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

Chapter 1: Design principles **1**
“State” is the heart of declarative views 1
Applying “State hoisting” to delegate states 2
Defining the “Source of Truth”. Who is responsible for providing the states? 4
ViewModel as Source of truth 6
Understanding data flow, “Unidirectional Data Flow Pattern” 7
Let’s connect View and ViewModel components 8
Structures represented as states 10
Modeling and grouping events 13
Summary 15

Chapter 3: OrderNow, A Real Application **16**
About the application 16
Screens 16
Technologies 19
Summary 20

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
)
```

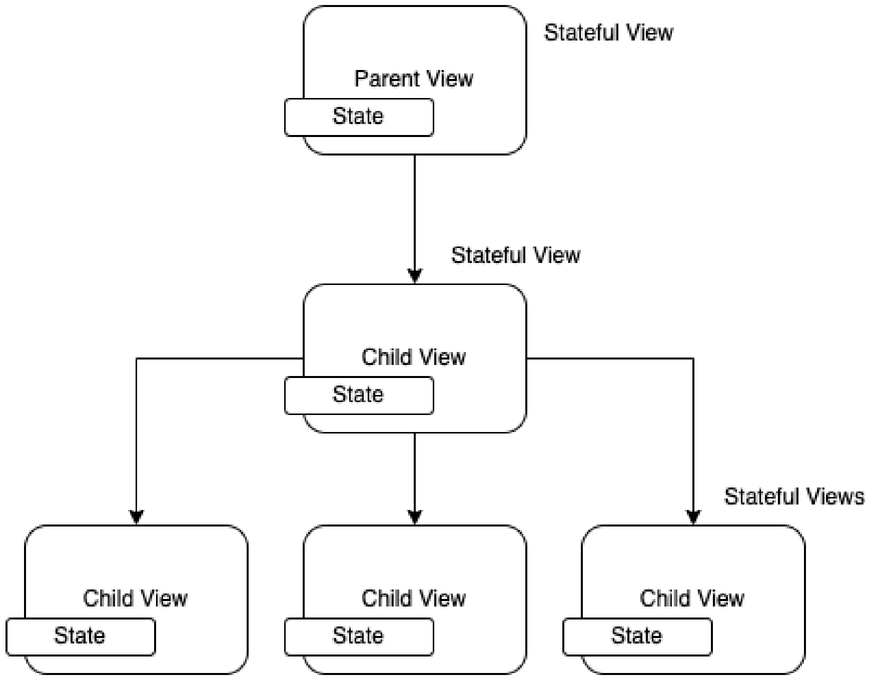


Figure 1.1 Declarative Views’s Hierarchy

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

```kotlin
// 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
) {

    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")
            },
        ) {
            Text("Pay order")
        }
    }
}
```

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

```kotlin
@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*[1], 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[2]) to reflect the changes in the UI.

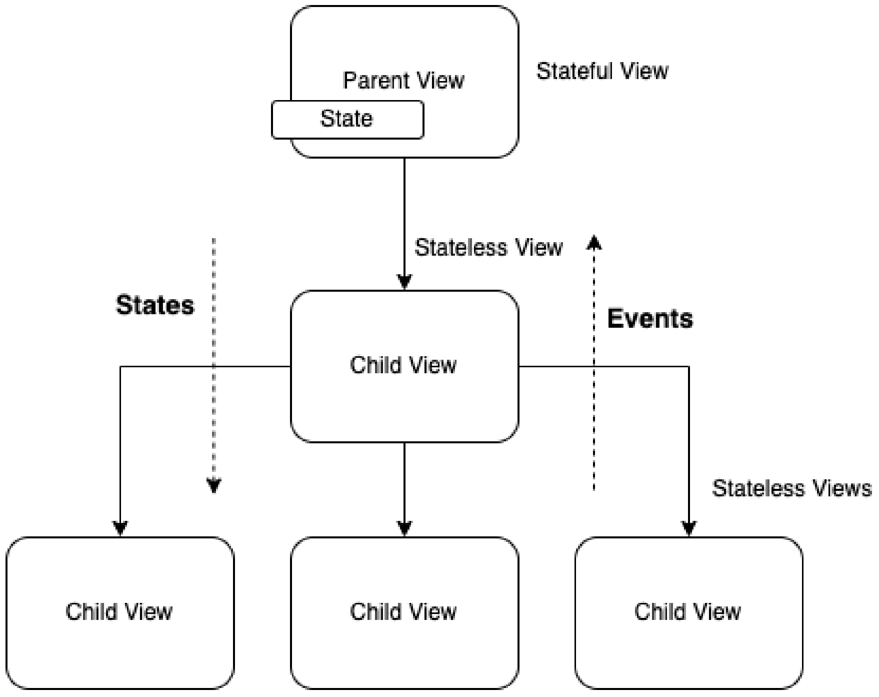


Figure 1.2 Delegating state handling to a View

[1]Manage state with Unidirectional Data Flow
[2]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[3]. 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.

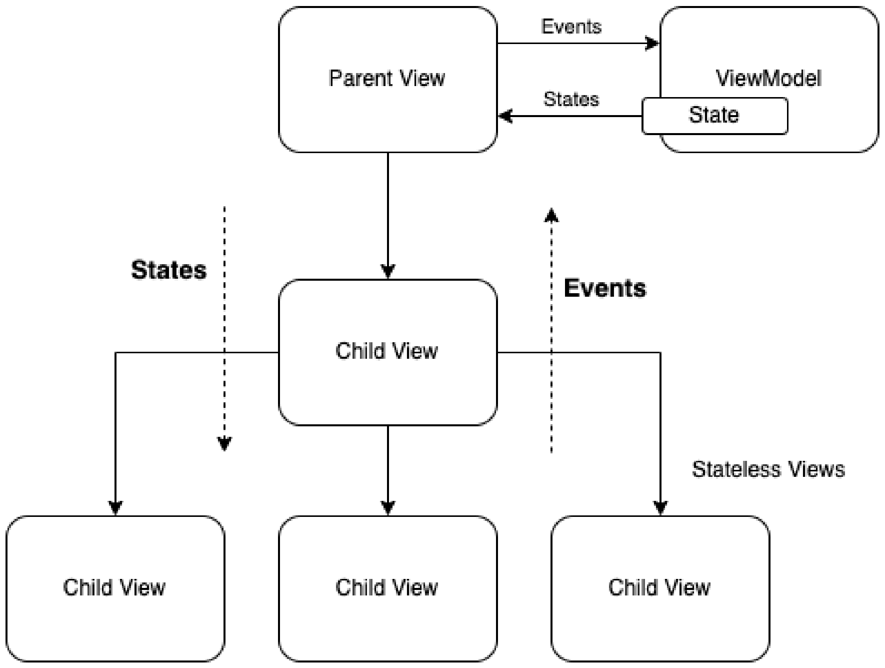


Figure 1.3 Delegating state handling to a ViewModel

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*[4].

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

[3]Managing state in Compose
[4]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:

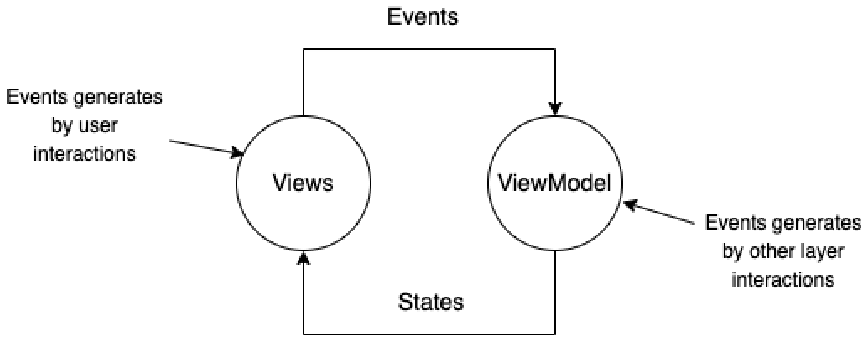

Events
Events generates by user interactions
Views
ViewModel
Events generates by other layer interactions
States

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

```kotlin
//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

```kotlin
// 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

```kotlin
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)

```kotlin
@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.

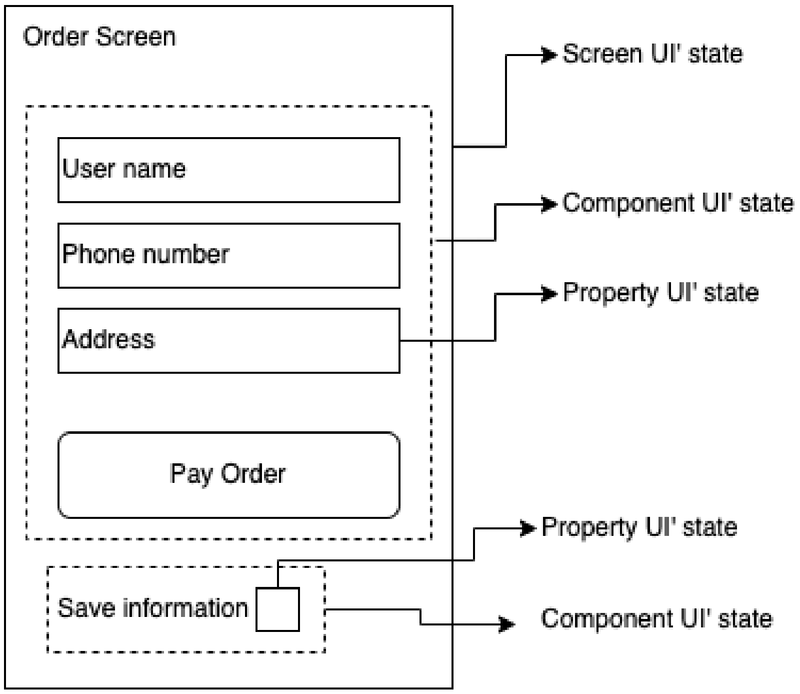


Figure 1.5 Structures represented by states

- **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

```kotlin
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

```kotlin
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:

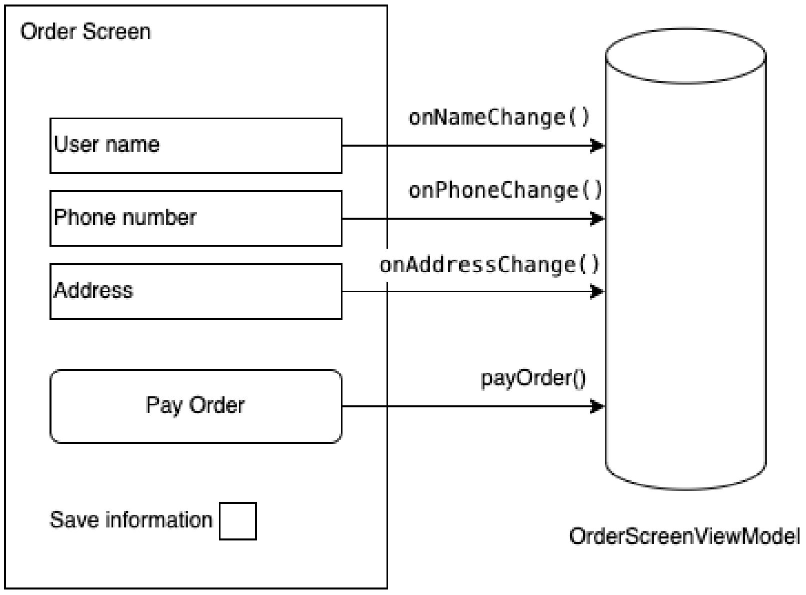


Figure 1.6 Multiple events

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:

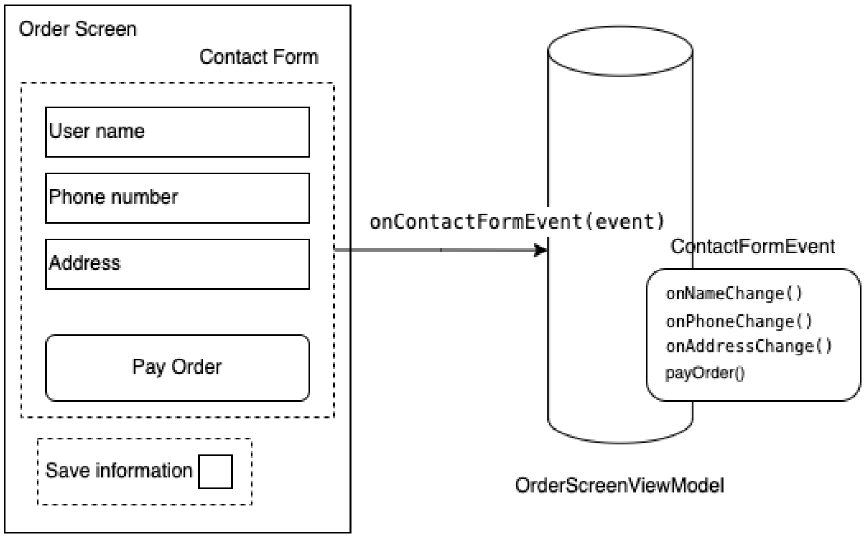


Figure 1.7 Grouping events

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 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

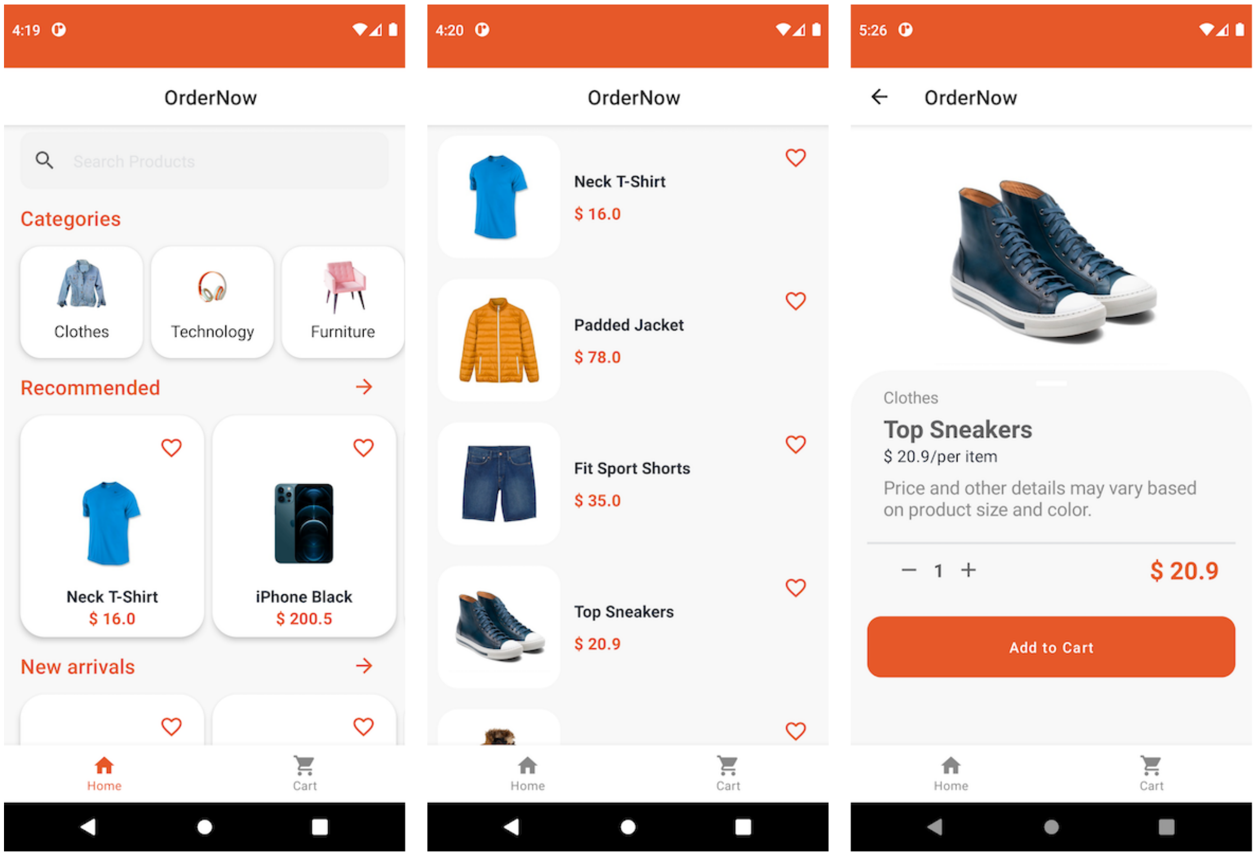
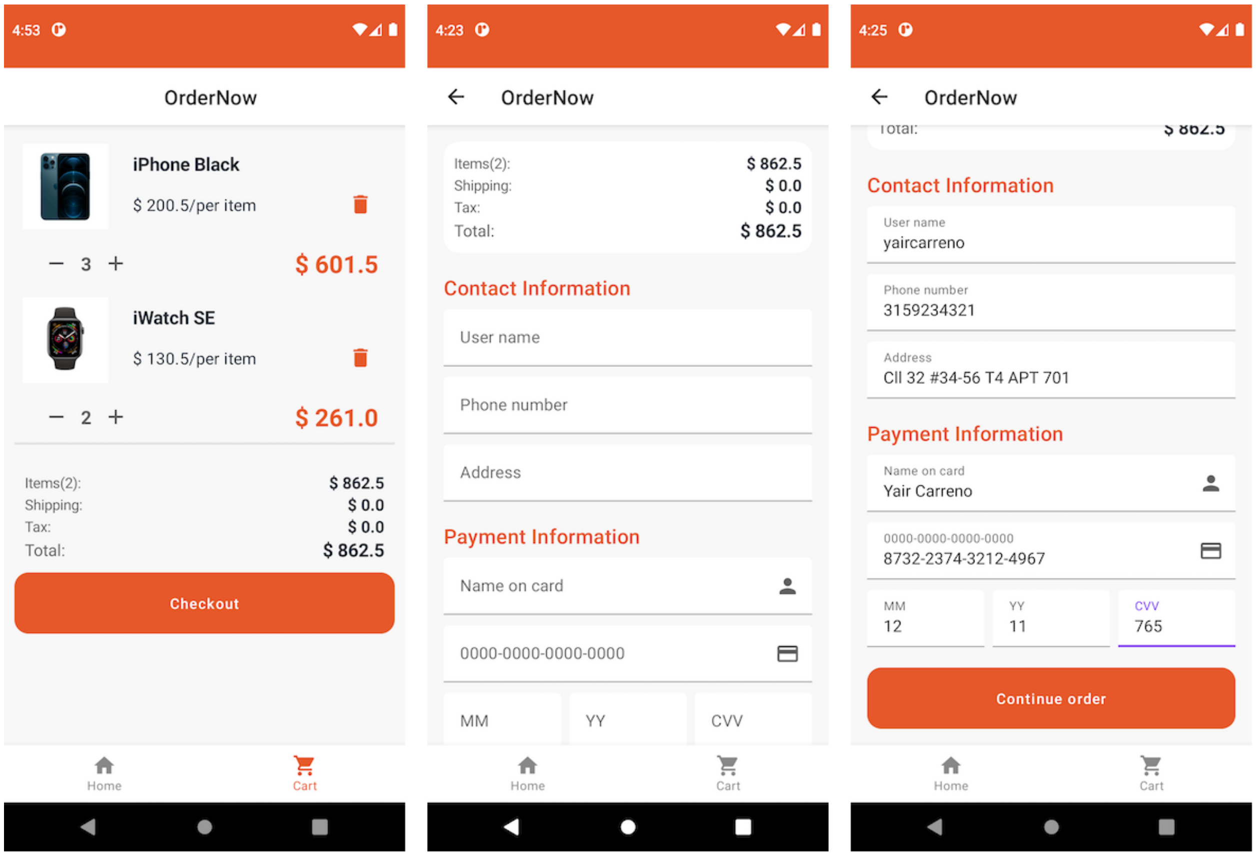


Figure 3.1 Screenshots: Home, Product List and Product detail

Cart and Checkout


4:53

OrderNow

iPhone Black
$ 200.5/per item
− 3 +
$ 601.5

iWatch SE
$ 130.5/per item
− 2 +
$ 261.0

Items(2):	$ 862.5
Shipping:	$ 0.0
Tax:	$ 0.0
Total:	$ 862.5

4:23

OrderNow

Items(2):	$ 862.5
Shipping:	$ 0.0
Tax:	$ 0.0
Total:	$ 862.5

Contact Information
User name
Phone number
Address

Payment Information
Name on card
0000-0000-0000-0000
MM YY CVV

4:25

OrderNow

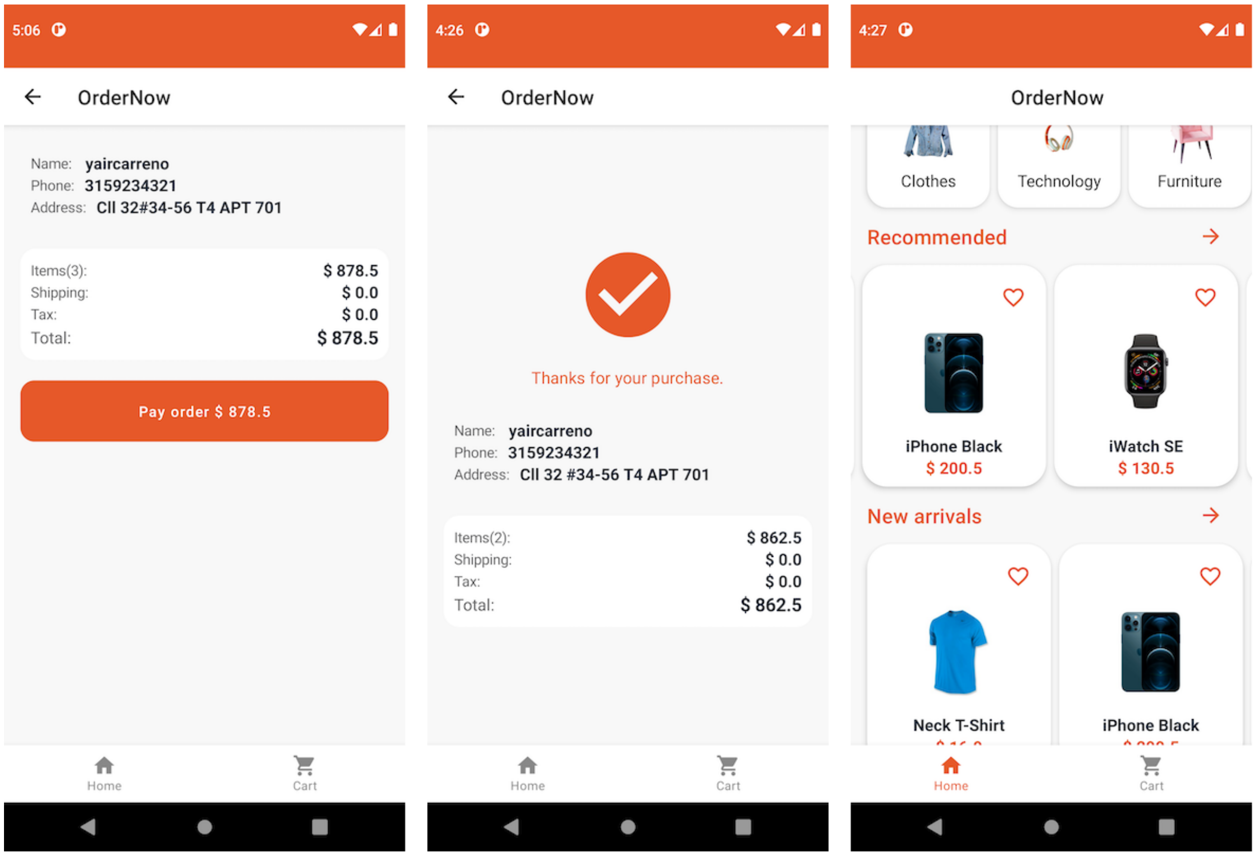
Total: $ 862.5

Contact Information
User name: yaircarreno
Phone number: 3159234321
Address: Cll 32 #34-56 T4 APT 701

Payment Information
Name on card: Yair Carreno
0000-0000-0000-0000: 8732-2374-3212-4967
MM: 12
YY: 11
CVV: 765

Figure 3.2 Screenshots: Cart and Checkout

Place Order



Name: yaircarreno
Phone: 3159234321
Address: Cll 32#34-56 T4 APT 701

Items(3): $ 878.5
Shipping: $ 0.0
Tax: $ 0.0
Total: $ 878.5

Thanks for your purchase.

Name: yaircarreno
Phone: 3159234321
Address: Cll 32 #34-56 T4 APT 701

Items(2): $ 862.5
Shipping: $ 0.0
Tax: $ 0.0
Total: $ 862.5

iPhone Black $ 200.5
iWatch SE $ 130.5
Neck T-Shirt
iPhone Black

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)*[5], which will be used to develop *OrderNow*.

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

Architecture components

- **Compose**[6]: It will be the framework for implementing declarative views in our presentation layer.
- **ViewModel**[7]: Architecture component in the presentation layer that we use to encapsulate business logic.
- **Flow**[8]: 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**[9]: Architecture component that we will use to implement navigation through the different screens of our application.

Dependencies

- Coil[10]: 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.

[6]Jetpack Compose
[7]ViewModel Overview
[8]Kotlin flows on Android
[9]Navigating
[10]Coil and Jetpack Compose