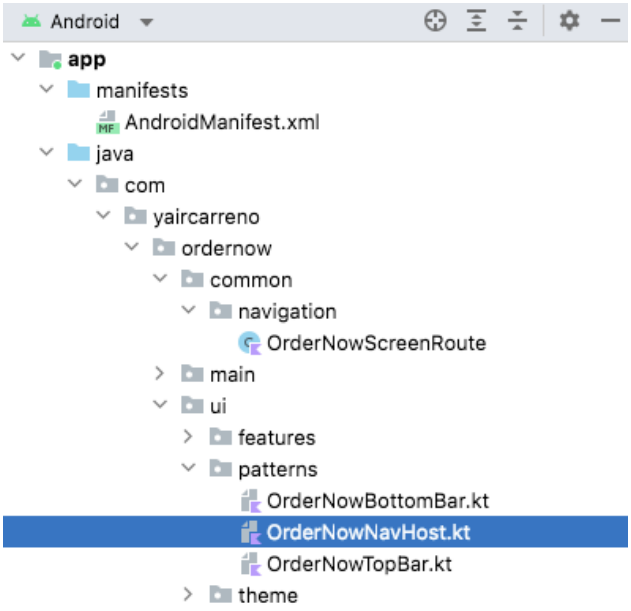

Figure 6.4 NavHost APP

Code snippet 6.5

```
@Composable
fun OrderNowNavHost(appState: OrderNowState) {
    NavHost(
        navController = appState.navController,
        startDestination = ...
    ) {
        appSoGraph(appState)
    }
}

fun NavGraphBuilder.appSoGraph(appState: OrderNowState) {
```

```
12      ...
13  }
```

From the previous code, code snippet 6.5, what we should highlight are the following definitions:

- *OrderNowNavHost* requires knowing the state of the APP.
- *NavController* is hosted and taken from the APP state.
- Navigation map (*appSoGraph*), will be created based on the state of the APP and is an extension defined within *OrderNowNavHost*.

To continue and finish the implementation of the navigation in *OrderNow*, we must add a helper class described below.

NavigationBarSection

NavigationBarSection is a helper class representing the screening group that makes up the *bottomBar menu* of the application.

Remember that we can start navigation from action in an options menu or other UI components such as links, buttons, or internal redirects.

In the next section, we'll make the changes for internal redirects (from Buttons, Links, and so on); for now, let's focus on navigation from *BottomBar*.

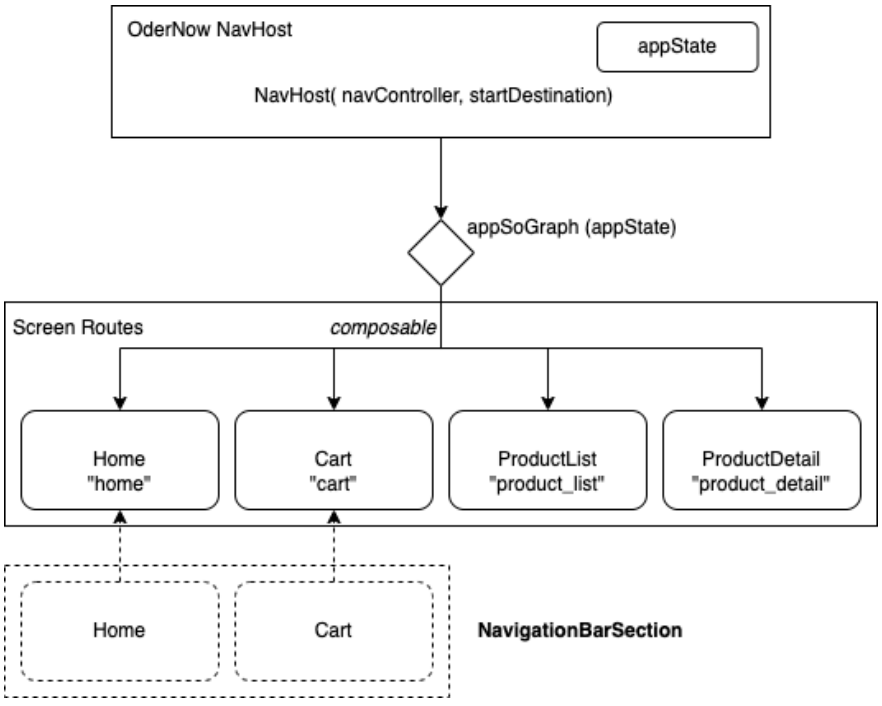

Figure 6.5 NavigationBarSection

In figure 6.5, we see how the *NavigationBarSection* helper class will group only the *Home* and *Cart* screens, which are the options we want to enable from the BottomBar Menu.

This class will be placed in the navigation directory like this:

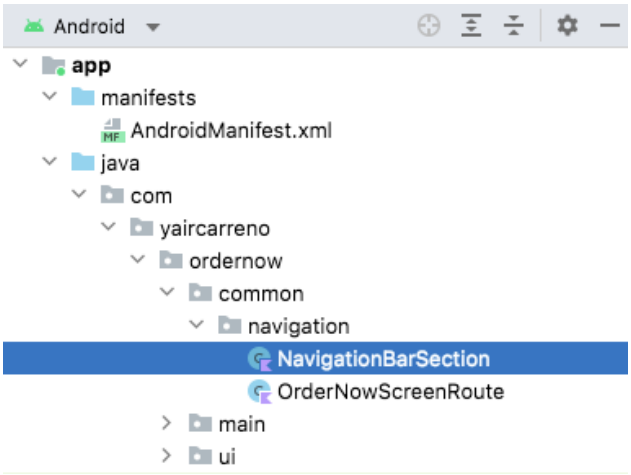

Figure 6.6 NavigationBarSection Class

And its implementation would be like this:

Code snippet 6.6

```
sealed class NavigationBarSection(
    @StringRes val title: Int,
    val icon: ImageVector,
    val route: String
) {
    companion object {
        val sections = listOf(
            Home,
            Cart
        )
    }

    object Home : NavigationBarSection(
        title = AppText.home,
        icon = Icons.Filled.Home,
        route = OrderNowScreenRoute.Home.route
    )

    object Cart : NavigationBarSection(
        title = AppText.cart,
        icon = Icons.Filled.ShoppingCart,
        route = OrderNowScreenRoute.Cart.route
    )
}
```

With the helper class added to the project, we proceed to update the *OrderNowNavHost* class like so:

Code snippet 6.7

```
@Composable
fun OrderNowNavHost(appState: OrderNowState, paddingValues: PaddingValues) {
    NavHost(
        navController = appState.navController,
        startDestination = NavigationBarSection.Home.route,
        modifier = androidx.compose.ui.Modifier.padding(paddingValues)
    ) {
        appSoGraph(appState)
    }
}

fun NavGraphBuilder.appSoGraph(appState: OrderNowState) {

    // Home Screen Graph
    composable(NavigationBarSection.Home.route) {
        HomeScreen()
    }

    // Cart Screen Graph
    composable(NavigationBarSection.Cart.route) {
        CartScreen()
    }

    // Product List Screen Graph
    composable(OrderNowScreenRoute.ProductList.route) {
        ProductListScreen()
    }

    // Product Detail Screen Graph
    composable(OrderNowScreenRoute.ProductDetail.route) {
        ProductDetailScreen()
    }
}
```

In the previous code snippet 6.7, the implementation of *appSoGraph* function is an extension of *NavGraphBuilder*, and there, we specify the navigation map for each screen of the App.

Also, through *startDestination* parameter, the default screen that will be the first to be presented is specified, that is, *Home* screen.

The next step to adopt the change would be to update the class called *OrderNowBottomBar* like so:

Code snippet 6.8

```
@Composable
fun OrderNowBottomBar(navController: NavHostController) {

    val navBackStackEntry by navController.currentBackStackEntryAsState()
    val currentDestination = navBackStackEntry?.destination
    BottomNavigation(
        backgroundColor = MaterialTheme.colors.background,
        contentColor = contentColorFor(MaterialTheme.colors.background),
        elevation = 10.dp
    ) {
        NavigationBarSection.sections.forEach { section ->
            val selected =
                currentDestination?.hierarchy?.any {
                    it.route == section.route
                } == true
            BottomNavigationItem(
                icon = {
                    Icon(
                        imageVector = section.icon,
                        contentDescription = stringResource(section.title)
                    )
                },
                label = { Text(text = stringResource(section.title)) },
                selected = selected,
                unselectedContentColor = Color.Gray,
                selectedContentColor = orange,
                onClick = {
                    navController.navigate(section.route) {
                        popUpTo(navController.graph.findStartDestination().id) {
                            saveState = true
                        }
                        launchSingleTop = true
                        restoreState = true
                    }
                })
        }
    }
}
```

For each item added in the *NavigationBarSection*, an option will be displayed in *BottomBar*. This implementation of *OrderNowBottomBar* is cleaner than the one made in the previous Chapter 5: Skeleton: Main structure.

Then, we update the *OrderNowScreen* view again like so:

Code snippet 6.8

```
@Composable
fun OrderNowScreen() {
    OrderNowTheme {
        Surface {
            val appState = rememberAppState()
            Scaffold(
                ...
                bottomBar = {
                    OrderNowBottomBar(navController = appState.navController)
                }
            ) { contentPadding ->
                OrderNowNavHost(appState, contentPadding)
            }
        }
    }
}
```

Now *OrderNowBottomBar* will require the reference to the *navController* responsible for navigation.

In the content section of *Scaffold*, the instance of *OrderNowNavHost* is added, which receives the general status of the APP as a parameter, as shown in code snippet 6.8.

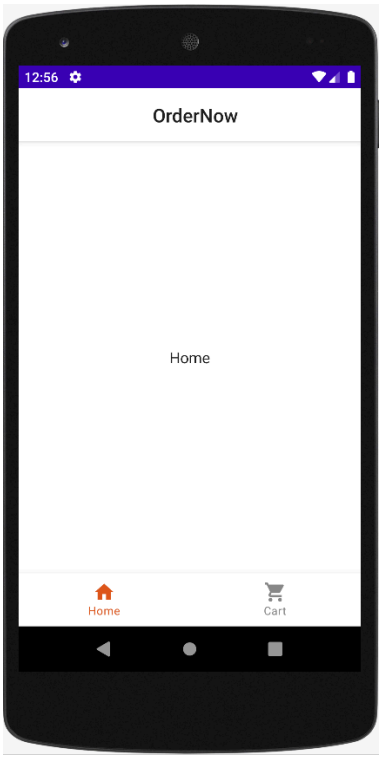

Figure 6.7 Order Now with simple navigation

With these changes, the application should run, as shown in figure 6.7.

Source code

You can find the source code at NavigationBottomBar[36].

Navigation from other UI elements

Now that we have the implementation ready to navigate from the *BottomBar* options, we need the definitions to navigate the application from other UI elements such as *Button*, *Links*, *DeepLinks*, or even programmatically at the request of other internal components of the App.

The first change we are going to make is to add a structure called *OrderNowNavigationState* that will allow us to extend the general state of the App.

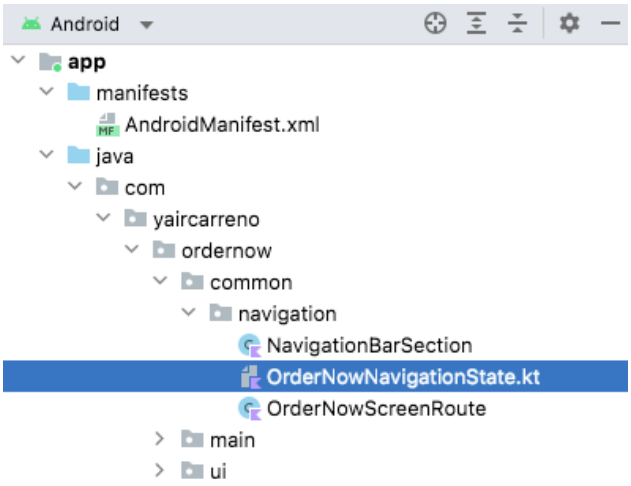

Figure 6.8 OrderNowNavigationState Class

OrderNowNavigationState, are extensions of the general state of the APP, that is, a group of extensions of the *OrderNowState* state used for navigation purposes. We will also use this structure to centralize there the navigation logic that depends on the state of the APP.

[36]https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_06/navigationBottomBar/OrderNow

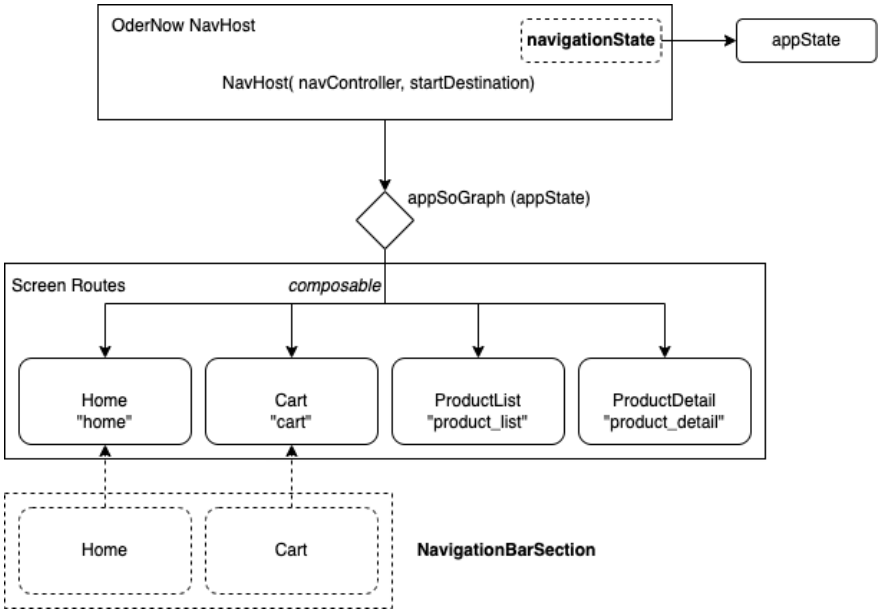

Figure 6.9 OrderNowNavigationState

Implementation of *OrderNowNavigationState* would be as follows:

Code snippet 6.9

```
fun OrderNowState.popUp() {
    navController.popBackStack()
}

fun OrderNowState.navigate(route: String) {
    navController.navigate(route) {
        launchSingleTop = true
    }
}

fun OrderNowState.navigateAndPopUp(route: String, popUp: String) {
    navController.navigate(route) {
        launchSingleTop = true
        popUpTo(popUp) { inclusive = true }
    }
}

fun OrderNowState.navigateSaved(route: String, popUp: String) {
    navController.navigate(route) {
        launchSingleTop = true
        restoreState = true
        popUpTo(popUp) { saveState = true }
    }
}

```

```
fun OrderNowState.clearAndNavigate(route: String) {
    navController.navigate(route) {
        launchSingleTop = true
        popUpTo(0) { inclusive = true }
    }
}
```

A few changes are made to the *Home*, *ProductList* and *ProductDetail* screens, as shown in the following figure:

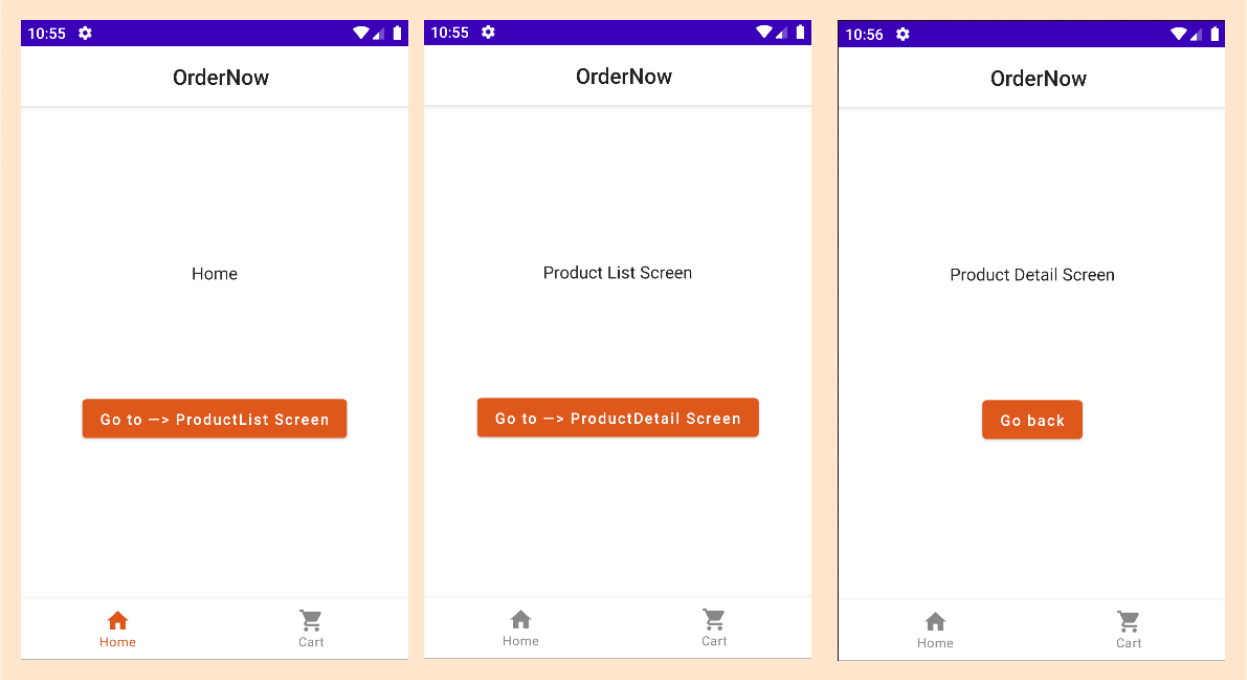

Figure 6.10 Home, ProductList, PrductDetail Screens

In-Home view, for example, the navigation action is executed by a button in the following way:

Code snippet 6.10

```
@Composable
fun HomeScreen(
    goToProductList: () -> Unit,
    modifier: Modifier = Modifier,
    viewModel: HomeViewModel = hiltViewModel()
) {
    Column {
        ...
        Button(
            onClick = goToProductList,
            ...
        ) {
```

```
            Text(text = stringResource(id = AppText.to_product_list_screen))
        }
    }
}
```

The important part of the code snippet 6.10 code is the definition of the action `onClick = goToProductList` of the *"Go to —> ProductList Screen"* button, where through the state hoisting technique explained in the Design Principles chapter, we delegate the action `goToProductList` to the *appSoGraph* navigation map defined in *OrderNowNavHost* like so:

Code snippet 6.11

```
fun NavGraphBuilder.appSoGraph(appState: OrderNowState) {

    val goToListFromHome: () -> Unit = {
        appState.navigateSaved(OrderNowScreenRoute.ProductList.route,
                               OrderNowScreenRoute.Home.route)
    }

    // Home Screen Graph
    composable(NavigationBarSection.Home.route) {
        HomeScreen(goToProductList = goToListFromHome)
    }

    ...
}
```

Recall that the function *navigateSaved* is part of the extensions defined in the structure called *OrderNowNavigationState*.

The same implementation is applied for navigation to the other *ProductList* and *ProductDetail* screens in such a way that the performance in *OrderNowNavHost* is as follows:

Code snippet 6.12

```
fun NavGraphBuilder.appSoGraph(appState: OrderNowState) {

    val homeRoute = OrderNowScreenRoute.Home.route
    val listRoute = OrderNowScreenRoute.ProductList.route
    val detailRoute = OrderNowScreenRoute.ProductDetail.route

    val goToListFromHome: () -> Unit = {
        appState.navigateSaved(listRoute, homeRoute)
    }

    val goToDetailFromList: () -> Unit = {
        appState.navigateSaved(detailRoute, listRoute)
    }
```

```
14
15      val goBack: () -> Unit = {
16          appState.popUp()
17      }
18
19      composable(NavigationBarSection.Home.route) {
20          HomeScreen(goToProductList = goToListFromHome)
21      }
22
23      composable(NavigationBarSection.Cart.route) {
24          CartScreen()
25      }
26
27      composable(OrderNowScreenRoute.ProductList.route) {
28          ProductListScreen(goToProductDetail = goToDetailFromList)
29      }
30
31      composable(OrderNowScreenRoute.ProductDetail.route) {
32          ProductDetailScreen(goToBack = goBack)
33      }
34  }
```

In summary, these would be the navigation definitions (navigation map) configured in the previous code snippet 6.12 code:

- Navigate from *Home* screen to *ProductList* screen.
- Navigate from *ProductList* screen to *ProductDetail* screen.
- Navigate from *ProductDetail* screen to the previously presented screen.
- Navigate from *BottomBar* menu to *Home* screen.
- Navigate from *BottomBar* menu to *Cart* screen.

Source code

You can find the source code at NavigationFromOthers[37].

At this point, we already have a great basic implementation of navigation in the *OrderNow* App. However, something is missing.

Any idea what might be missing?

We must include in the navigation the sending of data or information between screens.

That will be very easy to do thanks to the implementation that we have designed in this chapter, and we will see how to do it in Chapter 7: Implementing "Features".

[37] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_06/navigationFromOthers/OrderNow

Summary

We have concluded with this chapter the design of the main parts of the application that will be the foundations to add the functionalities later.

We know that navigation is a crucial part of the application and that we must consider it from the beginning of the APP's design.

In this chapter, we have used Google's recommendations regarding keeping the state in mind for navigation logic.

Also, the reader can appreciate that we do not involve the ViewModel or View directly in our strategy. That makes it flexible for testing purposes to check the navigation.

Chapter 7: Implementing “Features”

Before starting

And finally, we have arrived at the part of implementing functionalities.

The first screen that we are going to implement will be the *Home* or main screen of the App, as shown in the following figure 7.1:

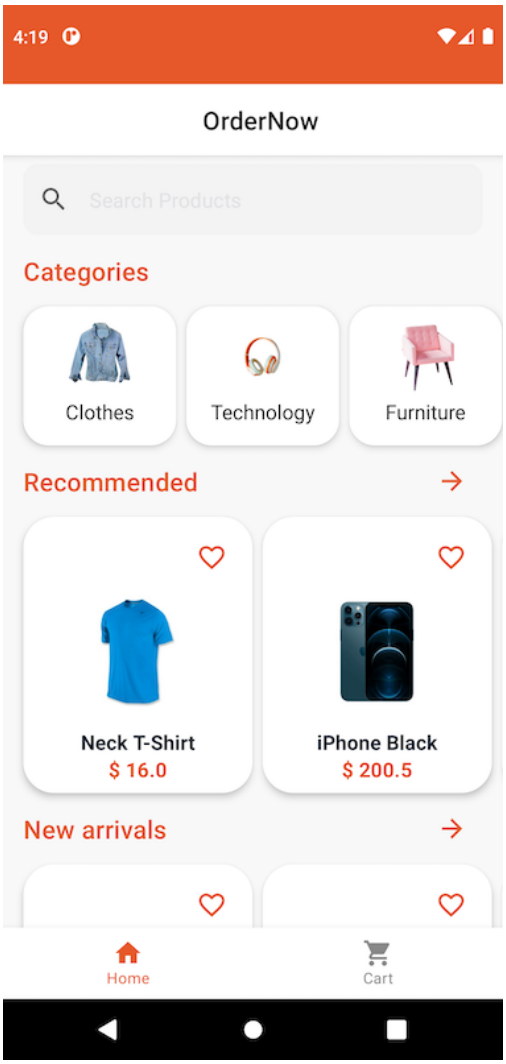

Figure 7.1 Home Screen

However, before we begin the implementation of the screen, we must ask ourselves the following questions:

- What information or data does the view require?

- How will the data be managed in the view?
- Where does the data used by the view come from?

The above reasoning is valid if we are designing declarative views on *Android* or *iOS* (yes, I know this is a dedicated Android book, but we can’t ignore good practices that it can apply between one ecosystem and another)[38].

What information or data does the view require?

Categories and Products.

How will the data be managed in the view?

They will be presented as query information in list form and as horizontal carousels.

Where does the data used by the view come from?

It must come from a *Source of truth*, which in this example will be repositories in the *Data layer*.

The three questions above seem to be somewhat trivial; however, it tells us what is initially required for the implementation:

- Two models, *Category* and *Product*.
- UI components of type lists are *Composables* of type *LazyRow* and *LazyColumn*.
- Two repositories to provide data on *Category* and *Product*.

According to the architecture guide that we analyzed in the Chapter 4: Application Architecture, we also require:

- Add the *Domain* layer that will host the different *Usecases*. For this screen, we’ll use the structure called *ProductUseCases*, which contains the granular use cases: *GetCategoryList*, *GetProductDetail* and *GetProductList*.
- Include *Ports* to increase the decoupling of the *Data* layer. Two ports are identified, for now, one to connect *CategoryRepository* and one to connect *ProductRepository*.

Source code

You can find the source code at OrderNow App[39].

[38] Data Essentials in SwiftUI
[39] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_07/OrderNow

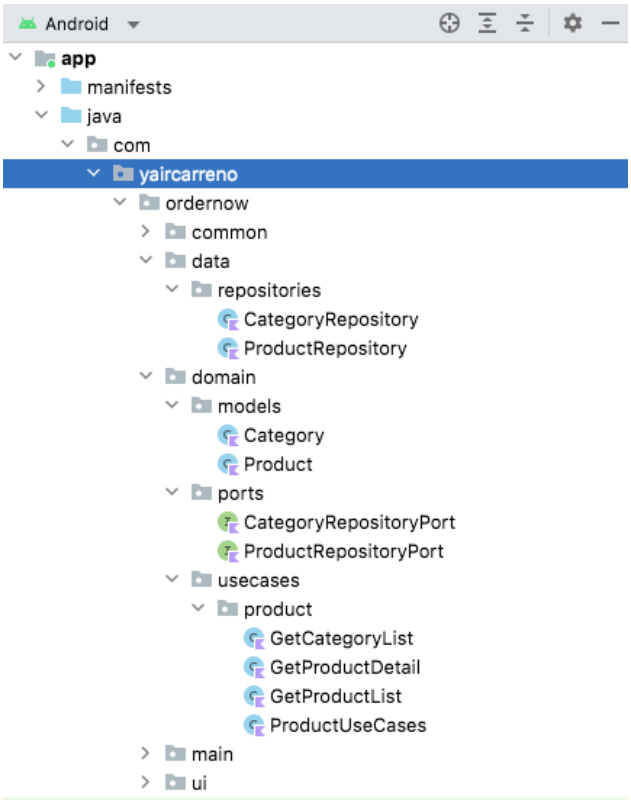

Figure 7.2 Domain and Data components

Implementations in Domain Layer

Models

Code snippet 7.1

```
data class Category(
    val id: String,
    val name: String,
    val label: String,
    val urlImage: String
)

data class Product(
    val id: String,
    val name: String,
    val description: String? = null,
    val urlImage: String,
    val price: Double,
    val category: Category
)
```

Repository Ports

Code snippet 7.2

```
interface CategoryRepository {
    fun getCategories(): Flow<List<Category>>
}

interface ProductRepositoryPort {
    fun getProducts(): Flow<List<Product>>
    fun getProducts(category: Category): Flow<List<Product>>
}
```

UseCases

Code snippet 7.3

```
data class ProductUseCases(
    val getProducts: GetProductList,
    val getProduct: GetProductDetail,
    val getCategories: GetCategoryList
)

class GetCategoryList(
    private val repository: CategoryRepositoryPort
) {

    fun getCategories(): Flow<List<Category>> {
        return repository.getCategories()
    }
}

class GetProductList(
    private val repository: ProductRepositoryPort
) {

    fun getProducts(): Flow<List<Product>> {
        return repository.getProducts()
    }

    fun getProducts(category: Category): Flow<List<Product>> {
        return repository.getProducts(category)
    }
}
```

Implementations in Data Layer

Repositories

Code snippet 7.4

```
class CategoryRepository(
    private val categoryDataSource: CategoryDataSource
) : CategoryRepositoryPort {

    override fun getCategories(): Flow<List<Category>> {
        return categoryDataSource.getCategories()
    }
}

class ProductRepository(
    private val productDataSource: ProductDataSource
) : ProductRepositoryPort {

    override fun getProducts(): Flow<List<Product>> {
        return productDataSource.getProducts()
    }

    override fun getProducts(category: Category): Flow<List<Product>> {
        return productDataSource.getProducts(category)
    }
}
```

DataSources

Code snippet 7.5

```
interface CategoryDataSource {
    fun getCategories(): Flow<List<Category>>
}

interface ProductDataSource {
    fun getProducts(): Flow<List<Product>>
    fun getProducts(category: Category): Flow<List<Product>>
}
```

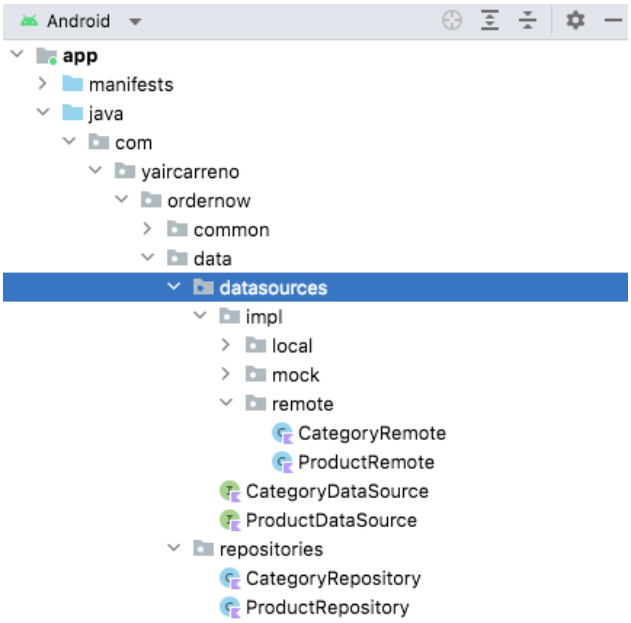

Figure 7.3 DataSource components

Mocking DataSources and Previews

For the *OrderNow* implementation, we will use *mocks* to simulate the information query and the backend payment process.

We will also make a simple *mocks* for the information presented in the *Previews* of each of the *Composables* screens.

In the *datasources* -> *impl* -> *mock* directory, add the following implementations:

Code snippet 7.6

```
class DataMocked {

    companion object Data {

        // Categories data mocked
        val category1 = Category(
            "100",
            "Clothes",
            "Clothes",
            "https://.../category_1.png"
        )
        ...
        // Products data mocked
        val product1 = Product(
            "11",
            "Neck T-Shirt",
```

```
            "Price and other details may vary based on product size...",
            "https://.../product_1.png",
            16.0,
            category1
        )
        ...
        // CartItem data mocked
        val cartItem1 = CartItem(
            1,
            product1
        )
        ...
    }
}


class CategoryMocked : CategoryDataSource {

    override fun getCategories(): Flow<List<Category>> {
        val categories: List<Category> =
            listOf(
                DataMocked.category1,
                DataMocked.category2,
                ...
            )
        return flowOf(categories)
    }
}

class ProductMocked : ProductDataSource {

    val products: List<Product> =
        listOf(
            DataMocked.product1,
            DataMocked.product2,
            ...
        )

    override fun getProducts(): Flow<List<Product>> {
        return flowOf(products)
    }

    override fun getProducts(category: Category): Flow<List<Product>> {
        return flowOf(products.filter { it.category.id == category.id })
    }
}
```

The following *mock* is defined in *common -> previews* directory. It will simulate the information in *Previews*.

Code snippet 7.7

```
class PreviewData {

    companion object PreviewList {

        val category = Category(
            "1",
            "Clothes",
            "Clothes",
            "category_1.png"
        )
        ...

        val product = Product(
            "1",
            "Neck T-Shirt",
            "Price and other details may vary based on product size...",
            "product_1.png",
            16.0,
            category
        )
        ...

        fun products(): List<Product> = listOf(product, ...)
        fun categories(): List<Category> = listOf(category, ...)
        fun cartProducts(): List<CartItem> = listOf(cart_product_1, ...)
    }
}
```

Graph of dependencies

Now that we have the main components in their respective layers in the added project, the next step is to define the dependency map, for which *Hilt* will be used like this:

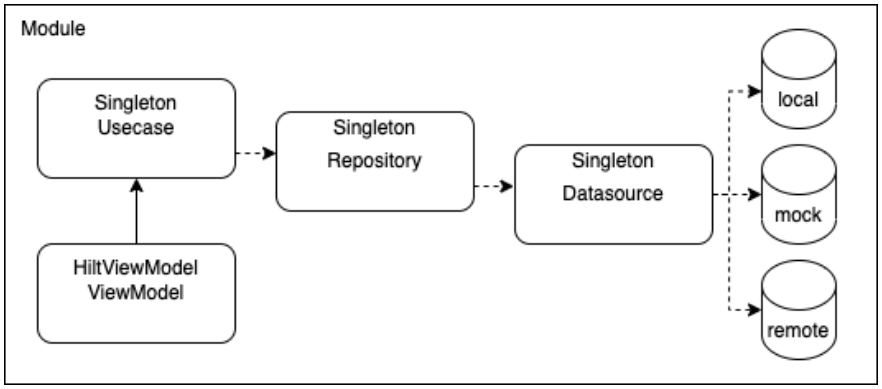

Figure 7.4 App’s Dependency Graph

Code snippet 7.8

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {

    @Provides
    @Singleton
    fun provideProductDataSource(): ProductDataSource {
        return ProductMocked()
    }

    @Provides
    @Singleton
    fun provideCategoryDataSource(): CategoryDataSource {
        return CategoryMocked()
    }

    @Provides
    @Singleton
    fun provideProductRepository(
        productDataSource: ProductDataSource
    ): ProductRepositoryPort {
        return ProductRepository(productDataSource)
    }

    @Provides
    @Singleton
    fun provideCategoryRepository(
        categoryDataSource: CategoryDataSource
    ): CategoryRepositoryPort {
        return CategoryRepository(categoryDataSource)
    }

```

```
    @Provides
    @Singleton
    fun provideProductUseCases(
        productRepository: ProductRepositoryPort,
        categoryRepository: CategoryRepositoryPort
    ): ProductUseCases {
        return ProductUseCases(
            getProducts = GetProductList(productRepository),
            getProduct = GetProductDetail(productRepository),
            getCategories = GetCategoryList(categoryRepository)
        )
    }
}
```

We must update this dependency model each time we add new components to the application that need to be made available through dependency injection.

Now, let's make a change to the *ViewModel* to verify that our implementation works correctly:

Code snippet 7.9

```
@HiltViewModel
class HomeViewModel @Inject constructor(
    private val productUseCases: ProductUseCases
    ) : ViewModel() {

    init {
        viewModelScope.launch {
            productUseCases.getProducts.getProducts().collect {
                println(it)
            }
        }
    }
}
```

In the console, if everything runs correctly, it should show in Logcat something similar to:

Code snippet 7.10

```
.ordernow I/System.out: [Product(id=11,
                                 name=Neck T-Shirt,
                                 description=Neck T-Shirt,
                                 urlImage=https://.../product_1.png,
                                 price=16.0), ...]
```

Source code

You can find the source code at OrderNow App[40].

[40] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_07/OrderNow

Home Screen

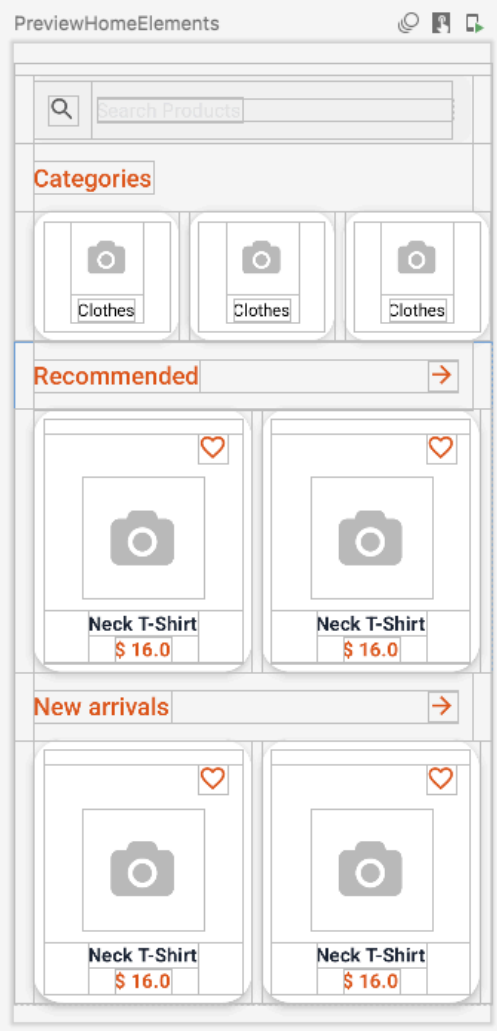

Figure 7.5 Home Screen Preview

Now that the services and component configurations in *Domain* and *Data* layers are ready, we will proceed to the layout of the *Home* view.

What information or data does the view require?

The *Sketch* of this screen indicates that it is made up of the following elements:

- Search Section.
- Carousel with category list.
- Carousel with recommended product list.
- Carousel with new products list.

We organize the Home screen into sections through *Slot APIs* like this:

Code snippet 7.11

```
@Composable
private fun HomeElements(
    categories: List<Category>,
    products: List<Product>,
    categorySelected: (category: Category) -> Unit,
    productSelected: (product: Product) -> Unit,
    modifier: Modifier = Modifier,
) {
    Column(
        modifier
            .verticalScroll(rememberScrollState())
            .padding(vertical = 16.dp)
    ) {
        SearchBar(Modifier.padding(horizontal = 16.dp))
        HomeSection(
            title = stringResource(AppText.categories),
            withArrow = false
        ) {
            CategorySection(
                categories = categories,
                categorySelected = categorySelected
            )
        }
        HomeSection(
            title = stringResource(AppText.recommended),
            withArrow = true
        ) {
            ProductSection(
                products = products,
                productSelected = productSelected
            )
        }
        HomeSection(
            title = stringResource(AppText.new_arrivals),
            withArrow = true
        ) {
            ProductSection(
                products = products,
                productSelected = productSelected
            )
        }
    }
}
```

Code snippet 7.12

```
@Composable
fun HomeSection(
    title: String,
    withArrow: Boolean,
    modifier: Modifier = Modifier,
    content: @Composable () -> Unit
) {
    Column(modifier) {
        Row(
            verticalAlignment = Alignment.CenterVertically,
            modifier = Modifier
                .heightIn(min = 56.dp)
                .padding(horizontal = 16.dp)
        ) {
            Text(
                text = title,
                style = MaterialTheme.typography.h6,
                color = orange,
                maxLines = 1,
                overflow = TextOverflow.Ellipsis,
                modifier = Modifier
                    .weight(1f)
                    .wrapContentWidth(Alignment.Start)
            )
            if (withArrow) {
                IconButton(
                    onClick = { },
                    modifier = Modifier.align(Alignment.CenterVertically)
                ) {
                    Icon(
                        imageVector = mirroringIcon(
                            ltrIcon = Icons.Outlined.ArrowForward,
                            rtlIcon = Icons.Outlined.ArrowBack
                        ),
                        tint = orange,
                        contentDescription = null
                    )
                }
            }
        }
        content()
    }
}
```

From the above code snippet 7.12, it should highlight the use of the property:

- *verticalScroll(rememberScrollState())*

Using this property in the column helps organize the sections more efficiently.

Another alternative implementation of the sections would have been using a *LazyColumn* component; however, for short content, it is better to use the *verticalScroll()* as *Google* tells us in *Basic layouts in Compose*[41].

Code snippet 7.13

```
@Composable
private fun HomeElements(
    categories: List<Category>,
    ...
) {
    Column(
        modifier
            .verticalScroll(rememberScrollState())
            .padding(vertical = 16.dp)
    ) {
        ...
    }
}
```

The use of *Slot APIs* helps the reuse of views and their dynamic composition:

Code snippet 7.14

```
@Composable
fun HomeSection(
    ...
    content: @Composable () -> Unit
) {
    Column(modifier) {
        Row(
            verticalAlignment = Alignment.CenterVertically,
            ...
        ) {
            ...
        }
        content()
    }
}
```

Through the *Slot APIs*, this technique is also clearly explained in *Basic layouts in Compose*[42].

[41] Basic layouts in Compose: Video by Simona and Jolanda in Google IO 2022, min 41
[42] Basic layouts in Compose: Video by Simona and Jolanda in Google IO 2022, min 52

Where does the data used by the view come from?

Data is provided by Usecase, which obtains it from the respective *Repository*.

A *repository* is the only reliable source that delivers the data; that is, it is the *source of truth*.

Usecase delivers the information to the *ViewModel* through a data flow defined with *Flow Coroutines* and transformed into states through *StateFlow*.

The view receives from the *ViewModel* the flow’s state of said information and reacts according to its values.

The definitions of *HomeViewModel* and *HomeScreen* are as follows:

Code snippet 7.15: HomeViewModel and HomeScreen

```
@HiltViewModel
class HomeViewModel @Inject constructor(
    private val productUseCases: ProductUseCases
    ) : ViewModel() {

    val productListState = productUseCases.getProducts.getProducts()
        .stateIn(viewModelScope, SharingStarted.Eagerly, emptyList())

    val categoryListState = productUseCases.getCategories.getCategories()
        .stateIn(viewModelScope, SharingStarted.Eagerly, emptyList())
}

@Composable
fun HomeScreen(
    categorySelected: (category: Category) -> Unit,
    productSelected: (product: Product) -> Unit,
    modifier: Modifier = Modifier,
    viewModel: HomeViewModel = hiltViewModel()
) {

    val products by viewModel.productListState.collectAsState()
    val categories by viewModel.categoryListState.collectAsState()

    HomeElements(
        categories,
        products,
        categorySelected,
        productSelected,
        modifier = modifier.background(lightGrayBackground)
    )
}
```

Adding "Home" to the navigation

As discussed in Chapter 6: Designing navigation in App, the configuration of application navigation can be supported with the *OrderNowState*. We will use this component to transport the data between the different views like so:

Code snippet 7.16: OrderNowState

```
class OrderNowState(
    val scaffoldState: ScaffoldState,
    val navController: NavHostController,
    private val resources: Resources,
    coroutineScope: CoroutineScope
){
    ...
    lateinit var categorySelected: Category
    lateinit var productSelected: Product
    ...
}
```

Additionally, the navigation logic associated with this screen is added to the *OrderNowNavHost* component like so:

Code snippet 7.17: OrderNowNavHost

```
fun NavGraphBuilder.appSoGraph(appState: OrderNowState) {

    val homeRoute = OrderNowScreenRoute.Home.route
    val listRoute = OrderNowScreenRoute.ProductList.route
    val detailRoute = OrderNowScreenRoute.ProductDetail.route

    val productSelectedInHome: (Product) -> Unit = { product: Product ->
        appState.productSelected = product
        appState.navigateSaved(detailRoute, homeRoute)
    }
    val categorySelectedInHome: (Category) -> Unit = { category: Category ->
        appState.categorySelected = category
        appState.navigateSaved(listRoute, homeRoute)
    }

    // Home Screen Graph
    composable(NavigationBarSection.Home.route) {
        HomeScreen(
            categorySelected = categorySelectedInHome,
            productSelected = productSelectedInHome
        )
    }
}
```

Other settings

Finally, for the implementation of the *Home*, the following configurations are required in the application: Installing Coil[43] for local or remote image loading and management required by the application:

Code snippet 7.18

```
implementation "io.coil-kt:coil-compose:2.0.0"
```

And add the internet permissions in the manifest to allow calling remote resources or services:

Code snippet 7.19

```
<uses-permission android:name="android.permission.INTERNET" />
```

An example of using *Coil* to fetch and render images might look like this:

Code snippet 7.20

```
@Composable
fun CategoryCard(
    category: Category,
    categorySelected: (category: Category) -> Unit
) {
    Card(
        modifier = Modifier
        ...
    ) {
        Column(
            ...
            verticalArrangement = Arrangement.Center
        ) {
            AsyncImage(
                model = ImageRequest.Builder(LocalContext.current)
                    .data(category.urlImage)
                    .crossfade(true)
                    .build(),
                placeholder = painterResource(R.drawable.placeholder),
                contentDescription = "",
                modifier = Modifier.size(60.dp),
            )
        }
    }
}
```

[43] Coil and Jetpack Compose

Source code

You can find the source code at OrderNow App[44].

Product List Screen

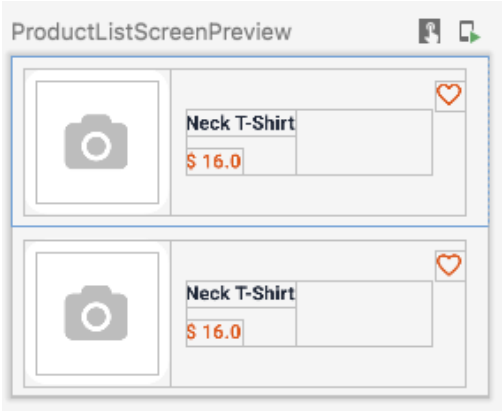

Figure 7.6 ProductList Screen Preview

For the implementation of this screen, *ProductItem* and *ProductItemList* components are created:

Code snippet 7.21: ProductItemList

```
@Composable
fun ProductItemList(
    products: List<Product>,
    productSelected: (Product) -> Unit,
    modifier: Modifier = Modifier,
) {
    LazyColumn(modifier = modifier) {
        items(products, key = { it.id }) { product ->
            ProductItem(product, productSelected)
        }
    }
}
```

[44]https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_07/OrderNow

Code snippet 7.22: ProductItem

```
@Composable
fun ProductItem(
    product: Product,
    productSelected: (Product) -> Unit
) {
    Row(
        ...
    ) {
        Box(
            ...
        ) {
            AsyncImage(
                model = ...
            )
        }
        Column(
            modifier = Modifier
                .fillMaxWidth(0.8f)
        ) {

            Text(
                text = product.name,
                fontWeight = FontWeight.Bold,
                fontSize = 16.sp,
                color = titleTextColor
            )
            Spacer(modifier = Modifier.height(10.dp))
            Text(
                text = buildAnnotatedString {
                    withStyle(
                        style = SpanStyle(
                            orange,
                            fontWeight = FontWeight.Bold
                        )
                    ) {
                        append("$ " + product.price)
                    }
                },
                style = MaterialTheme.typography.subtitle1,
                modifier = Modifier,
                fontSize = 16.sp
            )
        }
        Box(
            modifier = Modifier
```

```
                .padding(top = 20.dp, end = 20.dp)
                .align(Alignment.Top)
        ) {
            Icon(
                imageVector = Icons.Outlined.FavoriteBorder,
                contentDescription = "",
                tint = orange
            )
        }
    }
}
```

Then, *ProductListViewModel* is updated:

Code snippet 7.23

```
@HiltViewModel
class ProductListViewModel @Inject constructor(
    private val productUseCases: ProductUseCases
) : ViewModel() {

    val productListState = { category: Category ->
        productUseCases.getProducts.getProducts(category)
            .stateIn(viewModelScope, SharingStarted.Eagerly, emptyList())
    }
}
```

In the above code snippet 7.23, we should highlight the use of *StateFlow* to represent the product list’s state. That is an example of state representation studied in the section of the first chapter, Property UI’s state.

In this view, the definitions are as follows:

Code snippet 7.24

```
@Composable
fun ProductListScreen(
    category: Category,
    productSelected: (Product) -> Unit,
    modifier: Modifier = Modifier,
    viewModel: ProductListViewModel = hiltViewModel()
) {
    val products by viewModel.productListState(category).collectAsState()

    ProductListContent(
        products,
        productSelected,
        modifier = modifier.background(lightGrayBackground)
```

```
    )
}

@Composable
fun ProductListContent(
    products: List<Product>,
    productSelected: (Product) -> Unit,
    modifier: Modifier = Modifier
) {
    ProductItemList(
        products = products,
        productSelected = productSelected,
        modifier = modifier
    )
}
```

Adding “ProductList” to the navigation

Finally, the screen is added to the navigation graph like so:

Code snippet 7.25: OrderNowNavHost

```
fun NavGraphBuilder.appSoGraph(appState: OrderNowState) {

    ...
    val listRoute = OrderNowScreenRoute.ProductList.route
    val detailRoute = OrderNowScreenRoute.ProductDetail.route
    ...

    val productSelectedInList: (Product) -> Unit = { product: Product ->
        appState.productSelected = product
        appState.navigateSaved(detailRoute, listRoute)
    }

    ...

    // Product List Screen Graph
    composable(OrderNowScreenRoute.ProductList.route) {
        ProductListScreen(
            category = appState.categorySelected,
            productSelected = productSelectedInList
        )
    }
    ...
}
```

Source code

You can find the source code at OrderNow App[45].

Product Detail Screen

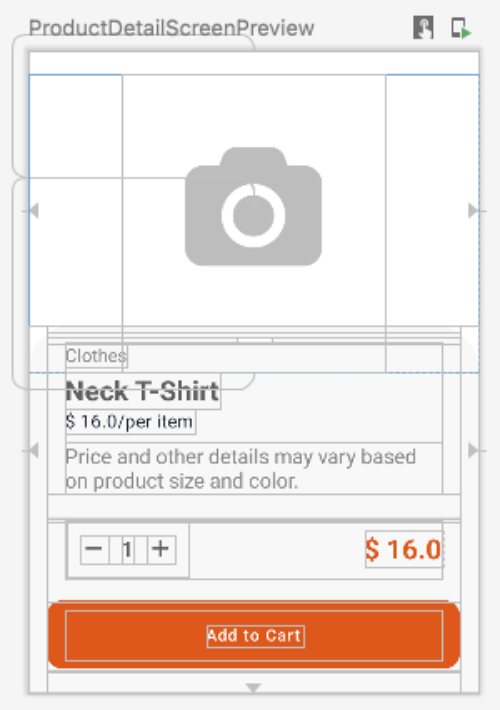

Figure 7.7 ProductDetail Screen Preview

For the product detail screen implementation, it is first required to add the following dependency to use the *Constraintlayout* components in *Jetpack Compose*:

Code snippet 7.26

```
implementation "androidx.constraintlayout:constraintlayout-compose:1.0.1"
```

ProductDetailScreen view is added:

[45]https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_07/OrderNow

Code snippet 7.27

```
@Composable
fun ProductDetailScreen(
    product: Product,
    goToCart: () -> Unit,
    modifier: Modifier = Modifier,
    viewModel: ProductDetailViewModel = hiltViewModel(),
) {
    BodyDetail(
        product = product,
        addToCart = { count -> viewModel.saveItemCart(product, count) },
        showAlert = viewModel.showAlert,
        onGoToCart = goToCart,
        onPopupDismissed = viewModel.onPopupDismissed(),
        modifier = modifier
            .fillMaxSize()
            .background(lightGrayBackground)
    )
}
```

And its corresponding *ProductDetailViewModel*:

Code snippet 7.28

```
@HiltViewModel
class ProductDetailViewModel @Inject constructor(
    private val cartUseCases: CartUseCases
) : ViewModel() {

    var showAlert by mutableStateOf(false)
        private set

    fun saveItemCart(product: Product, quantity: Int) {
        viewModelScope.launch {
            val cartItem = CartItem(
                quantity = quantity,
                product = product
            )
            cartUseCases.saveItemCart
                .saveItem(cartItem).collect {
                    showAlert = true
                }
        }
    }

    fun onPopupDismissed(): () -> Unit = {
```

```
23          showAlert = false
24      }
25  }
```

Adding “Back” to the TopAppBar

To allow the *Go Back* option to be visible in the *TopAppBar*, the state of the application is used again to configure those screens that require going back in the navigation, like this:

In *OrderNowScreenRoute*:

Code snippet 7.29

```
sealed class OrderNowScreenRoute (val route: String) {

    companion object {
        val withArrowBack = listOf(
            ProductDetail,
            Checkout,
            PlaceOrder
        )
    }

    object Home : OrderNowScreenRoute("home")
    object Cart : OrderNowScreenRoute("cart")
    object ProductList : OrderNowScreenRoute("product_list")
    object ProductDetail : OrderNowScreenRoute("product_detail")
    ...
}
```

The *withArrowBack* property will define which screens require the Go Back option to be presented.

In *OrderNowState*

Code snippet 7.30

```
class OrderNowState(
    val scaffoldState: ScaffoldState,
    ...
){
    private val screensWithArrowBack = OrderNowScreenRoute.withArrowBack.map { it.route }

    val shouldShowArrowBack: Boolean
        @Composable get() = navController
            .currentBackStackEntryAsState()
            .value?.destination?.route in screensWithArrowBack

```

```
    ...
}
```

The *shouldShowArrowBack* property will tell the view whether or not to display the option in *TopAppBar* like so:

Code snippet 7.31

```
@Composable
fun OrderNowTopBar(appState: OrderNowState) {
    if (appState.shouldShowArrowBack) {
        TopAppBarWithArrow(
            title = stringResource(id = R.string.app_name),
            goBack = appState.popUp()
        )
    } else {
        TopAppBarWithoutArrow(
            title = stringResource(id = R.string.app_name)
        )
    }
}
```

Source code

You can find the source code at OrderNow App[46].

[46] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_07/OrderNow

Cart Screen

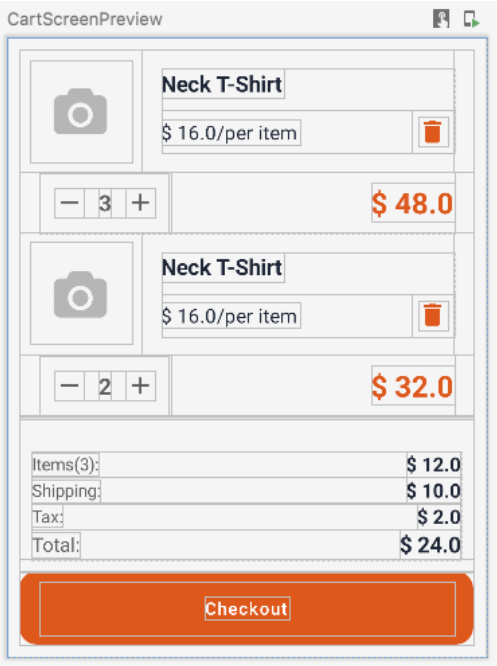

Figure 7.8 Cart Screen Preview

The *Sketch* of this screen indicates that it is made up of the following elements:

- The list of items added to the cart will be represented as a Component UI’s state called **cartState**.
- The purchase summary will also be represented as a Component UI’s state called **orderSummaryState**.
- The checkout process button will be a component whose state will depend on the two previous states, **cartState** and **orderSummaryState**.

In this design, I have chosen to define **cartState** and **orderSummaryState** as independent of each other; however, it could also be valid for **orderSummaryState** to depend on **cartState**.

The implementation of *CartViewModel* would be as follows:

Code snippet 7.32: CartViewModel

```
@HiltViewModel
class CartViewModel @Inject constructor(
    private val cartUseCases: CartUseCases,
) : ViewModel() {

    var cartState by mutableStateOf(CartState())
        private set

    var orderSummaryState by mutableStateOf(OrderSummaryState())
        private set
```

```
11
12      init {
13          viewModelScope.launch {
14              cartUseCases.getCartItems
15                  .getCartItems().collect {
16                      updateStates(cartItems = it)
17                  }
18          }
19      }
20
21      ...
22
23      private fun updateStates(cartItems: List<CartItem>) {
24          cartState = cartState.copy(cartItems = cartItems)
25          orderSummaryState = orderSummaryState.copy(cartItems = cartItems)
26      }
27      ...
28  }
```

And the definition of the Component UI’s state *CartState* and *OrderSummaryState* like so:

Code snippet 7.33: CartState

```
1  data class CartState(
2      val cartItems: List<CartItem> = emptyList()
3  )
4
5  val CartState.readyForCheckout: Boolean
6      get() = cartItems.isNotEmpty()
```

Code snippet 7.34: OrderSummaryState

```
1  data class OrderSummaryState(
2      val cartItems: List<CartItem> = emptyList(),
3      val shipping: Double = 0.00,
4      val tax: Double = 0.00
5  )
6
7  val OrderSummaryState.itemsTotal: Double
8      get() =
9          cartItems
10             .map { item -> item.product.price * item.quantity }
11             .fold(0.0) { total, next -> total + next }
12
13 val OrderSummaryState.allTotal: Double
14     get() = itemsTotal + shipping + tax
```

From the view side, the implementation could be like this:

Code snippet 7.35: CartScreen and CartUI

```
@Composable
fun CartScreen(
    goToCheckout: (SummaryTotals) -> Unit,
    modifier: Modifier = Modifier,
    viewModel: CartViewModel = hiltViewModel(),
) {
    CartUI(
        cartState = viewModel.cartState,
        orderSummaryState = viewModel.orderSummaryState,
        ...
        checkoutSelected = { goToCheckout(viewModel.getSummaryOrder()) },
        ...
    )
}

@Composable
fun CartUI(
    cartState: CartState,
    orderSummaryState: OrderSummaryState,
    ...
    checkoutSelected: () -> Unit,
    ...
) {
    if (cartState.cartItems.isEmpty()) {
        Empty("No cart items found")
    } else {
        CartElements(
            cartItems = cartState.cartItems,
            numberItems = cartState.cartItems.size,
            totalItems = orderSummaryState.itemsTotal,
            ...
        )
    }
}
```

In the previous code snippet 7.35, we have used the recommended naming in the Naming composable functions section for better clarity of each element.

Another point to highlight in the implementation is the reuse given to the *Composable* called **Quantity**:

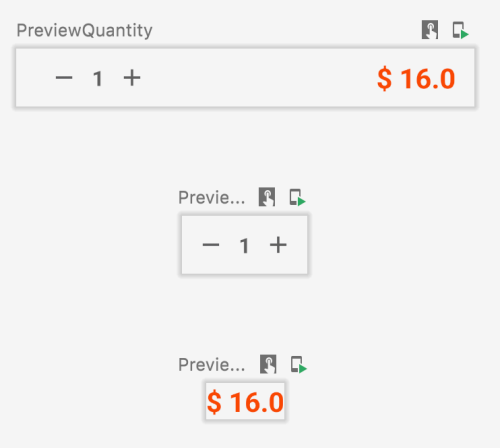

Figure 7.9 Quantity Composable

Code snippet 7.36: Quantity

```
@Composable
fun Quantity(
    count: Int,
    decreaseItemCount: () -> Unit,
    increaseItemCount: () -> Unit,
    price: Double
) {

    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(horizontal = 16.dp),
        horizontalArrangement = Arrangement.SpaceBetween,
        verticalAlignment = Alignment.CenterVertically
    ) {

        QuantitySelector(
            count = count,
            decreaseItemCount = decreaseItemCount,
            increaseItemCount = increaseItemCount
        )

        PriceView(price)
    }
}
```

We are using this *Quantity* view both in the detail screen and in the shopping cart screen, and it shows us the capacity that *Jetpack Compose* provides us to atomize components in the UI.

That is also possible because we have defined the *Quantity* view as a *stateless view* using the *state hoisting* technique, both described in Chapter 1: Design principles.

Another point to highlight in the implementation is the following:

Code snippet 7.37

```
@Composable
fun CartElements(
    cartItems: List<CartItem>,
    ...
) {
    LazyColumn(
        modifier = modifier,
        contentPadding = contentPadding
    ) {
        items(cartItems, key = { it.product.id }) { cartItem ->
            CartViewItem(...)
        }
        item {
            CartSummary(...)
            CartBottom(...)
        }
    }
}
```

Unlike the example implementation of the Home Screen, we are not using the property:

- *verticalScroll(...)*

Do you know the reason?

In this case, it is not convenient to use this property since its use would generate nesting of *scrolls* in the same direction. That is caused because the *cartItems* list is represented by a *LazyColumn*, which implicitly includes a vertical *scroll*.

I recommend this excellent documentation from *Google*, where they give us more detail, and other good practices *Avoid nesting components scrollable in the same direction*[47].

Source code

You can find the source code at OrderNow App[48].

[47] Avoid nesting components scrollable in the same direction
[48] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_07/OrderNow

Checkout Screen

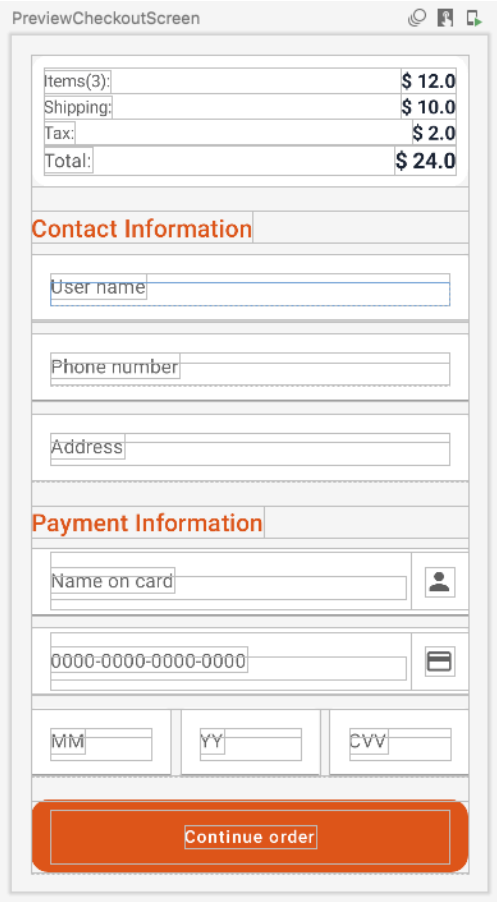

Figure 7.10 Checkout Screen Preview

To implement this screen, we will use the concepts learned in Chapter 2: Codelab - Practicing with states.

In that chapter, we saw how to organize a screen’s elements by grouping states and events to have a better clean code.

The *Sketch* of this screen indicates that it is made up of the following elements:

- Section with the summary of the purchase values, which the *CartScreen* previous view will deliver through navigation.
- The contact information component will be represented as a Component UI’s state called **ContactFormState**.
- The payment information component will be represented as a Component UI’s state called **PaymentFormState**.
- Button to continue with the purchase process, a component that will depend on the **ContactFormState** and **PaymentFormState** states.

The implementation of *CheckoutViewModel* would be as follows:

Code snippet 7.38: CheckoutViewModel

```
@HiltViewModel
class CheckoutViewModel @Inject constructor() : ViewModel() {

    var contactUiState by mutableStateOf(ContactFormState())
        private set

    var paymentUiState by mutableStateOf(PaymentFormState())
        private set

    val onContactFormEvent: (ContactFormEvent) -> Unit = { event ->
        contactUiState = when (event) {
            is ContactFormEvent.OnNameChange -> {
                contactUiState.copy(username = event.name)
            }
            is ContactFormEvent.OnPhoneChange -> {
                contactUiState.copy(phone = event.phone)
            }
            is ContactFormEvent.OnAddressChange -> {
                contactUiState.copy(address = event.address)
            }
        }
    }

    val onPaymentFormEvent: (PaymentFormEvent) -> Unit = { event ->
        paymentUiState = when (event) {
            is PaymentFormEvent.OnNameChange -> {
                paymentUiState.copy(name = event.name)
            }
            is PaymentFormEvent.OnNumberChange -> {
                paymentUiState.copy(number = event.number)
            }
            is PaymentFormEvent.OnMonthChange -> {
                paymentUiState.copy(month = event.month)
            }
            is PaymentFormEvent.OnYearChange -> {
                paymentUiState.copy(year = event.year)
            }
            is PaymentFormEvent.OnCodeChange -> {
                paymentUiState.copy(code = event.code)
            }
        }
    }
    ...
}
```

And the definition of the Component UI's state *ContactFormState* and *PaymentFormState* is like this:

Code snippet 7.39: ContactFormState

```
data class ContactFormState(
    val username: String = "",
    val phone: String = "",
    val address: String = "",
)

val ContactFormState.successValidated: Boolean
    get() = username.length > 1
            && phone.length > 1
            && address.length > 1
```

Code snippet 7.40: PaymentFormState

```
data class PaymentFormState(
    val name: String = "",
    val number: String = "",
    val month: String = "",
    val year: String = "",
    val code: String = ""
)

val PaymentFormState.successValidated: Boolean
    get() = name.length > 1
            && number.length > 1
            && month.length > 1
            && year.length > 1
            && code.length > 2
```

To reduce the number of parameters in the *composable functions* in the view, we will define an auxiliary structure that we will call **StateVsEvent** like this:

Code snippet 7.41

```
data class StateVsEvent(
    val value: String = "",
    val onValueChange: (String) -> Unit = {}
)
```

This structure will group as a pair, both the value (*state*) and the action (*event*), for instance, like this:

Code snippet 7.42

```
PaymentInformation(
    nameStateVsEvent = StateVsEvent(
        value = payment.name,
        onValueChange = {
            onPaymentEvent(PaymentFormEvent.OnNameChange(it))
        }),
        ...
)
```

Why use Data class instead of Pair?

It is a documented recommendation in *“Prefer data classes instead of tuples”*[49].

From the view side, and using the recommended nomenclature in the Naming composable functions section, the implementation could look like this:

Code snippet 7.43

```
@Composable
fun CheckoutScreen(
    summary: SummaryTotals,
    goToPlaceOrder: (Order) -> Unit,
    modifier: Modifier = Modifier,
    viewModel: CheckoutViewModel = hiltViewModel()
) {
    CheckoutUI(
        summary = summary,
        ...
    )
}
```

Code snippet 7.44

```
@Composable
fun CheckoutUI(
    summary: SummaryTotals,
    contactUiState: ContactFormState,
    onContactEvent: (ContactFormEvent) -> Unit,
    paymentUiState: PaymentFormState,
    onPaymentEvent: (PaymentFormEvent) -> Unit,
    onContinueOrder: () -> Unit,
    modifier: Modifier = Modifier,
) {
```

[49] Prefer data classes instead of tuples

```
    CheckoutElements(
        summary = summary,
        ...
    )
}
```

Code snippet 7.45: CheckoutElements

```
@Composable
fun CheckoutElements(
    summary: SummaryTotals,
    contact: ContactFormState,
    onContactEvent: (ContactFormEvent) -> Unit,
    payment: PaymentFormState,
    onPaymentEvent: (PaymentFormEvent) -> Unit,
    onContinueOrder: () -> Unit,
    modifier: Modifier = Modifier,
) {
    Column(
        ...
    ) {
        SummaryCard(
            numberItems = summary.numberItems,
            totalItems = summary.totalItems,
            ...
        )
        CheckoutSection(
            title = "Contact Information"
        ) {
            ContactInformation(
                nameStateVsEvent = StateVsEvent(
                    value = contact.username,
                    onValueChange = {
                        onContactEvent(ContactFormEvent.OnNameChange(it))
                    }),
                ...
            )
        }
        CheckoutSection(
            title = "Payment Information"
        ) {
            PaymentInformation(
                nameStateVsEvent = StateVsEvent(
                    value = payment.name,
                    onValueChange = {
                        onPaymentEvent(PaymentFormEvent.OnNameChange(it))
```

```
                    }),
                ...
            )
        }
        StandardButton(
            text = stringResource(R.string.continue_order),
            onClicked = onContinueOrder,
            enabled = contact.successValidated
                        && payment.successValidated
        )
    }
}
```

Before I finish, I need to highlight the following piece of code:

Code snippet 7.46: doPayment

```
@Composable
fun CheckoutScreen(
    ...
    goToPlaceOrder: (Order) -> Unit,
    ...
) {
    CheckoutUI(
        ...
        onContinueOrder = { goToPlaceOrder(viewModel.getOrder(summary)) },
        ...
    )
}
```

That is a pattern to send an event to *ViewModel* from *View* before handing control over to the state for further navigation.

Source code

You can find the source code at OrderNow App[50].

[50] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_07/OrderNow

Place Order Screen

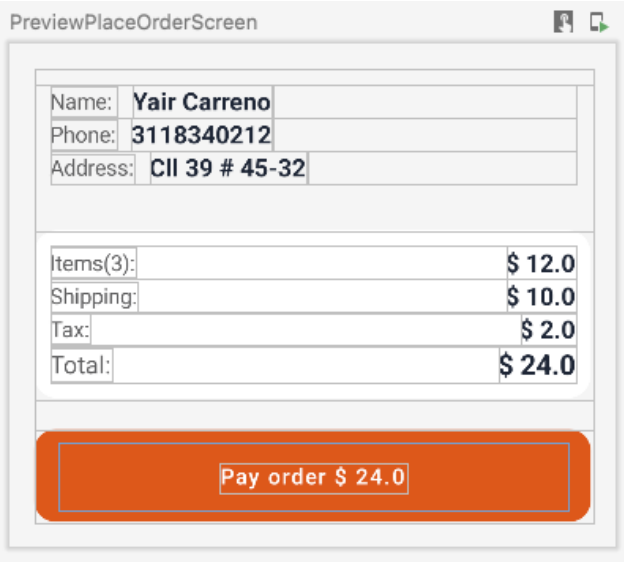

Figure 7.11 PlaceOrder Screen Preview

Finally, the last screen to be described is the one corresponding to the order payment process.

A mock operation has been created to simulate the payment process using *Coroutines* and a delay of 4 seconds like this:

Code snippet 7.47: doPayment

```
class PaymentMocked : PaymentDataSource {

    override suspend fun doPayment(order: Order): PaymentResult {
        return withContext(Dispatchers.IO) {
            delay(4000)
            PaymentResult(
                status = true,
                payment = Payment(
                    status = "SUCCESS",
                    reference = "123ABC")
            )
        }
    }
}
```

The most important part to highlight in the implementation of this screen is how the payment flow is defined with the following characteristics:

- The payment operation is modeled as a state through *mutableStateOf*. In this case, it has been named *PaymentState*.

- The view only relies on the *PaymentState* state for UI decision-making. That is an excellent recommendation by Manuel Vivo from *Google* in the article *ViewModel: One-off event antipatterns*[51].
- Depending on the result of the payment operation (status), the view (Composable) will make a dynamic recomposition of the elements presented on the screen. It's an excellent example to demonstrate the flexibility and power of Jetpack Compose again on Android.

The definition of the state would be as follows:

Code snippet 7.48: PaymentState

```
data class PaymentState(
    val paymentInformation: Payment = Payment(),
    val isLoading: Boolean = false,
    val paymentResult: PaymentResult? = null,
    val errorMessage: String? = null
)
```

In *PlaceOrderViewModel*, the operations that will depend on the state are defined as follows:

Code snippet 7.49: PlaceOrderViewModel

```
@HiltViewModel
class PlaceOrderViewModel @Inject constructor(
    private val paymentUseCases: PaymentUseCases,
) : ViewModel() {

    var paymentUiState by mutableStateOf(PaymentState())
        private set

    fun makePayment() {
        viewModelScope.launch {
            paymentUiState = paymentUiState.copy(isLoading = true)
            try {
                val paymentResult = paymentUseCases.doPayment
                    .doPayment(order = PreviewData.order)
                paymentUiState =
                    paymentUiState.copy(
                        isLoading = false,
                        paymentResult = paymentResult
                    )
            } catch (ioe: IOException) {
                paymentUiState = paymentUiState.copy(
                    isLoading = false,
                    errorMessage = "Error with the transaction"
                )
            }
```

[51] ViewModel: One-off event antipatterns by Manuel Vivo

```
        }
    }
}
```

And in this view side, the implementation is as follows:

Code snippet 7.50

```
@Composable
fun PlaceOrderScreen(
    order: Order,
    modifier: Modifier = Modifier,
    viewModel: PlaceOrderViewModel = hiltViewModel(),
) {
    OrderUI(
        order = order,
        paymentUiState = viewModel.paymentUiState,
        onPlaceOrder = { viewModel.makePayment() },
        modifier = modifier
            .fillMaxSize()
            .background(lightGrayBackground)
    )
}

@Composable
fun OrderUI(
    order: Order,
    paymentUiState: PaymentState,
    onPlaceOrder: () -> Unit,
    modifier: Modifier,
) {
    if (paymentUiState.isLoading) {
        Loading()
    } else {
        Column {
            if (paymentUiState.paymentResult != null) {
                val successful = paymentUiState.paymentResult.status
                if (successful) {
                    Results(userMessage = "Thanks for your purchase.") {
                        OrderSummary(
                            order = order
                        )
                    }
                }
            } else {
                OrderElements(
                    order = order,
```

```
                    onPlaceOrder = onPlaceOrder,
                    modifier = modifier
                )
            }
        }
    }
}
```

From code snippet 7.50 above, the part to highlight is the following:

Code snippet 7.51

```
@Composable
fun OrderUI(
    ...
    paymentUiState: PaymentState,
    ...
) {
    if (paymentUiState.isLoading) {
        ...
    } else {
        Column {
            if (paymentUiState.paymentResult != null) {
                val successful = ...
                if (successful) {
                    Results(userMessage = "Thanks for your purchase.") {
                        OrderSummary(...)
                    }
                }
            } else {
                OrderElements(...)
            }
        }
    }
}
```

The order summary view is atomized and can be presented differently depending on the state.

Source code

You can find the source code at OrderNow App[52].

[52] https://github.com/yaircarreno/building-modern-apps-for-android-code/tree/main/chapter_07/OrderNow

Summary

Wow! I think this has been the book’s longest chapter, but it is not for less. Here we have put together all the critical tabs of our *OrderNow* app.

Based on the basic concepts of the first chapters, we have finally experienced the relevant capabilities and techniques when implementing the solution.

We refined the navigation process using the state of the App as a reference. We also involved sending information through navigation in a flexible way.

We organize our code using *Clean Architecture* recommendations and applying *Clean Code* pragmatically, using a nomenclature that can be clear to any developer reviewing the sources.

We review how from the *Sketch* of a screen, we visualize the main components, their states, their events, and the relationships between them.

We reuse views across multiple screens to show the flexibility of *Compose*.

Through *Coroutines* and *Flow*, we also managed to generate the information flows required by the views and worked through the states with the support of *ViewModels*.

And this is the final chapter of the book. I hope this work has been helpful for the reader and serves as a reference for his projects.

Total thanks.

Changelog

Revision 1 (06-27-2022)

Initial version of the book.