UIKIT WITH SWIFTUI RECIPES

BY MOHAMMAD AZAM

Welcome to the recipes book on UIKit with SwiftUI. This book contains solutions to the common problems you will encounter, when using SwiftUI in a UIKit application and vice versa.

GitHub: https://github.com/azamsharp/UIKitUsingSwiftUIRecipes

You can also follow me on Twitter @azamsharp

How to load a SwiftUI view as a separate view in a UIKit application?

One of the best things about SwiftUI is that it is interoperable with UIKit framework. This means, views created in SwiftUI can easily be loaded into existing UIKit applications. In **Listing 1** we have created a StocksScreen in SwiftUI, which displays information about few stocks.

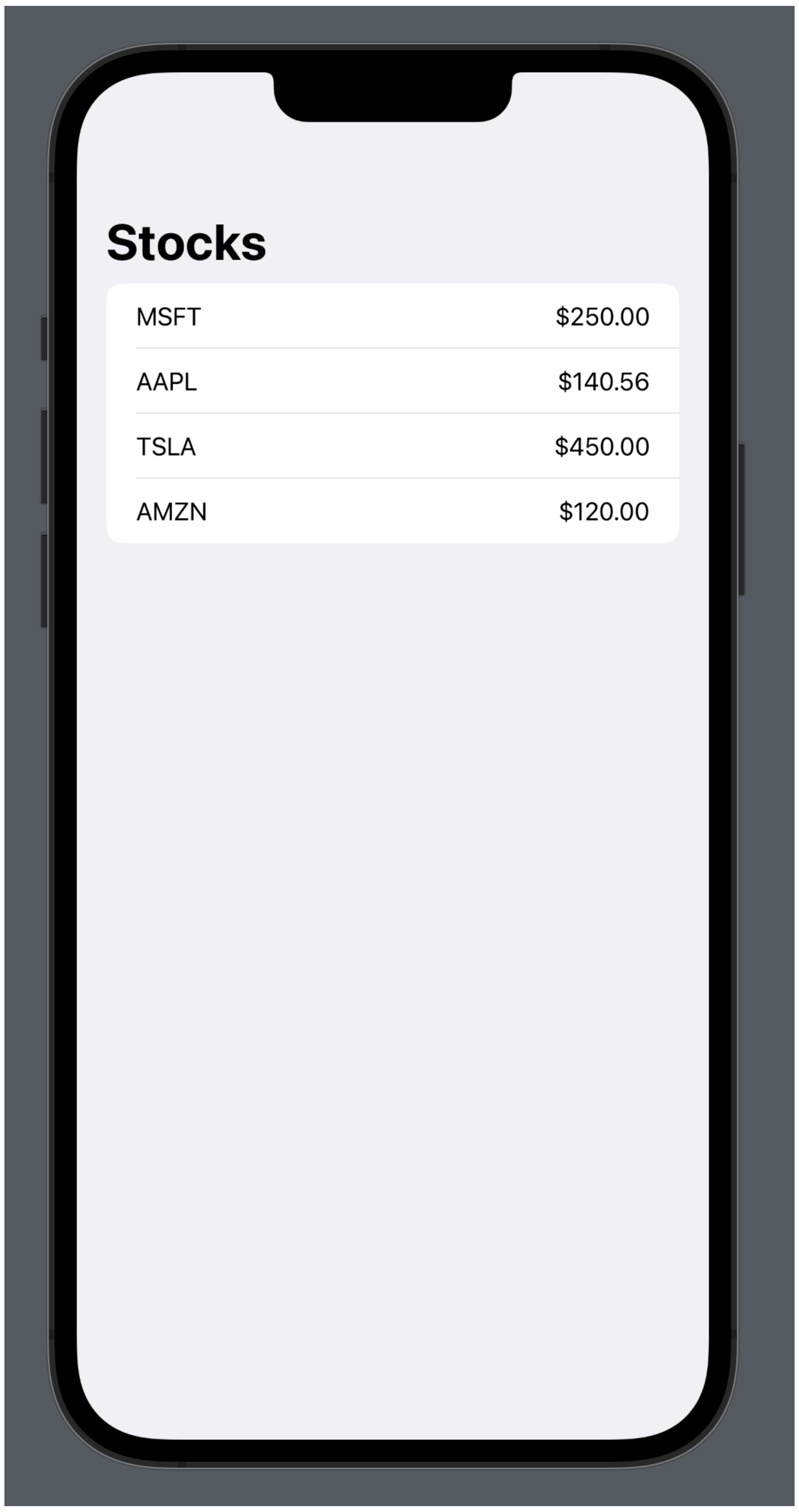
```
struct StockListScreen: View {

    let stocks = [Stock(name: "MSFT", price: 250), Stock(name: "AAPL", price:
140.56),Stock(name: "TSLA", price: 450), Stock(name: "AMZN", price: 120.00)]

    var body: some View {
        List(stocks) { stock in
            HStack {
                Text(stock.name)
                Spacer()
                Text(stock.price.formatAsCurrency())
            }
        }.navigationTitle("Stocks")
    }
}
```

Listing 1: StockListScreen SwiftUI view

The output is shown in **Figure 1**.


Stocks

MSFT	$250.00
AAPL	$140.56
TSLA	$450.00
AMZN	$120.00

Figure 1: StockListScreen SwiftUI view

Now, if we want to load StockListScreen view in our UIKit application then we can do that using the implementation in **Listing 2**.

```
class ViewController: UIViewController {

    lazy var navigateToStocksButton: UIButton = {
        let button = UIButton(type: .roundedRect)
        button.setTitle("Navigate to Stocks", for: .normal)
        button.translatesAutoresizingMaskIntoConstraints = false
        return button
    }()

    override func viewDidLoad() {
        super.viewDidLoad()
        self.view.backgroundColor = UIColor.green

        navigateToStocksButton.addAction(UIAction { _ in
self.navigationController?.pushViewController(UIHostingController(rootView:
StockListScreen()), animated: true)
        }, for: .touchUpInside)


        self.view.addSubview(navigateToStocksButton)

       // setup constraints ….
    }

}
```

Listing 2: UIKit controller navigating to the StockListScreen on button press

The heart of this integration between UIKit and SwiftUI is the UIHostingController. UIHostingController allows to manage the SwiftUI view hierarchy. If you run the app and tap on the "**Navigate to Stocks**" button then it will perform a navigation and take you to the StockListScreen (SwiftUI View).

How to embed an existing SwiftUI view into a UIKit UIView?

There are situations where you want to insert a SwiftUI view into an existing UIKit view instead of displaying a brand new view implemented in SwiftUI. **Listing 3** shows implementation of a RatingView in SwiftUI.

```
import SwiftUI

struct RatingView: View {

    @Binding var rating: Int?

    private func starType(index: Int) -> String {

        if let rating = self.rating {
            return index <= rating ? "star.fill" : "star"
        } else {
            return "star"
        }

    }

    var body: some View {
        HStack {
            ForEach(1...5, id: \.self) { index in
                Image(systemName: self.starType(index: index))
                    .foregroundColor(Color.orange)
                    .onTapGesture {
                            self.rating = index
                        }
                    }
                }
            }
}
```

Listing 3: RatingView implementing in SwiftUI

RatingView is responsible for displaying star ratings to the user. If we want to display RatingView in our existing UIKit view then we will take help from the UIHostingController. This is shown in **Listing 4**.

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        self.view.backgroundColor = .white

        let hostingController = UIHostingController(rootView:
RatingView(rating: .constant(3)))


        guard let ratingsView = hostingController.view else { return }
        self.addChild(hostingController)

        self.view.addSubview(ratingsView)

        // add constraints on the ratings view

    }

}
```

Listing 4: Adding RatingView to the ViewController

First we add the RatingView as a rootView to the UIHostingController. Next we add the hosting controller as a child controller to the container controller. Finally, we add the ratingsView to the main view. Make sure to add constraints to the ratingsView so it can be centered on the screen. **Figure 2** shows the final result.

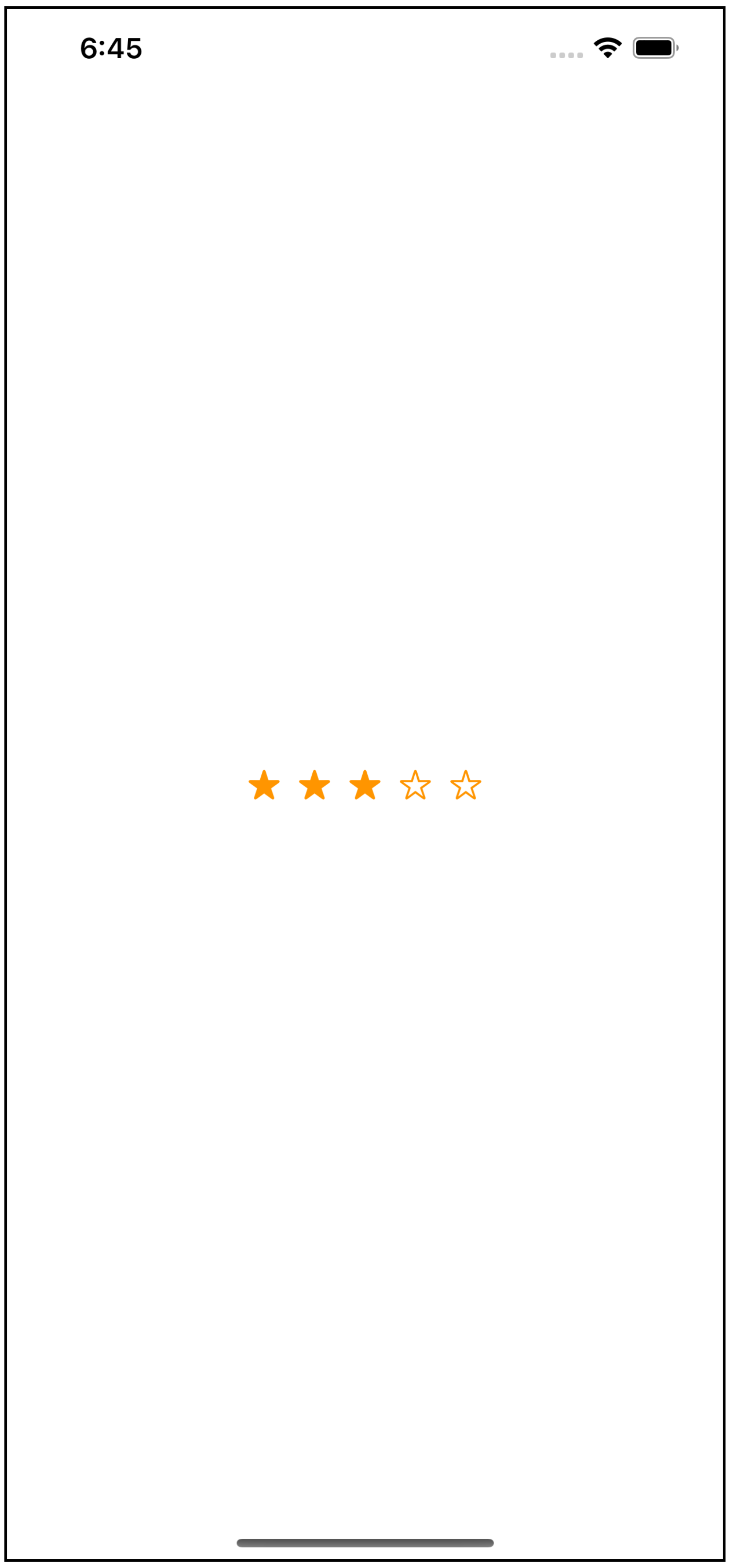


Figure 2: RatingView added to the UIKit view

How to send a value from a SwiftUI view to a UIKit view?

Loading a SwiftUI view and embedding it into a UIKit view is great, but sometimes we need to access a value from a SwiftUI view into our UIKit view. For this scenario, we will be working on the same RatingView with one limitation, we are not allowed to change the implementation of the RatingView.

This will certainly spice things up!

Our first implementation is shown in **Listing 5**.

```
class ViewController: UIViewController {

   @State private var rating: Int? = 3

    override func viewDidLoad() {
        super.viewDidLoad()
        self.view.backgroundColor = .white

        let hostingController = UIHostingController(rootView:
RatingView(rating: $rating))

        guard let ratingsView = hostingController.view else { return }
        self.addChild(hostingController)

        self.view.addSubview(ratingsView)


        // add constraints on the ratings view

    }
```

Listing 5: Using @State inside the ViewController

We are creating a rating variable inside the ViewController and decorating it with @State property wrapper. Later we passed rating as a binding to the RatingView. All looks nice but when you run the app, you will get the following warning.

Accessing State's value outside of being installed on a View. This will result in a constant Binding of the initial value and will not update.

The warning is due to the fact that you are trying to access @State property wrapper outside the SwiftUI view and that is not allowed.

One way to solve this problem is to pass a ObservableObject to the RatingView and update one of the properties of that ObservableObject. But remember, we are not allowed to change the implementation of the RatingView.

Another method that was suggested by Asperi and Andrew here involves wrapping the RatingView with a container/parent view. This is implemented in **Listing 6**.

```swift
class ViewController: UIViewController, ObservableObject {

    @Published var rating: Int? = 3
    var cancellable: AnyCancellable?


    lazy var ratingLabel: UILabel = {

        let label = UILabel()
        label.translatesAutoresizingMaskIntoConstraints = false
        label.textAlignment = .center
        return label

    }()


    private struct HolderView: View {

        @ObservedObject var vc: ViewController

        var body: some View {
            RatingView(rating: $vc.rating)
        }
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        self.view.backgroundColor = .white

        let hostingController = UIHostingController(rootView: HolderView(vc:
self))

        guard let ratingsView = hostingController.view else { return }
        self.addChild(hostingController)

        self.view.addSubview(ratingsView)


        self.cancellable = $rating.sink { [weak self] rating in

            if let rating {
                // Now you have access to the rating
                self?.ratingLabel.text = "\(rating)"
            }
        }

        // stack view
        let stackView = UIStackView(arrangedSubviews: [ratingLabel,
ratingsView])
        stackView.axis = .vertical

        self.view.addSubview(stackView)

        // add constraints on the ratings view
    }
}
```

Listing 6: Container view to wrap RatingView to access ObservedObject

The main point to note in **Listing 6** is the implementation of the **HolderView**. HolderView acts as a container for the RatingView and sets ViewController as an observed object. The ViewController consists of a property "rating" which is marked with @Published. This means when this property is set, it publishes an event which is handled in the view controller. ViewController subscribes to the rating by using the sink function and this allows the ViewController to get the latest value from the RatingView, without changing the implementation of the RatingView.

How to communicate between a UIKit view and SwiftUI view using a View Model?

In **Listing 6**, we loaded a SwiftUI view inside a UIKit view through the use of a container view (HolderView). Another way to communicate between a UIKit and SwiftUI view is by using a View Model. In this example, we will create a SwiftUI counter view. When the counter is pressed the updated value is passed to the UIKit and displayed on the label.

We will start by creating our CounterView and the CounterViewModel as shown in **Listing 7**.

```
class CounterViewModel {
    @Published var value: Int = 0
}

struct CounterView: View {

    let vm: CounterViewModel


    var body: some View {
        Button("Increment") {
            self.vm.value += 1
        }.buttonStyle(.borderedProminent)
    }
}
```

Listing 7: Implementation for CounterViewModel and CounterView

The CounterViewModel consists of a single property called value, which is marked with the **@Published** attribute. **CounterViewModel** is not decorated with the **@ObservableObjec**t since the CounterView has no intention of holding or displaying the counter value. Once, the button is pressed we simply increment the value property of the view model.

Now, let's take a look at the CounterViewController implementation in **Listing 8**.

```
class CounterViewController: UIViewController {

    let vm = CounterViewModel()
    var cancellable: AnyCancellable?

    lazy var counterLabel: UILabel = {

        let label = UILabel()
        label.translatesAutoresizingMaskIntoConstraints = false
        label.textAlignment = .center
        return label

    }()

    override func viewDidLoad() {
        super.viewDidLoad()

        self.view.backgroundColor = .white
        let hostingController = UIHostingController(rootView: CounterView(vm:
vm))

        guard let counterView = hostingController.view else { return }
        self.addChild(hostingController)

        self.view.addSubview(counterView)

        // add subscriptions
        self.cancellable = vm.$value.sink { [weak self] value in
            self?.counterLabel.text = "\(value)"
        }

        // stack view
        let stackView = UIStackView(arrangedSubviews: [counterLabel,
counterView])
        stackView.axis = .vertical
        self.view.addSubview(stackView)

        // adding constraints

    }
}
```

Listing 8: CounterViewController using CounterView

The CounterViewController creates an instance of CounterViewModel and then passes it to the CounterView, which is loaded using the UIHostingController.

Later, CounterViewController subscribe to the changes of the view model using the sink function on the publisher. This means whenever the value property is changed, the sink is going to get triggered, sending the new value in the closure to the counter view controller . Finally, the value is displayed using a UILabel called counterLabel.

How to communicate between UIKit view and SwiftUI view using View Model and update both views?

In the last recipe, if you try to display the updated counter value in a SwiftUI view, it will not work. This is shown in **Listing 9**.

```
struct CounterView: View {

    let vm: CounterViewModel


    var body: some View {
        VStack {
             // THIS WILL NOT GET UPDATED
            Text("\(vm.value)")
            Button("Increment") {
                self.vm.value += 1
            }.buttonStyle(.borderedProminent)
        }
    }
}
```

Listing 9: CounterView not displaying updated value of the counter

The main reason is that there is nothing telling the counter view to reload/refresh itself when the value increments. We can fix this problem by making sure that the CounterViewModel conforms to the ObservableObject protocol. This will allow us to use **@StateObject** or **@ObservedObject** property wrappers that can maintain the value between refreshes. In this particular scenario both @StateObject and @ObservedObject will work, since we only have single view "CounterView" and the counter view does not have any child views depending on the values from the view model. To learn more about the differences between @StateObject and @ObservedObject check out this free YouTube video.

The implementation is shown in **Listing 10**.

```
class CounterViewModel: ObservableObject {
    @Published var value: Int = 0
}

struct CounterView: View {

    @StateObject var vm: CounterViewModel

    var body: some View {
        VStack {
            Text("\(vm.value)")
            Button("Increment") {
                self.vm.value += 1
            }.buttonStyle(.borderedProminent)
        }
    }
}
```

Listing 10: Displaying updated counter value using @StateObject and ObservableObject

How to communicate between UIKit view and SwiftUI view using global environment object?

Sometimes, you have a scenario where you want to share state with multiple SwiftUI views. This can be achieved by using the **@EnvironmentObject** property wrapper, which represents the global state of the application. In this recipe, we are going to add another SwiftUI called "FancyCounterView".

FancyCounterView will also display the updated counter value, along with CounterView and the CounterViewController.

We will begin by creating a global state object. This is shown in **Listing 11**.

```
class AppState: ObservableObject {
    @Published var counter: Int = 0
}
```

Listing 11: Global state object called AppState

We will update our implementation for **CounterView** and also create a brand new view called **FancyCounterView.** Both views make use of the global state. This is shown in **Listing 12**.

```
struct FancyCounterView: View {

    @EnvironmentObject var appState: AppState

    var body: some View {
        VStack {
            Text("Fancy Counter View")
            Text("\(appState.counter)")
        }
    }
}


struct CounterView: View {

    @EnvironmentObject var appState: AppState

    var body: some View {
        VStack {
            Text("\(appState.counter)")
            Button("Increment") {
                self.appState.counter += 1
            }.buttonStyle(.borderedProminent)

            FancyCounterView()
                .padding()
                .foregroundColor(.white)
                .background {
                    Color.green
                }
        }
    }
}
```

Listing 12: CounterView and FancyCounterView using global state

Now, finally we will update the CounterViewController to inject the AppState as an environment object to the root view. This is shown in **Listing 13**.

```
 let appState = AppState()
 let hostingController = UIHostingController(rootView:
CounterView().environmentObject(appState))
```

Listing 13: Injecting global state from CounterViewController

If you run the app and click on the increment button, you will notice that the counter value is updating in CounterView, FancyCounterView and the CounterViewController. EnvironmentObject is a great option to share state with multiple SwiftUI views.

How to load a UIKit view in a SwiftUI application?

Sometimes you have a scenario, where you have to load a UIKit view into a SwiftUI application. In order to load a UIKit view into a SwiftUI view, you will have to implement **UIViewRepresentable** protocol. UIViewRepresentable protocol allows you to wrap your UIKit view to integrate into a SwiftUI application.

Take a look at **Listing 14**, where we have represented a UIActivityIndicatorView in UIKit so it can be used in a SwiftUI view.

```
import SwiftUI

struct LoadingView: UIViewRepresentable {

    var loading: Bool

    func makeUIView(context: Context) -> UIActivityIndicatorView {
        let activityIndicatorView = UIActivityIndicatorView(style: .medium)
        return activityIndicatorView
    }

    func updateUIView(_ uiView: UIActivityIndicatorView, context: Context) {
        if loading {
            uiView.startAnimating()
        } else {
            uiView.stopAnimating()
        }
    }

    typealias UIViewType = UIActivityIndicatorView

}
```

Listing 14: Wrapping a UIKit view through the use of UIViewRepresentable

The UIViewRepresentable protocol consists of few mandatory functions, which includes makeUIView and updateUIView. The purpose of makeUIView is to construct and return a UIKit view. The updateUIView is called whenever source of truth is changed.

Now, we can use our LoadingView as shown in **Listing 15**.

```
struct ContentView: View {

    @State private var isLoading: Bool = false

    var body: some View {
        VStack {
            LoadingView(loading: isLoading)
            Button(“Toggle") {
                isLoading.toggle()
            }
        }
    }
}
```

Listing 15: Wrapping a UIKit view through the use of UIViewRepresentable

Now, when you tap the “**Toggle**” button you can view and hide the activity indicator view.

NOTE: We used the UIActivityIndicatorView just for demonstration purposes. If you are trying to implement a loading view then SwiftUI already consists of ProgressView.

How to implement delegate methods of a UIKit view in a SwiftUI application?

Sometimes when you are loading a UIKit view in a SwiftUI application, you also need to implement the delegate methods exposed by UIKit view. This can be for variety of reasons, maybe you need to handle a specific UI event, which is exposed through those delegate methods.

In **Listing 16** we have created a simple MapView using the UIViewRepresentable protocol. This allows us to use MKMapView in a SwiftUI application.

```
import Foundation
import MapKit
import SwiftUI

struct MapView: UIViewRepresentable {

    typealias UIViewType = MKMapView

    func makeUIView(context: Context) -> MKMapView {
        let map = MKMapView()
        return map
    }

    func updateUIView(_ uiView: MKMapView, context: Context) {

    }

}

struct MapViewScreen: View {
    var body: some View {
        VStack {
            MapView()
        }
    }
}
```

Listing 16: MapView using the UIViewRepresentable protocol

The result is shown in **Figure 3**.

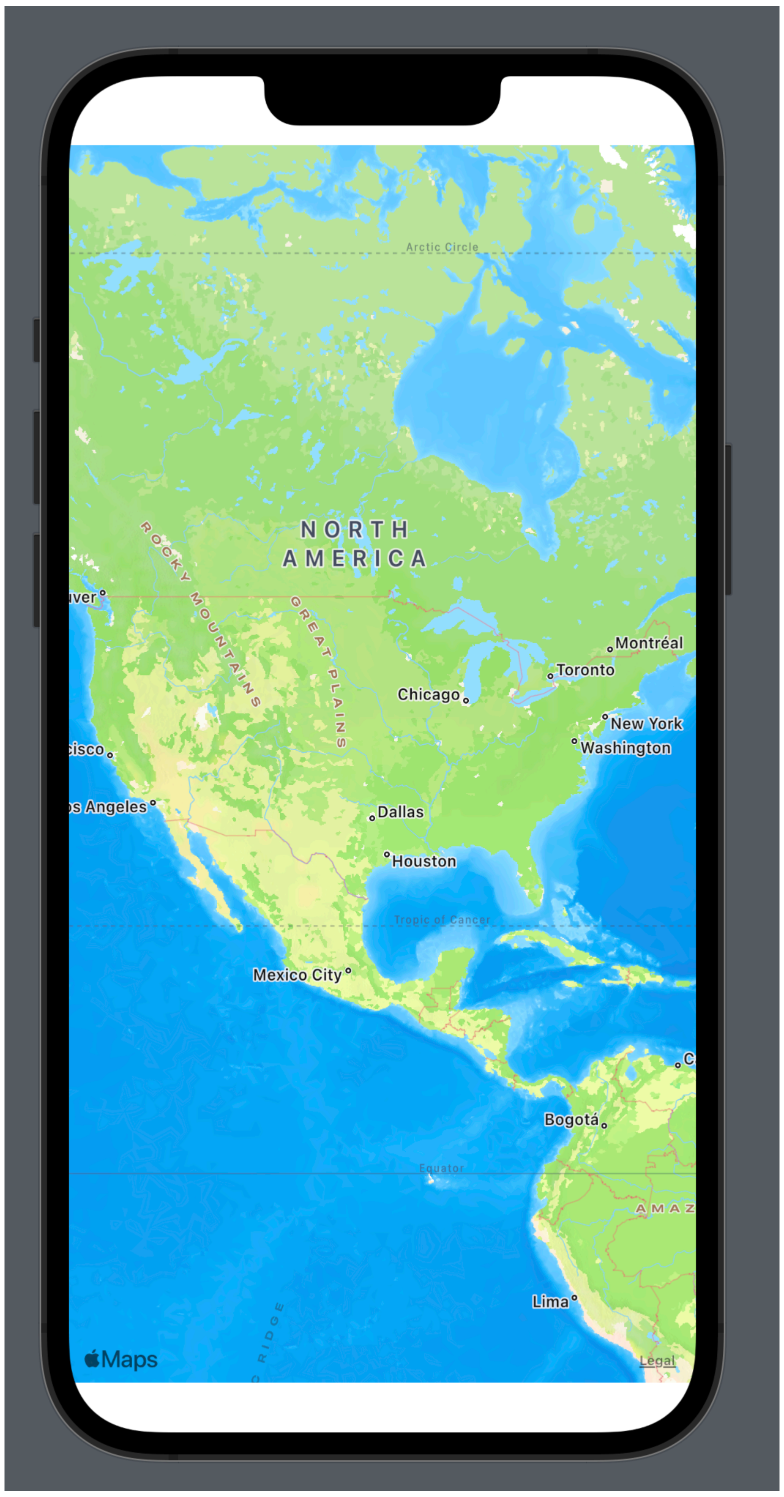


Figure 3: MapView using the UIViewRepresentable protocol

Pretty simple right!

We can even add an annotation on the map using MKPointAnnotation in the MapView struct. This is shown in **Listing 17**.

```
struct MapView: UIViewRepresentable {

    typealias UIViewType = MKMapView

    func makeUIView(context: Context) -> MKMapView {
        let map = MKMapView()

        // add annotation
        let pointAnnotation = MKPointAnnotation()
        pointAnnotation.title = "Apple Campus"
        pointAnnotation.coordinate = CLLocationCoordinate2D(latitude:
37.331821, longitude: -122.031181)

        map.addAnnotation(pointAnnotation)

        return map
    }


    func updateUIView(_ uiView: MKMapView, context: Context) {

    }


}
```

Listing 17: Adding annotation to the map

The result is shown in **Figure 4**.

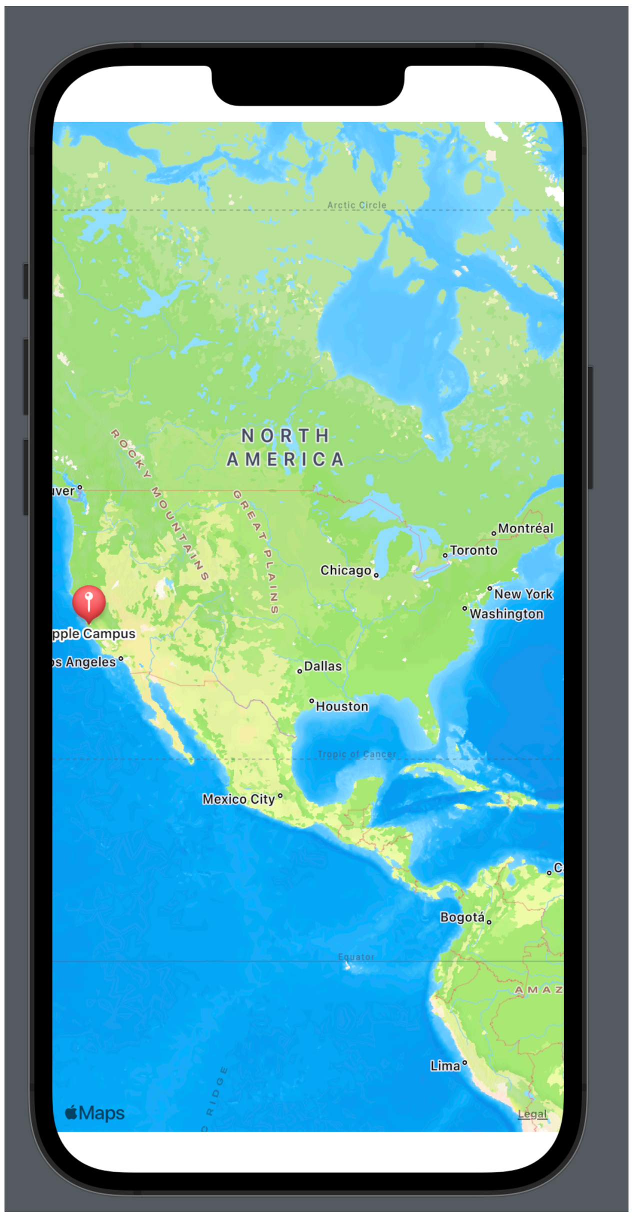


Figure 4: Displaying annotation on the map

Great!

But what if you want to change the annotation view. Maybe you are interested in displaying an Apple logo instead of the default annotation marker. Luckily, MKMapView provides delegate methods that allows you to change the

In order to use the delegate methods, we need to create a coordinator. The coordinator is implemented in **Listing 18**.

```
class MapViewCoordinator: NSObject, MKMapViewDelegate {
    var mapView: MKMapView?
}
```

Listing 18: Implementation of MapViewCoordinator

The MapViewCoordinator class is pretty basic at the moment. At present it inherits from **NSObject** and conforms to **MKMapViewDelegate** protocol. It also consists of the mapView property, which will be assigned later.

Next, we need to update our MapView to return the newly created MapViewCoordinator and also set the MKMapView delegate to our coordinator. This is implemented in **Listing 19**.

```
struct MapView: UIViewRepresentable {

    typealias UIViewType = MKMapView

    func makeUIView(context: Context) -> MKMapView {
        let map = MKMapView()
        map.delegate = context.coordinator
        context.coordinator.mapView = map

        // add annotation
        let pointAnnotation = MKPointAnnotation()
        pointAnnotation.title = "Apple Campus"
        pointAnnotation.coordinate = CLLocationCoordinate2D(latitude:
37.331821, longitude: -122.031181)

        map.addAnnotation(pointAnnotation)

        return map
    }


    func updateUIView(_ uiView: MKMapView, context: Context) {

    }
    func makeCoordinator() -> MapViewCoordinator {
        MapViewCoordinator()
    }

}
```

Listing 19: Returning coordinator from the MapView

If you run the app, nothing will change it will still shows you the default marker as an annotation. But now you have an option to return a different view for your annotation through the use of MKMapViewDelegate protocol. Update your MapViewCoordinator and implement the **viewForAnnotation** delegate function as shown in **Listing 20**.

```
class MapViewCoordinator: NSObject, MKMapViewDelegate {

    var mapView: MKMapView?

    override init() {
        super.init()
        registerMapAnnotationViews()
    }

    private func registerMapAnnotationViews() {

        guard let mapView = mapView else {
            return
        }

        mapView.register(AppleMarkerAnnotationView.self,
forAnnotationViewWithReuseIdentifier:
NSStringFromClass(AppleMarkerAnnotationView.self))
    }

    func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) ->
MKAnnotationView? {

        switch annotation {
            case is MKPointAnnotation:
                return AppleMarkerAnnotationView(annotation: annotation,
reuseIdentifier: NSStringFromClass(AppleMarkerAnnotationView.self))
            default:
                print("NIL")
                return nil
        }
    }
}
```

Listing 20: Custom annotation view

The viewForAnnotation delegate function returns a custom view called "**AppleMarkerAnnotationView**", which displays Apple logo instead of the default marker. **Figure 5** shows the result.

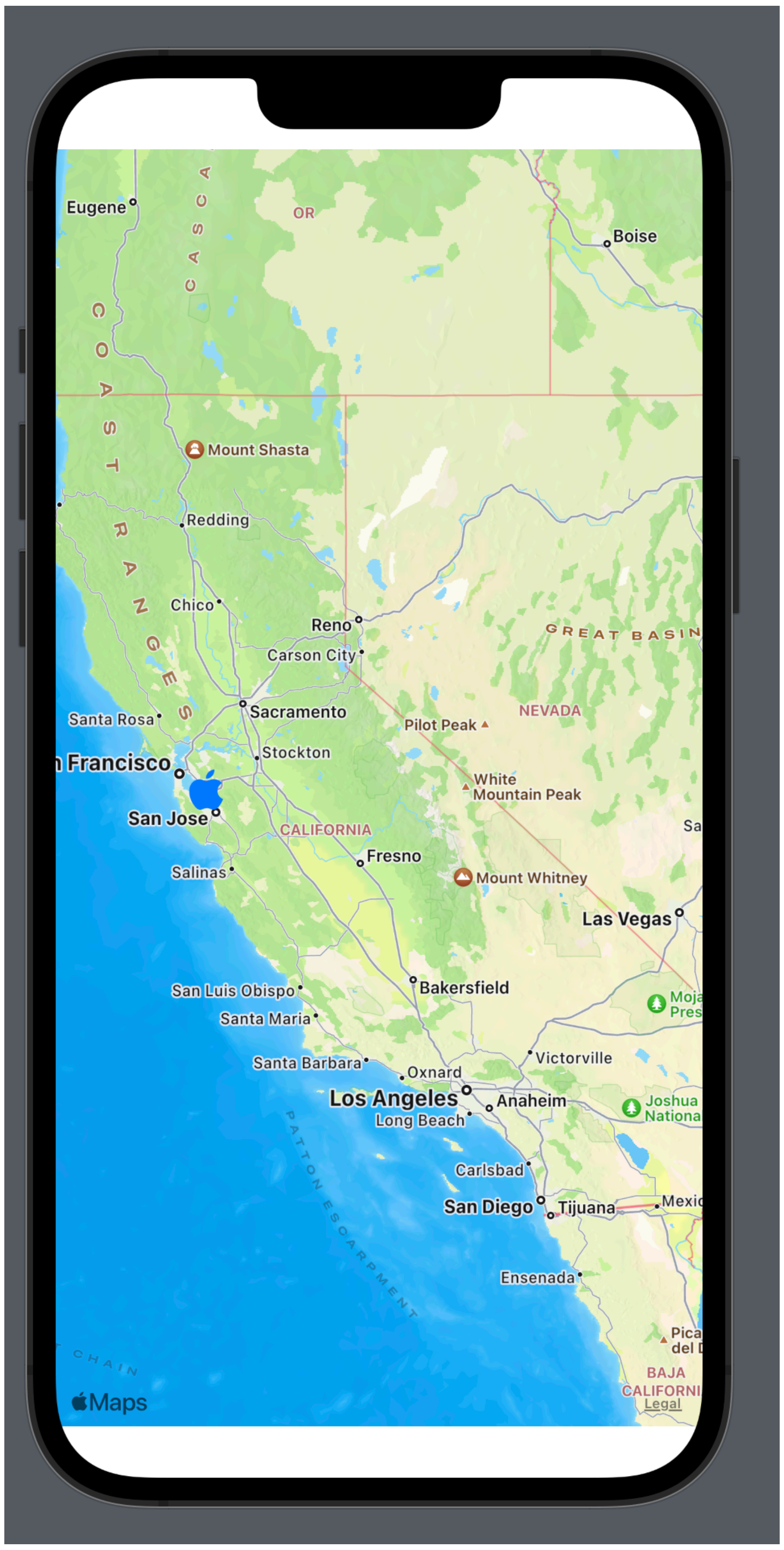


Figure 5: Custom annotation view

The MKMapViewDelegate also provides many other delegate functions, which can be used using the techniques discussed earlier.

How to load a SwiftUI view as a cell for UIKit UITableView?

In iOS 16 and Xcode 14, Apple introduced new hosting configurations, which allows you to load a SwiftUI view as a cell for your UITableView or UICollectionView. This is great news because now we can use the declarative API of SwiftUI to construct our UITableView layout.

The first step is to implement your cell using SwiftUI. This is implemented in **Listing 21**.

```
struct DonutCellView: View {

    let donut: Donut

    var body: some View {
        HStack {
            Image(donut.picture)
                .resizable()
                .frame(width: 75, height: 75)
                .clipShape(RoundedRectangle(cornerRadius: 10))

            Text(donut.name)
            Spacer()
            Image(systemName: "chevron.right")

        }.padding()
    }
}
```

Listing 21: Implementation of DonutCellView

Our cell is called DonutCellView and it displays the name and image associated with the donut. **Figure 6** shows the rendering of the DonutCellView.

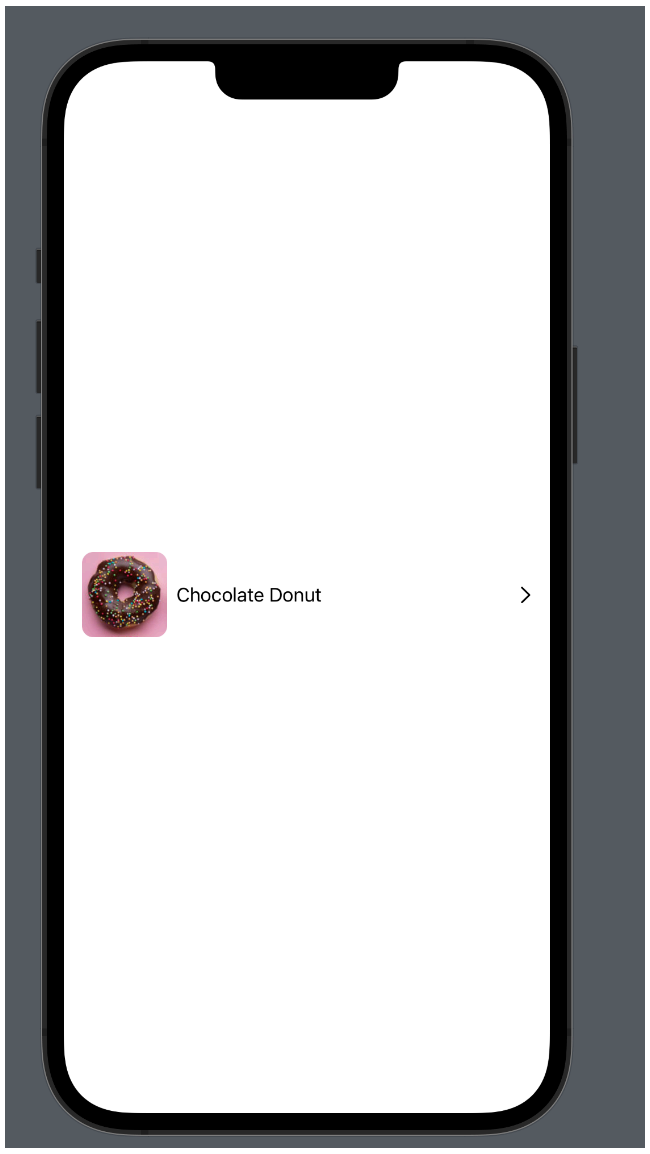


Figure 6: DonutCellView implemented in SwiftUI

Next, we will use the new contentConfiguration available on the UITableViewCell to load the DonutCellView in our UIKit app. The implementation is shown in **Listing 22**.

```
class DonutTableViewController: UITableViewController {

    let donuts = [Donut(name: "Donut 1", picture: "1"), Donut(name: "Donut 2",
picture: "2"),Donut(name: "Donut 3", picture: "3"),Donut(name: "Donut 4",
picture: "4")]

    override func viewDidLoad() {
        super.viewDidLoad()

        navigationController?.navigationBar.prefersLargeTitles = true
        self.title = "Donuts"

        // register the cell
        tableView.register(UITableViewCell.self, forCellReuseIdentifier:
"DonutCell")
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
        return donuts.count
    }
    override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {

        let donut = donuts[indexPath.row]
        let cell = tableView.dequeueReusableCell(withIdentifier: "DonutCell",
for: indexPath)

        cell.contentConfiguration = UIHostingConfiguration(content: {
            DonutCellView(donut: donut) // SwiftUI View
        })

        return cell
    }

}
```

Listing 22: DonutTableViewController using UIHostingConfiguration

The cellForRowAtIndex path delegate function of UITableViewDelegate is responsible for returning a new UITableViewCell. After dequeuing the cell we configure the cell using contentConfiguration. The contentConfiguration is of type UIHostingConfiguration and it allows to host SwiftUI view hierarchy inside the UITableViewCell and UICollectionViewCell.

Figure 7 shows the result.

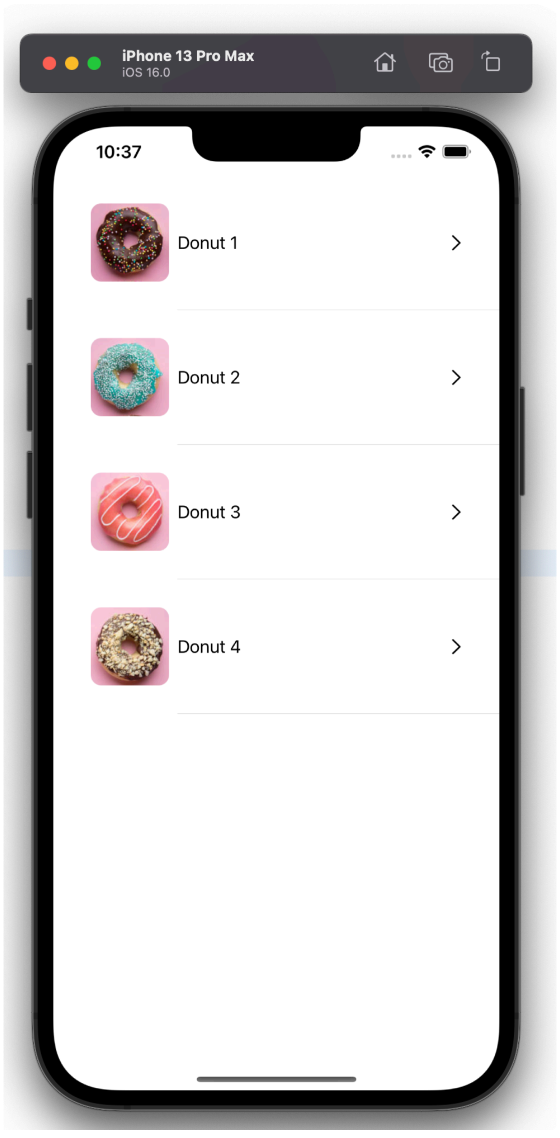


Figure 7: SwiftUI view hosted as the content for the UITableViewCell

How to show Xcode preview of UIViewController?

One of the best things about working in SwiftUI is the ability to visually see your user interface through the use of Xcode previews. I know Xcode previews does have some quirks but at least for me it works majority of the time. Wouldn't it be really cool, if we can also visualize our View Controller in UIKit using Xcode previews? Let's see how we can achieve it.

You can represent your ViewController in SwiftUI by conforming to **UIViewControllerRepresentable**. This way you can load your ViewController in a SwiftUI application. Check out the implementation of DonutTableViewController_Representable in **Listing 23**, which conforms to the UIViewControllerRepresentable protocol.

```swift
struct DonutTableViewController_Representable: UIViewControllerRepresentable {

    func makeUIViewController(context: Context) -> DonutTableViewController {
        DonutTableViewController()
    }

    func updateUIViewController(_ uiViewController: DonutTableViewController,
context: Context) {

    }
}
```

Listing 23: Conforming to UIViewControllerRepresentable

Finally, create a struct which conforms to PreviewProvider protocol. This is shown in **Listing 24**.

```swift
struct DonutDetailsViewController_Previews: PreviewProvider {
    static var previews: some View {
        DonutTableViewController_Representable()
    }
}
```

Listing 24: Implementing the Previews

Now, you will be able to visualize your DonutTableViewController using Xcode previews as shown in **Figure 8**.

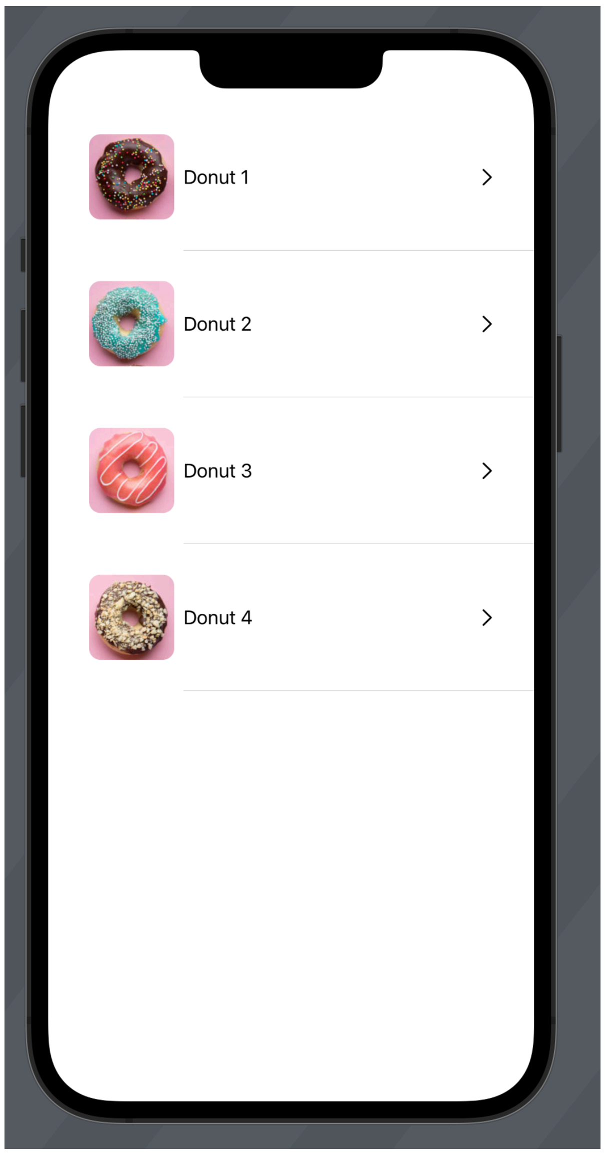


Figure 8: Xcode previews for view controller

Conclusion

I hope you have enjoyed this small recipe book on UIKit with SwiftUI. If you are interested in my other books then check out the section below.

Books:

1. Surviving the Coding Bootcamp - From no coding experience to earning a six-figure salary
2. Navigation API in SwiftUI for iOS 16
3. Check out my Udemy video courses