5
Car Order Form Data

When SwiftUI was first announced I spent most of the week working on designs. What was fascinating to me was being able to have so much more flexibility to do what you want but also the simplicity it took to do it. The iOS Navigational hierarchy is pretty rigid and makes this not difficult but more or less a pain to use. SwiftUI really has me made love taking a new design and recreating it in SwiftUI with limited assets. One thing I didn't expect to enjoy learning was State and Binding along with Combine.

In my job, we use Reactive programming and honestly, I am not a fan. The learning curve was steep, but I felt it was people forcing it to do something it wasn't meant for because they thought it was cool. I really enjoy taking a simple button and making it so that you do not have to create a delegate and all that comes with it. From that standpoint I definitely enjoyed it, but I felt that the library SwiftBond which is now Bond did a great job of making it super simple to use.

I feel Apple has done the same thing with using State and Binding and the more I work with it along with Combine I absolutely love it. We will see how we can leverage this in our app as well as how to use it so we do not have to pass data around like the way you would do it in a non SwiftUI app. We worked on the design of our app in the last chapter and now we are going to learn how to start leveraging State and Binding along with Combine.

In this chapter, we will be working with the following:

- Understanding State & Binding
- Combine Basics
- Networking with Combine

Understanding State & Binding

State is a topic that I really wanted to cover after you've had some understanding of SwiftUI. Using State in SwiftUI really simplifies our apps because we will now have one source of truth. When using state, a persistent storage is created by SwiftUI for each of our views. Here is an example of a State property:

```
@State private var isDriverEnabled: Bool = false
```

In this book, you have seen something similar to this but let's take the time now to break it down fully. By adding `@State` we are telling the system that the `isDriverEnabled` variable changes over time and that views will depend on this value. All changes to `@State` wrapped properties are identified by SwiftUI and then it initiates a re-rendering of the view which listen to the property values. Every change is dispersed to all of the view's children.

If you have two views inside of your main view which also need to react to a change, then these views would not use the `@State` property. The top-most view always owns the state wrapped properties. All the children views would use the `@Binding` property instead. For best practices, `@State` properties should always be private.

When you need to pass state through to other views the children of those views will use `@Binding`. Binding has read and write access to the value without any ownership. When using @Binding you do not need to worry about having default values because their values are passed in from the parent. In order to create a binding, you just need to pass a State property using the $prefix which provides a reference of the property to the child view. Here, is an example of a where we create a binding with the ChildView by passing the State property over:

```
@State private var isDriverEnabled: Bool = false


var body: some View {
    ChildView(driverEnabled: $isDriverEnabled)
}
```

Inside of the child view, you would just use the following:

```
@Binding driverEnabled: Bool
```

The @`Binding` wrapped properties are given a reference to the apps state. When values change on @`Binding` wrapped properties they trigger changes across the app to all the views dependent on that states values.

Now that we understand both `State` and `Binding`, we now can see that ownership is one of the primary differences between them.

Difference between @Binding and @State

Every view marked with `@State` has ownership and properties and using `@Binding` have read and write access but no ownership.

A good example of this would be when you are renting an apartment. You do not own the apartment you just have an agreement (aka your lease) that says you live there. If something happens to the building or anything inside the apartment, the management of the complex would take ownership and fix and/or maintain those issues.

We will use Binding and State throughout the book. Even though we will not do a ton of Combine in this book we need to at least discuss the Combine basics. Combine is a topic that is advanced and takes up an entire book. There is no way I can cover everything so please take the time to look into more detailed information on this topic.

Understanding the Combine Basics

Combine is a framework introduced in iOS 13, and it brings a native approach to Reactive Programming. Before iOS 13, RxSwift, ReactiveCocoa, and others were used. You may be familiar with reactive programming either from Swift or another programming language. Reactive Programming typically has a higher learning curve, but I think Apple has done a good job of simplifying the topic.

If you are new to reactive programming, you are probably asking why you should learn Combine. Using Combine helps you with synchronous and asynchronous tasks. For example, let's say you have a Sign-Up form. You typically have a username, two password fields and a Submit button. Let's say that your usernames are unique, and you need to verify that any new usernames don't already exist. The username will need to check the server to verify and with passwords you want to check that both passwords match. While this is happening, you want to disable the Submit button until all the criteria has been met. You can code all of these checks but using a framework like Combine makes your life easier. Let's see how in the next section.

Three Main Ingredients

Combine is composed of three main ingredients:

- Publishers
- Subscribers
- Operators

Let's look at each one of the three ingredients:

Publishers

Publishers transmit a sequence of values over time. Best way to think about Publishers and Signals is that their pattern is similar to Notification Center. There are four kinds of messages that a publisher can transmit:

1. **Subscription**: You cannot have a publisher without a Subscriber. The connection between the Publisher and Subscriber would be the subscription.
2. **Value**: The value can be any data type that you might want to be sent and received.
3. **Error**: You can transmit an error when one occurs, and the subscriber can respond accordingly.
4. **Completion**: This value is an optional value but transmits a signal when the stream has successfully ended, and no more data will be transmitted.

A publisher can be denoted as:

```
PublisherName<Output, Failure>
```

Subscribers

Subscribers declare a type that it can receive from a publisher. If your publisher is broadcasting a String type, then your Publisher must receive a String type. There are two parts to a Subscriber:

1. **Input**: The data type it can receive.
2. **Failure**: The error type it can receive.

A subscriber can be denoted as:

```
SubscriberName<Input, Failure>
```

Subscribers three key functions are receiving a subscription, receiving a value and receiving a completion or failure (error) from a publisher.

Operators

Operators are the middle men between the Publisher and the Subscriber. They converts the value to the correct type. Operators are very useful because you can chain one more together to implement very complex logic. Operators can be arranged in any order you like as there is no wrong order. Operators focal point is working with the data it receives from the previous operator, which then provides its data for the next operator in the chain.

As I stated earlier, this book is more on SwiftUI and not the Combine framework. Understanding the basics is all you need for this book and if we cover anything outside of this, I will explain what we are doing and why. Before we move back to our project let's take a look at one more topic, Networking with Combine and ObservableObject, both of which we use in this chapter.

Networking with Combine

We now have some of the basics, let's work on creating a networking layer that uses Combine. Most of this code structure is just a Networking layer but we will use URLSession. Let's get started by creating the API file. Create a new Swift file named API and save it inside of the View Models folder and then import Combine.

Then add the following:

```
struct API {

    enum Error: LocalizedError, Identifiable {

        var id: String { localizedDescription }
```

```
        case addressUnreachable(URL)
        case invalidResponse


        var errorDescription: String? {
            switch self {
                case .invalidResponse: return "The server
responded with garbage."
            case .addressUnreachable(let url): return
"\(url.absoluteString)"
            }
        }
    }


    // Add next step here
}
```

Remember, when using Subscribers, you can send Errors, and here we are creating a custom Error that we will use. Next, we need to setup our EndPoint enum. The EndPoint enum will be where we create our url and url request. Replace `// Add next step here` with the following:

```
enum EndPoint {
    static let base = URL(string: "https://reqres.in/")!


    case post


    var url: URL {
        switch self {
        case .post:
            return
EndPoint.base.appendingPathComponent("api/tesla-order")
        }
    }
```

```
    static func request(with url: URL,
                and order:OrderViewModel) -> URLRequest {

        guard let encoded = try?
              JSONEncoder().encode(order) else {

            fatalError("Invalid")

        }


        var request = URLRequest(url: url)

        request.setValue("application/json",
                    forHTTPHeaderField: "Content-Type")

        request.httpMethod = "POST"

        request.httpBody = encoded


        return request

    }
}


// Add next step here
```

Our endpoint is doing a post method and the URL we are using just sends back whatever we send. Basically, we send it our form data and it sends it back to us. The request method sets up the encoding and passes it to our URLRequest.

If you've used URLSession pre iOS 13 then you might be familiar with dataTask but with the release of iOS 13 and Combine we now have dataTaskPublisher. Finally, Replace `// Add next step here` with the following:

```
private let decoder = JSONDecoder() // (1)


// (2)
func post(with order: OrderViewModel) ->
                  AnyPublisher<OrderViewModel, Error> {

    // (3)

    URLSession.shared.dataTaskPublisher(for:
  EndPoint.request(with: EndPoint.post.url,
                    and: order))

        .map { $0.data } // (4)
```

```
        .decode(type: OrderViewModel.self,
             decoder: decoder) // (5)

        .mapError { error in // (6)

            switch error {

            case is URLError:

                return Error.
                   addressUnreachable(EndPoint.post.url)

            default: return Error.invalidResponse

            }

        }

        .print() // (7)

        .map { return $0 } // (8)

        .eraseToAnyPublisher() // (9)
}
```

We have a couple errors in this file, you can ignore them as we will fix these errors when we work on OrderViewModel. The code we just added has a lot going so let's breakdown each step:

1. We are creating an instance of `JSONDecoder()`.
2. The `post` method takes in an instance of the OrderViewModel() and returns a Publisher which has an output of OrderViewModel() and Failure of Error (our custom Error recreated earlier.
3. In this step, we are using the `dataTaskPublisher` and passing in our URL endpoint and the order.
4. Here, we use the map function to map the response json data. We then take that data and pass it down to the decode method.
5. We use the `decode` method next and this maps our json data to our model object. We also pass the `JSONDecoder()` into this method as well.
6. The `.mapError` method is used when we encounter an error.
7. The `.print()` is great for debugging when you are having issues. It is very useful to use especially when you are not getting any data.
8. Next, we use the `.map` method again which takes the object in this case the OrderViewModel and return it to whoever is calling this function.

9. Finally, `.eraseToAnyPublisher()` allows us to make the publisher an instance of AnyPublisher.

We will use this API a bit later but for now we are ready to make a "Post" request to our API and get a response mapped to our OrderViewModel. We haven't created this file yet so let's work on that next.

Understanding an ObservableObject

Earlier, we discussed State and Binding and you use State and Binding when you are working inside of the View. When you move properties into a model object the name changes a bit but overall the concept is the same.

The `ObservableObject` is part of both the Foundation and Combine framework. An `ObservableObject` is a custom class that keeps track of state. Using `ObservableObject` allows you to create Environment objects, which means you can have one source of truth.

Instead of using @State and @Binding you will create an object that subclasses `ObservableObject` and uses @Published instead of @State or @Binding. You might wonder when to use one over the other, if you need to keep state outside of the view use an `ObservableObject`. When you have state that lives only in one view then use @State and @Binding. Let's create our first `ObservableObject`.

Creating our Order View Model

Inside of the `OrderViewModel`, we have some basic prototyping code that we can delete. Now we need to add all of State driven code next. Now, add the following inside of `OrderViewModel`:

```
class OrderViewModel: ObservableObject {


    @Published var rentalAmount = 0
    @Published var amountOfCars = 0
    @Published var location = 0
    @Published var returnLocation = 0
    @Published var pickupTime = 0
    @Published var specialDriver = true
    @Published var isOrderCompleteVisible = false
    @Published var isCancelOrderVisible = false
    @Published var isModalVisible = false
```

```
    let pickupTimes = Array(stride(from: 60, through: 480,
by: 10))
    let rentalPeriods = Array(1..<5)
    let numberOfCars = Array(1..<4)
    let locations = ["MIA Inter. Airport",
                     "Ft. Lauderdale Inter. Airport",
                     "Palm Beach Inter. Airport"]

    let returnLocations = ["MIA Inter. Airport",
                           "Ft. Lauderdale Inter. Airport",
                           "Palm Beach Inter. Airport"]

    // Add Next step here
}
```

All of this code drives the data for our switch and pickers inside of our Form. Now this is all the code you need to drive that form part. We are now going to work on the making our code work with Codable.

Conforming to Codable

Codable allows us to work with JSON and we will make it conform to JSON both for sending and receiving data. To conform to codable follow these steps:

Inside of `OrderViewModel`, update `class OrderViewModel: ObservableObject` to `class OrderViewModel: ObservableObject, Codable`. Now that we are subclassing Codable we must conform to it. Conforming, will get rid of the errors inside of OrderViewModel, they will also get rid of the errors we were seeing in API.

Next, add the following by replacing `// Add Next step here` with the following:

```
private let api = API()
private var subscriptions = Set<AnyCancellable>()

init() {}

enum CodingKeys: String, CodingKey {
```

```
    case rentalAmount, amountOfCars, location, specialDriver,
pickupTime

}


// Add Next step here
```

The first variable we added is a variable for the API we created, and the second variable is used subscription we set to AnyCancellable which automatically deinits and tears down the subscription stream when done.

The enums we added are `CodingKey` we use for encoding and decoding. Let's move to the next step by replacing `// Add Next step here` and add the following:

```
required init(from decoder: Decoder) throws {

    let values = try decoder.container(keyedBy:
CodingKeys.self)


    rentalAmount = try values.decode(Int.self, forKey:
.rentalAmount)

    amountOfCars = try values.decode(Int.self, forKey:
.amountOfCars)

    location = try values.decode(Int.self, forKey: .location)

    pickupTime = try values.decode(Int.self, forKey:
.pickupTime)

    specialDriver = try values.decode(Bool.self, forKey:
.specialDriver)

}


// Add Next step here
```

We just created our decoder and next we will add our encode method:

```
func encode(to encoder: Encoder) throws {

    var container = encoder.container(keyedBy:
CodingKeys.self)


    try container.encode(rentalAmount, forKey: .rentalAmount)

    try container.encode(amountOfCars, forKey: .amountOfCars)

    try container.encode(location, forKey: .location)
```

```
    try container.encode(pickupTime, forKey: .pickupTime)


    try container.encode(specialDriver, forKey:
.specialDriver)
}


// Add Next step here
```

These last two methods are basics for working with models and JSON. Finally, we need to create our `sendOrder()` method for sending our form data. Replace `// Add Next step here` with the following method:

```
func sendOrder() {
    api.post(with: self)
        .receive(on: DispatchQueue.main)
        .sink(receiveCompletion: { response in
            print(response)
            print("=================")
        }, receiveValue: { value in
            print("Received response from Combine publisher")
            print(value)
    }).store(in: &subscriptions)
}
```

We have our `sendOrder()` method which we pass the `OrderViewModel` into the `post` method. We are using more Combine methods here and the `receive` method makes sure that this call is on the main thread. We are using the `.sink` method which handles receiving events from the publisher using Combine.

Now that our data is setup, we can now move on and update our UI to use the newly created data.

Creating an Environment Object

In SwiftUI, we can setup an environment object which allows us to have one source of truth. In our case, we will use our OrderViewModel as our environment object. Open the `SceneDelegate` and add the following on the line above the constant `contentView`:

```
let order = OrderViewModel()
```

Then replace `contentView` with the following:

```
let contentView = ContentView().environmentObject(order)
```

We now have an environment object setup; we already have our object setup in each of our views, but we need to make some updates to use the new properties we just added.

Updating FormView

Inside of our `FormView`, we are using prototype variables and now that we have the actual variables setup, we need to update our pickers to use the correct data. So, to begin with we need to add an instance of our OrderViewModel(). Let's do that by changing

```
@ObservedObject var order = OrderViewModel()
```

To the following:

```
@EnvironmentObject var order:OrderViewModel
```

Update the first picker to use `rentalAmount` instead of `prototypeAmt` and `rentalPeriods` instead of `prototypeArray`:

```
Picker(selection: $order.rentalAmount, label: Text("Rental
period")) {
    ForEach(0 ..< order.rentalPeriods.count, id: \.self) {
        Text("\(self.order.rentalPeriods[$0])").tag($0)
    }
}
```

Note

Inside of the `ForEach` I added `id`. If you do not add `id` you will get a warning message in the console.

In the next picker, use `amountOfCars` and `numberOfCars`:

```
Picker(selection: $order.amountOfCars, label: Text("Number of
cars")) {
    ForEach(0 ..< order.numberOfCars.count, id: \.self) {
        Text("\(self.order.numberOfCars[$0])").tag($0)
    }
}
```

Let's move to the next picker and use `pickupTime` and `pickupTimes`:

```
Picker(selection: $order.pickupTime, label: Text("Pick-up
time")) {
    ForEach(0 ..< order.pickupTimes.count, id: \.self) {
        Text("In \(self.order.pickupTimes[$0]) mins").tag($0)
    }
}
```

Move to the last two pickers and use `location` and `locations` for the first picker and `returnLocation` and `returnLocations` for the second picker:

```
Picker(selection: $order.location, label: Text("Pick-up
location")) {
    ForEach(0 ..< order.locations.count, id: \.self) {
        Text("\(self.order.locations[$0])").tag($0)
    }
}


Picker(selection: $order.returnLocation, label: Text("Return
location")) {
    ForEach(0 ..< order.returnLocations.count, id: \.self) {
        Text("\(self.order.returnLocations[$0])").tag($0)
    }
}
```

Next, let's update the `Toggle` to use `specialDriver`:

```
Toggle(isOn: $order.specialDriver) {
    Text("Drivers")
}
```

Finally, update your button action to the following:

```
Button(action: {
    self.order.isOrderCompleteVisible.toggle()
    self.order.sendOrder()
})
```

We are done updating `FormView` let's move to `CancelOrderView` next.

CancelOrderView

Inside of CancelOrderView, we need to add an instance of our OrderViewModel(). Do this by adding the following above the body variable:

```
@EnvironmentObject var order:OrderViewModel
```

Next, we need to update the ZStack main container by adding the following on to the `ZStack`:

```
.onAppear {
    self.order.isModalVisible = true
}.onTapGesture {
    self.order.isCancelOrderVisible.toggle()
    self.order.isModalVisible.toggle()
}
```

When you are finished your code should look like the following:

```
var body: some View {
    ZStack {
        // previous code minimized to make it easier to see
    }
    .onAppear {
        self.order.isModalVisible = true
    }.onTapGesture {
        self.order.isCancelOrderVisible.toggle()
        self.order.isModalVisible.toggle()
    }
}
```

I removed the code inside of the `ZStack` for clarity, but there are no changes made inside of this `ZStack`. We are added two properties `.onAppear` and `.onTapGesture`. The `.onAppear` is just going to set the `isModalVisible` to `true`. The `.onTapGesture` toggles both the `.isCancelOrderVisible` and `.isModalVisible`. Let's move over to the `CancelModalView` next.

Updaing the Cancel Modal View

Inside of the `CancelModalView`, we need to add an instance of our OrderViewModel(). Do this by adding the following above the body variable:

```
@EnvironmentObject var order:OrderViewModel
```

Next, we have our button actions. These buttons basically toggle visibility for our modal. There really is no purpose for these buttons as they are just for show. You can do whatever you want with these. In the completed project, I have them set to the following:

```
Button("NO, KEEP IT") {
    self.order.isCancelOrderVisible.toggle()
    self.order.isModalVisible.toggle()
}
Button(action: {
    self.order.isCancelOrderVisible.toggle()
    self.order.isModalVisible.toggle()
})
```

Both, the Cancel button and the Keep it button are doing the exact same thing we are just toggling the visibility if this was going to be an actual app then the ”Cancel” button would also cancel the order whereas the “Keep it” button would just confirm the order. Let’s move on to the `TopOrderView` next.

Updating Top Order View

Inside of the `TopOrderView`, we need to add an instance of our OrderViewModel(). Add the following above the body variable:

```
@EnvironmentObject var order:OrderViewModel
```

Now, update the button action to the following:

```
Button(action: { self.order.isOrderCompleteVisible.toggle()
}) {
    Image("close-btn")
}
```

Again, we are toggling the visibility of our modal and finally open `BottomOrderView`.

Updating Bottom Order View

Inside of the `BottomOrderView`, we need to add an instance of our OrderViewModel(). Add the following above the body variable:

```
@EnvironmentObject var order:OrderViewModel
```

Next, update the button action to the following:

```
Button(action: { self.order.isCancelOrderVisible.toggle() }){
    Text("CANCEL ORDER")
}
```

As we did in `TopOrderView` we are toggling the visibility of our modal. We are done updating our data for our app. You should now be able to send off your information using URLSession. We are using an environment object and have one source of truth even though our app is only a couple of files. You will notice that if you try to build and run and you hit Complete Order nothing happens. Open ContentView and add an instance of our OrderViewModel(). Add the following above the body variable:

```
@EnvironmentObject var order:OrderViewModel
```

Finally, look for the following:

```
CompleteOrderView()
    .opacity(0)
    .animation(.default)
```

Then, update the .opacity to the following:

```
CompleteOrderView()
    .opacity(order.isOrderCompleteVisible ? 1 : 0)
    .animation(.default)
```

Updating Complete Order View

Inside of the `CompleteOrderView`, we need to add an instance of our CompleteOrderView(). Add the following above the body variable:

```
@EnvironmentObject var order:OrderViewModel
```

Next, replace `// Add Cancel Order View` here to the following:

```
CancelOrderView()
    .opacity(order.isCancelOrderVisible ? 1 : 0)
    .animation(.default)
```

Congratulations, we are finally done. If you build the app you can now change values inside of the form and send them off once you hit “Complete Order”. After you hit “Complete Order” you will see the Order Completed screen which if you hit cancel you will see our custom modal with a blurred background. Hitting either buttons will just dismiss the screen.

Summary

In this chapter, we have looked at State and Binding and how it works inside of our Views. We also looked at `@Published` and how it works inside of an `ObservableObject`. We also looked at some Combine basics and how we can do networking with Combine. Finally, we created an `environmentObject` for our project in order to have one source of truth.

Let's move on to the next chapter and work on a new project. In the next chapter, we will create a financial app which uses Core Data. We will add on to the skills you learned in this chapter.